Algorithms in the Real World: Lecture Notes (Fall 1997)

Guy Blelloch

April 23, 1998

Abstract

This document contains the lecture notes taken by the students in the course "Algorithms in the Real World" taught at UC Berkeley during the Fall semester, 1997. The class covered the following set of ten topics: Compression, Cryptography, Linear Programming, Integer Programming, Triangulation, N-body simulation, VLSI Physical Design, Pattern Matching in Biology, Indexing, and Clustering. Between 2 and 3 lectures were dedicated to each topic. For all topics we looked both at algorithms and at case studies in which the problems are used in real-world applications. The class was a graduate class and assumed that the students came in with a working knowledge of the material covered in a standard algorithms textbook, such as Cormen, Leiserson and Rivest's "Introduction to Algorithms". A goal of the class was to have the students gain an appreciation of how interesting algorithms and data-structures are used in many real-world applications.

The notes contained in this document are based on what was covered in the lectures and are not meant to be complete, and although the scribe takers corrected many of the mistakes in my lectures, I expect many others got through. I must also apologize that the notes contain very few references to the original work on which they rely. Instead I have included at the end of the document a list of primary texts on the topics from which one can find further information. Also, there is a web page associated with the class that has many more online pointers

`http://www.cs.berkeley.edu/~guyb/algs.html`.

I would like to thank all the students who took the notes that appear in this document. They did an excellent job and in many cases went far beyond the call of duty in polishing them and generating figures.

Contents

List Of Algorithms

The following algorithms are covered to some extent in these notes, ranging from a short overview to a full lecture.

1 Introduction to Compression

- Information Theory
- Lossless Compression
  1. Huffman Coding
  2. Shannon-Fano Coding
  3. Arithmetic Coding
  4. LZ78 and LZW (GIF, UNIX Compress...)
  5. LZ77 (gzip, ZIP, ...)
  6. PPM (prediction by partial matching)
- Lossy Compression
  1. Transform Coding (Blocks: JPEG, MPEG, Wavelets...)
  2. Vector Quantization

1.1 Information Theory

How much can we compress any given data? It Depends on how much "information" the data contains.

Entropy

Shannon borrowed the the notion of *entropy* from statistical physics to capture the information contained in an information source. For a set of possible messages S, it is defined as,

$$H(S) = \sum_i p_i \log_2 \frac{1}{p_i}$$

where p_i is the probability of i^{th} message that can be sent by the source. The entropy is a measure of the average number of bits needed to send a message, therefore larger entropies represent more information. Perhaps counter intuitively, the more random a set of messages (the more even the probabilities) the more information they contain. The terms $\log_2 \frac{1}{p_i}$ represent the number of bits used to code message i (the information contained in S_i). Therefore messages with higher probability will use fewer bits.

	bits/char
bits $\lceil \log(96) \rceil$	7
entropy	4.5
Huffman Code (avg.)	4.7
Entropy (Groups of 8)	2.4
Asymptotically approaches:	1.3
Compress	3.7
Gzip	2.7
BOA	2.0

Table 1: Information Content of the English Language

Examples:

$$\begin{aligned} p_i &= \{0.25, 0.25, 0.25, 0.125, 0.125\} \\ H &= 3 \times 0.25 \times \log_2 4 + 2 \times 0.125 \times \log_2 8 \\ &= 1.5 + 0.75 \\ &= 2.25 \end{aligned}$$

$$\begin{aligned} p_i &= \{0.5, 0.125, 0.125, 0.125, 0.125\} \\ H &= 0.5 \times \log_2 2 + 4 \times 0.125 \times \log_2 8 \\ &= 0.5 + 1.5 \\ &= 2 \end{aligned}$$

$$\begin{aligned} p_i &= \{0.75, 0.0625, 0.0625, 0.0625, 0.0625\} \\ H & \quad = 0.75 \times \log_2(\frac{4}{3}) + 4 \times 0.0625 \times \log_2 16 \\ &= 0.3 + 1 \\ &= 1.3 \end{aligned}$$

Note that the more uneven the distribution, the lower the Entropy.

1.2 The English Language

We might be interested in how much information the English Language contains. This could be used as a bound on how much we can compress English, and could also allows us to compare the density (information content) of different languages.

One way to measure the information content is in terms of the average number of bits per character. Table 1 shows a few ways to measure the information of English in terms of bits-per-character. If we assume equal probabilities for all characters, a separate code for each character, and that there are 96 printable characters (the number on a standard keyboard) then each character would take $\lceil \log 96 \rceil = 7$ bits. The entropy assuming even probabilities is

Date	bpc	scheme	authors
May 1977	3.94	LZ77	Ziv, Lempel
1984	3.32	LZMW	Miller and Wegman
1987	3.30	LZH	Brent
1987	3.24	MTF	Moffat
1987	3.18	LZB	Bell
.	2.71	GZIP	.
1988	2.48	PPMC	Moffat
.	2.47	SAKDC	Williams
.	2.47	PPMD	Howard
Nov 1993	2.34	PPMC	Moffat
Oct 1994	2.34	PPM*	Cleary, Teahan, Witten
18 Nov 1994	2.33	PPMD	Moffat
1995	2.29	BW	Burrows, Wheeler
31 Jan 1995	2.27	PPMD	Teahan
1997	1.99	BOA	

Table 2: Lossless compression ratios for text compression on Calgary Corpus

$\log 96 = 6.6$ bits/char. If we give the characters a probability distribution (based on a corpus of English text) the entropy is reduced to about 4.5 bits/char. If we assume a separate code for each character (for which the Huffman code is optimal) the number is slightly larger 4.7 bits/char.

Note that so far we have not taken any advantage of relationships among adjacent or nearby characters. If you break text into blocks of 8 characters, measure the entropy of those blocks (based on measuring their frequency in an English corpus) you get an entropy of about 19 bits. When we divide this by the fact we are coding 8 characters at a time, the entropy (bits) per character is 2.4. If we group larger and larger blocks people have estimated that the entropy would approach 1.3 (or lower). It is impossible to actually measure this because there are too many possible strings to run statistics on, and no corpus large enough.

This value 1.3 bits/char is an estimate of the information content of the English language. Assuming it is approximately correct, this bounds how much we can expect to compress English text if we want lossless compression. Table 1 also shows the compression rate of various compressors. All these, however, are general purpose and not designed specifically for the English language. The last one, BOA, is the current state-of-the-art for general-purpose compressors. To reach the 1.3 bits/char the compressor would surely have to "know" about English grammar, standard idioms, etc..

A more complete set of compression ratios for the Calgary corpus for a variety of compressors is shown in Table 2. The Calgary corpus is a standard benchmark for measuring compression ratios and mostly consists of English text. In particular it consists of 2 books, 5 papers, 1 bibliography, 1 collection of news articles, 3 programs, 1 terminal session, 2 object files, 1 geophysical data, and 1 bit-map b/w image. The table shows how the state of the art has improved over the years.

2 Basic Compression Algorithms

We will first discuss codes that assign a unique bit-string for coding each symbol (Huffman codes and Shannon Fano codes) and then discuss Arithmetic coding which "blends" the codes from different symbols.

2.1 Prefix Codes

- each symbol (message) is a leaf in a binary tree
- the code is encoded in the path from root to the leaf (left = 0, right = 1)
- *Property*: No code is a prefix of another code

The average length of a prefix code assuming a probability distribution on the symbols is defined as $l_a = \sum_i p_i l_i$ where p_i is the probability of the i^{th} symbol and l_i is the length of its code (the depth of its leaf).

2.2 Huffman Codes

Huffman codes are an optimal set of prefix codes. They are optimal in the sense that no other prefix code will have a shorter average length.

To build a Huffman code tree:

- start with a forest of trees each representing a symbol and with weight $w_i = p_i$
- Repeat until single tree
  - select two trees with lowest weights (w_1 and w_2)
  - combine into single tree with weight $w_1 + w_2$

Properties of Huffman Coding:

1. The length of the code for symbol i is approximately $\log_2(\frac{1}{p_i})$. (In the original lecture I said that $\lceil \log_2(\frac{1}{p_i}) \rceil$ is an upper bound on the length of each code. This is actually not true, as pointed out by a student. Consider the probabilities $.01, .30, .34, .35$. A Huffman coding will give the probability .30 a length 3, which is $> \lceil \log_2(1/.3) \rceil = 2$. It is the last time I trust "Data and Image Compression" by Gilbert Held.)
2. The average length l_a of a Huffman code matches the entropy if $p_i = \frac{1}{2^j}$ where j is any integer, for all i.

2.3 Shannon-Fano Coding

Also a prefix code, but tree is built from the top down, as opposed to being built from bottom-up as in the Huffman Coding. First sort S by probabilities, then partition list of messages into two partitions, such that the subsums of the probabilities in each partition is as close to 50% as possible. Each partition is assigned a branch with a 1 or a 0. Repeat process until tree is fully built and all messages are separated into leaves.

```
make_codes(S)
    if |S| = 1 then S[0]
    else
       sort S by probabilities
       split S into S1 and S2 such that W(S1) approx. W(S2)
       return node(make_code(S1), make_code(S2))
```

where `W(S)` represents the sum of the probabilities in the set `S`. Unlike Huffman codes, Shannon-Fano codes do not always give optimal prefix codes.

2.4 Arithmetic Coding

Given the probability of each symbol, we start with a probability range of 1. Then we separate the range of 1 into discrete partitions according to the probabilities of each symbol. So if $P(a) = 0.6$ and $P(b) = 0.4$, then 60% of the range is one partition for the symbol a. As each symbol is read from the input, the range is further refined as the lower and upper bounds of the range are modified by the probability of the new symbol. If we use L_i and U_i to denote the lower and upper bounds, and l_i and u_i to denote the range of probability for that single symbol (i.e. $l_a = 0$ and $u_a = 0.6$), then we have:

$$L_i = L_{i-1} + (U_{i-1} - L_{i-1}) \times l_i$$

and

$$U_i = L_{i-1} + (U_{i-1} - L_{i-1}) \times u_i$$

Properties:

- might not be able to emit code until seen full message (in cases where the range straddles 0.5, the first bit cannot be output until the range falls to one side of 0.5)
- asymptotically approaches entropy
- can require arbitrary precision arithmetic to implement (although we will see a nice trick in the next class)
- often used together with other methods (i.e. as a back-end)

A solution to the straddling and precision problems is to separate the input into fixed size blocks (e.g. 100 symbols), and output compressed code for each block

2.5 Deriving Probabilities

For Huffman, Shannon-Fano and Arithmetic coding we need probabilities. How do we derive them.

- Static vs. Dynamic
  - Static for a single document: We could measure the probabilities over the document we plan to send and use those. In this case we have to send the probabilities (or generated codes) to the decoder.
  - Static for a language: We could use a pre-measured set of probabilities. In this case the decoder could already know the probabilities.
  - Dynamic: We could update the probabilities as we are sending a document based on what we have seen. In this case the decoder can also calculate the probabilities on the fly as it decodes the message (we don't have to send them).
- Single symbols vs. Groups of Symbols
  - We could measure the probability of each symbol independent of its context.
  - We could measure the probability based on a context (i.e. previous symbols). This is discussed further in the next class and will usually give much better compression.

PPM: Prediction by partial matching

This is one of general-purpose algorithms that gets the best compression ratios (although it is certainly not the fastest). It is based on generating static probabilities over the current document using the symbols and their context. The algorithm can be outlined as follows.

- Try finding all multiple occurrences (matches) of strings of length n in the message (n is a preset constant).
- For leftovers, try to find multiple occurrences of strings of $n-1$.
- repeat until all possible matches are found, and assign a probability to each string according to # of occurrences.
- Send the probabilities along with the arithmetic coding of the message that uses the probabilities.

3 Lempel-Ziv Overview

The Lempel-Ziv algorithms compress by building a table of previously seen strings. They do not use probabilities (a string is either in the table or it is not). There are two main variants based on two papers they wrote, one in 1977 (LZ77) and the other in 1978 (LZ78).

- LZ78 (Dictionary Based)

K	w	wK	output
a		a	
b	a	ab(256)	a
c	b	bc(257)	b
a	c	ca(258)	c
b	a	ab	
c	ab	abc(259)	256
a	c	ca	
-	ca	-	258

Table 3: LZ78 Encoding on "abcabca"

K	w	wK	output
a		a	
a	a	aa(256)	a
a	a	aa	
a	aa	aaa(257)	256
a	a	aa	
a	aa	aaa	
-	aaa		257

Table 4: LZ78 Encoding on "aaaaaa"

  - LZW: Lempel-Ziv-Welch
  - LZC: Lempel-Ziv Compress
  - LZMW: (Unix Compress, GIF)

- LZ77 (Sliding Window)
  - Window instead of dictionary
  - LZSS: LZ-Storer-Szymanski (gzip, ZIP, many others)

Traditionally, LZ77 is better but slower.

3.1 LZ78 and LZW

Basically starts with a string lookup table (dictionary) consisting of the standard characters, and generates a dynamic table of new string patterns of increasing length. The decompression algorithm depends on it being able to generate the same table as the compressor.

Encode:

```
w = NIL
while (K = readchar()) {
  if wK in dictionary
    w = wK
  else {
      output code for w
      add wK to dictionary
      w = K }
}
output code for w
```

Key is noting that the code for for a new string `wK` is not output until the second time it is seen. This makes it possible for the decoder to know what the code represents. If in the

input	w (at start)	output (wn)	add to dict (w wn[0])
a		a	
b	a	b	ab(256)
c	b	c	bc(257)
256	c	ab	ca(258)
258	ab	ca	abc(259)

input	w (at start)	output (wn)	add to dict (w wn[0])
a		a	
256	a	aa	aa(256)
257	aa	aaa	aaa(257)

Table 5: LZ78 Decoding on "a b c 256 258" and on "a 256 257"

`else` clause we output the new code for `wK` instead of `w` the message would be impossible to decode.

Tables 3 and 4 show two examples of the encoder. Note that the second example shows a contingency that must be accounted for, when the input has a long string of the same character. The decoder must be able to recognize this and know that code 256 is "aa". This is handled by the `else` clause below.

Decode:

```
w = lookup(readcode())
output w
while ( c = readcode()) {
  if C in dictionary
      wn = lookup(C)
  else
      wn = stringcat(w, w[0])
  output wn
  K = wn[0]
  add wK to dictionary
  w = wn
}
```

Tables 5 shows two examples of decoding.

Properties:

1. Dynamic: (e.g. works well with tar files which have many formats), adaptable.
2. No need to send the dictionary (In LZW need to send clear signals to clear dictionary/string table)

3. Can use Tries to store the dictionary/lookup tables, making lookups effectively executable in unit time.

What happens when dictionary gets too large?

1. Throw dictionary away when reaching a certain size (GIF)

2. Throw dictionary away when not effective (Unix Compress)

3. Throw Least Recently Used entry away when reaches a certain size (BTLZ - British Telecom Standard)

3.2 LZ77 (Sliding Windows)

This scheme involves a finite lookahead window, and the X most recently characters encoded as the dictionary. look in the lookahead window to find the longest string that starts at the beginning of the buffer and has already occurred in the dictionary. Then output: 1. position of prev. occurrence in window, 2. length of string, 3. next char. Then advance windows by length + 1.

Example:

```
1 2 3 4 5 6 7 8 9 10 11  12 13 14 15
a a c a a c a b c a  b || a  b  a  c <--- Data Source
<-----Dictionary-----> || <--L.B.-->
 (previously encoded)    (lookahead buffer)
 Lookahead buffer encoded as <10, 3, c>
```

LZSS Variant:
Don't output next char, instead there are two output formats:
(0, position, length)
(1, char)
Typically only output first format if length ≥ 3. Use Huffman coding to code the outputs.
Optimization: Use Trie to quickly search for match in window (gzip).
Example: Window size = 32

```
Input : aaaaaa
Output: (1, a)  (0, 31, 5)
Input : abcabca
Output: (1, a) (1, b) (1, c) (0, 29, 3) (1, a)
```

- **Review and expansion on previous class**
  - Can we compress all strings?
  - Huffman codes (grouping and choosing between codes with equal average length).
  - Integer implementation of arithmetic coding
  - Conditional entropy and Markov models
  - The JBIG standard
  - The use of Tries is LZW and space issues
  - Run length coding (used for Fax coding) and residual compression (used in the lossless JPEG standard)
- **Lossy compression**
  - Intro
  - Scalar quantization
  - Vector quantization
  - Intro to transform coding

1 Lossless Compression: review and expansion

1.1 Can we compress all strings?

Can a compression algorithm compress some messages without expanding others?

Not if all strings are valid inputs

For a character set of size m and messages of length n, if one string is compressed to $l < n \cdot log_2(m)$, then at least one string must expand to $l > n \cdot log_2(m)$.

GZIP will only expand slightly if no compression is possible. Alternatively, GZIP might check to see if no compression occurred.

1.2 Huffman Codes Revisited

It can be shown that $H(s) \leq R < H(s)+1$ where $H(s)$ is entropy and R is the average code length. Proof of this inequality will be a homework problem. The difference $H(s)-R$ is the overhead caused by Huffman coding.

A standard trick to reduce the overhead per-symbol is to group symbols. For example, if there are 6 symbols, we can generate the probabilities for all 36 pairs by multiplying the probabilities of the symbols (there will be at most 21 unique probabilities). Now generate a

Huffman code for these probabilities and use it to code two symbols at a time. Note that we are not taking advantage of patterns among characters as done with the LZ algorithms or Conditional Entropy discussed later (we are assuming the probability of each symbol is independent of others). Since the overhead is now distributed across pairs of symbols the average per-symbol overhead is bounded by 1/2. If we grouped 3 symbols it would be 1/3. This works fine for a few symbols, but the table of probabilities and the number of entries in the Huffman tree will quickly grow large when increasing the number of symbols in the group.

All Huffman codes for a given set of probabilities will have the same length, but there are different possible ways of encoding.

symbol	probability	code 1	code 2
a	0.2	01	10
b	0.4	1	00
c	0.2	000	11
d	0.1	0010	010
e	0.1	0011	011

Both codings produce an average of 2.2 bits per symbol

With these two codings, one has a high variance in length (code 1), while the other has a low variance (code 2). With low variance coding, a constant character transmission rate is more easily achieved. Also, buffer requirements in a low variance coding scheme might be less. Lowest variance coding can be guaranteed by the following construction.

- After joining two nodes with the lowest probabilities (highest priority), give this joint probability a lower priority than other existing nodes with the same probability.
- In the example above, after 'd' and 'e' are joined, the pair will have the same probability as 'c' and 'a' (.2), but is given a lower probability so that 'c' and 'a' are joined next.
- Now, 'ac' has the lowest priority in the group of 0.4 probabilities
- 'b' and 'de' are joined, and finally, 'ac' is joined with 'bde' revealing the tree shown in Figure 1

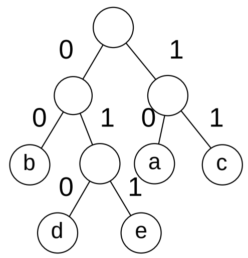


Figure 1: Binary tree for Huffman code 2

1.3 Arithmetic Coding Revisited

It can be shown that Arithmetic coding is bounded as follows: $m \cdot H(s) \leq l \leq m \cdot H(s) + 2$ where m is the length of the input, l is the length of the output, and H(s) is the per-symbol entropy.

An efficient and easy way of implementing arithmetic coding is by using the following integer implementation. The implementation won't give exact arithmetic coding because of roundoff error, but will be close. If both sender and receiver use the same arithmetic they will round in the same way making it possible for the receiver to decode the message properly.

The Integer Implementation is as follows:

- Assume probabilities are counts $c_1, c_2, \ldots, c_m$ (Fractions will also work)
- Cumulative Count: $C_i = \sum_{j=1}^{i} c_j$
- Total Count: $T = \sum_{j=1}^{m} c_j$
- As with any arithmetic coding we start with a fixed-sized region and narrow it each time we code a new character. In our integer implementation the region is initialized to $[0..(R-1)]$ where $R = 2^k$ (i.e. is a power of 2).
- To avoid narrowing to a region of size less than 1, we scale (expand) the region by a factor of 2 under the following conditions:
  - If the region is narrowed down such that it is either in the top or bottom half of the space, output a 1 or 0, respectively, and scale using the appropriate scaling rule.
  - If the region is narrowed down to the second and third quarters, then the region will be expanded about the middle using scale rule #3, described below.
- The Algorithm
  - Let u and l be integers $< K = 2^k$
  - Initial values: $u_0 = K - 1; l_0 = 0$
  - $u_j = l_{j-1} + \left\lfloor (u_{j-1} - l_{j-1} + 1)\frac{C_{x_j}}{T} \right\rfloor - 1$
  - $l_j = l_{j-1} + \left\lfloor (u_{j-1} - l_{j-1} + 1)\frac{C_{x_j - 1}}{T} \right\rfloor$
- Scaling Rules
  - **#1:** If $u_j = 1\ldots$ and $l_j = 1\ldots$ then (the region of interest is in the top half)
    output 1
    shift u to the left and insert a 1
    shift l to the left and insert a 0
    output s zeros, where s is the number of times expanded about the middle region
    set s = 0

- **#2:** If $u_j = 0\ldots$ and $l_j = 0\ldots$ then (the region of interest is in the bottom half)
  output 0
  shift u to the left and insert a 1
  shift l to the left and insert a 0
  output s ones, where s is the number of times expanded about the middle region
  set s = 0
- **#3:** If $u_j = 10\ldots$ and $l_j = 01\ldots$ then (the region of interest is in the second and third quarters of the range)
  $s = s + 1$
  shift u to the left and insert a 1
  shift l to the left and insert a 0
  complement the MSB (most significant bit) of l and u

Question:

If we expand about the middle, expand about the middle, and expand in the top, what output do we get?

Following the rules, s gets incremented twice by expanding about the middle. When we expand in the top half, we output a 1 and s number of zeros (in this case, s=2). Output: 100

Example:

Let $c_1 = 40, c_2 = 1, c_3 = 9$

Which gives $C_0 = 0, C_1 = 40, C_2 = 41, C_3 = T = 50$

We chose a k such that $4 \cdot T < K = 2^k$

In this case, $T = 50$, so $k = 8, K = 256$.

The message we want to transmit is 1321

i	u_i	binary	l_i	binary	output	scale rule	x_i	C_{x_i-1}	C_{x_i}
0	255	11111111	0	00000000			1	0	40
1	203	11001011	0	00000000			3	41	50
2	203	11001011	167	10100111	1	Rule 1			
2	151	10010111	78	01001110		Rule 3			
2	175	10101111	28	00011100			2	40	41
3	148	10010100	146	10010010	10	Rule 1			
3	41	00101001	36	00100100	0	Rule 2			
3	83	01010011	72	01001000	0	Rule 2			
3	167	10100111	144	10010000	1	Rule 1			
3	79	01001111	32	00100000	0	Rule 2			
3	159	10011111	64	01000000		Rule 3			
3	191	10111111	0	00000000			1	0	40
4	152	10011000	0	00000000					

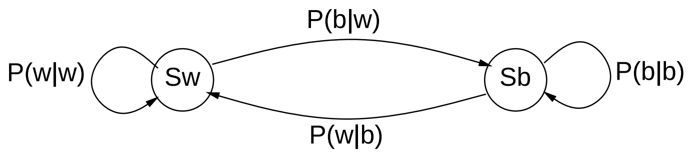


Figure 2: A two state Markov Model

1.4 Conditional Entropy and Markov Chains

Conditional entropy comes into use in both lossless and lossy techniques. Compression can be increased by knowing the context of the message (symbol). For example, in images, the surrounding pixels can be used as the context. In black and white faxes, if there is a black pixel, and the surrounding areas also contain black pixels, it is highly probable that the next pixel is also black. For text, previous characters can be used as the context.

The *conditional entropy* of T based on context S is defined by the equation:

$$\begin{aligned} H(T|S) &= \sum_j P(S_j) \sum_i P(T_i|S_j) log \frac{1}{P(T_i|S_j)} \\ where & \\ P(S_j) &= probability\ of\ context\ S_j \\ P(T_i|S_j) &= probability\ of\ T_i\ in\ context\ S_j \end{aligned}$$

It is not hard to show that if T is independent of the context S then $H(T|S) = H(T)$ otherwise $H(T|S) < H(T)$. In other words, knowing the context can only reduce the entropy.

A way to represent these conditional probabilities is through a Markov Model (or Markov Chain). Figure 2 shows a Markov Model where there are two states, white (Sw) and black (Sb), and conditional probabilities:

$$\begin{aligned} P(w|w) &= probability\ of\ getting\ a\ white\ pixel\ if\ the\ previous\ pixel\ was\ white \\ P(w|b) &= probability\ of\ getting\ a\ white\ pixel\ if\ the\ previous\ pixel\ was\ black \\ P(b|b) &= probability\ of\ getting\ a\ black\ pixel\ if\ the\ previous\ pixel\ was\ black \\ P(b|w) &= probability\ of\ getting\ a\ black\ pixel\ if\ the\ previous\ pixel\ was\ white \end{aligned}$$

In the case of a fax machine, the probabilities of staying in the same state, $P(b|b)$ and $P(w|w)$, are higher than the probability of switching states.

Conditional probabilities can be used with arithmetic coding to code messages with average length that asymptotically matches the conditional entropy. The idea is that for every context we have a separate narrowing function based on the probabilities for that context.

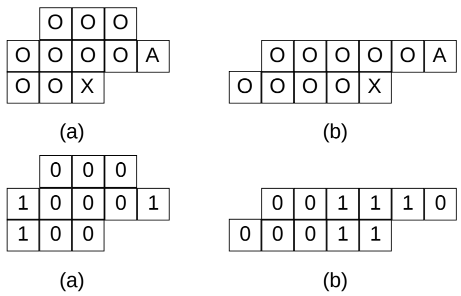


Figure 3: a) Three-line neighborhood b) Two-line neighborhood

1.5 The JBIG Standard

JBIG stands for the Joint Bilevel Image Processing Group. Probabilities are generated based on context. This standard is used for black and white, where the table of probabilities is only 2^k, where k is the number of pixels used as the context. Neighborhoods are used as the context. Example neighborhoods are shown in Figure 3. In the JBIG standard, there is also what is known as a roaming pixel. One of the context pixels, 'A' in Figure 3, is possibly located n pixels away. The current pixel is denoted by 'X'. This is useful when there are vertical lines being transmitted. If the JBIG standard was used on color images, then the table would grow too large to be practical. After building a probability table, the image is then sent after being arithmetically encoded. There is also a default probability table that was made after analyzing thousands of images.

1.6 Tries and use in LZW

An example trie structure is shown in Figure 4. When using a trie structure for LZW, searches can be done in unit time, since the previous search is the parent of the next search. The problem with using tries is that the amount of memory needed for a trie can get large. Consider a dictionary of size 4096. The pointers in the trie need to be $log_2(4096)$ bits = 1.5 bytes long. If we have 256 symbols the amount of memory we will need is 4096 entries * 256 child pointers * 1.5 bytes/pointer $\approx$ 1.5 Megabytes!

Another way of storing the dictionary is by using hash tables. The choice of whether to use a hash table or a trie structure depends mostly on the size of the dictionary and the available memory.

1.7 Other Lossless Compression Techniques

In older fax machines, run length compression is used to transmit and receive messages. The algorithm creates a list of numbers that correspond to the number of consecutive black dots then the number of consecutive white dots and repeats until the end of the message. The list is then Huffman coded and transmitted to the receiving fax, which decodes by applying the algorithm in reverse. This method works OK for black and white images, but JBIG is two to three times better.

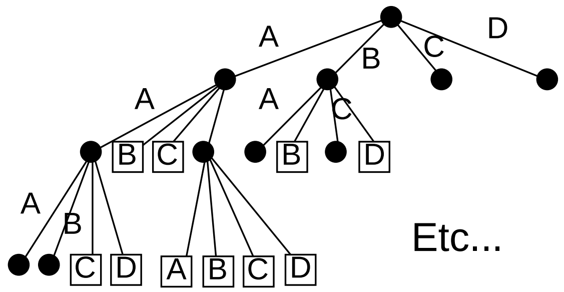


Figure 4: A trie structure

Residual compression is another lossless compression technique. It is used in lossless JPEG, which handles both grey-scale and color images. Residual compression guesses the next pixel value based on context and outputs the difference between the guess and actual value. The differences are Huffman coded and transmitted. Guesses of the next pixel are usually a weighted sum of pixels in the neighborhood. For the most part, the guesses are good. Instead of Huffman coding the original pixel values we now Huffman code (or arithmetic code) the residuals. The residuals will tend to have a lower entropy than the original data (i.e. 0s and low values have high probabilities, while high values have low probabilities) giving us a more efficient coding. Note that the residual values can be negative.

2 Lossy Compression

There are many ways to achieve lossy compression. Some of these include

- scalar quantization,
- vector quantization,
- transform coding (wavelets, block cosine in JPEG and MPEG),
- fractal coding, and model based coding.

Judging the effectiveness of lossy compression is difficult. Unlike lossless compression where measures are easily quantifiable (compression ratio, time, etc.), lossy compression also has some qualitative measures. In fact many people argue that the quantitative measures for lossy compression such as root-mean-square error are not good measures of human perception.

2.1 Scalar Quantization

This is the simplest technique. The idea is to group a large number of regions to a smaller number of regions. An example of this is using only the most significant 6 bits of an 8 bit message. This is an example of a uniform scalar quantization. The function is shown in Figure 5a.

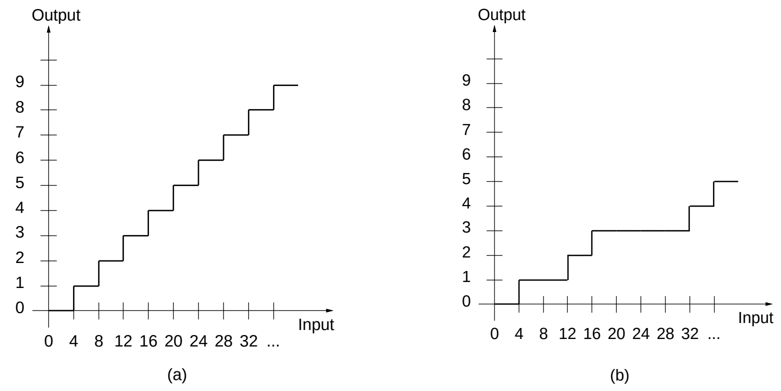


Figure 5: a) Uniform Scalar Quantization b) Nonuniform Scalar Quantization

Non-uniform scalar quantizations might have a function similar to Figure 5b. Non-uniform quantizations are useful when some regions have more useful information than others. For example, differences in low values of red are more sensitive to the eyes than differences in high values of red. In this case, keeping as much information when the values are low is important, while grouping some of the high values will lead to a very good approximation. For images, this allows for better quality for the same compression.

2.2 Vector Quantization

The idea of vector quantization is given a set S of possible vectors to generate a representative subset $R \subset S$ which can be used to approximate the elements of S. The set R are called the code vectors. The incoming data gets parsed into vectors and the encoder first finds the closest (by Euclidean norms) code vector. The index of the code vector can now be sent (possibly after the normal Huffman coding). The quality of this compression will depend on the size of the codebook. Figure 6 shows a block diagram of vector quantization.

What should be used as the vectors? In images, the values of red, green, and blue can be used as a vector. Vector quantization can be used to reduce 24 bit color to 8 bit color. Many terminals use some sort of vector quantization to save on screen memory, and it is supported by the graphics hardware. For speech, a pre-defined number, K, of consecutive samples can be used as the vector. Another way to break up an image into vectors is to represent blocks of pixels as a vector.

Vector quantization used along with a residual method produces a lossless compression. All that needs to be transmitted is the index of the code vector and the difference between the code vector and the actual vector.

2.3 Transform Coding

To use transform coding, a linear set of basis functions (ϕ_i) that span the space must first be selected. Some common sets include sin, cos, spherical harmonics, Bessel functions, wavelets, and rotation. Figure 7 shows some examples of the first three basis functions for

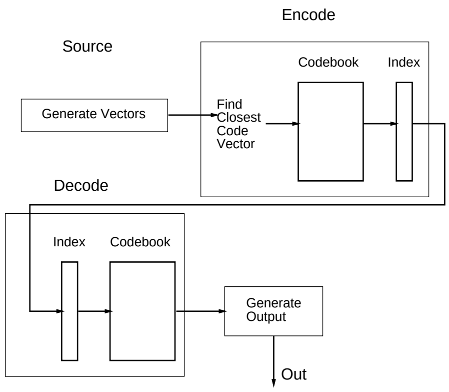


Figure 6: Vector Quantization

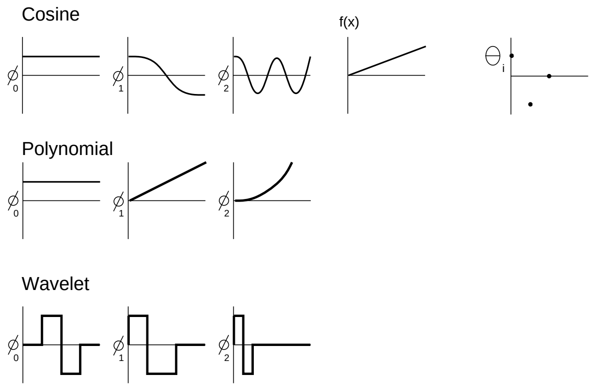


Figure 7: Transforms

cosine, polynomial, and wavelet transformations. The wavelet transform is especially good at capturing local features.

The general equation for generating the i^{th} transform coefficient Θ_i given a function $F(x)$ to transform is

$$\begin{aligned} \Theta_i &= \sum_{j=1}^{n} \phi_i(x_j) F(x_j) \\ where\ \Theta_i &= i^{th}\ transformed\ coefficient \end{aligned}$$

2.4 Choosing a Transform

The goals we have when choosing a transform are:

- Decorrelate the data
- Low coefficients for many terms
- Some basis functions can be ignored by perception

Low coefficients for many terms will allow us to either drop them and only send the larger coefficients. The higher frequency cosines will not be able to be perceived, so it is allowable to drop them. This is low pass filtering a signal (taking only the lower frequency components of a FFT).

Question:

Why not use a Fourier (or cosine) transform across the whole image?

Imagine a totally white background with a black pixel in the middle. This is represented as an impulse function, and the Fourier transform of an impulse function is 1 for all frequencies. To adequately represent the image, we will not be able to drop any coefficients since they are all of the same magnitude. JPEG blocks the image off before applying transforms.

Notes by Ben Liblit

- Case Studies
  1. JPEG Encoding
  2. MPEG Encoding
- Other Lossy Transform Codes
  1. Wavelet Compression
  2. Fractal Compression
  3. Model-Based Compression

1 Case Studies

In previous lectures we examined several algorithms for lossless and lossy compression. Now we draw together these isolated parts into complete compression schemes suitable for use on real-world data.

We will examine, as case studies, two popular lossy compression schemes: JPEG for still images, and MPEG for video.

1.1 JPEG

1.1.1 JPEG Introduction

JPEG is a lossy compression scheme for color and grayscale images. It has several important properties:

- designed for natural, real-world scenes

  JPEG sustains excellent compression rations on photographic material and naturalistic artwork. It performs poorly on cartoons, line drawings, and other images with large areas of solid color delimited by sharp boundaries.

- tunable lossy compression

  JPEG discards data at several phases of the compression. In general, data is discarded where it will have the smallest subjective impact on a human viewer. JPEG contains tunable quality "knobs" that allow one to trade off between compression effectiveness and image fidelity.

- 24-bit color

  JPEG maintains 24 bits of color data, in contrast to GIF's limit of 8 bits. In practice, this can mean that GIF discards more data than JPEG, since GIF must first quantize an image down to no more than 256 distinct colors.

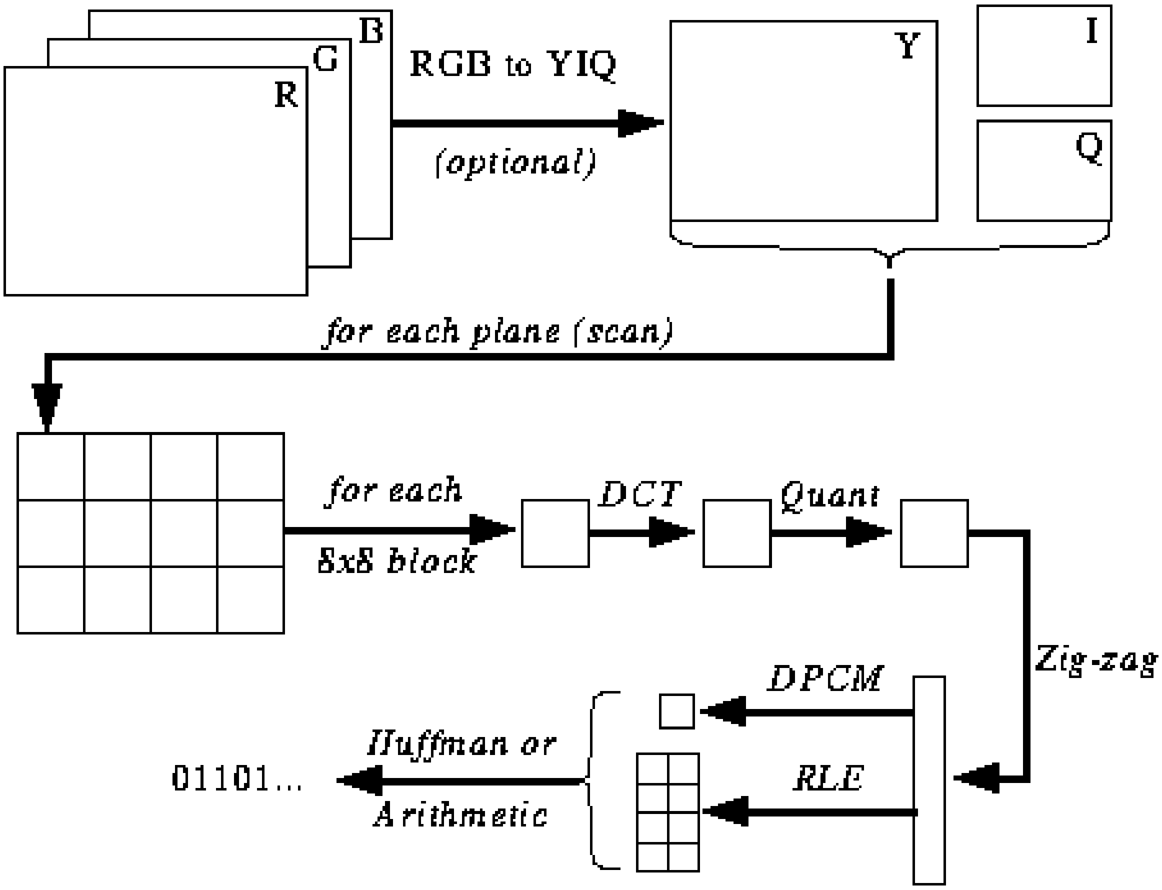


Figure 8: Steps in JPEG compression.

JPEG forms an excellent case study in compression because it synthesizes a number of distinct algorithms:

- Transform Coding
- Scalar Quantization
- Difference Coding
- Run Length Coding
- Huffman or Arithmetic Coding

We will highlight each of these techniques as they appear in the algorithm, below.

1.1.2 JPEG Compression Steps

1. (optional) Convert image from RGB to YIQ.

   RGB is a three-dimensional vector space that describes colors in a manner convenient for computer hardware. YIQ is a different color space that more closely matches subjective human color perception. The "I" represents the relative balance of orange versus cyan. The "Q" component represents the relative balance of green versus magenta. The "I" and "Q" components collectively represent the hue, or "chrominance" of any pixel. The "Y" component represents the overall brightness, or luminosity, of a color. This brightness is weighted to reflect the sensitivities of human vision: green contributes most to perceived luminance, red contributes about half as strongly, and blue has the least impact.

   When converting the the image to YIQ, we must choose how many bits will be used to represent each component. We generally devote four times as many bits to luminance (Y) as we do to chrominance (IQ). The eye is much more sensitive to small changes in brightness than it is to small changes in hue.

   Incidentally, NTSC television transmissions encode colors using the YIQ color space, with the largest portion of bandwidth devoted to luminance. Also, since brightness is represented using a single luminosity component, black and white televisions can display this value directly and simply ignore the two chrominance components.

2. Subdivide the image into blocks of 8×8 pixels.

   Small, fixed-size blocks will greatly simplify subsequent computation. Small blocks also help to limit cosine transformation "leakage" should the image contain sharp (high-frequency) discontinuities. Without these blocks, a single sharp feature could bleed into the entire rest of the image.

3. For each 8×8 block, apply the discrete cosine transformation to produce a transformed 8×8 block with the same bit depth.

   This is a deployment of transform coding, independently applied to the horizontal and vertical axes of each block.

4. Reduce the overall depth of these transformed blocks by a scaling factor.

   This is a deployment of scalar quantization. The quantization error that this necessarily introduces is the most significant source of information loss in JPEG compression.

   One could simply divide all entries by some uniform value, but this will not actually give the best results. When images are intended for use by human viewers, some bits are more important than others. Instead, variable scaling factors are derived from an 8×8 quantization table. Each element in the transformed block is divided by the corresponding element in the quantization table. This allows us to be selective in where we discard information.

   The JPEG standard defines two such tables. One is used for chrominance (I, Q), and is not presented here. The other is used for luminance (Y), and is shown in Table 6. Notice that the largest coefficients are in the lower right corner, while the smallest

16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

Table 6: JPEG default quantization table, luminance plane.

are toward the upper left. This will discard the most bits from high frequencies, and the fewest bits from low frequencies. Psychometric studies show that the human eye is more sensitive to lower frequencies, so this is where we can least afford to discard information.

Abstractly, the default tables represent the relative importance of features in any image. If we were to uniformly double all of the quantization coefficients, we would still be using the same relative importance, but we would be discarding more bits. Thus, the entire quantization table may be scaled by any single factor to improve compression while reducing image quality.

In most JPEG compressors, this single factor is the "quality knob" made available to the user. A scaling factor of 8 is a rough upper limit for high-fidelity reproduction; factors up to 16 will be lossy but presentable; factors as high as 32 may still be recognizable but are unusable for most practical tasks.

After quantization, the DCT coefficients will lie in a much narrower range. This reduces their overall entropy and will make subsequent compression more effective. In fact, much of the high-frequency region toward the lower-right may have been reduced to zeros.

5. Encode the DC components.

   This is a deployment of difference coding followed by Huffman or arithmetic coding.

   The DC component contains most of the information in the image. Across an entire image, the DC component is large and varied. Within any local region, though, the DC component is often close to the previous value. Thus, instead of encoding the (high-entropy) DC values directly, we encode the (low-entropy) differences between DC values. Specifically, for each DC component in each 8×8 block, we encode the difference from the corresponding DC component in the previous 8×8 block.

   These differences will tend to be small, and thus amenable to entropy coding. Either Huffman or arithmetic coding is used to further compress the DC difference stream.

6. Linearize the AC components.

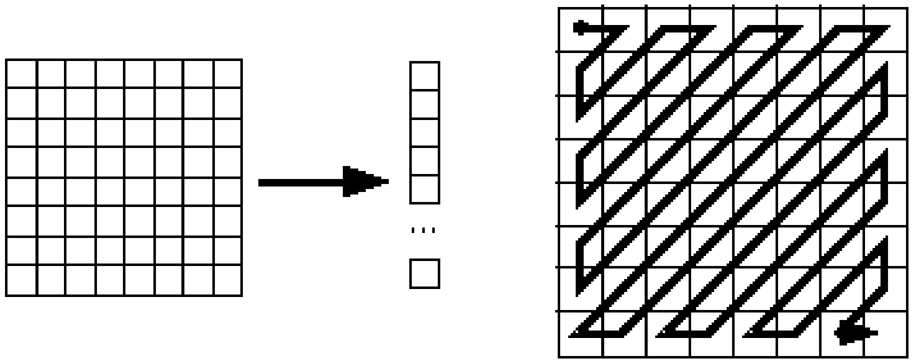


Figure 9: Zig-zag scanning of JPEG blocks.

In order to sequentially encode the AC components of an 8×8 block, we must first represent it as a simple 1×64 vector. The obvious approach would be to iterate across the 64 cells in row-major or column-major order. A much better approach is to zig-zag across the block, as shown in Figure 9.

Why is this desirable? Recall that the upper-left region represents high-frequency features, while the lower-right represents low-frequency features. A zig-zag path groups low- and high-frequency coefficients close together in the 1×64 vector.

This means that successive coefficients will tend to have similar values. Furthermore, many coefficients will be zero, particularly toward the low-frequency lower-right corner. These correlations will make subsequent compression much more effective.

7. Encode the AC components.

   This is a deployment of run length encoding followed by Huffman or arithmetic coding.

   AC components are encoded as (skip, value) pairs, where skip is the number of zeros and value is the next non-zero component. The special pair (0, 0) designates the end of a block.

   The code pairs are then further compressed using Huffman or arithmetic coding.

1.2 MPEG

1.2.1 MPEG Introduction

Correlation improves compression. This is a recurring theme in all of the approaches we have seen; the more effectively a technique is able to exploit correlations in the data, the more effectively it will be able to compress that data.

This principle is most evident in MPEG encoding. MPEG compresses video streams. In theory, a video stream is a sequence of discrete images. In practice, successive images are highly interrelated. Barring cut shots or scene changes, any given video frame is likely to

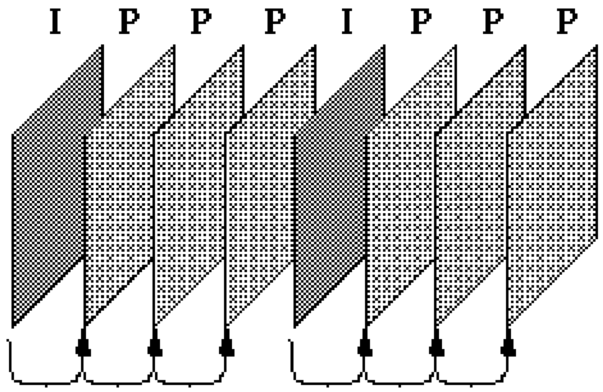


Figure 10: MPEG I-frames interspersed with P-frames.

Playback order:	0	1	2	3	4	5	6	7	8	9
Frame type:	I	B	B	P	B	B	P	B	B	I
Data stream order:	0	2	3	1	5	6	4	8	9	7

Table 7: MPEG B-frames postponed in data stream.

bear a close resemblance to neighboring frames. MPEG will exploit this strong correlation to achieve far better compression rates than would be possible with isolated images.

Each frame in an MPEG image stream is encoded using one of three schemes:

I-frame , or intra-frame. I-frames are encoded as isolated images. They do not take advantage of redundant information in any other frames.

P-frame , or predictive coded frame. A P-frame can take advantage of redundant information in the previous I- or P-frame.

B-frame , or bidirectionally predictive coded frame. A B-frame can take advantage of similar information in both the preceding and the succeeding I- or P-frame.

Figure 10 shows a simplified MPEG stream containing just I- and P-frames. Figure 11 shows a more complex stream that adds B-frames.

I-frames and P-frames appear in an MPEG stream in simple, chronological order. However, B-frames are moved forward so that they appear *after* their neighboring I- and P-frames. This guarantees that each frame appears after any frame upon which it may depend. An MPEG encoder can decode any frame by buffering the two most recent I- or P-frames encountered in the data stream. Table 7 shows how B-frames are postponed in the data stream so as to simplify decoder buffering.

MPEG encoders are free to mix the frame types in any order. When the scene is relatively static, P- and B-frames could be used, while major scene changes could be encoded using

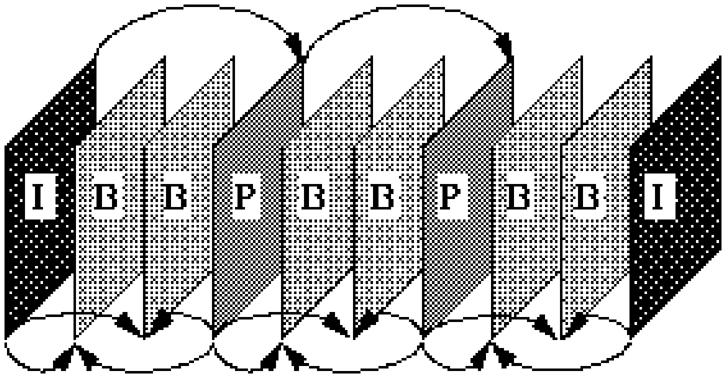


Figure 11: MPEG I- and P-frames interspersed with B-frames.

I-frames. In practice, most encoders use some fixed pattern like the ones shown here: two or three P-frames between each pair of I-frames, and two or three B-frames between those.

1.2.2 MPEG Encoding: I-Frames

Because I-frames are independent, an I-frame can be decoded quickly and simply, without requiring additional information from elsewhere in the MPEG stream. Thus, I-frames form anchor points for fast random access, reverse playback, error recovery, and the like.

I-frames are encoded using a variant of JPEG. First, each frame is converted into the YCrCb color space, a variant of JPEG's YIQ. The frame is then subdivided into 16×16 pixel blocks, called "macroblocks". The luminance data (Y) is represented at full resolution: each 16×16 macroblock is simply split into four 8×8 subblocks. Chrominance data (Cr and Cb) is represented at one quarter resolution: each 16×16 macroblock is reduced to a single 8×8 subblock for each hue axis.

We now have six 8×8 blocks (four Y, one Cr, one Cb). Encoding of these 8×8 blocks proceeds using same JPEG scheme presented above. Each block is cosine-transformed, quantized, zig-zag sequenced, run-length encoded, and Huffman encoded to a final bit stream.

One important difference from MPEG concerns the quantization phase. In its original form, MPEG-I did not support JPEG's variable quantization table. Rather, all coefficients were scaled by a single constant value. MPEG-II restored variable, nonuniform quantization.

1.2.3 MPEG Encoding: P-Frames

P-frames are subdivided into macroblocks using the same technique described above. However, we will not encode the macroblocks directly. Rather, we will attempt to encode the *difference* between each macroblock and a similar macroblock from an earlier frame.

For each macroblock in the current ("target") frame, we search for a matching group of pixels in the preceding ("reference") I-frame or P-frame. In many cases, we may find an

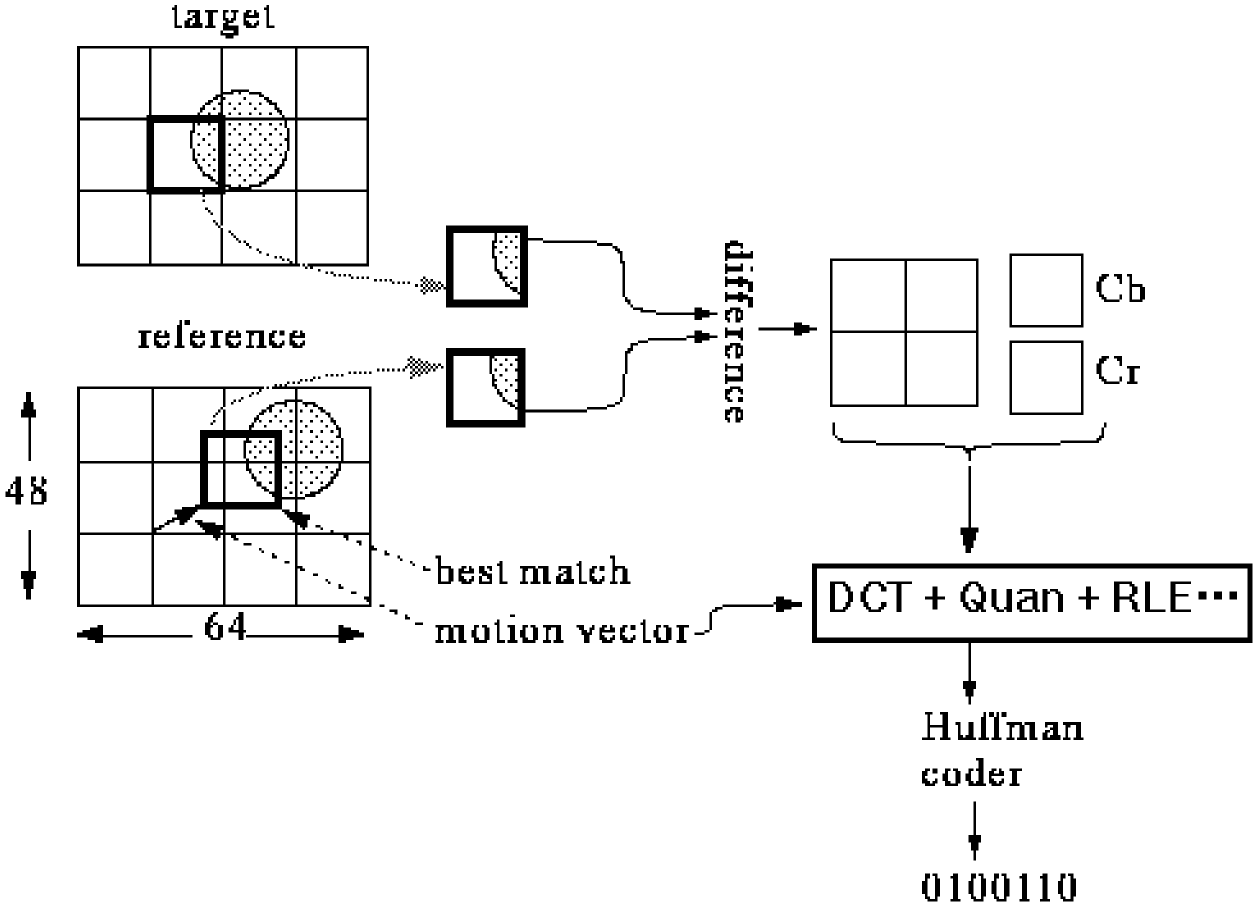


Figure 12: P-frame encoding.

excellent match at the same position in the reference frame. If that region of the frame represents static background scenery, it will likely be unchanged from target to reference.

In general, though, we wish to accommodate motion. Thus, we search for a good target match throughout some larger region of the reference image. Ultimately, we will find our best match at some (x, y) offset from the position of our target macroblock. This offset, or "motion vector", is Huffman encoded for each target macroblock.

The motion vector may direct us to the best match, but it is unlikely to direct us to a set of pixels that are truly identical. Here we use difference coding. We form a macroblock representing the difference between the target macroblock and best-match reference macroblock. This difference macroblock is encoded using the same JPEG technique used for I-frame macroblocks: DCT, quantize, RLE, etc.

In practice, the search for a good match for each target macroblock is the most computationally difficult part of MPEG encoding. With current technology, realtime MPEG encoding is only possible with the help of custom hardware. Note, though, that while the *search* for a match is extremely difficult, *using* the motion vector so derived is as simple as extracting a fixed-size block of pixels given an (x, y) offset into a buffered frame. Encoders are hard; decoders are easy.

One final point: it is entirely possible that for some target macroblock, we will find *no* similar region in the reference frame. Should this happen, we simply give up on difference coding for that block. We omit the motion vector and JPEG-encode the target macroblock directly. In a sense, we have fallen back onto I-frame coding for just that macroblock.

1.2.4 MPEG Encoding: B-Frames

B-frames were not present in MPEG's predecessor, H.261. They were added in an effort to address the following situation: portions of an intermediate P-frame may be completely absent from all previous frames, but may be present in future frames. For example, consider a car entering a shot from the side. Suppose an I-frame encodes the shot before the car has started to appear, and another I-frame appears when the car is completely visible. We would like to use P-frames for the intermediate scenes. However, since no portion of the car is visible in the first I-frame, the P-frames will not be able to "reuse" that information. The fact that the car is visible in a later I-frame does not help us, as P-frames can only look *back* in time, not forward.

B-frames look for reusable data in both directions. The overall technique is identical to that used in P-frames. For each macroblock in the target image, search for a similar collection of pixels in the reference. However, whereas P-frames searched in only the preceding I- or P-frame, B-frames also search in the succeeding I- or P-frame. We find the best match in each frame, and record both. The best match in the previous frame is encoded as a backward-predicted motion vector; the best match in the next frame is encoded as a forward-predicted motion vector.

Recall that a P-frame also encodes the difference between the target and the reference. B-frames have two motion vectors, and therefore two references. We simply merge the two reference blocks and encode the difference between this average and our target.

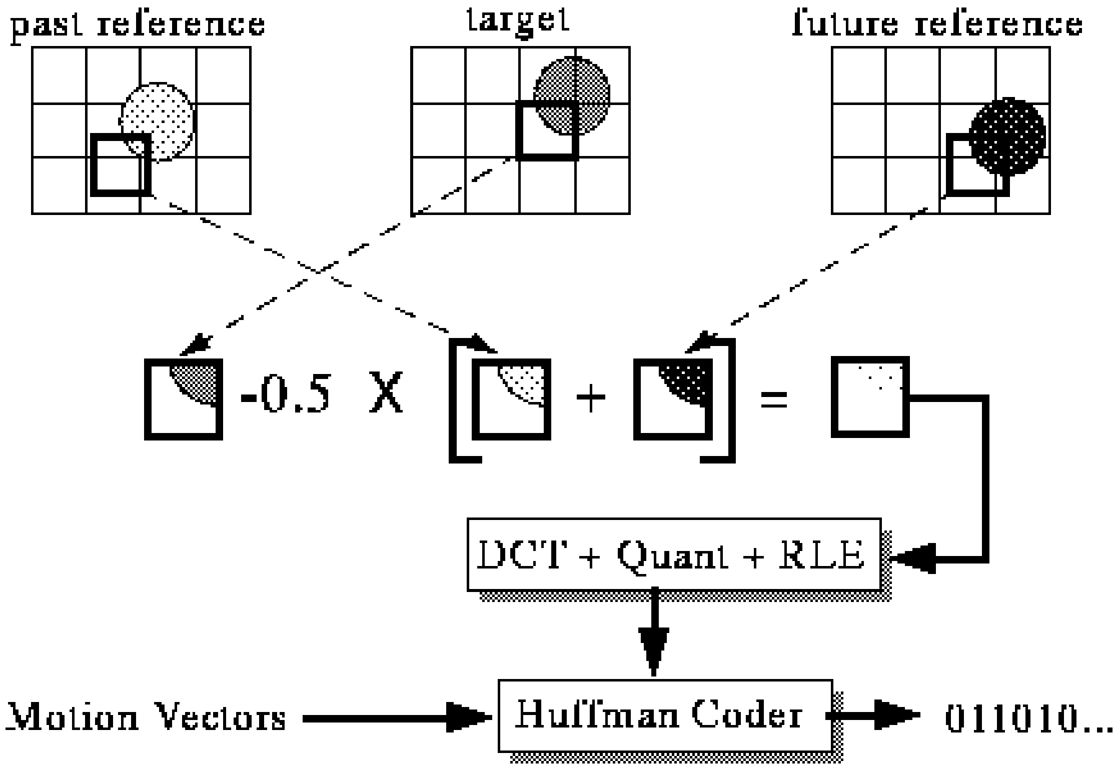


Figure 13: B-frame encoding.

Like P-frames, B-frames may fail to find a good match in either the forward or backward reference. Should this happen, we omit that motion vector and rely exclusively upon the other. In a sense, we will have fallen back on P-frame or backward P-frame coding for just that macroblock. If no good match can be found in either direction, we directly encode the raw target macroblock as though for an I-frame.

1.2.5 MPEG Compression Effectiveness

How effective is MPEG compression? We can examine typical compression ratios for each frame type, and form an average weighted by the ratios in which the frames are typically interleaved.

Starting with a 356×260 pixel, 24-bit color image, typical compression ratios for MPEG-I are:

Type	Size	Ratio
I	18 Kb	7:1
P	6 Kb	20:1
B	2.5 Kb	50:1
Avg	4.8 Kb	27:1

If one 356×260 frame requires 4.8 Kb, how much bandwidth does MPEG require in order to provide a reasonable video feed at thirty frames per second?

$$30 frames/sec \cdot 4.8 Kb/frame \cdot 8b/bit = 1.2 Mbits/sec$$

Thus far, we have been concentrating on the visual component of MPEG. Adding a stereo audio stream will require roughly another 0.25 Mbits/sec, for a grand total bandwidth of 1.45 Mbits/sec.

This fits nicely within the 1.5 Mbit/sec capacity of a T1 line. In fact, this specific limit was a design goal in the formation of MPEG. Real-life MPEG encoders track bit rate as they encode, and will dynamically adjust compression qualities to keep the bit rate within some user-selected bound. This bit-rate control can also be important in other contexts. For example, video on a multimedia CD-ROM must fit within the relatively poor bandwidth of a typical CD-ROM drive.

1.2.6 MPEG in the Real World

MPEG has found a number of applications in the real world, including:

1. Direct Broadcast Satellite. MPEG video streams are received by a dish/decoder, which unpacks the data and synthesizes a standard NTSC television signal.

2. Cable Television. Trial systems are sending MPEG-II programming over cable television lines.

3. Media Vaults. Silicon Graphics, Storage Tech, and other vendors are producing on-demand video systems, with twenty file thousand MPEG-encoded films on a single installation.

4. Real-Time Encoding. This is still the exclusive province of professionals. Incorporating special-purpose parallel hardware, real-time encoders can cost twenty to fifty thousand dollars.

2 Other Lossy Transform Codes

2.1 Wavelet Compression

2.1.1 Wavelet Introduction

JPEG and MPEG decompose images into sets of cosine waveforms. Unfortunately, cosine is a periodic function; this can create problems when an image contains strong aperiodic features. Such local high-frequency spikes would require an infinite number of cosine waves to encode properly. JPEG and MPEG solve this problem by breaking up images into fixed-size blocks and transforming each block in isolation. This effectively clips the infinitely-repeating cosine function, making it possible to encode local features.

An alternative approach would be to choose a set of basis functions that exhibit good locality without artificial clipping. Such basis functions, called "wavelets", could be applied to the entire image, without requiring blocking and without degenerating when presented with high-frequency local features.

How do we derive a suitable set of basis functions? We start with a single function, called a "mother function". Whereas cosine repeats indefinitely, we want the wavelet mother function, ϕ, to be contained within some local region, and approach zero as we stray further away:

$$\lim_{x \to \pm\infty} \phi(x) = 0$$

The family of basis functions are scaled and translated versions of this mother function. For some scaling factor s and translation factor l,

$$\phi_{sl}(x) = \phi(2^s x - l)$$

2.1.2 Haar Wavelets

Haar wavelets are derived from the following mother function:

$$\phi(x) = \begin{cases} 1 & : \quad 0 < x \leq 1/2 \\ -1 & : \quad 1/2 < x \leq 1 \\ 0 & : \quad x \leq 0 \text{ or } x > 1 \end{cases}$$

Figure 14 shows a family of seven Haar basis functions. Of the many potential wavelets, Haar wavelets are probably the most described but the least used. Their regular form makes

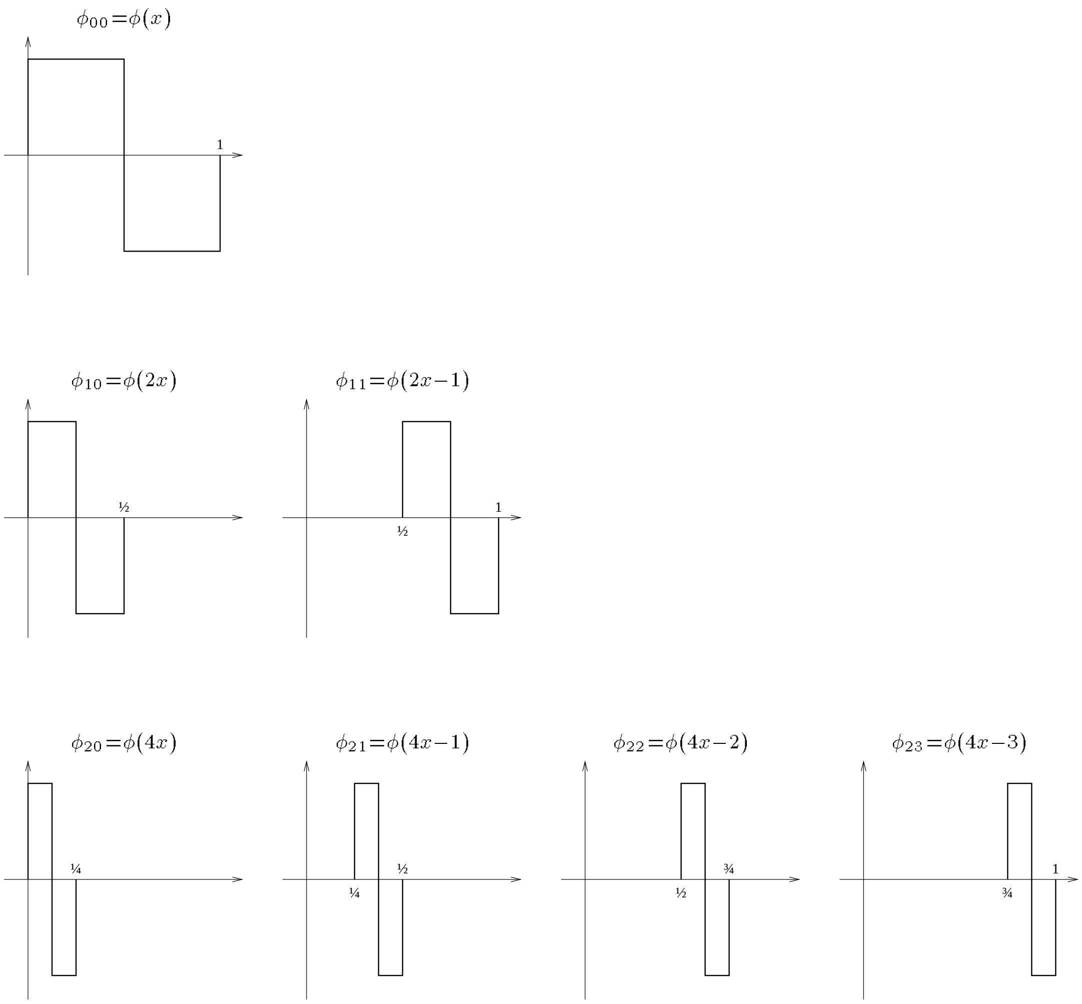


Figure 14: A small Haar wavelet family of size seven.

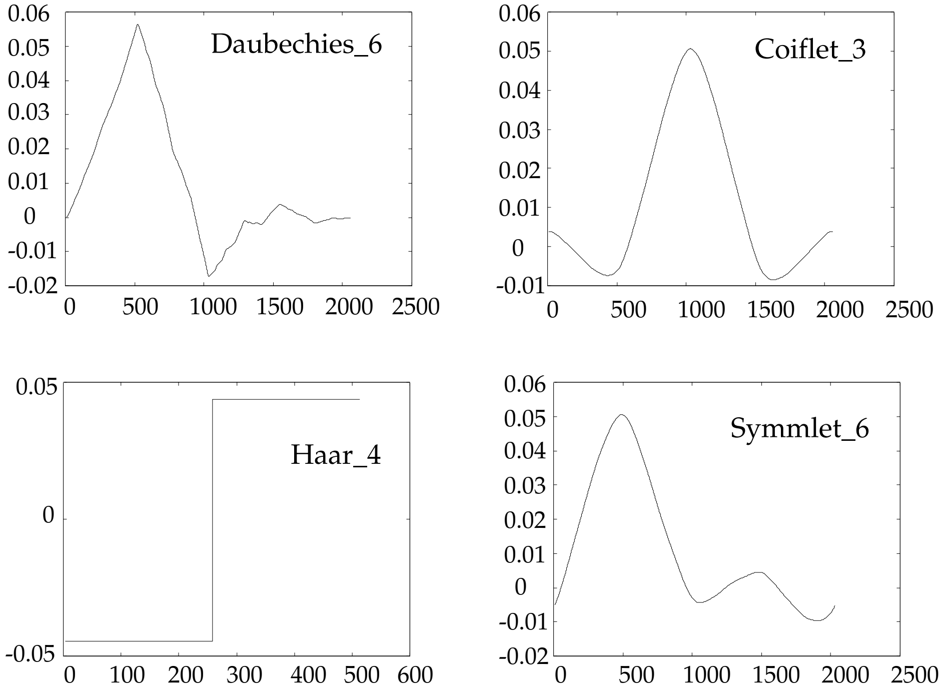


Figure 15: A sampling of popular wavelets.

the underlying mathematics simple and easy to illustrate, but tends to create bad blocking artifacts if actually used for compression.

Many other wavelet mother functions have also been proposed. The Morret wavelet convolves a Gaussian with a cosine, resulting in a periodic but smoothly decaying function. This function is equivalent to a wave packet from quantum physics, and the mathematics of Morret functions have been studied extensively. Figure 15 shows a sampling of other popular wavelets. Figure 16 shows that the Daubechies wavelet is actually a self-similar fractal.

2.1.3 Wavelets in the Real World

Summus Ltd. is the premier vendor of wavelet compression technology. Summus claims to achieve better quality than JPEG for the same compression ratios, but has been loathe to divulge details of how their wavelet compression actually works. Summus wavelet technology has been incorporated into such items as:

- Wavelets-on-a-chip for missile guidance and communications systems.
- Image viewing plugins for Netscape Navigator and Microsoft Internet Explorer.
- Desktop image and movie compression in Corel Draw and Corel Video.
- Digital cameras under development by Fuji.

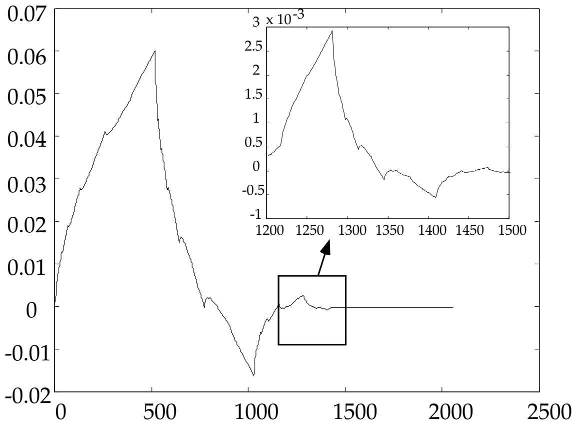


Figure 16: Self-similarity in the Daubechies wavelet.

In a sense, wavelet compression works by characterizing a signal in terms of some underlying generator. Thus, wavelet transformation is also of interest outside of the realm of compression. Wavelet transformation can be used to clean up noisy data or to detect self-similarity over widely varying time scales. It has found uses in medical imaging, computer vision, and analysis of cosmic X-ray sources.

2.2 Fractal Compression

2.2.1 Function Fixed Points

A function $f(x)$ is said to have a fixed point x_f if $x_f = f(x_f)$. For example:

$$\begin{aligned} f(x) &= ax + b \\ \Rightarrow x_f &= \frac{b}{1-a} \end{aligned}$$

This was a simple case. Many functions may be too complex to solve directly. Or a function may be a black box, whose formal definition is not known. In that case, we might try an iterative approach. Keep feeding numbers back through the function in hopes that we will converge on a solution:

$$\begin{aligned} x_0 &= \text{guess} \\ x_i &= f(x_{i-1}) \end{aligned}$$

For example, suppose that we have $f(x)$ as a black box. We might guess zero as x_0 and iterate from there:

$$
\begin{aligned}
x_0 &= 0 \\
x_1 &= f(x_0) = 1 \\
x_2 &= f(x_1) = 1.5 \\
x_3 &= f(x_2) = 1.75 \\
x_4 &= f(x_3) = 1.875 \\
x_5 &= f(x_4) = 1.9375 \\
x_6 &= f(x_5) = 1.96875 \\
x_7 &= f(x_6) = 1.984375 \\
x_8 &= f(x_7) = 1.9921875
\end{aligned}
$$

In this example, $f(x)$ was actually defined as $1/2x + 1$. The exact fixed point is 2, and the iterative solution was converging upon this value.

Iteration is by no means guaranteed to find a fixed point. Not all functions have a single fixed point. Functions may have no fixed point, many fixed points, or an infinite number of fixed points. Even if a function has a fixed point, iteration may not necessarily converge upon it.

2.2.2 Fractal Compression and Fixed Points

In the above example, we were able to associate a fixed point value with a function. If we were given only the function, we would be able to recompute the fixed point value. Put differently, if we wish to transmit a value, we could instead transmit a function that iteratively converges on that value.

This is the idea behind fractal compression. However, we are not interested in transmitting simple numbers, like "2". Rather, we wish to transmit entire images. Our fixed points will be images. Our functions, then, will be mappings from images to images.

Our encoder will operate roughly as follows:

1. Given an image, i, from the set of all possible images, $Image$.
2. Compute a function $f : Image \rightarrow Image$ such that $f(i) \approx i$.
3. Transmit the coefficients that uniquely identify f.

Our decoder will use the coefficients to reassemble f and reconstruct its fixed point, the image:

1. Receive coefficients that uniquely identify some function $f : Image \rightarrow Image$.
2. Iterate f repeatedly until its value converges on a fixed image, i.
3. Present the decompressed image, i.

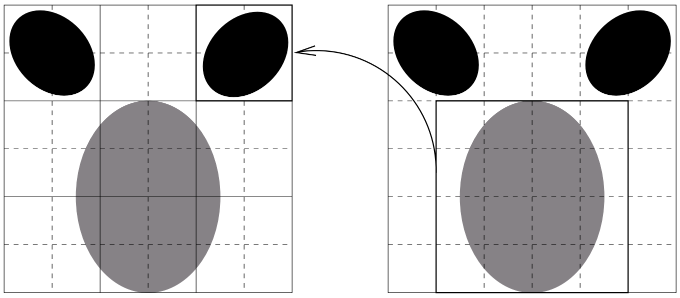


Figure 17: Identifying self-similarity. Range blocks appear on the left; one domain block appears on the left. The arrow identifies one of several collage function that would be composited into a complete image.

Clearly we will not be using entirely arbitrary functions here. We want to choose functions from some family that the encoder and decoder have agreed upon in advance. The members of this family should be identifiable simply by specifying the values for a small number of coefficients. The functions should have fixed points that may be found via iteration, and must not take unduly long to converge.

The function family we choose is a set of "collage functions", which map regions of an image to similar regions elsewhere in the image, modified by scaling, rotation, translation, and other simple transforms. This is vaguely similar to the search for similar macroblocks in MPEG P- and B-frame encoding, but with a much more flexible definition of similarity. Also, whereas MPEG searches for temporal self-similarity across multiple images, fractal compression searches for spatial self-similarity within a single image.

Figure 17 shows a simplified example of decomposing an image info collages of itself. Note that the encoder starts with the subdivided image on the right. For each "range" block, the encoder searchers for a similar "domain" block elsewhere in the image. We generally want domain blocks to be larger than range blocks to ensure good convergence at decoding time.

2.2.3 Fractal Compression in the Real World

Fractal compression using iterated function systems was first described by Dr. Michael Barnsley and Dr. Alan Sloan. Although they claimed extraordinary compression rates, the computational cost of encoding was prohibitive. The major vendor of fractal compression technology is Iterated Systems, cofounded by Barnsley and Sloan.

Today, fractal compression appears to achieve compression ratios competitive with JPEG at reasonable encoding speeds. Details on Iterated's current techniques are, sadly, few and far between.

Fractal compression describes an image in terms of itself, rather than in terms of a pixel grid. This means that fractal images can be somewhat resolution-independent. Indeed, one can easily render a fractal image into a finer or coarser grid than that of the source image. This resolution independence may have use in presenting quality images across a variety of screen and print media.

2.3 Model-Based Compression

We briefly present one last transform coding scheme, model-based compression. The idea here is to characterize the source data in terms of some strong underlying model. The popular example here is faces. We might devise a general model of human faces, describing them in terms of anatomical parameters like nose shape, eye separation, skin color, cheekbone angle, and so on. Instead of transmitting the image of a face, we could transmit the parameters that define that face within our general model. Assuming that we have a suitable model for the data at hand, we may be able to describe the entire system using only a few bytes of parameter data.

Both sender and receiver share a large body of *a priori* knowledge contained in the model itself (e.g., the fact that faces have two eyes and one nose). The more information is shared in the model, the less need be transmitted with any given data set. Like wavelet compression, model-based compression works by characterizing data in terms of a deeper underlying generator. Model-based encoding has found applicability in such areas as computerized recognition of four-legged animals or facial expressions.

 Notes by Tzu-Yi Chen

- Cryptography
  1. Definitions and Primitives
  2. Protocols
     (a) Digital Signatures
     (b) Key Exchange
     (c) Authentication
     (d) Some Other Protocols
  3. Symmetric (private-key) Algorithms
     (a) Symmetric Ciphers
        i. Block Ciphers
     (b) DES (Data Encryption Standard)

In the original class the first 15 minutes was spent on Fractal and model compression. This material was incorporated into the lecture notes of the previous class.

1 Definitions and Primitives

Like compression, this is a large field in its own right. In this class we focus on the algorithmic aspects of the field, paying little or no attention to other issues such as politics.

We begin with a series of basic definitions, building up to some primitives.

1.1 Definitions

Cryptography General term referring to the field.

Cryptology The mathematics behind cryptography.

Encryption Encoding. Sometimes this is used as a general term.

Cryptanalysis The breaking of codes.

Steganography The hiding of messages. This is an emerging field which studies the hiding of messages inside messages. A simple example is the hiding of a watermark in a piece of paper.

Cipher A method or algorithm for encrypting and decrypting.

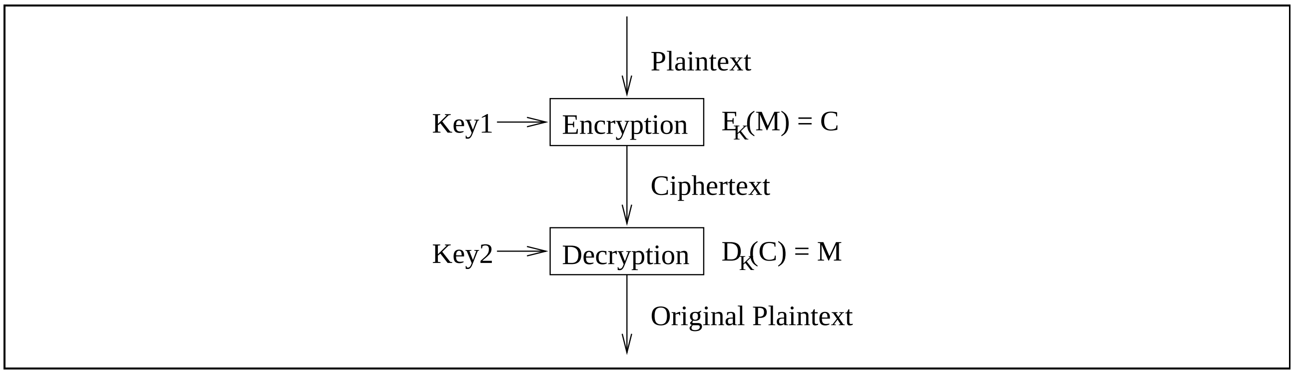


Figure 18: Flowchart showing the encryption and decryption of a message.

A few more terms are best described with the aid of Figure 18. Referring to Figure 18, algorithms in which the two keys **key1** and **key2** are the same are often called *symmetric* or *private-key* algorithms (since the key needs to be kept private) and algorithms in which the two keys are different are often called *asymmetric* or *public-key* algorithms (since either **key1** or **key2** can be made public depending on the application).

Next we list some commonly used names ...each name represents a specific role.

Alice initiates a message or protocol.

Bob second participant.

Carol third participant.

Trent trusted middleman.

Eve eavesdropper.

Mallory malicious active attacker.

Finally we define security by asking ourselves what it means to be secure. There are two definitions.

Truly Secure An algorithm is truly secure if encrypted messages cannot be decoded without the key.

In 1943 Shannon proved that for something to be truly secure, the key must be as long as the message. Informally, the intuition is that something is not secure if the message contains more information than the key.

An example of a secure algorithm is the *one-time-pad* in which the encoder and decoder both have the same random key which is the same length as the message. The encoder `XOR`s the message with the key, the decoder then decodes the message by `XOR`ing what he receives with the same random key. Each random key can only be used once.

Security Based on Computational Cost Here something is secure if it is "infeasible" the break — meaning that no (probabilistic) polynomial time algorithm can decode a message.

Notice that if $\mathcal{P} = \mathcal{NP}$, nothing is secure based on computational cost since we could effectively nondeterministically try all keys.

1.2 Primitives

Here we define several primitives, giving examples of most.

One-way Functions A one-way function $y = f(x)$ is a function where it is easy to compute y from x, but "hard" to compute x from y. One-way functions can be further classified based on the meaning of the word "hard".

Strong A strong one-way function f is one where for most y, no single polynomial time algorithm can compute x (i.e. $1 - \frac{1}{|x|^k}$).

Weak A weak one-way function f is one where for a reasonable number of y no single polynomial time algorithm can compute x (i.e. $\frac{1}{|x|^k}$).

Some examples of one-way functions are the following:

Factoring $y = u \cdot v$, where u and v are primes. If u and v are primes, it is hard to generate them from y.

Discrete Log $y = g^x \bmod p$, where p is a prime and g is a generator (i.e. $g^1, g^2, g^3, \ldots$ generate all i such that $0 \leq i < p$. This is believed to be about as hard as factoring.

DES with a fixed message $y = \mathrm{DES}_x(M)$.

Note that these are only conjectured to be one-way functions.

One-way Trapdoor Functions This is a one-way function with a "trapdoor". The trapdoor is a key that makes it easy to invert the function.

An example is RSA, where the function is $y = x^e \bmod N$, where $N = p \cdot q$. p,q, and e are prime, p and q are trapdoors (only one is needed). So, in public-key algorithms, $f(x)$ is the public key (e.g. e, N), whereas the trapdoor is the private key (e.g. p or q).

One-way Hashing Functions A one-way hashing function is written $y = H(x)$. H is a many-to-1 function, and y is often defined to be a fixed length (number of bits) independent of the length of x. The function H is chosen so that calculating y from x is easy, but given a y calculating any x such that $y = H(x)$ is "hard".

2 Protocols

Here we describe a number of basic protocols, showing possible implementations. We note that in general a protocol which can be implemented using a symmetric (private-key) cryptosystem can also be implemented using an asymmetric (public-key) cryptosystem — a primary difference is that the former may require a trusted middleman.

2.1 Digital Signatures

Digital signatures can be used to:

- Convince a recipient of the author of a message.
- Prove authorship.
- Make tampering with a signed message without invalidating the signature difficult.

They can be implemented both using symmetric and asymmetric cryptosystems as illustrated below.

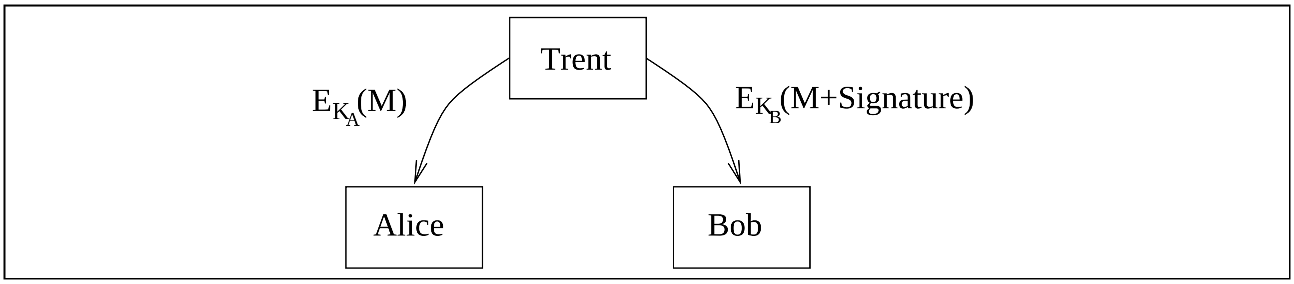


Figure 19: Flowchart showing an implementation of digital signatures using a symmetric cryptosystem.

Using a symmetric Cyptosystem: In Figure 19 Alice and Trent share the private key K_A and Bob and Trent share the private key K_B. Trent (a trusted middleman) receives a message from Alice which he can decrypt using K_A. Because he can decrypt it using K_A, he knows Alice sent the message. He now encrypts the decoded message plus a signature saying that Alice really sent this message using K_B. He sends this new version to Bob, who can now decode it using the key K_B.

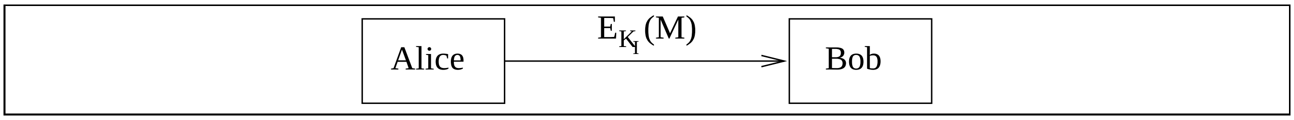


Figure 20: Flowchart showing an implementation of digital signatures using an asymmetric cryptosystem.

Using an asymmetric Cyptosystem: In Figure 20 K_I is Alice's private key. Bob decrypts the message using her public key. The act of encrypting the message with a private key that only Alice knows represents her signature on the message.

The following is a more efficient implementation using an asymmetric cryptosystem. In Figure 21 $H(M)$ is a one-way hash of M. Upon receiving M, Bob can verify that it indeed come from Alice by decrypting $H(M)$ using Alice's public key K_I and checking that M does in fact hash to the value encrypted.

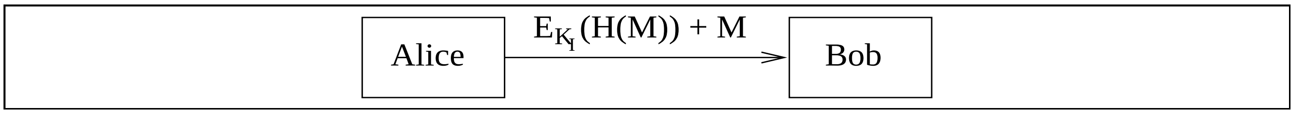


Figure 21: Flowchart showing a more efficient implementation of digital signatures using an asymmetric cryptosystem.

2.2 Key Exchange

These are protocols for exchanging a key between two people. Again, they can be implemented both with symmetric and asymmetric cryptosystems as follows:

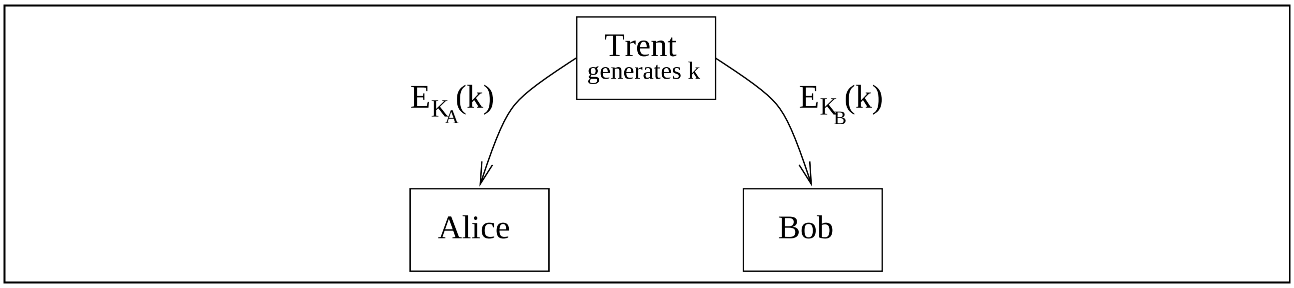


Figure 22: Flowchart showing an implementation of key exchange using a symmetric cryptosystem.

Using a symmetric Cyptosystem: In Figure 22, Alice and Trent share the private key K_A and Bob and Trent share the private key K_B. Trent generates a key k, then encodes k with K_A to send to Alice and encodes k with K_B to send to Bob.

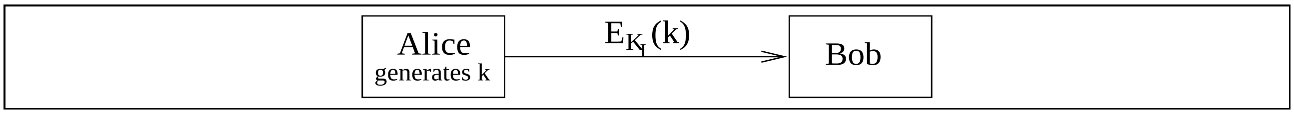


Figure 23: Flowchart showing an implementation of key exchange using an asymmetric cryptosystem.

Using an asymmetric Cyptosystem: In Figure 23 K_I is Bob's public key. Alice generates a key k, which she encrypts with Bob's public key. Bob decrypts the message using his private key.

2.3 Authentication

These protocols are often used in password programs (e.g. the Unix `passwd` function) where a host needs to authenticate that a user is who they say they are.

Figure 24 shows a basic implementation of an authentication protocol. Alice sends p to the host. The host computes $f(p)$. The host already has a table which has y and checks to see that $f(p) = y$. f is a one-way function.

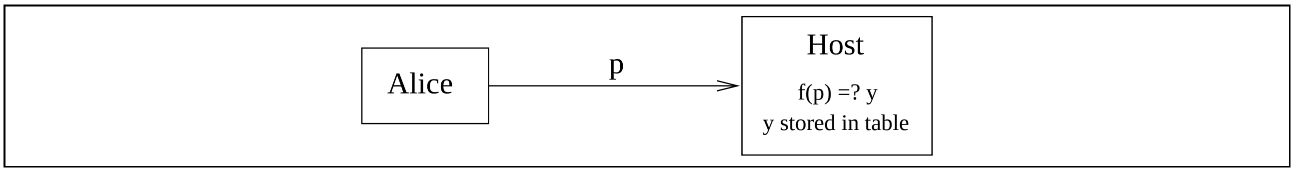


Figure 24: Flowchart showing a basic implementation of an authentication protocol.

To make it less likely that Mallory will be able to pretend to be Alice to the host, sequences of passwords can be used. Sequences of passwords are used by having Alice give the host a random number R. The host computes $x_0 = R, x_1 = f(R), x_2 = f(f(R)), \ldots x_n = f^n(R)$ and gives all but x_n to Alice. The host keeps x_n. Now, the first time Alice logs in, she uses x_{n-1} as the password. The host computes $f(x_{n-1})$ and compares it to the x_n which it has. If the two are equivalent, Alice has been authenticated. Now the host replaces x_n with x_{n-1}. The next time Alice logs in, she uses x_{n-2}. Because each password is only used once by Alice, and f is a one-way function, sequences of passwords are more secure.

2.4 Some Other Protocols

Some other protocols which are not discussed here are:

- Secret sharing
- Timestamping services
- Zero-knowledge proofs
- Blind-signatures
- Key-escrow
- Secure-elections
- Digital Cash

3 Symmetric (private-key) Algorithms

We discuss block ciphers and then briefly introduce the well known and widely used block cipher DES. We begin discussion of how DES works, saving much for the next lecture.

3.1 Symmetric Ciphers (Block Ciphers)

Block ciphers divide the message to be encrypted into blocks (e.g. 64 bits each) and then convert one block at a time. Hence:

$$C_i = f(k, M_i) \qquad\qquad M_i = f'(k, C_i)$$

Notice that if $M_i = M_j$, $C_i = C_j$. This is undesirable because if someone (Eve or Mallory) notices the same block of code going by multiple times, they know that this means identical blocks were coded. Hence they've gained some information about the original message.

There are various ways to avoid this. For example, something called *cipher block chaining (CBC)* encrypts as follows.

$$C_i = E_k(C_{i-1} \oplus M_i)$$

This has the effect that every code word depends on the code works that appear before it and are therefore not of any use out of context. Cipher block chaining can be decrypted as follows.

$$M_i = C_{i-1} \oplus D_k(C_i)$$

Unfortunately, this could still cause problems across messages. Note that if two messages begin the same way, their encodings will also begin the same way (but once the two messages differ, their encodings will be different from then on). Because many messages that people want to send begin with standard headers (e.g. specifying the type of file being sent), this means someone could gain information about the message being sent by creating a table of all the encrypted headers that they have seen. One way to fix this problem is to attach a random block to the front of every message.

3.2 DES (Data Encryption Standard)

DES was designed by IBM for NIST (the National Institute of Standards and Technology) with "consultation" from the NSA in 1975.

DES uses 64-bit blocks and 56-bit keys. It is still used heavily, for example by banks when they encrypt the PIN that you type in at an ATM machine. However, DES is no longer very safe — supercomputers can crack it in a few hours. Nevertheless, because it is so common, we describe how it works here, beginning by describing some of the computing components used in DES.

Specifically, we describe some bit manipulations:

Substitution A fixed mapping. For example:

$$\begin{array}{ccc} 000 & \rightarrow & 011 \\ 001 & \rightarrow & 101 \\ 010 & \rightarrow & 000 \\ 011 & \rightarrow & 001 \\ 100 & \rightarrow & 100 \\ & \vdots & \end{array}$$

This is often stored as a table.

Permutation A rearrangement of order. For example:

$$b_0b_1b_2b_3b_4 \quad \rightarrow \quad b_2b_4b_3b_0b_1$$

This is also often stored as a table.

Expansion Permutation A rearrangement of order that may repeat some bits of the input in the output. For example:

$$b_0 b_1 b_2 b_3 \quad \rightarrow \quad b_2 b_1 b_2 b_0 b_3 b_0$$

Notes by David Oppenheimer

Outline

- Why do DES and other block ciphers work?
- Differential and linear cryptanalysis
- Sample block cipher: IDEA
- Sample stream cipher: RC4
- Public-key algorithms
  - Merkle-Hellman Knapsack Algorithm
  - RSA

1 Block ciphers

Block ciphers encrypt plaintext in blocks and decrypt ciphertext in blocks.

1.1 Feistel networks

Many block ciphers take the form of *Feistel networks*. The structure of a Feistel network is illustrated in Figure 25.

In a Feistel network, the input is divided evenly into a left and right block, and the output of the i'th round of the cipher is calculated as

$$\begin{aligned} L_i &= R_{i-1} \\ R_i &= L_{i-1} \oplus f(R_{i-1}, K_i) \end{aligned}$$

The good property of Feistel networks is that f can be made arbitrarily complicated, yet the input can always be recovered from the output as

$$\begin{aligned} R_{i-1} &= L_i \\ L_{i-1} &= R_i \oplus f(L_i, K_i) \end{aligned}$$

Therefore even if f is not invertible, the algorithm itself will be invertible, allowing for decryption (if you know the K_i's, of course).

Feistel networks are part of DES, Lucipher, FEAL, Khufu, LOKI, GOST, Blowfish, and other block cipher algorithms. You can see the Feistel structure in one round of DES, which is depicted in Figure 26.

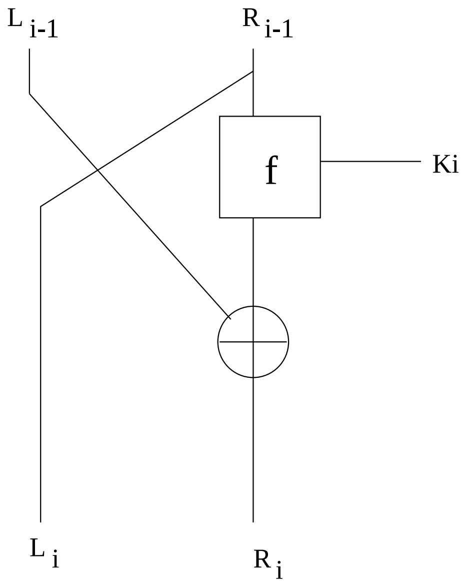


Figure 25: Feistel Network

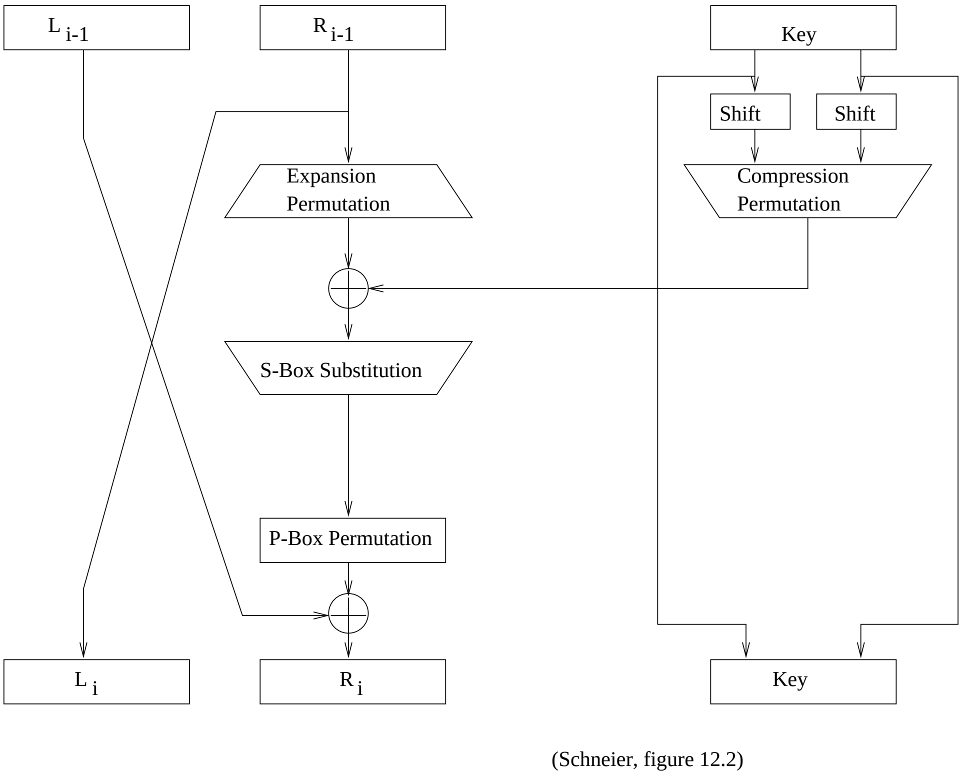


Figure 26: One round of DES (there are 16 rounds all together).

1.2 Security and Block Cipher Design Goals

This raises the question of how to design f so that the cipher is secure.

Confusion, diffusion, and the avalanche effect are three goals of block cipher design. *Confusion* decorrelates the output from the input. This can be achieved with *substitutions*. *Diffusion* spreads (diffuses) the correlations in the input over the output. This can be achieved with *permutations*: the bits from the input are mixed up and spread over the output. A good block cipher will be designed with an *avalanche effect*, in which one input bit rapidly affects all output bits, ideally in a statistically even way. By "rapidly" we mean after only a few rounds of the cipher.

The ideal f for a block cipher with 64-bit blocks would be a totally random 64-bit $\rightarrow$ 64-bit key-dependent substitution. Unfortunately this would require a table with $2^{64} \cdot 2^{56} = 2^{120}$ entries for a cipher with 56-bit keys (like DES): for each of the 2^{56} keys you'd need a mapping from each of the 2^{64} possible inputs to the appropriate 64-bit output.

Since this would require too much storage, we approximate the ideal by using a pseudorandom mapping with substitutions on smaller blocks. An algorithm that repeatedly performs *substitutions* on smaller blocks (so the storage requirements for the mapping are feasible) and *permutations* across blocks is called a *substitution-permutation network*. This is an example of a *product cipher* because it iteratively generates confusion and diffusion. A cipher with the same structure in each *round*, repeated for multiple rounds, is called an *iterated block cipher*. DES is an example of an iterated block cipher, a substitution-permutation network, and a product cipher.

1.3 DES Security

DES was designed with the above goals in mind. It was also designed to resist an attack called *differential cryptanalysis*, which was not publicly known at the time.

1.3.1 E-box and P-box

The DES *E-box* (labeled as the expansion permutation in Figure 26) takes 32 bits as input and produces 48 bits as output as shown in Figure 27. The 48 output bits are the 32 input bits with some of the input bits repeated. In particular, for each block of 4 input bits, the first and fourth bits appear twice in the output while the two middle bits appear once. The E-box therefore enhances the avalanche effect, because 4 outputs from an S-box in round R will become the input to 6 S-boxes in round $R+1$. The E-box also enables the use of 48 bits of the key in the XOR operation that follows the E-box.

The DES *P-Box* is a straight 32-bit permutation (every input bit is used exactly once in the output)

1.3.2 S-box

There are 8 S-boxes; each takes six bits as input and produces 4 bits as output. Some important security properties of the S-boxes are

- The output bits are not a linear function of the inputs, nor close to a linear function

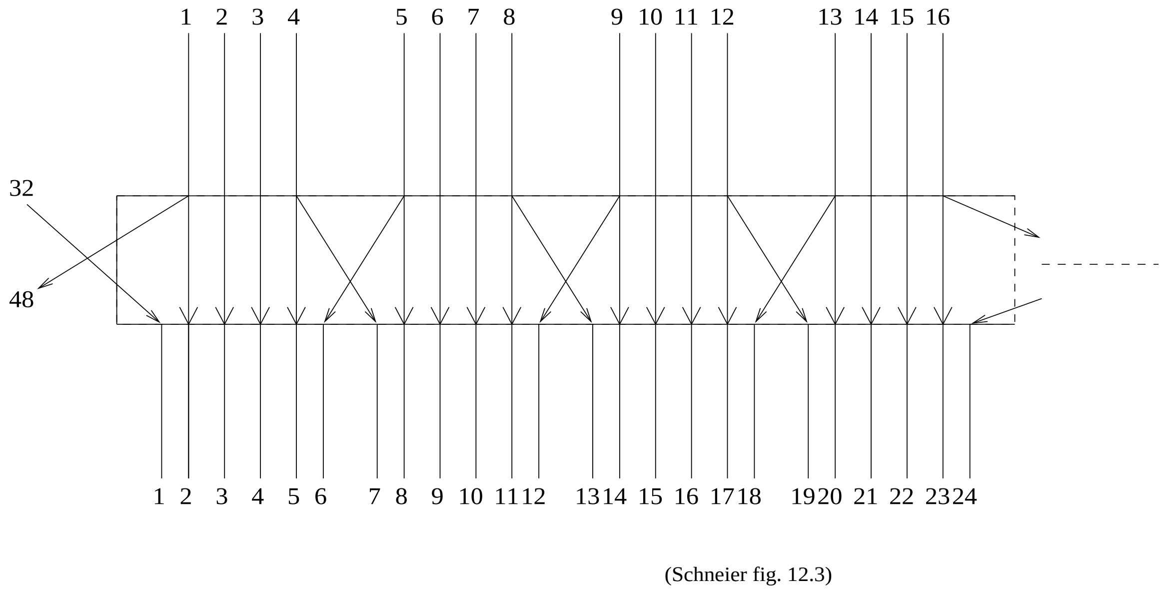


Figure 27: The DES E-box

- A difference of 1 bit in the input to an S-box affects at least 2 bits in the output
- For any 6-bit input difference $i_1 \oplus i_2$, no more than 8 of the 32 possible input pairs with difference $i_1 \oplus i_2$ may result in the same output difference.

We will see that this last property is important in making DES resistant to differential cryptanalysis.

1.3.3 Weaknesses of DES

- Some keys are insecure. Of the 2^{56} possible DES keys, 64 of them are poor choices. Luckily this is few enough that a DES encryption program can check to see if the key you want to use is one of these bad keys, and if so tell you not to use it.
  - Weak keys: There are four *weak keys*: 0x00000000000000, 0x0000000FFFFFFF, 0xFFFFFFF0000000, and 0xFFFFFFFFFFFFFF. When you use a weak key, the subkey used in each DES round is identical. This makes cryptanalysis a lot easier. It also makes $E_k(X) = D_k(X)$ since DES *decryption* follows the same algorithm as *encryption* but uses the subkeys in the reverse order. (If the subkeys are all the same, then obviously the forward and reverse order of them are identical.)
  - Semiweak keys: There are twelve *semiweak keys*. When you use a semiweak key, there are only two different subkeys used among the 16 DES rounds (each unique subkey is used 8 times). This makes cryptanalysis easier.
  - Possibly weak keys: There are forty-eight *possibly weak keys*. When you use a possibly weak key, there are only four unique subkeys generated; each one is used 4 times. Again, this makes cryptanalysis easier.

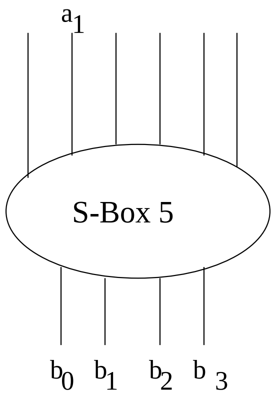


Figure 28: S-Box 5

- S-box 5 is strongly biased. Refer to Figure 28. $b_0 \oplus b_1 \oplus b_2 \oplus b_3 = a_1$ for 12 of the 64 possible inputs to S-box 5. Ideally this should be true for exactly half (32) of the possible inputs. Linear cryptanalysis can take advantage of this bias.

1.3.4 DES in the Real World

DES is used frequently in the "real world."

- Banks
  - bank machines
  - inter-bank transfers
- ISDN (some)
- Kerberos
- Unix password authentication (almost)
- Privacy-enhanced mail (PEM)

DES is not used in many newer systems because of the small 56-bit keysize. However, *triple-DES* (or *3-DES*) provides significantly more security while still using the DES algorithm. The basic idea is to encrypt multiple times using different keys. In 3-DES,

$$\begin{aligned} C &= E_{K1}(D_{K2}(E_{K3}(P))) \\ P &= D_{K3}(E_{K2}(D_{K1}(C))) \end{aligned}$$

Note that you *cannot* simply use $C = E_{K1}(E_{K2}(P))$ for DES (or any other block algorithm) and get much better security than simply using one key. If the algorithm is a group, then this is completely equivalent to saying $C = E_{K3}(P)$ for some K_3 and you have gained nothing. If the algorithm is not a group, there is a special attack which lets you find the key in 2^{n+1} encryptions for an algorithm with an n-bit key, instead of the expected 2^{2n} encryptions. DES is not a group.

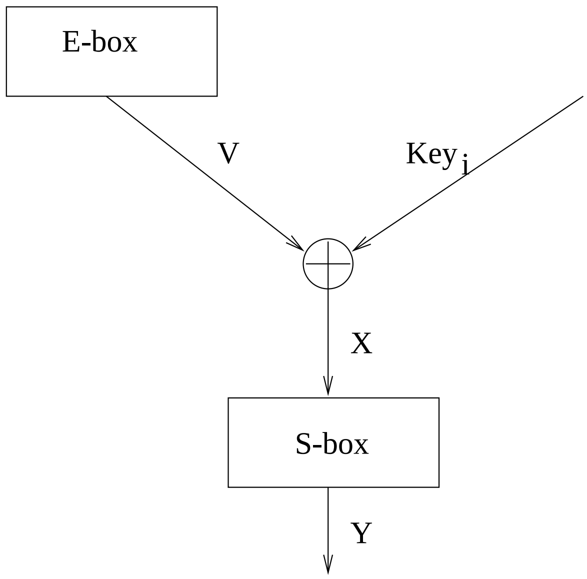


Figure 29: Differential Cryptanalysis

1.4 Differential Cryptanalysis

Define $\Delta V = V_1 \oplus V_2$ and $\Delta Y = Y_1 \oplus Y_2$.

Refer to Figure 29 which shows part of one round of DES. The basic idea behind differential cryptanalysis is that for any given ΔV, not all values of ΔY are equally likely. If a cryptanalyst is able to force someone to encrypt a pair of plaintext messages with a ΔV that he knows, he can look at the corresponding pair of encrypted messages, compute their ΔY, and use ΔV, ΔY, and the V's to get some information about Key_i. Of course, you need lots of pairs of messages before the deltas become statistically significant enough to give you information about the key.

This idea is applied one round at a time. The more rounds there are in the cipher, the harder it is to derive the original key using this technique because the statistical bias decreases with each round.

Differential cryptanalysis makes 14-round DES significantly easier to break than brute force, but has little effect on the security of 16-round DES. The designers of DES knew about differential cryptanalysis and designed the S-boxes (and chose the number of rounds) to resist this attack.

1.5 Linear Cryptanalysis

Linear cryptanalysis allows an attacker to gain some information about one bit of a key if he knows a plaintext and its corresponding ciphertext. The idea is that (the XOR of some plaintext bits) $\oplus$ (the XOR of some ciphertext bits), a one-bit by one-bit XOR that gives you a one-bit result, gives you information about the XOR of some of the key bits. This works when there is a bias in the linear approximation for an S-box. S-box 5 is the most biased S-box for DES.

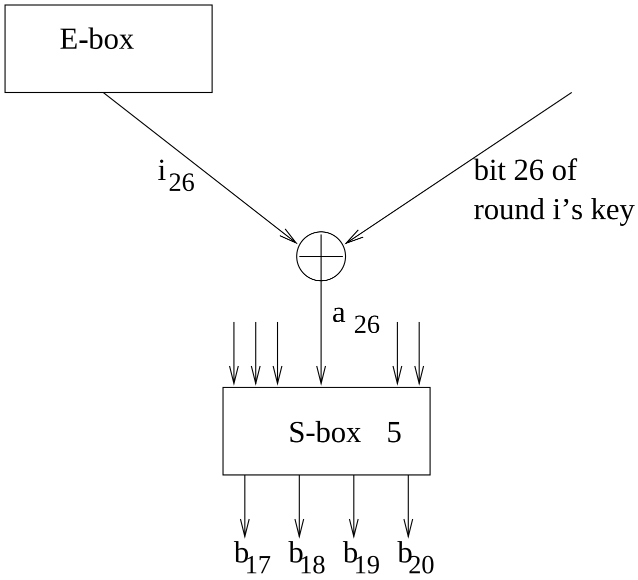


Figure 30: Linear Cryptanalysis using S-Box 5

Refer to Figure 30 which shows the operation of S-box 5. If you use linear cryptanalysis, knowing something about i_{26} (the second input bit to S-box 5, if we number input bits to the array of S-boxes starting with bit 1 on the left) and b_{17}, b_{18}, b_{19}, and b_{20} (the seventeenth through twentieth output bits from the array of S-boxes) will tell you something about the 26'th bit of the key used in that round. This is because (1) i_{26} is the XOR of the 26'th bit of the key used in that round, with the 26'th bit of the output from the expansion permutation for that round (which you know if you know the input bits to that round), and (2) S-box 5 has the following statistical bias: $i_{26} = b_{17} \oplus b_{18} \oplus b_{19} \oplus b_{20}$ for only 12 of the 64 possible inputs to S-box 5. Armed with enough plaintext-ciphertext pairs, linear approximations for the S-boxes, and a model of the S-boxes' biases, you can do statistical analysis to learn about the key.

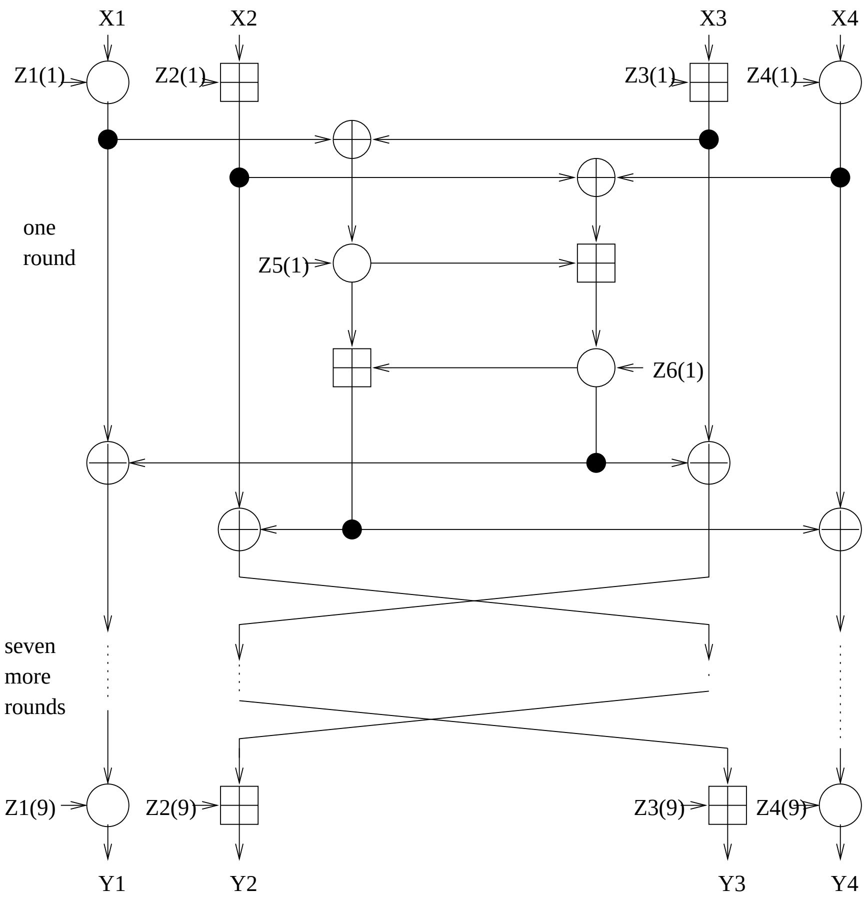
Linear cryptanalysis is the most successful attack on DES. Still, it's not much better than a brute force key search.

1.6 IDEA

IDEA, illustrated in Figure 31, is another block cipher. It operates on 64-bit blocks and uses 128-bit keys. IDEA is a product cipher. It composes three algebraic groups:

- Addition (which provides the diffusion property)
- Multiplication (which provides the confusion property)
- XOR

IDEA has some weak keys, but they're not weak in the same sense as DES keys. (If you use a weak IDEA key, an attacker can figure out which key you used by using a chosen-plaintext attack.) There are 2^{32} weak IDEA keys, but since there are so many more possible



Xi = 16-bit plaintext subblock

Yi = 16 bit ciphertext subblock

Zi(r) = 16-bit key subblock

$\oplus$ = bit-by-bit XOR of 16-bit subblocks

$\boxplus$ = addition modulo 2^{16} of 16-bit integers

$\bigcirc$ = multiplication modulo 2^{16} + 1 of 16-bit integers with the zero subblock corresponding to 2^{16}

(Schneier fig. 13.9)

Figure 31: The IDEA cipher

IDEA keys than DES keys, the chances of picking one of these randomly is smaller than the chance of picking one of the weak DES keys randomly.

Eight-round IDEA has stood up to all attacks to date. IDEA is used in PGP.

1.7 Block Cipher Encryption Speeds

Encryption Speeds of some Block Ciphers on a 33 MHz 486SX

algorithm	speed (KB/sec)	algorithm	speed (KB/sec)
Blowfish (12 rounds)	182	MDC (using MD4)	186
Blowfish (16 rounds)	135	MDC (using MD5)	135
Blowfish (20 rounds)	110	MDC (using SHA)	23
DES	35	NewDES	233
FEAL-8	300	REDOC II	1
FEAL-16	161	REDOC III	78
FEAL-32	91	RC5-32/8	127
GOST	53	RC5-32/12	86
IDEA	70	RC5-32/16	65
Khufu (16 rounds)	221	RC5-32/20	52
Khufu (24 rounds)	153	SAFER (6 rounds)	81
Khufu (32 rounds)	115	SAFER (8 rounds)	61
Luby-Rackoff (using MD4)	47	SAFER (10 rounds)	49
Luby-Rackoff (using MD5)	34	SAFER (12 rounds)	41
Luby-Rackoff (using SHA)	11	3-Way	25
Lucifer	52	Triple-DES	12

from Schneier, Table 14.3

2 Stream ciphers

2.1 Stream ciphers vs. Block cipher

Block ciphers operate on blocks of plaintext; for each input block they produce an output block. *Stream ciphers* usually encrypt/decrypt one bit at a time. The difference is illustrated in Figure 32, which depicts the operation of a typical block cipher and a typical stream cipher. Stream ciphers seem to be more popular in Europe, while block ciphers seem to be more popular in the United States.

A stream cipher uses a function (F in the picture) that takes a key and produces a stream of pseudorandom bits (b_i in the picture). One of these bits is XORed with one bit of the plaintext input (m_i), producing one bit of the ciphertext output (c_i).

A block cipher *mode* combines the cipher algorithm itself with feedback (usually). Figure 33 illustrates a block algorithm in CBC (Cipher Block Chaining) mode. Because the ciphertext for block *i-1* is used in calculating the ciphertext for block i, identical plaintext blocks will not always encrypt to the same ciphertext (as is true when a block algorithm is used directly without feedback).

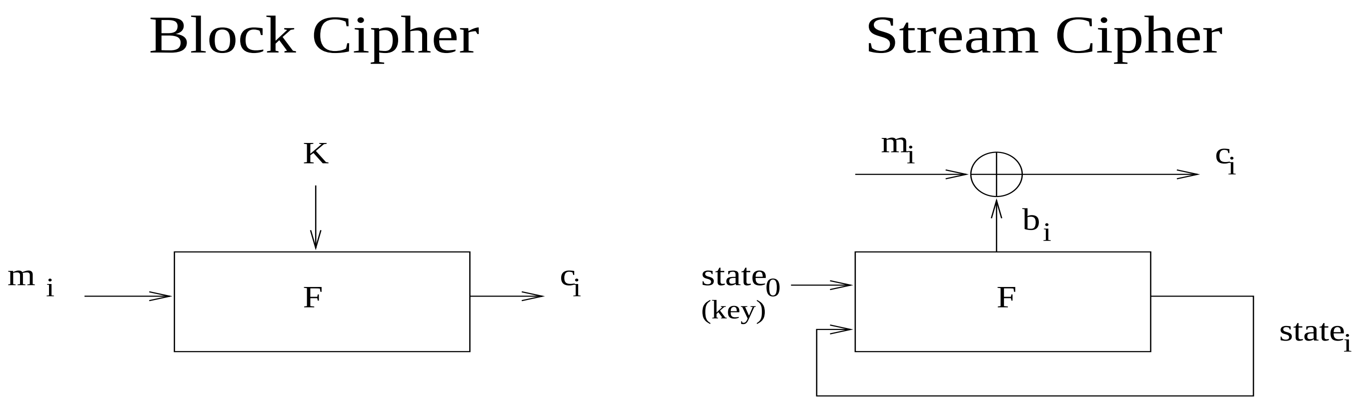


Figure 32: Block v. Stream Cipher

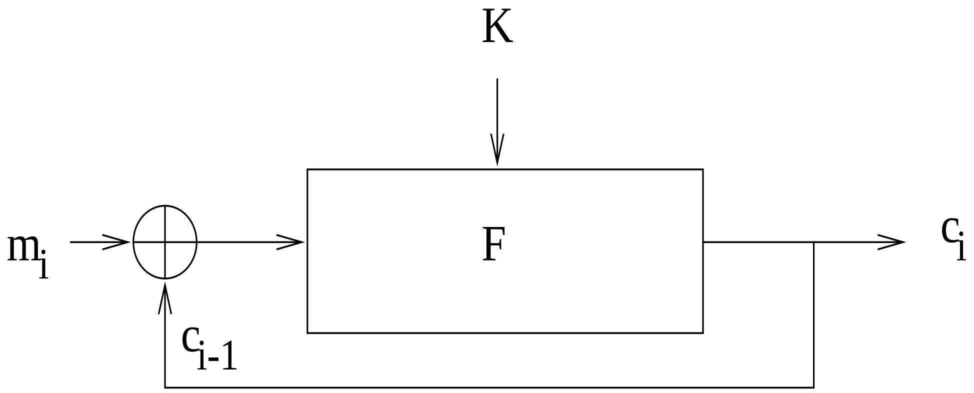


Figure 33: A block cipher in CBC mode

Block ciphers are typically more efficiently implemented in software than stream ciphers because block ciphers operate on many bits at a time.

2.2 RC4

RC4 is a stream cipher that generates one byte at a time. The key is used to generate a 256-element table $S_0, S_1, S_2, \ldots S_{255}$ which contains a permutation of the numbers 0 to 255. Once this table has been initialized, the following pseudocode for RC4 generates one byte on each round.

```
i=j=0
while (generating) {
        i = (i+1) mod 256
        j = (j+S_i) mod 256
        swap S_i and S_j
        t = (S_i + S_j) mod 256
        output = S_t
}
```

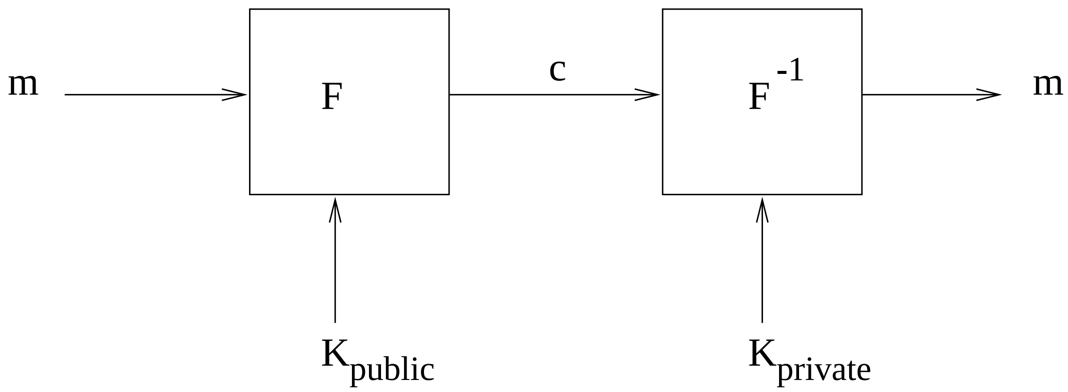


Figure 34: One-way trapdoor function

The byte-sized `output` is XORed with a byte of plaintext to produce a byte of ciphertext on each iteration.

RC4 is used in Lotus Notes, Oracle Secure SQL, and other products.

3 Public-Key Algorithms

Public-key algorithms are based on one-way trapdoor functions. As previously mentioned, a *one-way function* is "easy" to apply in the forward direction but "hard" to apply in the reverse direction. One-way functions with a *trapdoor* are "easy" to apply in the forward direction and "hard" to apply in the reverse direction *unless* you know a secret. If you know the secret, then the function is also "easy" to apply in the reverse direction.

Public key algorithms use two keys: a *private key* and a *public key*. Alice generates a public-private keypair. She publishes her public key and keeps the corresponding private key private. To send Alice a message, Bob encrypts using Alice's public key. To decrypt the message, Alice uses her private key. In contrast to symmetric encryption, two parties wishing to communicate using a public-key algorithm don't need to meet in person to exchange a secret key – they just need to look up each other's public keys in the public key directory. Exactly how to implement a secure public key distribution system (e.g. a secure public key directory) is quite another story.

Encrypting a message with a public key can be thought of as applying the "forward" direction of a one-way function. The corresponding private key is the "trapdoor" that makes the one-way function easy to reverse, allowing the encrypted message to be decrypted. This is illustrated in Figure 34.

3.1 Merkle-Hellman Knapsack Algorithm

The *Merkle-Hellman Knapsack Algorithm* is a public-key algorithm that derives its "security" from the NP-complete *knapsack problem*. The designers of this algorithm believed that to decrypt a message encrypted using the algorithm, a person would need to solve the knapsack problem. Unfortunately they were wrong: while the knapsack problem itself is NP-complete, the instances of the problem which are used for encryption using their algorithm make it possible to decrypt without having to solve an NP-complete problem.

The *knapsack problem* says: Given a vector M of weights and a sum S, calculate the boolean vector B such that $\sum_i B_i M_i = S$.

How can this be turned into a public-key algorithm? To encrypt, first break the input into blocks of size $\|M\|$ bits. (M should be a very long vector.) For each block calculate a sum $C_i = \sum_i B_i M_i$, where B_i is the plaintext. The series of C_i values calculated is the ciphertext. The vector M used to encrypt each block is the public key.

The corresponding private key is a different vector of weights; we'll call it M'. M' has the following important properties: (1) it is a series of superincreasing weights (we assume M was not), and (2) $\sum_i B_i M'_i = S$ – in other words, M' can also be used to solve for B given S.

The facts that makes this a sensible public-key algorithm are that (1) one can find B given a superincreasing M' in polynomial time, but not in polynomial time if given only a non-superincreasing M, and (2) it is easy to turn a superincreasing knapsack sequence into a normal knapsack sequence. (2) Makes it easy to compute your public key once you've selected your private key. Unfortunately, the way the authors generate the public key from the private key makes it possible for an adversary to compute the private key from the public key; the message can then be decrypted in polynomial time. This is the weakness in the algorithm.

The U.S. patent on the Merkle-Hellman knapsack public key algorithm expired on August 19, 1997.

3.2 RSA

While Merkle-Hellman's security was based on the difficulty of solving the knapsack problem, the RSA algorithm's security is based on the difficulty of factoring large numbers (in particular, products of large primes).

To compute a public-private keypair:

1. Pick big, random primes p and q
2. Calculate $n = pq$
3. Choose e relatively prime to $(p-1)(q-1)$. Note that e does not need to be random.
4. Calculate $d = e^{-1} \bmod (p-1)(q-1)$

The *public key* is (e, n). The *private key* is (d, n).

To encrypt: $c = m^e \bmod n$
To decrypt: $m = c^d \bmod n$

We'll now show that $E_d(E_e(m)) = m$.

$$\begin{aligned} E_d(E_e(m)) &= (m^e)^d \\ &= m^{ed} \end{aligned}$$

$$
\begin{aligned}
&= m^{k(p-1)(q-1)+1} \\
&= m \cdot m^{k(p-1)(q-1)} \\
&= m \cdot m^{(p-1)(q-1)^k} \\
&= m \cdot 1^k \\
&= m
\end{aligned}
$$

(Note that all operations are performed *mod n*.)
The one tricky part of this proof is the step $m \cdot m^{(p-1)(q-1)^k}$ (mod n) $= m \cdot 1^k$ (mod n). This step is explained in the next lecture after a discussion of some group theory. Briefly, it is because $\|Z_{pq}\| = (p-1)(q-1)$ and a theorem that says *For all finite groups G, for every a in G,* $a^{\|G\|} = I$, where in this case $I = 1$.

If factoring is easy, then breaking RSA becomes easy.

Notes by Marat Boshernitsan

- Some group theory
- RSA summary
- ElGamal
- Probabilistic Encryption (Blum-Goldwasser)
- Quantum Cryptography
- Kerberos case Study

1 Some group theory

1.1 Groups: Basics

A group is a set G with binary operation $*$ defined on G, such that the following properties are satisfied:

1. **Closure.** For all $a, b \in G$, we have $a * b \in G$.
2. **Associativity.** For all $a, b, c \in G$, we have $(a * b) * c = a * (b * c)$.
3. **Identity.** There is an element $I \in G$, such that for all $a \in G$, $a * I = I * a = a$.
4. **Inverse.** For every $a \in G$, there exists a unique element $b \in G$, such that $a * b = b * a = I$.

For example, integers **Z** under the operation of addition form a group: identity is 0 and the inverse of a is $-a$.

1.2 Integers modulo n

For prime p, Z_p is a group of integers modulo p:

- $Z_p \equiv \{1, 2, 3, ..., p-1\}$
- $* \equiv$ multiplication mod p
- $I \equiv 1$
- $a^{-1} \equiv a * a^{-1} = 1 \pmod{p}$

Example: $Z_7 = \{1, 2, 3, 4, 5, 6\}$. Note that if p is **not** a prime, Z_p is not a group under $*$, because there isn't necessarily an inverse. All encryption algorithms assume existence of an inverse. So, for n not prime, we define:

- $Z_n^* \equiv \{m \ : \ 1 \leq m < n, \ gcd(n, m) = 1\}$ (i.e. integers that are relatively prime to n)
- $* \equiv$ multiplication mod n
- $I \equiv 1$
- $a^{-1} \equiv a * a^{-1} = 1 \pmod{p}$

Example: $Z_{10}^* = \{1, 3, 7, 9\}$, $1^{-1} = 1$, $3^{-1} = 7$, $7^{-1} = 3$, $9^{-1} = 9$. It is not too hard to show that the size of Z_n^* is $\prod_{p \in P} p \left(1 - \frac{1}{p}\right)$ where P are the primes dividing n, including n if it is prime. If n is a product of two primes p and r, then $|Z_n^*| = pr(1-1/p)(1-1/r) = (p-1)(r-1)$.

1.3 Generators (primitive elements)

For a group G, element $a \in G$ is a **generator of** G if $G = \{a^n \ : \ n \in Z\}$ (i.e. G is "generated" by the powers of a). For Z_7 we have:

x	x^2	x^3	x^4	x^5	x^6
1	1	1	1	1	1
2	4	1	2	4	1
3	**2**	**6**	**4**	**5**	**1**
4	2	1	4	2	1
5	**4**	**6**	**2**	**3**	**1**
6	1	6	1	6	1

Here, 3 and 5 are the generators of Z_7 while 1, 2, 4 and 6 are not since none of them will generate 3. For Z_{10}^* we have:

x	x^2	x^3	x^4
1	1	1	1
3	**9**	**7**	**1**
7	**9**	**3**	**1**
9	1	9	1

So, 3 and 7 are the generators of Z_{10}^*. Note that any element to the power of the size of the group gives identity. In fact, there's a theorem stating that this is always true:

Theorem 1 *For all finite groups G and for any $a \in G$, $a^{|G|} = I$, where I is the identity element in G.*

From this theorem, it follows that for $a \in Z_p$, $a^{|Z_p|} = a^{p-1} = 1$, and for two primes p and q and $a \in Z_{pq}^*$, $a^{|Z_{pq}^*|} = a^{(p-1)(q-1)} = I$. The latter fact is essential for RSA.

1.4 Discrete logarithms

If g is a generator of Z_p (or Z_p^* if p is not prime), then for all y there is a unique x such that $y = g^x \pmod{p}$. We call this x the **discrete logarithm** of y, and use the notation $\log_g(y) = x \pmod{p}$.

For Example: in Z_7, $\log_3(6) = 3$, but $\log_4(5) = \emptyset$ (no inverse), and $\log_4(2) = \{2, 5\}$ (multiple inverses) because 4 is not a generator of Z_7.

2 Public Key Algorithms (continued)

2.1 RSA algorithm (continued)

Here are the 5 steps of the RSA algorithms, again

1. Pick two random large primes p and q (on the order of a few hundred digits). For maximum security, choose p and q to be of the same length.

2. Compute their product $n = pq$.

3. Randomly choose the encryption key e, such that e and $(p-1)(q-1)$ are relatively prime. Then, find the decryption key d, such that $ed \bmod (p-1)(q-1) = 1$. I.e., $ed = k(p-1)(q-1) + 1$, for some integer k. The numbers e and n are the public key, and the numbers d and n are the private key. The original primes p and q are discarded.

4. To encrypt a message m: $c = m^e \pmod{n}$.

5. To decrypt a cipher c: $m = c^d \pmod{n}$.

Why does this work? Note that $m = c^d = (m^e)^d = m^{ed} = m^{k(p-1)(q-1)+1} = (m^{(p-1)(q-1)})^k m$ (all operations mod$n$). But m is relatively prime to n (because $n = pq$ and p, q are primes), and so m is in Z_n^*. By the theorem in Section 1.3, m to the power of the size of Z_n^* is 1, so $(m^{(p-1)(q-1)})^k m = 1^k m = m$. In practice, we pick small e to make m^e cheaper.

The performance figures of RSA (relative to block ciphers) are fairly unimpressive. Running on Sparc 2 with 512-bit n, 8-bit e, the encryption throughput is 0.03 secs/block, which translates into 16 Kbits/sec. The decryption throughput is even worse with 0.16 secs/block (4 Kbits/sec). Such speeds are too slow even for a modem. With special-purpose hardware (circa 1993), we can boost the throughput to 64 Kbits/sec which is fast enough for a modem, but is slow for pretty much everything else. In comparison, IDEA block cipher implemented in software can achieve 800 Kbits/sec. Because of its slowness, RSA is typically used to exchange secret keys for block ciphers that are then used to encode the message.

In the "real world", the RSA algorithm is used in X.509 (ITU standard), X9.44 (ANSI proposed standard), PEM, PGP, Entrust, and many other software packages.

2.2 Factoring

The security of RSA is based on the difficulty of factoring large numbers. The state of the art in factoring is as follows:

- Quadratic sieve (QS): $T(n) = e^{(1+O(1))(\ln(n))^{\frac{1}{2}}(\ln(\ln(n)))^{\frac{1}{2}}}$. This algorithm was used in 1994 to factor 129-digit (428-bit) number. It took 1600 Machines 8 months to complete this task.
- Number field sieve (NFS): $T(n) = e^{(1.923+O(1))(\ln(n))^{\frac{1}{3}}(\ln(\ln(n)))^{\frac{2}{3}}}$. This algorithm is about 4 times faster than QS on the 129-digit number and is better for numbers greater than 110 digits. It was used in 1995 to factor 130 digit (431-bit) number with approximately the same amount of work as taken by QS on 129-digits.

2.3 ElGamal Algorithm

The ElGamal encryption scheme is based on the difficulty of calculating discrete logarithms. The following are the steps of ElGamal:

1. Pick a prime p.
2. Pick some g, such that g is a generator for Z_p.
3. Pick random x and calculate y, such that $y = g^x \pmod p$. Now, distribute p, g, and y as the public key. The private key will be x.
4. To encrypt, choose a random number k, such that k is relatively prime to $p-1$. Calculate $a = g^k \pmod p$ and $b = y^k m \pmod p$. The cypher is the pair (a, b).
5. To decrypt, calculate $m = \frac{b}{a^x} \pmod p$.

Why does ElGamal work? Because $m = \frac{b}{a^x} = \frac{y^k m}{g^{kx}} = \frac{g^{kx} m}{g^{kx}} = m \pmod p$. To break ElGamal, one would need to either figure out x from y or k from a. Both require calculating discrete logs which is difficult.

Performance-wise, ElGamal takes about twice the time of RSA since it calculates two powers. It is used by TRW (trying to avoid the RSA patent).

3 Probabilistic Encryption

One of the problems with RSA encryption is that there is a possibility of gaining some information from public key. For instance, one could use public key E_{public} to encrypt randomly generated messages to see if it is possible to get some information about the original message.

This problem can be remedied by mapping every message m to multiple c randomly. Now, even if cryptoanalist has c, m, and E_{public}, she cannot tell whether $c = E_k(m)$ (i.e. k was used to encrypt m).

One such probabilistic encryption scheme, Blum-Goldwasser, is based on generating strings of random bits.

3.1 Generation Random Bits

This method is called Blum-Blum-Shub (BBS) and is based on the difficulty of factoring: given a single bit, predicting the next bit is as hard as factoring. Here are the steps:

1. Pick primes p and q, such that $p \bmod 4 = q \bmod 4 = 3$.
2. Calculate $n = pq$ (called Blum integer).
3. For each m we want to encrypt, choose random seed x that is relatively prime to n ($gcd(x, n) = 1$).
4. The state of the algorithm is calculated as follows:
   - $x_0 = x^2 \pmod{n}$
   - $x_i = x_{i-1}^2 \pmod{n}$
5. The i^{th} bit is then calculated by taking the least-significant bit of x_i.

Note that now we have $x_i = x_0^{2^i \bmod (p-1)(q-1)} \pmod{n}$ and $x_0 = x_i^{-2^i \bmod (p-1)(q-1)} \pmod{n}$. In other words, given x_0 and th prime factors p and q of n it is not hard to generate any i^{th} bit and vice versa. We shall now see how these properties can be used to construct an encryption scheme.

3.2 Blum-Goldwasser

Blum-Goldwasser is a public key probabilistic encryption scheme using BBS. The public key is n, the private key is the factorization of n, i.e. p and q. The following are the steps to encrypting a message:

1. For each bit m_i of the message $(0 \leq i < l)$ compute $c_i = b_i \otimes m_i$ where b_i is i^{th} bit of BBS (as in usual stream cipher).
2. Append x_l to the end of message, where l is the length of ciphertext in bits.

To decrypt, we use the private key p and q to generate x_0 from x_l ($x_0 = x_l^{-2^l \bmod (p-1)(q-1)}$) and then use the BBS formula to regenerate x_i, and hence b_i. The security of this scheme is based on difficulty of calculating x_0 from x_l without knowing the factorization of n.

Note that this encryption scheme achieves the desired probabilistic property: each time we code the same message with the same key we get different cipher (provided x is chosen randomly for each message). Furthermore the extra information we need to send (x_l) is minimal.

4 Quantum Cryptography

[This section has been expanded based on the material from
`http://eve.physics.ox.ac.uk/NewWeb/Research/crypto.html`]

Quantum cryptography makes use of the fact that measuring quantum properties can change the state of a system, e.g. measuring the momentum of a particle spreads out its position. Such properties are called *non-commuting observables*. This implies that someone eavesdropping on a secret communication will destroy the message making it possible to devise a secure communication protocol.

Photon polarization is one such property that can be used in practice. A photon may be polarized in one of the four polarizations: 0, 45, 90, or 135 degrees. According to the laws of quantum mechanics, in any given measurement it is possible to distinguish only between rectilinear polarizations (0 and 90), or diagonal polarizations (45 and 135), but not both. Furthermore, if the photon is in one of the rectilinear polarizations and someone tries to measure its diagonal polarization the process of measurement will destroy the original state. After the measurement it will have equal likelihood of being polarized at 0 or 90 independent of its original polarization.

These properties lead to the following is a key exchange protocol based on two non-commuting observables (here, rectilinear and diagonal polarizations):

1. Alice sends Bob photon stream randomly polarized in one of 4 directions.

2. Bob measures photon polarization, choosing the kind of measurement (diagonal or rectilinear) at random. Keeping the results of the measurement private, Bob then tells Alice (via an open channel), the kinds of measurement that he used.

3. Alice tells Bob which of his measurements are correct (also via an open channel).

4. Bob and Alice keep only the correct values, translating them into 0's and 1's.

Although this scheme does not prevent eavesdropping, Alice and Bob will not be fooled, because any effort to tap the channel will be detected (Alice and Bob will end up with different keys) and the eavesdropper will not have the key either (since she is very unlikely to have made the same measurements as Bob).

While practical uses of quantum cryptography are yet to be seen, a working quantum key distribution system has been built in a laboratory at the IBM T. J. Watson research center. This prototype implementation can transmit over admittedly modest length of 30cm at rate it 10 bits/second. Nevertheless, as new photon sources, new photo-detectors and better optical fibers are being built, we shall see quantum cryptography becoming more practical.

5 Kerberos (A case study)

Kerberos is a trusted authentication protocol. A Kerberos service, sitting on the network, acts as a trusted arbitrator. Kerberos provides secure network authentication, allowing

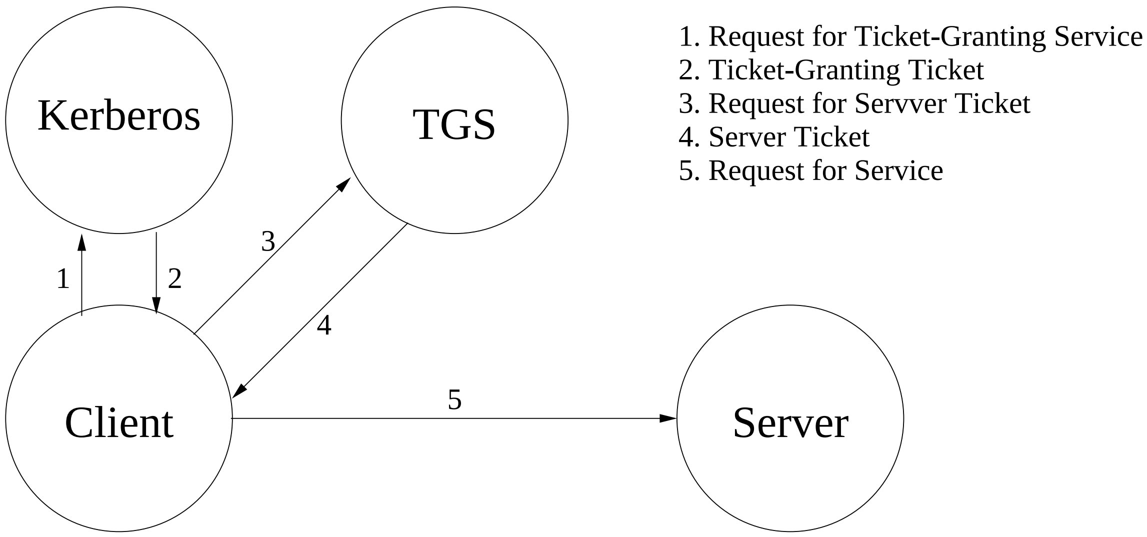


Figure 35: Kerberos authentication steps.

c	client
s	server
a	client's network address
v	beginning and ending validity time for ticket
t	timestamp
K_x	x's secret key
$K_{x,y}$	session key for x and y
$\{m\}K_x$	m encrypted in x's secret key
$T_{x,y}$	x's ticket to use y
$A_{x,y}$	authenticator from x to y

Table 8: Kerberos Notation

a person to access different machines on the network. Kerberos is based on symmetric cryptography (DES).

5.1 How Kerberos Works

The basic Kerberos authentication steps are outlined in Figure 35. The client, a user or an independent software program, requests a *Ticket-Granting Ticket (TGT)* from Kerberos server located on a trusted and secure machine. Kerberos replies with this ticket, encrypting it with client's secret key. To use the requested service, the client requests a ticket for that server from *Ticket-Granting Sever (TGS)*. TGS replies to client with the server ticket, which the client uses along with authenticator to request services from the server.

5.2 Tickets and Authenticators

Kerberos uses two kinds of credentials: tickets and authenticators. The tickets have the following format (refer to Table 8 for notation):

$$T_{c,s} = s, \{c, a, v, K_{c,s}\}K_s \tag{1}$$

A ticket is good for a single server and a single client. It contains the client's name and network address c, a, the server's name s, a timestamp v, and a session key $K_{c,s}$. Once the client gets this ticket, he can use it multiple times to access the server – until the ticket expires.

The authenticator takes this form:

$$A_{c,s} = \{c, t, key\}K_{c,s} \tag{2}$$

The client generates an authenticator each time he wants to use service on a server. The authenticator contains the clients name c, a timestamp t, and an optional additional session key. An authenticator can only be used once, but the client can regenerate authenticators as needed.

5.3 Kerberos messages

There are five Kerberos messages:

1. Client to Kerberos: c, tgs. This message is sent whenever the client wants to talk to Ticket-Granting Server. Upon receipt of this message, Kerberos looks up client in its database and generates the session key to be used between the client and TGS.

2. Kerberos to client: $\{K_{c,tgs}\}K_c, \{T_{c,tgs}\}K_{tgs}$. There are two parts in this message, one containing session key to be used between client and TGS ($K_{c,tgs}$) encrypted with clients key (K_c), and the other containing TGT ($T_{c,tgs}$) encrypted with TGS's secret key (K_{tgs}). The client cannot decrypt this latter part, but uses it to talk to TGS. Note that Kerberos does not communicate with TGS directly and the tickets have limited lifetime.

3. Client to TGS: $\{A_{c,s}\}K_{c,tgs}\{T_{c,tgs}\}K_{tgs}$. This is the message sent by the client to TGS. The first part of this message contains information that proves client's identity to TGS. The second part is precisely the TGT Kerberos sent to client.

4. TGS to client: $\{K_{c,s}\}K_{c,tgs}\{T_{c,s}\}K_s$. This message is similar to Kerberos' response to clients initial request. It contains the session key to be used between client and the server as well as the ticket for the client to use when requesting service. These also have limited lifetimes, depending on the kind of service requested.

5. Client to server: $\{A_{c,s}\}K_{c,s}\{T_{c,s}\}K_s$. With this message the client authenticates itself to the server as well as presents it with the ticket it received from TGS.

- Linear Programming and Optimization
  1. Introduction
  2. Geometric Interpretation
  3. Simplex Method
  4. Improvements to Simplex

1 Introduction

Optimization is a very important problem in industry. Numerous companies use it to streamline their operations by finding optimal tradeoffs between demand for products or services, manufacturing costs, storage costs, repair costs, salary costs, etc.

- 50+ software packages available
- 1300+ papers just on interior-point methods
- 100+ books in the library
- 10+ courses at most universities
- Dozens of companies
- Delta Airlines claims they save $100 million a year with their optimization application

1.1 Some Optimization Problems

Optimization problems typically try to minimize (or maximize) some cost function, which is often called the *objective function* given some set of constraints. Here are some common optimization problems

Unconstrained Optimization:

$$\min\{f(x) : x \in \mathcal{R}^n\}$$

where f is an objective function of n real values.

Constrained Optimization:

$$\min\{f(x) : c_i(x) \leq 0, i \in \mathcal{I}, c_i(x) = 0, i \in \mathcal{E}, x \in \mathcal{R}^n\}$$

where $\mathcal{I}$ is a set of indices of inequalities, $\mathcal{E}$ is a set of indices of equalities, and c_i is a set of constraint functions each on n variables.

Quadratic Programming:

$$\min\{\frac{1}{2}x^TQx + c^Tx : a_i^Tx \leq b_i, i \in \mathcal{I}, a_i^Tx = b_i, i \in \mathcal{E} x \in \mathcal{R}^n\}$$

Here we constrain the objective function to be quadratic in the input variables x and the constraint functions to be linear.

Linear Programming:

$$\min\{c^Tx : Ax \geq b, x \geq 0, x \in \mathcal{R}^n, c \in \mathcal{R}^n, b \in \mathcal{R}^m, A \in \mathcal{R}^{m\times n}\}$$

Here both the objective and the constraint functions are linear. We also do not include any equalities, but these are easy to express with the inequalities. We will see that there are many equivalent formats to express linear programs.

Integer Linear Programming:

$$\min\{c^Tx : Ax \geq b, x \geq 0, x \in \mathcal{Z}^n, c \in \mathcal{R}^n, b \in \mathcal{R}^m, A \in \mathcal{R}^{m\times n}\}$$

Same as the linear program except that the variables x are integers instead of reals. Often referred to as just Integer Programs (IPs). We will also consider problems in which some of the variables are integers and some are reals. These are called *mixed integer linear programs* (MIPs).

In this class and the next we will be discussing solutions to linear programs. In the following two classes we will discuss integer programs. Note that while linear programs can be solved in polynomial time, solving integer programs is NP-hard. We will not discuss non-linear programs (general constrained optimization), although some of the solutions we will discuss for linear programs, *e.g.* the interior point methods, can also be applied to nonlinear objective functions.

1.2 Applications of Linear Programming

- As a substep in pretty much all the integer and mixed-integer linear programming (MIP) techniques.
- Selecting a mix: Oil mixtures, portfolio selection, ...
- Distribution: How much of a commodity should distribute to different locations.
- Allocation: How much of a resource should allocate to different tasks.
- Network Flows

Although linear programs do have a reasonable number of direct applications in industry, most larger applications require MIP programs since

1. many resources are integral, and
2. integer variables can be used to model yes/no decisions (0-1 programming).

Therefore the most used application of linear programming is probably the first one mentioned above (substep of MIP programs).

Name	Rows	Columns	Nonzeros
binpacking	42	415,953	3,526,288
distribution	24,377	46,592	2,139,096
forestry	1,973	60,859	2,111,658
maintenance	17,619	79,790	1,681,645
crew	4,089	121,871	602,491
airfleet	21,971	68,136	273,539
energy	27,145	31,053	268,153
4color	18,262	23,211	136,324

Table 9: Examples of Large Models

1.3 Overview of Algorithms

- Simplex (Dantzig 1947)
- Ellipsoid (Khachian 1979) - the "first" polynomial-time algorithm
- Interior Point - the "first" practical polynomial-time algorithm
  - Projective Method (Karmarkar 1984)
  - Affine Method (Dikin 1967)
  - Logarithmic Barrier Method (Frisch 1955, Fiacco 1968, Gill et. al 1986)

The word "first" appears in quotes because other polynomial-time algorithms were around when Khachian and Karmarkar published their work, but they were not known to be polynomial-time algorithms until after these algorithms were published. Note that many of the interior point methods can be applied to nonlinear programs.

In the constraint equation $Ax \geq b$ of a linear program, x and b are vectors and A is a matrix. Each row of A corresponds to one constraint equation and each column corresponds to one variable. In practice the matrix A is often sparse (most entries are zero) since each constraint typically only depends on a few of the variables. This allows many of the algorithms to use special, fast subroutines designed to work with sparse-matrices. The current state of the art in such solvers allows us to work with problems where the matrices have 10-100 thousand rows/columns, and where 100 thousand to 1 million of the items in the matrices are nonzeros. Table 9 shows the size of some problems.

Though the simplex algorithm is not a polynomial-time algorithm in the general case (contrived examples can take exponential time), it is very fast for most problems. For this reason, it is still an open question as to whether or not interior point methods are "superior" to the simplex method and for what problem types. Here are two quotes that illustrate the battle between simplex and interior point methods:

- "The results of this paper demonstrate that the initial claims of the computational superiority of interior point methods are now beginning to appear." - Lustig, Marsten and Shanno, 1994 (ORSA Journal of Computing, 6 (1)).

Name	Simplex (Primal)	Simplex (Dual)	Barrier + Crossover
binpacking	29.5	62.8	560.6
distribution	18,568.0	won't run	12,495,464
forestry	1,354.2	1,911.4	2,348.0
maintenance	57,916.3	89,890.9	3,240.8
crew	7,182.6	16,172.2	1,264.2
airfleet	71,292.5	108,015.0	37,627.3
energy	3,091.1	1,943.8	858.0
4color	45,870.2	won't run	44,899,242

Table 10: Running Times of Large Models in seconds.

- "It is our opinion that the results of Lustig, Marsten and Shanno somewhat overstate the performance of [the interior point methods] relative to the simplex method." - Bixby, 1994 (ORSA Journal of Computing, 6 (1)).

Table 10 shows a comparison of the simplex method (running on either the original problem, or the *dual*, as explained in the next lecture) and an interior point method, the Barrier + Crossover method. The example problems are the same ones in Table 9. Note that the algorithms have widely varying performance depending on the problem given. The running time of both algorithms is very much dependent on the type of problem. This is part of why it is difficult to see whether or not interior point methods are "superior" to the simplex method.

It is clear, however, that (1) interior point methods are becoming more dominant over time and (2) the appearance of interior point methods has sparked a resurgence of interest in finding more efficient techniques for the simplex method.

1.4 Linear Programming Formulation

There are many ways to formulate linear programs, and different algorithms often use different formulations as inputs. Fortunately it is typically easy to convert among the formulations. For example the formulation presented earlier

minimize $c^T x$ (this is the *objective* function)
subject to $Ax \geq b$, $x \geq 0$

can be converted to

minimize $c^T x$
subject to $Ax = b$, $x \geq 0$

by adding *slack* variables. For example:

$$7x_1 + 5x_2 \geq 7$$

can be converted to

$$7x_1 + 5x_2 - y_1 = 7 \text{ and } y_1 \geq 0$$

That is, slack variables like y_1 must be added to the vector x for each row or A to convert the expression $Ax \geq b$ to $Ax = b$. This formulation with the equality constraints $Ax = b$ is the formulation that is used by the simplex method.

1.5 Using Linear Programming for Maximum-Flow

Before describing the algorithms for solving linear programs we describe how linear programs can be use to solve maximum flow problems. Consider the flow problem shown in Figure 36. In this problem the numbers on the edges represent the maximum flow allowed through the edges (its capacity). The goal is to maximize the total flow from the input (IN) to the output (OUT) with the constraints that the net flow into each node has to be zero, and no flow along an edge can be greater than the capacity. To convert this to a linear programming problem, take every edge e_i and create two edges, x_i and x'_i, one for positive flow in one direction, and the other for positive flow in the other direction. The linear programming formulation will have one variable for each direction of each edge.

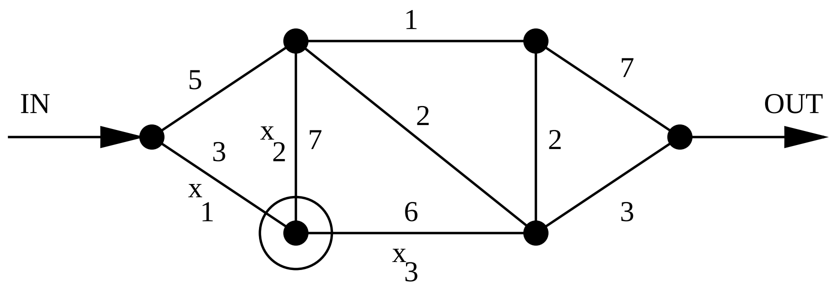


Figure 36: A network maximum-flow problem.

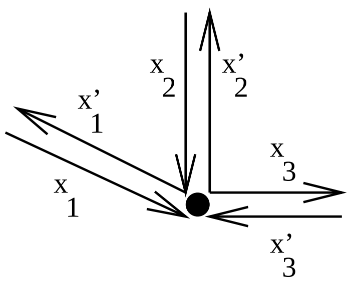


Figure 37: Part of a Network Converted for Linear Programming

For each vertex, we add a constraint that forces the flow into the vertex to equal the flow out of the vertex. Look at the highlighted vertex in Figure 36 connected to the edges labeled x_1, x_2, and x_3. This will be converted to the network shown in Figure 37. Adding the constraint that input flow equal output flow, we have

$$x_1 + x_2 + x'_3 = x'_1 + x'_2 + x_3$$

We then add two inequality constraints per edge to include the capacity constraints. For edge x_1 in Figure 36, these inequalities will be $x_1 \leq 3$ and $x_1' \leq 3$. Finally, we add one more edge x_0 connecting "OUT" to "IN", but we place no constraints on this flow. This completes the formulation of the network flow problem as a linear programming problem. Maximizing the value of x_0 will find the maximal flow for this network.

2 Geometric Interpretation

We can visualize linear programming problems geometrically. We present a two dimensional example. (Here we are maximizing instead of minimizing, but the ideas are still the same.)

$$\begin{aligned} \text{maximize } z &= 2x_1 + 3x_2 \\ \text{subject to } x_1 - 2x_2 &\leq 4 \\ 2x_1 + x_2 &\leq 18 \\ x_2 &\leq 10 \\ x_1, x_2 &\geq 0 \end{aligned}$$

Each inequality represents a halfspace, as shown in Figure 38. The intersection of these halfspaces is the region of points that abide by all the constraints. This is called the *feasible region*. The figure also shows that the objective function increases in one direction. The maximal value of x is the point in the feasible region that is farthest in the direction of this vector.

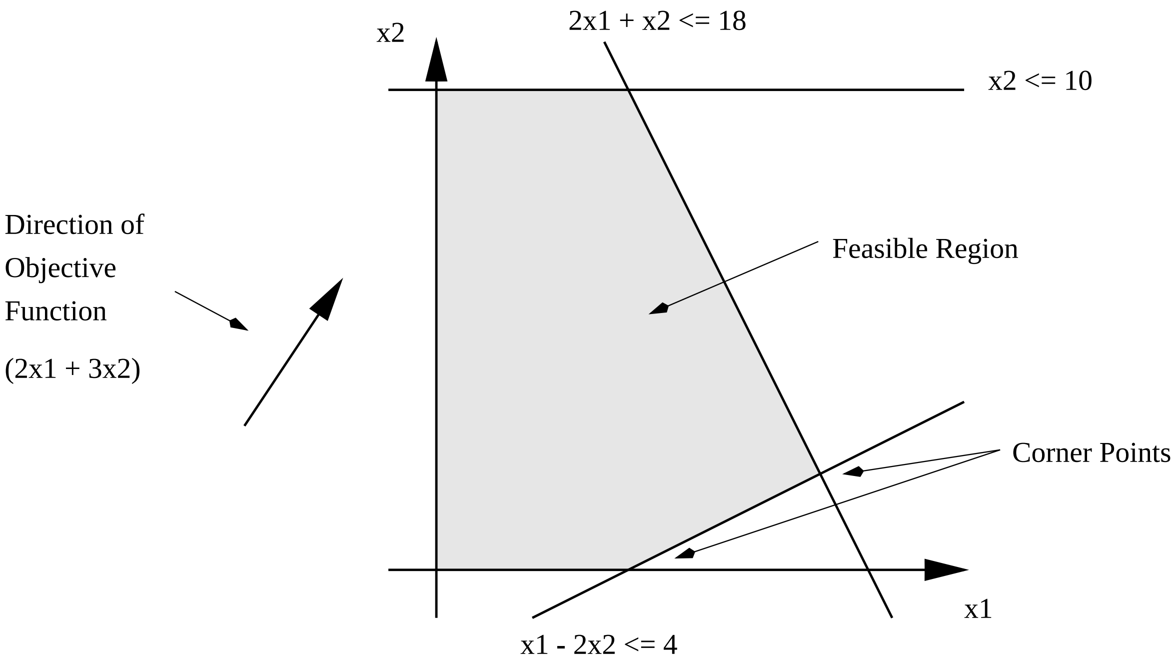


Figure 38: Geometric interpretation of Linear Programming

There are some interesting things to note here. First of all, the maximal point *must* be on a *corner* formed by the intersection of the halfspace boundaries. These corners are sometimes called *extreme points*. Also note that the feasible region could be infinite, allowing the maximum value of the objective function to be infinite, or it might not exist at all, meaning there is no solution.

As discussed earlier, we can add the slack variables x_3, x_4, and x_5, to the first, second, and third constraints, and the new formulation will look as follows.

$$\begin{aligned} &\text{maximize } z = 2x_1 + 3x_2 \\ &\text{subject to } x_1 - 2x_2 + x_3 = 4 \\ &\qquad 2x_1 + x_2 + x_4 = 18 \\ &\qquad\qquad x_2 + x_5 = 10 \\ &\quad x_1, x_2, x_3, x_4, x_5 \geq 0 \end{aligned}$$

Figure 39 shows a geometric interpretation for slack variables. As each variable increases, we move further into the feasible region. When the slack variable is 0, we are actually on the line corresponding to that slack variable.

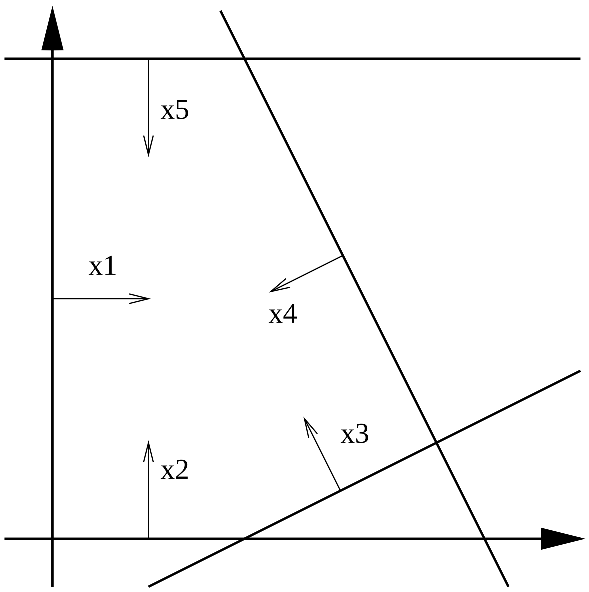


Figure 39: Geometric interpretation of Slack Variables

3 Simplex Method

This is one of the earliest methods for solving linear programs. Its running time is exponential in the worst case, but for many problems it runs much faster than that. Intuitively, the basic algorithm works as follows

1. Find any corner of the feasible region (if one exists).

2. Let's say this corner is at the intersection of n hyperplanes. For each plane, calculate the dot product of the objective function cost vector c with the unit vector normal to the plane (facing inward).

3. If all dot products are negative, then DONE (the problem is already maximized).

4. Else, select the plane with the maximum dot product.

5. The intersection of the remaining $n-1$ hyperplanes forms a line. Move along this line until the next corner is reached.

6. Goto 2.

Following these steps, the algorithm moves from corner to corner until the maximal solution is reached. At each step, it moves away from a plane in such that the rate of change of the objective function is maximized. There are actually different possible ways of deciding which plane to leave and more complicated variants of simplex use different methods. Figure 40 shows how simplex can start at (0,0) and move to the maximal solution. On the first step it chooses the hyperplane (line in 2 dimensions) marked $x2$ to move away from, and on the second step it chooses the hyperplane marked $x1$.

At each step, the variable corresponding to the hyperplane that the current point leaves is called the *entering* variable, and the variable corresponding to the hyperplane that the current point moves to is called the *departing* variable. In the first step of the example in Figure 40 variable $x2$ is the entering variable and variable $x5$ is the departing variable. This sounds counter-intuitive, but it will make more sense as we delve into the details of simplex. In particular the value of an entering variable will change from zero to nonzero, while the value of a departing variable will change from nonzero to zero.

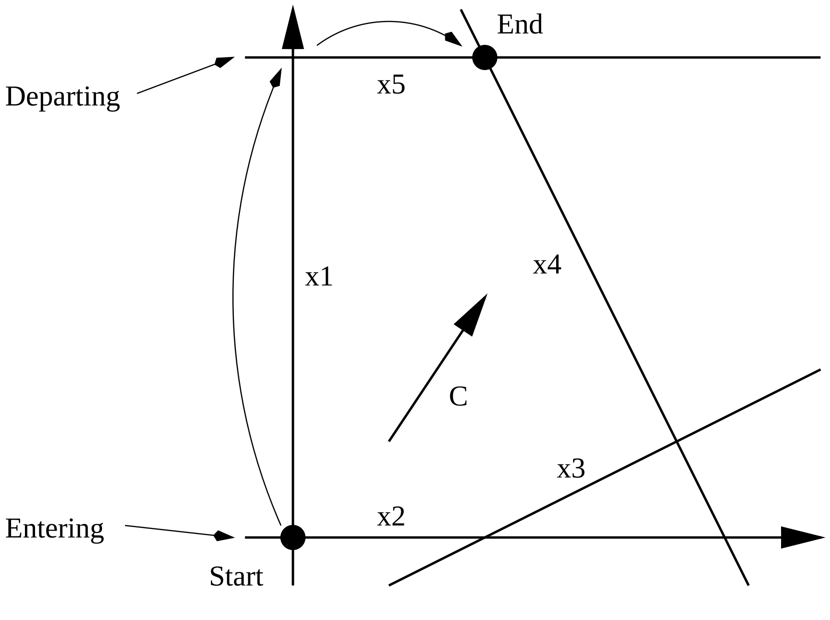


Figure 40: Geometric View of the Simplex Algorithm

Note that this is a geometrical explanation of simplex. The actual algorithm is a little cleverer than this... we don't have to find a unit vector pointing in the direction of each of the outgoing lines at each step.

3.1 Tableau Method

The Tableau Method is a rather elegant organization for the simplex method. Our formulation once again is as follows.

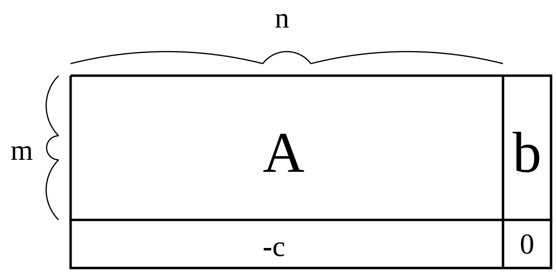


Figure 41: The Tableau

maximize $c^T x$
subject to $Ax = b$, $x \geq 0$

The Tableau for this problem is shown in Figure 41. It is an $(m+1)\times(n+1)$ matrix. The $m+1$ rows correspond to the m constraint equations plus the negative of the cost vector. (Without going into too much detail, it is negative because we are maximizing. If we were minimizing, the vector c would be positive here.) The $n+1$ columns correspond to the n components of the vector x (including original variables and slack variables) and the vector b in the equation $Ax = b$.

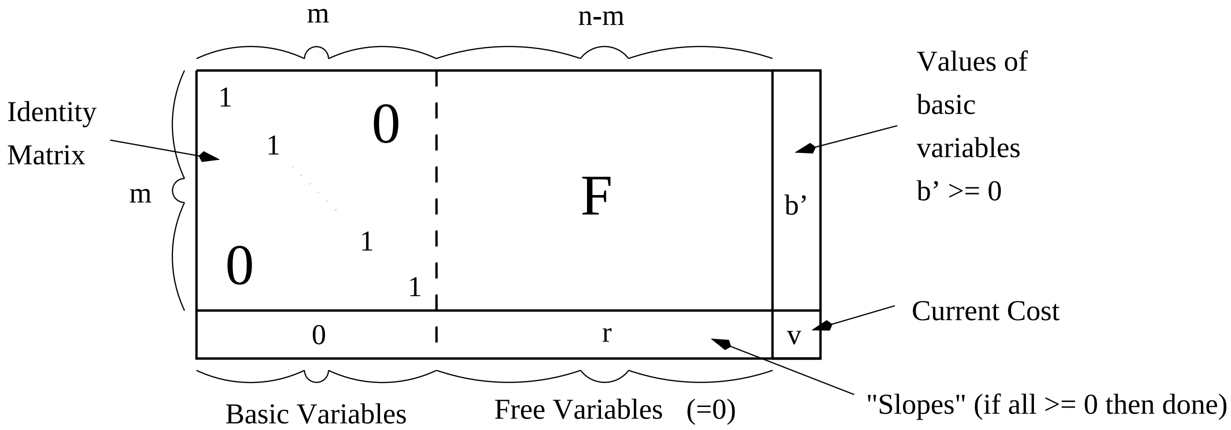


Figure 42: The Tableau after an Initial Point has been Found

The first step of the simplex algorithm is to find a corner (extreme point) of the feasible region. For the moment, let's forget about how we find this point, and look a what this does to the Tableau, as shown in Figure 42.

In a corner, $n-m$ of the variables must have the value 0 (corresponding to the intersection of $n-m$ hyperplanes). One way to see why this happens is to refer to Figure 39. Since, we have 5 variables, and 3 constraints, 2 variables must be 0. Remember that when a variable has a zero value, the current point lies on that hyperplane. Since a corner, in this example, is where two lines meet, two of our variables must be 0. This is an easy way to see what is happening in a tableau. We'll return to this example later.

The $n-m$ variables that are 0 are the *free variables*. The other m variables have positive values, and are called the *basic variables*. If we arrange our tableau so that the m basic variables are on the left, we can use Gaussian elimination on the tableau to put it in the form shown in Figure 42, with ones on the diagonal. When arranged this way, the elements of the vector b' on the right become the values of the basic variables at the current point x (remember, the free variables are all 0). Furthermore, since we are assuming the solution is feasible then all the elements of b' must be nonnegative.

We can also use Gaussian elimination to make components of the cost vector $-c$ corresponding to the basic variables equal to 0. Through the process of elimination, the bottom, right corner of the tableau comes to hold the current cost. Also, the components of the cost vector corresponding to the free variables become the *slopes*, i.e. the rate at which the current cost would decrease if that free variable were increased from 0.

Now we can see how the simplex method proceeds. At each step we are given a tableau like the one in Figure 42. The most negative element of the cost vector tells us which free variable we need to increase from zero. In our two-dimensional example of Figure 39, this tells us which line we need to move away from in order to maximally increase our objective function. The variable x_i which we must increase is called the *entering variable* because it is entering the set of basic variables. Figure 43 (a) illustrates this step.

We have chosen which line to move away from, but we have not chosen which line to move to. In other words, we have to choose a *departing variable* which will leave the set of basic variables and become zero. Figure 43 (b) shows how this is done. Look at the vector u corresponding to the entering variable chosen above, and look at the vector b' that contains the values of the basic variables at our current point x. For each basic variable x_i, compute the ratio b'_i/u_i. This ratio indicates how much the objective function can increase if we use x_i as the departing variable. In the 2-D example, this tells us how much the function can increase if we move from one line to the other. (Note: it cannot increase more than this, because we would have to leave the feasible region). We must choose the basic variable with the smallest ratio as the departing variable, because we cannot leave the feasible region. Note: this ratio is taken only over the positive components of u, because a negative ratio would indicate that the cost function would *decrease* if we moved toward that line, and we don't want to do that!

Now that the entering variable and the departing variable have been chosen, we simply swap their columns in the tableau, as shown in Figure 43(c), and then perform Gaussian elimination, as shown in Figure 43 (d), to get the tableau back into the state it was at the beginning of the simplex step. We repeat these steps until there are no negative slopes (in which case there is no way to increase the objective function and the maximum has been reached), or until there are no positive ratios b'_i/u_i (in which case the maximum value is infinite - it is difficult to explain why this is true, but it is).

An example is in order. Figure 44 shows how the tableau would look for our earlier example in Figures 38 and 39. Figure 44 (a) shows the initial tableau for this example. Our initial point is (0,0), so it only takes a little rearranging to put the tableau in the state

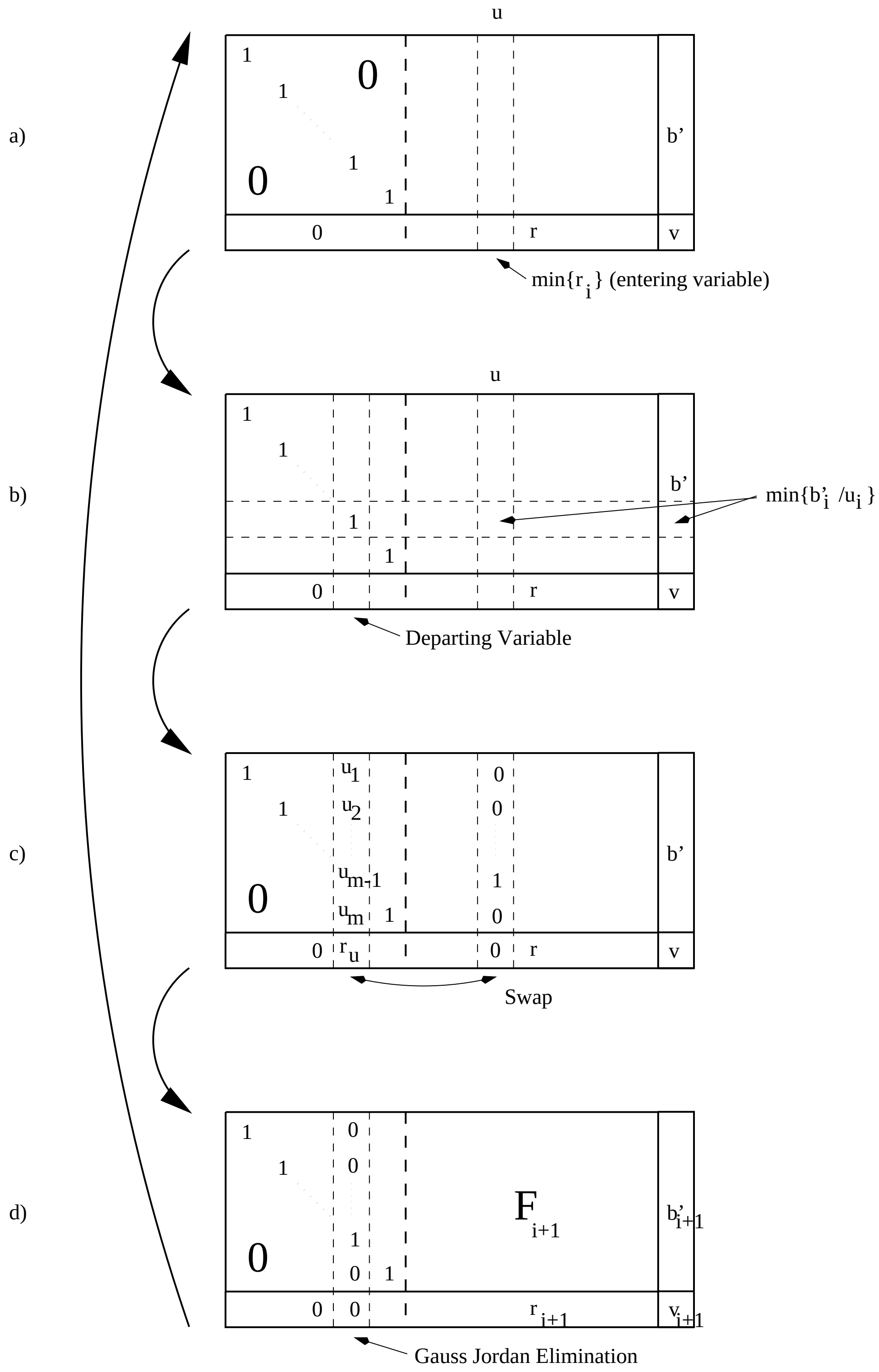


Figure 43: Simplex Steps Performed on a Tableau

x1	x2	x3	x4	x5	
1	-2	1	0	0	4
2	1	0	1	0	18
0	1	0	0	1	10
-2	-3	0	0	0	0

x1-2x2+x3 = 4

2x1+x2+x4 = 18

x2+x5 = 10

2x1+3x2

Initial Tableau
(a)

x3	x4	x5	x1	x2	
1	0	0	1	-2	4
0	1	0	2	1	18
0	0	1	0	1	10
0	0	0	-2	-3	0

x1 = x2 = 0
Starting Point Chosen
(b)

x3	x4	x5	x1	x2 (u)	
1	0	0	1	-2	4
0	1	0	2	1	18
0	0	1	0	1	10
0	0	0	-2	-3	0

Choose Minimum Value
For Maximum Slope
(c)

x3	x4	x5	x1	x2		bi/ui
1	0	0	1	-2	4	-2
0	1	0	2	1	18	18
0	0	1	0	1	10	10
0	0	0	-2	-3	0	

min

swap

Find Minimum Positive Ratio
(d)

x3	x4	x2	x1	x5	
1	0	-2	1	0	4
0	1	1	2	0	18
0	0	1	0	1	10
0	0	-3	-2	0	0

Perform Swap
(e)

x3	x4	x2	x1	x5	
1	0	0	1	2	24
0	1	0	2	-1	8
0	0	1	0	1	10
0	0	0	-2	3	30

Perform Gaussian Elimination
to Complete the Step
(f)

Figure 44: An Example of Simplex Steps on a Tableau

shown in Figure 44 (b). In Figure 44 (c) we find the minimum value of the cost vector, which corresponds to the maximal slope. Figure 44 (d) shows the ratios b'_i/u_i, and which columns we choose to swap. In Figure 44 (e) we see the tableau after the swap, and in Figure 44 (f) we see the tableau after Gaussian elimination.

3.2 Finding an Initial Corner Point

We have not yet discussed how to find the initial corner point needed to start the simplex method. If we were originally given the problem in the form $Ax \leq b, x \geq 0$ and $b \geq 0$, then we can add slack variables and use them as our initial basic variables (*i.e.*, our initial corner is where all variables x are 0 and the slack variables y equal b). However, if we are given the problem in the form $Ax = b$, we will need to find an initial corner. One way to do this is to use the simplex method to solve a slightly modified version of the original problem. Given the tableau in Figure 41, we can extend it further by adding m artificial variables, one for each row (these are analogous to the m slack variables we previously considered). These will form an identity matrix as shown in Figure 45. (Note: the vector b must be positive, but any problem can be put in this form by negating the rows with negative values in b.)

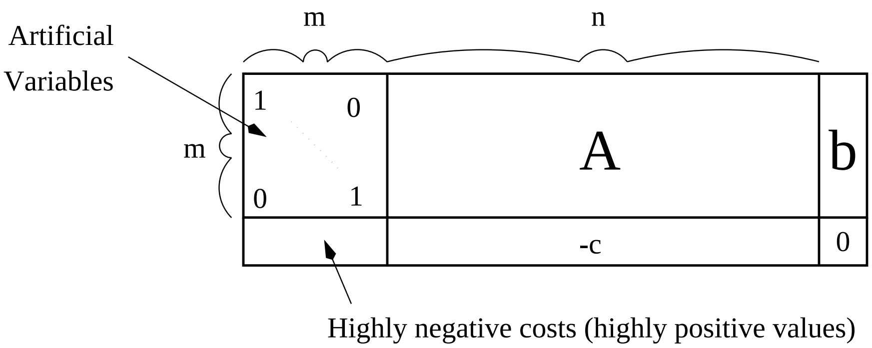


Figure 45: A Modified Tableau that will find a Starting Corner Point

If we give each of these artificial variables a highly negative cost (which corresponds to high positive values in the tableau because it holds $-c$). We can solve this with the simplex method as if it were a normal tableau. The artificial variables will be the set of basic variables in the first step (again, we can make the cost vector 0 below them by Gaussian elimination) and the other variables will be free.

Our vector x will quickly move away from the artificial variables because the algorithm maximizes the objective function (cost) and these variables have high negative values. Soon, the artificial variables will be free variables, and some other set of m variables will be the basic variables. When this happens, we can remove the columns corresponding to the artificial variables, and we will have a tableau that we can use to start solving the original problem. On the other hand, if this process ends and there are still artificial variables in the basis, then there is no feasible region for the original problem.

3.3 Another View of the Tableau

Figure 46 shows another way to look at the tableau. Once we have moved the basic variables to the left and the free variables to the right, we end up with the matrix shown in Figure 46 a. Gaussian elimination turns the matrix B into the identity matrix, which is equivalent to left-multiplying the top three blocks by B^{-1}. Elimination of c_b on the bottom row is equivalent to subtracting c_b times the top of the matrix from the bottom. This is shown in Figure 46 b.

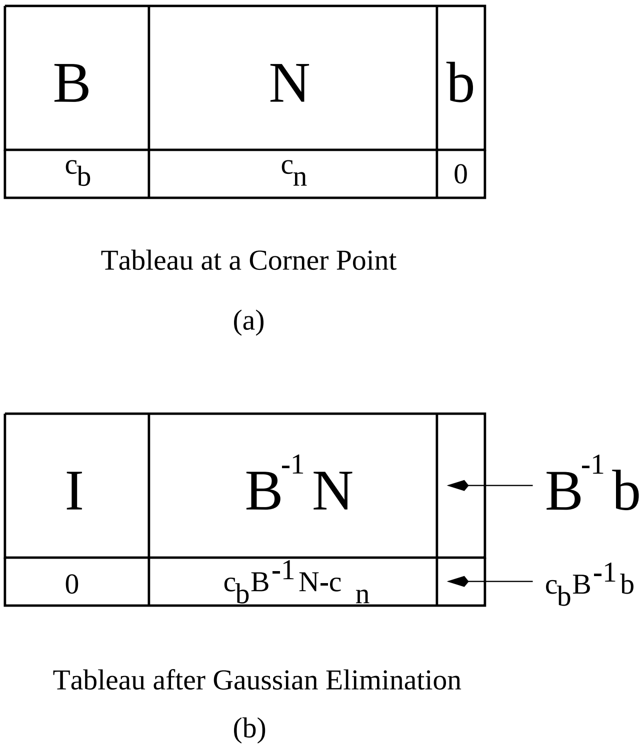


Figure 46: Another way to look at a Tableau

Note that the tableau is likely to be very dense. Even when the original formulation of the problem is very sparse the inverse B^{-1} will typically be dense.

3.4 Improvements to Simplex

The density of the tableau, mentioned in the previous section, is a major problem. Typically the matrix A starts out very sparse—it is not unusual that 1 out of every 10,000 elements that are nonzero. Since B^{-1} will be dense, however, algorithms cannot afford to calculate it directly. For this reason, it is common to keep the matrix B^{-1} in the factored form...

$$U_1^{-1}U_2^{-1}\cdots U_m^{-1}L_m^{-1}\cdots L_1^{-1}$$

...which can be much sparser than B^{-1}. The trick (and major improvement over the last decade) is updating this factorization in place when exchanging variables.

Here are some other improvements.

- Improved pricing (i.e. don't always go down the maximum slope). One example of this is *normalized pricing* in which prices are weighted by the length of the path to the departing variable.

- Better factoring routines (for initial factorization).

- "Perturbation Method" for dealing with stalling. The Simplex method can stall when it hits a degeneracy (i.e. a place where more than m lines meet in a corner in the 2-D case - one of them is redundant) and will sometimes stall by leaving a redundant line, only to return to the same corner.

- Various hacks in how to store the sparse matrices (*e.g.* store both in column and row major order).

- Better Processing.

Speeds have improved by 3-100 times (depending on model) due to algorithmic improvements over the last 10 years.

Notes by Steven Czerwinski

- Duality
- Ellipsoid Algorithm
- Interior Point Methods
    1. Affine scaling
    2. Potential reduction
    3. Central trajectory

1 Duality

Duality refers to the property that linear programming problems can be stated both as minimization and maximization problems. For any problem, there is a *primal* and a *dual*. The primal usually refers to the most natural way we describe the original problem. This can either be a minimization or maximization problem, depending on the specifics. The dual represents an alternative way to specify the original problem, such that it is a minimization problem if the primal is a maximization and vice versa. Solving the dual is equivalent to solving the original problem.

For example, suppose we have the following:

Primal problem: Maximize $z = cx$
such that $Ax \leq b$ and $x \geq 0$
(n equations, m variables)

Then, for our dual, we have: (*notice the change in m and n*)

Dual problem: Minimize $z = yb$
such that $yA \geq c$ and $y \geq 0$
(m equations, n variables)

So why do we care about the dual for a problem? One reason is that we often find that the dual is easier to solve. The dual can also be used for *sensitivity analysis* which determines how much the solution would change with changes in c, A or b. Finally, it can be used in conjunction with the primal problem to help solve some of the interior point methods.

1.1 Duality: General Case

In converting maximization problems to minimization problems and vice versa, we find that every variable in the primal produces a constraint in the dual. Likewise, every constraint in the primal produces a variable in the dual.

Maximization Problem		**Minimization Problem**
Constraints		Variables
$\leq$	$\Leftrightarrow$	≥ 0
$\geq$	$\Leftrightarrow$	≤ 0
$=$	$\Leftrightarrow$	unrestricted
Variables		Constraints
≥ 0	$\Leftrightarrow$	$\geq$
≤ 0	$\Leftrightarrow$	$\leq$
unrestricted	$\Leftrightarrow$	$=$

Table 11: Rules for converting between Primals and Duals

We also find there is correspondence between the variable's restrictions in the primal, with the constraint's equality sign in the dual. For example, if a variable must be greater than 0 in a minimization problem, then the constraint the variable produces in the dual will be stated as a "less than" relationship. Similarly, there is a correspondence between the constraints' equality signs in the primal to the restrictions on the variables in the dual. Table 11 summaries these correspondences.

1.2 Example

We are given the following maximization problem:

$$\text{maximize:} \quad 4x_1 + 2x_2 - x_3$$

$$\begin{aligned} \text{s.t.} \quad & x_1 + x_2 + x_3 = 20 \\ & 2x_1 - x_2 \geq 6 \\ & 3x_1 + 2x_2 + x_3 \leq 40 \\ & x_1, x_2 \geq 0 \end{aligned}$$

From this description, we easily determine the following:

$$\begin{aligned} x &= \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \\ c &= \begin{bmatrix} 4 & 2 & -1 \end{bmatrix} \\ b &= \begin{bmatrix} 20 \\ 6 \\ 40 \end{bmatrix} \\ A &= \begin{bmatrix} 1 & 1 & 1 \\ 2 & -1 & 0 \\ 3 & 2 & 1 \end{bmatrix} \end{aligned}$$

We can now produce the dual:

$$\begin{aligned}
\text{minimize:}\quad & 20Y_1 + 6Y_2 - 40Y_3 \\
\text{s.t.}\quad & Y_1 + 2Y_2 + 3Y_3 \geq 4 \\
& Y_1 - Y_2 + 2Y_3 \geq 2 \\
& Y_1 + Y_3 = -1 \\
& Y_2 \leq 0, Y_3 \geq 0
\end{aligned}$$

The coefficients come from transposing A. To determine the equality/inequality type we use the rules in Table 11. For example the first inequality $Y_1 + 2Y_2 + 3Y_3 \geq 4$ corresponds to variable x_1 in the primal. Since $x_1 \geq 0$ in the primal, the inequality type is also $\geq$ in the dual.

Notice, since the first constraint is produce by multiplying A's first column vector with Y, it corresponds to the x_1 variable in the primal (because the first column in A contains all of x_1's coefficients). Therefore, the first constraint in the dual must have an "equal" relationship since x_1 is unrestricted in the primal.

1.3 Duality Theorem

The Duality theorem states:

If $\overline{x}$ is feasible for **P** (given below) and $\overline{y}$ is feasible for **D** then: $c\overline{x} \leq \overline{y}b$ and at optimality, $c\overline{x} = \overline{y}b$. We call $\overline{y}b - c\overline{x}$ the *Duality Gap*. The magnitude of the duality gap is a good way to determine how close we are to an optimal solution and is used in some of the interior point methods.

$$\begin{aligned}
\mathbf{P}\text{: Max}\quad & z = cx \\
\text{s.t.}\quad & Ax \leq b \\
& x \geq 0
\end{aligned}$$

$$\begin{aligned}
\mathbf{D}\text{: Min}\quad & Z = yb \\
\text{s.t.}\quad & yA \geq c \\
& y \geq 0
\end{aligned}$$

2 Ellipsoid Algorithm

Created by Khachian in '79, the ellipsoid algorithm was the first known polynomial-time algorithm for linear programming. (There were other poly-time algorithms around, but they had not been proven to be so.) The ellipsoid algorithm does not solve linear programming problems of the general form, but rather it solves any problem stated in the following way:

Find x subject to $Ax \leq b$ (i.e. find any feasible solution).

However, we will show that any linear programming problem can be reduced to this form.

If L equals the number of bits used to represent A and b, then the running time of the algorithm is bound by $O(n^4L)$. Unfortunately the algorithm takes this much time for most problems (it is very unusual that it finds a solution any faster). Simplex's worse case running time is larger than $O(n^4L)$, but for most problems it does much better. Therefore Simplex

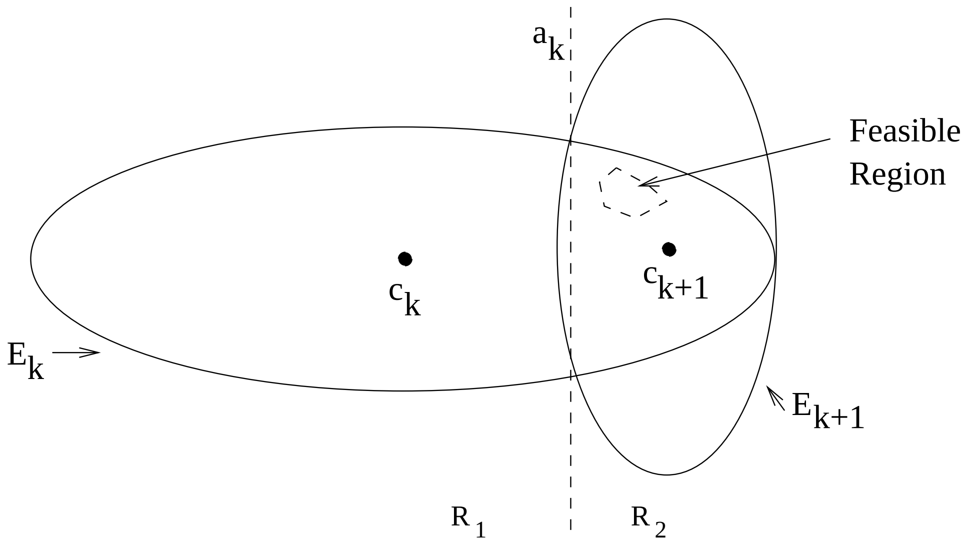


Figure 47: An iteration in the Ellipsoid Algorithm

ends up being better than the Ellipsoid algorithm in practice for most problems even though its worst-case running time is worse.

2.1 Description of algorithm

The idea behind the ellipsoid algorithm is that we start with some ellipsoid E_0 which encloses an area in the solution space, and is guaranteed to contain the feasible region of the solution. Basically, at each iteration of the algorithm, we create new ellipsoids E_k that are smaller than the previous, but still are guaranteed to contain the feasible region. Eventually, we will find a point that satisfies the problem (recall that we are satisfied with any feasible point).

Specifically, for each iteration k of the algorithm we do the following:

1. Determine c_k = the center of E_k.
2. Determine a_k = the constraint most violated by c_k.
3. Use the a_k hyperplane to split E_k into two regions. One of these regions R_1 contains c_k, and the other R_2 contains the feasible region.
4. We then construct E_{k+1} such that it contains all points in $E_k \cap R_2$, but is smaller than E_k.

Figure 47 illustrates these steps.

For n variables Khachian showed the following property of this algorithm:

$$\frac{Volume(E_{k+1})}{Volume(E_k)} = \frac{1}{2^{\frac{1}{2n+1}}}$$

2.2 Reduction from general case

As stated, we can convert the general form of linear programming problems to a form suitable to use the ellipsoid algorithm on.

Suppose we have the following problem stated in the general form: Maximize cx subject to $Ax \leq b$, and $x \geq 0$. We can convert it to the form $A'x' \leq b'$ as shown below:

Find any x, y such that

$$\begin{aligned} Ax &\leq b && (3) \\ -x &\leq 0 && (4) \\ yA &\leq -c && (5) \\ -y &\leq 0 && (6) \\ -cx + by &\leq 0 && (7) \end{aligned}$$

Here x and y together form x', the combination of the left sides of the equations form A', and the combination of the right sides of equations form b'. Equations (3) and (4) come from the original problem, and equations (5) and (6) come from its dual. Equation (7) will only be true at a single point when the original problem is optimized (the duality gap must be non-negative). In this way, if any solution to $A'x' \leq b'$ is found, then it will be the optimal solution for the original problem.

3 Interior Point Methods

Interior point methods are considered the first practical polynomial time methods for linear programming. Unlike simplex, the interior point methods do not touch any boundaries until the end when they find the optimal solution. Instead, these methods work by traveling through the interior of the feasible region (see Figure 48) by using a combination of the following two terms:

Optimizing term This moves the algorithm towards the objective.

Centering term This keeps it away from the boundaries.

Generally, interior point methods take many fewer steps than the simplex method (often less than 100), but each step is considerably more expensive. The overall relative cost of simplex and interior point methods is very problem specific and is effected by properties beyond just the size of the problem. For example the times depend strongly on the matrix structure.

Interior point methods have been used since the 50s for nonlinear programming. Roughly categorized, the methods include the following:

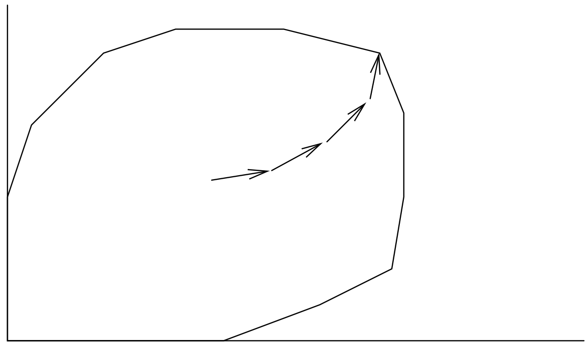


Figure 48: Walking on the interior points

	Fuel	**Continent**	**Car**	**Initial**
Size	13x31k	9x57k	43x107k	19x12k
Non-zeros	186k	189k	183k	80k
Iterations	66	64	53	58
Cholesky non-zeros	1.2M	.3M	.2M	6.7M
Time	2364s	771s	645s	9252s

Table 12: Performance of the central trajectory method on 4 benchmarks

Name	**Number of iterations**
Affine scaling	no known time bounds
Potential reduction	$O(nL)$
Central trajectory	$O(\sqrt{x}L)$

Karmarkar proved polynomial time bounds for his method (potential reduction) in 1984. Table 12 shows some example running times for a central trajectory method (Lustig, Marsten, and Shanno 94) on four different benchmark problems. As the table shows, the time spent in each iteration is greatly influenced by the number of Cholesky non-zeros present in the matrix.

In the following discussion we will assume we are trying to solve a linear program of the form

$$\begin{array}{rl} \text{minimize:} & cx \\ \text{subject to:} & Ax = b \\ & x \geq 0 \end{array}$$

3.1 Centering

There are several different techniques to compute the centering term for interior point methods. This section describes a few of those techniques.

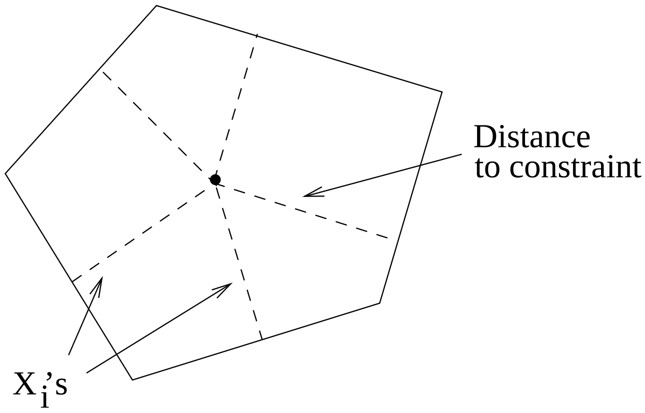


Figure 49: The analytical center

3.1.1 Analytical center

We can use analytical centering by minimizing

$$\sum_{i=1}^{m} \lg x_i$$

If $x_i = 0$ are the boundaries of the feasible region (we assume they are), then x_i are the distances to the boundaries from our current point. See Figure 49.

3.1.2 Elliptical Scaling

In elliptical scaling, we start with a small ellipsoid centered around the current point. We then expand the ellipsoid in each dimension of the problem. The expansion in a dimension stops when the ellipsoid touches a constraint. Since our inequality constraints are of the form $x \geq 0$, the equation will be of the form:

$$\frac{x_1^2}{c_1^2} + \frac{x_2^2}{c_2^2} = 1$$

The ellipsoid will then be a measure of how uncentered the point is and can be used to scale space to bias our selection of a direction toward the longer dimensions of the ellipsoid, and thus away from constraints which are close by (the shorter dimensions). See Figure 50. We will see how this is used in the affine scaling method.

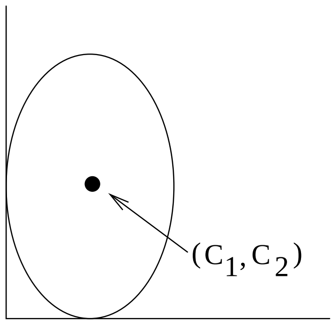


Figure 50: Elliptical Scaling

3.2 Optimizing techniques

3.2.1 Projection

If our problem is constrained to $Ax = b$, then we know that our feasible region is contained in a hyperplane. If A has n rows and m columns, then it is contained in a $m - n$ dimensional hyperplane in m dimensional space. (Assuming the rows in A are independent.) The hyperplane basis is called the *null space* of A We can also view A as being the "slopes" for the hyperplane, while b represents the "offsets".

We can use this to find the optimal solution for the problem. If $\vec{c}$ is the direction of the optimal solution, then we should take a step in the direction of $\vec{d}$. See Figure 51. We must do this because $\vec{c}$ would take us off the hyperplane and therefore, out of the feasible region. Instead, we project c into the hyperplane, and step in that direction.

Here's how we compute $\vec{d}$:

$$\begin{aligned}
\vec{d} &= \vec{c} - \vec{n} \\
&= c - A^{\mathrm{T}}(AA^{\mathrm{T}})^{-1}Ac \\
&= (I - A^{\mathrm{T}}(AA^{\mathrm{T}})^{-1}A)c \\
&= P \cdot c
\end{aligned}$$

where $P = (I - A^{\mathrm{T}}(AA^{\mathrm{T}})^{-1}A)$, the projection matrix.

Calculating $P \cdot c$ (the projection) must be done at each iteration of the method since, as we will see, A will change. So, solving for the projection matrix is a major computational concern.

We find that it is more efficient to solve $AA^{\mathrm{T}}w = Ac$ for w. We can then use w to calculate the projection because of the following relationship:

$$Pc = (I - A^{\mathrm{T}}(AA^{\mathrm{T}})^{-1}A)c = c - A^{\mathrm{T}}w$$

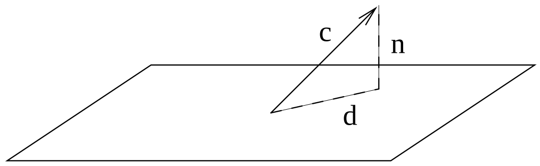


Figure 51: Projection into the null space

Computing $AA^{\mathrm{T}}w = Ac$ is more efficient because it can be solved using a sparse solver (e.g. Cholesky Factorization). Otherwise, calculating P directly can generate a dense matrix, which is problematic both in terms of runtime and in the space it would require.

3.2.2 Newton's Method

We can use Newton's Method to help determine the direction in which we should travel in the interior of the feasible region to optimize the solution. Often, the function we are trying to optimize will not be linear in x, and could be composed of transcendental functions such as $\lg or \ln$, as in the central trajectory case. We can approximate the minimum of these functions using a Newton's method, and more easily determine the direction of the optimal solution. We can see how this is done through a little math.

First, we will just look at the general case of minimizing some 1-dimensional function $f(x)$. We can write $f(x)$ as a Taylor expansion around some point x_0.

$$f(x) \approx f(x_0) + f'(x_0)(x - x_0) + \frac{1}{2}f''(x_0)(x - x_0)^2$$

To get the minimum of the function, we take the derivative and set it to 0, which gives:

$$\begin{aligned} 0 &= f'(x_0) + f''(x_0)(x - x_0) \\ x &= x_0 - \frac{f'(x_0)}{f''(x_0)} \end{aligned}$$

In multiple dimensions we can write this in matrix form as

$$x - x_c = \vec{d} = (F(x_c))^{-1}\nabla f(x_c)^{\mathrm{T}}$$

where $f(x)$ is now a function of n variables, x_c is the point we are expanding about, and $F(x_c) = \nabla \times \nabla f(x)$ evaluated at x_c, an $n \times n$ matrix which is called the Hessian matrix, i.e.

$$F(x) = \begin{bmatrix} \frac{\partial f}{\partial x_1 x_1} & \frac{\partial f}{\partial x_1 x_2} & \cdots & \frac{\partial f}{\partial x_1 x_n} \\ \frac{\partial f}{\partial x_2 x_1} & \frac{\partial f}{\partial x_2 x_2} & \cdots & \frac{\partial f}{\partial x_2 x_n} \\ & & \cdots & \\ \frac{\partial f}{\partial x_n x_1} & \frac{\partial f}{\partial x_n x_2} & \cdots & \frac{\partial f}{\partial x_n x_n} \end{bmatrix}$$

$\nabla f(x_c)$ is a vector of the first derivatives of $f(x)$ evaluated at x_c.

If we constrain the newton step to be on a hyperplane (i.e. $Ax = b$), it is called a *constrained Newton step*. In general we first pick a direction $\vec{d}$ based on a Newton step (or other method) and then project the step onto the constraining hyperplane, as discussed in the previous section.

3.3 Affine scaling method

The affine scaling method is basically a biased steepest descent search (biased away from the boundaries of the feasible region). Suppose we had the following problem to solve:

$$\begin{aligned} \text{minimize: } & cx \\ \text{s.t. } & Ax = b, \\ & x \geq 0 \end{aligned}$$

To solve this using the affine scaling method, at each iteration we solve:

$$\begin{aligned} \text{minimize: } & cz \\ \text{s.t. } & Az = 0, \\ & zD^{-2}z \leq 1 \end{aligned}$$

where D is a matrix whose only non-zero terms are along its main diagonal. The terms on this diagonal are the coordinates of the current point: $x_1, x_2, x_3, \cdots$. The equation $zD^{-2}z \leq 1$ is the definition for the interior of an ellipsoid, as described earlier (see Figure 50).

We update our x at each iteration by setting $x_{k+1} = x_k + \alpha * z$, where z is the direction we found in the above minimization problem, and α is selected to bring us close to the boundary, but not all the way there.

In this process, the cz minimization acts as the optimization term, while the $zD^{-2}z \leq 1$ acts as the centering term. In order to understand how this works, we should realize $Ax = b$ is a slice of the ellipsoid. In fact, it is the same hyperplane that we described in the ellipsoid scaling method. In order to minimize the function, we should step in the c' direction, where c' is the projection of c into the hyperplane. However, the affine scaling method steps in the direction of z.

How do we get z? We construct hyperplanes perpendicular to c, and keep moving them back until we find one such hyperplane that intersects the ellipsoid only once. We then construct z to be the difference between the current point and the intersection point. By stepping in the direction of z, we are centering the point because z will tend to point to constraints that are far away. In fact, as the current point comes closer and closer to a constrain, z will point further and further away from that constraint.

See Figure 52 for a picture of this construction of z, c', and the hyperplanes.

3.3.1 Computation at each iteration

In order to ease the computation of z, we make a substitution of variables, by setting $z = Dz'$. Then, we have

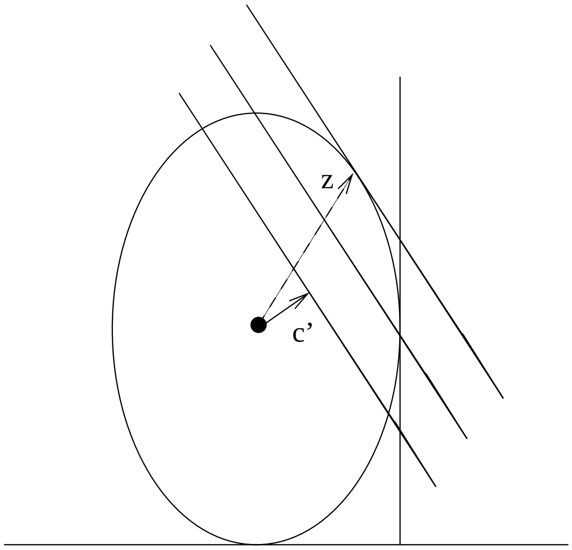


Figure 52: An iteration of the Affine Scaling Method

$$\begin{array}{rl} \text{minimize:} & cDz' \\ \text{s.t.} & ADz' = 0, \\ & z'^{\mathsf{T}} D^{\mathsf{T}} D^{-2} D z' \leq 1 \end{array}$$

The last equation simplifies to $zz \leq 1$, which is just a sphere. With this in mind, we can say our direction z' is unbiased and throw out the $zz \leq 1$ constraint. To find the minimum we now just take a steepest descent step on cDz' that keeps us within the hyperplane defined by $ADz' = 0$. This leads us to,

$$z' = -\underbrace{(I - B^{\mathsf{T}}(BB^{\mathsf{T}})^{-1}B) \cdot cD}_{\texttt{projection}}$$

where $B = AD$. To get z in our original space we just substitute back $z = Dz'$. The vector z' is negative since we are trying to find a minimum and therefore descending.

3.3.2 Overall computation

Now that we know what we have to compute during each iteration, let's step back and see how the entire process works. In the following algorithm we will assume we find the projection by using Cholesky factorization to solve the linear system, as described in Section 3.2.1. The linear equation could also be solved using other techniques, such as iterative solvers.

1. Find a starting feasible interior point x_0
2. Symbolically factor AA^{T}. Since the non-zero structure of BB^{T} on each step is going to be the same as AA^{T}, we can precompute the order of the operations in the Cholesky

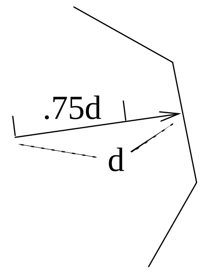


Figure 53: Picking an alpha

factorization, along with its non-zero structure. Since we are only doing this once, we can potentially spend some time trying to minimize the "fill" (the number of additional non-zero's that appear in the Cholesky factorization that did not appear in AA^{T}).

3. Repeat until done

   - $B = AD_i$, where D_i is a diagonal matrix with the x_i along the diagonal.
   - Solve for w in $BB^{\mathsf{T}}w = BDc$ by multiplying out BDc and factoring BB^{T} with help from the precomputed symbolic factorization.
   - $z' = -(Dc - B^{\mathsf{T}}w)$
   - $z = D_i z'$ (substituting back to the original space)
   - $x_{i+1} = x_i + \alpha z$

As mentioned, we should pick α so that the step is taken some fraction of the way to the nearest boundary. See Figure 53. In practice α is selected so that it goes .96 of the way to the boundary.

3.4 Potential reduction

In the potential reduction method, we reword the minimization problem to be

$$\begin{array}{rl} \text{minimize:} & q\ln(cx - by) - \sum_{j=1}^{n} \ln(x_j) \\ \text{s.t.} & Ax = b,\ x \geq 0 \\ & yA + s = 0,\ s \geq 0 \end{array}$$

As we can see, this method makes use of the problem's dual. The minimization criteria is partly based on the duality gap $(cx - by)$. Also, the s variables are the slack variables.

The first quantity in the minimization $(q\ln(cx - by))$ represents the optimizing term. It will go to $-\infty$ at the optimal solution. Therefore, this term will dominate near the optimal solution.

$\sum_{j=1}^{n} \ln(x_j)$ is the analytical center term.

In order to optimize the solution at each iteration, we can either do a constrained Newton step, or a transformation with a steepest descent. This second method is what Karmarkar used in his version of the potential reduction method.

There are several advantages of this method over the affine scaling method:

- There are guaranteed bounds on the running time.
- The duality gap used in the minimization allows the user to specify an error tolerance for the solution.
- It is more efficient in practice.

3.5 Central trajectory

The central trajectory is the method of choice these days. It falls under the category of Log Barrier methods, which date back to the 50s where they were used to solve nonlinear problems.

To perform central trajectory, at each iteration step k, we do the following:

1. $$\begin{array}{rl} \text{minimize:} & cx - \mu_k \sum_{j=1}^{n} \ln(x_j) \\ \text{s.t.} & Ax = b, x > 0 \end{array}$$

   (approximate using a constrained Newton step)

2. Select $\mu_{k+1} \leq \mu_k$

This will terminate when μ_k approaches 0.

It turns out that with the "appropriate" schedule of μ_k, the central trajectory approach reduces to the potential reduction approach. We can therefore view it as a generalization.

Currently the primal-dual version of the central trajectory approach using "higher order" approximations (rather than Newton's) seems to be the best interior-point algorithm in practice.

4 Algorithm summary

1. Overall, the current state of the art uses very sophisticated algorithms to solve linear programming problems in practice. Even the linear algebra packages used to perform the computations use complicated math in order to optimize computation time.
2. Practice matches theory reasonably well.
3. We find interior-point methods dominant when:
   - Large n
   - Small Cholesky factors
   - The problem is highly degenerate, meaning the many of the constraints cross each other.

4. Ellipsoid algorithms are not currently practical, since they always take the worse case time.
5. Large linear programming problems take hours/days to solve. Parallelism is very important to speed up the optimization.
6. There are many interior-point methods, but they all share the same intuition.

We looked at a slide that showed the running times of large models. For some interior point methods, $AA^ty = Ac$ was too dense, so the programs ran out of memory and couldn't complete their tasks.

Some algorithms use interior point methods to get close to a solution, then snap to a boundary and use simplex methods to finish off the answer.

5 General Observation

Mathematics has become increasingly important for "Real World" algorithms. Just over the last 10 years, the complexity of math employed in algorithms has increased dramatically.

- Compression
  1. Information theory
  2. Markov models
  3. Transforms (for lossy compression)
- Cryptography
  1. Group theory
  2. Number theory
- Linear Programming
  1. Matrix algebra
  2. Optimization
  3. Graph theory

Notes by Andrew Begel

- History
- Knapsack Problem
- Traveling Salesman Problem
- Algorithms
  1. Linear Program as Approximation
  2. Branch and Bound (integer ranges)
  3. Implicit (0-1 variables)
  4. Cutting Planes (*postponed*)

1 Introduction and History

Linear integer programming (often just called integer programming, or IP) is a linear program with the additional constraint that all variables must be integers, for example:

$$\begin{array}{rl} \text{maximize} & cx \\ \text{subject to} & Ax \leq b \\ & x \geq 0 \\ & x \in \mathcal{Z}^n \end{array}$$

There are several problems related to integer programming:

Mixed integer programming (MIP): Some of the variables are constrained to be integers and others are not.

zero-one programming: All the variables are constrained to be binary (0 or 1).

Integer quadratic programming: The cost function can be quadratic.

Integer programming was introduced by Danzig in 1951. In 1954, he showed that TSP (traveling salesman problem) was a special case of integer programming. Gomory found the first convergent algorithm in 1958. In 1960, Land and Doig created the first general branch and bound techniques for solving integer programming problems.

Integer and mixed integer programming has become "dominant" over linear programming in the past decade. It saves many industries (e.g. airline, trucking) more than one billion dollars a year. The state of the art in algorithms is branch-and-bound + cutting plane + separation (solving sub-problems). General purpose packages don't tend to work as well with integer programming as they do with linear programming. Most techniques employ some form of problem-specific knowledge.

2 Applications

IP and MIP can be applied to a number of different problem areas:

- Facility location – Where should we locate a warehouse or a Burger King to minimize shipping costs?
- Set covering and Set partitioning – Scheduling crews for Delta
- Multicomodity distribution – Routing deliveries for Safeway
- Capital budgeting – Developing a budget for Microsoft
- Other applications – VLSI layout, clustering, etc.

Here we will describe how various problems can be expressed as integer programs.

2.1 Knapsack Problem

It looks trivial, but it's NP-complete.

$$\begin{array}{ll} \text{Maximize} & cx \\ \text{subject to} & ax \le b, \\ & x \text{ binary} \end{array}$$

where:

$$\begin{aligned} b &= \text{maximum weight} \\ c_i &= \text{utility of item } i \\ a_i &= \text{weight of item } i \\ x_i &= \begin{cases} 1, & \text{if item } i \text{ is selected} \\ 0, & \text{otherwise} \end{cases} \end{aligned}$$

2.2 Traveling Salesman Problem

$$\text{We want to minimize} \quad \sum_{i=1}^{n}\sum_{j=1}^{n} c_{i,j}x_{i,j}$$

$$\begin{aligned} &\text{subject to} \\ \sum_{j=0}^{n} x_{i,j} &= 1, \qquad \text{for } i = 1,\ldots,n \\ \sum_{i=0}^{n} x_{i,j} &= 1, \qquad \text{for } j = 1,\ldots,n \\ t_i - t_j + nx_{i,j} &\le n-1, \qquad \text{for } i,j = 2,\ldots,n \end{aligned}$$

where:

$$
\begin{aligned}
c_{i,j} &= \text{distance from city } i \text{ to city } j \\
x_{i,j} &= \begin{cases} 1, & \text{if city } j \text{ visited immediately after city } i \\ 0, & \text{otherwise} \end{cases} \\
t_i &= \text{arbitrary real numbers we need to solve for,} \\
&\quad\ \text{but we don't really care about their values}
\end{aligned}
$$

The first set of n constraint restricts the number of times the solution enters each city to 1. The second set of n constraint restricts the number of times the solution leaves each city to 1. The last set of constraints are used to eliminate subtours. In particular the constraints make sure that any tour includes city 1, which in conjunction with the first and second set of constraints forces a single tour.

To see why the last set of constraints forces any tour to include city 1, lets consider a tour that does not include the city and show a contradiction. Lets say the tour visits the sequence of cities $v_1, v_2, v_3, \ldots, v_l$ none of which is city 1. Lets add up the equations taken from the last set of constraints for each pair of cities that are visited one after the other in the tour. Since it is a tour, each t_{v_i} will appear once positively and once negatively in this sum, and will therefore cancel. This leaves us with $nl \leq (n-1)l$, which is clearly a contradiction.

2.3 Other Constraints Expressible with Integer Programming

Logical constraints (x_1, x_2 binary):

$$
\begin{aligned}
\text{Either } x_1 \text{ or } x_2 &\Rightarrow x_1 + x_2 = 1 \\
\text{If } x_1 \text{ then } x_2 &\Rightarrow x_1 - x_2 \leq 0
\end{aligned}
$$

Integer constraints can also be applied to combine constraints:

$$
\text{Either or} \quad \begin{array}{lcl} a_1 x & \leq & b_1 \\ a_2 x & \leq & b_2 \end{array} \quad \Rightarrow \quad \begin{array}{lcl} a_1 x - My & \leq & b_1 \\ a_2 x - M(1-y) & \leq & b_2 \\ y \text{ binary} & & \end{array}
$$

The M is a scalar and needs to be "large" (x, a_1, and a_2 are vectors). This works since if $y = 1$ 1 then the first equation is always true and the second needs to be satisfied, while if $y = 0$ then the second is always true and the first needs to be satisfied. Since y is constrained to be binary, one needs to be satisfied.

A similar approach can be used to handle m out of n constraints.

2.4 Piecewise Linear Functions

Consider a minimization problem with the following objective function

$$
c(x) = \begin{cases} 0 & x = 0 \\ k + cx & x \geq 0 \end{cases}
$$

and assume that $x \leq M$. This can be expressed as the following integer program:

$$
\begin{array}{ll}
\text{minimize} & ky + cx \\
\text{such that} & x_i \leq My \\
& x_i \geq 0 \\
& y \in \{0, 1\}
\end{array}
$$

The first inequality forces y to be 1 if x is anything other than 0. This basic approach can be generalized to arbitrary piecewise linear functions.

3 Algorithms

3.1 Linear Programming Approximations

The LP solution is an upper bound on the IP solution. If the LP is infeasible, then the IP is infeasible. If the LP solution is integral (all variables have integer values), then it is the IP solution, too.

Some LP problems (e.g. transportation problem, assignment problem, min-cost network flow) will always have integer solutions. These can be classified as problems with a unimodular matrix A. (Unimodular $\Rightarrow detA = 0, 1$ or -1).

Other LP problems have real solutions that must be rounded to the nearest integer. This can violate constraints, however, as well as being non-optimal. It is good enough if integer values take on large values or the accuracy of constraints is questionable. You can also preprocess problems and add constraints to make the solutions come out closer to integers.

3.2 Search Techniques

We will now describe two search techniques for solving integer programms both based on a general Branch-and-Bound approach. The first technique works for general integer programs and the second for 0-1 programs. For historical reasons, the first is called the Branch and Bound algorithm, even though both algorithms are in fact based on branch and bound. Both algorithms require exponential time in the general case.

3.2.1 Branch and Bound

The branch and bound approach basically builds a tree of possible solutions by splitting at each node of the tree based on a value for one of the variables (*e.g.,* all solutions with $x < 3$ go to the left branch and all solutions with $x \geq 3$ go to the right branch). Such a tree can easily end up having a number of nodes that is exponential in the number of variables. However, by judicially pruning the tree we can hope to greatly reduce the number of nodes that have to be searched. The "bound" refers to the pruning of the tree so that we don't have to search further.

Each time the algorithms branches left or right it adds a constraint (*e.g.,* $x < 3$). The pruning is based on keeping the best previous solution found (lets assume a maximization problem) and then using a linear program to get an upper bound on the solution given the current constraints. If the solution to the linear program is less than the best previous solution then the node does not have to be searched further. There is freedom in the basic

function $IP(C, c, z*)$
  // C is the set of constraints $Ax \leq b$
  // c is the objective vector
  // $z*$ is the cost of the current best solution (initially $-\infty$)
  // Returns the cost of the optimal solution with constraints C
    if better than $z*$, otherwise returns z*

$z, x, f = LP(C, c)$
  // f indicates whether $LP(C, c)$ is feasible or not

if not(f) **return** $z*$ // no feasible solution
if $z \leq z*$ **return** $z*$ // upper bound less than current best solution
if integer(x) **return** z // new best solution found

pick a non integer variable x_i' from x

$C_d = C, x_i \leq \lfloor x_i' \rfloor$
$z* = IP(C_d, c, z*)$

$C_u = C, -x_i \leq -\lceil x_i' \rceil$
$z* = IP(C_u, c, z*)$

return $z*$
end

Figure 54: The Branch and Bound Algorithm for Integer Programming.

algorithm in selecting the next node to expand. The most common approaches are to use either a depth first search (DFS) or best-first search. Depth-first search has the advantage that it more quickly finds an initial solution and leaves less nodes "open" (each of which requires space to store). Best-first search has the advantage that it might find the best solution faster. A compromise is to use beam search by limiting the number of partial solutions that can be left open in a best-first search.

Figure 54 shows a branch-and-bound algorithm that uses DFS and pruning to solve the maximization problem

$$\begin{array}{ll} \text{maximize} & cx \\ \text{subject to} & Ax \leq b,\ x \geq 0,\ x \in \mathcal{Z}^n \end{array}$$

The algorithm passes around the current best solution $z*$ and adds constraints to the constraint set C each time it expands a node by making recursive calls. The algorithm stops expanding a node (bounds or prunes the node) if either (1) the linear program given the current set of constraints is infeasible (2) the solution to the linear program is less than the current best solution (3) the linear program has an integer solution.

```
function IP(C, c, x_f, z*)
    // C is the set of constraints Ax ≤ b
    // c is the objective vector
    // x_f a subset of the variables which are fixed
    // z* is the cost of the current best solution

  x = x_f + 0 (set free variables to zero)
    // this is the best solution ignoring inequalities since c ≥ 0
    // it is not necessarily feasible

  if cx ≥ z* return z*          // lower bound greater than current best solution
  if feasible(C, x) return cx          // new best feasible solution found
  if no feasible completion to x_f return z*

  pick a non fixed variable x_i from x

  x_f0 = x_f ∪ {x_i = 0}
  z* = IP(A, c, x_f0, z*)

  x_f1 = x_f ∪ {x_i = 1}
  z* = IP(A, c, x_f1, z*)

  return z*
end
```

Figure 55: The Implicit Enumeration Algorithm for 0-1 Programming.

Note that given the solution of the linear program at a node, solving the problem at the two children is much easier than solving it from scratch since we are already very close to the solution. The dual simplex method can be used to quickly resolve an LP after adding one additional constraint.

3.3 Implicit Enumeration (0-1 Programs)

We want to solve the problem:

$$\begin{array}{ll} \text{minimize} & cx \\ \text{subject to} & Ax \leq b,\ c \geq 0,\ x \in \{0,1\}^n \end{array}$$

Note that this formulation assumes that the coefficients of c are all positive. It is not hard to convert problems with negative coefficients into this form.

Figure 55 shows an algorithm based on the "implicit enumeration" with depth-first-search for solving such a problem. Unlike the previous technique, implicit enumeration does not require linear programming. To generate a lower bound on the solution (upper bound for a maximization problem) it uses a solution in which all variables are binary but that does not necessarily obey the inequality constraints. The previous branch-and-bound technique generated the upper bound by using a solution (the LP solution) that obeys all the inequality constraints, but is not necessarily integral.

Here's how to implement the substeps. To check for feasible(C, x) we can just plug x into C. To check if there are no feasible completions we can check each constraint.

For example, if

$$x_f = \{x_1 = 1, x_3 = 0\}$$

and one of the constraints is

$$3x_1 + 2x_2 - x_3 + x_4 \leq 2$$

then

$$\begin{aligned} 3 \cdot 1 + 2x_2 - 1 \cdot 0 + x_4 &\leq 2 \\ 2x_2 + x_4 &\leq -1 \end{aligned}$$

which is impossible since x_1 and x_2 are binary.

Note that this check is not necessarily always going to tell us that there are no possible completions even if there aren't. It is also possible to run more complicated tests that involve multiple inequalities at a time, which might find a completion is infeasible when the simple technique does not.

Notes by Stephen Chenney

- Algorithms
  1. Problem specific aspects
  2. Cutting Plane method
  3. Recent developments
- Airline Crew Scheduling
  1. The Problem
  2. Set Covering and Partitioning
  3. Generating Crew Pairings
  4. TRIP
  5. New system

1 Algorithms (continued)

1.1 Problem Specific Aspects

There are many places in the branch-and-bound and implicit-enumeration algorithms in which domain specific knowledge can help the algorithms. Here we mention some of these.

- In branch-and-bound domain specific knowledge can be used to choose the next variable to branch on in the search algorithms, and which branch to take first. For example based on the domain we might know that certain variables are "more" important than others, or that solutions are more likely to lie to one side or the other of the branch. The implicit enumeration algorithm is similar—domain specific knowledge can help choose a variable to fix and which branch to try first.
- Domain specific knowledge can be used to help generate cuts. In the cutting plane algorithm (later in these notes), a constraint must be added at each step. Domain knowledge can be used to choose a more restrictive constraint, moving you toward the solution faster. Domain knowledge also determines if approximations are acceptable, which allows for more aggressive cutting planes.
- Domain specific heuristics can be used for predicting upper bounds for more aggressive pruning of the tree. Knowing an upper bound a-priori means that many branches can be eliminated based on their linear program's cost even before any integer solution is found.

- Domain specific knowledge can be used to help quickly check for feasibility. For instance, knowing which constraints are more likely to be unsatisfiable suggests an ordering for testing constraints.
- Domain specific knowledge can help solve the linear program or specify that a particular approximate solution is good enough. See the Airline Crew Scheduling problem later in these notes for an example of both these cases.

1.2 Cutting Plane Algorithm

The basic idea is to add a new constraint (geometrically, a new hyperplane) at each step. This plane will cut out non-integer solutions and move the solution closer to an integer solution. Figure 56 shows the intuition. A new plane is added at each step to cut off the corner of the feasible region containing the linear program solution. In Figure 57 the limiting case is shown, where the new constraints approximate a curve bounding the acceptable integer solutions.

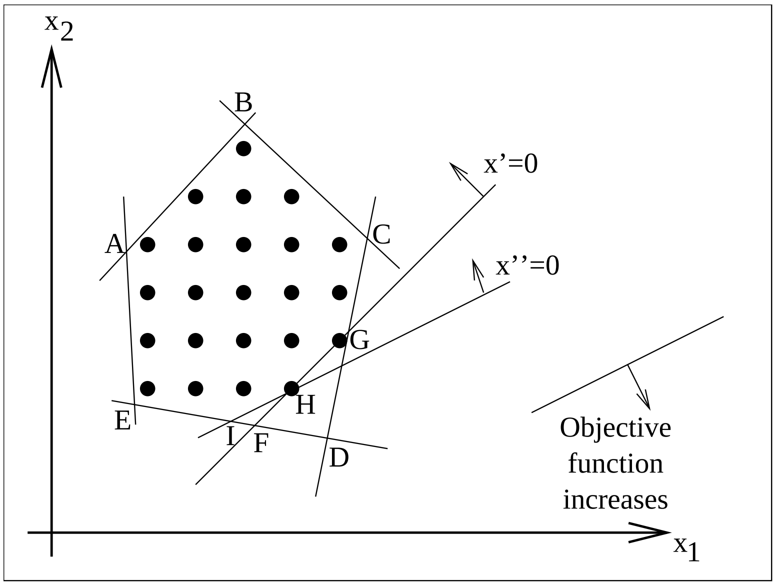


Figure 56: The Basic Approach. Initially, the solution to the linear program is at D. This is not an integer solution, so constraint x' is added, moving the linear solution to F. Then constraint x" is added, giving H as the linear, and integer, solution.

The cutting plane algorithm is:

function $IP(\mathbf{A}, c)$
$x = LP(\mathbf{A}, c)$
while not(integer(x) or infeasible(x))
begin
  add constraint to $\mathbf{A}$
  $x = LP(\mathbf{A}, c)$
end

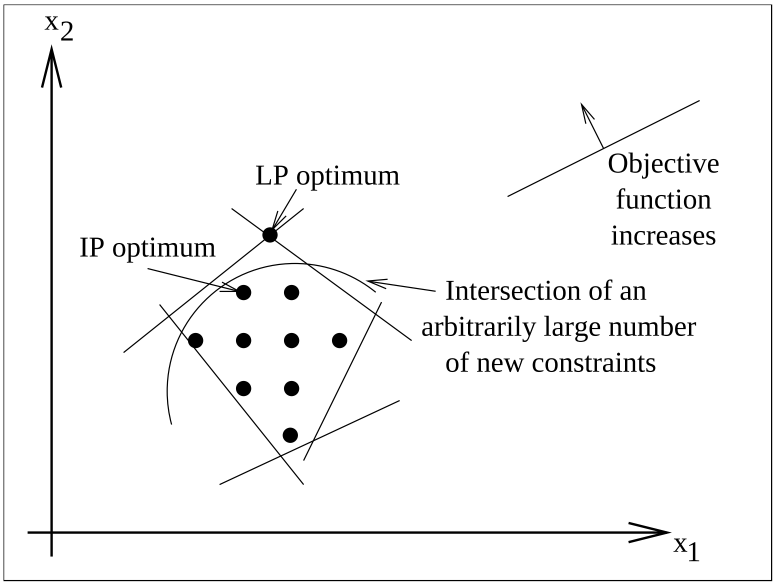


Figure 57: Curve Phenomenon in the Basic Approach

Under some circumstances (problem specific) it is acceptable to add constraints that cut off the optimal solution if it makes finding a good solution faster.

1.3 Recent Developments

- Good formulations—heuristics and theory. The goal is to get an LP solution as close as possible to the IP solution. Examples: Disaggregation and adding constraints (cuts).
- Preprocessing—Automatic methods for reformulation. There's some interesting graph theory involved.
- Cut generation (branch and cut). The idea is to add cuts during the branch-and-bound.
- Column generation—improve formulation by introducing an exponential number of variables. Solve an integer subproblem.

2 Airline Crew Scheduling (A Case Study)

2.1 The Problem

American Airlines, and all other airlines, must schedule crew (pilots and flight attendants) on flight segments to minimize costs.

- over 25,000 pilots and flight attendants.
- over 1.5 billion dollars per year in crew cost.

This process assumes that flight segments are already fixed. In other words, the airline has already decided which planes will fly where and when. Obviously, determining the aircraft schedules is an optimization problem in itself, and the 2 problems are not independent. But treating them independently makes them tractable and gives good answers.

In these notes 2 methods are covered:

- 1970-1992: TRIP (Trip Reevaluation and Improvement Program) This is a local optimization approach which starts with a (hopefully good) guess.
- 1992-present?: Global optimization, approximately.

2.2 Set Covering and Partitioning

Before describing airline crew scheduling we briefly discuss the general problem of set covering and partitioning. Set partitioning will be used in the crew scheduling algorithm.
Given m sets and n items.

$$
\begin{aligned}
A_{ij} &= \begin{cases} 1, & \text{if set } j \text{ includes item } i \\ 0, & \text{otherwise} \end{cases} \quad \begin{array}{l} \text{Columns = sets} \\ \text{Rows = items} \end{array} \\
c_j &= \text{cost of set } j \\
x_j &= \begin{cases} 1, & \text{if set } j \text{ is included} \\ 0, & \text{otherwise} \end{cases}
\end{aligned}
$$

Set Covering:

$$
\begin{array}{ll} \text{minimize} & cx \\ \text{subject to} & Ax \geq 1, x \text{ binary} \end{array}
$$

Set Partitioning:

$$
\begin{array}{ll} \text{minimize} & cx \\ \text{subject to} & Ax = 1, x \text{ binary} \end{array}
$$

As a standard application, consider a distribution problem with some possible warehouse locations (the sets) and some areas (the items) that must be served. There is a cost associated with building and operating each warehouse, and we want to choose the warehouses to build such that all the areas are covered and the cost of building and operating is minimal.

For example:

set	members	cost
s_1	$\{a, c, d\}$	.5
s_2	$\{b, c\}$	.2
s_3	$\{b, e\}$	.3
s_4	$\{a, d\}$	.1
s_5	$\{c, e\}$	.2
s_6	$\{b, c, e\}$	.6
s_7	$\{c, d\}$	.2

$$A = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \end{bmatrix}$$

The best cover is $\{s_2, s_4, s_5\} = .5$. The best partition is $\{s_4, s_6\} = .7$.

2.3 Generating Crew Pairings

A *crew pairing* is a set of flights a crew member will fly, starting and finishing at their *base*. Pairings are broken into duty periods (one per day, separated by overnight rests) which are broken into segments (individual flights). An example, with base at DFW, is:

Duty period 1

Sign in: 8:00

DFW 9:00 - 10:00 AUS (segment 1)

AUS 11:00 - 13:00 ORD (segment 2)

ORD 14:00 - 10:00 SFO (segment 3)

Sign out: 15:15

Overlay is SFO

Duty period 2

Sign in: 8:00

SFO 8:00 - 9:00 LAX (segment 4)

LAX 10:00 - 11:45 SAN (segment 5)

SAN 13:00 - 19:30 DFW (segment 6)

Sign out: 19:45

National pairings typically last 2-3 days (*i.e.*, 2-3 duty periods). International pairings may last for a week, because there are generally fewer, longer flights. A crew member will work 4 or 5 pairings per month. Pairings are actually assembled into groups, called bidlines, which is yet another optimization process. Recent work has focussed on the bidline problem, but we will not discuss it here. A crew member bids for bidlines each month. Bidlines are then distributed according to seniority based rank.

The optimization problem is: given all the segments that need to be covered for a month, find the optimal grouping of segments into pairings, according to the constraints and costs we will define next.

The constraints, apart from the obvious ones (such as segments in a pairing can't overlap in time) are related to union and Federal Aviation Agency (FAA) rules. For example:

- Maximum 8 hours of flying time per duty period
- Maximum 12 hours total duty time (which is flying plus time between flights).
- There is a minimum lay-over time, which depends on the number of hours flown in the previous duty period.
- There is a minimum time between flight in the duty period, so that the crew can move from plane to plane.

The cost of a pairing can include both direct and indirect costs (such as employee satisfaction). Example contributors to the cost are:

- Total duty period time - obviously related to the salary paid.
- Time away from base - also related to salary paid, but also part of employee satisfaction.
- Number and location of overlays - hotels (at least) are paid for by the airline, and some cities are more expensive than others.
- Number of time-zone changes - an employee satisfaction, and maybe union requirement.
- Cost of changing planes.

The overall goal is to **cover all segments with a valid set of pairings that minimize cost**. We must also consider the number of crew available at crew bases (this will also be a factor in scheduling flights in the first place). The problem is simplified in part because flights are roughly the same each day, but it is made more difficult because flight schedules change at the end of the month. For this reason, in part, pairings are generally done on a monthly basis.

One possible approach is to consider all valid pairings and generate costs for each. Then solve it as a set covering problem:

$$\begin{array}{ll} \text{minimize} & cx \\ \text{subject to} & Ax \geq 1, x \text{ binary} \end{array}$$

$$\begin{aligned} A_{ij} &= \begin{cases} 1, & \text{if pairing } j \text{ covers segment } i \\ 0, & \text{otherwise} \end{cases} \\ c_j &= \text{cost of pairing } j \\ x_j &= \begin{cases} 1, & \text{if pairing } j \text{ is included} \\ 0, & \text{otherwise} \end{cases} \end{aligned}$$

Problem: There are billions of possible pairings. For the methods we will discuss, a subset of possible pairings is considered and in some way optimized. For example, with the segments:

1. DFW 900-1200 LGA
2. LGA 1300-1500 ORD
3. ORD 1600-1800 RDU
4. ORD 1700-1900 DFW
5. RDU 1900-2100 LGA
6. RDU 1900-2100 DFW
7. LGA 1400-1600 ORD
8. DFW 1600-1800 RDU

we might generate the possible pairings (this is not an exhaustive list):

1. DFW 9-12 LGA 14-16 ORD 17-19 DFW
2. LGA 13-15 ORD 16-18 RDU 19-21 LGA
3. ORD 16-18 RDU 19-21 DFW 9-12 LGA 13-15 ORD
4. DFW 16-18 RDU 19-21 DFW
5. DFW 16-18 RDU 19-21 LGA 14-16 ORD 17-19 DFW

which corresponds to the matrix

$$A = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 \end{bmatrix}$$

where each column is a pairing (set) and each row is a segment (element that needs to be covered). Someone can then price each pairing for us giving us a cost vector

$$c = [c_1 c_2 c_3 c_4 c_5]$$

2.4 TRIP

TRIP (Trip Reevaluation and Improvement Program) was in use until around 1992. As the name suggests, it reevaluates a previous solution and then attempts to improve it. The initial solution is generally taken to be that of the previous month, modified to adjust for schedule changes. We then repeat the following steps until no improvement is found:

1. Select a small set of pairings from the current solution. The pairings selected should come from roughly the same region (covering similar cities). We will generally choose 5-10.

2. Generate all the valid pairings that cover the same segments. There will typically be a few thousand. Note that if we chose pairings from disjoint cities in the previous step, we would be able to generate no new pairings for this step.

3. Generate the cost of each new pairing.

4. Optimize over these pairings by solving the set partitioning problem. This will generate a new set of pairings that cover the same segments but at minimal cost.

The advantage of this method is that it solves small sub-problems. The big disadvantage is that it only does local optimization. If finding a better solution requires optimizing over a large number of possible pairings and segments, we won't find it.

2.5 New System

Described by Anbil, Tanga and Johnson in 1992 (IBM Systems Journal, 31(1)). The system may have been in use since then, although we don't know if it has been replaced by a better technique. A later paper on the bidline problem suggests it hasn't. The basic algorithm is:

1. Generate a large pool of pairings (around 6 million).

2. Solve an LP approximation using a specialized technique.

3. Uses branch-and-bound for the IP solution, but with heuristic pruning.

Each linear program solved takes around 1 hour. The algorithm does not guarantee the best solution for the chosen pool because of pruning, but it does generate significantly better solutions than the previous method.

2.5.1 Generating a Pool of Pairings

This is speculative, since the authors say very little about it.

We need to generate 6 million initial pairings out of billions of possible pairings.

1. - Node: time and airport
   - Edge: (directed) flight segment, wait time, or overlay
   - Edge weight: "excess cost" of edge. The excess cost is the amount that edge would cost above the minimum it could cost. For example, there might be a fixed cost for a wait independent of its length (just to get on and off the plane) and we don't include this, and a variable cost based on the length, and we do include this.

2. Find the 6 millions shortest paths in the graph that start and end at a crew base.

Nodes in the graph will generally have many edges coming in and going out. Incoming edges will be wait times and overlays ending at that time, and flights arriving at that time. Outgoing edges will be overlays or wait times starting at that node, and flights leaving at that time.

The algorithm given is a heuristic that prefers short time away from base.

2.5.2 Solving the LP Approximation

An LP problem with 6 million columns is too big to solve directly, so instead the following is done (referred to as the SPRINT approach, Figure 58):

1. A = select a small set of m columns (each column is a pairing). $m \approx 5000$
2. Repeat until the optimal is found:
   - Optimize based on A
   - Use the basis generated to "price" the remaining variables. Then set A to the m new "best" columns. ($i.e.$, minimum $r = c_b B^{-1} N - c_n$)

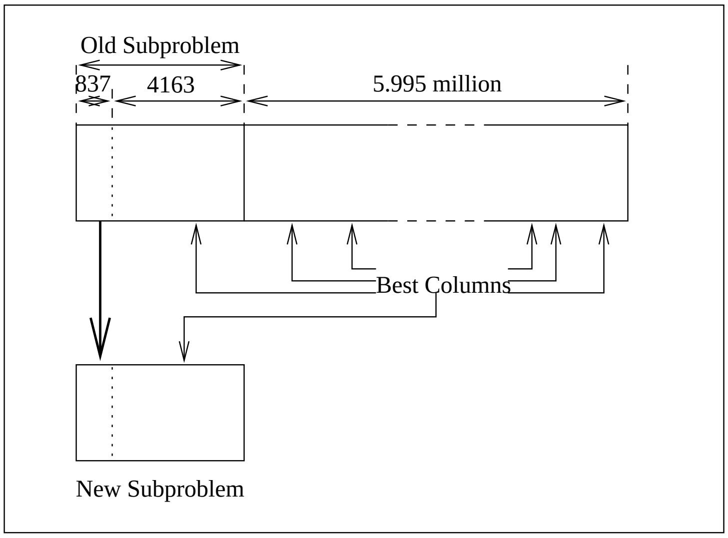


Figure 58: The SPRINT approach to solving the 6 million column problem. The number of rows is 837, which also gives the number of columns in the basis. The basis is always kept for the next round. The remaining $5000 - 837 = 4163$ columns from the subproblem are combined with all other columns to select the best ones for the next round.

2.5.3 Using the LP for the IP

In practice, the variable values for the optimal LP solution are spread between 0 and 1. This says that the optimal linear solution is to use pieces of many pairings to cover all the segments—it's like saying we want part of a crew person. Obviously, we need to either use all or none of a pairing.

The intuition for generating the IP solution is to look for follow on segments. A follow on for a segment is simply the segment that follows it in the final set of pairings. The heuristic we will use for pruning is to assume that a pair of adjacent segments that appear in many good pairings should be a follow on in the final solution.

So we run the following algorithm to select exactly which pairings to use:

- Solve the LP approximation across 6 million columns (as just discussed).
- Select approximately 10K pairings with the best reduced costs.
- Repeat until all segments have a follow on (or end at base):
    1. For all columns (pairings) with nonzero values in the linear program solution, consider all adjacent segments (s_i, s_j) in the itineraries (this includes wraparound so the last segment in the itinerary is followed by the first). Note that if there are n rows (segments) then at most n columns will have nonzero values in the linear program solution (i.e. only the basis can have nonzero values).
    2. For each follow-on (s_i, s_j) being considered weight it with the sum of the values of the linear program solution for the column (pairing) it appears. For example if (2,3) are adjacent in pairing 23 which has value .6 and pairing 95 which has value .2, then its total weight will be .8. Select the follow-on with the largest weight. The intuition is that this follow-on occurs in many good segments, so we will assume it belongs in the final solution. This is similar to choosing a variable to fix in the implicit enumeration algorithm.
    3. Fix (s'_i, s'_j) and throw out all the pairings that include $(s'_i, s_k), k \neq j$. This is the pruning step.
    4. Resolve the LP (across the 10K pairings minus those that have been deleted).
    5. If the solution becomes infeasible, add columns from the original 6 million that satisfy all the follow-ons we have fixed.

As an example, recall our previous set of possible pairings, and the associated matrix

$$A = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 \end{bmatrix}$$

An LP solution is $x_1 = x_2 = x_3 = x_4 = x_5 = 1/2$.
The pairings, expressed in term of the order of their segments s_i were:

$$\begin{gathered} x_1 = (s_1, s_7, s_4) \\ x_2 = (s_2, s_3, s_5) \\ x_3 = (s_3, s_6, s_1, s_2) \\ x_4 = (s_8, s_6) \\ x_5(s_8, s_5, s_7, s_4) \end{gathered}$$

The follow-ons (including wraparound) and their weights are:

Follow-on Pair	Summed Weight
$(1,2)$	1/2
$(1,7)$	1/2
$(2,3)$	1
$(3,5)$	1/2
$(3,6)$	1/2
$(4,1)$	1/2
$(4,8)$	1/2
$(5,2)$	1/2
$(5,7)$	1/2
$(6,1)$	1/2
$(6,8)$	1/2
$(7,4)$	1
$(8,5)$	1/2
$(8,6)$	1/2

The follow-ons $(2,3)$ and $(7,4)$ are the only ones that appear twice. We fix these two follow-ons (whenever multiple follow-ons have weight 1, we fix them all). Fixing these follow-ons does not eliminate any pairings. For the next step we will have to fix the other follow ons in turn, and see which is best, or add more pairings.

2.5.4 Additional Constraints

We must also somehow account for the number of crew available at each base. To do so, we add constraints on the maximum hours available from each base. This is easy to do in the formulation of the linear-program by adding a row for each base and then for each pairing (column) we place the number of hours it requires in the row of its home (where it starts and ends). We then use this as an inequality constraint so the sum of each base-row is less than the total number of hours available at that base.

We also need to patch between months. To do this we separately schedule the first 2 days of each month with additional constraints put in (referring to the previous months ending). Finally, we must have a scheme to manage cancelled or delayed flights. Currently this is done by hand. Every base has a small set of reserve crew which can be put on duty if required.

2.5.5 Conclusions

- Special purpose techniques were used.
- The optimization process was largely separated from the cost and constraints rules.
- A 6 million variable LP was solved as a sub-step, plus a large number of smaller LPs.
- It is currently difficult to find out exactly how much money is saved (older papers were far more forthcoming).

- Current and future work:
  - Putting pairings together into bidlines and optimizing jointly. This is really the same as making the pairings a month long, with lay-over holidays as part of the "pairings" and additional constraints.
  - Handling real-time constraints (cancellations, rerouting).
  - Joint flight and crew scheduling. This is a much bigger problem.

Outline

- Introduction to Triangulation
- Basic Geometric Operations
  1. Degeneracies and Precision Issues
  2. Line Side Test
  3. In-Circle Test
- Convex Hulls
  1. Dual Problem: Convex Polytopes
  2. Algorithms
     - Giftwrapping
     - Graham Scan
     - Mergehull
     - Quickhull
- Voronoi Diagrams
- Delaunay Triangulations (*dual of Voronoi diagrams*)
  1. Properties
  2. Algorithms
     - Naive Algorithm
     - Solution via Reduction to 3D Convex Hull
     - Direct Algorithms

The following three lectures cover triangulation, which is one of the many problems in the larger field of "computational geometry". Problems in computational geometry have found many applications in industry. We will also cover convex-hulls and Voronoi-diagrams because they are closely related to triangulation.

1 Introduction to Triangulation

Triangulation is a process that takes a region of space and divides it into subregions. The space may be of any dimension (although we will focus primarily on the two-dimensional problem here), and the subregions typically are shaped as triangles (2D), tetrahedrons (3D), etc. The goal of triangulation is to produce a subdivision that obeys certain "nice" properties, for example one in which the output triangles are as close to equilateral as possible.

Triangulation is used in many real-world applications. One such application is *finite element simulation*, in which a partial differential equation (PDE) is simulated over a continuous surface or space by first using a triangulation to break the space into small, discrete elements, then simulating the PDE discretely over the elements. Finite element simulations are common in fluid flow applications and similar problems; an example that illustrates the role of the triangulation can be seen at:

`http://www.cs.cmu.edu/~scandal/applets/airflow.html`

Other important applications of triangulation include:

- *Surface approximation*: a continuous surface is approximated by a mesh of triangles. This is used in graphics applications (e.g., face representation), compression of surface representations, interpolation of surfaces, and construction applications (*e.g.*, optimizing dirt movement to contour a region for new construction).
- *Nearest neighbor structure identification*: the triangulation includes data identifying the nearest neighbors of each point in the triangulated data set. This is potentially useful for clustering applications, etc.

In general, triangulation operates on several different types of input. The simplest case (and the one that we focus on in these notes) is when the input is just a set of points which define the vertices of the output triangulation. Another possibility is that the input data is a boundary defining a region of space. In this case, the triangulation must generate the interior points needed to complete the triangulation. Finally, the input can be specified as a density function $\varphi(x, y, z)$, which specifies the density of triangles across the region to be triangulated. This last format is especially useful in simulation, as it allows the person developing the model to focus triangle detail (and thus simulation accuracy) on critical portions of the model.

2 Basic Geometrical Operations

2.1 Degeneracies and Numerical Precision

Before getting into the triangulation algorithms themselves, we first need to consider two important issues that arise when these algorithms are implemented in the real world: handling of Degeneracies and numerical precision.

Degeneracies occur (in two dimensions) when three or more points are co-linear, or when four or more points lie on the same circle. Special care is often needed in triangulation algorithms to make them robust with respect to degeneracies.

Another related problem that crops up in the real world is the issue of numerical precision. Since (in general) computers operate with only a fixed amount of floating-point precision errors are introduced by most arithmetic operations. These errors can lead to inconsistent answers, especially when points are almost degenerate. For example if three points a, b and c are almost on a line the answer to whether a is above $b - c$ could differ depending on how you calculate it. There are three potential solutions to the numerical precision problem (and the related degeneracy problem):

1. Assume that the problem doesn't exist. Many existing implementations do this, but it is a bad idea.

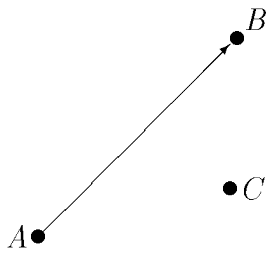
2. Design the algorithm to be robust with respect to degeneracies and to lack of precision (e.g., by using exact or adaptive arithmetic). This is the best solution.

3. Randomly perturb the input data to avoid degeneracies. This may work in most cases, but the possibility of degeneracies or precision problems still remains.

Luckily, most triangulation algorithms only use two basic operations which are sensitive to degeneracies and numerical precision: the *line side test* and the *in-circle test*. These operations encapsulate all of the needed coordinate comparisons, and thus they are the only routines that need be implemented in infinite-precision or adaptive-precision arithmetic. The rest of the triangulation algorithms are primarily combinatorics and graph manipulations.

2.2 The Line-side Test

Given a point C and a line AB, the *line-side test* answers the question: "is C above or below AB?"



The line-side test is easily implemented via a matrix determinant operation; if the determinant operation is built with adaptive-precision arithmetic, the line-side test will be stable with respect to numerical precision. Let the line be defined by points $A = (x_a, y_a)$ and $B = (x_b, y_b)$, and let the test point be $C = (x_c, y_c)$. Then let:

$$D = \begin{vmatrix} x_a & y_a & 1 \\ x_b & y_b & 1 \\ x_c & y_c & 1 \end{vmatrix}$$

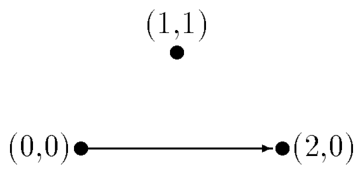


Figure 59: Example of applying the line-side test.

Then the point C is "above" the line $A - B$ iff $D > 0$. Notice as well that the area of triangle ABC is given by $D/2$.

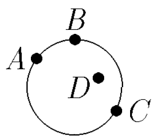
As an example, consider the line AB and the point C with $A = (0,0)$, $B = (2,0)$, and $C = (1,1)$, as illustrated in Figure 59. In this case,

$$D = \begin{vmatrix} 0 & 0 & 1 \\ 2 & 0 & 1 \\ 1 & 1 & 1 \end{vmatrix} = 2 > 0.$$

Thus, C is above AB, and the triangle area is $\frac{2}{2} = 1$.

2.3 The In-circle Test

Given three points A, B, C, and D, the *in-circle test* answers the question: "is D in the circle defined by A, B, and C?"



As in the line-side test, the in-circle test can be implemented with a matrix determinant. Again, if the determinant operation uses adaptive-precision arithmetic, the in-circle test should be stable. Notice, however, that A, B, and C only define a unique circle if they are not colinear.

As before, let $A = (x_a, y_a)$, $B = (x_b, y_b)$, $C = (x_c, y_c)$, and $D = (x_d, y_d)$. Then the particular matrix determinant used for the in-circle test is:

$$D = \begin{vmatrix} x_a & y_a & x_a^2 + y_a^2 & 1 \\ x_b & y_b & x_b^2 + y_b^2 & 1 \\ x_c & y_c & x_c^2 + y_c^2 & 1 \\ x_d & y_d & x_d^2 + y_d^2 & 1 \end{vmatrix}$$

Then the point D is in the circle defined by ABC iff $D > 0$.

Notice that this algorithm generalizes to higher dimensions (as does the line-side test). In this case, in d dimensions, $d + 1$ points determine the d-dimensional sphere, and D becomes a $(d + 2) \times (d + 2)$ determinant.

3 Convex Hulls

Given a set of points S, the *convex hull problem* is to find the smallest convex region that contains all points in S. The boundary of the region is called the *convex hull* of S, and is usually specified by a set of points and their neighbor relation.

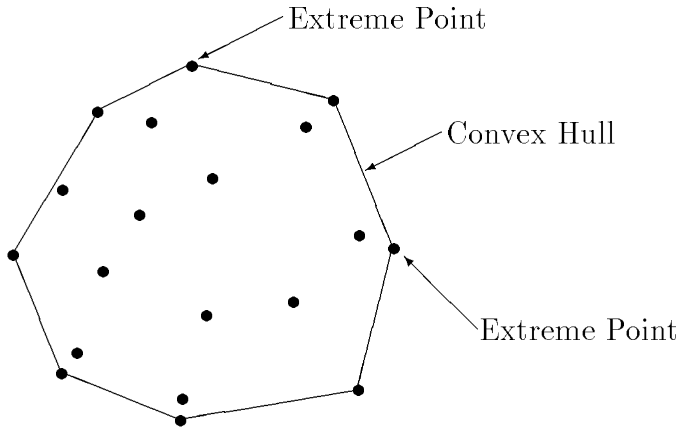


Figure 60: Example of a 2D Convex Hull

Convex hulls are well-defined in any dimension; a 2-dimensional example is given in Figure 60. The 2-D convex hull can be thought of the shape that results when a rubber band is stretched to surround a set of nails, one at each point in S; similarly, the 3-D convex hull is similar to the shape that would be formed if a balloon were stretched to tightly surround the points S in 3-space.

3.1 Dual: Convex Polytopes

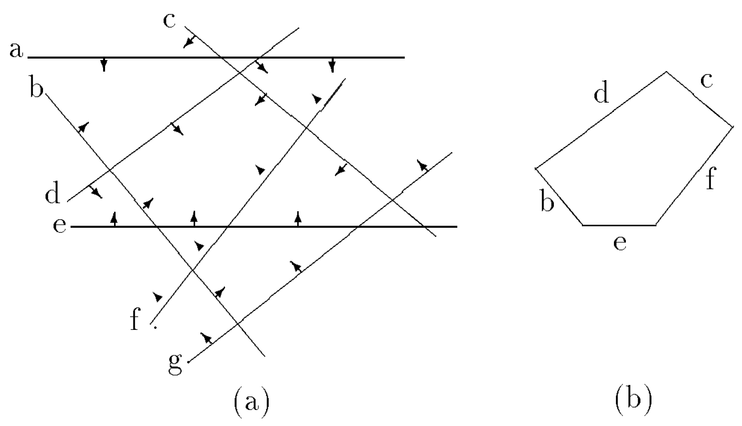


Figure 61: (a) Example 2D Convex Polytope Problem (b) Solution

The convex hull problem has a dual formulation in the problem of *convex polytopes*. The convex polytope problem is to find the region in space that is the intersection of a set of halfspaces (we assume all the halfspaces include the origin). This problem arises in certain linear programming algorithms. Figure 61a illustrates an example of a 2-D convex polytope problem. The solution to a convex polytope problem in general is a list of the "non-redundant" half-spaces (e.g., those lines, planes, or hyperplanes that define the convex region). In the example of Figure 61a, the solution is (b, d, c, f, e), illustrated in Figure 61b.

The convex hull problem and the convex polytope problems are duals, as a problem and solution to one can be mapped into a problem and solution to the other. Thus an algorithm for finding convex hulls can be used to find convex polytopes as well.

The transformation that takes an instance of the convex polytope problem to a convex hull problem is the following, in 2D. Let the ith half-space be defined by the line $\ell_i : a_i x + b_i y = 1$. In higher dimensions, the equation is of the form:

$$\ell_i : \sum_{j=1}^{d} a_{i_j} x_j = 1.$$

Then the dual problem is to find the convex hull of the points p_i, where (in 2D):

$$P_i = \left(\frac{a_i}{a_i^2 + b_i^2}, \frac{b_i}{a_i^2 + b_i^2} \right).$$

In higher dimensions,

$$P_i = \left(\frac{a_{i_1}}{a_{i_1}^2 + a_{i_2}^2 + \cdots + a_{i_d}^2}, \frac{a_{i_2}}{a_{i_1}^2 + a_{i_2}^2 + \cdots + a_{i_d}^2}, \ldots, \frac{a_{i_d}}{a_{i_1}^2 + a_{i_2}^2 + \cdots + a_{i_d}^2}, \right).$$

See Figure 62 for a graphical illustration.

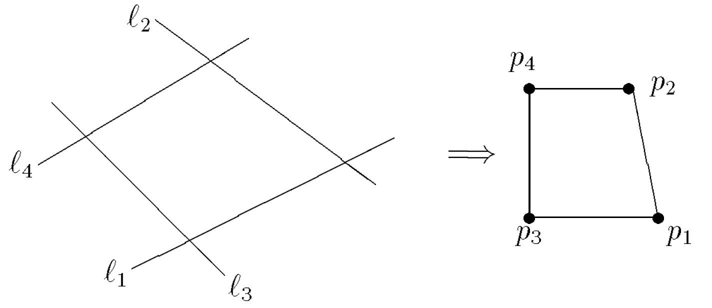


Figure 62: Mapping Convex Polytopes to Convex Hulls

3.2 Algorithms for Finding Convex Hulls

The following is a summary of the most popular algorithms for finding convex hulls. Notice that the algorithms are classified into categories that are similar to those used to classify sorting algorithms. Those marked with a dagger (†) are considered further below. In the time bounds, n is the number of points in the input set S, and h is the number of points in the hull H. The algorithms and bounds we list here are for 2-dimensions although many algorithms generalize to higher dimensions.

- Basic algorithms
  - Giftwrapping†: $O(nh)$
- Incremental algorithms
  - Graham Scan†: $T(\text{sort}) + O(n)$
  - Unordered Insertion: $T(\text{sort}) + O(n)$
- Divide and Conquer algorithms
  - "Merge-based" D&C algorithms
    * Mergehull†: $T(\text{sort}) + O(n)$
  - "Divide-based" D&C algorithms
    * Quickhull†: $O(n)$ best, $O(n^2)$ worst, $O(n \log n)$ average
    * Kirkpatrick-Seidel: $O(n \log h)$
  - Random Sampling: $O(n \log n)$

The theoretical lower-bound on finding a convex hull is $O(n \log h)$.

3.2.1 Giftwrapping

The idea behind the giftwrapping algorithm (in 2-d) is to move from point to point along the convex hull. To start, an extreme point is chosen as the first point on the hull (this can be done easily by choosing the point in S with largest x-value).

Then, on each step, the point in S with the smallest angle α is chosen to be the next point in the hull, where α is defined as the angle between a ray originating at the current point and continuing along the ray that connects the previous and current points, and a ray connecting the current point and the test point. This is explained much more clearly in a picture in Figure 63.

Each step of the algorithm takes $O(n)$ time, since $|S| = n$ and potentially all points in S must be examined to determine the one with minimum angle. Since the algorithm does h steps (one for each point on the hull), the running time of the algorithm is $O(nh)$. If all the points lie on the hull then the algorithm will take $O(n^2)$ time. Because of this worst-case behavior the algorithm is rarely used in practice, although it is parallelizable and does generalize to 3D.

Note that this algorithm also requires additional primitives besides the line-side and in-circle tests.

3.2.2 Graham Scan

The Graham Scan algorithm constructs the convex hull in two pieces (the top and bottom), then connects them together. For each half, it begins by sorting the points in decreasing order of their x coordinates. Then, starting at the largest (or smallest) x, it scans through the points, building the hull along the way. As it encounters a new point, it connects it to

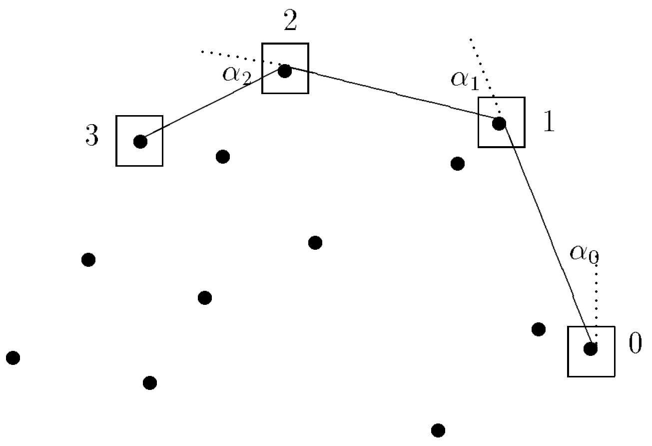


Figure 63: Giftwrapping Algorithm

the previous point, assuming that it will be in the hull. If adding that point causes the hull to be non-convex, the algorithm adds a *bridge* to the appropriate point to ensure that the hull remains convex. This is illustrated in Figure 64; the dashed lines are the bridges added.

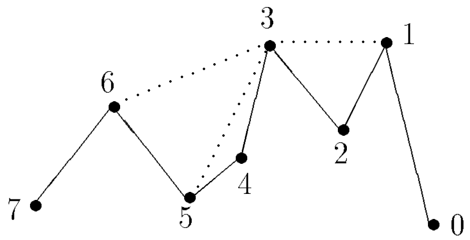


Figure 64: Graham Scan Algorithm

The running time for the Graham Scan algorithm is $T(n) = T(\text{sort}) + O(n)$, since the algorithm must take n steps (one for each point in the scan), and the amortized cost per step can be shown to be $O(1)$. Since $T(\text{sort})$ is $\Omega(n \log n)$ for comparison-based sorts, the Graham Scan algorithm is essentially an $O(n \log n)$ algorithm (unless the data lends itself to a non-comparison-based sorting algorithm).

Finally, note that this algorithm is inherently sequential, and thus does not parallelize well. Even on a sequential machine, it is not the most efficient in practice. Furthermore, it does not generalize to higher dimensions. However, it does have an interesting historical note in that it was developed at Bell Labs in response to a real application, and thus is one of the few hull algorithms that were not theoretically motivated.

3.2.3 Mergehull

The Mergehull algorithm is a divide-and-conquer algorithm that is similar in structure to merge sort. Mergehull has a trivial divide phase, and does most of its work during the merge phase.

Mergehull's divide stage is simple: the algorithm merely splits the input set S into two subsets S_1, S_2 by one of the coordinates. It then recursively solves S_1 and S_2 (e.g., it computes the convex hull for both S_1 and S_2).

After the recursion returns, the algorithm must merge the two convex hulls for S_1 and S_2 into a single convex hull for S. Conceptually, this involves locating two *bridges* between the hulls—line segments connecting two points in S_1 to two points in S_2. Once the bridges are added to connect the two hulls, the interior segments of the subhulls are discarded. See Figure 65 for an illustration. Finding the bridges and discarding the inside edges requires an Overmars dual binary search, which takes time $O(\log n)$. Note that it is not sufficient just to place the bridges between the minima and maxima of the two sub-hulls.

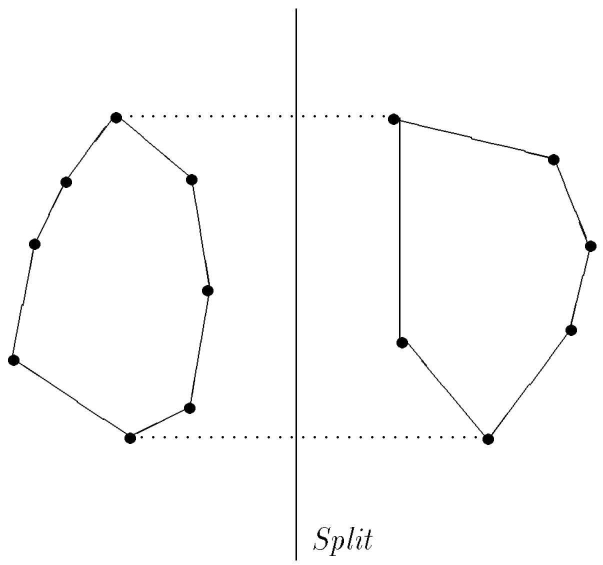


Figure 65: Mergehull Algorithm

If we assume that the points are presorted (to make the subdivision in the divide phase take constant time) and that tree data structures are used throughout, the recurrence for the mergehull algorithm is:

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + O(\log n) \\ &= O(n) \end{aligned}$$

When added to the time of the original sort, the total time is $O(n \log n)$.

It is interesting to note that the mergehull algorithm parallelizes, and that the parallel algorithm does well in practice (it obtains a nearly linear speedup with the number of processors). The parallelization is rather simple: the set S of input points is first divided into P regions (rather than in half, as above). Each of the P processors is then assigned one of the regions, for which it sequentially generates a convex hull. Then, the P regions are merged by finding bridges in parallel.

Finally, the mergehull algorithm can be generalized to three or more dimensions. As the dimensionality increases, however, the merge becomes more difficult. For example, in 3D, the bridges between subhulls are no longer line segments, but instead are surfaces that connect the hulls. Finding these surfaces is more expensive: in 3D, the merge takes $O(n)$ time, so the total time for mergehull is $O(n \log n)$, even when the input points are presorted.

3.2.4 Quickhull

The Quickhull algorithm is another variant on the recursive divide-and-conquer method for finding convex hulls. Unlike the mergehull algorithm that we have just seen, quickhull concentrates its work on the divide phase of the algorithm, and uses a trivial merge step. It is thus very similar in structure to the quicksort sorting algorithm, and we will see below that it has similar time complexity.

Quickhull works (in 2D) by locating three points A, B, and C on the convex hull of its input S (with C above both A and B), then recursively refining the hull between AC and CB. Since both halves of the recursion meet at C, which is on the hull, the merge is essentially a no-op. When the algorithm completes, the upper half-hull between A and B will be complete. The algorithm is then repeated for the lower half-hull. Figure 66 details the process graphically.

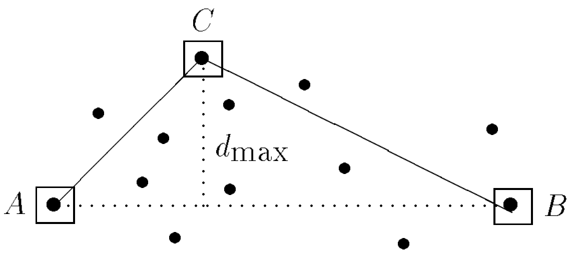


Figure 66: Quickhull Algorithm

The tricky part in Quickhull is implementing the divide stage. A typical algorithm (which works well in the average case but does not guarantee the best time bound) is the following. We start out with two points A, B on the convex hull. These can be found on the first iteration by taking the points in S with minimum and maximum x coordinate. Then, apply the following algorithm:

1. Find the furthest point C from the line joining A and B. To do this, the line-side test (described in Section 2.2) can be used, as it returns the area of the triangle ABC, and

the triangle with largest area is the one that has the furthest point C. Note that, when building the upper half-hull, C should be above AB, whereas C should be above the inverse line BA when building the lower half-hull.

2. Discard all points in triangle ABC, as they clearly will not be on the boundary of the final hull (since ABC is contained within the hull).

3. Recurse on A, C and C, B to fully refine the hull between A and C, and between C and B.

4. Connect the two resulting hulls at C.

5. Repeat for the lower half-hull, and connect it at A and B.

The worst-case running time of the quickhull algorithm is $O(n^2)$, for if the splits (choices of C) are bad, the algorithm may only eliminate one point on each recursion, resulting in a recursion depth of n and a per-recursion cost of n. The input distribution that forces this worst-case time is if all the input points are distributed along a semicircle with exponentially decreasing distance from the left-hand point (see Figure 67).

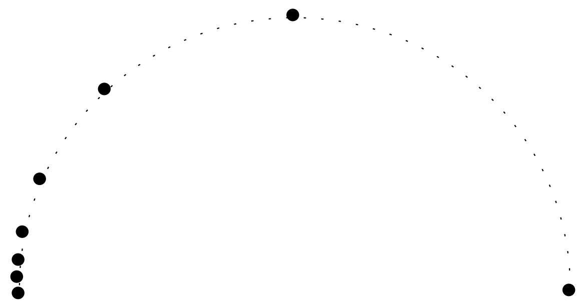


Figure 67: Worst-case Input Distribution for Quickhull

Despite this poor worst-case bound, quickhull runs in $O(n \log n)$ time for most "realistic" point sets. In fact, in practice it can often run as fast as $O(n)$, since the algorithm discards points in step 2, above, reducing the size of the input set for later recursions.

There is also a technique developed by Kirkpatrick and Seidel that locates the ideal "pivot" point C using two-variable linear programming. This variant guarantees the optimal time complexity of $O(n \log h)$. The technique is conceptually similar to the method of using the true sample median to force quicksort to run in $\Theta(n \log n)$. The technique involves finding the bridge segment that connects across the median x coordinate before recursively solving the subhulls themselves. This is the same bridge as found by mergehull, but instead of requiring the solved subhulls for each half it uses linear-programming to find the bridge directly in $O(n)$ time.

Finally, note that quickhull parallelizes easily, and generalizes to three dimensions (in which case the running time is $O(n \log^2 h)$ for an algorithm by Edelsbrunner and Shi).

4 Delaunay Triangulations and Voronoi Diagrams

We will not consider the problem of generating Delaunay triangualations, but we first consider the dual problem of generating Voronoi Diagrams.

4.1 Voronoi Diagrams

A *Voronoi diagram* (or Dirichlet tessellation) is a partition of space into regions $V(p_i)$ based on a set of input points $p_i \in S$ such that:

$$V(p_i) = \{p : |p_i - p| \leq |p_j - p|, \forall\, j \neq i\}.$$

where $|p_i - p|$ indicates the distance from p_i to p (here we assume this distance is Euclidean, but later we will briefly discuss other distance metrics). What this property means is that, for a point p_i, its Voronoi region $V(p_i)$ will contain all points that are closer to p_i than to any other point in S.

Voronoi diagrams are well-defined in any dimension. In two dimensions, the Voronoi regions are convex polygons; in three dimensions, they are convex polyhedra, and so forth. Figure 68 gives an example of a 2D Voronoi diagram.

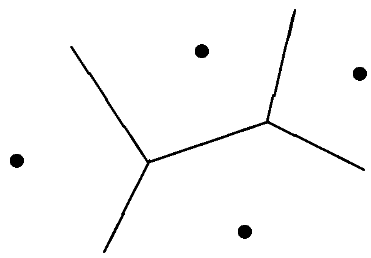


Figure 68: A 2D Voronoi Diagram

Note that there are actually many different ways to represent the diagram. One way is to return for each original point the set of Voronoi neighbors (points that share a Voronoi edge). Another is to return the graph structure formed by the Voronoi edges.

4.2 Delaunay Triangulations

The dual graph of a Voronoi diagram is called a *Delaunay triangulation*. Given a Voronoi diagram, the Delaunay triangulation is formed by connecting the points at the centers of every pair of neighboring Voronoi regions. See Figure 69 for an example. Note that the the Delaunay triangulation will always consist of triangles, since the Delaunay edges join neighboring Voronoi regions, and two neighbor regions of adjacent Voronoi edges will themselves be neighbors. Note that a Delaunay edge is always perpendicular to the Voronoi edge that the two neighbors share, but that the Delaunay edge does not necessarily cross the corresponding Voronoi edge, as illustrated in Figure 70.

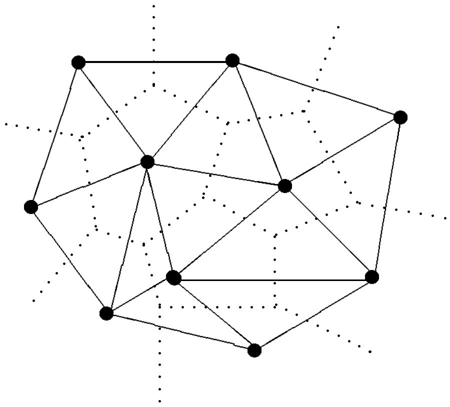


Figure 69: A Delaunay Triangulation (and associated Voronoi diagram)

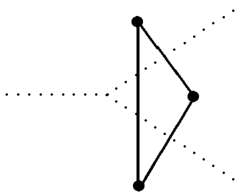


Figure 70: Example of a Delaunay edge that does not cross its Voronoi edge

Finally, note that degeneracies can be encountered in forming a Delaunay triangulation. In particular, if two Voronoi regions intersect only at a corner, there will be more than one possible triangulation (recall that a triangulation contains only non-overlapping triangles). This is illustrated in Figure 71.

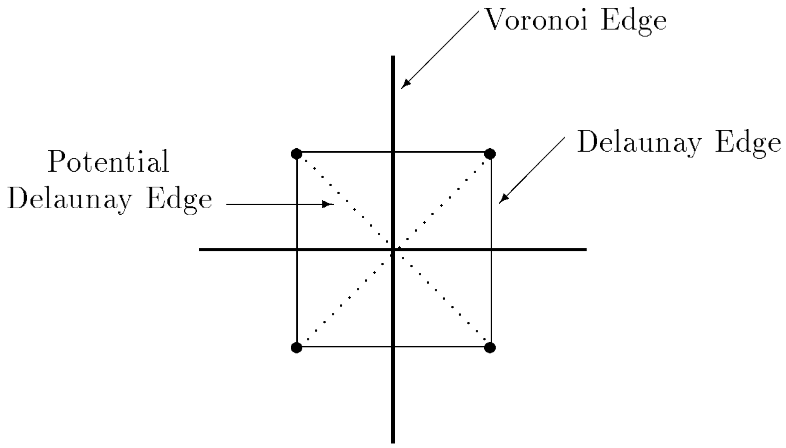


Figure 71: A degenerate case for Delaunay triangulation

4.3 Properties of Delaunay Triangulations

Delaunay triangulations have certain "nice" properties, including:

- The Delaunay triangulation of a non-degenerate set of points is unique. Thus there is only one triangulation that obeys the following set of properties.
- A circle through the three points of a Delaunay triangle contains *no* other points. Also, the intersection of the three corresponding Voronoi edges is at the center of the circle, as shown in Figure 72. This property generalizes to spheres in 3D and so forth.
- The minimum angle across all the angles in all the triangles in a Delaunay triangulation is greater than the minimum angle in any other triangulation of the same points.

These properties tend to result in triangles that are "fat" (closer to equilateral) rather than "skinny". This is a property that is highly desirable in applications, especially finite element simulations where the geometry of the finite elements affects the accuracy and quality of the simulation.

4.4 Algorithms for Delaunay Triangulation

The Delaunay triangulation for a set of points can be found using one of several types of algorithms: a naive direct algorithm, by reducing the problem to a convex hull problem and applying a convex hull algorithm, and by using a direct algorithm. We will consider each of these cases in turn.

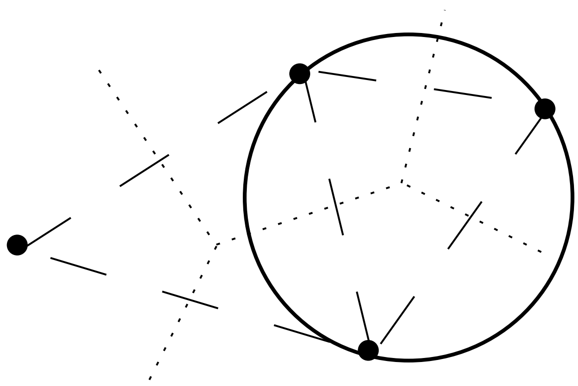


Figure 72: Circle test for Delaunay triangles

4.4.1 Naive Algorithm

A naive way of finding the Delaunay triangulation is the following: for every three-point-subset of the input set, check if any other point is in its circle. Iff not, those three points define a Delaunay triangle.

This algorithm runs in $O(n^4)$ time, since there are n^3 triples of points to check, and each check takes $O(n)$ time (since the in-circle test must be applied n times). Although polynomial-time, this algorithm is quite impractical since, as we will see, there are algorithms that run in $O(n \log n)$ time (and the constant factors in the big-O are quite small).

4.4.2 Solution via Reduction to 3D Convex Hull

A more practical method for finding the Delaunay triangulation of a set of d-dimensional points uses a reduction of the Delaunay triangulation problem to the problem of finding a convex hull in $d+1$ dimensions. For the typical 2D triangulation case, a 3D convex hull is used.

The reduction is as follows. The original input points P are projected upward onto a three-dimensional parabolic surface centered at the origin. Then, the 3D convex hull of those projected points is found. When the hull is represented as a neighbor graph and projected back down to the two-dimensional plane, the result is the Delaunay triangulation of P. More precisely, given P, we first form:

$$P' = \left\{(x, y, x^2 + y^2) : (x, y) \in P\right\}.$$

Then let C =LowerConvexHull(P'), represented as a neighbor graph. Then, the neighbor graph of C is equivalent to Delaunay(P).

The reason that this works is related to the properties of the Delaunay triangulation, in particular, the property that states that any circle through three points of the input set containing no other points circumscribes a Delaunay triangle through those three points. Recall that any slice of a 3D parabolic surface maps onto a circle in the x-y plane. Also, notice that any face of the 3D convex hull defines a slice through the parabola. Thus the projection of this slice onto the plane is a circle, and the circle will pass through the projection of the three points on the plane. Since the slice is only for a single hull face, there will be no other projected points in that circle, so the projection of the hull face defines a Delaunay triangle. Since this works for all hull faces, the result faces of the convex hull in 3-D are the Delaunay triangles in 2-D (when projected down).

Using this reduction to find the Delaunay triangulation gives an efficient algorithm that runs in $O(n \log n)$. However, there are still better algorithms in practice, such as some of the direct algorithms described in the next section.

4.4.3 Direct Algorithms

The following algorithms can be used to find either Voronoi diagrams or Delaunay triangulations, as the two problems are duals of each other. Those algorithms marked with a dagger (†) are easy to parallelize.

- Intersecting Halfspaces†: $O(n^2 \log n)$

  This algorithm works as follows: for each input point, draw a line to every other point. Generate a halfspace separating the input point from every other point (by bisecting the lines drawn in the first step). Compute the halfspace intersection of these halfspaces (a convex hull problem running in $O(n \log n)$). This gives a Voronoi region. Repeat for all n points.

- Radial Sweep
- Incremental Construction: $O(n \log n)$ (expected time)
- Plane Sweep: $O(n \log n)$
- Divide and Conquer Algorithms
  - Merge-based: $O(n \log n)$
  - Split-based†: $O(n \log n)$

In this class we will only discuss the incremental and the split-based divide-and-conquer algorithms.

Notes by Franklin Cho

- Divide-and-conquer Delaunay triangulation and its parallel implementation
- Meshing (triangulation) given a boundary
  - Applications to solid modeling and computer graphics

1 Divide-and-Conquer Delaunay Triangulation

In this section, we will examine the divide-and-conquer approach to Delaunay triangulation by Blelloch, Miller and Talmor. This algorithm performs most of the work on split rather than merge, which allows for easy parallelization. For each recursive call, we keep

- B = Delaunay boundary of region
- P = Interior points

Initially, we set $B = \text{Convex\_Hull}(P_{input})$ and $P = P_{input} - B$. We will assume 2-D data set for now. We also assume that the input points are pre-sorted in x and y. Here is the pseudo-code for the Delaunay triangulation algorithm:

Function $DT(P, B)$

if $|P| = 0$ then boundary_solve(B)
else
  x_m = median point along x-axis
  L = split boundary (set of Delaunay edges along $x = x_m$)
  l = points in L
  $P_{left} = \{p \in P : p_x < x_m, p \notin l\}$
  $P_{right} = \{p \in P : p_x > x_m, p \notin l\}$
  $T_{left} = DT(P_{left}, \text{intersect}(B, L))$
  $T_{right} = DT(P_{right}, \text{intersect}(B,\text{reverse}(L))$
  Return T_{left} and T_{right}

In the first line of the function, boundary_solve is called if the region contains no interior points. There are known linear-time algorithm to triangulate a region with no interior points. The difficult part of the algorithm is finding the Delaunay edges along the line $x = x_m$ (Figure 73). As shown in the previous lecture, we can obtain the Delaunay triangulation of a set of points P by projecting the points onto a 3-dimensional parabola $P' = \{(x, y, x^2+y^2) : (x, y) \in P\}$ and then projecting LowerConvexHull(P') back onto the x-y plane. Based on this idea, we find the Delaunay edges along $x = x_m$ by first projecting the points onto the parabola $P' = \{(x - x_m, y, (x - x_m)^2 + y^2) : (x, y) \in P\}$, which is centered on the line $x = x_m$. Notice that the edges of the lower convex hull of P' correspond to the Delaunay edges on the x-y plane (Figure 74). We then form the set P'' by projecting the points in P' onto the plane $x = x_m$, i.e., $P'' = \{(y, (x - x_m)^2 + y^2) : (x, y) \in P\}$ (note we have just dropped the first coordinate from P'. The lower convex hull of P'' (in 2-D) corresponds to the Delaunay edges along $x = x_m$ (Figure 75).

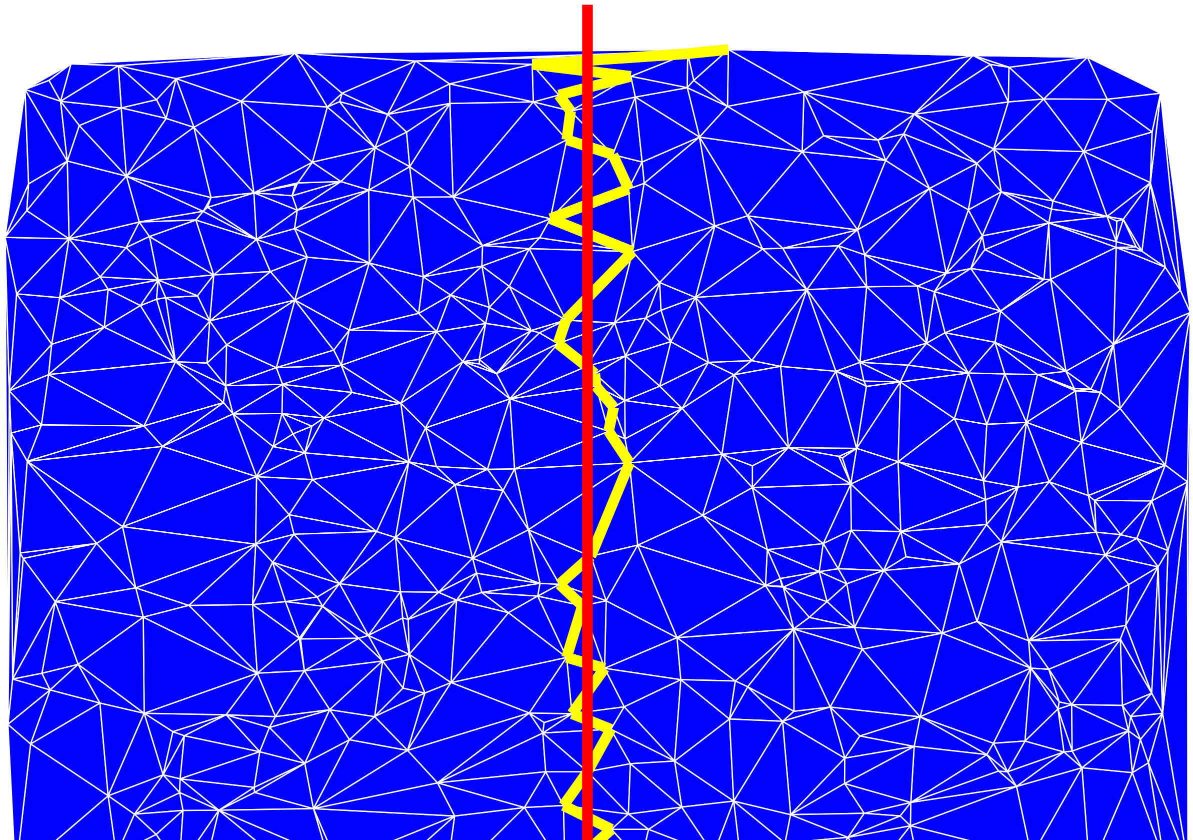


Figure 73: Set of Delaunay edges along $x = x_m$

Let us analyze the asymptotic behavior of the above algorithm. Let n be the total number of interior points ($n = |P|$), let m be the number of points on the border B, and let N be the total number of points ($N = n + m$).

The median x_m can be found in $O(1)$ since we assume that the input points are pre-sorted. The translation and projection trivially takes $O(n)$. The convex hull of C can be found in $O(n)$ since the points are sorted. Since we are dividing the points by their median, we are guaranteeing that each sub-problem has no more than half as many interior points

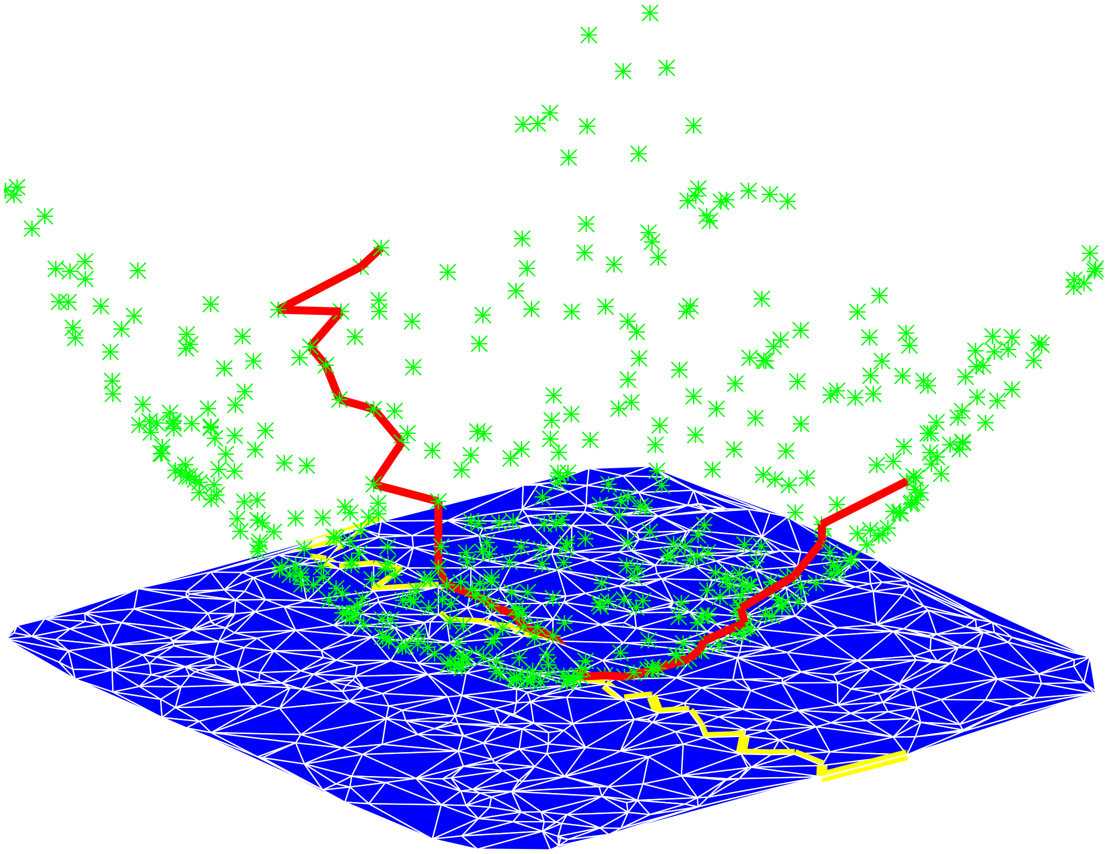


Figure 74: Edges of the Delaunay triangulation on the x-y plane correspond to the edges of the lower convex hull of the points projected onto a paraboloid (P')

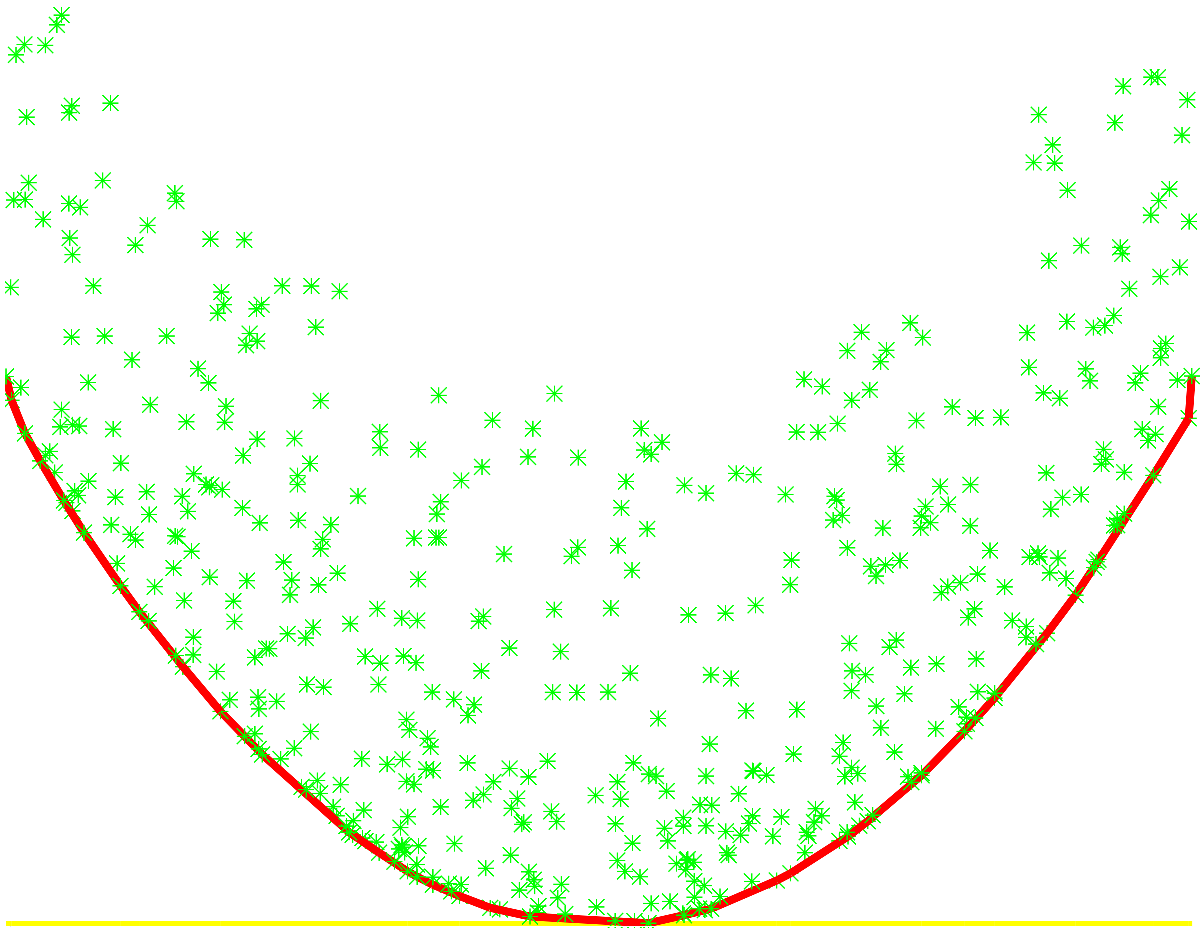


Figure 75: The points in P' projected onto the $x = x_m$ plane (C). The Delaunay edges along $x = x_m$ correspond to the edges on the 2-D lower convex hull of C

as the original problem. Therefore, the running time of the operations on the interior points can be describe by the following recurrence relation:

$$T(n) < 2T(\frac{n}{2}) + O(n)$$

By solving this recurrence relation, we obtain the total running time of $O(n \log n)$ for the operations on interior points.

Now, let us analyze the border-merge operation. For each level of recursion, the border-merge operations take $O(m)$ time. Note that we cannot guarantee that the size of the border in each sub-problem is half the size of the original border. However, we can still place an upper bound on the total amount of time spent on border-merge. Given that there are N total points, the number of edges in the triangulation of the points is bounded by $3N$. Since each boundary edges is counted twice, the absolute upper bound on the number of edges in each level of recursion is $6N$. Therefore, an upper bound on m on each level of recursion is $6N$. Since there are at most $\log N$ levels of recursion, the total time spent merging borders is bounded by $6N \log N = O(N \log N)$.

Counting both operations on points and borders, the total running time of the algorithm is $O(N \log N)$ (since $n < N$). Note that each operation on points and borders can be parallelized easily.

In practice, the following test results were obtained:

- When this algorithm was tested on both uniform and non-uniform inputs, it was found that input sets with uniform and normal distribution bucket well (i.e., it divides nicely when the input set is divided along the median). On the other hand, Kuzmin (zoomed) and line distributions don't bucket well.
- Overall, the running time with non-uniform distributed input set is at most 1.3 times slower than the running time with uniform distributed input. (Previously, the non-uniform inputs took at least 5 times longer).
- This algorithm scales well with problem size, and can solve problems too big for serial machine. The parallel algorithm is only about 2 times slower than a sequential code. (Previously, the parallel algorithm ran 4 times slower than a sequential code).
- This algorithm is a practical parallel 2D Delaunay triangulation algorithm, and it is the first to be competitive with serial code.

Does this algorithm generalize to 3D? Yes in theory, but the algorithm requires 3D convex hull generation as a subroutine, and it is not clear how to parallelize this operation. Also, the border merge is more complicated with 3D borders. Generalizing this algorithm to 3D would be an interesting future work.

2 Incremental Delaunay Triangulation Algorithm

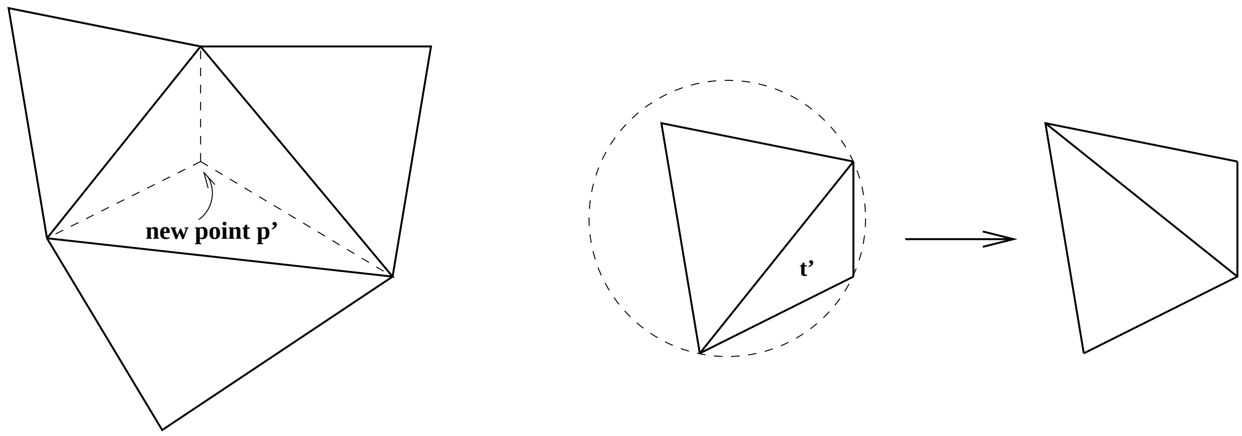
In this section, we will discuss the incremental approach to constructing the Delaunay triangulation. The incremental approach means that the points are added one at a time.

Whenever a point is added, we update the Delaunay triangulation so that at each stage, a complete Delaunay triangulation is generated with the points added so far.

In this algorithm, we assume that the points in P are already Delaunay triangulated. To add a new point p':

1. Locate the triangle t in $DT(P)$ which contains p'
2. Add p' and add the edges from p' to the three corners of t
3. Fix up neighboring regions by "edge flipping"

Step 3 requires more explanation. For each of the newly formed triangle T_{new}, we perform the in-circle-test to see if T_{new} is well-formed: we form the circumcircle of T_{new}, and test if it contains any of the other points. If it does, then T_{new} is not well-formed, and we "flip" the edge (Figure 76). This is repeated for newly created triangles and is discussed in more detail in Lecture 17.



(a) new point p' is added to triangle t, and new edges are formed to the three corners of t

(b) Perform in-circle-test for each sub-triangle, and fix up the neighboring region by "edge flipping"

Figure 76: Step 3 of the Incremental Delaunay triangulation. If the in-circle-test fails, then the edge is "flipped".

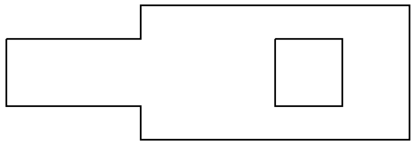
3 Mesh Generation

In this section, we will discuss mesh generation, also called unstructured grid generation, or just triangulation. Given a boundary, we want to triangulate the interior by possibly adding points. We want the final triangulation to have the following properties:

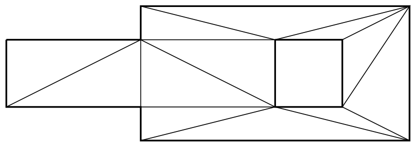
1. Bounded aspect ratio for triangles, i.e. triangles that are not skinny (don't have any small angles).

2. As few triangles as possible. This criterion is important for finite element analysis, for example, because the running time is a function of the number of triangles.

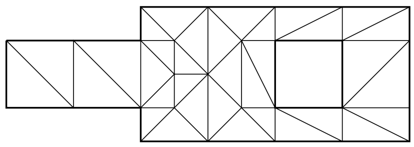
Although, as discussed previously, Delaunay triangulation guarantees the triangulation of a set of points that has the greatest minimum angle (at least in 2-D), as Figure 77 illustrates it is often possible to create triangles with larger minimum angles by adding extra points.



(a) Input Widget



(b) Triangulation without adding points



(c) Triangulation with 6 interior and 10 boundary points added

Figure 77: Extra points can be added to reduce the aspect ratio of the triangles

3.1 Ruppert's Algorithm

Given a set of 2D segments and an angle $\alpha \leq 20°$, Ruppert's algorithm triangulates the input set such that for each triangulated region:

1. no angle is less than α
2. number of triangles is within a constant factor of optimal

Although $\alpha \leq 20°$ is the limit that has been proved, $\alpha \leq 30°$ works well in practice. Ruppert's algorithm works very well in practice, and is used in the "triangle" application.

The basic idea behind Ruppert's algorithm is:

1. Start with Delaunay triangulation of endpoints of segments. As shown previously, Delaunay triangulation would produce the maximum minimum angle.

2. Get rid of skinny triangles by

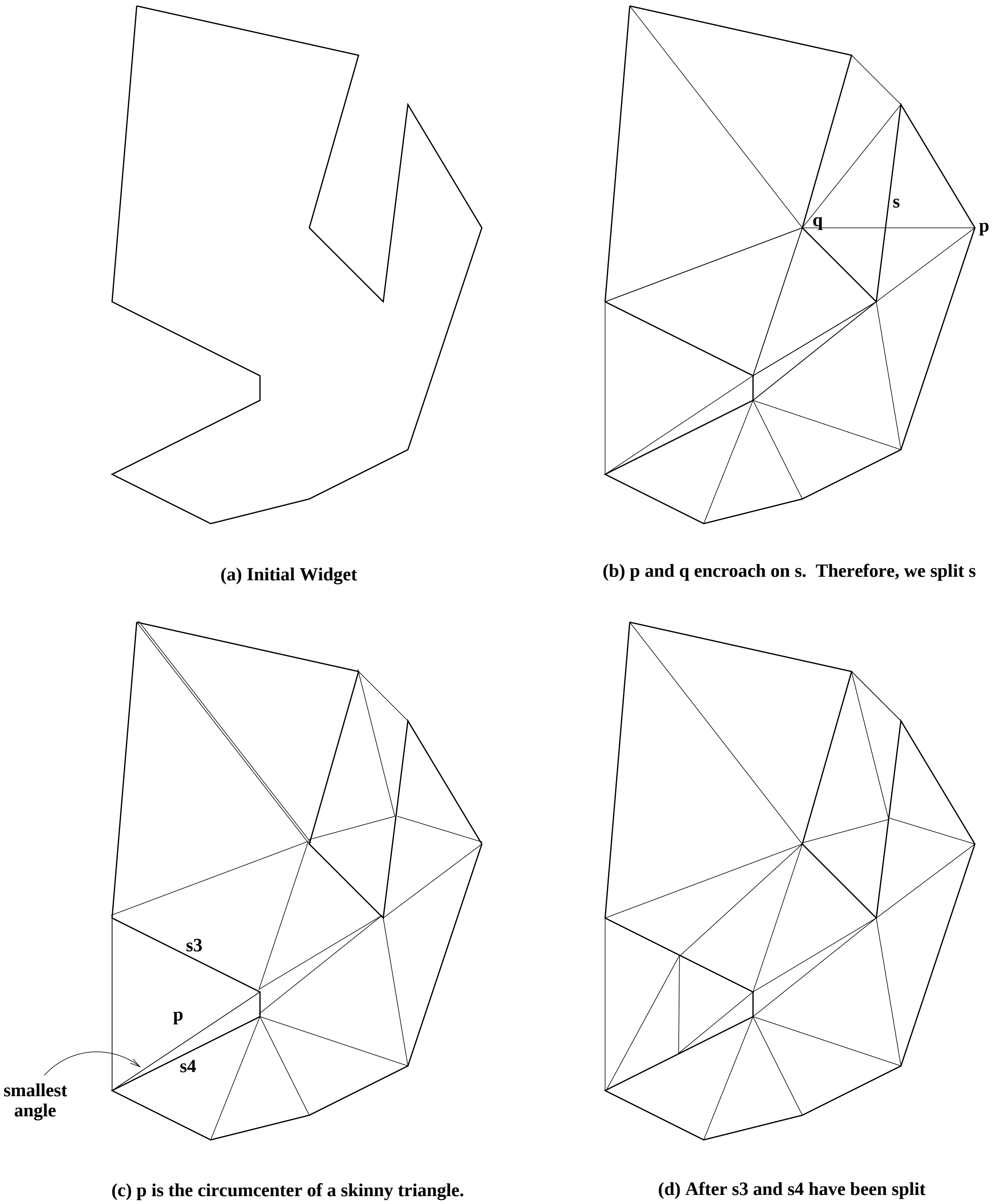
   - splitting segments in half
   - adding a new point in the center of existing triangle

To describe the algorithm, we will use the following terminology. A *segment* denotes either the input segment, or part of an input segment. An *edge* denotes a Delaunay edge. A *vertex* denotes endpoint of a segment or an edge (the Delaunay vertices). A point *encroaches* on a segment if it is inside the segment's *diametrical circle*. A diametrical circle of a segment is a circle with the segment as its diameter.

Here is the pseudo-code for Ruppert's Algorithm:

S = input segments
V = endpoints of S
$T = DT(V)$ // Delaunay Triangulation
Repeat
    while any $s \in S$ is <u>encroached</u> upon, split s in half, update T
    let t be a skinny triangle in T (i.e., t contains an angle smaller than α)
    p = circumcenter(t)
    if p <u>encroaches</u> on segment $s_1, s_2, .., s_k$,
        split $s_1, s_2, .., s_k$ in half, update T
    else
        $V = V \cup p$, update $T = DT(V)$
Until no segment is encroached upon and no angle is $< \alpha$
Output T

Figure 78 illustrates Ruppert's algorithm:



(a) Initial Widget

(b) p and q encroach on s. Therefore, we split s

(c) p is the circumcenter of a skinny triangle.
p encroaches on s3 and s4, and therefore
we must split s3 and s4

(d) After s3 and s4 have been split

Figure 78: Successive steps of Ruppert's algorithm

Let us analyze the running time of this algorithm. If output has n triangles, then the algorithm goes through at most n iterations, and each iteration makes at most n updates. Therefore, the worst-case running time is $O(n^2)$. The algorithm runs in $O(n \log n)$ time in practice. It is currently not known whether Ruppert's algorithm can be parallelized.

The proof of "within constant factor of optimal" clause is based on the concept of "local feature size". Given a point x, the local feature size $\mathrm{lfs}(x, y)$ is defined to be the minimum radius of the circle centered at (x, y) that contains two non-incident segments. $\mathrm{lfs}(x)$ is a continuous function. We will state without proof that:

$$V < C_1 \int_A \frac{1}{(\mathrm{lfs}(x, y))^2} dx dy$$

where V is the number of vertices returned by Ruppert's triangulation, C_1 is a constant and A ranges over the area of the widget. Furthermore it is possible to prove that any triangulation requires at least

$$C_2 \int_A \frac{1}{(\mathrm{lfs}(x, y))^2} dx dy$$

vertices ($C_2 < C_1$). These together show that the number of vertices generated by Ruppert's algorithm is within a constant factor of optimal.

- Mark Adams : Finite Element Methods
  1. FEM Intro
  2. Approximating the Weak Form with Subspaces
  3. Picking Good Subspaces
  4. Linear System Solvers
  5. Multigrid
  6. Unstructured Multigrid Algorithms
  7. Parallel Multigrid Solver for FE
- Distance Metrics for Voronoi Diagrams and Delaunay Triangulations
- Surface Modeling Using Triangulation
  1. Terrain Modeling
     (a) Methods for Representing Terrains
     (b) Benefits of Triangulation
     (c) Algorithm

1 Triangulation in Finite Element Methods (Mark Adams)

We begin with a look at the finite element method (FEM) and a parallel algorithm for solving FEM problems which makes use of the Delaunay triangulation.

1.1 FEM Intro

We will examine the basic FEM approach by applying it to the heat equation,

$$k\left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} + \frac{\partial^2 T}{\partial z^2}\right) + Q = \rho c \frac{\partial T}{\partial t}$$

defined on a domain Ω with boundary $\partial\Omega$ where T is temperature, and Q is the heat generated in the domain. Take T = 0 on the boundary of Ω, and for simplicity assume that we are looking at the steady state case in 1D, so

$$\frac{\partial T}{\partial t} = \frac{\partial^2 T}{\partial y^2} = \frac{\partial^2 T}{\partial z^2} = 0.$$

Thus, we have

$$k\frac{\partial^2 T}{\partial x^2} + Q = 0.$$

This is referred to as the strong form of the equation. In order to apply the finite element method, we must develop a computable form of this equation : the weak form. We start by writing a residual for the strong form as

$$R_\Omega = k\frac{\partial^2 T}{\partial x^2} + Q - 0.$$

Using this equation, the so-called weighted residual form is

$$\int_\Omega W \cdot R_\Omega \partial\Omega = 0$$

where W is some arbitrary function of position with the property that $\forall W\ s.t.\ W = 0$ on $\partial\Omega$. This assumption of W = 0 on the boundary, called a Dirichlet boundary condition, gives us a particular weak form of the original equation called the Galerkin form. We have

$$\int_\Omega \left(W \cdot k\frac{\partial^2 T}{\partial x^2} + W \cdot Q\right)\partial\Omega = 0,$$

and by applying integration by parts, we arrive at the weak form.

$$\int_\Omega \left(\frac{\partial W}{\partial x}k\frac{\partial T}{\partial x} - W \cdot Q\right)\partial\Omega = 0$$

1.2 Approximating the Weak Form with Subspaces

We now apply the finite element approximation by discretizing the domain of the problem into nodes and introducing functions which interpolate the solution over these nodes. In other words, we construct basis sets ϕ and φ such that

$$\tilde{T} \in span(\phi_1, \phi_2, ...\phi_n) \qquad \tilde{T} = \sum_{i=1}^{n} u_i \cdot \phi_i$$

$$\tilde{W} \in span(\varphi_1, \varphi_2, ...\varphi_m) \qquad \tilde{W} = \sum_{i=1}^{n} w_i \cdot \varphi_i$$

where the u_i form the solution we seek, and the w_i are the weights introduced earlier. The sets ϕ and φ are often taken to be the same. This produces a particular sort of approximation known as a Bubnov-Galerkin approximation.

Applying these subspace definitions to the weak form of the heat equation, we get

$$\sum_{i=1}^{n} w_i \int_\Omega \left(\sum_{j=1}^{n} u_j \frac{\partial\varphi_i}{\partial x}k\frac{\partial\phi_j}{\partial x} - \varphi_i \cdot Q\right)\partial\Omega = 0 \qquad \forall w_i.$$

Thus, for $i = 1 \ldots n$

$$\sum_{j=1}^{n} u_j \int_\Omega \left(\frac{\partial\varphi_i}{\partial x}k\frac{\partial\phi_j}{\partial x}\right)\partial\Omega = \int_\Omega \varphi_i \cdot Q\partial\Omega.$$

We now have a linear system $Au = f$ which we can solve for u. In this case,

$$A_{ij} = \int_\Omega \left(\frac{\partial \varphi_i}{\partial x} k \frac{\partial \phi_j}{\partial x} \right) \partial\Omega$$

and

$$f_i = \int_\Omega \Big(\varphi_i \cdot Q \Big) \partial\Omega.$$

1.3 Picking Good Subspaces

In order to achieve accurate results, we must take care in selecting the subspaces ϕ and φ. Methods which make use of sine transforms and wavelets, referred to as spectral methods and element free FEM, respectively, are occasionally used, primarily in research. However, the majority of FE work uses piecewise linear (or quadratic etc.) polynomials. Figure 79 illustrates the use of 1D piecewise linear polynomials for the subspaces. Here we have chosen $n = 5$, and we have applied the Bubnov-Galerkin approximation of $\phi = \varphi$. As can be seen in the figure, the domain Ω has been discretized into nodes, represented by empty circles. The hat functions interpolate between solutions at nodal points. Note that we assume that we have solution values at the boundaries of the domain. Figure 80 depicts a similar situation but this time in 2D with 2D piecewise linear polynomials. The dotted lines show the hat function over one node in the mesh.

The nodes which discretize the domain define the elements which give the method its name. In 2D these elements are usually triangles or quadrilaterals. In 3D they are normally tetrahedra or hexahedra. Hexahedral elements have a part of a higher order space than tetrahedral elements, so one can use larger hexahedral elements and less of them to get a solution which is as accurate as a solution generated from a mesh with a larger number of smaller tetrahedral elements. This can cut down on computation time. However, it is much easier to automate the process of generating triangular and tetrahedral meshes, so these elements are particularly appropriate for large, complex models.

1.4 Linear System Solvers

We now turn our attention to methods for solving the linear systems generated by the FEM. This is normally the most costly step in a FEM solution. Recall that we want to solve for u in $Au = f$. In this equation, A is sparse and there are roughly a constant number of non-zeros per row. In addition, A is symmetric in structure (*i.e.*, nonzeros vs zeros are symmetric), and can have symmetric or non-symmetric coefficients. Finally, we normally have a fairly large system with $10^4 - 10^{7+}$ equations. We can solve this system using either direct or iterative methods which we briefly examine below.

1.4.1 Direct Methods (*e.g.*, Gauss Elimination)

Start by factoring $A \Rightarrow LU$ *s.t.* $L^T, U \in$Upper Triangular. The cost of this step $\approx O(n^2)$.

Now, solve $Ly = f$ for y and $Uu = y$ for u. The cost $\approx O(n^{3/2})$.

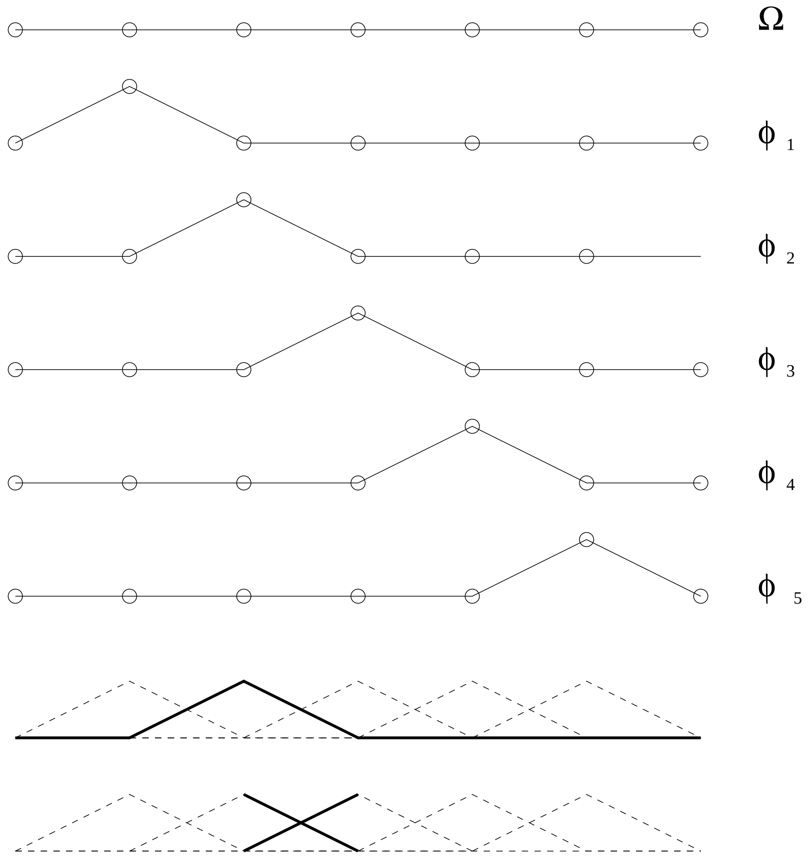


Figure 79: 1D Piecewise Linear FE Shape Functions

The cost of this method is sensitive to vertex ordering, so we have a graph problem. This ordering problem has been studied extensively, and is NP complete. Successful solutions include nested dissection, a divide and conquer algorithm, and minimum degree, a greedy algorithm.

1.4.2 Iterative Methods

The total cost of any of these methods is (number of iterations) $\cdot O(n)$, since for FE discretizations a matrix vector multiply has cost $= O(n)$.

Matrix Splitting

Take $A = (D - U)$ where, for example, $\quad D = diagonal \Rightarrow \quad Jacobi$

Substituting into the equation $Au = f$ and rearranging, we have $u = D^{-1}(f + Uu) \Rightarrow$, so the iterative form is $u_{k+1} = D^{-1}(f + Uu_k)$.

The number of iterations $\approx O(n)$

Conjugate Gradients (CG)

Look for the "best" solution in $span\{f, Af, A^2f, A^3f, ...\}$

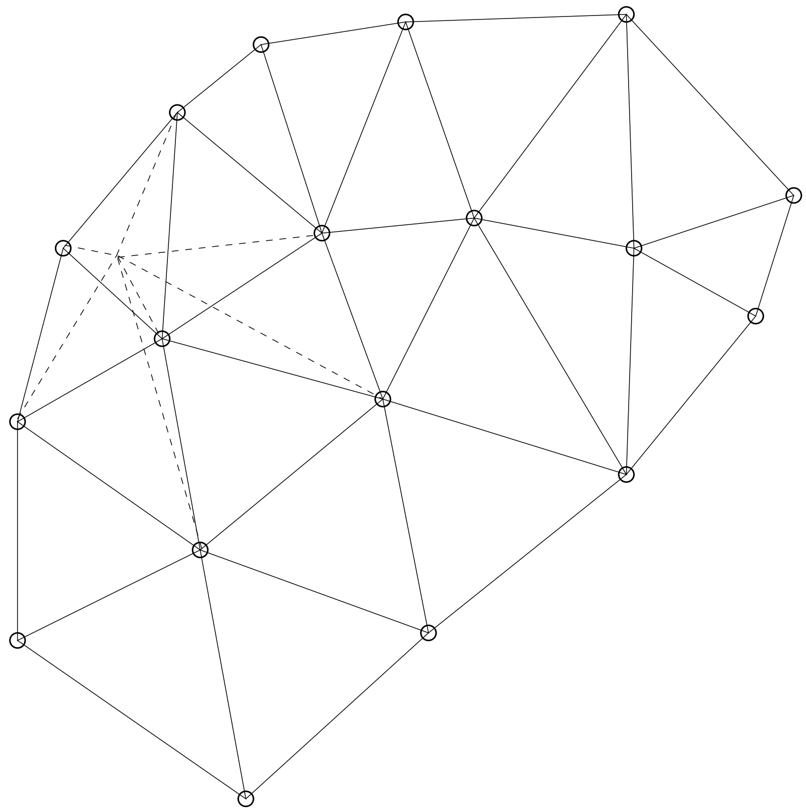


Figure 80: 2D FE Mesh

The number of iterations $\approx O(n^{1/2})$

Domain Decomposition

"Divide and conquer" the spatial domain.

Multigrid

"Divide and conquer" the frequency domain.
number of iterations $\approx O(1)$

Preconditional Conjugate Gradient (PCG)

The mathematical idea is to premultiply both sides of the equation by the matrix $M \approx A^{-1}$. Thus, $MAu = Mf$. Note that we don't explicitly compute $MAu = Mf$ in practice. We just want a preconditioner for which solving $Mz = y$ is cheap in comparison to its effectiveness.

One can choose from a variety of different preconditioners which vary greatly in their cost and overall effectiveness. Note that by effectiveness we refer to how fast the iterative method converges to a solution. Examples of preconditioners include :

- Matrix splitting, in which we use the inverse of a diagonal matrix D. This method is cheap but ineffective.

- Direct method, in which we directly calculate $M = A^{-1}$ in 1 iteration. This is extremely effective, but also very expensive.
- Incomplete factorization, in which we take $A \approx LU$. This method is not all that cheap but is fairly effective.
- Domain decomposition, in which we use a block diagonal matrix instead of a diagonal matrix as in the case of matrix splitting. This has cost and effectiveness similar to that of incomplete factorization.
- Multigrid. This method is rather expensive, but it is very effective. We will use this method in the algorithm discussed below.

1.5 Multigrid

Simple iterative methods are good for large problems and reduce "high frequency" error, *i.e.*, they smooth the residual. Hence, we will use cheap iterative methods at multiple scales of resolution so that the iterative methods are used in cases for which they work best. The basic idea is that we want to generate coarser meshes from an initial fine mesh by reducing the number of vertices in a finer mesh and then remeshing. We then solve the problems on the coarser mesh and interpolate back to the finer mesh. Figure 81 shows a simple progression of coarser meshes.

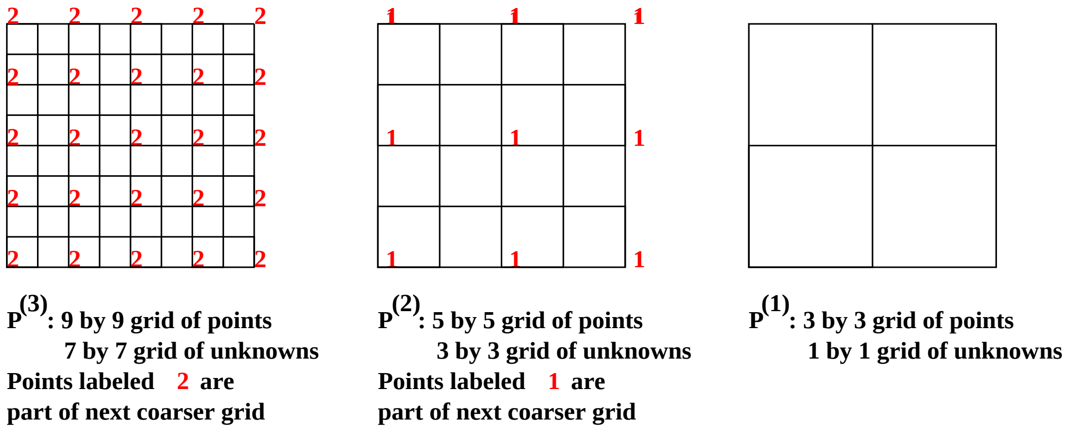


Figure 81: Classic Multigrid Sequence of Grids

1.6 Unstructured Multigrid Algorithms

We apply the following algorithm recursively :

- Evenly coarsen vertex set $\rightarrow$ smaller vertex set
- Mesh this vertex set
- Create interpolation operators R_i and matrices A_i, for coarse grids

We coarsen a set of mesh nodes by finding the maximal independent set of these nodes and using this as the set of nodes for the new coarse set. This new set is meshed using Delaunay tessellation. We use the finite element shape functions (the ϕ's) to generate interpolation operators (R_i), and we calculate the coarse grid matrices using $A_{i+1} = R_i A_i R_i^T$. Figures 82 and 83 show examples of the grid coarsening process.

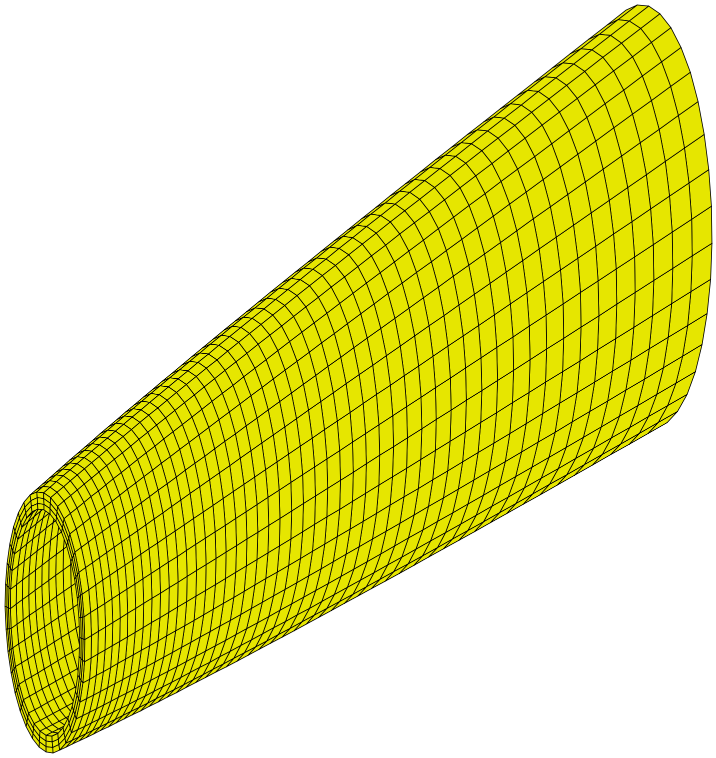


Figure 82: Sample Input Grid

1.7 Parallel Multigrid Solver for FE

Mark Adams has implemented a parallel version of the algorithm discussed above by making use of and developing a variety of software components. The system uses MPI for the message passing implementation, and a package called ParMetis to partition a mesh into sections which can then be assigned to separate processors. The R_i and A_i are determined by Adam's MGFeap program. The matrix A_0 is generated using a serial FE code called FEAP. A program called PETSc handles the parallel preconditional conjugate gradient calculations and the multigrid infrastructure and provides numerical primitives with a multigrid preconditioner.

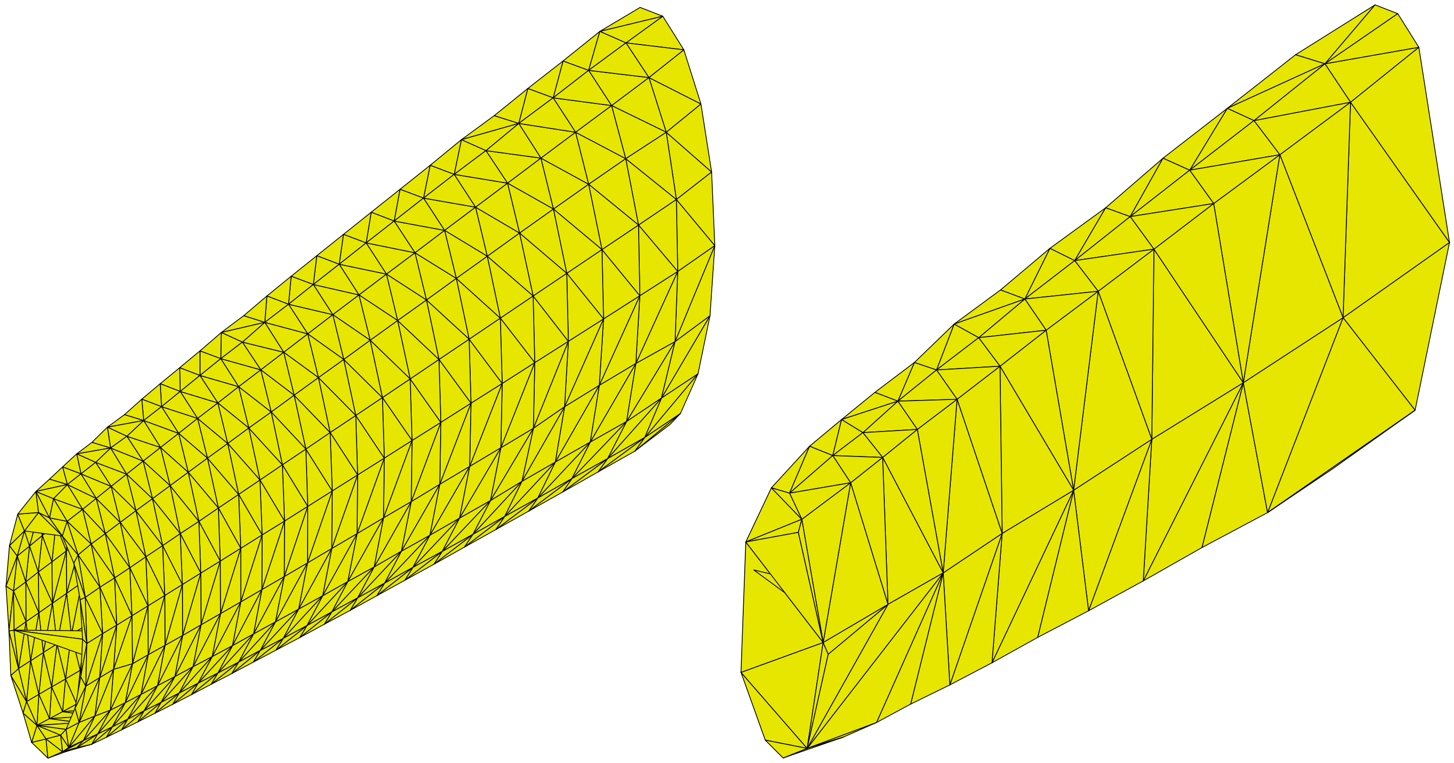


Figure 83: Sample Coarse Grids

2 Distance Metrics for Voronoi Diagrams and Delaunay Triangulations

We are not restricted to using only Euclidean distances when computing a Voronoi diagram or its dual, the Delaunay triangulation. Other possible distance metrics include :

- distances measured on spheres or conics
- Manhattan distance in which only the paths along coordinate axis directions are considered, *i.e.*, $d = \Delta x_1 + \Delta x_2$
- Supremum distance for which $d = max(\Delta x_1, \Delta x_2)$
- Karlsruhe distance which is a sort of polar distance for which

$$d = \begin{cases} |r_1 - r_2| + |\phi_1 - \phi_2| & if |\phi_1 - \phi_2| < \pi/2 \\ |r_1 + r_2| & otherwise \end{cases}$$

- weighted distance for which $d(p, p_i) = 1/w_i |x - x_i|$ for all points p where p_i are the seeds
- shortest path on a graph, for example the shortest path between two points on a city map if the path must stay on streets

3 Surface Modeling Using Triangulation

We will examine the use of triangulations in modeling and interpolating surfaces, particularly with respect to terrain modeling. In the first case, we wish to reduce the time and space required to represent and manipulate surfaces. For the second, we want to find interpolating surfaces between data points. Techniques to achieve these goals can be applied to problems in fields such as terrain modeling, computer graphics, medical imaging in which surface models are built by processing consecutive 2D slices of a volume of interest, and geophysical modeling used, for example, in oil exploration.

3.1 Terrain Modeling

Terrain modeling has been used for decades in military vehicle simulators and map displays in order to accurately represent landscape features. The dirt moving application discussed in an earlier lecture also makes use of terrain modeling.

3.1.1 Methods for Representing Terrains

We could use a variety of different approaches to model terrains. For example, we could describe a terrain as a collection of line segments approximating the curves of a contour plot. However, it would be difficult to convert such a representation from one measurement system to another, *e.g.*, from feet to meters. In addition, it is unclear how one would determine elevation values for points near but not on contours. Another option is to use a height field to store elevation data, but this representation is discontinuous and uses large amounts of space to store regions with constant elevations which could be compressed.

We could also view terrain modeling as a lossy compression problem. From this point of view we need to select an appropriate set of basis functions for the compression. If we use triangles as the bases, then we can use large triangles in terrain areas in which the slope is not changing much and smaller ones only in areas of changing slope. Thus, we compress the regions of constant slope. In particular, the second derivative of the terrain surface gives a measure of local feature size, so we want to have a greater density of triangles in regions with large second derivative values. This is, in fact, the representation used in terrain modeling.

It is interesting to note that the use of TINS, or triangulated irregular networks, in terrain modeling dates from the 1970s, before efficient Delaunay algorithms were known.

3.1.2 Benefits of Triangulation

Using a triangulation to represent a terrain provides a number of important benefits. For example, as discussed above, such a representation does not require much space, since large triangles can be used to represent areas of constant elevation, and we only need dense concentrations of triangles in areas where the slope changes rapidly. In addition, most modern graphics systems are optimized for rendering triangles, so rendering the terrain is quite efficient. It is also straight forward to find the slope of a triangle, so finding the slope at a particular point in the terrain is easy. Furthermore, constant height lines in the terrain are straight lines on the approximating triangles, so reconstructing contour lines is fairly easy.

3.1.3 Algorithm

Our basic problem is to generate a triangulation based on a given mesh of height values. Once the triangulation has been calculated, we can find a height value for any point (x, y) by interpolating between the heights of the corners of the triangle containing (x, y). We can calculate an error metric for our triangulation by finding the difference between the height value in the original mesh for point (x, y) and the interpolated value from the triangulation. Thus, if $I_t(x, y)$ = interpolated value at (x, y) based on corners of triangle containing the point, $z(x, y)$ = value at x, y in the original mesh, and error at $x, y = e(x, y)$, then $e(x, y) = |z(x, y) - I_t(x, y)|$. We make use of this error metric in the following algorithm due to Garland and Heckbert to calculate a triangulation.

function Surface_Model(M, ϵ)
  // M is the input mesh of altitudes
  // ϵ is the bound on the maximum error
  // The Output P is a subset of the original mesh points and its triangulation T
 P = the 4 corners of M
 T = DelaunayTriangulation(P)
 repeat while $(\max(\{e(x, y) : (x, y) \in M\}) > \epsilon)$
  select point p with maximum error
  $P = P \cup \{p\}$
  T = DelaunayTriangulation(P)
 return P and T
end

3.1.4 Finding $\max(e(x, y))$

We use a priority queue to find the max error at each step. The queue contains one entry per triangle, and the entries are ordered on the maximum error for each triangle. For each new triangle, we calculate its maximum error and insert it into the queue. Note that for each iteration i, there are $4 + i$ points and $2i$ triangles.

3.1.5 Running Time

Assume there are n points in the original mesh and m points in the final solution ($m \leq n$). We can use the incremental method to add a new point to the Delaunay triangulation on each step (we need not start from scratch). Then the times for the i^{th} iteration are

	"expected"	worst case
find maximum error :	$O(\log(i))$	$O(\log(i))$
modify Delaunay triangulation :	$O(1)$	$O(i)$
find errors for each point in new triangles :	$O(n/i)$	$O(n)$

and the total times are

Total (expected) :

$$\sum_{i=4}^{m} \log(i) + n/i = O((n+m)\log(m))$$

Total (worst case) :

$$\sum_{i=4}^{m} \log(i) + n = O(nm)$$

Note that for the "expected time" for finding the errors for each point in new triangles all of the triangles are assumed to be approximately the same size, so for n points and i triangles, each triangle will cover n/i of the points.

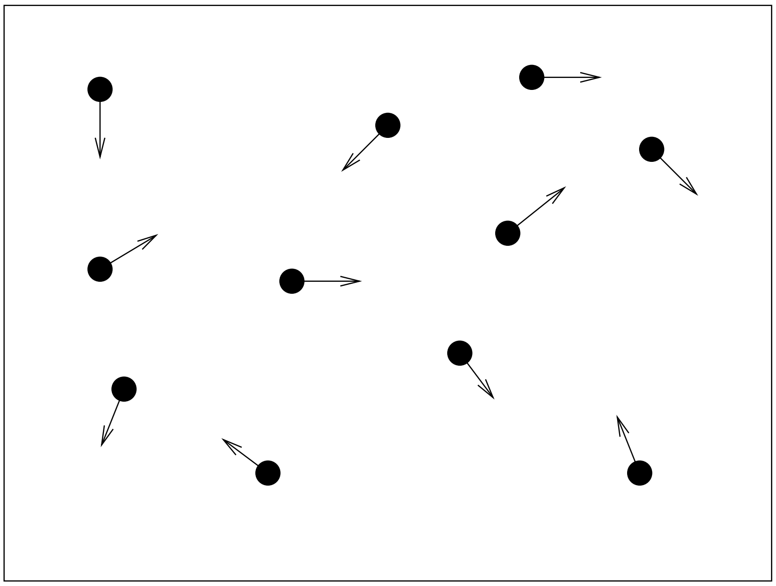
 Notes by Tajh Taylor

- Delaunay Triangulation - more applications
- N-Body Problems
  1. Introduction
     - Basic Assumptions
     - Applications
     - History of Algorithms
  2. Mesh-based Algorithms
     - Particle mesh (using FFT)
  3. Tree-based Algorithms
     - Barnes-Hut Algorithm
     - Fast Multipole Method

1 Delaunay Triangulation - more applications

- Studying structure of galaxies
  use Euclidean minimum spanning tree to study clustering characteristics
- Prediction of protein folding
  run statistics on triangulated protein structures
- Studies of structural damage in metals
  e.g. study growth of "voids"
- Animal and plant ecology
- Human space patterns
  "central space" theory

2 N-Body Introduction



Particles in motion

The goal of N-Body problems is to determine the motion over time of bodies (particles) due to forces from other bodies. Typical calculated forces include electrostatic force and gravity. The basic method used for solving the problem is to loop forever, stepping discretely through time and doing the following at each timestep:

- Update positions using velocities ($\vec{x}_{i+1} = \vec{x}_i + \Delta t \vec{v}_i$)
- **Calculate forces** $\vec{F}$
- Update velocities ($\vec{v}_{i+1} = \vec{v}_i + \frac{1}{m}\Delta t \vec{F}_i$)

Our focus will be on methods for calculating the forces. Note that it is possible to use multiple resolutions for time, i.e. different Δt for different particles, but we will only consider uniform Δt.

2.1 Basic Assumptions

We will consider the particles to be in three-dimensional space, and we will use the following definitions:

$$\begin{array}{lrcll} \text{Potential} & \phi(\vec{x}) & \propto & \frac{1}{r} \text{ (for 1 body)} & \\ \text{Coulomb Force} & \vec{F} & = & m_2 \nabla \phi(\vec{x}) & \propto \frac{1}{r^2} \end{array}$$

In general the potential $\phi(\vec{x})$ is a continuous function over space, and if it is known everywhere, we can calculate forces on all the bodies by taking the gradient.

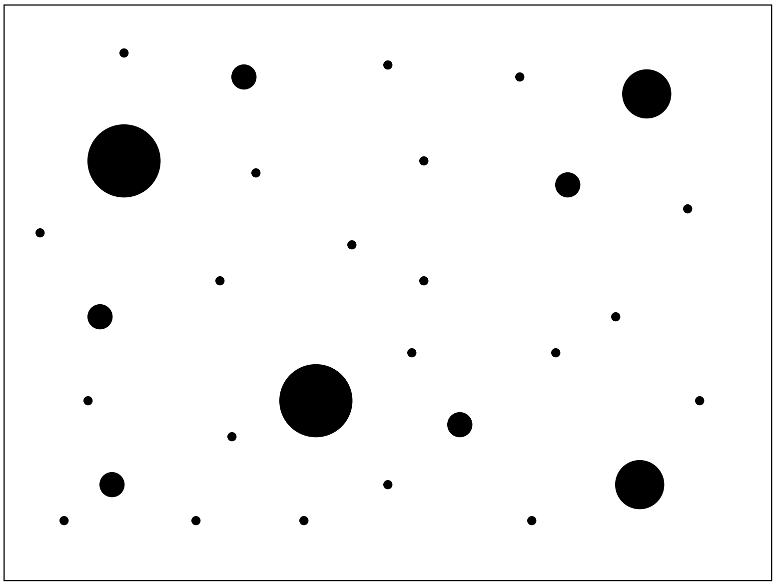


Figure 84: An area over which the potential $\phi(\vec{x})$ is defined. The balls represent different sized particles.

2.2 Applications

Astronomy : Formation of galaxies

- Questions:
  Is the universe expanding?
  How much "dark matter" is out there?
- Forces are gravitational
- State of the art simulations:
  $\approx 10^7$ bodies running for 10^3 steps.
  1 week on 500 processors (1994).

Biology/Chemistry: Molecular Dynamics

- Electrostatic + other forces
  (Bond forces, Lennard-Jones potential)
- State of the art (protein folding):
  10^6 bodies for 10^4 timesteps
- Note that it is easier to use a cutoff radius for limiting electrostatic calculations, because distant charges cancel each other out.

Others

- Plasma Physics

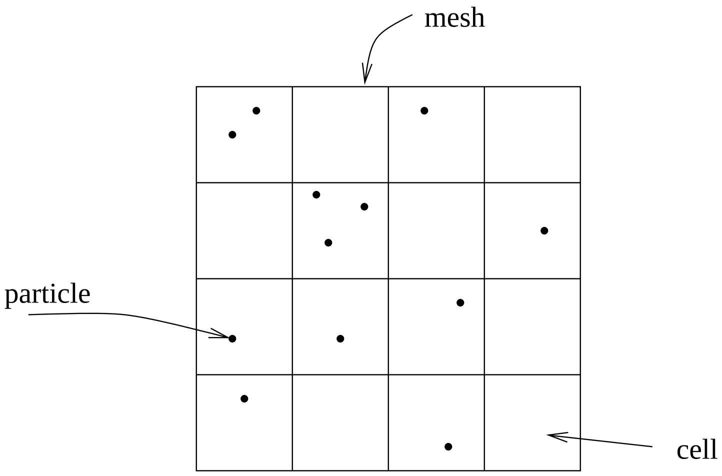


Figure 85: Mesh-based N-body methods

- Partial Differential Equations
- Fluid Dynamics – "vortex particles"

2.3 History of Algorithms

2.3.1 Particle-Particle (the naive method)

This method simply calculates the force between all pairs of particles.

- The time required to calculate all interactions is $O(n^2)$.
- This is inefficient for systems with 10^6+ bodies.

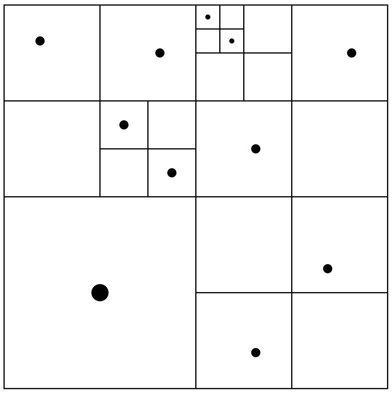
2.3.2 Mesh Based Methods

Particle Mesh (PM): Breaks space into a uniform mesh and places each particle in the appropriate cell of the mesh. Uses a discrete Fourier transform (DFT) to solve a partial differential equation (PDE) over the mesh.

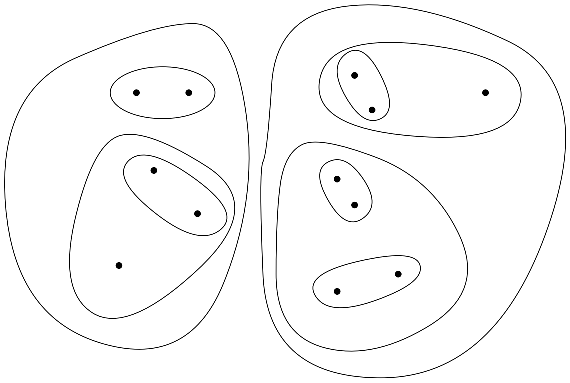
- Invented in the 1970s.
- For m cells, $O(m \log m)$ time is required.

Particle-Particle Particle-Mesh (P3M): A combination of direct particle-particle interactions for close particles and the particle-mesh method for distant particles.

- Invented in the early 1980s.
- Provides better accuracy for the same time as PM.



(a) A quadtree in a Top-down method



(b) Nearest neighbor groupings in a bottom-up method

Figure 86: Tree-based N-body methods

Nested-Grid Particle-Mesh (NGPM): Uses a nested mesh with different "resolutions".

- Invented around 1988.
- Is related to multigrid methods.

2.3.3 Tree Based Methods

Top-down particle-cell: Divides space into a quadtree (2 dimensional) or octree (3 dimensional). When a cell (inner node of the quad or octtree) is far enough away from a particle we can calculate the force to the center of mass of the cell instead of having to calculate the force to each particle in the cell. For example, in Figure 86 (a) the particle in the upper left could group the two particles in the lower right together and calculate the force to their center-of-mass.

- Appel (1985) produced the first tree-based solution.
- Barnes-Hut refined it in 1986.
- For n particles, $O(n \log n)$ time is required (under certain assumptions).

Bottom-up particle-cell: Similar to the top-down method but the tree is created bottom-up by grouping neighboring particles or cells (see Figure 86 (b)).

- Press produced an algorithm in 1986.
- $O(n \log n)$ time is required (under certain assumptions).

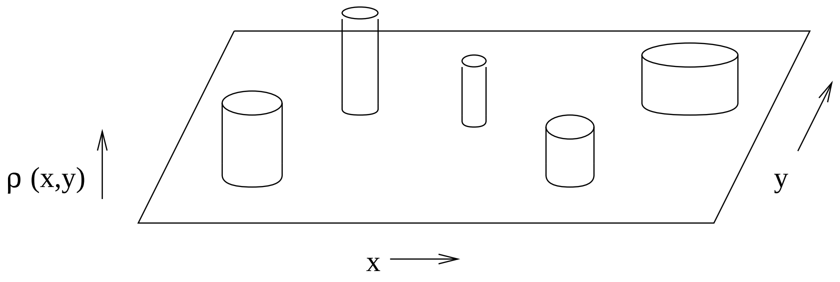


Figure 87: Visualization of the mass density function.

Cell-cell (Fast multipole method, FMM): Also uses a tree but allows for "cell-cell interactions" as well as "particle-cell interactions".

- Incorporates four main ideas: multipole expansions, error bounds on dropping terms, dual expansion, and translation of expansions.
- These are the fastest known methods for solving N-Body problems.
- Done by Greengard in 1988.
- $O(n)$ time is required for uniform distributions. Non-uniform distributions may increase the required time.

Tree-code particle-mesh: Mixes ideas from particle-mesh methods and tree-based methods.

- Work by Xi in 1989.

3 Mesh-based Algorithms

The goal is to solve the following PDE (Poisson's equation)

$$\nabla^2\phi(\vec{x}) = 4\pi G\rho(\vec{x})$$

where $\rho(\vec{x})$ is the "mass density" (Figure 87 shows an example). The solution to this equation gives the potential over space, which can then be used to calculate the force on each particle. The Poisson equation is the same as used for heat flow, or fluid flow without compression or viscosity. Poisson's equation can be solved by various methods, including spectral methods (DFT), finite element, or finite volume methods. Here we will discuss using spectral methods and the FFT, an approach that has been used extensively in astronomical n-body problem.

Particle mesh (using FFT)

We are given the density function $\rho(\vec{x})$ and want to solve for the potential $\phi(\vec{x})$ in the equation $\nabla^2\phi(\vec{x}) = 4\pi G\rho(\vec{x})$. We first discretize space by breaking it into a uniform mesh. Solving for the potential on this mesh will give us an approximate answer to the continuous solution and as long as the mesh is "fine enough" the approximation will be reasonably good. We will discuss how fine it has to be later.

For simplicity let's start by assuming 1-dimensional space, and let's consider the Discrete Fourier transform of the density function $\rho(x)$ (assuming x goes from 0 to $n-1$ in discrete steps of 1). The DFT is

$$\theta_j = \sum_{x=0}^{n-1} \rho(x) e^{2\pi i x j/n}$$

which transforms our density function into a set of frequency components (i is the imaginary number $\sqrt{-1}$). Since Poisson's equation is linear we can solve for each frequency component separately. Lets use the notation $w_j = 2\pi j/n$ and consider the j^{th} frequency component of the density function: $f_j(x) = \theta_j e^{iw_j x}$. If we plug this component into Poisson's equation, we get

$$\nabla^2 \phi_j(x) = 4\pi G \theta_j e^{iw_j x}$$

The solution to this equation is

$$\phi_j(x) = -4\pi G \theta_j \frac{1}{w_j^2} e^{iw_j x} = -4\pi G \frac{1}{w_j^2} f_j(x)$$

since $\nabla^2 e^{iw_j x} = -w_j^2 e^{iw_j x}$. In other words, to solve for the j_{th} frequency component of the potential ϕ we simply multiply the j_{th} frequency component of the density function by $-4\pi G \frac{1}{w_j^2}$.

We can similarly solve for each of the other frequency components of the potential, then add up the solutions to get the solution to our initial problem. This is equivalent to convolving the frequency component j with $1/w_j^2$ and then transforming back to coordinate space, which can be done with a second DFT. This idea can be applied in multiple dimensions by taking a multidimensional DFT. In this case the solution for a particular component with frequencies w_j, w_k and w_l would be

$$\phi_{jkl}(\vec{x}) = -4\pi G \frac{1}{w_j^2 + w_k^2 + w_l^2} f_{jkl}(\vec{x})$$

where j, k and l are the frequency components in the x, y and z dimensions respectively.

So the overall algorithm in three dimensions has five steps:

- Break space into discrete uniform mesh of points.
- Assign each particle to a point (or set of points) to generate ρ for each point.
- Do a 3-D FFT on the mesh.
- Convolve the results with the sum of inverse square frequencies.
- Do an inverse FFT back to the original mesh points.

This will give $\phi(\vec{x})$ over the mesh from which we can derive the forces. For m mesh points the algorithm will take $O(m \log m)$ time.

The problems with this method is that in order to get reasonable accuracy there should be no more than one particle per cell, and ideally there should be a few cells separating each particle. This works fine if the particles are uniformly distributed, but if the particles are nonuniformly distributed the simulation would require a number of cells that is very much larger than the number of particle. This is because the mesh has to fine enough to accommodate the smallest distance between particles. For sufficiently nonuniform points the number of mesh points required could become greater than n^2 and therefore run even slower than the naive all-pair interactions.

4 Tree-based Algorithms

The approach taken by tree based methods is to approximate far away particles by some sort of center of mass rather than by direct interactions.[1] The further away you are from a set of points, the more you can group together and consider as a center-of-mass. These "centers of mass" are then organized hierarchically (in a tree) so that the approaches can determine which sets of points to approximate by their centers of mass. The root of the tree will include all the particles, and at each node of the tree the children will partition the set of particles of that node, usually by partitioning the space taken by that node. The closer you are to the points, the further down in the tree you will need to go so that you will consider smaller sets of particles.

We will call a set of particles grouped together with a center of mass, a *cell*. In the algorithms we will consider three types of interactions, particle-particle (direct interactions between two particles, particle-cell (interaction between a particle and the center of mass of a cell), and cell-cell (interactions between two centers-of-mass). These are shown in Figure 88.

In this class we will only consider top-down tree-based algorithms.

4.1 Barnes-Hut Algorithm

The first tree-based algorithm we consider is the Barnes-Hut algorithm, which use particle-particle and particle-cell interactions, but no cell-cell interactions. It consists of building a tree using an oct-tree in 3 dimensions or a quad-tree in 2 dimensions, and then using the tree to calculate the force on each particle. Each division evenly splits up the space so, for example, in a quadtree each step will consists of dividing a square into 4 squares. Note that this will not necessarily evenly divide the particles, so the tree in general will not be balanced. The algorithm for building the tree works top down and is defined by the following code

[1]Center of mass is used informally here since later we will show that some of the algorithms approximate a set of particles by more complicated functions.

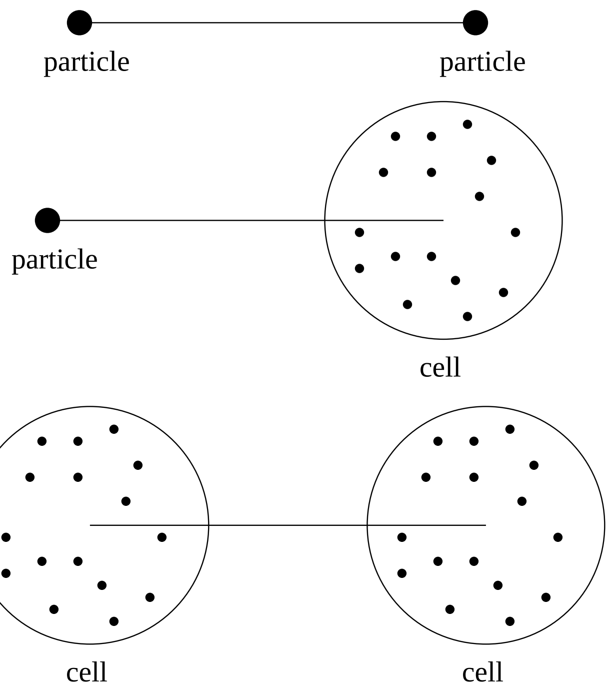


Figure 88: Particle-particle, particle-cell, and cell-cell interactions.

function Build_Tree(P)
  // P is a set of points along with a bounding box
  if Particle_Count(P) $\leq \tau$ // where τ is an integer threshold ($\tau \geq 1$)
    return P
  else
    $(P_1, P_2, P_3, \ldots)$ = partition(P)   // the quadtree or octree division (divides space evenly)
    a = Make_Empty_Tree()
    for $\pi \in (P_1, P_2, P_3, \ldots)$
      a = Add_Child(a, build_tree(π))
    return a

Figure 89 gives an example of a set of points in 2-dimensions along with the tree built by this code. Once the tree has been built the force algorithm is quite simple and is given in Figure 90. The **well_separated** function tells when to approximate the force for a group of particles using the center of mass for the group (see Figure 91). θ is a user-specified parameter that controls the accuracy of the results. A smaller θ will increase accuracy but also increase runtime, and larger θ will decrease accuracy and runtime.

For "reasonable" distributions each particle interacts with a constant number of cells on each level of the tree, and there are $O(\log n)$ levels. Therefore each particle takes $O(\log n)$ time and the total time for the Barnes-Hut algorithm is $O(n \log n)$. One can show, for example, that for uniform distributions in 3-d:

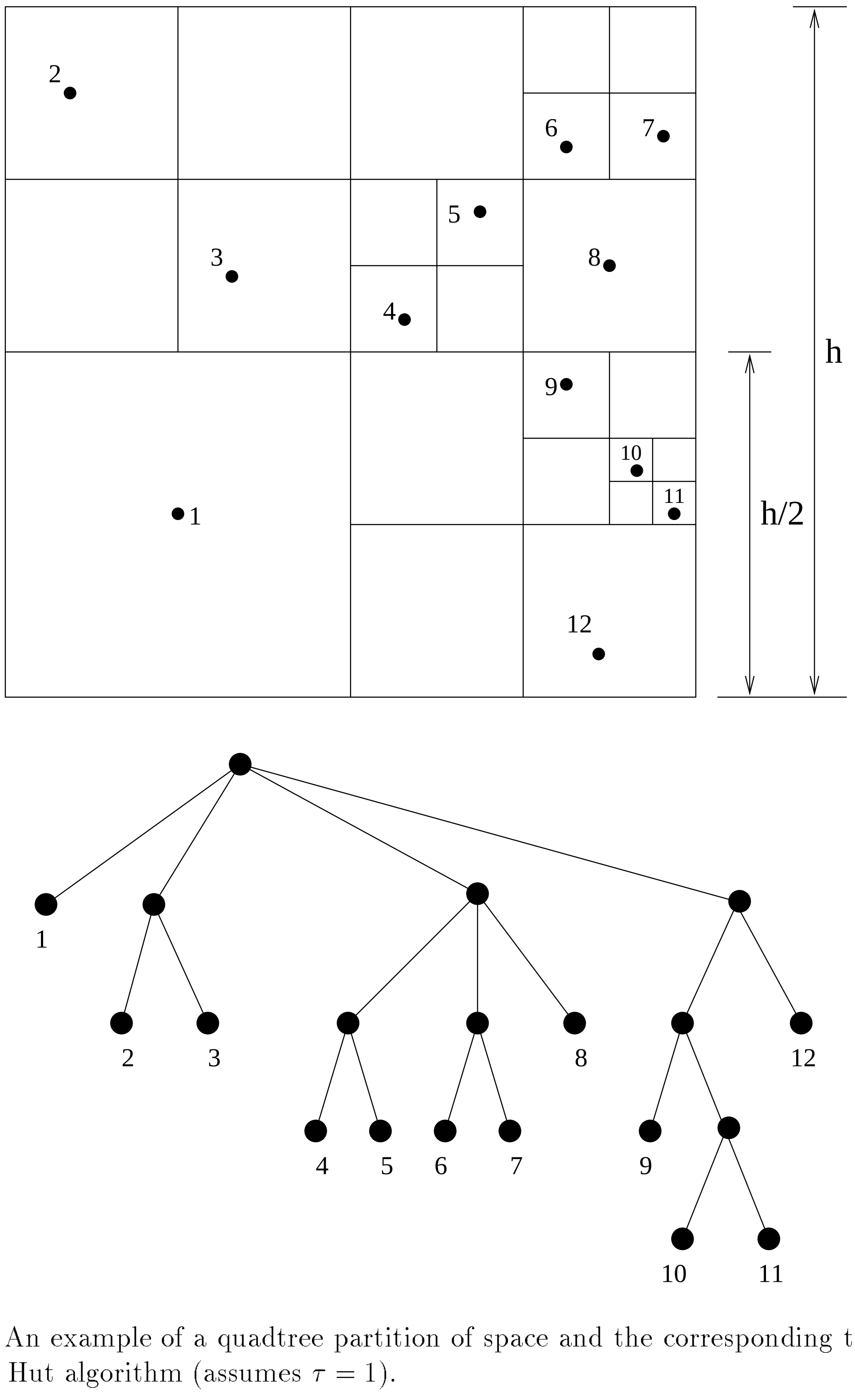


Figure 89: An example of a quadtree partition of space and the corresponding tree used in the Barnes Hut algorithm (assumes $\tau = 1$).

```
function Barnes_Hut(P)
    T = Build_Tree(P)
    r = root of T
    for p ∈ P
        F⃗_p = Calculate_Force(p, r)

function Calculate_Force(p, c)
    // p is a particle
    // c is a node of the tree, which is either a particle or cell
    if Is_Particle(c)
        if p = c return 0              // don't include the point itself
        else return Force(p, c)    // particle-particle force calculation
    if Well_Separated(p, c)
        return Force(p, Centermass(c))     // particle-cell force calculation
    F⃗ = 0
    for c′ ∈ children of c
        F⃗ += Calculate_Force(p, c′)
    return F⃗

function Well_Separated(p, c)
    return s(c) < θ · d(p, Center(c))
```

Figure 90: Code for the Barnes-Hut algorithm. Assumes the leaves of the tree are individual particles, *i.e.*, $\tau = 1$

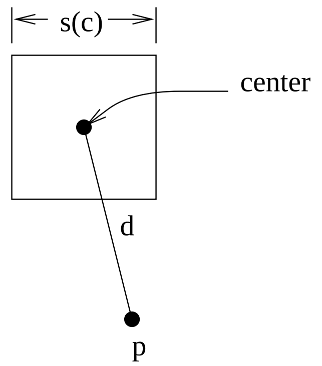


Figure 91: Well_Separated calculation on a cell c and particle p.

$$\text{Total Interactions: } I(n, \theta) \approx \frac{28\pi}{3\theta^3} n \log_8(n)$$

For $\theta = .3$, which is typically required for good accuracy, this gives $\approx 350n \lg n$. Using a very rough calculation (we are dropping some lower order terms) this outperforms calculating all pairs interactions when $350n \lg n \leq n^2/2$, which is when $n > 10000$. At $\theta = .5$, which is the about the highest used, the tradeoff is at around 2000 particles. Barnes-Hut is therefore not useful for a small number of particles. It, however, can do much better than the all-pairs algorithm for larger n.

Exercise: For a million particles and $\theta = .3$ about how much faster is Barnes-Hut over using all pairs?

The Barnes-Hut algorithm is highly parallelizable since the loop over the particles once the tree is built is completely parallel, and it is not too hard to build the tree in parallel. There have been many parallel implementations of the Barnes-Hut algorithm.

4.2 Fast Multipole Method

This method was pioneered by Greengard and Rokhlyn in 1987, and it won an ACM Thesis Award for Greengard. It was the first linear-time algorithm for N-Body problems (for uniform or near uniform points). They were also the first to have proven error bounds.

The method has four main ideas in it, each of which will be covered in more detail:

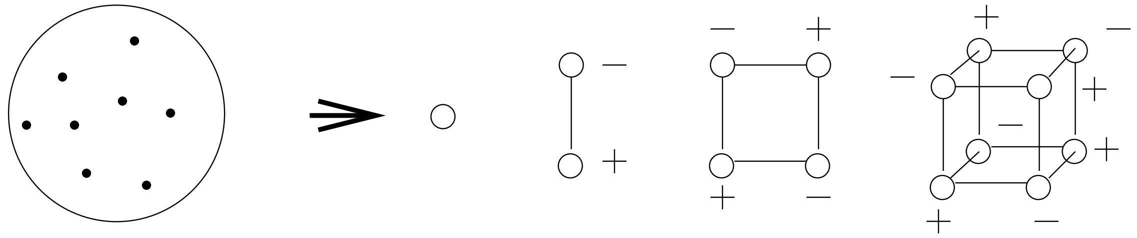
- Multipole expansions
- Error bounds on dropping terms
- Dual local expansion
- Translation of expansions

For nonuniform points, FMM can take $O(n \log n)$ time to find a set of "well-separated" points, but the force calculation is still $O(n)$.

Idea 1: Multipole expansions

For any cluster of particles, their net force can be approximated using a multipole expansion (see Figure 92) which is taken around a fixed point. The first term of the expansion is often called the monopole term, the next group of 3 terms are called the dipole terms, and the next group of 5 independent terms are called the quadrapole terms.

The multipole expansion is analogous to a Taylor-series expansion, but in 3d. Like the Taylor-series expansion the earlier terms have a larger magnitude when far enough from the origin, and the function can be approximated by dropping higher order terms. In the case of multipole-expansions the force and potential due to higher-order terms drops off faster as a function of r (see the last two rows Figure 92), which is what allows us to drop these terms far from the center.

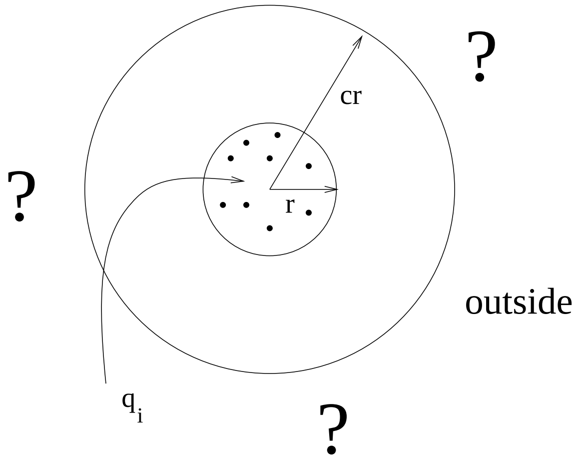


p	1	2	3	4
Terms	1	3	5	7
Potential	$\frac{1}{r}$	$\frac{1}{r^2}$	$\frac{1}{r^3}$	$\frac{1}{r^4}$
Force	$\frac{1}{r^2}$	$\frac{1}{r^3}$	$\frac{1}{r^4}$	$\frac{1}{r^5}$

Figure 92: Multipole expansion.

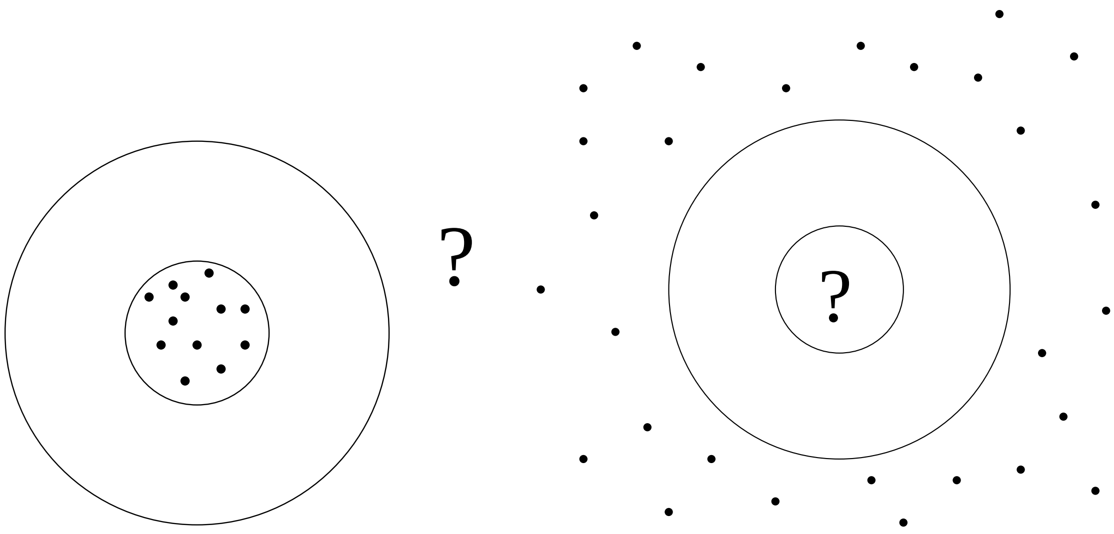
Idea 2: Error Bounds

For distributions with only positive sources (*e.g.*, gravitation) we can bound the ratio of the force due to higher terms of the potential relative to the first term, and therefore bound the error due to dropping terms after a given p. This is possible at a sufficient distance from the particles since the higher terms diminish faster than the first term (see the third row in Figure 92). To see this consider the following diagram:



Lets define $M = \sum q_i$ as the sum of the masses inside the sphere defined by r. A lower bound on the force due to the monopole of these masses (first term of the expansion) on a mass m_1 on the outer sphere is

$$F_1 \geq \frac{m_1 M}{(((c+1)r)^2)}$$



(a) multipole expansion approximation (b) local expansion approximation

Figure 93: Multipole (outer) and local (inner) expansions.

Similarly an upper bound on the force due to the other multipole terms on m_1 is

$$F_p \leq \frac{r^{p-1} m_1 M}{((c-1)r)^{p+1}} .$$

With these bounds we can then bound the ratio of the contribution of the p^{th} term to the first term by

$$E_p = \frac{F_p}{F_1} \leq \frac{(c+1)^2}{(c-1)^{p+1}}$$

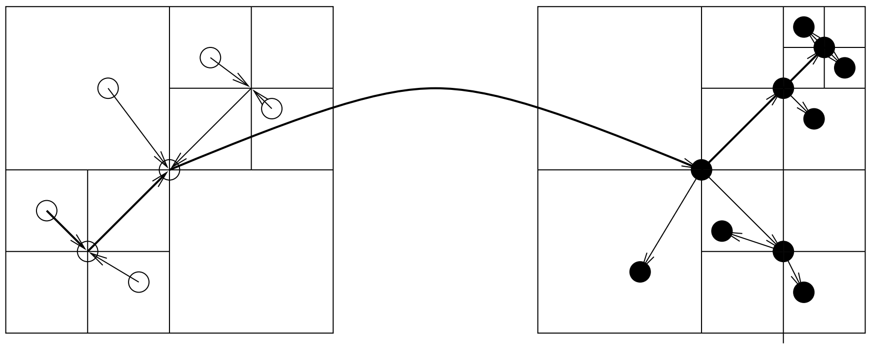
Assuming c is greater than 2, this bound decreases exponentially with p and allows us to bound the number of terms p we need for a given accuracy (we can actually get tighter bounds if we are more careful). It also shows how having larger c gives better error bounds. As we go outside the outer sphere our approximation can only get better.

Idea 3: Dual Expansion

Our previous multipole expansion assumed we know the sources inside an inner sphere and are trying to approximate the potential (or forces) due to them outside an outer sphere (see Figure 93(a)). There is also a dual expansion in which the sources are known outside the outer sphere and we want an approximation the potential contribution due to them inside the inner sphere (see Figure 93(b)). These dual expansions turn out to be very important in the Fast Multipole Method. We will refer to them as the *local* or *inner* expansion. We will refer to our original expansion as the *multipole* or *outer* expansion.

Idea 4: Translation of Expansions

The final idea is that the expansions can be translated from one center to another. Furthermore we can translate between the two different kinds of expansions. This is illustrated in



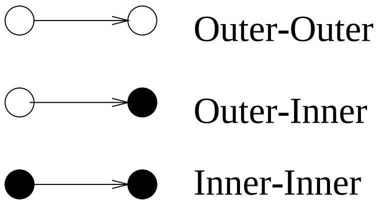


Figure 94: Example translations

Figure 94. In this figure we are trying to approximate the forces due to the particles in the left box on the particles in the right box. We first combine the outer (multipole) expansions on the left in a tree like fashion by translating each expansion to its parent in the tree and summing the children into the parent. We then translate the total outer expansion to an inner (local) expansion in the right box. Finally we translate this inner expansion down the tree to each of the locations on the right. The running time for each translation is dependent on the number of terms. With a naive implementation each translation can run in $O(p^4)$ time but this can be reduced to $O(p^2 \log p^2)$ with a FFT-style translation.

The translation of the outer expansion to an inner expansion in another box is called a cell-cell interaction. This is because we are effectively having the cells interact rather than having particles interact directly (note that in the example of Figure 94 no particle on the right interacts directly with a particle on the left).

Why cell-cell interactions?

To motivate why cell-cell interactions and both forms of expansions are important consider Figure 95. In the example the goal is to determine the potential (or force) on each particle due to all other particles outside of its own cell. To do this we will use expansions for each cell. We first consider the cost (number of interactions) when using just particle-cell interactions and then show that we can do much better by using cell-cell interactions.

Particle-Cell Cost

- $\sqrt{n} \times \sqrt{n}$ to calculate multipole expansions within each cell each cell
- $n \times \sqrt{n}$ for particle-cell interactions (each particle has to look at the other $\sqrt{n}$ cells.

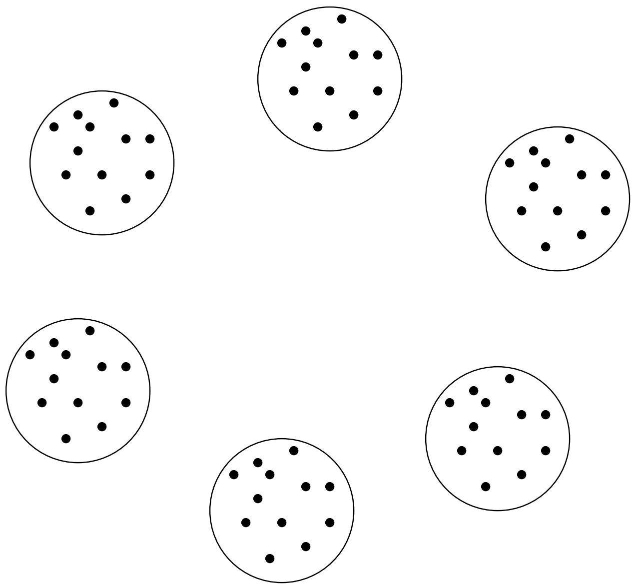


Figure 95: Example cell-cell interactions. Assume there are n total particles with $\sqrt{n}$ particles within each cell (circle) and that the cells are well separated from each other.

- Total $= O(n^{3/2})$

Cell-Cell Cost

- $\sqrt{n} \times \sqrt{n}$ to calculate multipole expansions within each cell
- $\sqrt{n} \times \sqrt{n}$ for outer-inner (multipole-local) translations — every cell translates to every other cell and we add up the contribution at each cell
- $\sqrt{n} \times \sqrt{n}$ to translate the local expansion from the center-of-mass of each cell to each of its $\sqrt{n}$ elements.
- Total $= O(n)$

FMM: Using a tree

We now consider putting the ideas together into the Fast Multiple Algorithm. There are three phases, each which uses a tree.

- Bottom up: forming multipole expansions
  - outer-outer translation from children to parents
- Top down: forming local expansions
  - inner-inner translation from parent to child
  - outer-inner translation from interaction list

- Direct interactions with neighbors at leafs

For a parameter k and for cells on the same level (i.e. of the same resolution), we make the following definitions.

Definition: two cells are *near neighbors* of each other if they are within k cells of each other. Note that two cells with touching corners are considered to have a distance of one.

Definition: two cells are *well separated* if they are not near neighbors.

Definition: a cell c_2 is in c_1's *interaction list* if

1. well-separated(c_1, c_2) and
2. near-neighbor$(p(c_1), p(c_2))$

where $p(c)$ indicates the parent of c.

In practice $k = 2$, and in 3-dimensional space the size of the interaction lists is then 875. In general it is not hard to show that the size of an interaction list is $(2^d - 1)(2k + 1)$.

Total Time (for a uniform distribution)

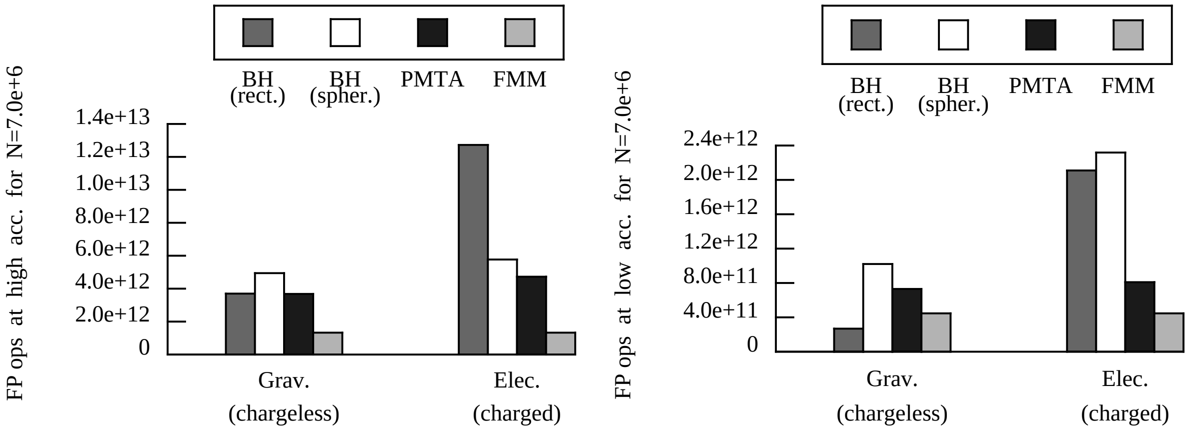
Assume $k = 2$. Place p^2 particles per leaf cell (n/p^2 leaf cells).

Assuming translations take p^4 time:

$$\begin{array}{ll}
\text{Leaf multipoles} & p^4\left(\dfrac{n}{p^2}\right) \\
\text{Up Tree} & \displaystyle\sum_{l=1}^{d} 8^{l+1} p^4 \\
\text{Down Tree} & \displaystyle\sum_{l=1}^{d} (875 \cdot 8^l p^4 + 8^l p^4) \\
\text{Local expansion} & p^4\left(\dfrac{n}{p^2}\right) \\
\text{Total Time} & = O(np^2) \\
\text{Error Bound} & \approx 2.6^{-p}
\end{array}$$

There is a clear tradeoff between time and accuracy.

Figure 96 shows a comparison of runtimes for the various algorithms for two different accuracies. The BH (rect.) is a variant of Barnes-Hut that uses multipole expansions in rectilinear coordinates. The BH (spher.) is a a variant of Barnes-Hut that uses multipole expansions in spherical coordinates. The PMTA is an algorithms that combines the ideas of Barnes-Hut and FMM. Comparisons are given for sets of particles that all have the same charge (chargeless) and for particles which have both positive and negative charges (charged).



(a) high accuracy execution times (b) low accuracy execution times

Figure 96: Runtimes for the various algorithms.

- N-body continued: Callahan & Kosaraju Algorithm
- Fast Multipole Method (Review)
  1. The Four Ideas
  2. Interaction Lists
- Callahan and Kosaraju's Algorithm
  1. Well Separated Pair Decompositions
  2. Definitions
  3. Algorithm: Tree Decomposition
  4. Algorithm: Well-Separated Realization
  5. Bounding the size of the realization
- N-body Summary

1 Fast Multipole Method (Review)

1.1 The Four Ideas

The four main ideas being the fast multipole method (FMM) were:

- **Multipole expansion** - a Taylor-series like expansion of the potential due to a cluster of particles around an origin. The expansion converges at points outside the cluster so that the potential can be approximated by only keeping the first p terms. We call this the outer or multipole expansion.
- **Error bounds on dropping terms** - the number of terms in the expansion dictates the accuracy of the expansion
- **Dual local expansion** - an expansion of the potential due to particles outside a cluster which converges for points inside the cluster. We call this the inner or local expansion.
- **Translation of expansions** - being able to translate a inner or outer expansion to a new origin, or translate an outer to an inner expansion at a different origin.

A fifth idea (Figure 97) is to use a tree-structure to organize the particles and to determine a strategy for grouping the particles and performing the outer/outer, outer/inner, and inner/inner translations. We call each node of the tree a *cell.*

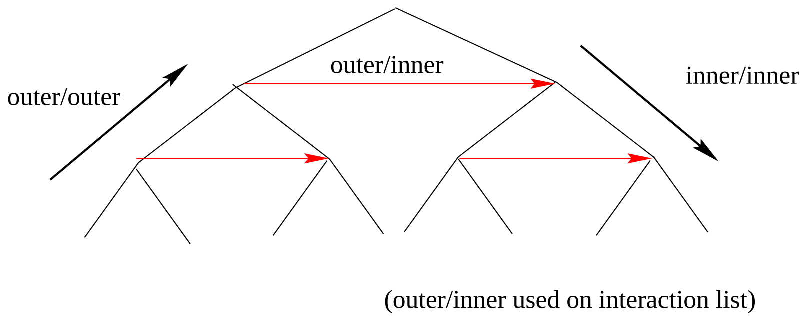


Figure 97: Fifth idea - using a tree for translations

1.2 Interaction Lists

In the fast multipole method, the interaction list (Figure 98) dictates the particles that a given cell will interact with (cross edges in Figure 97). The *interaction list* for a cell c consists of all the cells at the same level of the tree which (1) are well-separated from c and (2) are children of cells that are not well-separated from the parent of c. In Figure 98, for example, we assume a cell is well separated if it is 2 cells away. Clearly all the outer shaded cells are 2 away from the center cell c and are therefore well-separated. Also the parents of these cells are not well-separated from the parent of c (note that parent cells are marked with the thicker lines). One issue is how to find particles in the interaction list. Here are two possible methods:

1. **Uniform distributions:** a hierarchy of meshes can be used in this situation. Simply look at neighbors at an appropriate resolution in the mesh hierarchy.

2. **Non-uniform distributions:** a mesh hierarchy may be very sparsely populated, so a Barnes-Hut tree-like structure may be more appropriate in this case.

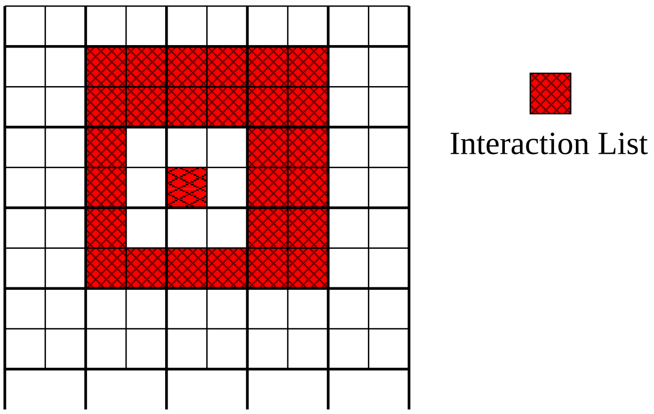


Figure 98: The Interaction List

2 Callahan and Kosaraju's Algorithm

We now discuss an algorithm due to Callahan and Kosaraju for generating a tree and interactions-lists for non-uniform distributions ("A Decomposition of Multi-Dimensional Point-Sets with Applications to k-Nearest-Neighbors and n-Body Potential Fields.", JACM, 42(1), January 1995). The algorithm has the nice property that it is very simple, although the proof of its bounds is a little involved. The particular tree-structure it creates is called a well-separated pair decomposition. It uses a somewhat more general definition of interaction lists than used in the previous section (for example, it allows interactions between cells on different levels). The algorithm works in any dimension, requires at most $O(n \log n)$ time to build the tree for n particles and guarantees that there will be a total of $O(n)$ interactions (size of the interaction lists summed over all cells).

2.1 Well Separated Pair Decompositions

The tree structure used by the algorithm is based on the notion of a tree decomposition of points. Given the set of points {a, b, c, d, e, f, g}, a *tree decomposition* of these points repeatedly splits this set into subsets, as in Figure 99.

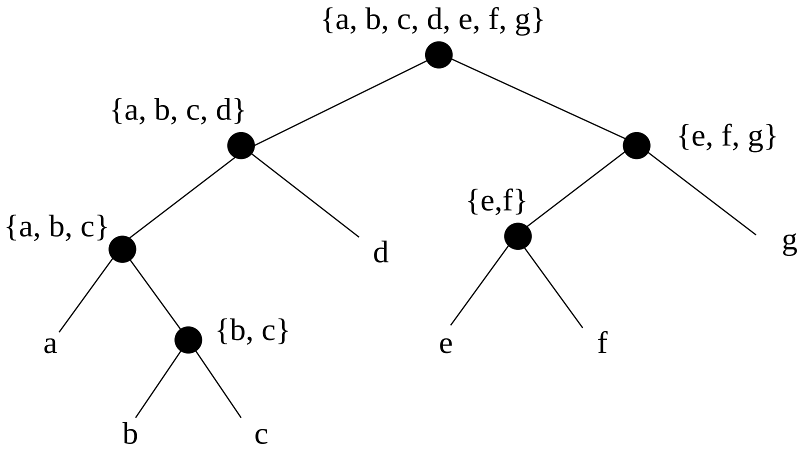


Figure 99: A tree decomposition D of points.

A *well-separated realization* of a tree decomposition is a set of interaction edges between vertices (cells) in the tree (edges represent outer/inner translations) such that (1) the cells at the two ends of the edge are well-separated (we will define well-separated later in these notes) and (2) there is exactly one path between any two leaves that goes up the tree any number of levels (including 0), across a single interaction-edge, and back down the tree any number of levels. The second property guarantees that each particle will interact exactly once with every other particle. Figure 100 shows an example of a well-separated realization for the tree decomposition given in Figure 99. You can verify that there is exactly one path between every leaf. We will define well-separated realizations more formally in Section 2.2.

Together the tree-decomposition and the well-separated realization on that tree are called a *well separated pair decomposition*. The Callahan-Kosaraju algorithm consists of generating

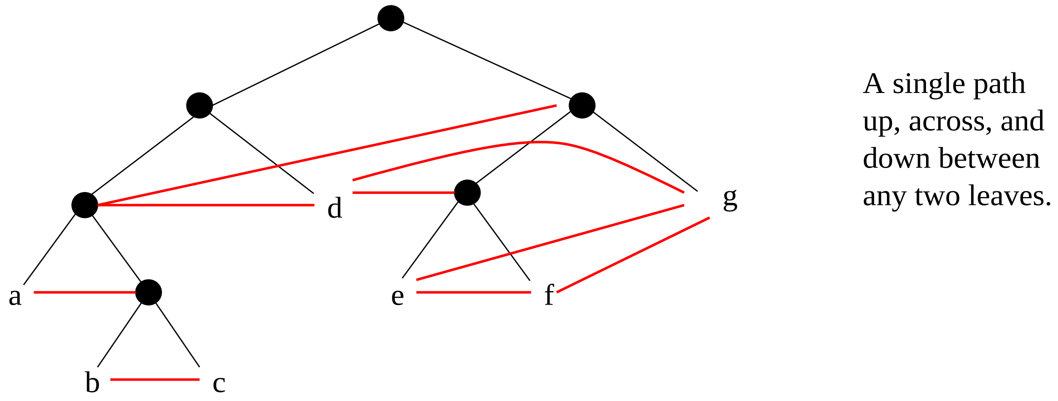


Figure 100: A well-separated "realization" of D.

the tree and the realization in two steps. We will see that we can build the tree-decomposition in $O(n \log n)$ time and find the well-separated realization given the tree in $O(n)$ time. Furthermore we can bound the size of the well-separated realization (total number of interaction edges) by $O(n)$.

The well-separated pair decomposition has applications both to the N-body problem and to computational geometry. For the N-body problem the tree is used for outer/outer translations (going up the tree) and inner/inner translations (going down the tree), and the well-separated realization is used for outer/inner translations. In computational geometry the well-separated pair decomposition can be used to find the k-nearest-neighbors for each point. This requires some augmentation to the tree and is discussed in the Callahan and Kosaraju paper.

For the N-body problem the time taken by one step of building the tree, finding the interaction edges, and doing all the translations to calculate the forces is $k_1 n \log n + k_2 n$, where the first term is for building the tree and the second linear term is for finding the interaction edges and translating the expansions. In practice the constant k_2 is very much larger than k_1. Furthermore in many cases it is possible to use the same tree (or only slightly modified tree) for multiple steps. Therefore from a practical standpoint the algorithm runs in almost linear time.

2.2 Definitions

- **Bounding Rectangle** $R(P)$: the smallest rectangle that contains points P and is aligned with the axes (Figure 101).

- $l_{\max}$: maximum length over the dimensions of a rectangle. $l_{\max}(P)$ is the maximum length of $R(P)$.

- **Well-separated**: Given two rectangles, let r be the smallest radius that contains them both (see Figure 102). For a given *separation constant* s two rectangles are well separated if $d > sr$.

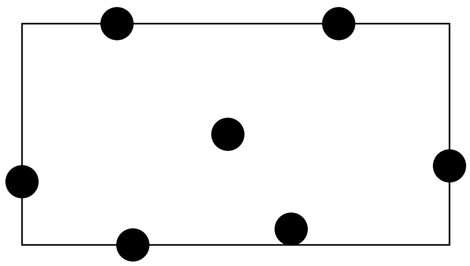


Figure 101: Bounding Rectangle

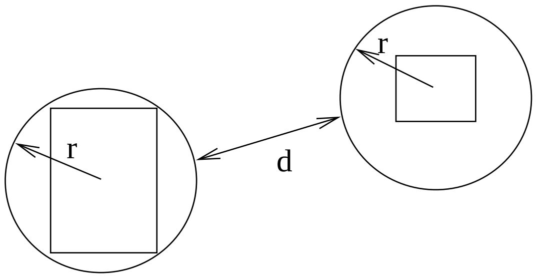


Figure 102: Well Separated

- **Interaction Product** $A \otimes B$:

$$A \otimes B = \{\{p, p'\} | p \in A, p' \in B, p \neq p'\}$$

- **Realization of** $A \otimes B$ is a set $\{\{A_1, B_1\}, \dots \{A_k, B_k\}\}$ such that:

  1. $A_i \subseteq A, B_i \subseteq B$ for all $i = 1 \dots k$
  2. $A_i \cap B_i = \emptyset$
  3. $(A_i \otimes B_i) \cap (A_j \otimes B_j) = \emptyset$ for all $i \neq j$, *i.e.*, there are no repeated interactions
  4. $A \otimes B = \cup_{i=1}^{k} A_i \otimes B_i$, *i.e.*, all interactions are included.

  This formalizes the interaction edges in the tree and guarantees that there is only one path between two leaves. In particular rule 2 guarantees that no leaf interacts with itself, rule 3 guarantees that no leaf interacts more than once with another leaf and rule 4 guarantees that every leaf interacts with every other leaf.

- **A well separated realization** is a realization such that for all i, A_i and B_i are well separated.

- **A well-separated pair decomposition** is a tree decomposition plus a well-separated realization $P \otimes P$ where subsets are nodes of the tree (Figure 103).

2.3 Algorithm: Tree Decomposition

The algorithm partitions space by recursively cutting bounding rectangles in half along their longest dimension. It can be defined as follows:

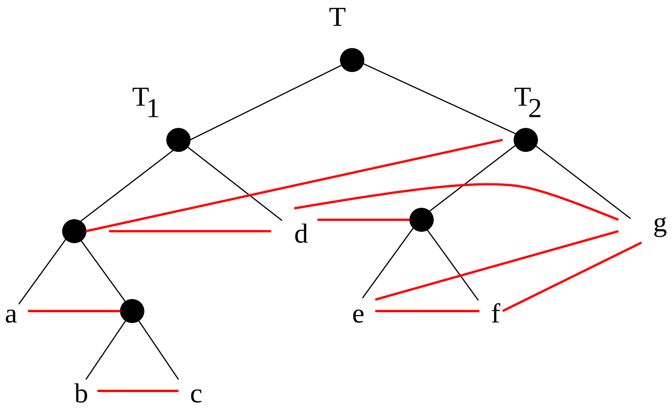


Figure 103: Subsets are nodes of the tree

function Build_Tree(P)
  // P is a set of points
  if $|P| = 1$ **then return** Leaf(P)
  else
    S = plane that splits $R(P)$ perpendicular to its longest dimension
    P_1, P_2 = Split P by the plane S
    return Tree_Node(Build_Tree(P_1), Build_Tree(P_2), $R(P)$)
end

An example of the algorithm on a set of 8 points is given in (Figure 104). The algorithm

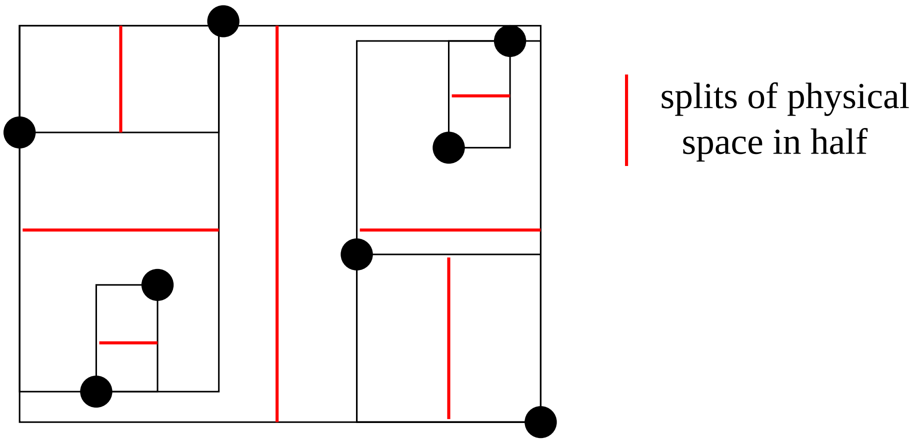


Figure 104: Building the tree

returns a tree with the bounding rectangle stored at each internal node and the original points at the leaves. The bounding rectangles are needed for building the well-separated realization (*e.g.* to determine when nodes are well separated).

For uniformly distributed sets of points the Build_Tree algorithm will run in $O(n \log n)$ time without any special tricks since finding the bounding rectangle and splitting P take $O(n)$ time and the two recursive calls will be on sets of approximately the same size (*i.e.*, we have a standard $T(n) = 2T(n/2) + O(n)$ style recurrence). Furthermore the algorithm is highly parallel since the two recursive calls can be made in parallel and the data can be split in parallel In fact for most "practical" sets of points this will be true.

If the cut does not evenly divide the points, however, the naive algorithm could take $O(n^2)$ time. For example, if there is a sequence of points that are exponentially approaching a fixed point. In this case the algorithm can be slightly modified so that it still runs in $O(n \log n)$ time. This modified version is outlined as follows.

1. Keep the points in linked lists sorted by each dimension
2. In the selected dimension, come in from both sides until the cut is found
3. Remove cut elements and put aside
4. Repeat making cuts until the size of the largest subset is less than $(2/3)n$
5. Create subsets and make recursive calls.

The runtime for building the tree is thus specified by

$$T(n) = \sum_{i=1}^{k} T(n_i) + O(n)$$

such that $\sum_{i=1}^{k} n_i = n$ and $\max_{i=1}^{k}(n_i) = \frac{2}{3}n$.

This solves to $O(n \log n)$ independent of how inbalanced the tree ends up being. A problem is that it does not parallelize easily. Callahan and Kosaraju show how it can be parallelized, but the parallel version is even more complicated and is not work-efficient (the processor-time product is $O(n \log^2 n)$). We will not cover this complicated parallel version since in practice parallelizing the simple version is both more efficient and likely to be highly effective.

2.4 Algorithm: Well-Separated Realization

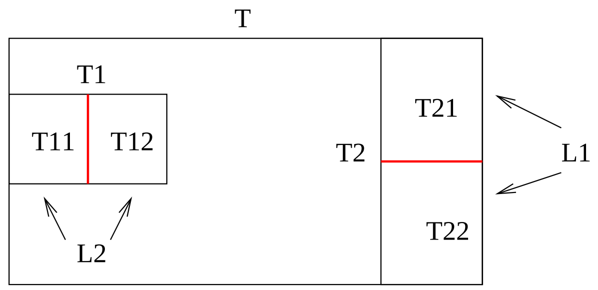
Once we have a decomposition tree we can use it to build the well-separated realization. The following algorithm returns the set of interaction edges in the well-separated realization. The basic idea is to first generate interaction edges within the left and right children (recursive calls to WSR) and then generate interaction edges between the two children (call to WSR_Pair).

function WSR(T)
  // T is a decomposition tree
  if Is_Leaf(T) **return** $\emptyset$
  else
    return WSR(Left(T)) $\cup$
        WSR(Right(T)) $\cup$
        WSR_Pair(Left(T),Right(T))
end

function WSR_Pair(T_1, T_2)
  // T_1 and T_2 are two disjoint decomposition trees
  if Well_Separated(T_1, T_2) **return** {Edge(T_1, T_2)}
  else
    if ($l_{\max}(T_1) > l_{\max}(T_2)$)
      return WSR_Pair(Left(T_1), T_2) ∪ WSR_Pair(Right(T_1), T_2)
    else
      return WSR_Pair(T_1, Left(T_2)) ∪ WSR_Pair(T_1, Right(T_2))
end

The idea of WSR_Pair(T_1, T_2) is to return an interaction edge if T_1 and T_2 are well separated. Otherwise it picks the node with a larger $l_{\max}$ and splits that node and calls itself recursively on the smaller and the two halves of the larger.

Figure 105 gives an example of the algorithm along with call tree for WSR_Pair(T_1, T_2). In the example the algorithm determines that $l_{\max}(T_2) > l_{\max}(T_1)$ and therefore splits T_2 in the recursive calls. At the next level it splits T_1, at which point it determines that T_{11} is well separated from T_{21} and therefore makes no more recursive calls on that branch.



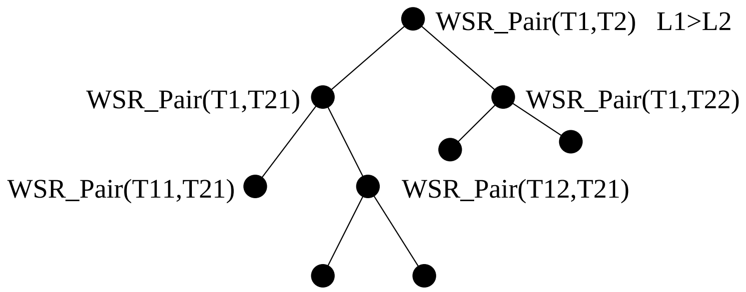


Figure 105: Generating the Realization. The tree represents the call tree for WSR_Pair.

2.5 Bounding the size of the realization

By bounding the size of the realization we both bound the number of interaction edges and also bound the runtime for building the realization since the runtime of WSR is proportional to the number of edges it creates.

We will use the following to bound the total number of interaction edges generated by WSR (size of realization):

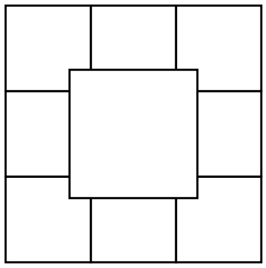


Figure 106: Non-intersecting rectangles

- Properties of the tree nodes - show bounding rectangles are not "too thin"
- Bound # of non-intersecting rectangles that can touch a cube of fixed size (Figure 106). This assumes the rectangles are not too thin.
- Bound # of calls to WSR_Pair(T_1, T_2) for each cell T_1 or T_2.

We define an *outer rectangle* for each bounding rectangle such that these outer rectangles fill space (*i.e.*, the union of all rectangles at a given level of a tree fill the initial space). The outer rectangle for each node in the decomposition tree is defined as follows and illustrated in Figure 107.

$$\hat{R}(P) = \begin{cases} \text{Smallest enclosing cube} & \text{at root} \\ & \\ \text{Use hyperplane that} & \\ \text{divides } R(P) \text{ to} & \text{internally} \\ \text{divide } \hat{R}(P) & \end{cases}$$

Lemma 1 *$l_{\min}(\hat{R}(A)) \geq \frac{l_{\max}(p(A))}{2}$, where $p(A)$ = parent of A in the tree.*

Proof: By induction.
Base: true when $p(A)$ = root, since $l_{\min}(\hat{R}(A)) = \frac{l_{\max}(p(A))}{2}$.
Induction: if true for $p(A)$ then true for A,

Case 1: $l_{\min}(\hat{R}(A)) = l_{\min}(\hat{R}(p(A)))$. True, since $l_{\max}(p(A)) \leq l_{\max}(p(p(A)))$ and by induction $l_{\min}(\hat{R}(p(A))) = \frac{l_{\max}(p(p(A)))}{2}$.

Case 2: $l_{\min}(\hat{R}(A)) < l_{\min}(\hat{R}(p(A)))$. First we note that $l_{\max}(p(A))$ and $l_{\min}(\hat{R}(A))$ must be along the same dimension since Build_Tree cut perpendicular to $l_{\max}(p(A))$ and if $l_{\min}(\hat{R}(A))$ has decreased it must have been cut. Since $\hat{R}(p(A))$ contains $R(p(A))$ cutting $R(p(A))$ in half will mean that $l_{\min}(\hat{R}(A)) \geq \frac{l_{\max}(p(A))}{2}$.

□

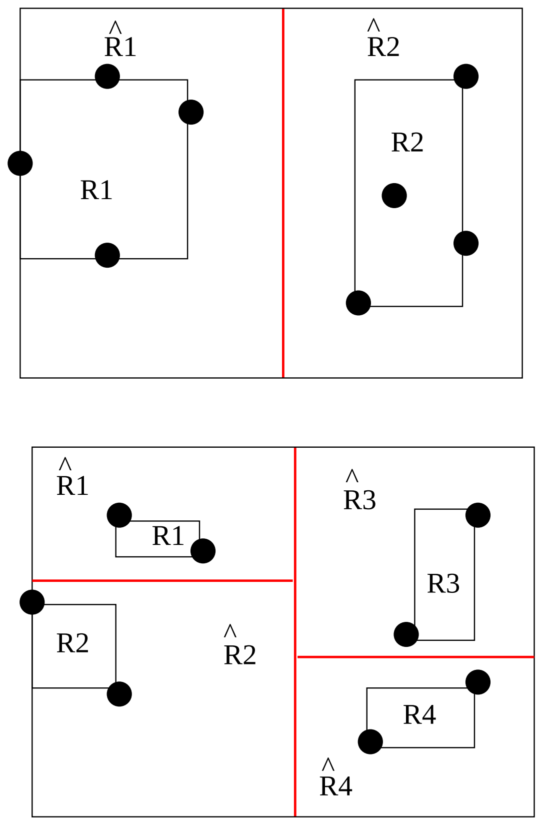


Figure 107: Outer Rectangle

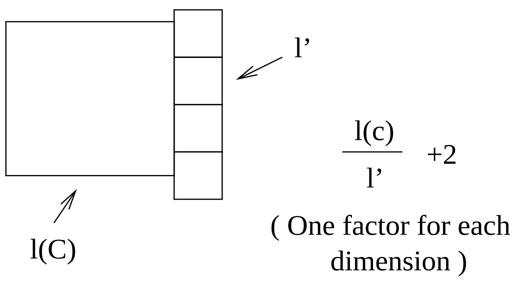


Figure 108: Proof of Lemma 2

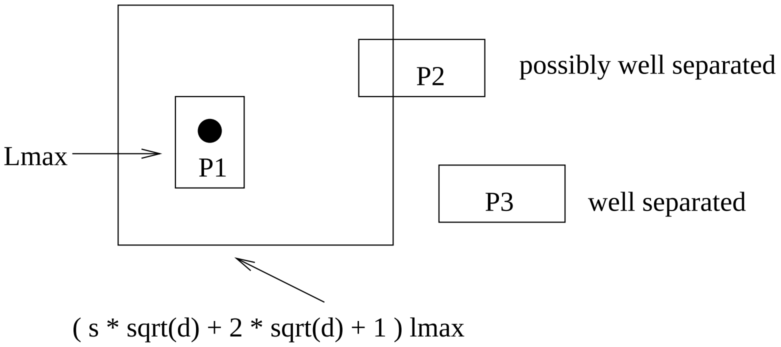


Figure 109: Well separatedness

Lemma 2 *Given a d-cube C and a set S of disjoint d-cubes of length l' that overlap C, then*

$$|S| \leq \left(\frac{l(C)}{l'} + 2\right)^d$$

.

Proof: By picture (Figure 108). □

It's not hard to show that if $l_{\max}(P_1) \geq l_{\max}(P_2)$, then for P_2 not to be well separated from P_1 it must intersect a d-cube of length $(s\sqrt{d} + 2\sqrt{d} + 1)l_{\max}(P_1)$ centered at the center of P_1 (Figure 109).

Note: we will only split P_1 in WSR_Pair(P_1, P_2) if P_1 and P_2 are not well separated.

Putting everything together

Consider for a given node P of the decomposition tree, all invocations WSR_Pair(P,P') such that $l_{\max}(P) \geq l_{\max}(P')$.

1. all P' must be independent (consider recursive structure of the algorithm WSR_Pair)
2. we know $l_{\min}(\hat{R}(P')) \geq \frac{l_{\max}(p(P'))}{2}$ so we can bound the number of P' to

$$|P'| < (2(s\sqrt{d} + 2\sqrt{d} + 1) + 2)^d$$

.

Therefore, the total # of calls to WSR_Pair(P_1,P_2) is bound by

$$2n(2(s\sqrt{d} + 2\sqrt{d} + 1) + 2)^d = O(n)$$

The factor of two at the front comes from the fact that we are counting non-leaf invocations of WSR_Pair(T_1,T_2) while the interactions are created by leaf invocations. However, the total number of leafs is at most twice the number of internal nodes—see Figure 110.

We note that this bound is very conservative and in practice the algorithm actually uses many fewer interactions.

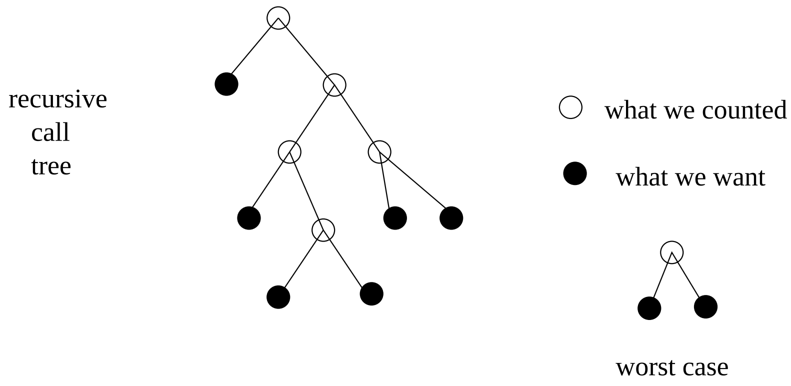


Figure 110: Counting leaves only

3 N-body Summary

- **Algorithms**
  - Particle-Particle
  - Particle-mesh (several variants)
  - Particle-cell (Appel, Barnes-Hut, Press)
  - Cell-cell (FMM, Callahan-Kosaraju)
- **Applications**
  - Astronomy
  - Molecular Dynamics
  - Fluid Dynamics (elementless methods)
  - Plasma Physics

Notes by Rich Vuduc

- Notes regarding the homework assignment
  1. Representing Delaunay triangulations
  2. Incremental Delaunay
  3. Ruppert meshing
  4. Surface approximation
- Intro to VLSI Physical Design
  1. Overview of the VLSI process
  2. The role of algorithms in physical design
- Summary
- References

1 Some notes regarding the homework assignment

Here we discuss a few implementation details that are relevant to a few of the recent algorithms which we have considered.

1.1 Representing a Delaunay triangulation

There are a number of application-dependent approaches to storing the mesh triangulations, such as the one shown in Figure 111. Below, we discuss three possibilities: neighbor graph, quad edge, and triangle representations.

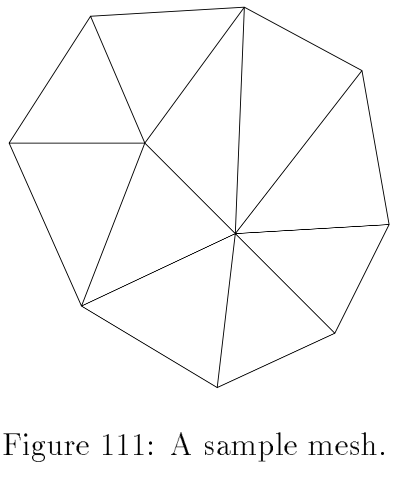


Figure 111: A sample mesh.

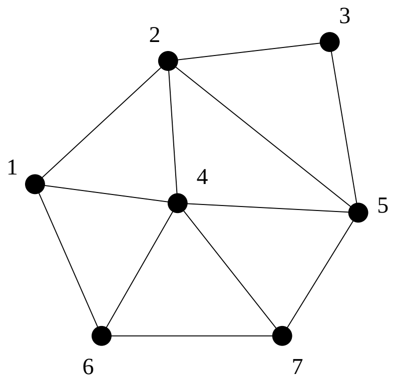


Figure 112: In the *neighbor graph* representation, we interpret the triangulation as a graph and store each vertex with a list of its neighboring vertices.

1.1.1 Neighbor graph

In the *neighbor graph* representation, we treat the nodes of the mesh as vertices in a graph, as shown in Figure 112. There, for each vertex we would store its adjacent vertices:

vertex	neighbors
1	{ 2, 6, 4 }
2	{ 1, 4, 5, 3 }
3	{ 2, 5 }
4	{ 1, 6, 7, 5, 2 }
5	{ 2, 4, 7, 3 }
etc.	...

It is generally helpful to keep each list of adjacent vertices ordered (in the above example, we have stored them in counterclockwise order, but you could use any ordering that is convenient for your application).

1.1.2 Quad edge

Instead of using only vertices for our primary representation, we could store edges instead. One clever way to store edges is called the *quad edge* data structure (Guibas and Stolfi, ACM Trans. Graphics, 4(2), 1985)[2]. This data structure is defined along with an "edge algebra" which allows one to traverse a triangulation easily. In this representation we keep a structure for each edge which stores the 4 adjoining edges as we travel clockwise and counter clockwise from each endpoint of the edge. For example in Figure 113 the four edges adjoining edge 4 are 3, 8, 1 and 5, and the four edges adjoining edge 1 are 3, 4, 5 and 2. In fact the data structure keeps a pointer to one of the 4 fields within each of these adjoining edges. This makes it possible to easily circle around a vertex or an edge. The table below shows an example of our internal representation for the mesh shown in Figure 113. The value after the dot represents which of the 4 entries in the edge is being pointed to.

[2]This was the format used by Garland and Heckbert [1] in their original work. However, in comments to Prof. Blelloch they mentioned that this representation was not particularly well-suited to their application.

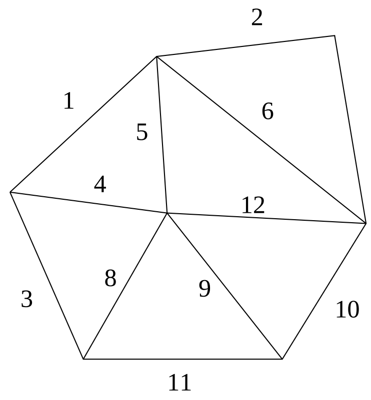


Figure 113: For the quad-edge format, we store each edge along with the 4 adjoining edges (when traveling clockwise and counterclockwise).

edge	neighboring edges
1	{ 2.0, 5.1, 4.2, 3.1 }
	...
4	{ 5.0, 8.3, 3.2, 1.1 }
5	{ 12.2, 4.3, 1.0, 6.1 }
	...
8	{ 4.0, 9.1, 11.2, 3.3 }
9	{ 10.0, 11.3, 8.0, 12.1 }
	...
12	{ 6.0, 10.3, 9.2, 5.1 }
	...

To circle around a vertex we start at an even field of an edge structure (.0 or .2) and follow the pointers, and to circle around a face (triangle) we start at an odd field (.1 or .3). For example, to circle around the vertex in the middle starting at edge 4 we simply start by following 4.0 and continue around touching edges 5, 12, 9, 8 and back to 4 (more specifically 5.0, 12.2, 9.2, 8.0, 4.0). Similarly to circle around triangle (1,4,5) we can start from 4.3 and visit 1.1, 5.1 and back to 4.3.

The algebra on the quad-edge data structure suggested by Guibas and Stolfi uses the three operations `next`, `rotate`, and `symmetric` which take a pointer and return a new pointer. The `next` operation circles around a vertex or triangle, the `rotate` operation switches from a vertex to a triangle (the one that shares the edge, next edge, and vertex) or vise versa, and the `symmetric` operation switches to the vertex at the other end of the edge. Assuming each quad-edge is an array of 4 "pointers" each consisting of a pointer to another quad-edge and an index within the quad-edge, then for a pointer `(e,i)` the operations are defined as:

```
next(e,i) = e[i]
rotate(e,i) = (e, i+1)
symmetric(e,i) = (e, i+2)
```

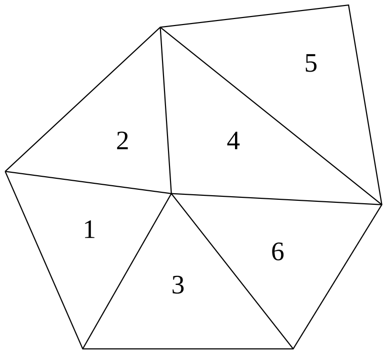


Figure 114: Mesh representation using triangles. Triangle 1 shares sides with triangles 2 and 3.

where `+` is defined modulo 4.

1.1.3 Triangles

Lastly, we turn to what is perhaps the most natural representation: *triangles*. With each triangle, it may be useful to store the 3 other with which it shares a side. So, for the example shown in Figure 114, our table of data might look something like:

triangle	side-adjacent triangles
1	{ 3, 2, □ }
2	{ 1, 4, □ }
3	{ 6, 1, □ }
4	{ 6, 5, 2 }
5	{ 4, □, □ }
6	{ 4, 3, □ }

In addition to the neighboring triangles it is typically useful to store the vertices of each triangle. In this representation the function `opposite(p,T)` which given a triangle `T` and a vertex (point) `p` on that triangle returns the triangle opposite that point is often useful. For example, in Figure 114 given the leftmost point and triangle 2, `opposite` would return triangle 4.

1.2 Incremental Delaunay

Recall that in the incremental Delaunay algorithm of previous lectures, we are given a triangulation to which we wish to add an additional point. The suggested approach is based on a depth-first search style algorithm. Figure 115 illustrates the action of the following pseudocode to insert a point p into an existing triangulation.

```
function Insert(p, R)
    // p a point
    // R an existing triangulation

    T = the triangle containing p
    (T1, T2, T3) = the triangles created by splitting T by p
    Patch(p, T1)
    Patch(p, T2)
    Patch(p, T3)
end

function Patch(p, T)
    T' = Opposite(p, T)
    if InCircle(p, T')
        (T1, T2) = Flip(T, T')
        Patch(p, T1)
        Patch(p, T2)
end
```

The function Flip flips two adjacent triangles so the crossedge changes.

1.3 Ruppert Meshing

In this section we discuss two issues to consider when implementing Ruppert's meshing algorithm [2].

The first relates to how one finds encroaching points. First, recall that a point p *encroaches* upon a segment s if p lies within the diametrical circle defined by s (see Figure 116). To determine which points encroach upon s, it is sufficient to check endpoints of all Delaunay edges that come out of an endpoint of the segment. In a practical implementation, one would probably need a data structure that facilitates moving from point to point within the triangulation; one possibility would be a data structure that keeps neighboring triangles with each point.

The second issue relates to the creation of *phantom small angles*, which is best explained by an example. In particular, suppose that our input polygon is the one shown in Figure 117 (*top*). Recall that an initial step in computing the triangulation requires that we compute the convex hull of the input polygon; this could result in the creation of *phantom small angles* such as the one shown in Figure 117 (*bottom-left*). An easy solution is to first create a bounding box around the input polygon, as shown in Figure 117 (*bottom-right*).

1.4 Surface Approximation

One important step in the surface approximation algorithm discussed in a previous lecture is to find the largest error in a newly added triangle. This requires examining every point within the triangle. One technique for doing so might be to bound the triangle under consideration by a rectangle first, then scan the pixels within the triangle row- or column-wise. For each

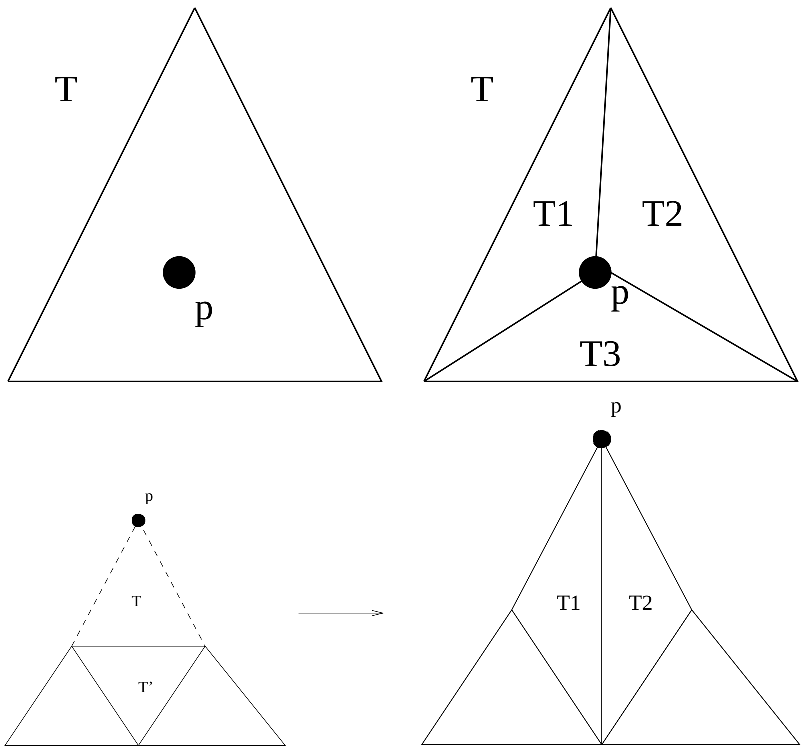


Figure 115: (*Top-left*) We wish to add the point p to an existing triangulation, and p happens to fall within the triangle T. (*Top-right*) Adding p creates the smaller triangles T_1, T_2, T_3. (*Bottom-left*) We consider the action of the `Patch` subroutine at some point in the recursion for the point p and some other triangle T (i.e., not necessarily the same as above). (*Bottom-right*) Result after executing `flip`(p, T').

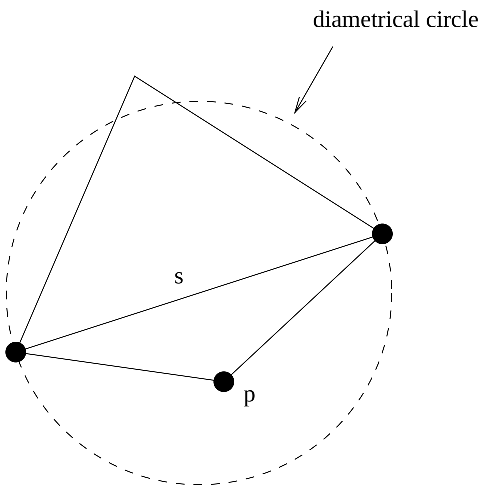


Figure 116: In the situation shown, we wish to see if there are any points in the triangulation that encroach on the segment s (e.g., p).

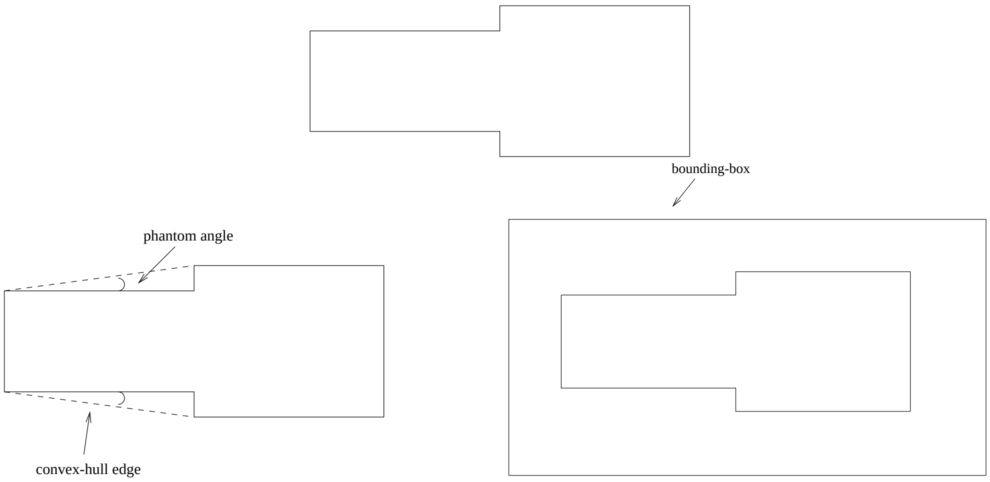


Figure 117: (*Top*) Input polygon. (*Bottom-left*) Initial convex-hull computation results in creation of tiny "phantom small angles." (*Bottom-right*) One easy solution is to placing a bounding box around the figure, and then triangulate this new figure.

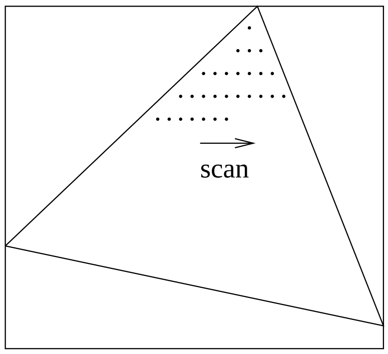


Figure 118: In computing the error of a newly added triangle in the surface approximation algorithm, we can bound the triangle by a box and scan, testing whether or not the pixel lies within the triangle before using it to compute the error.

pixel that we scan, we can check whether or not it falls within the triangle before computing its error. (See Figure 118.) A fancier method would find the edges that border on the left and right and only scan the horizontal lines between them.

2 Intro to VLSI Physical Design Automation

Since the dark ages (i.e., the 1960's), a typical integrated circuit design has grown from a hundred transistors to the millions of transistors required by modern microprocessors. *Very large scale integration* (VLSI) refers to the process of integrating such a large number of transistors for fabrication; *VLSI physical design* is the process of converting a circuit description into a layout. The ability to lay out such enormous circuits has been made possible by the intelligent application of computer algorithms.

Here, we give a brief overview of the overall VLSI physical design process, and show where algorithms play a key role. We will be making a number of references to diagrams shown in [3].[3]

2.1 Overview

The following is a high-level description of the VLSI design process. The focus of our discussions will be on step 5 below.

1. **System specification**: The first step involves the creation of a formal, high-level representation of the system. In this stage, we consider the desired functionality, performance, and physical requirements of the final product.

2. **Functional design**: In this step, we break down the system into functional sub-units, and specify the relationships between them (for example, by generating a timing diagram or showing the desired flow between sub-units).

3. **Logic design**: In this process, we formalize the logical structure of our functional design, typically by generating boolean formulas. An interesting problem is how to minimize such expressions, and neat algorithms like binary decision diagrams have found applications here. However, we will not be discussing these.

4. **Circuit design**: From the logic design, we next need to generate a circuit representation. We have to consider power and speed requirements of our original design, and often perform simulations of electrical behavior in this stage (e.g., through the use of PSpice software).

5. **Physical design**: In this stage (the focus of our discussions), we must actually lay out the circuit designs. We must ensure that our resulting layout conforms to the physical and geometric constraints of our system, which we formulate as *design rules*. These design rules are essentially guidelines based on physical constraints due to limitations in the fabrication process.

6. **Fabrication**: This stage is the actual physical manufacturing step, in which we prepare the wafer and deposit/diffuse various materials onto the wafer.

7. **Packaging, testing, and debugging**: In this stage, we test the actual manufactured and packaged chip.

It is important to note that this entire design process is iterative, i.e., there is a kind of "feedback" effect in which we examine the results, perform any necessary hand-tuning, and re-run the process, continually refining our previous results.

[3] In fact, most of the material here is simply a paraphrase of material that appears in [3].

2.1.1 Physical design

We now turn our attention to the physical design cycle which will be our focus. The following is a brief summary of the high-level process. Note that the input to this process is the circuit design, and the output will be a layout ready for the fabrication stage.[4]

1. **Partitioning**: We must first break down, or *partition*, the input circuit into more manageable blocks (or subcircuits, or modules). In the partitioning process, we must consider the size and number of blocks, as well as the number of interconnections which will obviously affect the routing. The set of interconnections is called a *netlist*. For large designs, the blocks may be hierarchical.

2. **Floorplanning and placement**: In *floorplanning*, we consider such factors as the exact size of the blocks generated in the previous step. This is a computationally difficult step, and often requires a design engineer to do this by hand. During *placement*, we position the blocks exactly as they will appear on the chip. We want to lay out the blocks in such a way so as to minimize the area required to complete all the interconnections. This is generally an iterative process in which we lay out the blocks, and continually refine the placements.

   Furthermore, it is crucial to use a good placement strategy because the placement essentially determines the routing (see below).

3. **Routing**: In the *routing* phase, we lay out the interconnections (i.e., put down wires) specified by the netlist that was generated in the partitioning phase (above). We refer to the space not occupied by blocks as the *routing space*. We partition this space into rectangular regions called *channels* and *switchboxes*. Our router must make the connections using only these two types of regions, and it must do so in such a way so as to minimize wire length.

   The routing process can be divided into two phases. In the first phase, called *global routing*, in which the global router must make a list of channels (i.e., passageways) through which to route the wire. The next phase, called *detailed routing*, completes the actual point-to-point wiring.

   Although the routing problem is computationally hard, many heuristic algorithms have been developed.

4. **Compaction**: In this final phase, we compress the generated layout to reduce the total area. A smaller area would enable us to increase the yield.

2.1.2 Design styles

Although we have taken the entire physical design process and broken it up into smaller subproblems, we have noted that each subproblem is still computationally hard. To make these problems more tractable (and thus reduce the time-to-market and increase the yield), we can further restrict our design methodology into one of the following *design styles*:

[4] Automation of the entire physical design process is akin to the idea of building a "silicon compiler."

1. **Full custom**: We use this term to refer to a design style which places few (if any) constraints on the physical design process. For example, we would place no restrictions on where a block could be placed (e.g., on a grid). Clearly, this is the most difficult design style to automate because of the high degree of freedom. A *full custom* approach is really only appropriate when performance must be high. For instance, this is the method of choice for the layout of modern microprocessors. An example is shown in [3] (Figure 1.3).

2. **Standard cell**: In the *standard cell* design style, we predesign certain functional units as standard cells, or standard blocks, and create a *cell library* of "pre-packaged" blocks that we can then use for placement. Each cell is preanalyzed and pretested. Given that the cells are predesigned, the main challenge is the layout and routing. An example of a standard cell layout is shown in [3] (Figure 1.4). Notice that the blocks are all of a similar shape and size, and that their layout is much more regular than the layout of the full-custom design shown in [3] (Figure 1.3). Observe that in the standard cell design, our only (roughly speaking) degree of freedom is the ability to shift blocks to open up routing channels between placed cells.

3. **Gate arrays**: A *gate array* is like a standard cell except that all the "cells" are the same, and laid out in an array form as shown in [3] (Figure 1.5). The name is indicative of the idea that the cells could be as simple as a single gate (a 3-input NAND gate is shown). Furthermore, the routing channels are basically predetermined. The advantage of the gate array style over the previous two is its simplicity. In particular, it is cheaper and easier to produce a wafer based on this style. Routing is also simplified, since we only have to make connections and not worry about minimizing the area used. An example of a routed gate array for a simple logic circuit is shown in [3] (Figure 1.6).

   However, the cost is the rigidity imposed—we would certainly be forced to produce larger chips and incur a performance penalty.

4. **Field programmable gate arrays**: A *field programmable gate array* (FPGA) is something in-between the gate array and standard cell. An example is shown in [3] (Figure 1.7). Each "cell" of a FPGA (labeled by B_i in the figure) can be viewed as a lookup table that can be programmed. Furthermore, there is no free region over which we do our routing; instead, the wires are fixed, and we connect the cells by burning in *fuses* (shown as squares and circles). The figure shows a sample logic circuit which has been partitioned into four regions. We construct the lookup table for each region, and this is what is programmed into the cells of the FPGA (in this example, the four left-most cells labeled B_i). The connections appropriate fuses are burned in (shown as darkened circles and squares). The partitioning problem for FPGAs is different from that of the other design styles, but the placement and routing problems are similar to those of the gate array.

We summarize the various tradeoffs alluded to above as follows [3] (table 1.2):

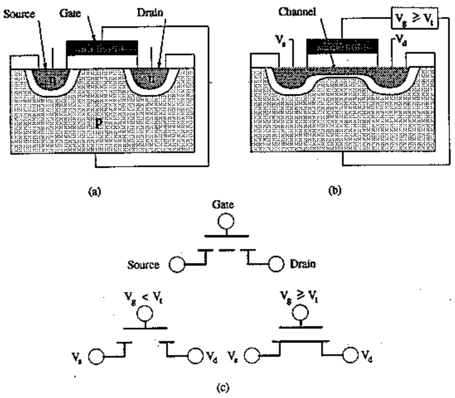


Figure 119: The operation of an nMOS transistor. (a) A vertical cross-section of a transistor based on the $n-p-n$ junction. (b) When a voltage has been applied to the gate, the induced electric field alters the conducting properties of the channel, thereby allowing current to flow. (c) Schematic representations of the process. [Taken from [3] (Figure 2.5).]

	style			
Area	compact	compact to moderate	moderate	large
Performance	high	high to moderate	moderate	low
Fabrication layers	all	all	routing layer	none

In general, within a given column of the table there is a correlation between the rows. Moreover, the columns are roughly ordered in decreasing cost.

2.1.3 Transistor level

When we discuss algorithms in future lectures, it will be sufficient to formulate the routing and placement problem at an abstract level. However, in this section, we will very briefly look at the "small picture" for a little perspective.

Figure 119 shows a diagram of the vertical cross-section of a nMOS transistor, as well as its schematic representations. An nMOS transistor is an example of a more general kind of transistor called a *field-effect transistor*. In such a device, we can control the flow of current between the two terminals (labeled *source* and *drain*) by controlling the voltage applied to the non-conducting terminal called the *gate*. In particular, applying a voltage to the gate sets up an electric field that changes the conducting properties of the region labeled *channel* thereby allowing current to flow from the source to the drain. For a more general introduction to the basic solid-state physics behind the operation of these devices, see [4] and [5].

By combining several transistors, we can construct basic logic gates such as the NAND gate shown in Figure 120. See the figure caption for a rough description of its operation.

Finally, note that in the figure of the NAND gate, there are various rectangular shaded regions which represent different materials. These materials must obey certain rules, namely, they must be of certain dimensions and must be separated by different amounts, as shown in Figure 121. These are the kinds of rules and constraints we will be subject to in our automation algorithms.

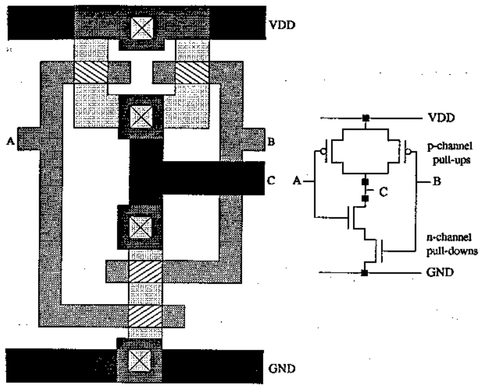


Figure 120: The layout of CMOS NAND gate (*left*) using four transistors, and its schematic representation (*right*). The input to the NAND are labeled A and B, and the output is C. The bottom two transistors are n-channel and open when their gates are high while the top two are p-channel and open when their gates are low. (*Basic operation*) When A and B are both high, the channels of the upper-two transistors are closed and the bottom two are opened, thus enabling a connection from GND to C. In all other cases, the channels of at least one of the lower transistors will be closed which will break the connection between GND and C, whereas when either channel of the upper two transistors are open, C will be pulled up to VDD. [Taken from [3] (Figure 2.7).]

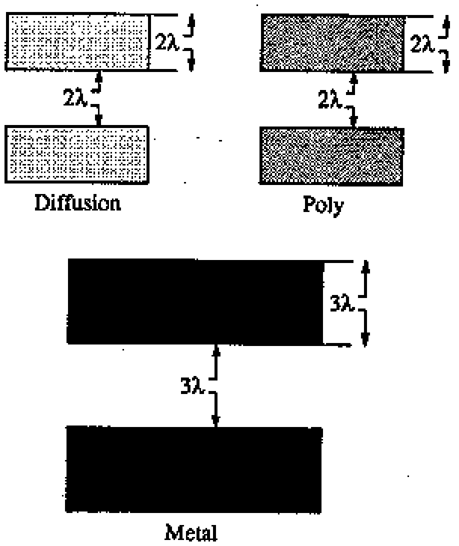


Figure 121: Size and separation rules. The dimensions have been characterized in terms of the design parameter λ. The ".35" of the terms ".35 micron technology," et al., refer to this parameter. [Taken from [3] (Figure 2.10).]

2.2 The role of algorithms

Algorithms have played a crucial role in the explosive growth of VLSI technology. Graph theory has played a particularly important part. Here is a list of some of the many algorithms that have been applied to VLSI physical design automation:

1. Graphs
    - spanning trees
    - shortest path
    - matching
    - Steiner trees
    - maximum independent set
    - maximum clique
2. Computational geometry
    - intersection of lines or rectangles
3. Optimization
    - integer programming

These algorithms have found applications in many of the design areas discussed in the physical design overview, namely

- layout
- partitioning
- placement
- routing
- compaction

Furthermore, these algorithms are applicable in all the different design styles (full-custom, standard cell, etc.). We will be paying particular attention to the routing problem.

Our overall goal will be either to minimize area given physical constraints on our design, or to maximize functionality in a given fixed area. We will have to be concerned with how to:

minimize delays : In particular, we can accomplish by minimizing wirelength and reducing capacitance (by keeping the overall layouts small).

reducing the number of layers Increasing the number of layers required in a design greatly increases fabrication costs.

reducing the number of pins : Of particular concern in multi-chip modules.

All of the major designers and manufacturers of VLSI chips are investing heavily in automation research and techniques. However, hand-tuning still has an important place in the design phases.

2.3 Evolution

We will close by summarizing the history of development tools in physical design automation.

Year	Design Tools
1950-1965	Manual design
1965-1975	Layout editors Automatic routers Efficient partitioning algorithms
1975-1985	Automatic placement tools Well defined phases of design of circuits Significant theoretical development in all phases
1985-present	Performance driven placement and routing tools (in particular, an emphasis on space and speed)
	Parallel algorithms for physical design Significant development in underlying graph theory (i.e., the use of theoretical results) Combinatorial optimization problems for layout

3 Summary

In the first section, we gave some notes about implementation methods for several triangulation algorithms. In particular, we discussed some ideas for data structure representation, with the triangle representation seeming the most useful.

We then gave a brief overview of the entire VLSI design process, and introduced the physical design stage in more detail. We showed that algorithms play an important role, our overall goal being to minimize the layout area (to increase our chip's performance) while subject to various constraints of our physical design (e.g., separation).

References

[1] M. Garland and P. Heckbert. "Fast polygonal approximation of terrains and height fields." CMU Technical Report, CMU-CS-95-181, Carnegie Mellon University, Pittsburgh, PA, September 1995.

[2] J. Ruppert. "A Delaunay refinement algorithm for quality 2-dimensional mesh generation." NASA Technical Report, RNR-94-002, January 1994.

[3] N. Sherwani. *Algorithms for VLSI Physical Design Automation.* Kluwer, 1995.

[4] C. Kittel. *Introduction to Solid-State Physics*. Merriam, (fill in details), 1995.

[5] On-line course notes on "Heterojunction Devices" from the University of Colorado. [http://ece-www.colorado.edu/~bart/6355.htm]

Notes by Mehul Shah

- Introduction to VLSI Routing
- Global Routing
- Two terminal routing (shortest paths)
  1. Dijkstra's algorithm
  2. Maze routing
- Multi terminal routing (Steiner trees)
  1. Steiner tree definitions
  2. Optimal rectilinear steiner trees

1 Introduction to VLSI Routing

In the physical design of VLSI circuits, the *routing* phase consists of finding the layouts for the wires connecting the terminals on circuit blocks or gates. It assumes the locations of the blocks have already been determined, although some routing schemes allow a certain amount of movement of the blocks to make space for wires. The basic problem is illustrated in Figure 122. A *terminal* is a location on a block, typically along one of the edges, which needs to be connected to other blocks. A *net* is a set of terminals that all need to be connected to each other. A *netlist* is a collection of nets that need to be routed. The *routing space* is the space in which we are allowed to place wires, typically any space not taken by blocks. The *routing space* between two blocks is called a *channel*. The routing problem is the problem of locating a set of wires in the routing space that connect all the nets in the netlist. The capacities of channels, width of wires, and wire crossings often need to be taken into consideration in the problem.

Given that there can be millions of transistors on a chip and tens of thousands of nets to route, each with a large number of possible routes, the routing problem is computationally very difficult. In fact, as we will see, even finding an optimal layout (minimum wire-length) for a single net is NP-hard. Because of the hardness of the problem, most routing algorithms are approximate and will not necessarily find the best solution. The first simplification that is typically made is to divide the problem into two phases called *global routing* and *detailed routing*. In global routing the exact geometric details are ignored and only the "loose route" for the wires are determined. More specifically, the open regions through which each wire flows is computed. This loose route is converted to an exact layout in the latter phase. Detailed routing completes the point-to-point wiring by specifying geometric information such as the location and width of wires and their layer assignments. Even with this breakdown into global and detailed routing each phase remains NP-hard and further

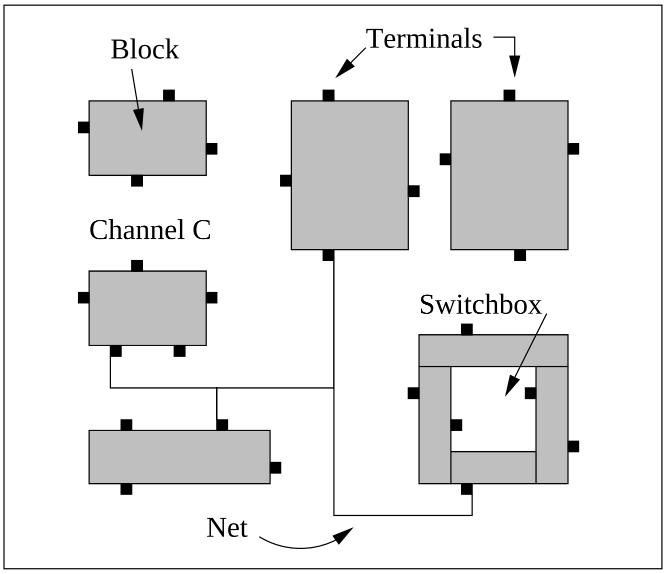


Figure 122: Some definitions.

approximations are used to solve them. In this lecture, we concentrate on the global routing phase.

The optimality goal of global routing is dependent upon the nature of the chip design. For full custom and standard cell designs, the objective is to minimize the total area of the final chip. Thus, in the full custom designs the total wire length is minimized, or the maximum wire length is minimized, or changes to placement are minimized. In the standard cell design, the maximum wire length is minimized or the total channel heights are minimized. In gate arrays, the previous phases assign gates to blocks, and the primary goal is to determine the feasibility of routing. If feasible, the maximum wire lengths are minimized; otherwise, different assignments are sought. Variants of shortest paths algorithms are employed for many of these computations.

2 Global Routing Algorithms

In the global routing phase, the input is a netlist consisting of a set of nets each of which is a set of terminals. We will distinguish between two-terminal nets and multi-terminal nets. Most chips will have both types and different algorithms are often used for each.

In addition to the netlist we need some way to specify the routing space. This is typically done as a graph. There are two graph models which are often used: the grid graph model and the channel-intersection graph model. In the *grid graph* model the chip is modeled as a regular $n \times m$ grid graph with vertices that do not belong to the routing space removed (*e.g.*, vertices occupied by a block, or possibly already routed to capacity). Wires can be routed along any edges of this graph and one typically assumes each edge has unit capacity (*i.e*, only one wire can be routed on it). All edges in this model have the same length. Figure 126 shows an example—the shaded vertices belong to blocks and therefore do not belong to the

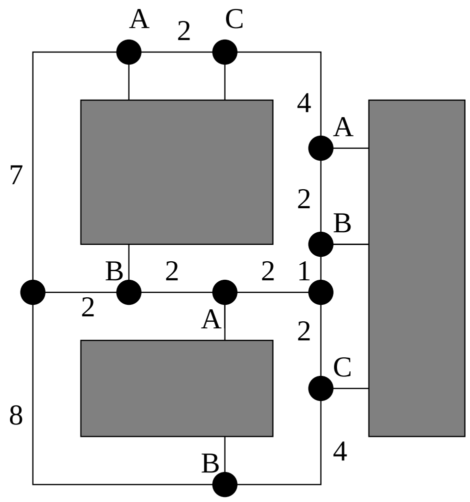


Figure 123: An example of a channel-intersection graph model. The circles are the vertices, the letters are terminals, and the numbers specify the length of each edge.

graph. In the *channel-intersection graph* model vertices are placed at all intersections of the channels, and at the terminals. The lengths of the edges are no longer all the same and the capacity of each edge represents the number of wires that can be routed along it. Figure 123 shows an example. We will consider algorithms for both types of graphs.

Generally, two approaches are taken to route the nets. The first approach is to route sequentially by laying down one net at a time based on constraints from previous nets, and the second is to route all the nets simultaneously. The first approach will typically not give as good routes, but is computationally much more feasible. In this lecture we will just consider the first approach and in the next lecture we will consider an algorithm based on integer-programming for the second approach.

When laying down the nets sequentially, depending on the order the nets are laid out, some nets already routed may block others yet to be routed. One approach to mollify this problem is to sequence the net according to various factors: its criticality, perimeter of the bounding rectangle, and number of terminals. However, sequencing techniques are not always sufficient, in which case, "rip-up and reroute" and "shove-aside" techniques are employed. The "rip-up and reroute" technique involves removing the offending wires which block the affected nets and rerouting them. The "shove-aside" technique moves wires to allow for completion of failed connections. Another approach is to route nets with two or three terminals first since there are few choices for routing them and they comprise a large portion of the netlist.

We now consider the following algorithms for laying down each net.

- Two terminal algorithms (shortest paths)
  1. Dijkstra's algorithm (intersection graphs)
  2. Maze routing and Hadlock's algorithm (grid graphs)
- Multi-terminal algorithms (Steiner trees)

1. Optimal rectilinear steiner trees

3 Two-Terminal Nets: Shortest Paths

The shortest path problem we consider involves an undirected graph $G = (V, E)$ with positive edge weights (representing the length of each edge). Our goal is to find a path from a given source to a destination that minimizes the sum of edges weights along the path.

3.1 Dijkstra's Algorithm

Dijkstra's algorithm is one such single source shortest path algorithm, and in fact in its general form finds the shortest path from the source to all vertices. The algorithm starts from the source vertex and proceeds outward maintaining a set S of vertices that have been searched. Any vertex that is not in S but is a neighbor of a vertex in S is said to be in the *fringe*. It proceeds until all vertices are labeled with their shortest distance from the source (or until the destination is encountered). The algorithm and an example of one of the steps is shown in Figure 124. The last three lines maintain the invariant between iterations since $d[v]$ is correct when v is added to S and because for all new fringe nodes their shortest path through $S \cup \{v\}$ is either through S, which was previously correct, or through v which has length $d[v] + w(u, v)$.

If the fringe vertices are maintained in a priority queue with their distance to S as the priority, then each `findmin` and modification of the $d[u]$ takes $O(\log n)$ time, where $n = |V|$. Since there are at most n `findmin` operations and at most $m = |E|$ modifications to the $d[u]$ (one for each edge), the time is bound by $O(m \log n)$. A Fibonacci heap implementation of the queue, however, can reduce the amortized running time to $O(n \log n + m)$ (see CLR). Note, if all edge weights are constant and equal then this method reduces to breadth-first search.

If all we want to find is the shortest path to a single destination d, then Dijkstra's algorithm can terminate as soon as $d \in S$ (as soon as d is visited). The algorithm, however, moves uniformly in all directions from the source in a best-first manner, so in general a large part of the graph might be visited before d. For graphs which can be embedded in Euclidean space (i.e. the triangle inequality holds for edges within the graph), the efficiency of this search can be improved by biasing the search toward the destination. This optimization can be very beneficial when making two-terminal connections in VLSI layout.

To bias the search toward the destination the modified algorithm uses a different priority for selecting the next vertex to visit. Instead of $d[v]$, the length of the path from the source, it uses $d[v] + L(v, d)$, where $L(v, d)$ is the Euclidean distance from v to the destination d. By including $L(v, d)$, the search will be biased toward vertices with shorter distance to the destination. However, we have to make sure that the algorithm still works correctly. This is guaranteed by the following theorem, which is illustrated Figure 125.

Theorem 2 *For a step of the modified Dijkstra's algorithm, in Figure 125 if* $P_1 + L_1 \leq P_2 + L_2$ *then* $P_1 \leq P_2 + P_3$.

function Dijkstra(G, s)
  // $G = (V, E)$ is a input graph with nonnegative edge weights $w(u, v), (u, v) \in E$
  // s is the source vertex

  // Result: for each $v \in V$, $d[v]$ = shortest distance from s to v

  forall $(v \in V)$, $d[v] = \infty$
  $d[s] = 0$
  $S = \emptyset$
  while $S \neq V$ // not all vertices have been searched
    // **Loop Invariants:**
    //   for all $v \in S$, $d[v]$ is shortest path from s to v
    //   for all $v \in Fringe$, $d[v]$ is shortest path exclusively through S from s to v

    v = node $v \in Fringe$ with minimum $d[v]$

    // Note that $d[v]$ is the shortest path to v from s because
    //   1: shortest path through S to v is $d[v]$ (by the loop invariant)
    //   2: any path through a vertex $u \in (V - S)$ must be at least as long as $d[v]$
    //   since $d[u] \geq d[v]$ and all edge weights are nonnegative.

    $S = S \cup \{v\}$ // "visit" v
    for $u \in$ neighbors(v)
      $d[u] = \min(d[u], d[v] + w(v, u))$
end

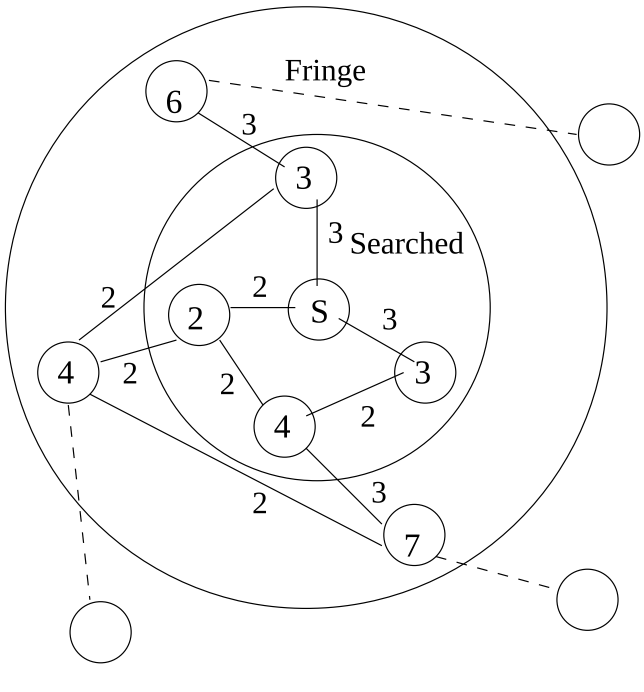


Figure 124: The code for Dijkstra's algorithm and a snapshot of one iteration.

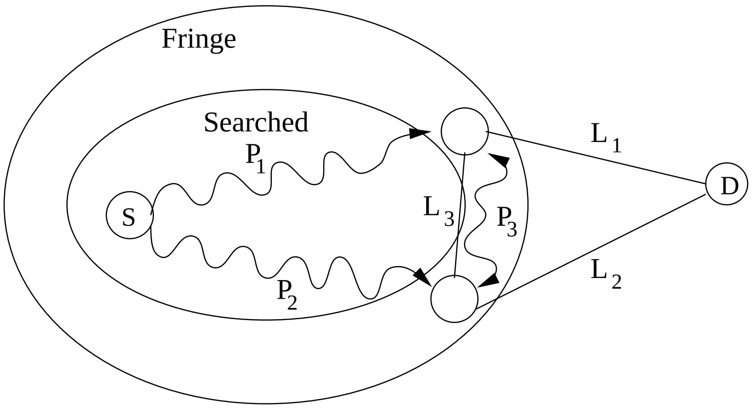


Figure 125: Incorporating the Euclidean distance into the priority maintains the invariant while forcing Dijkstra's algorithm to move in the direction of the destination.

Proof: $P_3 \geq L_3$, a property of Euclidean graphs. Also, $L_3 \geq L_2 - L_1$, which follows from the triangle inequality. Our assumption states:

$$P_1 \leq P_2 + L_2 - L_1$$

Thus,

$$P_1 \leq P_2 + L_3$$

$$P_1 \leq P_2 + P_3$$

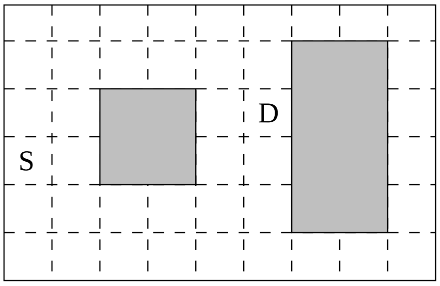
□

This theorem implies that if we use $P + L$ as the priority then when $P_1 + L_1$ is the fringe vertex with the minimum priority we are sure there is no shorter path to it through some u not in the searched vertices S and our invariant is maintained.

Exercise: *Consider an $n \times n$ grid graph in which all edges have unit weight. If the source is in the middle of the left boundary and the destination is in the middle of the right boundary, how many vertices will the modified Dijkstra's algorithm take compared to the original? How about if the source is on the lower left and the destination on the upper right?*

3.2 Maze Routing

Finding shortest paths on a grid-graph is a special case of the general shortest path problem where all weights are the same (unit). In the context of VLSI routing, finding such shortest paths for two-terminal nets is called maze routing. A simple approach for maze routing is to perform a breadth-first search from source to destination (this is referred to as "Lee's



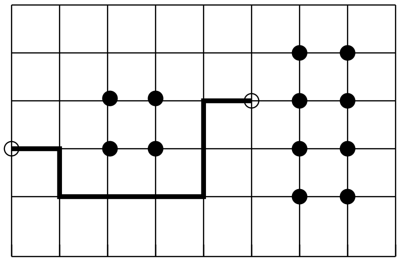


Figure 126: The right figure shows the path determined by a maze route between S and D on a grid graph model of the circuit area in the left figure. For Hadlock's algorithm, $M(S,D) = 6$, and $d(P) = 1$ because the path turns away from D once.

Algorithm" in the VLSI community). Such a search is faster than Dijkstra's algorithm since it only takes constant time to visit each vertex. However, like Dijkstra's unmodified algorithm it makes no attempt at heading toward the destination.

In a similar spirit to the modified version of Dijkstra's algorithm Hadlock observed that the length of a path on a grid from s to t is $M(s,t) + 2d(P)$, where $M(s,t)$ is the Manhattan distance and $d(P)$ is the number of edges on path P directed away from the target (see Figure 126). Hence, the path can be minimized if and only if $d(P)$ is minimized. Thus, instead of labeling each vertex on the fringe with the distance, it is labeled with the number of times the best path to it has turned away from the target. Hadlock's algorithm, therefore uses the detour number $d(P)$ for the priority in the shortest-path search. As with the modified version of Dijkstra's algorithm, this has the effect of moving the search toward the destination.

4 Multi-Terminal Nets: Steiner Trees

4.1 Steiner Trees

Given a graph with a subset of vertices labeled as demand points, a minimum Steiner tree is the minimum set of edges which connect the demand points. Figure 127 gives an example of a minimum Steiner tree. A minimum spanning tree (MST) is a tree which connects all vertices with edges with minimum total weight. Finding the minimum Steiner tree is an NP-hard problem; however, finding the minimum spanning tree is tractable and, in fact, quite efficient. Thus, finding a minimum spanning tree is the initial step in many of the algorithms for approximating minimum Steiner trees.

Usually in VLSI circuits, wires are laid down only in horizontal or vertical directions. Thus, for multi-terminal nets, the demand points are a set of vertices on a grid, and the routing problem is to find a minimum rectilinear Steiner tree (RST), one with horizontal or vertical edges. Finding the minimum RST remains NP-hard so heuristics are employed. A useful fact to note, however, is that the cost of a minimum RST is at most 3/2 the cost of a

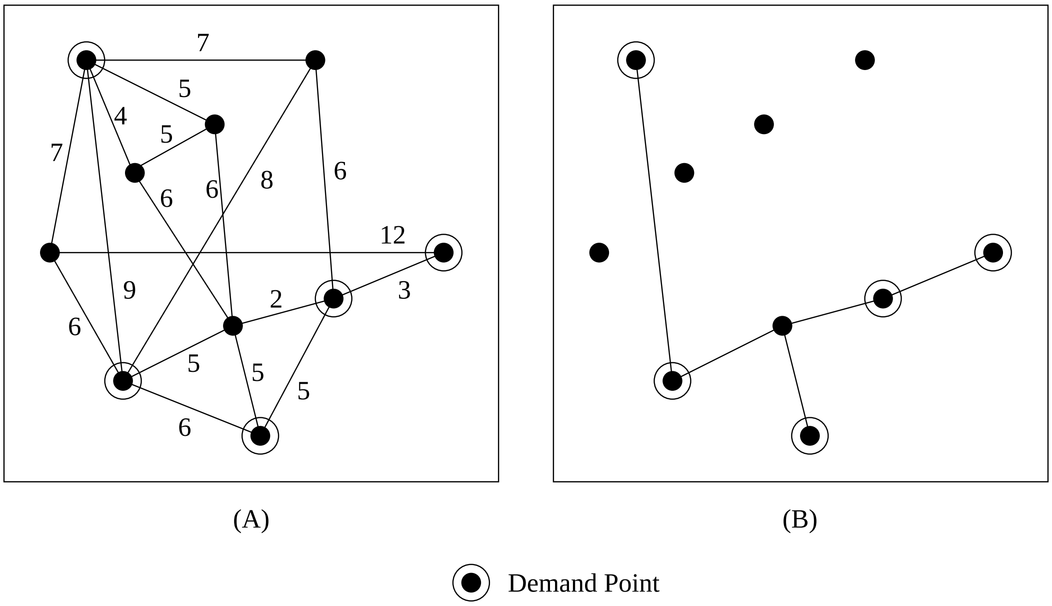


Figure 127: Given the demand points, (B) shows the minimum cost Steiner tree of (A).

minimum spanning tree. Thus, in practice, an approximate minimum RST is computed by rectilinearizing each edge of a minimum spanning tree (see Figure 128).

Various methods for rectilinearizing edges in a MST exist. A layout which transforms each diagonal edge using only one turn on the grid is called an L-RST. A layout which uses two turns for each diagonal edge is called a Z-RST. A layout which uses an arbitrary number of turns but never goes "away" from the destination is called an S-RST. These variants are illustrated in Figure 129. Notice that an optimal RST may have costs smaller than the optimal S-RST, which may have cost smaller than the optimal Z-RST, which likewise may have a cost smaller than the optimal L-RST. The fewer the restrictions, the better the approximation to a minimum RST.

4.2 Optimal S-RST for separable MSTs

If a MST of a given graph obeys the separability property, then there exists a method to find an optimal S-RST from the MST. The separability property states that any pair of non-adjacent edges must have non-overlapping bounding rectangles (see Figure 130). Given such a graph the approximate RST algorithm works as follows. First, it calculates the minimum spanning tree of the given points. Then, the MST is hung by one of the leaves of the tree. Then starting at the root edge it recursively computes the optimal Z-layouts of the subtrees rooted at the children. Each combination of optimal Z-RSTs of the children is merged in with the Z-layout of the root node. The least cost merged layout is the optimal Z-layout of the entire tree. Pseudocode for this algorithm is given in Figure 130 (D).

Note, the separability of the MST insures that the optimal S-RST has the same cost as the optimal Z-RST. This algorithm runs in polynomial time and provides an optimal S-RST, but not necessarily an optimal RST!

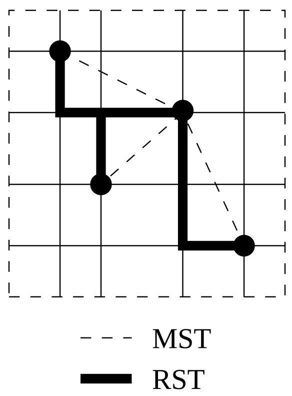


Figure 128: A comparison of a rectilinear Steiner tree (RST) and minimum spanning tree (MST) on a grid graph.

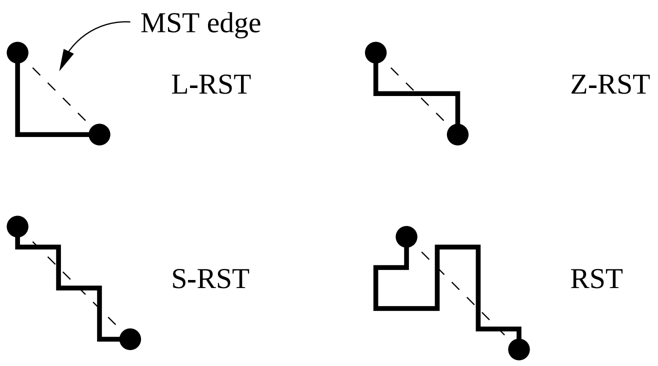
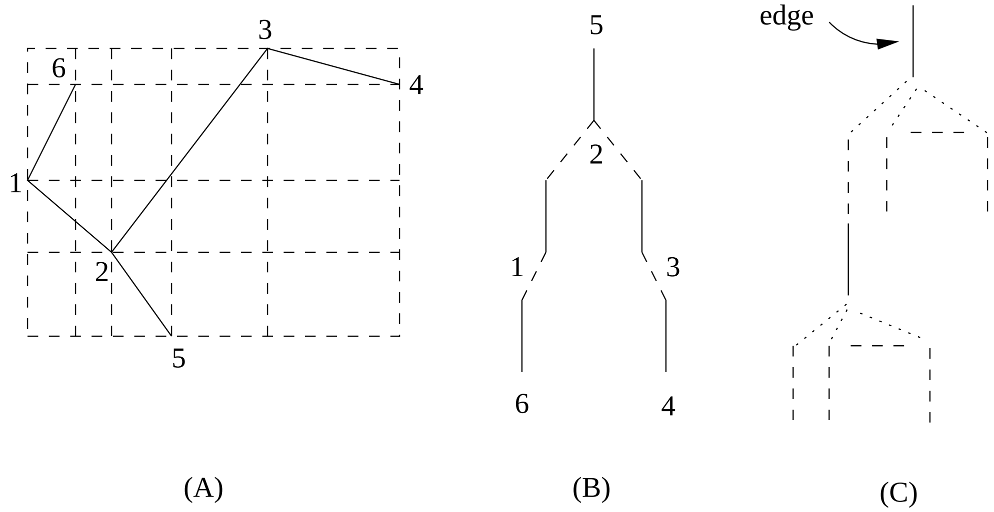


Figure 129: This illustrates the various transformations of an MST edge into rectilinear edges.



function LeastCost(z_e,T_e)

// z_e a Z-layout for edge e

// T_e the subtree of edges hanging from e (including e)

// Returns the minimum cost Z-layout of edges in T_e with the constraint that

// edge e is laid out using z_e (all other edges are unconstrained).

begin

  if no children in T_e **then return** z_e

  else

    for each child edge e_i of T_e

      for each Z-layout z_{ij} of e_i

        L_{ij} = LeastCost(z_{ij},T_{e_i})

    for each combination $(L_{1x}, L_{2y}, \ldots)$

      $M_{xy\ldots}$ = Layouts $L_{1x}, L_{2y}, \ldots$ merged with z_e

    return minimum cost $M_{xy\ldots}$

end

(D)

Figure 130: (A) shows a separable MST, (B) shows the MST in (A) hung by leaf 5, (C) shows a generic MST hung by one of its leaves upon which the LeastCost function operates, and (D) shows the code for the LeastCost function.

- Integer Programming For Global Routing
  - Concurrent Layout
  - Hierarchical Layout
- Detailed Routing (Channel Routing)
  - Introduction
  - Left Edge Algorithm (LEA)
  - Greedy
- Sequence Matching And Its Use In Computational Biology
  - DNA
  - Proteins
  - Human Genome Project
  - Sequence Alignment

1 Integer Programming For Global Routing

1.1 Concurrent Layout

Concurrent layout optimizes all nets together rather than adding one net to the solution at a time. The goal is to minimize total wire length, and it can be met with an integer program.

Recall that a net is a set of terminals that must be all connected to each other, and that a netlist is a set of nets that need to be routed. Suppose that the netlist consists of n nets. Let $T_i = \{T_{i1}, T_{i2} \ldots T_{il_i}\}$ be all the Steiner trees that connect net i (l_i is the number of such trees). The following integer program finds and optimal route for our nets.

$$
\begin{array}{lll}
\text{minimize} & \sum_{i=1}^{n} \sum_{j=1}^{l_i} L(T_{ij}) \cdot x_{ij} & \\
\text{subject to} & \sum_{j=1}^{l_i} x_{ij} = 1 & i = 1, 2 \ldots, n \\
 & \sum_{i,j:e \in T_{ij}} x_{ij} \leq C(e) & e \in E \\
\text{where} & x_{ij} = \begin{cases} 1 & \text{if } T_{ij} \text{ is used} \\ 0 & \text{otherwise} \end{cases} & \\
 & L(T_{ij}) = \text{total length of tree } T_{ij} & \\
 & C(e) = \text{capacity restriction of edge } e & \\
 & E = \text{set of all edges} &
\end{array}
$$

It minimizes the total length of all included Steiner trees across all nets. The first constraint ensures that only one tree per net is included while the second ensures that the solution

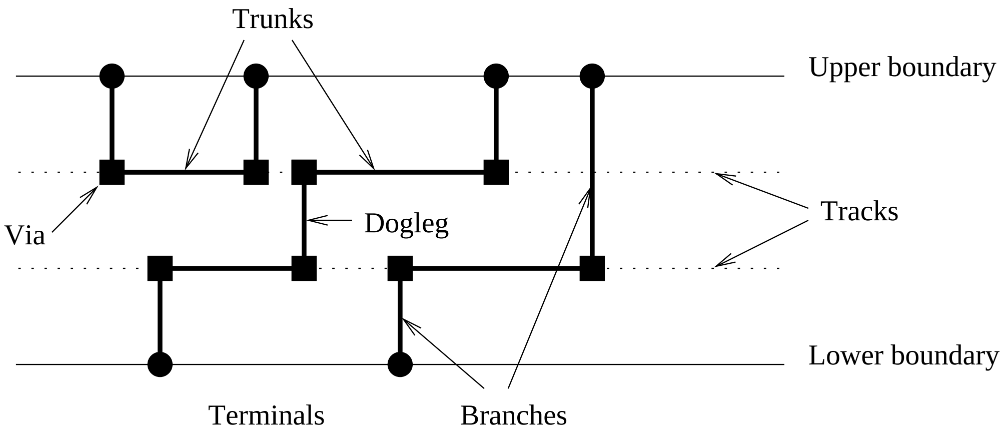


Figure 131: Channel routing definitions.

does not exceed the capacity of any edge. Although this is a nice concise description of the problem, it is infeasible to solve for any reasonably sized circuit. To make the problem more feasible, a hierarchical application of the approach is used.

1.2 Hierarchical Layout

In hierarchical layout, a floorplan is preprocessed by partitioning it into a hierarchy of sub-blocks. At each node of the resulting tree, the sub-blocks can then be routed with a modified version of the concurrent routing program. The modification involves replacing the first constraint with

$$\sum_{j=1}^{l_i} x_{ij} = n_i \quad i = 1, 2 \ldots, n$$

where n_i is the number of nets that connect the same set of blocks. (In partitioning the floorplan, we create equivalence classes of nets. Since the algorithm actually routes these equivalence classes, the n_i account for the fact that they may contain more than one net.) The modified program is solved at each node in the tree in top-down order.

The number of variables in both the concurrent and hierarchical programs can be reduced using various techniques; see chapter 6 of N. Sherwani, *Algorithms for VLSI Physical Design Automation*, Kluwer, 1995.

2 Detailed Routing (Channel Routing)

2.1 Introduction

A channel is the space between two standard-cell blocks, and is used for routing wires between the terminals of those and other blocks. This type of routing is known as channel routing. For channel routing, we know where wires enter the channel (at the terminals of the blocks that bound the channel). The goal is to optimize the routing of wires within the channel to allow the best compaction of blocks afterward. Figure 131 describes some terminology for channel routing.

Switchboxes are formed by the intersection of two channels. Routing algorithms for switchboxes are slightly different from those for channels, but they have the same flavor.

Channel routing can be grid-based or gridless. Grid-based routing requires that wires fit on a regular grid. Gridless routing is more flexible in its wire placement. This flexibility makes it useful for routing wires of different thicknesses, which arise when the wires must carry different currents.

Semiconductor technology usually provides multiple wiring layers, and channel routing algorithms must be able to make use of these layers. Two types of model exist for the use of multiple layers: reserved and unreserved. Reserved layer-models restrict all wires within a given layer to running in a single orientation (either horizontally or vertically), and are commonly used for routing programmable logic arrays (PLA's). An unreserved layer-model, in which wires can be routed both horizontally and vertically within the same layer, is more flexible.

There are four popular two-layer routing algorithms.

- Left-edge algorithm (LEA) (Hashimoto and Stevens 1971)
- Constraint graph (1982)
- Greedy (Rivest Fiduccia1982)
- Hierarchical (1983)

LEA and Greedy are discussed below.

2.2 Left Edge Algorithm (LEA)

The three steps of LEA are as follows.

1. Determine the interval of each net. The interval of a net is simply the region bounded by its leftmost and rightmost terminals. ($\mathrm{O}(n)$)
2. Sort these intervals by their leftmost points. ($\mathrm{O}(n \log n)$)
3. For each interval, find the first free track and place the interval on that track. Information about currently-occupied tracks can be maintained in a balanced tree so that inserting, deleting, and finding the minimum take logarithmic time.

This algorithm effectively does plane-sweeping, which appears in computational geometry. In plane-sweeping, elements are sorted in one of two dimensions, and then swept through in the other. Figure 132 shows an example.

Simple LEA has two limitations.

- Each net must be routed on a single track. This limits the quality of the resultant routes.
- It cannot account for vertical constraints. This makes some nets impossible to route.

Various extensions (such as allowing doglegs) solve these problems.

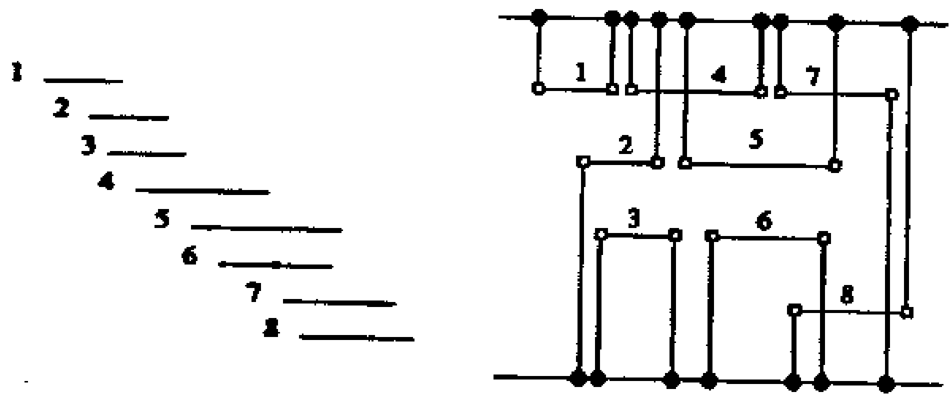


Figure 132: Example of the LEA channel routing algorithm. The left half show the intervals that are routed in the right half.

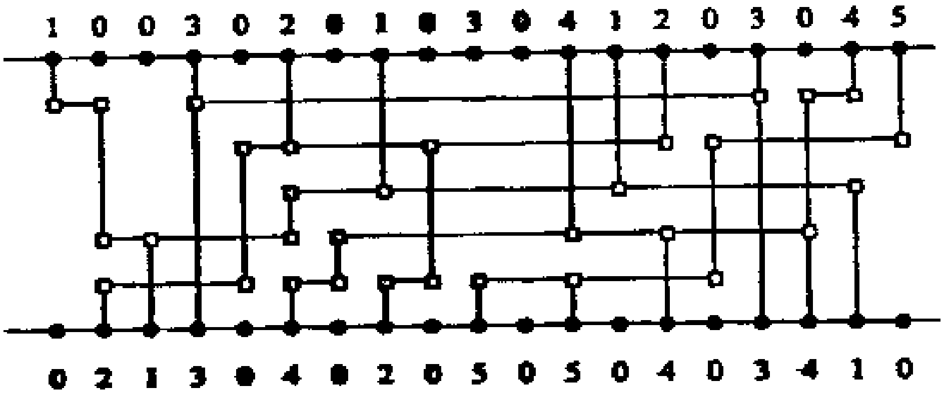


Figure 133: Example of the greedy channel router.

2.3 Greedy

Here is the greedy channel router algorithm. For each column i in order:

1. Connect the terminals at column i either to the tracks containing their nets or to the first empty tracks. Note that connecting both terminals to tracks containing their nets might not be possible, in which case one of the tracks must be split. Splitting a track involves simply using two tracks (and thus two parallel wires) for some part of a single net.

2. Join any tracks that were split during the previous iteration (i.e., that were split when processing the previous column).

3. If any split tracks could not be joined in the previous step, reduce the distance between those tracks.

4. Move each net toward the boundary on which its next terminal lies.

Figure 133 shows an example of a greedily-routed channel.

3 Sequence Matching And Its Use In Computational Biology

3.1 DNA

DNA is a sequence of base-pairs (bps) of the four nucleotide bases—Adenosine, Cytosine, Thymine, and Guanine (or a, c, t, and g). A human genome consists of DNA: approximately $3 \cdot 10^9$ bps divided into 46 chromosomes, each with between $5 \cdot 10^7$ and $25 \cdot 10^7$ bps. Each chromosome—because it is DNA—is a sequence of bps, and is used to generate proteins:

$$\text{DNA} \stackrel{\text{transcription}}{\longrightarrow} \text{mRNA} \stackrel{\text{translation}}{\longrightarrow} \text{Protein}$$

3.2 Proteins

Proteins comprise 20 different amino acids (gly, trp, cys, ...). Each bp triple in DNA codes one of these amino acids. Since there are 64 possible triples, this is a many-to-one mapping. (Four of triples serve to mark the beginnings and ends of proteins rather than coding amino acids). The structure of a protein, as determined by its amino acids, controls how it folds, which in turn determines its function. Protein folding—and therefore function—can be computed as an N-body problem.

Goals in molecular biology are

1. to extract genome sequences for various organisms,
2. to determine the structure and purpose of the proteins coded by these sequences, and
3. to use that knowledge to study and cure genetic diseases, design new drugs, and study evolution.

3.3 Human Genome Project

The goal of the human genome project is to generate a map of each human chromosome. The basic approach is as follows.

1. Divide chromosomes using restriction enzymes, which function as microscopic scalpels. These enzymes break chromosomal DNA between different bps.
2. Clone the DNA fragments using host cells (usually bacteria).
3. Find an exact map of a portion of the fragments (e.g., the first and last 600 bps). This can be done with a divide-and-conquer technique.
4. Use sequence alignment to determine how the mapped fragments fit together.
5. Find an exact map of the desired subset of the fragments.

3.4 Sequence Alignment

Sequence alignment answers this question: "How similar are two bp sequences?" It tries to find the closest match, allowing for mutations (X $\rightarrow$ Y), insertions (_ $\rightarrow$ Y), and deletions (X $\rightarrow$ _), but minimizing the number of these differences. For example,

```
acta_gtc__tac
 | | |||  ||
_cgacgtcgata_
```

contains one mutation, three insertions, and two deletions. Notice that mutations can be modeled as an insertion-deletion pair (_X $\rightarrow$ Y_). Insertions and deletions are know collectively as indels.

There are three types of sequence alignment.

Global Aligns two sequences A and B, like the `diff` utility.

Local Aligns A with any part of B (i.e., finds the best match or matches for A within B), like the `agrep` utility.

Multiple Finds matches among n sequences (*i.e.*, align them all up together).

For both global and local alignment, we may want to find the n best matches rather than the single best match.

Multiple alignment problems typically are NP-hard, so approximations are used.

- Longest Common Subsequence (LCS)
- Global sequence alignment
  - Recursive algorithm
  - Memoizing
  - Dynamic programming
  - Space efficiency
- Gap models
- Local alignment
- Multiple alignment
- Biological applications

1 Longest Common Subsequence

Definition: A **subsequence** is any subset of the elements of a sequence that maintains the same relative order. If A is a subsequence of B, this is denoted by $A \subset B$.

For example, if $A = a_1a_2a_3a_4a_5$, the sequence $A' = a_2a_4a_5$ is a subsequence of A since the elements appear in order, while the sequence $A'' = a_2a_1a_5$ is not a subsequence of A since the elements a_2 and a_1 are reversed.

The Longest Common Subsequence (LCS) of two sequences A and B is a sequence C such that $C \subset A$, $C \subset B$, and $|C|$ is maximized.

Example:

$A =$ `abacdac`; $B =$ `cadcddc`; $C =$ `acdc`

```
_abacdac
 |  || |
cad_cddc
```

In this example, C is a LCS of A and B since it is a subsequence of both and there are no subsequences of length 5.

Applications:

- The UNIX `diff` command works by finding the LCS of two files, where each line is treated as a character. It then prints out lines which are not in the LCS, indicating insertions, deletions, and changes.

- For screens A and B, we can define the edit distance of A and B, $D(A,B) = |A| + |B| - 2|LCS(A,B)|$. The edit distance gives the number of characters in either screen which are not in the LCS, and hence measures the number of commands needed to change the screen from A to B. Maximizing the LCS minimizes the edit distance.

2 Global Sequence Alignment

The global sequence alignment problem is a generalization of LCS; instead of only noticing whether characters are the same or different, we have an arbitrary cost function which defines the relationship between characters. (In what follows, we will be maximizing "costs"; hence, the cost function is really a measure of similarity.)

Let $c(a,b)$ be a cost function, giving a value for any pair a and b of alphabet symbols, including the blank character (_). We can formally define an alignment of A and B as a pair of sequences A' and B' which are just A and B with added blanks, and for which $|A'| = |B'| = l$. The value of an alignment is given by

$$V(A', B') = \sum_{i=1}^{l} c(A'[i], B'[i]).$$

Our goal then is to find an optimal alignment, i.e. one whose value is a maximum.

Note that if we define costs so that $c(x,y) = 1$ if $x = y$, and 0 otherwise, we get LCS as a special case.

Example:

$A =$ `abacdac`, $B =$ `cadcddc`

```
A' = _abacdac          A'' = aba_cdac
       |  || |                |  || |
B' = cad_cddc          B'' = _cadcddc
```

Both of these alignments have the same value when we use the LCS cost function, but for the following cost function (which treats b's and c's as being similar), (A'', B'') has a higher value.

	a	b	c	d	_
a	2	0	0	0	0
b	0	2	1	0	0
c	0	1	2	0	0
d	0	0	0	2	0
_	0	0	0	0	-1

Notice the negative cost of two blanks. This is necessary to prevent unnecessary padding of sequences with extra blanks.

2.1 Cost Functions for Protein Matching

With proteins, we have an alphabet of 20 amino acids, and hence a cost matrix with 210 entries. (We assume the matrix is symmetric, since we have no reason to order the proteins we are comparing.) There are a number of possible cost functions:

- **Identity:** We could use the LCS cost function and view amino acids as just being the same or different.
- **Genetic code:** We can assign costs which are inversely related to the minimum number of DNA changes required to convert a DNA triple for one amino acid into a DNA triple for the other.
- **Chemical similarity:** We can take into account the relative sizes, shapes, and charges of different amino acids.
- **Experimental:** We could use standard matrices (e.g. Dayhoft, Blosum) which are based on observed properties of amino acids and/or their observed relative effect on tertiary structure.

In practice the cost function of choice is application specific and a topic of hot debate.

3 Sequence Alignment Algorithms

3.1 Recursive Algorithm

A recursive solution to the sequence alignment problem takes the last character of each sequence and tries all three possible alignments: either the characters are aligned together, the first character is aligned with a blank, or the second character is aligned with a blank. Taking the maximum over all three choices gives a solution. ML-style code for this would look like the following:

```
OptAlgn(_,_)      = C(_,_).
OptAlgn(_,A:a)    = C(_,a) + OptAlgn(_,A).
OptAlgn(B:b,_)    = C(b,_) + OptAlgn(B,_).
OptAlgn(B:b,A:a) = max(C(b,a) + OptAlgn(B,A),
                        C(b,_) + OptAlgn(B,A:a),
                        C(_,a) + OptAlgn(B:b,A)).
```

For the special case of LCS, all costs involving a blank are 0, and we know that it is always to our advantage to align characters if they match. Hence, we get the following code:

```
LCS(_,A:a)   = 0.
LCS(B:b,_)   = 0.
LCS(B:x,A:x) = 1 + LCS(B,A).
LCS(B:b,A:a) = max(LCS(B,A:a), LCS(B:b,A)).
```

A naive implementation of this recursive solution has an exponential running time, since each call spawns three recursive calls over which we take the maximum. This is easily rectified by memoizing, however.

3.2 Memoizing

If $|A| = n$ and $|B| = m$, note that there are only nm distinct calls we could make to `OptAlgn`. By simply recording the results of these calls as we make them, we get an $O(nm)$ algorithm. The code for LCS follows: (We assume the matrix M is initialized to `INVALID`.)

```
int LCS(int i, int j) {
  if (M[i,j] != INVALID) return M[i,j];
  if (i == 0 || j == 0) r = 0;
    else if (A[i] == B[j]) r = 1 + LCS(i-1, j-1);
    else r = max(LCS(i-1, j), LCS(i, j-1));
  return M[i,j] = r;
}
```

3.3 Dynamic Programming

Instead of filling in the matrix of values as they are computed, we can fill in the values row by row. This is the dynamic programming solution, for which we have the following code:

```
for i = 1 to n
  for j = 1 to m
    if (A[i] == B[j]) M[i,j] = 1 + M[i-1,j-1];
    else M[i,j] = max(M[i-1,j], M[i,j-1]);
```

For example, with $A =$ `tcat` and $B =$ `atcacac`, we get the following matrix:

	_	a	t	c	a	c	a	c
_	0	0	0	0	0	0	0	0
t	0	0	1	1	1	1	1	1
c	0	0	1	2	2	2	2	2
a	0	1	1	2	3	3	3	3
t	0	1	2	2	3	3	3	3

To actually find the optimal alignments, we can think of each entry as having edges to the neighboring entries from which it could have been calculated. That is, each entry can have edges pointing to its upper, left, and upper left neighbors, depending on whether it could have been calculated by adding one to its diagonal neighbor and whether its value is the same as either its upper or left neighbors. A portion of the induced graph for the matrix above looks like:

$$
\begin{array}{c|ccccccc}
 & \mathrm{t} & & \mathrm{c} & & \mathrm{a} & & \mathrm{c} \\
\hline
\mathrm{t} & 1 & \leftarrow & 1 & \leftarrow & 1 & \leftarrow & 1 \\
 & \uparrow & \nwarrow & & & & \nwarrow & \\
\mathrm{c} & 1 & & 2 & \leftarrow & 2 & \leftarrow & 2
\end{array}
$$

An optimal alignment is then a path from the lower right corner of the table to the upper left. In the matrix fragment above, the two paths correspond to aligning the `tc` as `tc_` or as `t_c`. Finding all possible paths gives us all optimal alignments. As with memoizing, this algorithm runs in $O(nm)$ time.

3.4 Space Efficiency

Our time bound is essentially the best possible for the general case[5], but we would like to use less space. In particular, notice that each row of the matrix depends only on the previous row. Hence, by filling in the matrix row-by-row and reusing the space for each row, we could compute the value of the optimal alignment using just $O(m)$ space. Without something clever, however, this does not give us an actual alignment since we throw away the path as we go (remember that we have to traverse backwards through the matrix to generate the alignment path).

(As an aside, note that if we were to run this algorithm in parallel, we would want to fill in the matrix diagonal by diagonal so that each entry would be independent of the others currently being worked on.)

In order to actually generate an alignment using only $O(m)$ space, we use the following divide-and-conquer algorithm:

1. Calculate the entries of the matrix row-by-row, discarding intermediate results, until we reach the middle row.

2. Once we get to the middle row as we calculate each additional row, we keep track of where in the middle row each new entry came from. (This is easily derived from the corresponding information about the previous row.) If we have more than one possibility, we choose arbitrarily. (Hence, this algorithm will only output one solution.)

3. When we reach the last row, we will know which entry in the middle row the bottom right corner came from (*i.e.* we will know the middle node along the solution path). Suppose that middle row entry is entry k. Now we recursively solve the problem of finding the path from the bottom right to $(n/2, k)$ and from $(n/2, k)$ to the upper left.

It may seem that this algorithm is doing redundant work since it recursively resolves the same solution over and over again. Although this is indeed true note, however, that in the recursion the algorithm only needs to recur on the upper left and bottom right quadrants. Hence, it is doing half as much work at each level of the recursion, leading to a constant factor increase in time, not a log factor as one might suppose. More formally, the recurrence we get is:

$$T(n, m) = T(n/2, k) + T(n/2, m - k) + O(nm) = O(nm).$$

[5] We will describe a more efficient method for small edit-distances in the next class.

Since we only keep track of an extra set of pointers one per column, this algorithm uses only $O(m)$ space. (A slight optimization is to orient our matrix so that m is the smaller of the two dimensions.) The algorithm was invented in the context of finding the longest common subsequence by Hirschberg ("A linear-space algorithm for computing maximal common subsequences", Communications of the ACM, 18:107–118, 1975) and generalized to alignment by Myers and Miller ("Optimal alignments in linear space", Computer Applications in the Biosciences, 4:11–17, 1988).

4 Gap Models

For many applications, consecutive indels (insertions or deletions, also called gaps) really need to be treated as a unit. In other words, a gap of 2 characters isn't really twice as bad as a gap of 1 character. There are various ways of dealing with this problem, depending on what sort of gap model we want to allow.

In the most general case, we can define arbitrary costs x_k for gaps of length k. For this we have the Waterman-Smith-Beyer algorithm:

$$\begin{aligned}
S_{00} &= 0 \\
S_{i0} &= x_i \\
S_{0j} &= x_j \\
S_{ij} &= \max \begin{cases} S_{i-1,j-1} + C(a_i, b_j) \\ \max_k (S_{i,j-k} + x_k) \\ \max_k (S_{i-k,j} + x_k) \end{cases}
\end{aligned}$$

This is similar to our previous algorithms, except that for each entry, instead of only looking at gaps of length 1, which corresponds to looking at the left and upper neighbors in the matrix, we look at all possible gap lengths. To calculate S_{ij} this requires us to look at the entire row i and column j generated so far. Hence, the running time of the algorithm is $O(nm^2 + n^2m)$. Furthermore, since we need all previous rows, we cannot make this algorithm space efficient.

In practice, our gap models are likely to be relatively simple, and for these simpler models, we can again achieve the time/space bounds we were able to get before. For example, with an affine gap model, we have $x_k = \alpha + \beta k$. (This is popular in computational biology, where something like $x_k = -(1 + k/3)$ is typical.) An algorithm due to Gotoh for the affine gap model is as follows:

$$\begin{aligned}
E_{ij} &= \max \begin{cases} E_{i-1,j} + \beta \\ S_{i-1,j} + \alpha + \beta \end{cases} \quad E_{0j} = -\infty \\
F_{ij} &= \max \begin{cases} F_{i,j-1} + \beta \\ S_{i,j-1} + \alpha + \beta \end{cases} \quad F_{i0} = -\infty \\
S_{ij} &= \max \begin{cases} S_{i-1,j-1} + C(a_i, b_j) \\ E_{ij} \\ F_{ij} \end{cases} \quad \begin{matrix} S_{0j} = \alpha + \beta j \\ S_{i0} = \alpha + \beta i \end{matrix}
\end{aligned}$$

The E_{ij} matrix calculates optimal alignments in which the second sequence ends in blanks, the F_{ij} matrix calculates optimal alignments in which the first sequence ends in blanks, and the S_{ij} matrix calculates overall optimal alignments. The basic idea is that since each additional blank (after the first one) has the same cost, we can essentially use the same algorithm as before, just using the E and F values to be sure we add in α when we introduce the first blank in a gap. This algorithm runs in $O(nm)$ time and can be made space efficient as before.

The following is a summary of some different gap models and the running times of their associated algorithms.

Gap function form	Running time
General	$O(nm^2 + n^2m)$
Linear ($x_k = \alpha + \beta k$)	$O(nm)$
Logarithmic ($x_k = \alpha + \beta \log k$)	$O(nm)$
Concave downwards	$O(nm \log n)$
Piecewise linear, l segments	$O(lnm)$

5 Local Alignment

Sometimes instead of aligning two entire sequences, we want to align a smaller sequence at multiple locations within a larger one. For example, to find areas of DNA which might have similar functionality to some segment, we can find the best matches to our segment within the DNA.

Global alignment algorithms are easily converted into local ones by simply adding an extra argument 0 to the max function computing the optimal alignment. So, for example, for the general gap model

$$S_{ij} = \max \begin{cases} S_{i-1,j-1} + C(a_i, b_j) \\ \max_k(S_{i,j-k} + x_k) \\ \max_k(S_{i-k,j} + x_k) \\ 0 \end{cases}$$

This has the effect of not penalizing an alignment for the large gaps at the ends of the sequence. For example, with

$$C(a,b) = \begin{cases} 1 & \text{if } a = b \\ -1/3 & \text{if } a \neq b \end{cases}$$

and $x_k = -(1 + k/3)$, we get the following alignment matrix.

	_	a	t	c	a	c	a	c
_	0	0	0	0	0	0	0	0
t	0	0	1	0	0	0	0	0
c	0	0	0	2	2/3	1	0	1
a	0	1	0	2/3	3	5/3	2	2/3
t	0	0	2	2/3	5/3	8/3	4/3	5/3

In this case, the best match aligns `tca` with `tca` (the one 3 in the matrix). In general, we might want to find multiple matches with high scores. In this case we might pick out the highest k entries from the matrix, although we probably don't want to include entries that are just extensions of previous matches we have found. In the above example we might not want to pick the value 8/3 that follows 3, but instead pick the 2.

6 Multiple Alignment

Another problem is to align multiple sequences at once so as to maximize the number of pairs of aligned symbols. This is used to reconstruct a sequence from its overlapping fragments; the overlaps will not be perfect because of imprecision in the sequencing of the fragments. Alternatively, we can also use multiple alignment with sequences from different family members, in order to find genes shared by the family members.

Unfortunately, multiple alignment is NP-hard. Hence, there are two types of algorithms: those that take exponential time and those that are approximate. In exponential time, we can extend our earlier dynamic programming algorithms to work in a p-dimensional matrix, where p is the number of sequences we are trying to align. This takes $O(n^p)$ time and space and is impractical for p more than about 4.

Alternatively, we can extend the pairwise methods hierarchically to get an approximate solution. For this, we do all pairwise comparisons, cluster the results, and then build an alignment bottom-up from the cluster tree.

7 Biological Applications

One of the most intensive uses of pattern matching is in database search engines used by biologists to compare new sequences with databases of old ones. These databases can have as many as 100,000s of sequences with querying by web or email. Such shared search engines are useful because algorithms and data are updated more rapidly and more centrally, and considerable computing power can be brought to bear on the problem. Both local and global alignment algorithms are used.

On a smaller scale, pattern matching is also used for genome sequencing. This can be done in one of two ways. The first method involves sequencing the 600 bp (base pair) ends of the fragments of a sequence, which are then assembled using multiple alignment. Alternatively, one piece can be fully sequenced, and the rest of the sequence can be constructed outward using single alignment.

Notes by Helen J. Wang

- Pattern Matching (continued)
  1. Ukkonen's Algorithm for more efficient dynamic programming
- Introduction to Indexing and Searching
- Inverted file indexing
  1. Compressing the posting lists
  2. Representing and accessing the lexicon

1 Pattern Matching (continued)

So far we have discussed algorithms that take at least $O(nm)$ time for two sequences of length n and m. In an application such as `diff` this can be prohibitively expensive. If we had two files of $100,000$ lines each, the algorithm would require 10^{10} string compares, which would take hours. However, `diff` can actually run much faster than this when the files to be compared are similar (*i.e.*, the typical situation in which they both come from the same source and just involve minor changes).

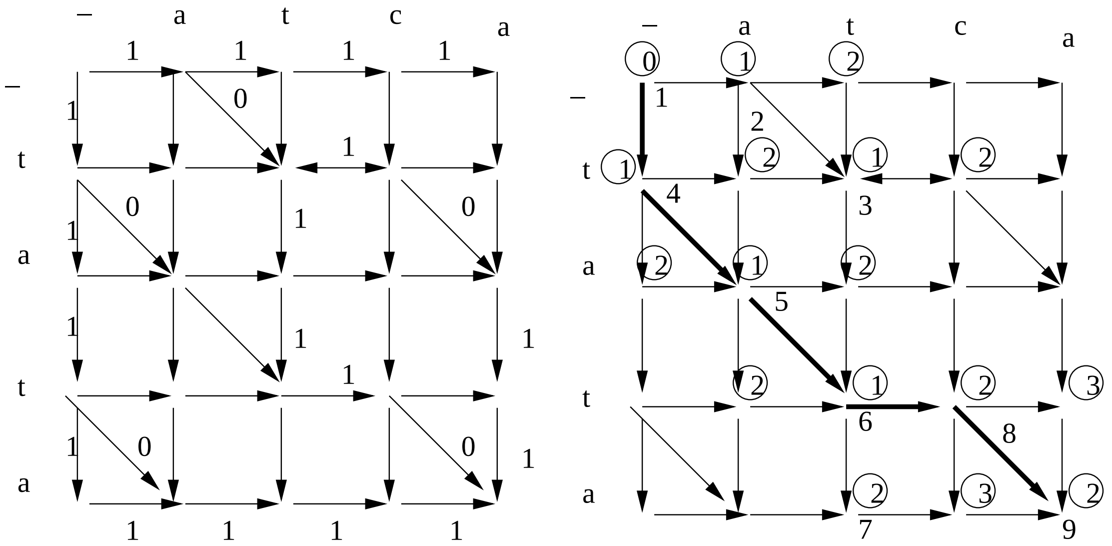
We will now consider an algorithm for edit-distance which takes $O(\min(n, m)D)$ time, where D is the edit distance between the two strings. The runtime is therefore output dependent. This algorithm can also be used for the longest common subsequence and a variant is used in the `diff` program.

1.1 Ukkonen's Algorithm

Consider the minimum edit distance problem:

$$\begin{aligned} D_{i0} &= i \\ D_{0j} &= j \\ D_{ij} &= \begin{cases} D_{i-1,j-1} & a_i = b_i \\ 1 + min(D_{i,j-1}, D_{i-1,j}) & \text{otherwise} \end{cases} \end{aligned} \tag{8}$$

This problem can be viewed as a weighted directed graph laid over the $n \times m$ matrix of $D_{i,j}$. It can then be solved by finding a shortest path in the graph from $D_{0,0}$ to $D_{n,m}$. This graph has a unit weight edge between neighbors in the same row or column. These edges represent an insertion or deletion—the $1 + min(D_{i,j-1}, D_{i-1,j})$ case in Equation 8. The graph also has zero weight diagonal edges from $D_{i-1,j-1}$ to $D_{i,j}$ whenever $a_i = b_i$—representing the $a_i = b_i$ case in Equation 8. Figure 134(a) shows an example of such a graph. In this graph the length of a path represents the cost of those set of modifications. Therefore the shortest path from $D_{0,0}$ to $D_{n,m}$ gives the set of changes for the minimum edit distance. Note that we don't actually have to construct the graph since we can determine the edges from a location i, j simply by looking at a_i, a_{i+1}, b_j and b_{j+1}.



(a) Graph (b) Shortest Path

Figure 134: Graph representation of the minimum edit distance problem for the strings atca and tata. In the right figure the circled values specify the distances and the uncircled values specify the visiting order for Dijkstra's algorithm. The shaded path represents the shortest path.

To find the shortest path we can use a standard algorithm such as Dijkstra's algorithm (Figure 134(b) gives an example). Since all the fringe nodes in the algorithm will have one of two priorities (the distance to the current vertex, or one more) the algorithm can be optimized to run in constant time for each visited vertex. If we searched the whole graph the running time would therefore be $O(nm)$, which is the same as our previous algorithms for edit distance. As we will show, however, we can get much better bounds on the number of vertices visited when the edit distance is small.

Theorem 3 $D_{ij} \geq |j - i|$

Proof: All edges in the graph have weight ≥ 0 and all horizontal and vertical edges have weight 1. Since $|j - i|$ is the minimum distance in edges from the diagonal, and the path starts on the diagonal (at 0,0), any path to location (i, j) must cross at least $|j-i|$ horizontal and vertical edges. The weight of the path must therefore be at least $|j - i|$. $\square$

Given that Dijkstra's algorithm visits vertices in the order of their distance D_{ij}, this theorem implies that the algorithm will only search vertices that are within D_{nm} of the diagonal (see Figure 135). This leads to a running time of

$$T = O(D_{nm}\min(n,m))$$

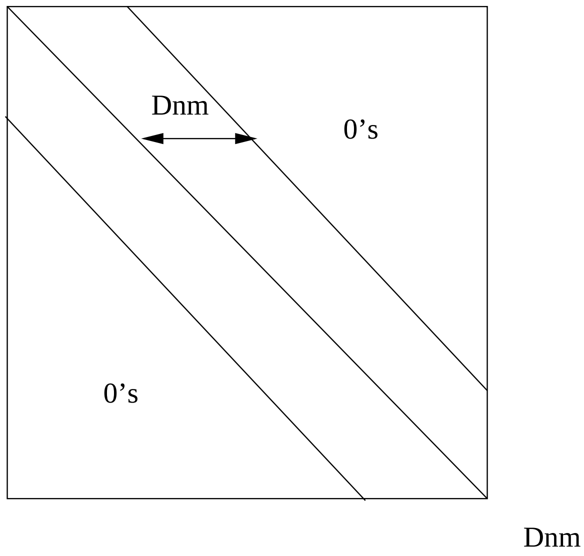


Figure 135: Bounding visited vertices

In practice programs such as `diff` do not use Dijkstra's algorithm but instead they use a method in which they try increasingly large bands around the diagonal of the matrix (see Figure 136). On each iteration the width of the band is doubled and the algorithm only fills in the entries in the band. This is continued while $D_{nm} > w/2$ where w is the width of the band. The code for filling each band is a simple nested loop.

The technique can be used in conjunction with our previous divide-and-conquer algorithm to generate a space-efficient version.

2 Introduction to Indexing and Searching

Indexing addresses the issue of how information from a collection of documents should be organized so that queries can be resolved efficiently and relevant portions of the data extracted quickly. We will cover a variety of indexing methods. To be as general as possible, a *document collection* or *document database* can be treated as a set of separate documents, each described by a set of *representative terms*, or simply *terms* (each term might have additional information, such as its location within the document). An index must be capable of identifying all documents that contain combinations of specified terms, or that are in some other way judged to be relevant to the set of query terms. The process of identifying the documents based on the terms is called a *search* or *query* of the index. Figure 137 illustrates the definitions.

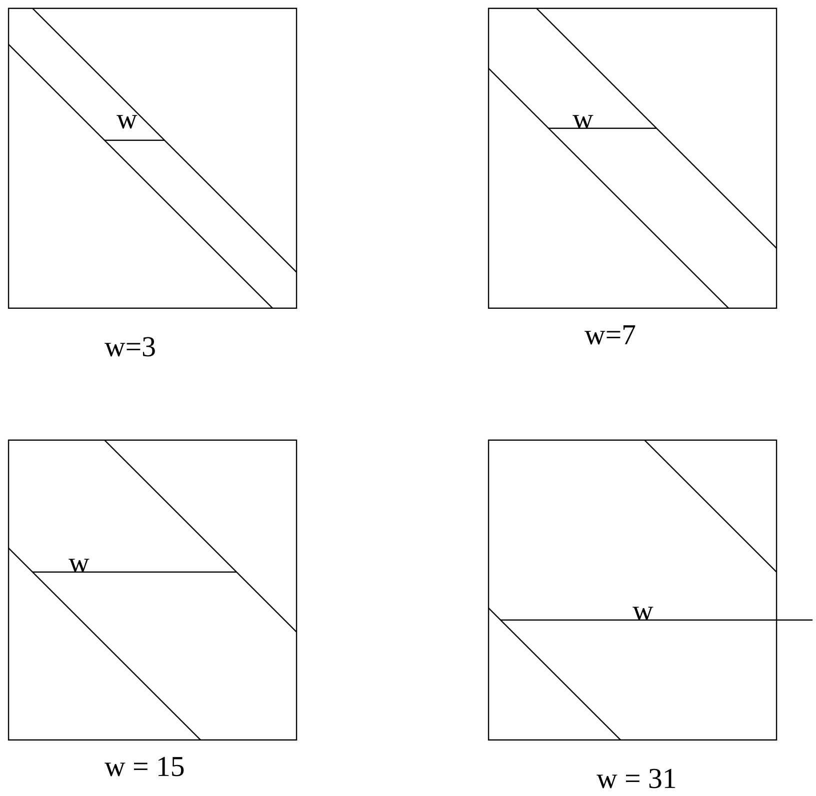


Figure 136: Using increasingly wide bands.

Applications of indexing:

Indexing has been used for many years in a wide variety of applications. It has gained particular recent interest in the area of web searching (e.g. AltaVista, Hotbot, Lycos, Excite, ...). Some applications include

- Web searches
- Library article and catalog searches
- Law, patent searches
- Information filtering, e.g. get interesting New York Time articles.

The goals of these applications:

- Speed – want minimal information retrieval latency
- Space – storing the document and indexing information with minimal space
- Accuracy – returns the "right" set of documents
- Updates – ability to modify index on the fly (only required by some applications)

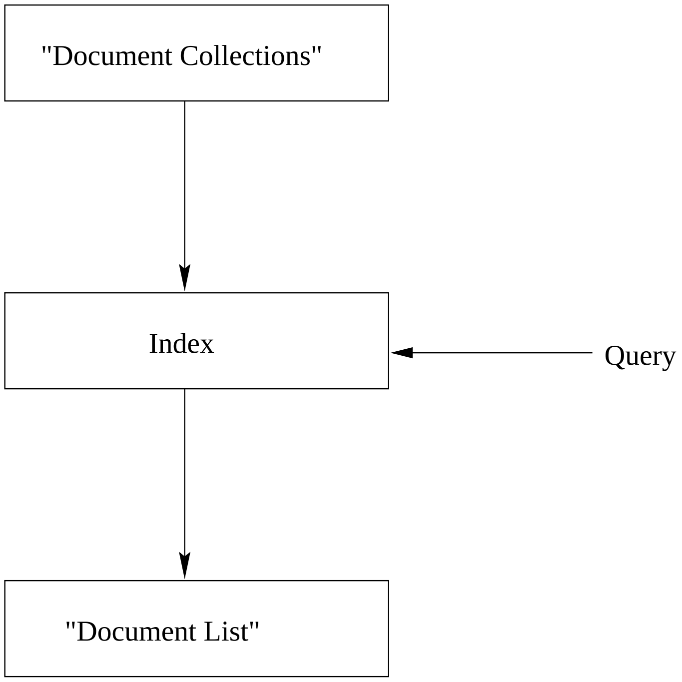


Figure 137: Overview of Indexing and Searching

The main approaches:

- Full text scanning (e.g. grep, egrep)
- Inverted file indexing (most web search engines)
- Signature files
- Vector space model

Types of queries:

- boolean (*and, or, not*)
- proximity (*adjacent, within*)
- key word set
- in relation to other documents (relevance feedback)

Allowing for:

- prefix matches (AltaVista does this)
- wildcards
- edit distance bounds (egrep)

Techniques used across methods

- case folding: London = london
- stemming: *compress* = *compression* = *compressed*
  (several off-the-shelf English language stemmers are available)
- ignore stop words: *to, the, it, be, or, ...*
  Problems arise when search on *To be or not to be* or the month of *May*
- Thesaurus: *fast* = *rapid*
  (handbuilt clustering)

Granularity of Index

The *Granularity* of the index refers to the resolution to which term locations are recorded within each document. This might be at the document level, at the sentence level or exact locations. For proximity searches, the index must know exact (or near exact) locations.

3 Inverted File Indexing

Inverted file indices are probably the most common method used for indexing documents. Figure 138 shows the structure of an inverted file index. It consists first of a *lexicon* with one entry for every term that appears in any document. We will discuss later how the lexicon can be organized. For each item in the lexicon the inverted file index has an *inverted file entry* (or *posting list*) that stores a list of pointers (also called *postings*) to all occurrences of the term in the main text. Thus to find the documents with a given term we need only look for the term in the lexicon and then grab its posting list. Boolean queries involving more than one term can be answered by taking the intersection (conjunction) or union (disjunction) of the corresponding posting lists.

We will consider the following important issues in implementing inverted file indices.

- How to minimize the space taken by the posting lists.
- How to access the lexicon efficiently and allow for prefix and wildcard queries.
- How to take the union and intersection of posting lists efficiently.

3.1 Inverted File Compression

The total size of the posting lists can be as large as the document data itself. In fact, if the granularity of the posting lists is such that each pointer points to the exact location of the term in the document, then we can in effect recreate the original documents from the lexicon and posting lists (*i.e.*, it contains the same information). By compressing the posting lists we can both reduce the total storage required by the index, and at the same time potentially reduce access time since fewer disk accesses will be required and/or the compressed lists can

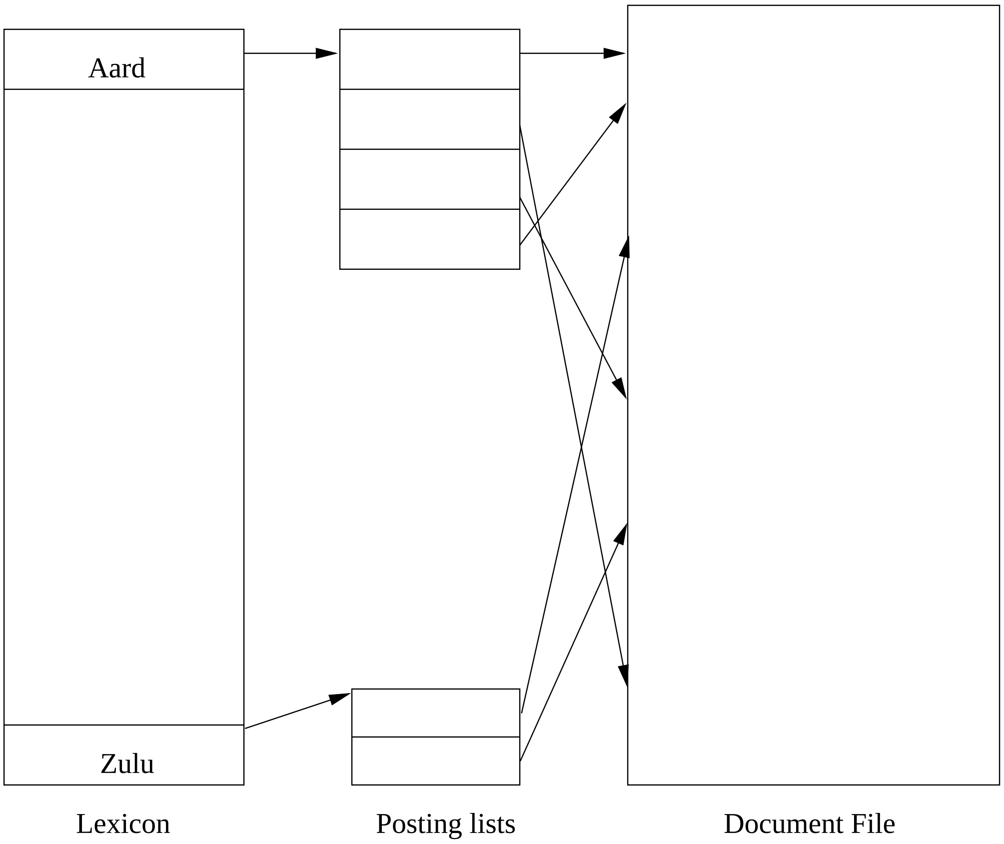


Figure 138: Structure of Inverted Index

fit in faster memory. This has to be balanced with the fact that any compression of the lists is going to require on-the-fly uncompression, which might increase access times. In this section we discuss compression techniques which are quite cheap to uncompress on-the-fly.

The key to compression is the observation that each posting list is an ascending sequence of integers (assume each document is indexed by an integer). The list can therefore be represented by a initial position followed by a list of gaps or deltas between adjacent locations. For example:

original posting list: elephant: [3, 5, 20, 21, 23, 76, 77, 78]

posting list with deltas: elephant: [3, 2, 15, 1, 2, 53, 1, 1]

The advantage of using the deltas is that they can usually be compressed much better than indices themselves since their entropy is lower. To implement the compression on the deltas we need some model describing the probabilities of the deltas. Based on these probabilities we can use a standard Huffman or Arithmetic coding to code the deltas in each posting list. Models for the probabilities can be divided into global or local models (whether the same probabilities are given to all lists or not) and into fixed or dynamic (whether the probabilities are fixed independent of the data or whether they change based on the data).

An example of a fixed model is the γ code. Think of a code in terms of the implied probability distribution $P(c) = 2^{-l(c)}$. This is the inverse of the definition of entropy. For

Decimal	γ Code
1	0
2	100
3	101
4	11000
5	11001
6	11010
7	11011
8	1110000

Table 13: γ coding for 1 through 8

example, binary code gives a uniform distribution since all codes are the same length, while the unary codes ($1 = 0, 2 = 10, 3 = 110, 4 = 1110, \ldots$) gives an exponential decay distribution ($p(1) = 1/2, p(2) = 1/4, p(3) = 1/8, p(4) = 1/16, \ldots$). The γ code is in between these two extreme probability distributions. It represents the integer i as a unary code for $1+\lfloor(\log_2(i))\rfloor$ followed by the binary code for $i-2^{\lfloor \log_2(i)\rfloor}$. The unary part specifies the location of the most significant non-zero bit of i, and then the binary part codes the remaining less significant bits. Figure 139 illustrates viewing the γ codes as a prefix tree, and Table 13 shows the codes for 1-8. The length of the γ codes is

$$l_i = 1 + 2\log(i)$$

The implied probabilities are therefore

$$P[i] = 2^{-l(i)} = 2^{1+2\log(i)} = \frac{1}{2i^2}$$

This gives a reasonable model of the deltas in posting lists and for the TREC database (discussed in the next class) gives a factor of 3 compression compared with binary codes (see Table 14). By adjusting the code based on the length of the posting list (i.e. using a local method), the compression can be improved slightly. An example is the Bernoulli distribution (see Table 14).

In dynamic methods the probabilities are based on statistics from the actual deltas rather than on a fixed model such as the γ code. As with the static models, these methods can either be global or local. For a global method we simply measure the probability of every delta and code based on these probabilities. For local methods we could separately measure the probabilities for each posting list, but this would take too much space to store the probabilities (we would have to store separate probabilities for every list). A solution is to batch the posting lists into groups based on their length. In particular create one batch for each integer value of $\lfloor\log(length)\rfloor$. In this way we only have $\log(n)$ sets of probabilities, where n is the length of the longest posting list. Results comparing the methods are shown in Table 14 (from *Managing Gigabytes*, by Witten, Moffat, and Bell, Van Nostrand Reinhold, 1994).

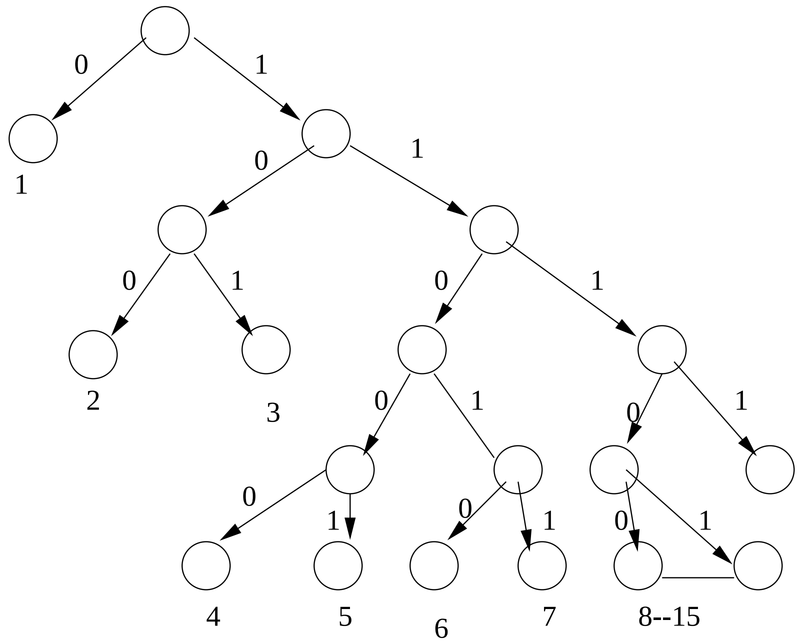


Figure 139: γ code

Method	Bits/pointer
unary	1719
binary	20
γ	6.43
Bernoulli	5.73
dynamic global	5.83
batched dynamic local	5.27

Table 14: Compression of posting files for the TREC database using various techniques in terms of average number of bits per pointer.

3.2 Representing and Accessing Lexicons

There are many ways to store the lexicon. Here we list some of them

- Sorted – just store the terms one after the other in a sorted array
- Tries – store terms as a trie data structure
- B-trees – well suited for disk storage
- Perfect hashing – assuming lexicon is fixed, a perfect hash can be calculated
- Front-coding – stores terms sorted but does not repeat front part of terms (see Table 15). Requires much less space than a simple sorted array.

Term	Complete front coding	Partial 3-in-4 front coding
7, jezebel	3, 4, ebel	0, 7, jezebel
5, jezer	4, l, r	4, l, r
7, jezerit	5, 2, it	5, 2, it
6, jeziah	3, 3, iah	3, 3, iah
6, jeziel	4, 2, el	0, 6, jeziel
7, jezliah	3, 4, liah	3, 4, liah
6, jezoar	3, 3, oar	3, 3, oar
9, jezrahiah	3, 6, rahiah	3, 6, rahiah
7, jezreel	4, 3, eel	0, 7, jezreel
11,jezreelites	7, 4, ites	7, 4, ites
6, jibsam	1, 5, ibsam	1, 5, ibsam
7, jidlaph	2, 5, dlaph	2, 5, dlaph

Table 15: Front coding (the term before jezebel was jezaniah). The pair of numbers represent where a change starts and the number of new letters. In the 3-in-4 coding every 4th term is coded completely making it easier to search for terms.

Number	Term
1	abhor
2	bear
3	laaber
4	labor
5	laborator
6	labour
7	lavacaber
8	slab

Table 16: Basic terms

When choosing among the methods one needs to consider both the space taken by the data structure and the access time. Another consideration is whether the structure allows for easy prefix queries (*e.g.*, all terms that start with `wux`). Of the above methods all except for perfect hashing allow for easy prefix searching since terms with the same prefix will appear adjacently in the structure.

Wildcard queries (*e.g.*, `w*x`) can be handled in two ways. One way is to use n-grams, by which fragments of the terms are indexed (adding a level of indirection) as shown in Table 17 for the terms listed in Table 16. Another way is to use a rotated lexicon as shown in Table 18.

Digram	Term numbers
$a	1
$b	2
$l	3,4,5,6,7
$s	8
aa	3
ab	1,3,45,6,7,8
bo	4,5,6
la	3,4,5,6,7,8
or	1,4,5
ou	6
ra	5
ry	5
r$	1,2,3, 4,5,6,7
sl	8

Table 17: N-Gram

Rotated Form	Address
$abhor	(1,0)
$bear	(2,0)
$laaber	(3,0)
$labor	(4,0)
$laborator	(5,0)
$labour	(6,0)
$lavacaber	(7,0)
$slab	(8,0)
aaber$l	(3,2)
abhor$	(1,1)
aber$la	(3,3)
abor$l	(4,2)
aborator$l	(5,2)
abour$l	(6,2)
aber$lavac	(7,6)
ab$sl	(8,3)
r$abho	(1,5)
r$bea	(2,4)
r$laabe	(3,6)
r$labo	(4,5)
r$laborato	(5,9)
r$labour	(6,6)
r$lavacabe	(7,9)
slab$	(8,1)

Table 18: Rotated lexicon

Notes by Ben Horowitz

- Evaluating information retrieval effectiveness
- Signature files
  - Generating signatures
  - Accessing signatures
  - Query logic on signature files
  - Choosing signature width
  - An example: TREC
- Vector space models
  - Selecting weights
  - Similarity measures
  - A simple example
  - Implementation
  - Relevance feedback
  - Clustering
- Latent semantic indexing (LSI)
  - Singular value decomposition (SVD)
  - Using SVD for LSI
  - An example of LSI
  - Applications of LSI
  - Performance of LSI on TREC

1 Evaluating information retrieval effectiveness

Two measures are commonly used to evaluate the effectiveness of information retrieval methods: *precision* and *recall*. The *precision* of a retrieval method is the fraction of the documents retrieved that are relevant to the query:

$$\text{precision} = \frac{\text{number retrieved that are relevant}}{\text{total number retrieved}}$$

The *recall* of a retrieval method is the fraction of relevant documents that were retrieved:

$$\text{recall} = \frac{\text{number relevant that are retrieved}}{\text{total number relevant}}$$

2 Signature files

Signature files are an alternative to inverted file indexing. The main advantage of signature files is that they don't require that a lexicon be kept in memory during query processing. In fact they do not require a lexicon at all. If the vocabulary of the stored documents is rich, then the amount of space occupied by a lexicon may be a substantial fraction of the amount of space filled by the documents themselves.

Signature files are a probabilistic method for indexing documents. Each term in a document is assigned a random *signature*, which is a bit vector. These assignments are made by hashing. The *descriptor* of document is the bitwise logical OR of the signatures of its terms. As we will see, queries to signature files sometimes respond that a term is present in a document when in fact the term is absent. Such *false matches* necessitate a three-valued query logic.

There are three main issues to discuss: (1) generating signatures, (2) searching on signatures, and (3) query logic on signature files.

2.1 Generating signatures

The *width* W of each of the signatures, is the number of bits in each term's signature. Out of these bits some typically small subset of them are set to 1 and the rest are set to zero. The parameter b specifies how many are set to 1. Typically, $1{,}000 \leq W \leq 10{,}000$. (*Managing Gigabytes* seems to suggest that typically $6 \leq b \leq 20$.) The probability of false matches may be kept arbitrarily low by making W large, at the expense of increasing the lengths of the signature files.

To generate the signature of a term, we use b hash functions as follows

for $i = 1$ to b
    $signature[hash_i(term) \ \% \ w] = 1$

In practice we just have one hash function, but use i as an additional parameter.

To generate the signature of a document we just take the logical or of the term signatures. As an example, consider the list of terms from the nursery rhyme "Pease Porridge Hot" and corresponding signatures:

Term	Signature
cold	1000 0000 0010 0100
days	0010 0100 0000 1000
hot	0000 1010 0000 0000
in	0000 1001 0010 0000
it	0000 1000 1000 0010
like	0100 0010 0000 0001
nine	0010 1000 0000 0100
old	1000 1000 0100 0000
pease	0000 0101 0000 0001
porridge	0100 0100 0010 0000
pot	0000 0010 0110 0000
some	0100 0100 0000 0001
the	1010 1000 0000 0000

Note that a term may get hashed to the same location by two hash functions. In our example, the signature for *hot* has only two bits set as a result of such a collision. If the documents are the lines of the rhyme, then the document descriptors will be:

Document	Text	Descriptor
1	Pease porridge hot, pease porridge cold,	1100 1111 0010 0101
2	Pease porridge in the pot,	1110 1111 0110 0001
3	Nine days old.	1010 1100 0100 1100
4	Some like it hot, some like it cold,	1100 1110 1010 0111
5	Some like it in the pot,	1110 1111 1110 0011
6	Nine days old.	1010 1100 0100 1100

2.2 Searching on Signatures

To check whether a term T occurs in a document, check whether all the bits which are set in T's signature are also set in the document's descriptor. If not, T does not appear in the document. If so, T *probably* occurs in the document; because some combination of other term signatures might have set these bits in T's descriptor, it cannot be said with certainty whether or not T appears in the document. For example, since the next-to-last bit in the signature for *it* is set, *it* can only occur in the fourth and fifth documents, and indeed it occurs in both. The term *the* can occur in documents two, three, five, and six; but in fact, it occurs only in the second and fifth.

The question remains of how we efficiently check which documents match the signature of the term (include all its bits). Consider the following (naive) procedure: to find the descriptor strings for which the bits in a given signature are set, pull out all descriptors, and for each descriptor, check whether the appropriate bits are set for each descriptor. When the descriptors are long and there are many files, this approach will involve many disk accesses, and will therefore be too expensive.

A substantially more efficient method is to access columns of the signature file, rather than rows. This technique is called *bit-slicing*. The signature file is stored on disk in transposed form. That is, the signature file is stored in column-major order; in the naive approach, the signature file was stored in row-major order. To process a query for a single term, only b columns need to be read from disk; in the naive approach, the entire signature file needed to be read. The bitwise AND of these b columns yields the documents in which the term probably occurs. For example, to check for the presence of *porridge*, take the AND of columns two, six, and eleven, to obtain $[110110]^T$. (This query returned two false matches.)

2.3 Query logic on signature files

We have seen that collisions between term signatures can cause a word to seem present in a document when in fact the word is absent; on the other hand, words that seem absent are absent. If, for example, *pease* is absent from a document D, then the answer to "Is *pease* is D?" is No; if *pease* seems to be present, then the answer is Maybe.

For queries with negated terms (for example, "Is *pease* absent from document d?") the situation is reversed. For example, since *pease* has bits six, eight, and 16 set, it cannot occur in documents three, four, or six, so for these documents the appropriate answer is Yes. However, the other documents might contain *pease*, so for these documents the appropriate answer is Maybe.

More generally, queries in signature files need to be evaluated in three-valued logic. Atomic queries (e.g. "Is *nine* present?") have two responses, No and Maybe. More complex queries — queries built from atomic queries using NOT, AND, and OR — get evaluated using the following rules:

NOT	
N	Y
M	M
Y	N

AND	N	M	Y
N	N	N	N
M	N	M	M
Y	N	M	Y

OR	N	M	Y
N	N	M	Y
M	M	M	Y
Y	Y	Y	Y

Consider, for example, the evaluation of "(some OR NOT hot) AND pease":

Doc.	s	h	p	NOT h	s OR NOT h	(s OR NOT h) AND p
1	M	M	M	M	M	M
2	M	M	M	M	M	M
3	N	N	N	Y	Y	N
4	M	M	N	M	M	N
5	M	M	M	M	M	M
6	N	N	N	Y	Y	N

2.4 Choosing signature width

How should W be chosen so that the expected number of false matches is less than or equal to some number z? It turns out that W should be set to

$$\frac{1}{1-(1-p)^{1/B}} \tag{9}$$

where

p, the probability that an arbitrary bit in a document descriptor is set, is given by $\frac{z}{N}^{1/b}$;

B, the total number of bits set in the signature of an average document, is given by $\frac{f}{N} \cdot \frac{b}{q}$;

N is the number of documents;

f is the number of (*document, term*) pairs;

b is the number of bits per query; and

q is the number of terms in each query.

The analysis used to derive (9) can be found in Witten, Moffat, and Bell, *Managing Gigabytes*, chapter 3, Section 3.5. One assumption made in this analysis is that all documents contain roughly the same number of distinct terms. Clearly, this assumption does not always hold.

2.5 An example: TREC

As an example, consider the TREC (Text REtrieval Conference) database. The TREC database, which is used to benchmark information retrieval experiments, is composed of roughly 1,000,000 documents — more than 3Gbytes of ASCII text — drawn from a variety of sources: newspapers, journal abstracts, U.S. patents, and so on. (These figures were true of TREC in 1994. It has probably grown in size since then.) In this example, we assume that TREC contains a mere 750,000 documents, with 137 Million document-term pairs totaling more than 2Gbytes of text.

Consider making queries on a single term and assume we want at no more than 1 false match. We assume $b = 8$. To calculate W by equation (9) we have, $f = 137 * 10^6, N = .75 * 10^6, z = 1, q = 1$, which gives:

$$\begin{aligned} p &= \left(\frac{1}{.75 * 10^6}\right)^{1/8} = .185 \\ B &= \frac{137 * 10^6}{.75 * 10^6} * \frac{8}{1} = 1470 \\ W &= 7200 \end{aligned}$$

Thus, the total space occupied by the signature file is $(7,200/8) \cdot 750$Kbytes = 675Mbytes (about 1/3 of the space required by the documents themselves). For $b = 8$, a query on a term will read 8 slices of 750Kbits, which is 750Kbytes (about .1% of the total database).

3 Vector space models

Boolean queries are useful for detecting boolean combinations of the presence and absence of terms in documents. However, Boolean queries never yield more information than a Yes or No answer. In contrast, vector space models allow search engines to quantify the *degree* of similarity between a query and a set of documents. The uses of vector space models include:

Ranked keyword searches, in which the search engine generates a list of documents which are ranked according to their relevance to a query.

Relevance feedback, where the user specifies a query, the search engine returns a set of documents; the user then tells the search engine which documents among the set are relevant, and the search engine returns a new set of documents. This process continues until the user is satisfied.

Semantic indexing, in which search engines are able to return a set of documents whose "meaning" is similar to the meanings of terms in a user's query.

In vector space models, documents are treated as vectors in which each term is a separate dimension. Queries are also modeled as vectors, typically 0-1 vectors. Vector space models are often used in conjunction with clustering to accelerate searches; see Section (3.6) below.

3.1 Selecting weights

In vector space models, documents are modeled by vectors, with separate entries for each distinct term in the lexicon. But how are the entries in these vectors obtained? One approach would be to let the entry $w_{d,t}$, the weight of term t in document d, be 1 if t occurs in d and 0 otherwise. This approach, however, does not distinguish between a document containing one occurrence of *elephant* and a document containing fifty occurrences of *elephant*. A better approach would be to let $w_{d,t}$ be $f_{d,t}$, the number of times t occurs in document d. However, it seems that five occurrences of a word shouldn't lead to a weight that is five times as heavy, and that the first occurrence of a term should count for more than subsequent occurrences. Thus, the following rule for setting weights is often used: set $w_{d,t}$ to $\log_2(1 + f_{d,t})$. Under this rule, an order of magnitude increase in frequency leads to a constant increase in weight.

These heuristics for setting $f_{d,t}$ all fail to take account of the "information content" of a term. If *supernova* appears less frequently than *star*, then intuitively *supernova* conveys more information. Borrowing from information theory, we say that the weight w_t of a term in a set of documents is $\log_2(N/f_t)$, where N is the number of documents and f_t is the number of documents in which the term appears.

One way to represent the weight of a term t in document d is by combining these ideas:

$$w_{d,t} = \log_2(N/f_t)\log_2(1 + f_{d,t})$$

It should be emphasized that this rule for setting $w_{d,t}$ is one among many proposed heuristics.

3.2 Similarity measures

To score a document according to its relevance to a query, we need a way to measure the similarity between a query vector v_q and a document vector v_d. One similarity measure is inverse Euclidean distance, $1/\|v_q - v_d\|$. This measure discriminates against longer documents, since v_d which are distant from the origin are likely to be farther away from typical v_q. Another is the dot product $v_q \cdot v_d$. This second measure unfairly favors longer documents. For example, if $v_{d_1} = 2v_{d_2}$, then $v_q \cdot v_{d_1} = 2v_q \cdot v_{d_2}$. One solution to the problems with both these measures is to normalize the lengths of the vectors in the dot product, thereby obtaining the following similarity measure:

$$\frac{w_q \cdot v_d}{\|v_q\|\|v_d\|}$$

This similarity measure is called the *cosine measure*, since if θ is the angle between vectors $\vec{x}$ and $\vec{y}$, then $\cos\theta = \vec{x}\cdot\vec{y}/(\|\vec{x}\|\|\vec{y}\|)$.

3.3 A simple example

Suppose we have the set of documents listed in the following table. These documents give rise to the frequency matrix $[f_{d,t}]$, frequencies f_t of terms, and informational contents $\log_2(N/f_t)$ in the table:

Doc. no.	Document	Frequency matrix $[f_{d,t}]$				
		a	b	c	d	e
1	apple balloon balloon elephant apple apple	3	2	0	0	1
2	chocolate balloon balloon chocolate apple chocolate duck	1	2	3	1	0
3	balloon balloon balloon balloon elephant balloon	0	5	0	0	1
4	chocolate balloon elephant	0	1	1	0	1
5	balloon apple chocolate balloon	1	2	1	0	0
6	elephant elephant elephant chocolate elephant	0	0	1	0	4
f_t		3	5	4	1	4
$\log_2(N/f_t)$		1.00	0.26	0.58	2.58	0.58

For simplicity, suppose $w_{d,t}$ is calculated according to the rule $w_{d,t} = f_{d,t}\cdot\log_2(N/f_t)$. Then the weight matrix $[w_{d,t}]$ would be as follows (the norms of each row of the weight matrix are underneath $\|v_d\|$):

	a	b	c	d	e	$\|v_d\|$
1	3	.52	0	0	.58	3.10
2	1	.52	1.74	2.58	0	3.31
3	0	1.3	0	0	.58	1.42
4	0	.26	.58	0	.58	0.86
5	1	.52	.58	0	0	1.27
6	0	0	.58	0	2.32	2.39

Then, for the queries listed below, the cosine measure gives the following similarities:

Doc. no.	Query				
	d $\|v_q\| = 2.58$	c $\|v_q\| = 0.58$	c, d $\|v_q\| = 2.64$	a, b, e $\|v_q\| = 1.18$	a, b, c, d, e $\|v_q\| = 2.90$
1	0.00	0.00	0.00	0.95	0.39
2	0.78	0.53	0.88	0.29	0.92
3	0.00	0.00	0.00	0.40	0.16
4	0.00	0.67	0.15	0.40	0.30
5	0.00	0.46	0.10	0.76	0.40
6	0.00	0.24	0.05	0.48	0.24

Note that in the second query, the fourth document beats the second because the fourth is shorter overall. For each of the other queries, there is a single document with a very high similarity to the query.

3.4 Implementation of cosine measures using inverted lists

Directly computing the dot product of a query vector and all document vector s is too expensive, given the fact that both vectors are likely to be sparse. A more economical alternative is to keep (document, weight) pairs in the posting lists, so that a typical inverted file entry would look like

$$\langle t; [(d_{t,1}, w_{d_{t,1},t}), (d_{t,2}, w_{d_{t,2},t}), \ldots, (d_{t,m}, w_{d_{t,m},t})]\rangle.$$

Here, t is a term, $d_{t,i}$ is a pointer to the ith document containing term t, and $w_{d_{t,i},t}$ is weight of t in document $d_{t,i}$. (Heuristics for calculating $w_{d,t}$ were given in Section 3.1.) For example, the inverted file entry for *apple* might be

$$\langle \text{apple}; [(3,3), (6,1), (29,6)]\rangle.$$

Fortunately, the weights $w_{d_{t,i},t}$ can typically be compressed at least as well as the distances $d_{t,i+1} - d_{t,i}$ in the inverted file entry.

These posting lists help quicken the search for the documents which are most similar to a query vector. In the following algorithm Q is a list of query terms, which we assume are unweighted; $A = \{a_d \mid a_d \text{ is the score so far for document } d\}$ is a set of accumulators; and w_d is the weight of document d, which we assume has been precomputed. This algorithm returns the k documents which are most relevant to the query.

```
Search(Q)
For each term t ∈ Q
    ⟨t; P_t⟩ = Search lexicon for t
    P_t = Uncompress(P_t)
    For each (d, w_{d,t}) in P_t
        If a_d ∈ A
            a_d = a_d + w_{d,t}
        Else
            a_d = w_{d,t}
            A = A ∪ {a_d}
For each a_d ∈ A
    a_d = a_d / W_d
Return the k documents with the highest a_d.
```

In this algorithm, the inverted file entries for every term $t \in Q$ are processed in full. Each document d that appears in some such inverted file entry adds a cosine contribution to the accumulator a_d. At the end, the accumulator values are normalized by the weights W_d. Note that there is no need to normalize by the weight w_q of the query, as w_q is constant for any particular query.

The size of A can grow very large, so some search engines place an *a priori* bound on the number of accumulators a_d in A. One might think this approach would result in poor retrieval effectiveness, since the most relevant documents may be found by the last terms in the query. However, experiments with TREC and a standard collection of queries have shown that 5,000 accumulators suffice to extract the top 1,000 documents.

3.5 Relevance feedback

In the query systems discussed so far, the user poses a query v_{q_0}, the search engine answers it, and the process ends. But suppose the user has the ability to mark a set of documents R_0 as relevant, and a set I_0 as irrelevant. The search engine then modifies v_{q_0} to obtain v_{q_1} and fetches a set of documents relevant to v_{q_1}. This is called *relevance feedback*, and continues until the user is satisfied. It requires the initial query to be adapted, emphasizing some terms, de-emphasizing others, and perhaps introducing entirely new terms. One proposed strategy for updating the query is the *Dec Hi* strategy:

$$v_{q_{i+1}} = v_{q_i} + \left(\sum_{d \in R_i} v_d \right) - v_n.$$

To obtain the $(i+1)$th query, the vectors for the most relevant documents R_i (chosen by the user) are added to the ith query vector, and the vector v_n of the least relevant document is subtracted. A more general update rule is

$$v_{q_{i+1}} = \pi v_{q_0} + \omega v_{q_i} + \alpha \sum_{d \in R_i} v_d + \beta \sum_{d \in I_i} v_d,$$

where π, ω, α, and β are weighting constants, with $\beta \leq 0$.

One potential problem with these update rules is that the query vectors vq_i for $i \geq 1$ can have far more terms than the initial query v_{q_0}; thus, these subsequent queries may be too expensive to evaluate. One solution is to sort the terms in the relevant documents by decreasing weight, and select a subset of them to influence $v_{q_{i+1}}$. Another solution is to use clustering, which is described in the next section.

Experiments indicate that one round of relevance feedback improves responses from search engines, and two rounds yield a small additional improvement.

3.6 Clustering

Clustering may be used to speed up searches for complicated queries, and to find documents which are similar to each other. The idea is to represent a group of documents which are all close to each other by a single vector, for example the centroid of the group. Then, instead of calculating the relevance of each document to a query, we calculate the relevance of each *cluster* vector to a query. Then, once the most relevant cluster is found, the documents inside this cluster may be ranked against the query. This approach may be extended into a hierarchy of clusters; see figure 3.6.

There are many techniques for clustering, as well as many other applications. Clustering will be discussed in the November 21 lecture.

4 Latent semantic indexing (LSI)

All of the methods we have seen so far to search a collection of documents have matched words in users' queries to words in documents. These approaches all have two drawbacks. First,

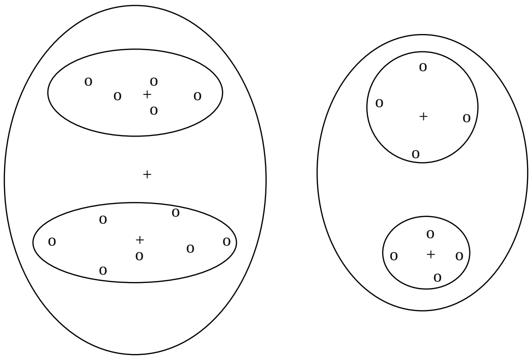


Figure 140: A hierarchical clustering approach

since there are usually many ways to express a given concept, there may be no document that matches the terms in a query even if there is a document that matches the meaning of the query. Second, since a given word may mean many things, a term in a query may retrieve irrelevant documents. In contrast, latent semantic indexing allows users to retrieve information on the basis of the conceptual content or meaning of a document. For example, the query *automobile* will pick up documents that do not contain *automobile*, but that do contain *car* or perhaps *driver*.

4.1 Singular value decomposition (SVD)

4.1.1 Definition

LSI makes heavy use of the singular value decomposition. Given an $m \times n$ matrix A with $m \geq n$, the singular value decomposition of A (in symbols, SVD(A)), is $A = U\Sigma V^T$, where:

1. $U^T U = V^T V = I_n$, the $n \times n$ identity matrix.

2. $\Sigma = \mathrm{diag}(\sigma_1, \ldots, \sigma_n)$, the matrix with all 0's except for the σ_i's along the diagonal.

If Σ is arranged so that $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_n$, then the SVD of A is unique (except for the possibility of equal σ). Further, if r denotes rank(A), then $\sigma_i > 0$ for $1 \leq i \leq r$, and $\sigma_j = 0$ for $j \geq r+1$. Recall that the rank of a matrix is the number of independent rows (or columns).

We will use U_k to denote the first k columns of U, V_k the first k columns of V, and Σ_k the first k columns and rows of Σ. For latent semantic indexing, the key property of the SVD is given by the following theorem:

Theorem: Let $A = U\Sigma V^T$ be the singular value decomposition of the $m \times n$ matrix A. Define

$$A_k = U_k \Sigma_k V_k^T.$$

Define the distance between two $m \times n$ matrices A and B to be

$$\sum_{i=1}^{m}\sum_{j=1}^{n}(a_{ij}-b_{ij})^2.$$

Then A_k is the matrix of rank k with minimum distance to A.

This basically says that the best approximation of A (for the given metric) using a mapping of a k dimensional space back to the full dimension of A is based on taking the first k columns of U and V.

In these scribe notes, we will refer to $A_k = U_k \Sigma_k V_k^T$ as the *truncated SVD* of A. In the singular value decomposition $U\Sigma V^T$, the columns of U are the orthonormal eigenvectors of AA^T and are called the *left singular vectors*. The columns of V are the orthonormal eigenvectors of $A^T A$ and are called the *right singular vectors*. The diagonal values of Σ are the nonnegative square roots of the eigenvalues of AA^T, and are called the *singular values* of A.

The calculation of the SVD uses similar techniques as for calculating eigenvectors. For dense matrices, calculating the SVD takes $O(n^3)$ time. For sparse matrices, the Lanczos algorithm is more efficient, especially when only the first k columns of U and V are required.

4.2 Using SVD for LSI

To use SVD for latent semantic indexing, first construct a term-by-document matrix A. Here, $A_{t,d} = 1$ if term t appears in document d. Since most words do not appear in most documents, the term-by-document matrix is usually quite sparse. Next, computing SVD(A), generate U, Σ, and V. Finally, retain only the first k terms of U, Σ, and V. From these we could reconstruct A_k, which is an approximation of A, but we actually plan to use the U, Σ, and V directly.

The truncated SVD A_k describes the documents in k-dimensional space rather than the n dimensional space of A (think of this space as a k dimensional hyperplane through the space defined by the original matrix A). Intuitively, since k is much smaller than the number of terms, A_k "ignores" minor differences in terminology. Since A_k is the closest matrix of rank (dimension) k to A, terms which occur in similar documents will be near each other in the k-dimensional space. In particular, documents which are similar in meaning to a user's query will be near the query in k-dimensional space, even though these documents may share no terms with the query.

Let q be the vector representation of a query. This vector gets transformed into a vector $\hat{q}$ in k-dimensional space, according to the equation

$$\hat{q} = q^T U_k \Sigma_k^{-1}.$$

The projected query vector $\hat{q}$ gets compared to the document vectors, and documents get ranked by their proximity to the $\hat{q}$. LSI search engines typically use the cosine measure as a measure of nearness, and return all documents whose cosine with the query document exceeds some threshold. (For details on the cosine measure, see Section 3.2.)

4.3 An example of LSI

Suppose we want to apply LSI to the small database of book titles given in Figure 141 (a). Note that these titles fall into two basic categories, one having to do with the mathematics of differential equations (e.g. B8 and B10) and one having to do with algorithms (e.g. B5 and B6). We hope that the LSI will separate these classes. Figure 141 (b) shows the term-by-document matrix A for the titles and for a subset of the terms (the non stop-words that appear more than once).

Now for $k = 2$ we can generate U_2, Σ_2 and V_2 using an SVD. Now suppose that we are interested in all documents that pertain to *application* and *theory*. The coordinates for the query *application theory* are computed by the rule $\hat{q} = q^T U_2 \Sigma_2^{-1}$, as follows:

$$\hat{q} = (0.0511 \quad -0.3337) = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}^T \begin{pmatrix} 0.0159 & -0.4317 \\ 0.0266 & -0.3756 \\ 0.1785 & -0.1692 \\ 0.6104 & 0.1187 \\ 0.6691 & 0.1209 \\ 0.0148 & -0.3603 \\ 0.0520 & -0.2248 \\ 0.0066 & -0.1120 \\ 0.1503 & 0.1127 \\ 0.0813 & 0.0672 \\ 0.1503 & 0.1127 \\ 0.1785 & -0.1692 \\ 0.1415 & 0.0974 \\ 0.0105 & -0.2363 \\ 0.0952 & 0.0399 \\ 0.2051 & 0.5488 \end{pmatrix} \begin{pmatrix} 4.5314 & 0 \\ 0 & 2.7582 \end{pmatrix}^{-1}$$

All documents whose cosine with $\hat{q}$ exceeds 0.9 are illustrated in the shaded region of Figure 142. Note that B5, whose subject matter is very close to the query, yet whose title contains neither *theory* nor *application*, is very close to the query vector. Also note that the mapping into 2 dimensions clearly separates the two classes of articles, as we had hoped.

4.4 Applications of LSI

As we have seen, LSI is useful for traditional information retrieval applications, such as indexing and searching. LSI has many other applications, some of which are surprising.

Cross-language retrieval. This application allows queries in two languages to be performed on collections of documents in these languages. The queries may be posed in either language, and the user specifies the language of the documents that the search engine should return.

For this application, the term-by-document matrix contains documents which are composed of text in one language appended to the translation of this text in another language. For example, in one experiment, the documents were a set of abstracts that had versions

Label	Titles
B1	A Course on Integral Equations
B2	Attractors for Semigroups and Evolution Equations
B3	Automatic Differentiation of Algorithms: Theory, Implementation, and Application
B4	Geometrical Aspects of Partial Differential Equations
B5	Ideals, Varieties, and Algorithms – An Introduction to Computational Algebraic Geometry and Commutative Algebra
B6	Introduction to Hamiltonian Dynamical Systems and the N-Body Problem
B7	Knapsack Problems: Algorithms and Computer Implementations
B8	Methods of Solving Singular Systems of Ordinary Differential Equations
B9	Nonlinear Systems
B10	Ordinary Differential Equations
B11	Oscillation Theory for Neutral Differential Equations with Delay
B12	Oscillation Theory of Delay Differential Equations
B13	Pseudodifferential Operators and Nonlinear Partial Differential Equations
B14	Sinc Methods for Quadrature and Differential Equations
B15	Stability of Stochastic Differential Equations with Respect to Semi-Martingales
B16	The Boundary Integral Approach to Static and Dynamic Contact Problems
B17	The Double Mellin-Barnes Type Integrals and Their Applications to Convolution Theory

(a)

Terms	Documents																
	B1	B2	B3	B4	B5	B6	B7	B8	B9	B10	B11	B12	B13	B14	B15	B16	B17
algorithms	0	0	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0
application	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1
delay	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0
differential	0	0	0	1	0	0	0	1	0	1	1	1	1	1	1	0	0
equations	1	1	0	1	0	0	0	1	0	1	1	1	1	1	1	0	0
implementation	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0
integral	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
introduction	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0
methods	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0
nonlinear	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0
ordinary	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0
oscillation	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0
partial	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0
problem	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	1	0
systems	0	0	0	0	0	1	0	1	1	0	0	0	0	0	0	0	0
theory	0	0	1	0	0	0	0	0	0	0	1	1	0	0	0	0	1

(b)

Figure 141: An example set of document titles and the corresponding matrix A. Taken from "Using Linear Algebra for Intelligent Information Retrieval" by Berry, Dumais and O'Brien, CS-94-270, University of Tennessee.

12 Berry, Dumais and O'Brien

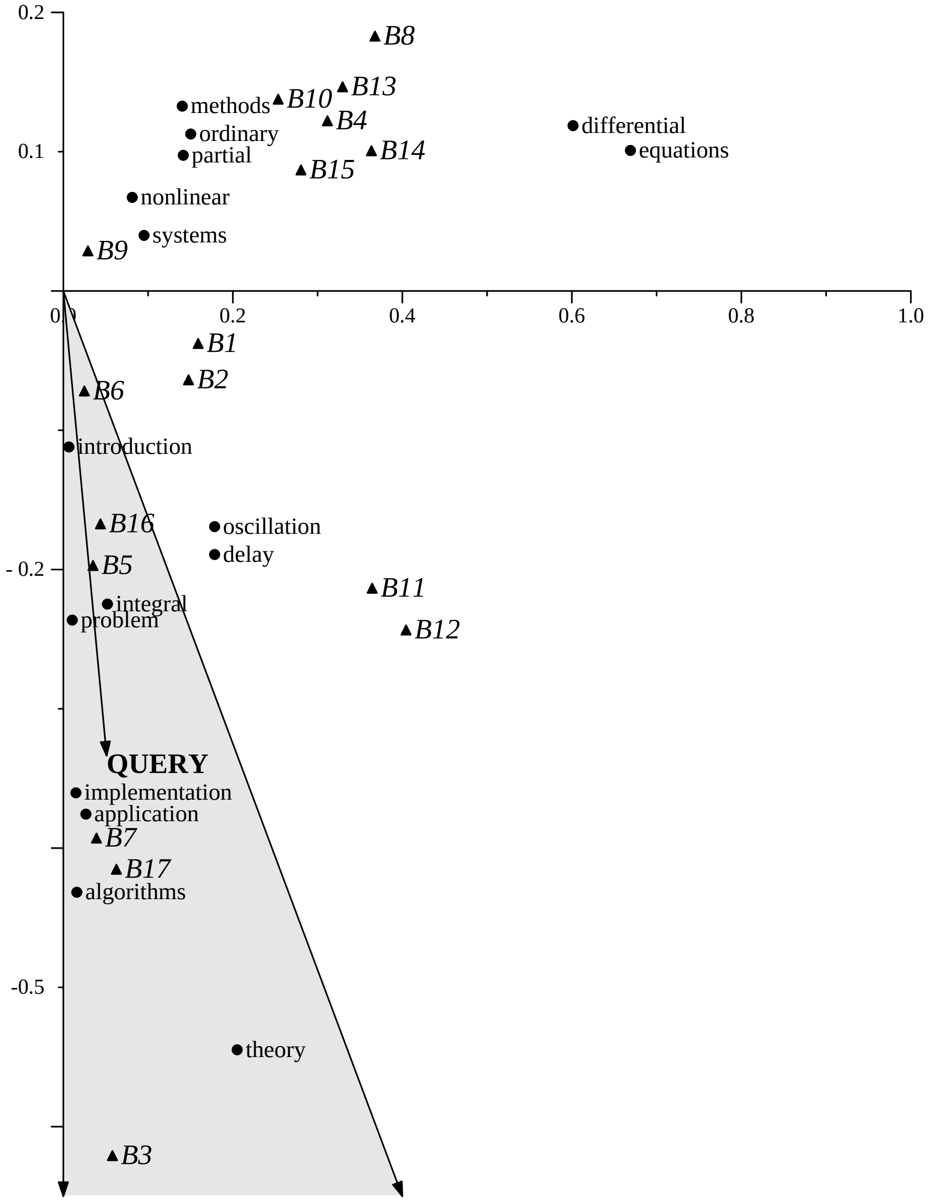


FIG. 6. *A Two-dimensional plot of terms and documents along with the query* **application theory**.

Figure 142: The 2-dimensional positioning of the documents and terms for a set of 17 articles.

in both French and English. The truncated SVD of the term-document matrix is then computed. After the SVD, monolingual documents can be "folded in" — a monolingual document will get represented as the vector sum of its constituent terms, which are already represented in the LSI space.

Experiments showed that the retrieval of French documents in response to English queries was as effective as first translating the queries into French, then performing a search on a French-only database. The cross-language retrieval method was nearly as effective for retrieving abstracts from an English-Kanji database, and for performing searches on English and Greek translations of the Bible.

Modeling human memory: LSI has been used to model some of the associative relationships in human memory, in particular, the type of memory used to recall synonyms. Indeed, LSI is often described as a method for finding synonyms — words which have similar meanings, like *cow* and *bovine*, are close to each other in LSI space, even if these words never occur in the same document. In one experiment, researchers calculated the truncated SVD of a term-by-document matrix for an encyclopedia. They then tested the truncated SVD on a synonym test, which had questions like

levied:
- (A) imposed
- (B) believed
- (C) requested
- (D) correlated

For a multiple-choice synonym test, they then computed the similarity of the first word (e.g. *levied*) to each choice, and picked the closest alternative as the synonym. The truncated SVD scored as well as the average student.

Matching people: LSI has been used to automate the assignment of reviewers to submitted conference papers. Reviewers were described by articles they had written. Submitted papers were represented by their abstracts, and then matched to the closest reviewers. In other words, reviewers were first placed in LSI space, and then people who submitted papers were matched to their closest reviewers. Analysis suggested that these automated assignments, which took less than one hour, were as good as those of human experts.

4.5 Performance of LSI on TREC

Recall that the TREC collection contains more then 1,000,000 documents, more than 3Gbytes of ASCII text. TREC also contains 200 standard benchmarking queries. A panel of human judges rates the effectiveness of a search engine by hand-scoring the documents returned by the search engine when posed with these queries. These 200 queries are quite detailed, averaging more than 50 words in length.

Because TREC queries are quite rich, a smaller advantage can be expected for any approach that involves enhancing user's queries, as LSI does. Nonetheless, when compared with the best keyword searches, LSI performed fairly well. For information retrieval tasks, LSI performed 16% better. For information filtering tasks, LSI performed 31% better. (In information filtering applications, a user has a stable profile, and new documents are matched against these long-standing interests.)

The cost of computing the truncated SVD on the full TREC collection was prohibitively expensive. Instead, a random sample of 70,000 documents and 90,000 terms was used. The resulting term-by-document matrix was quite sparse, containing only less than .002% non-zero entries. The Lanczos algorithm was used to find A_{200}; this computation required 18 hours of CPU time on a SUN SPARCstation 10.

Outline

Today we have two student lectures.

- Evolutionary Trees (*Andris Ambainis*)
  1. Parsimony
  2. Maximum-Likelihood
  3. Distance Methods
- Finding Authoritative Web Pages Using the Web's Link Structure (*David Wagner*)

1 Evolutionary Trees (Andris Ambainis)

An evolutionary tree, or *phylogeny*, is a tree which describes the ancestral relationships among a set of species. The leaves of an evolutionary tree correspond to the species being compared, and internal nodes correspond to (possibly extinct) ancestral species. Figure 143 shows a simple evolutionary tree for cats, dogs and lynxes. Domestic cats and lynxes share a feline ancestor distinct from dogs, but all three species share a common ancestor as well.

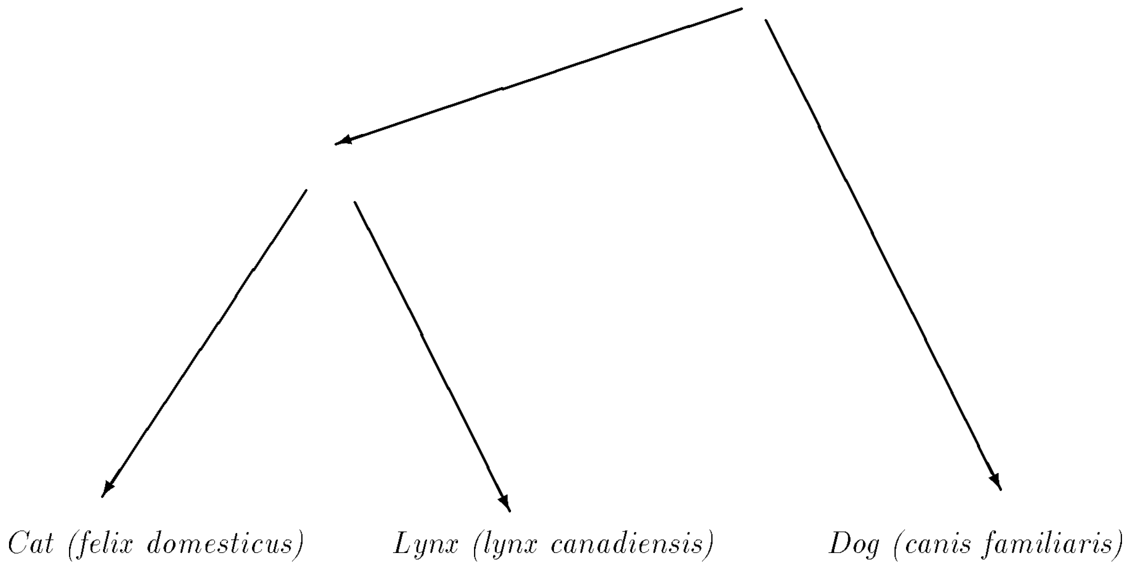


Figure 143: An evolutionary tree for cats, dogs and lynxes

Scientists have been studying and creating phylogenies for over a century. Traditionally, morphological information and taxonomies were used as data for creating evolutionary trees. This process was often guided more by intuition than a rigorous set of rules. Today, more systematic methods based on DNA sequencing are used. DNA sequencing provides a much larger body data which can be precisely analyzed. Such data both demands and facilitates computer algorithms for generating evolutionary trees.

Three algorithms for creating evolutionary trees will be discussed in the following subsections. In each case, we assume that the DNA sequences have already been aligned using multiple alignment, and that each position in the sequence is independent.

1.1 Parsimony

In the parsimony method, the best tree is the one which explains the data using the smallest number of evolutionary changes:

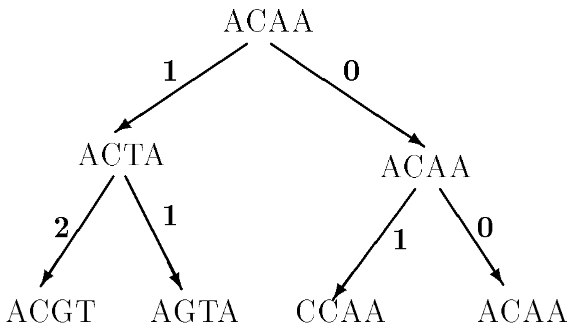


Figure 144: A parsimony model for sequences ACGT, AGTA, CCAA and ACAA. The numbers on the edges indicate the number of mutations between nodes

Parsimony can be formulated as a Steiner Tree problem. For example, given a set of m aligned DNA sequence of length n, consider the n dimensional hypercube $G = \{A, G, C, T\}^n$. The weights on the edges of this graph are the distance metric between the two endpoints. Each DNA sequence is a vertex on this graph. The problem of parsimony is to find the minimum steiner tree with these vertices as demand points.

Analysis

If we are given the tree structure with the leaves assigned, we can find a minimum-cost assignment for the internal nodes using dynamic programming. However, to find the tree that minimizes the number of changes is an *NP*-hard problem.

We can approximate the solution, however, using local search heuristics. First, we find a "good" tree. One way to do this is using a greedy method: starting with the set of species, keep connecting the two subtrees with the most closely-related roots until there is only one tree. We then apply rearrangement operations to the tree and check if the rearranged tree has a lower cost. Possible rearrangement operations include:

- Swap two subtrees that are children of neighboring nodes.

- Remove a subtree and attach it to some other node.

This process can be repeated a fixed number of times, or until we find a local minimum.

Discussion

Parsimony assumes that mutations are rare. In fact, this may not the case. Indeed, it is possible using parsimony to construct evolutionary trees that are "positively misleading" (*i.e.*, the probability of getting the wrong tree approaches one as the amount of data approaches infinity) [J78].

Despite being NP-hard and positively misleading, parsimony remains popular among evolutionary biologists. It has the advantage of being simple, intuitive and appealing. A recent study of Rice and Warnow shows that on artificial data, parsimony performs quite well [RW97].

1.2 Maximum-Likelihood

The maximum-likelihood method generates an evolutionary tree from a stochastic model of evolution. The same model can be used to generate trees for different sets of species.

To simplify this presentation, we first assume that DNA sequences are binary (*i.e.*, sequences of 0's and 1's).

We will describe stochastic models called *Cavender-Farris trees*. Such a tree has a probability associated with each edge as well as an initial state probability associated with the root. Given a tree with root probability p_R and probabilities p_e for each edge e, each position of the binary sequence is generated as follows:

1. The root is labeled 1 with probability p_R and 0 with probability $1 - p_r$.
2. Each labeled node broadcasts its label to all its children.
3. When a label is broadcast over edge e, it gets reversed with probability p_e.

Figure 145 shows an example of a Cavender-Farris tree. We generate a sequence for a node X (*i.e.*, species X) by repeatedly running the stochastic process and adding the label of X to the sequence, as illustrated in figure 146. This method relies on the assumption that all positions in a DNA sequence are independent.

When using four states (*i.e.*, A,G,C,T), each edge is assigned a probability matrix instead of a scalar probability p_e:

$$\begin{bmatrix} p_{\mathrm{AA}} & p_{\mathrm{AC}} & p_{\mathrm{AG}} & p_{\mathrm{AT}} \\ p_{\mathrm{CA}} & p_{\mathrm{CC}} & p_{\mathrm{CG}} & p_{\mathrm{CT}} \\ p_{\mathrm{GA}} & p_{\mathrm{GC}} & p_{\mathrm{GG}} & p_{\mathrm{GT}} \\ p_{\mathrm{TA}} & p_{\mathrm{TC}} & p_{\mathrm{TG}} & p_{\mathrm{TT}} \end{bmatrix}$$

where p_{XY} is the probability that X changes to Y. Biologists often assume that some or all p_{XY} are related, or even equal.

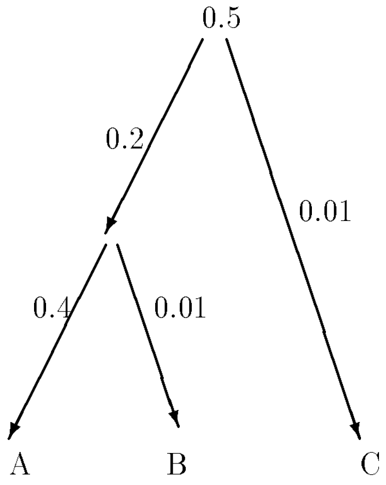


Figure 145: A stochastic model as a Cavender-Farris tree

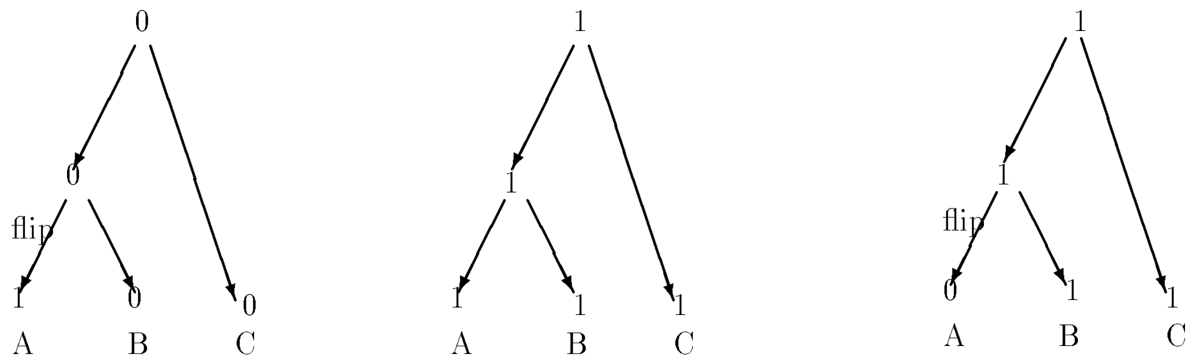


Figure 146: Generating sequences from the stochastic model in figure 145. Edges over which the label has been flipped are marked with “flip.”

Given a stochastic model of evolution, the maximum-likelihood method finds a tree T that maximizes $p_T(S)$ where S is the set of DNA sequences and $p_T(S)$ is the probability that T will generate all the sequences in S.

This method is much more general than parsimony, which has a fixed model of evolution that may be incorrect. Assuming that the chosen stochastic model of evolution is accurate, this method will give a correct evolutionary tree for any set of species. However, it is the most computationally expensive of the methods discussed in this lecture, and is rarely used on large trees.

1.3 Distance methods

In the distance method, we create a matrix D consisting of similarity measures between pairs of species in the set being studied. Then, we find a tree that generates the distance matrix closest to D. This method is illustrated by the following example.

An example

Take three species, a, b and c with the following DNA sequences:

a: ACAAAAACAA
b: ACTTGAACAT
c: TCTAAAACAT

Now generate a similarity matrix for a, b and c:

$$\begin{bmatrix} - & 0.4 & 0.3 \\ 0.4 & - & 0.3 \\ 0.3 & 0.3 & - \end{bmatrix}$$

The similarity measure used is $d_{xy} = M/L$ where:
M is the number of positions with synonymous characters.
U is the number of positions with non-synonymous characters.
G is the number of positions with gaps in one sequence and residue in another.
W_G is the weight of gaps. (For this example $W_G = 1$)
$L = M + U + W_G G$

Transform the similarity matrix into a matrix with estimates of "evolutionary time" between species. We estimate the evolutionary time t_{xy} between two species x and y as $t_{xy} = \ln(1 - 2d_{xy})$.

$$\begin{bmatrix} - & 1.62 & 0.93 \\ 1.62 & - & 0.93 \\ 0.93 & 0.93 & - \end{bmatrix}$$

Now, find a tree with nodes a, b and c such that the distance between each of these nodes is the same as the evolutionary time between the corresponding species:
Note that the tree in figure 147 is undirected.

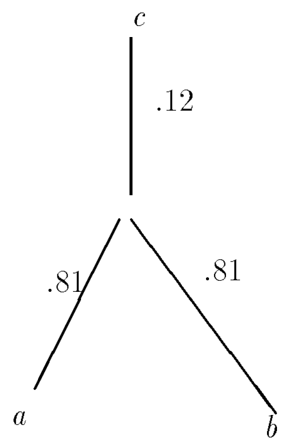


Figure 147: An evolutionary that fits the time matrix for species a, b and c.

Analysis

If the data precisely fits some tree, then this method is relatively simple. However, real data have random errors. If no tree exactly fits the data, finding the tree that is closest to a distance matrix is *NP*-complete under several definitions of "closeness." In practice, the following heuristic, called *nearest-neighbor joining* [SN87] is often used:

- Introduce closeness measure $d'(x,y) = (n-2)d(x,y) - \sum_x d(x,z) - \sum_y d(y,z)$
- Repeat until only one node remains:
  1. Search for x and y that minimize $d'(x,y)$.
  2. Replace x and y with a new node xy.
  3. Calculate $d(xy,z) = (d(x,z) + d(y,z) - d(x,y))/2$

The evolutionary tree will the tree where each pair of nodes x and y is connected its joint xy:

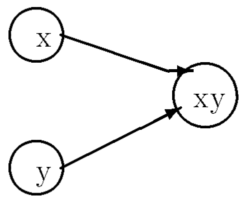


Figure 148: A branch of the evolutionary tree joining nearest neighbors x and y.

2 Finding authoritative web pages using link structure (David Wagner)

Today's web search engines do keyword searching. This technique is very powerful, but it handles broad queries quite poorly. The quantity of hits is often very high, and the quality of these hits varies widely.

We want to be able to identify authoritative sources for broad queries. This lecture presents an algorithm which:

1. Views the web as a directed graph.
2. Picks a relevant subgraph using standard search techniques.
3. Analyzes the resulting link structure.

Why analyze the link structure? Take two pages p and q such that p has a link to q. Through this link, the creator of page p has in some sense conferred authority on q. For example, the "Algorithms in the Real World" class page contains a large number of links to related pages. These linked pages can be considered authoritative sources on algorithms in the real world.

2.1 A naive approach using adjacency

The "authoritativeness" of p can be measured using the *in-degree* of p, or the number of pages which link to p. More precisely, we want the in-degree of p in a subgraph induced by the keyword search for the query. Such a subgraph can be generated by taking the top 200 pages returned by a standard search engine (like AltaVista or HotBot) as a root set, and then adding all "adjacent" pages (*i.e.*, pages with links into and out of the root set).

This approach does not work very well in practice. For example, a search on the word "java" will probably return pages on the programming language, the island in Indonesia and the popular caffeinated beverage. It is likely that the user only wanted one of these topics.

However, an analysis of the link structure in this query result will likely reveal three subgraphs, separated roughly by topic and with few interconnecting links between them. This property leads to an improved search algorithm.

2.2 An algorithm for finding communities within topics

Hubs are pages that point to multiple relevant authoritative pages. The best known example of a hub is Yahoo. An *authority* is a page that is pointed to by multiple relevant hubs. For example, the JavaSoft home page is an authority, since is linked by most hubs that catalogue Java links.

We want to find *communities*, or sets of hubs H and authorities A such that $H + A$ resembles a dense bipartite graph.

We will now define an iterative algorithm for finding communities. Page p is assigned weights $x[p]$ and $y[p]$ that represent authority weight and hub weight, respectively. The hub weight $x[p]$ is the sum of authority weights of its outgoing links. Likewise, $y[p]$ is the sum of the hub weights of pages that point to p. Figure 150 illustrates how $x[p]$ and $y[p]$ are calculated.

The algorithm can be implemented using linear algebra. Let A be the adjacency matrix of the graph induced by the keyword search. A_{pq} is 1 if p has a link to q and 0 otherwise. At

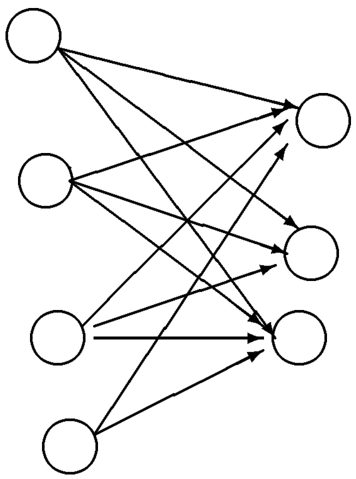


Figure 149: A subgraph representing a community. Hubs (on the left) point to authorities (on the right).

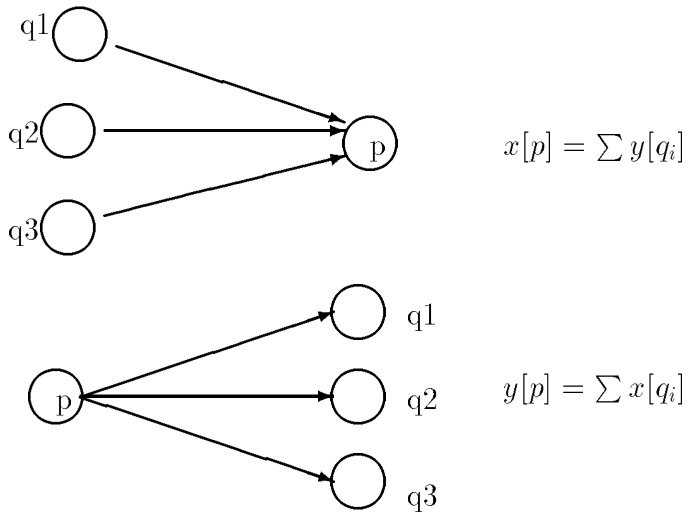


Figure 150: Calculating the hub weight and authority weight of a page p

each iteration, let $x = A^T y$ and $y = Ax$. Normalize x and y, and repeat the process until x and y converge. From linear algebra, we know that x and y will eventually converge to the principal eigenvectors of $A^T A$ and AA^T.

Each eigenvector corresponds to one community. The principal eigenvector corresponds to the most prominent community, but other eigenvectors can be used to examine other communities in the topic. It is interesting to note the similarities between the linear algebra approach to finding communities and the use of SVD in latent semantic indexing as described in the previous lecture. Recall that in finding the $SVD(A) = U\Sigma V^T$ that the columns of U are the normalized eigenvectors of AA^T and the columns of V are the normalized eigenvectors of $A^T A$. Using U_1 and V_1 gives us the principal eigenvectors (the ones with the highest eigenvalue). In fact, the iterative approach mentioned above (often called repeated squaring) is just one of the many approaches to solve for the first eigenvectors for the SVD.

Here are some examples of real queries using this algorithm:

- The principal eigenvector of the query "java" includes JavaSoft, the `comp.lang.java` FAQ, etc.
- The query "net censorship" includes the sites for EFF, CDT, VTW and ACLU, all of which are organizations that play a prominent role in free speech issues on the net.
- The query "abortion" returns pro-life pages as one eigenvector and pro-choice pages as another. Each side of the issue tends to form a bipartite graph with very few cross-links to the opposition.

This method works surprisingly well for broad queries, and shows promise for some other search tasks, such as "similar page" queries (*i.e.* search for pages with information similar to a given page). However, for very specific queries, the authoritative search does not work very well. Rather, it tends to find authorities for generalizations of the query topic. It might be possible to use lexical ranking function to keep the authoritative search oriented, but this is left open for future work.

For further reading

More details on the authoritative searching technique can be found elsewhere. [K97].

References

[J78] J. Felsenstein, "Cases in which parsimony or compatibility methods will be positively misleading." *Systematic Zoology*, 27:401-410, 1978.

[K97] J. Kleinberg. "Authoritative Sources in a Hyperlinked Environment." IBM Research Report RJ 10076, May 1997. This paper will also appear in the Proceedings of the ACM SIAM Symposium on Discrete Algorithms, 1998.

[RW97] Kenneth Rice and Tandy Warnow, "Parsimony is Hard to Beat." *COCOON97.*

[SN87] N. Saifou, M. Nei. "The neighbor-joining method: a new method for unconstructing phylogenic trees." *Mol.Biol.Evol.*, 4:406-425, 1987.

Notes by Josh MacDonald

- Principle Component Analysis
  1. LSI
  2. Authoritative Sources
- Introduction to Clustering
  1. Applications
  2. Distance and similarity measures
- Clustering Algorithms
  1. Bottom-up Hierarchical
  2. Top-down Hierarchical
  3. Optimization
  4. Density Search

1 Principle Component Analysis

The study of *principle component analysis* (PCA) goes back at least as far as 1901 and has a number of closely related fields:

- factor analysis
- latent semantic indexing
- latent vectors
- Hotelling
- singular value decomposition
- principle components
- SV transform

The objective of PCA is to find the eigenvectors and associated eigenvalues of a similarity or correlation matrix. For n items an $n \times n$ similarity matrix S is one in which the entry S_{ij} represents the similarity between items i and j—the matrix is typically symmetric. The eigenvector of S with the highest eigenvalue specifies the direction that best "separates" the items. In particular it is the projection of the items into a 1 dimensional space that best maintains the similarity metric (*i.e.*, similar items have close values in the eigenvector).

Here we will briefly look at how latent semantic indexing and authoritative sources are just special cases of PCA. Recall that the singular value decomposition of an $m \times n$ matrix A yields the following:

$$A = U\Sigma V^T$$

where

$$U^T U = V^T V = I$$

and Σ is a diagonal matrix with diagonal entries $\sigma_1, \sigma_2, \ldots$. If A has rank r (number of independent rows or columns) then U has dimension $m \times r$, V has dimension $n \times r$, and Σ has dimension $r \times r$. (In lecture 22 we assumed that for $m \geq n$ that r is replaced with n in the stated sizes, but when $r < n$ only the first r columns, or rows, are actually important and the rest can be dropped.) By using the above equations, we get

$$\begin{aligned} A^T A &= (U\Sigma V^T)^T (U\Sigma V^T) \\ &= (V\Sigma^T U^T)(U\Sigma V^T) \\ &= V\Sigma^2 V^T \end{aligned} \tag{10}$$

and by multiplying by V on the right we get

$$\begin{aligned} A^T A V &= V\Sigma^2 V^T V \\ &= V\Sigma^2 \end{aligned}$$

This shows that each column of V is an eigenvector of $A^T A$ with eigenvalue σ_i^2. Similarly, each column of U is an eigenvector of AA^T with eigenvalue σ_i^2. We can therefore think of the columns of V as the principal components of a similarity matrix $A^T A$ and the columns of U as the principal components of a similarity matrix AA^T. Since $A^T A$ is just all the dot products of the columns, the "similarity" measure being used by V is dot product of columns, and the "similarity" measure being used by U is dot product of rows.

Latent Semantic Indexing

We discussed latent semantic indexing as a method for improving text-retrieval with the hopes of finding content which did not directly match our query, but was semantically "close". Here the matrix elements A_{ij} represents the weight of each term i in a document j. Thus each row represent a term in the dictionary, and each column represent a document. In this framework, U represents the principal components of the dot product of rows (*i.e.*, term-term similarities), and V represents the principal components of the dot product of columns (*i.e.*, the document-document similarities). If we normalize the rows of A, for example, then the similarity measure being used to generate U is simply the cosine measure.

Authoritative Sources

The hyper-linked graph of documents represent a directed graph from which we construct an adjacency matrix A (*i.e.*, place a 1 in A_{ij} if there is a link from i to j and a 0 otherwise).

The matrix $A^T A$ now represents for each pair of documents i and j how many links they have in common, and the matrix AA^T represent for each pair of documents i and j how many common pages link to both of them. These matrices can be thought of as similarity measures between sources, and destinations, respectively. In the definitions given in the last class a "hub" is a page that points to multiple relevant authoritative pages. The principal component of $A^T A$ (also the first column of U in the SVD decomposition of A) will give the "principal hub". An "authority" is a page that is pointed to by multiple relevant hubs. The principal component of AA^T (also the first column of V in the SVD decomposition of A) will give the "principal authority".

2 Introduction to Clustering

The objective of *clustering* is, given a set of objects and a measure of similarity or distance between the objects, cluster into groups of similar (nearby) objects. Clustering is also called:

- typology
- grouping
- clumping
- classification
- unsupervised learning
- numerical taxonomy

2.1 Applications

There are many applications of clustering, here are a few:

- biology: multiple alignment, evolutionary trees
- business: marketing research, risk analysis
- liberal arts: classifying painters, writers, musicians
- computer science: compression, vector quantization, information retrieval, color reduction
- sociology and psychology: personality types, classifying criminals, classifying survey responses and experimental data.
- indexing and searching: group results into categories (Hearst and Pederson, 1996)

2.2 Similarity and Distance Measures

To cluster the items in a set we need some notion of distance and/or similarity between items. Typically when one talks about a distance measure $d(i, j)$ between items i and j in a set E one assumes that d is a *metric* for E, and therefore abides by the following conditions:

$$\begin{aligned} d(i,j) \geq & \quad 0 \\ d(i,i) = & \quad 0 \\ d(i,j) = & \quad d(j,i) \\ d(i,j) \leq & \quad d(i,k) + d(k,j) \end{aligned}$$

The fourth condition is the familiar triangle inequality. These conditions are met for common metrics such as the Euclidean distance and the Manhattan distance. Metrics may be combined, so, for example, the maximum of two metrics is itself a metric space. Many clustering algorithms that use distance measures assume all four conditions are met.

A similarity is a measure of how close two points are in a space. We are familiar with cosine measures, dot-products, and identity as measures of similarity. In general, however, similarities might be supplied by a black box or by human judgment.

One question is whether similarities and distances can be used interchangeably. Given an arbitrary distance measure we can always come up with a corresponding similarity measure:

$$s(i,j) = \frac{1}{1 + d(i,j)}$$

Given a similarity measure (ranging from 0 to 1) we might also try the corresponding distance metric

$$d(i,j) = \frac{1}{s(i,j)} - 1$$

This measure, however, does not necessarily form a metric. Therefore one should be careful when using such a transformation to generate a distance from a similarity, especially if it is going to be applied in a clustering algorithm that assumes the distance function forms a metric.

3 Clustering Algorithms

We will be considering four classes of algorithms for clustering.

- Bottom-up Hierarchical (agglomerative)
- Top-down Hierarchical (divisive)
- Optimization
- Density searches

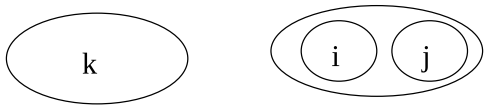


Figure 151: Lance and Williams' Rule

The various algorithms we consider not only vary in how they work but in what they are optimizing. In some applications it is important to have a fixed number of clusters, while in others it is better to find "natural" clusters which will return a number of clusters that depends on the nature of the data. In some applications, such as some index searching techniques, a hierarchy of clusters is required while in others it is not. In some applications it is best to minimize the maximum distance within each cluster, while in others it is better to minimize the average distance. There is therefore no single "correct" clustering of a set of items, but instead what constitutes a good clustering will depend on the application.

3.1 Bottom-Up Hierarchical Algorithms

The basic algorithm for bottom-up, or agglomerative, methods is to begin with a set of points and appropriate distance measure, and initially place each point in its own group. And then while the number of groups is greater than one, merge the closest pair of groups into a single group. This basic structure leads to a whole collection of different methods depending on how distance between groups is measured. Some common measures are minimum, maximum, centroid, average, or sum-of-squares distance between the points in each group. The last, sum-of-squares method is also known as "Wards' method", and uses the function:

$$SS(G_{12}) - SS(G_1) - SS(G_2)$$

where

$$SS(G) = \frac{1}{2|G|} \sum_{x \in G, y \in G} (d_{xy})^2$$

Note that when using minimum as the distance measure the method simply finds the minimum spanning tree of the graph in which the items are the vertices and the distance $d(i, j)$ is used as the weight of an edge from i to j.

An interesting generalization of the above five group-distance measures is due to Lance and Williams, 1967. They show a function that generalizes the previous metrics for the distance between cluster k and the cluster consisting of sub-clusters i and j. The metric must be recursively applied. There are four parameters, α_i, α_j, β, and γ. n_x represents the number of elements inside cluster x.

$$d_{k(ij)} = \alpha_i d_{ki} + \alpha_j d_{kj} + \beta d_{ij} + \gamma |d_{ki} - d_{kj}|$$

With this formula, we can produce the nearest neighbor metric,

$$\alpha_i = \alpha_j = \frac{1}{2};\ \beta = 0;\ \gamma = -\frac{1}{2}$$

the furthest neighbor metric,

$$\alpha_i = \alpha_j = \frac{1}{2};\ \beta = 0;\ \gamma = \frac{1}{2}$$

the centroid metric,

$$\alpha_i = \frac{n_i}{n_i + n_j};\ \alpha_j = \frac{n_j}{n_i + n_j};\ \beta = -\alpha_i \alpha_j$$

and Wards' method,

$$\alpha_i = \frac{n_k + n_i}{n_k + n_i + n_j};\ \alpha_j = \frac{n_k + n_j}{n_k + n_i + n_j};\ \beta = \frac{-n_k}{n_k + n_i + n_j}$$

3.2 Top-Down Hierarchical Algorithms

Top-down clustering approaches are broken into *polythetic* and *monothetic* methods. A monothetic approach divides clusters based on analysis of one variable at a time. A polythetic approach divides clusters into arbitrary subsets.

Splinter Group Accumulation

A technique called *splinter group accumulation* removes the most outstanding item in a cluster and adds it to another, repeating until moving the item will increase the total cost measure. Pseudo-code for this algorithm follows, where we assume the measure we are trying to optimize is the average distance within a group.

$C_1 = P$
$C_2 = \{\}$
loop
    for each $p_i \in C_1$
        $d_i = \frac{1}{|C_2|} \sum_{p_j \in C_2} d(i,j) - \frac{1}{|C_1|} \sum_{p_j \in C_1} d(i,j)$
    if $\max d_i > 0$
        $C_2 = C_2 \cup P_i$
        $C_1 = C_1 - P_i$
    else
        return

We could also use a different measure in the code above, such as Wards' method.

Graph Separators

A cut of a graph separates vertices V into two sets, V_1 and V_2. The weight of a cut is the sum of each edge weight crossing the cut:

$$W_{cut} = \sum_{i \in V_1} \sum_{j \in V_2} w_{ij}$$

Assuming we use a similarity measure, the goal is to find a cut which minimizes W_{cut}, where the weights are the similarities, while keeping the two clusters approximately the same size. Various conditions can be used, such as forcing the ratio of sizes to be bounded by some number, usually less than two. In the general case, finding the optimal separator is NP-hard.

Finding good separators for graphs is a topic of its own. Other applications of graphs separators include

- Solving sparse linear systems (nested dissection). The goal is to minimize *fill*.
- Distributing graphs or sparse matrices on multiprocessors. The goal is to minimize parallel communication during program execution.
- VLSI layout–finding placements that minimize communication.

For geometric problems, methods such as the *kd*-tree or circular cuts can be used to compute separators. Without geometric information, the *spectral method* uses a technique similar to singular value decomposition (or principal component analysis). The principal component (actually the second eigenvector since the first turns out to be "trivial") is used to separate the points — the points are split by the median value of the vector.

3.3 Optimization Algorithms

Expressed as an optimization problem, clustering attempts to divide data into k groups to maximize or minimize some measure, *e.g.*,

- sum of intra-group distances
- sum squared of intra-group distances
- maximum intra-group distances

Most measures lead to an NP-hard problem and heuristics are often applied.

Optimization as an Integer Program

One technique uses integer programming to minimize the sum of intra-group distances. Stated as an integer programming problem:

minimize $\sum_{i=1}^{n}\sum_{j=1}^{n} d_{ij}x_{ij}$
subject to
$1 \leq i \leq n$
$1 \leq j \leq n$
$1 \leq k \leq g$
$\sum_{j=1}^{g} y_{ij} = 1$
$\frac{1}{g}\sum_{k=1}^{g} z_{ijk} \leq x_{ij}$
$y_{ik} + y_{jk} - 1 \leq z_{ijk}$

where g is the number of groups, $x_{ij} = 1$ if object i and j are in the same group, $y_{ik} = 1$ if object i is in group k, and $z_{ijk} = 1$ if objects i and j are in group k. The first condition states that each object must be in one group. The second condition states that i and j may appear in at most one group together. The final condition relates y and z, stating that if two objects are in the same group, z is true.

3.4 Density Search Algorithms

Density search techniques try to find regions of "high density". There are many such techniques. A particularly simple method starts with the minimum-spanning-tree for the graph:

1. find MST
2. remove all edges with weights much greater than the average for its vertices
3. find connected components.

This method has the advantage of finding "natural" groupings rather than fixing the number of groups, because the size and granularity of the clusters it produces are dependent on the parameter used to remove edges.

3.5 Summary

There are many algorithms for clustering. The key question to be answered before picking an algorithm is what the model of a good clustering is. The challenge for authors of these algorithms today seems to be making clustering fast for large collections of data.

- Comment on Assignment (Guy)
- Goals of HotBot
- Internal Architecture
  1. Semantics
  2. Hardware
  3. Data Base Model
  4. Score
  5. Filters
- Optimizations
  1. Caching
  2. Compression
  3. Parallelism
  4. Misc.
- Crawling
- Summary
- References

1 Comment on Assignment (Guy)

Problem 3

Problem 3 in the last assignment could be done by combining the affine gap and space efficiency examples given in class and in the readings. There, however, were a few caveats which needed to be addressed.

First, the algorithm must keep separate pointers for the E and S entries. This is necessary because the path could have come from an E or S crossing. The F entries can be generated on the fly.

Second, gaps which cross the middle row must be dealt with carefully (Fig. 152). If not the algorithm could double charge for that gap. A solution is to store both pointers back to the middle row for both E and S.

Only four arrays of size O(m) are required in a space efficient algorithm (see Fig. 153). They hold E, S, and the pointers pointing back to the location in the middle row from where E and S came.

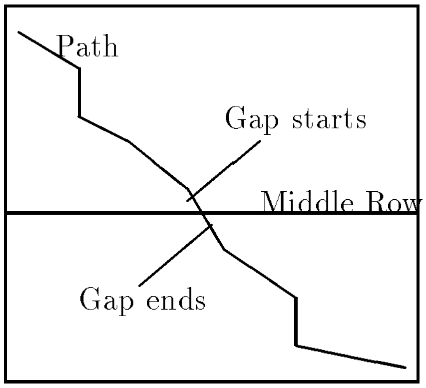


Figure 152: A gap may cross the middle row

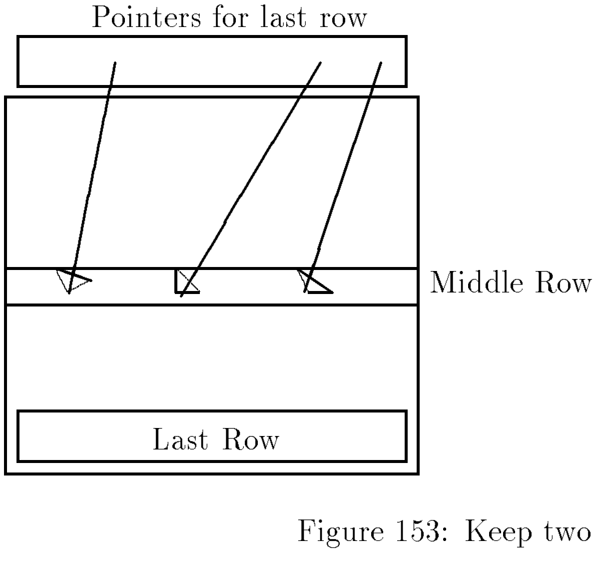


Figure 153: Keep two arrays for each of S and E

2 Goals of HotBot

History

HotBot originated in the Berkeley NOW project. An application was needed that would show the benefits of clustering. A search engine was decided upon as the killer app. because its requirements closely matched what cluster computing could provide.

Availability

In order to achieve 24x7 uptime, a mission critical infrastructure is necessary. Fault tolerance and redundancy are cornerstones in achieving this goal. The multiple individual machines in clusters provide a natural method.

Scalability

Additionally, a search engine must expand its service as client requests grow. Since growth in the WEB, however, is occurring at a staggering rate, big, expensive machines, quickly would become overloaded, with replacement the only upgrade option. So, incremental scalability could provide a large economic advantage over the longer term.

Not only the hardware, but the software design must take all this into account. Fine grained parallelism and multi-processor support should adapt well in this environment.

Both the availability and scalability requirements for a search engine could be addressed by the NOW clusters.

3 Internal Architecture

3.1 Semantics

The semantics of a search engine are simple. Although the search engine itself must be highly available, that data it contains does not. The WEB can always be used as a last recourse to retrieve any requested information.

Additionally, a very weak consistency model can be maintained. Known as BASE [1], it allows stale information to persist for a limited time.

3.2 Hardware

Currently 330 UltraSPARC CPUs power HotBot. The system is very homogeneous, which allows a node to take over for an arbitrary one when necessary. These are connected to a vast number of RAID disk subsystems which contain over 0.5TB of data.

If a single node goes down, the data associated with that node becomes temporarily unavailable. This is okay due to the weak consistency model. Also, this constitutes only a small fraction of the total data. Note that the system as a whole, however, continues to process requests despite this.

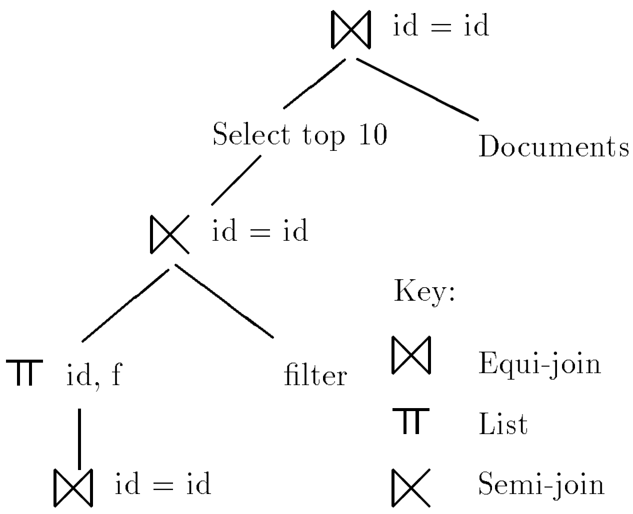


Figure 154: An example of a search query

In comparison to a single, large, powerful SMP system, HotBot's commodity parts are faster and cost less. For example, in order to expand the service, additional, ordinary systems can be bought, adding their power to the current set. SMP's, however, are not additive in this respect, nor do they provide any form of fault tolerance in system up-time. At the very least, replication is necessary, requiring additional, expensive hardware.

3.3 Data Base Model

The search engine algorithms can be described in relational data base (RDBMS) terms. There exists a relation w (a row in a table) for each unique word in each document found in the WEB: w[document ID, score, position information]. The entry keeps track of the document in which the word was found (using a unique ID). The score helps in matching the word to a search and is discussed below. The position information helps in phrase searching. There are currently about 10M words in HotBot's database with an average of about 1K lines per word. HotBot has to deal with some words which occur in almost every document and with many words that occur very infrequently.

There is also a relation D for all documents on the WEB where: D[ID, URL, etc ...]. This identifies each document with a unique ID, as well as information to retrieve the document later.

When the RDBMS needs to be updated with fresh data, an atomic swap operation swaps a new data base to replace the old between queries. This allows hot updates to occur without any interruption of service.

When a search query is received by HotBot, it goes through a series of relational joins. For example, if the query "A B filter:C" is received (i.e. search for A and B, then apply a boolean filter on the output based on C), Fig. 154 shows the relational operations. The first equi-join (lower left) is doing the search for A (think of it as an "and" operation). It then creates a list of the result, sorted by score (f). The semi-join then executes the boolean filter on the list. The top ten scores are selected and a final equi-join is done to get the associated

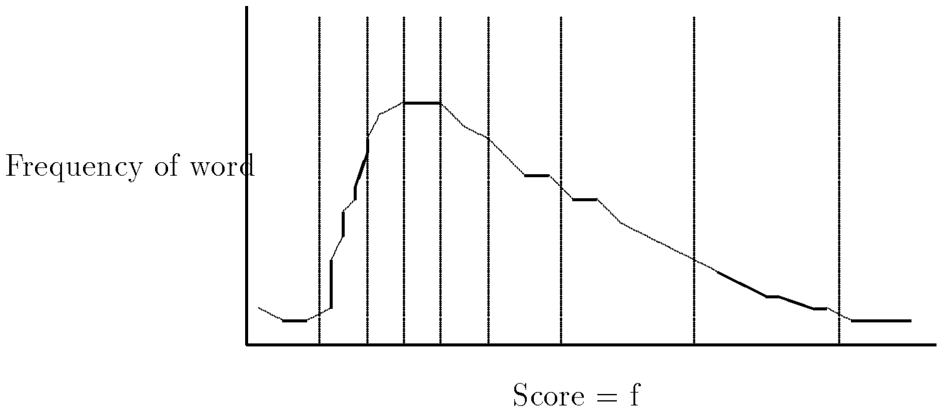


Figure 155: Determining the score

documents for the result page to return to the client.

3.4 Score

The score (f) determines when a word matches a query and how closely. Higher scores (weights) cause the document to appear higher on HotBot's priority lists to the RDBMS joins and on the result page returned to the client. The value is calculated as a function of a few different attributes, which includes:

1. Number of times a word appears in the document
2. Document length
3. If the word appears in the document title

This function is then broken down into equal area slices, and a "score" from 0 to 255 is assigned to each slice. See Fig. 155 for an example of how the function and slices could appear.

3.5 Filters

As shown in an example above, filters allow the query to express boolean matches. In fact, filters are implemented as meta-words, where w contains all documents that would match the specific filter. As a result, a filter can be thought of as the addition of this meta-word on the search line. Filters are use-full in expressing searches on features of pages that cannot be expressed otherwise (such as does the page contain an image). Note, however, that less-than and greater-than operators cannot be easily addressed by this mechanism. Dates, however, are important and do require some type of less-than, greater-than functionality. HotBot uses a trick which expands a date query into log(n) boolean queries, where "n" is the number of bits in the date specification.

4 Optimizations

In order to achieve better throughput of query transactions, optimizations help to streamline the search process.

4.1 Caching

Instead of just caching documents, HotBot also caches search results. In fact, the "next 10" button is actually another full search that completes quickly because of the cache.

4.2 Compression

Extensive compression helps to minimize the footprint of the data. Score and position information are split with IDs stored in both lists. Compression then allows the position information, which is used infrequently, to occupy minimal space. A type of delta-encoding is used (similar to Huffman) that uses a variable number of bits.

In HotBot, because of the large amount of data, nothing is uncompressed, even when loaded directly into memory, until necessary. The reasoning is that the additional page faults that would be generated by uncompressing when loading into memory outweighs the time to uncompress the data more often [2, 3].

4.3 Data Base

HotBot uses a technique called join selectivity to speed its RDBMS operations. It orders operations which have the highest probability of resulting in a short list first. This reduces the work of future RDBMS operations. For example, applying very specific boolean filters first could reduce list sizes for later equi-joins.

In addition, HotBot spreads the RDBMS data across many nodes in the cluster (by document ID in the D relation) at a relatively fine grain. Single node failures become manageable and the amount of data affected minimal. Also, when updating, the fine grains allow atomic swaps of the data to be simple and require little additional space.

4.4 Parallelism

When optimizing for parallelism, "balance" is the key word. If all nodes are working to their fullest, and there are no single bottlenecks, this allows for the highest degree of parallelism. HotBot attempts to address this by:

1. Randomizing data across the nodes

2. Use a large number of disks to hide seek latency

The randomizing reduces the risk of hot spots forming within the cluster (similar to the arguments in [4]).

HotBot also executes each query in parallel. Because each node is responsible for part of the total database, it first issues the query to each node (Fig. 156). The nodes then send the

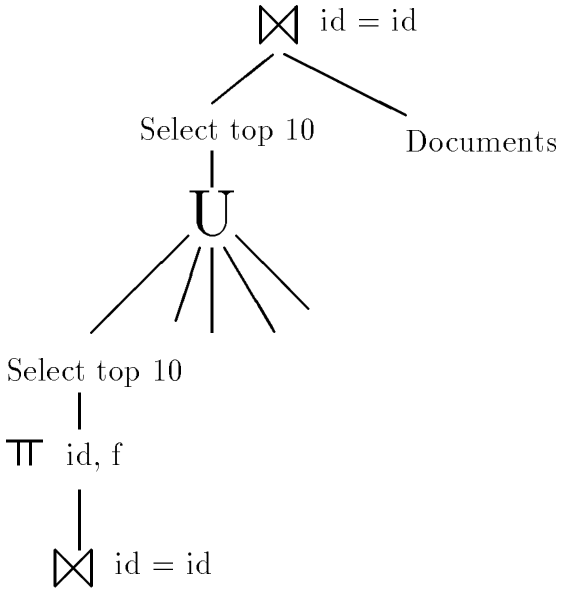


Figure 156: Parallelization of queries

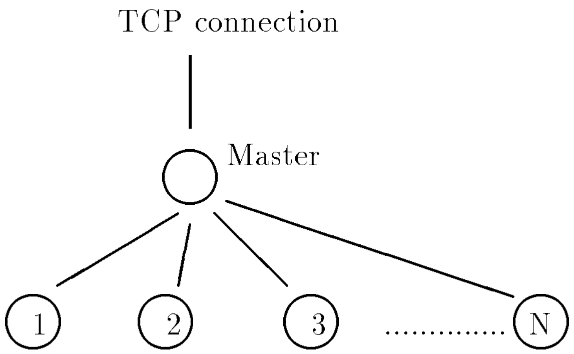


Figure 157: Naive fanout

highest scoring subset of their results back to a "master" node. This then takes the union of the answers, returns the top scores among all of the nodes responding, then does the final equi-join to associate the answers with the documents. Note that every query utilizes every node.

Another problem is how the fanout from the original master node, that initially receives the query, happens as the queries are being distributed (Fig. 157). A naive solution would be to send the queries in sequence starting from the first node. This, however, could lead to hot spots or head of line blocking (queries are artificially blocked on a resource due to a previous query) occurring in the fanout chain.

HotBot attempts to avoid problems by using two techniques (Fig. 158):

1. Fanout in multiple levels (a fat tree structure)

2. At each stage, fanout to random nodes

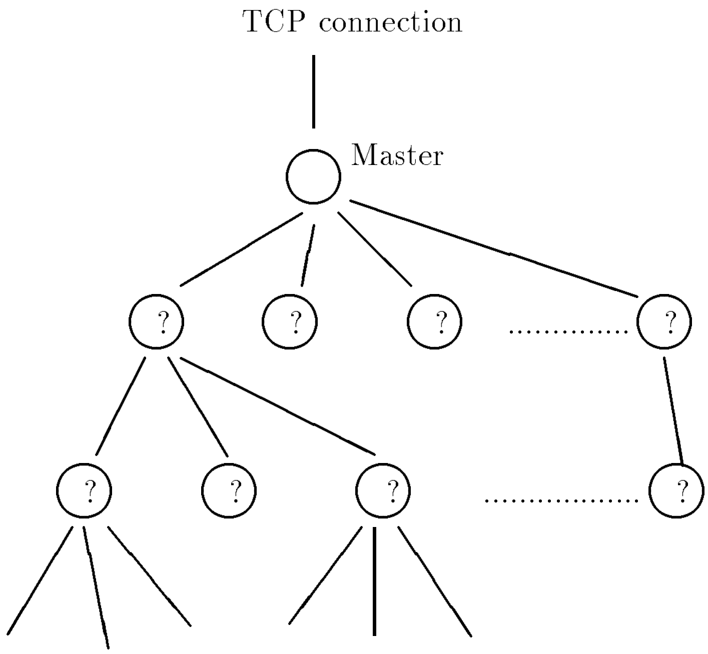


Figure 158: Optimized fanout (each ? is a random node)

In the future, HotBot could take advantage of knowledge of the physical network layout. A more optimized fanout may know where the limiting bisection bandwidths between cut-sets of the network are located and incur less traffic over those connections.

4.5 Misc.

Some additional optimizations are done to speed up each query:

1. Precomputation: As much information as possible is computed off-line. For example, the scores can be computed and stored as the pages are read in from the WEB. Also, filters (i.e. meta-words) can be created and managed before queries requesting the information arrive.

2. Partitions of the nodes are updated at varying times. This gives a better illusion that the data is up-to-date. Some data, such as news groups are updated more frequently.

3. HotBot will dynamically skip words that are too expensive to compute. This usually occurs with words that appear in many documents, such as "a" or "the".

5 Crawling

HotBot has a separate partition that crawls the WEB at about 10M pages a day. It uses "popular" sites (such as possibly Yahoo) as the roots of these searches. These updates eventually enter the HotBot database at various times.

6 Summary

The benefits gained from clustering matched the expectations placed on search engines well. Their availability, scalability, and price/performance ratio cannot be achieved even by more expensive SMP servers.

References

[1] A. Fox, S. Gribble, Y. Chawathe, E. Brewer, and P. Gauthier, *Cluster-Based Scalable Network Services*, (To appear at SOSP-16, Oct. 1997).

[2] Fred Douglis. *The Compression Cache: Using On-line Compression to Extend Physical Memory.* USENIX Proceedings, Winter 1993.

[3] D. Banks, M Stemm. *Investigating Virtual Memory Compression on Portable Architectures.* `http://HTTP.CS.Berkeley.EDU/~stemm/classes/cs262.ps.gz`

[4] Waldspurger and Weihl. *Lottery Scheduling: Flexible Proportional-Share Resource Management.* Proceedings of the First Symposium of Operating System Design and Implementation, November 1994.

1 References by Topic

1.1 Compression

Primary Texts

Khalid Sayood. *Introduction to Data Compression.* Morgan Kaufmann, 1996.

Other Sources

Timothy C. Bell, John G. Cleary, and Ian H. Witten. *Text compression.* Prentice Hall, 1990.

Gilbert Held and Thomas R. Marshall. *Data and Image Compression: Tools and Techniques.* Wiley, 1996 (4th ed.)

Mark Nelson. *The Data Compression Book.* M&T Books, 1995.

James A. Storer (Editor). *Image and Text Compression.* Kluwer, 1992.

Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images.* Van Nostrand Reinhold, 1994.

1.2 Cryptography

Primary Texts

Alfred J. Menezes, Paul C. van Oorschot and Scott A. Vanstone. *Handbook of Applied Cryptography.* CRC Press, 1996.

Bruce Schneier. *Applied Cryptography.* Wiley, 1996.

Douglas R. Stinson. *Cryptography: Theory and Practice.* CRC Press, 1995.

Other Sources

G. Brassard. *Modern Cryptology: a tutorial.* Spinger-Verlag, 1988.

N. Koblitz. *A course in number theory and cryptography.* Springer-Verlag, 1987.

C. Pfleeger. *Security in Computing.* Prentice-Hall, 1989.

A. Saloma. *Public-key cryptography.* Springer-Verlag, 1990.

Jennifer Seberry and Josed Pieprzyk. *Cryptography: An Introduction to Computer Security.* Prentice-Hall, 1989.

D. Welsh. *Codes and Cryptography.* Claredon Press, 1988.

1.3 Linear and Integer Programming

Texts

M. Bazaraa, J. Jarvis, and H. Sherali. *Linear Programming and Network Flows.* Wiley, 1990.

Dimitris Bertsimas and John N. Tsitsiklis. *Introduction to Linear Optimization.* Athena Scientific, 1997

J. P. Ignizio and T. M. Cavalier, *Linear Programming.* Prentice Hall, 1994

G. L. Nemhauser, A.H.G. Rinnooy Kan, and M. J. Todd (ed.). *Optimization.* Elsevier, 1989.

George L. Nemhauser and Laurence A. Wolsey. *Integer and combinatorial optimization.* Wiley, 1988.

H. M Salkin and K. Mathur. *Foundations of Integer Programming.* North-Holland, 1989.

Robert J. Vanderbei. *Linear Programming: Foundations and Extensions.* Academic Publishers, 1996

Stavros A. Zenios (ed.). *Financial optimization.* Cambridge University Press, 1993.

Useful Overview Papers

G. L. Nemhauser. *The age of optimization: solving large-scale real-world problems.* Operations Research, 42(1), Jan.-Feb. 1994, pp. 5-13.

E. L. Johnson and G. L. Nemhauser. *Recent developments and future directions in mathematical programming.* IBM Systems Journal, vol.31, no.1; 1992; pp. 79-93.

I. J. Lustig, R. E. Marsten and D. F. Shanno. *Interior point methods for linear programming: computational state of the art.* ORSA Journal on Computing, vol.6, no.1; Winter 1994; pp. 1-14.

1.4 Triangulation

Primary Texts

M. de Berg, M. van Kreveld M. Overmars, and O. Schwarzkopf. *Computational Geometry, Algorithms and Applications.* Springer-Verlag, 1997.

Other Texts

J. E. Goodman, and J. O'Rourke (Editors). *Handbook of Discrete and Computational Geometry.* CRC Press, 1997.

K. Mulmuley. *Computational Geometry: An Introduction Through Randomized Algorithms.* Prentice Hall, 1994.

J. Nievergelt and K. Hinrichs. *Algorithms and data structures: with applications to graphics and geometry.* Prentice Hall, 1993.

J. Pach and P.K. Agarwal. *Combinatorial Geometry.* John Wiley and Sons, 1995.

Preparata, Franco P. *Computational geometry: an introduction.* Springer-Verlag, 1988.

A. Okabe, B. Boots, and K. Sugihara. *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams.* John Wiley, 1992.

O'Rourke. *Computational Geometry in C.* Cambridge University Press, 1994.

1.5 N-body Simulation

Susanne Pfalzner and Paul Gibbon. *Many Body Tree Methods in Physics.* Cambridge University Press, 1997.

L. Greengard. *Fast algorithms for classical physics.* Science, vol. 265, no. 5174, 12 Aug. 1994, pp. 909-14.

J. E. Barnes and P. Hut. *A hierarchical O(N Log N) force calculation algorithm.* Nature, 324(4):446-449, December 1986.

1.6 VLSI Physical Design

Primary Texts

N. Sherwani. *Algorithms for VLSI Physical Design Automation, Second Edition.* Kluwer, 1995.

M. Sarrafzadeh and C. K. Wong. *An Introduction to VLSI Physical Design.* McGraw-Hill, 1996.

Other Texts

P. Banerjee. *Parallel Algorithms for VLSI Computer-Aided Design.* Prentice Hall, 1994.

A. B. Kahng and G. Robins. *On Optimal Interconnections for VLSI.* Kluwer Academic Publishers, 1995.

B. Korte, L Lovasz, H. J. Promel, and A. Schrijver (eds.). *Paths, flows, and VLSI-layout.* Springer Verlag, 1990.

T. Lengauer, *Combinatorial algorithms for integrated circuit layout.* Wiley, 1990.

M. Pecht and Y. T. Wong (editors). *Advanced Routing of Electronic Modules.* CRC Press, 1995.

M. Sarrafzadeh and D.T. Lee (editors). *Algorithmic Aspects of VLSI Layout (Lecture Notes Series on Computing, Vol 2).* World Scientific, 1994.

1.7 Pattern Matching in Biology

Dan Gusfield. *Algorithms on String, Trees, and Sequences.* Cambridge University Press, 1997.

Arthur M. Lesk. *Computational molecular biology: sources and methods for sequence analysis.* Oxford University Press, 1988.

Graham A Stephen. *String searching algorithms.* World Scientific, 1994.

M.S. Waterman. *Introduction to Computational Biology: Maps, Sequences, Genomes.* Chapman & Hall, 1995.

1.8 Indexing and Searching

Christos Faloutsos. *Searching Multimedia Data Bases by Content.* Kluwer Academic, 1996.

Frakes and Baeza-Yates (ed.). *Information Retrieval: Data Structures and Algorithms.* Prentice Hall, 1992

Frants V. J., Shapiro J., Voiskunskii V. G. *Automated Information Retrieval Theory and Methods.* Academic Press, Aug 1997.

Lesk M. *Practical Digital Libraries Books, Bytes & Bucks.* Morgan Kaufman Publishers, 1997.

Salton. *Automatic Text Processing: The Transformation, Analysis and Retrieval of Information by Computer.* Addison-Wesley, 1989.

Sparck Jones K. and Willett P. (editors). *Readings in Information Retrieval.* Morgan Kaufman Publishers, 1997.

Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images.* Van Nostrand Reinhold, 1994.

1.9 Clustering

Leonard Kaufman and Peter J. Rousseeuw. *Finding Groups in Data : An Introduction to Cluster Analysis.* Wiley, 1990.

Everitt, Brian. *Cluster analysis (3d ed.).* Wiley, 1993.

E. Backer. *Computer-Assisted Reasoning in Cluster Analysis.* Prentice Hall, 1995.

2 Assignments

Algorithms in the Real World
Assignment #1 (Compression) Due Date: 12 Sept 97

Collaboration policy: You are welcome to talk about this or other assignments with others and even discuss problems using a whiteboard or scratch paper. You are not allowed, however, to show or give any work that you did on your own pertaining to the assignment to someone else.

Problem 1: Assume an alphabet $S = \{a, b, c\}$ has the probabilities

$$p_a = .2 \quad p_b = .7 \quad p_c = .1$$

(a) What is the entropy $H(S)$?

(b) What is the average length of the Huffman code for S (remember to weigh the average based on the probabilities).

(c) What is the average length of the Huffman code if you group characters into pairs.

(d) For a long string what is the average per-character length of an arithmetic code for S (assume the asymptotic value and that the message abides by the probabilities).

(e) For a long string what is the average per-character length of an arithmetic code for S assuming your model are the probabilities given above, but you are coding a message which consists of just c's (i.e., your data does not fit your model).

(f) Given the following conditional probabilities what is the conditional entropy (this is the average per-character length that would be achieved by a good code that takes advantage of the probabilities).

$$\begin{array}{lll} p(a|a) = .25 & p(b|a) = .1 & p(c|a) = .8 \\ p(a|b) = .65 & p(b|b) = .8 & p(c|b) = .1 \\ p(a|c) = .1 & p(b|c) = .1 & p(c|c) = .1 \end{array}$$

The notation $p(x|y)$ is the probability of x given that the previous character is y.

Problem 2: Prove that the average length $l_a = \sum p_i l_i$ for a Huffman code for a set of messages S is bounded by the inequality $l_a \leq H(S) + 1$. To prove this you can assume the Huffman code is optimal. Note that given a maximum probability p_m a tigher upper bound of

$$l_a < \begin{cases} H(S) + p_m & p_m \geq .5 \\ H(S) + p_m + .086 & p_m < .5 \end{cases}$$

can be shown. Feel free to prove this stronger result.

Problem 3: Assume we are coding a 4 letter alphabet with LZW and we start with the dictionary containing the 4 letters $\{a, b, c, d\}$ in locations 0, 1, 2 and 3. Entries are added to the dictionary in sequential order. Decode the following message that was coded with LZW.

```
0 0 1 2 4 7
```

Problem 4: Using the LZSS (the variant of LZ77 described in the reading "Introduction to data compression" by Peter Gutmann) and assuming a window size W and a lookahead buffer size B, how many code words will it take to code a message of length L that contains all the same character. Assume each ¡offset,size¿ pair or character is a codeword. Also assume the window is initialized to some other character.

Problem 5: Lets say we had the following sequence of values

```
10 11 12 11 12 13 12 11 10 -10 8 -7 8 -8 7 -7
```

(a) Would we would get better compression by doing a cosine transform across the whole group of 16 or by blocking the 16 entries into two groups of 8 and transforming each separately? Explain briefly?

(b) If using a wavelet transform are we better off blocking the entries into two groups of 8?

Algorithms in the Real World
Assignment #2 (Cryptography) Due Date: 1 Oct 97

Problem 1:

Assume you design an encryption scheme with a single round DES (instead of the usual 16 rounds and ignoring the initial and final permutations). If you know a message cipher pair (m_1, c_1) for a key k for the values given below, what is m_2 assuming $c_2 = DES1_k(m_2)$. Assume the following hexidecimal values for m_1, c_1 and c_2.

$$\begin{aligned} m_1 &= \texttt{3F72D4E6 04CD825B} \\ c_1 &= \texttt{04CD825B 8E25E338} \\ c_2 &= \texttt{04CD825B 327C7D29} \end{aligned}$$

If you show the first 2 (leftmost) hexidecimal digits of m_2 that will convince me you know the rest.

Problem 2:

Assume we have a 3×3-bit s-box used in a block ciphers. For each of the following three substitutions explain what is wrong.

Input	out_1	out_2	out_3
0	2	0	4
1	5	4	3
2	3	1	7
3	7	5	2
4	0	2	5
5	4	6	0
6	3	3	6
7	1	7	1

Problem 3:

(a) Assuming p and q are primes, express $x^{-1} \bmod pq$ as a positive power of x.

(b) Given public key $e = 3$, $n = 391$ encode the value 45 using RSA.

(c) Assuming the same public key, decode the value 105 (you might want to write a short program to do it).

(d) In RSA for some n lets say $e = 3$ (a standard choice in some systems). If I accidentally give away d show how one can use this information to factor n.

Problem 4:

(a) Using ElGamal Encryption with the public key $p = 11, g = 7, y = 5$ encrypt the value 9 with random $k = 3$.

(b) Decrypt the pair (2,2).

(c) Given $p = 11$ why is 10 a bad value for g?

Problem 5:

(a) Generate the first 4 bits of the BBS random bit generator assuming that $n = 77, r_0 = 39$.

(b) For an arbitrary Blum integer $n = pq$ give an upper bound on the period of the generator (how many iterations it takes before it repeats). This need not be the strongest upper-bound but you will get more credit the stronger it is.

Algorithms in the Real World
Assignment #3 (Integer and Linear Programming) Due Date: 15 Oct 97

Please feel free to send me email if there are any ambigouous questions.

Problem 1: Strang Chapter 8 problem 8.1.9

Problem 2: Strang Chapter 8 problem 8.2.1.

Problem 3: Strang Chapter 8 problem 8.2.13.

Problem 4:

Consider the following problem

$$\begin{array}{ll} \text{maximize} & x_1 + x_2 \\ \text{subject to} & 2x_1 + x_2 \leq 4 \\ & x1, x2 \geq 0 \end{array}$$

(a) Starting with the initial solution $(x_1, x_2) = .1, .1$ solve this problem using the affine scaling interior-point method.

(b) Illustrate the progress of the algorithm on the the graph in $x_1 - x_2$ space.

Problem 5:

Let $S = \{x \in R^n : Ax \leq b\}$ and $T = \{x \in R^n : Bx \leq d\}$. Assuming that S and T are nonempty, describe a polynomial time algorithm (in n) for checking whether $S \subset T$.

Problem 6:

Formulate the following problem as a mixed integer programming problem.

$$\begin{array}{lll} \text{maximize} & f_1(x) + f_2(x) & \\ & & x_1 + 3x_2 \leq 12 \\ \text{subject to} & \text{at least two of the following} & 2x_1 + x_2 \leq 16 \\ & & x_1 + x_2 \leq 9 \\ & x_1, x_2 \geq 0 & \end{array}$$

where

$$f_1(x_1) = \begin{cases} 10 + 2x_1, & 0 \leq x_1 \leq 5 \\ 15 + x_1, & \text{otherwise} \end{cases}$$

$$f_2(x_2) = \begin{cases} 8 + x_2, & 0 \leq x_2 \leq 2 \\ 4 + 3x_2 & \text{otherwise} \end{cases}$$

Problem 7:

Solve the following problem using the branch-and-bound integer programming procedure describe in class. Feel free to do it graphically.

$$\begin{array}{ll} \text{maximize} & 2x_1 + x_2 \\ \text{subject to} & 2x_1 - 2x_2 \leq 3 \\ & -2x_1 + x_2 \leq 2 \\ & 2x_1 + 2x_2 \leq 13 \\ & x_1, x_2 \geq 0 \\ & x_1, x_2 \text{ integers} \end{array}$$

Problem 8:

Solve the following problem using implicit enumeration (the 0-1 programming technique) describe in class.

$$\begin{array}{ll} \text{minimize} & 3x_1 + 2x_2 + 5x_3 + x_4 \\ \text{subject to} & -2x_1 + x_2 - x_3 - 2x_4 \leq -2 \\ & -x_1 - 5x_2 - 2x_3 + 3x_4 \leq -3 \\ & x_1, x_2, x_3, x_4 \text{ binary} \end{array}$$

Implement **one** of the following algorithms:

1. Rupert's Meshing Algorithm
2. Garland and Heckbert's Surface Modeling Algorithm
3. Callahan and Kosaraju's Well-Separated Pair Decomposition Algorithm

Feel free to work in groups of two, although I will expect a better job from a group of two. Also feel free to come and discuss the algorithms with me. I've described each of these algorithms in class and have also made papers available that describe the algorithms in more detail. Two of the papers are available on the web, and the third is in a JACM issue that is available the CS reading room (Soda 681).

Note: Source code for some of these algorithms is available on the web. If you choose to do a problem, please do not look at anyone elses source until you have handed in your implementation. You should feel free, however, to use other peoples code for various subroutines such as implementing a heap. Make sure you reference the code.

I have made some code for incremental Delaunay Triangulation available, which you might find useful for either of the first two algorithms. This is taken from the book "Graphics Gems IV", edited by Paul Heckbert, Academic Press, 1994. A copy of the associated chapter is available outside my door. The code is also available as part of a compressed tar file associated with the book. You might find, however, that it is just as easy if not easier to design your code from scratch since deciphering the code is not trivial. Also, as I mentioned in class, you might find a triangle based representation simpler (the code uses the quadedge representation). If you do use any part of the code, make sure to note in your code which parts you used.

You might also find Jonathan Shewchuk's Show-me code (showme.c) useful for plotting a set of triangles, or boundaries.

Rupert's Meshing Algorithm

You should implement the algorithm and run it on the test sets given below and report back the number of triangles for minimum angles in the ranges specified. Ideally you should generate a postscript drawing of the final triangulations. Although the drawings are not required they will certainly help you debug your program. The algorirthm is described in:

> Jim Rupert. "A Delaunay Refinement Algorithm for Quality 2-Dimensional Mesh Generation." Journal of Algorithms, 18(3):548-585, May 1995.

The test data files: (Note: these use the Poly format from Jonathan Schewchuk's Triangle code)

- The letter A from Rupert's paper. Run this from 15 to 25 degrees in steps of 2 degrees.
- Lake Superior from Rupert's paper. Run this from 10 to 15 degrees in 1 degree steps. Note that this input has an internal angle that is just slightly larger than 15 degrees, so it will not work with larger angles.
- A guitar from Jonathan Schewchuk. Run this from 15 to 25 degrees in steps of 2 degrees.

Note that all data is included in a bounding box. This simplifies the problem of dealing with exterior triangles (all triangles in the box are considered interior). In your implementation you don't need to remove the triangles in the exterior (between box and surface) nor in the "holes". You can therefore ignore the "holes" data at the end of the .poly files. If you do chose to remove the triangles, however, you can use the "holes" data by starting at each triangle with the hole point and removing triangles outwards until you hit a boundary segment. If you choose to remove them please state this in your results.

Garland and Heckbert's Surface Modeling Algorithm

You should implement the algorithm and run it on the test sets given below and report back the number of triangles for a range of maximum errors. Again generating a drawing of the triangulation will be helpful but is not necessary. The algorirthm is described in:

> Michael Garland and Paul Heckbert. "Fast Polygonal Approximation of Terrains and Height Fields." CMU-CS-95-181, CS Dept, Carnegie Mellon U., Sept. 1995.

The algorithm I want you to implement is number III in the paper (the version I described in class).

The test data files: (Note: these are binary files in STM format)

- Crater lake
- Ashby Gap, Virginia
- Desert around Mt. Tiefort, California
- Ozark, Missouri

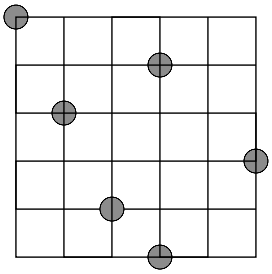
Callahan and Kosaraju's Well-Separated Pair Decomposition Algorithm:

You should implement the algorithm in 2 or 3 dimensions and run it on the Kuzmin model with 10K and 100K points (I originally said Plummer model, but I cannot find an equation for generating a Plummer distribution...if you do, please feel free to use it, but tell me.) For both sizes you should report the number of interactions as a function of the separation constant s ranging from 1 to 2 in increments of .2. The algorirthm is described in:

> Paul B. Callahan and S. Rao Kosaraju. "A Decomposition of Multi-Dimensional Point-Sets with Applications to k-Nearest-Neighbors and n-Body Potential Fields." JACM, 42(1):67-90, January 1995.
> Available for copying in the CS reading room.

Please feel free to send me email if there are any ambiguous questions.

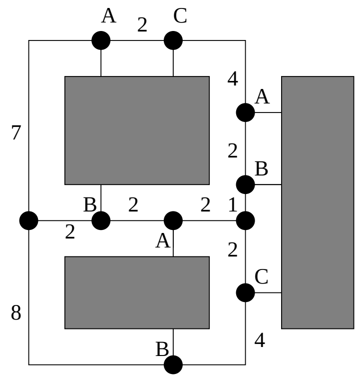
Problem 1: Consider the following graph.



A. Find a minimum cost Rectilinear Steiner Tree (RST) for the following set of points (the shaded points are the demand points).

B. Is this an S-RST? If not show the minimum cost S-RST.

Problem 2: For the following routing problem the three nets A, B and C need to be connected. The numbers on the edges represent their lengths, and assume that the capacity of every edge is 2.



A. Specify an integer (0-1) program for which the solution gives the minimum total wire length.

B. Solve the program (you do not need to show your work).

Problem 3: Describe an $O(nm)$ time $O(n+m)$ space algorithm for finding a minimum cost alignment assuming an affine gap cost model. Pseudocode with comments would be the best way to describe it.

Problem 4: (**Extra Credit.**) Describe an $O(nm \log(m+n))$ time algorithms for finding a minimum cost alignment assuming the gap cost function $x(k)$ (where k is the length of the gap) is concave downward (*i.e.* the second derivative is always negative).