 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.

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

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

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

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

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):

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

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 ≤ b
    // c is the objective vector
    // z* is the cost of the current best solution (initially −∞)
    // 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 ≤ 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 ≤ ⌊x'_i⌋
  z* = IP(C_d, c, z*)

  C_u = C, −x_i ≤ −⌈x'_i⌉
  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.