

15-750 — Graduate Algorithms — Spring 2008

Miller and Sinop and Wu

Assignment 0 Due date: Friday, January 25

(Part of the solutions are taken from the homework of Dongsu Han, Bin Fan, Jiquan Ngiam)

1 Asymptotic Notation

1.1 List 1: Fast growing functions

Take log and we have:

$$\begin{aligned} \log\left((\log^* n)^{\log n}\right) &= \log n \cdot \log(\log^* n) \\ \log\left(n^{\log^* n}\right) &= \log n \cdot \log^* n \\ \log n! &\approx \log\left(e^{-n} n^n \sqrt{2\pi n}\right) = n(\log n - \log e) + \frac{1}{2}(\log n + \log 2\pi) \\ \log 2^{n\sqrt{\log n}} &= n\sqrt{\log n} \end{aligned}$$

So we have:

$$(\log^* n)^{\log n} < n^{\log^* n} < 2^{n\sqrt{\log n}} < n!$$

1.2 List 2: Slow growing functions

First we have:

$$\log \log \sqrt{n} < \sqrt{n} < n.$$

To compare $2^{\sqrt{\log n}}$ and \sqrt{n} first. We take log to each function and get:

$$\log(2^{\sqrt{\log n}}) = \sqrt{\log n} \text{ and } \log \sqrt{n} = \frac{1}{2} \log n.$$

Thus we know $2^{\sqrt{\log n}} < \sqrt{n}$.

For $2^{\log^* n}$ and $\log \log \sqrt{n}$, take log and we have:

$$\begin{aligned} \log(\log \log \sqrt{n}) &= \log \log \frac{1}{2} \log n = \log \log \log n \\ \text{and } \log(2^{\log^* n}) &= \sqrt{\log n}, \end{aligned}$$

easily we get: $\log \log \sqrt{n} < 2^{\sqrt{\log n}}$.

Last let us compare $2^{\log^* n}$ and $\log \log \sqrt{n}$.

$$\begin{aligned} \log(2^{\log^* n}) &= \log^* n = 3 + \log^*(\log \log \log n) = O(\log^*(\log \log \log n)) = o(\log \log \log n) \\ \text{and } \log(\log \log \sqrt{n}) &= O(\log \log \log n). \end{aligned}$$

Finally, we conclude:

$$2^{\log^* n} < \log \log \sqrt{n} < 2^{\sqrt{\log n}} < \sqrt{n} < n.$$

2 Whack-a-Mole

Define a function that returns points given time and the location of mallet.

$$p(t, pos) = \begin{cases} p_t & \text{if } m_t = pos \\ 0 & \text{else} \end{cases}$$

This algorithm computes $P[t, pos]$, the maximum points at time t provided that the mallet is at pos in time t .

Algorithm

Input: m_t

Output: x_t

// 1. Initialize

For $i = 1$ to n

$P[1, i] = p(1, i)$

// 2. Record intermediate results

For $t = 2$ to T

For $i = 0$ to n

$P[t, i] = \max(P[t-1, i+1], P[t-1, i], P[t-1, i-1]) + p(t, i)$

// 3. Find the maximum path

$lastpos = \operatorname{argmax}_i P[T, i]$

$x[T] = lastpos$

For $t = T - 1$ to 1

$x[t] = \operatorname{argmax}_{lastpos-1 \leq k \leq lastpos+1} P[t, k]$

return $x_t = (x[1], x[2], \dots, x[T])$

Correctness

We first prove the correctness of the algorithm in computing P by induction on t .

Base case. $t=1$ then $P[1, pos] = p(1, pos)$. Step 1 correctly reflects this.

Induction. By induction hypothesis, $P[t-1, \text{pos}]$ gives the maximum points possible at time $(t-1)$ provided that the mallet is at pos in time $(t-1)$. In calculating $P[t, \text{pos}]$ we only need to consider $P[t-1, \text{pos}-1]$, $P[t-1, \text{pos}]$, and $P[t-1, \text{pos}+1]$. In other words, the mallet can only be at $\text{pos}-1$, pos , or $\text{pos}+1$ in time $(t-1)$. Therefore $P[t, \text{pos}] = \max(P[t-1, \text{pos}-1], P[t-1, \text{pos}], P[t-1, \text{pos}+1]) + p(t, \text{pos})$. This is done in step 2 of the algorithm.

So the function P is correct.

Finding the optimal sequence given P is done in step 3, which is exactly tracing back where the maximum came from. This step is trivial and the algorithm is correct.

Efficiency

Step 1 takes $O(n)$, step 2 takes $O(nT)$, and step 3 $O(T)$. Therefore it's polynomial in n and T .

3 Upper and Lower Bounds

1. We give the following recursive algorithm to find the n th smallest element in sorted array A and B (assume in non-decreasing order):

Algorithm 1 FindMedian(A, B)

```

1: n = size of A
2: if n == 1 then
3:   return min(A[1], B[1])
4: else
5:   if A[⌊n/2⌋] > B[⌊n/2⌋] then
6:     return FindMedian(A[1...⌊n/2⌋], B[⌊n/2⌋...n])
7:   else
8:     return FindMedian(A[⌊n/2⌋...n], B[1...⌊n/2⌋])
9:   end if
10: end if

```

Run Time Analysis: Every time FindMedian is called, the size of problem is reduced by one half. So finally the algorithm will give the answer in $O(\log n)$ time.

2. The output of the algorithm could be any element in $A \cup B$ given arbitrary sorted array A and B . Thus initially the set of all possible outputs, denoted by S , is actually $A \cup B$. We can think of each comparison as splitting S into two groups: those for which the comparison result is GREATER and those for which the result is LESS. Given A and B , each comparison will make the size of possible output set smaller and smaller. Finally the size of S will be reduced to 1 when we have the answer for the corresponding A and B . In the worst case, the comparison can be always the one corresponding to the larger group. Then, each comparison cuts down the size of S by at most a factor of 2. We will need at least $\log(2n)$ comparisons to reduce $|S|$ to 1. Therefore any comparison-based algorithm must make $\Omega(\log n)$ comparisons for the worst case.

4 Probability

1. $1/2$

$$P(e_{ij} \text{ is a cut}) = P(v_i \in S, v_j \in \bar{S}) + P(v_i \in \bar{S}, v_j \in S) = 1/4 + 1/4 = 1/2$$

2. Every edge has 1/2 chance to be a cut and such event on every edge are independent. Therefore the expected value of total cut is $\frac{1}{2}m$.
3. Set $S = \bar{S} = \phi$. We keep adding vertex $v \in V$ into S or \bar{S} depending on which will give a bigger cut increase. Suppose the sequence of vertex we add is v_1, v_2, \dots, v_n . When we add v_i , at least half of the edges from v_i to $v_j (j < i)$ will be added. Therefore, when the algorithm finishes, our final cut will have

$$\text{cut}(S, \bar{S}) \geq 1/2 \sum_{i=1}^n \# \text{edges from } v_i \text{ to } v_j (j < i) = m/2.$$

5 The Silhouette

Using the heap data structure, consider a new function `deletekey(H, k)` that deletes key k from the heap. This operation can be easily performed in $O(\log n)$ time as long as pointers directly to the heap elements are maintained. One means of implementation is to take the last element (in a binary heap) and replace the element for key k and let it float up or sink down.

In the following algorithm, we use a max-heap, where each element of the heap is a 2-tuple, (key, priority).

Correctness

The algorithm scans left to right, keeping track of the set of buildings at each position.

Lines 8-9 sort the buildings so that we process them in order of left to right as well as in order of introduction and removal - this is required if one wished to handle buildings overlapping at the same position or buildings with infinitesimal width.

One should first observe that points for the output for the silhouette must occur at left and right locations of the buildings.

The loop invariant maintained is that the set of buildings in the heap at each position (each iteration of the loop) is exactly the set of buildings overlapping at that point. This is easily seen as buildings are added and removed only when the left or right edge are met. At the same time, the inner loop (16-20) adds all the buildings at the same position. This is so that we do not output more than once when buildings happen to overlap at exactly the same x-location. Finally, the heap stores all the buildings present using their height as the priorities. Then, at every iteration with a change in height is detected, an output is created based on the change and this will exactly draw the silhouette.

Timing

The two stable sorting operations take $O(n \log(n))$ to execute (merge-sort for example). Every heap operation takes $O(\log n)$ with a basic binary heap. There are $O(n)$ calls to these heap operations, making a total of operations $O(n \log(n))$ as well. The rest of the operations are clearly dominated by these procedures.

Hence, the runtime of the algorithm is $O(n \log n)$.

Algorithm 2 Silhouette(B,n)

```

1: Initialize a max-heap H (for heights)
2: Insert (0, 0) into H // Base-Height
3: Construct an empty array A of size 2n
4: for Bi = (li, ri, hi) do
5:   Insert (li, left, i, hi) into A
6:   Insert (ri, right, i, hi) into A
7: end for
8: Stable-Sort array A by the values in the second value of each 4-tuple put the left entries first
   before right
9: Stable-Sort array A by the values in the first value of each 4-tuple
10: i = 1
11: for i=1.. 2n do
12:   Let (pos, mark, key, h) = Ai
13:   Let currentHeight = findMax(H)
14:   Let currentPos = pos
15:   if mark == left then
16:     while 1 do
17:       Insert (key, h) into H
18:       i++
19:       Let (pos, mark, key, h) = Ai
20:     end while mark =right OR currentPos ≠ pos OR i > 2n
21:   else
22:     while 1 do
23:       Delete (key) from H
24:       i++
25:       Let (pos, mark, key, h) = Ai
26:     end while mark =left OR currentPos ≠ pos OR i > 2n
27:   end if
28:   Let newHeight = findMax(H)
29:   If newHeight currentHeight Then
30:     Output (pos, currentHeight) (pos, newHeight)
31: end for

```
