Bounding functions

Contents

1 Bounding functions

(see Donald Kreher and Douglas Stinson, *Combinatorial Algorithms*, section 4.6 for a reference on this material)

The pruning we've done so far was not very powerful. We need a framework in which to discuss better pruning. We'll do this with bounding functions

We need some vocabulary

Definition. Given a feasible solution X, let profit(X) be the profit of X.

Given a partial feasible solution X, let $P(X)$ be the maximum profit of any descendant of X in the state space tree.

Note that $P(\emptyset)$ is the optimal profit for the problem.

It is too slow to calculate $P(X)$ exactly – in general you need to traverse the whole subtree rooted at X, so what we want is to approximate $P(X)$ with something easier to compute

Definition. Let B be a function from the set of vertices of the state space tree to the positive integers. Suppose that for any partial solution X

$$B(X) \geq P(X).$$

Then we say B is a bounding function for the state space tree.

The point is that if we are at node X in a backtracking algorithm and we find that $B(X) \leq$ OptP, the optimal profit so far, then we have

$$P(X) \leq B(X) \leq \text{OptP}$$

so the subtree rooted at X can't improve the optimal profit and the subtree can be pruned away. A good bounding function B is

- close to $P(X)$
- easy to compute

These two features must be balanced.

This gives us a meta algorithm for bounded backtracking of a maximization problem

Meta algorithm: BoundedMaxBacktrack

```
arguments: m
global: X = [ x[1],x[2],...,x[n] ], OptP, OptX,
        C = [ C[1],C[2],...,C[n] ]
        Profit() // Profit(x[1],...,x[m]) is the profit if [x[1],...,x[m]] is feasible
        B()      // A bounding function which satisfies B(x[1],...,x[m]) >= Profit(X')
                 // for every feasible descendant X'=[x[1],...,x[m],x[m+1],...,x[k]]
if m = 0
    initialize data

if [x[1],...,x[m]] is a feasible solution
   CurP := Profit(x[1],...,x[m])
   if CurP > OptP
        OptP := CurP
        OptX := [x[1],...,x[m]]

if B(x[1],...,x[m]) <= OptP          //  Can we prune this node?
    return

Find the choice set C[m]
for x[m+1] in C[m]
    BoundedMaxBacktrack(m+1)
```

Note the above is phrased for maximizing (profit) problems. For minimizing (cost) the inequalities need to be flipped in the definitions of $P(X)$ and $B(X)$ and in the algorithm.

2 Bounding for the Knapsack problem

2.1 Rational Knapsack problem

To find a good bounding function for the Knapsack problem, consider a related problem: Given profits $p_1, \ldots, p_n$, weights $w_1, \ldots, w_n$ and capacity M, find rational numbers x_1, x_2, ..., x_n which solves the linear program

$$\text{Maximize} \sum_{i=1}^{n} p_i x_i$$

subject to

$$\sum_{i=1}^{n} w_i x_i \leq M$$
$$0 \leq x_i \leq 1, \text{ for } 1 \leq i \leq n.$$

This is called the Rational Knapsack Problem. The Knapsack Problem is the integer linear program obtained from this by additionally requiring that each x_i is an integer. In other words, Rational Knapsack is the *linear relaxation* of this integer program formulation of Knapsack. This change makes a big difference in how easily the problem can be solved. Here we can just take a greedy approach. First we relabel the items such that

$$\frac{p_1}{w_1} \geq \frac{p_2}{w_2} \geq \cdots \geq \frac{p_n}{w_n}.$$

Then we fill the knapsack with items in that order until the knapsack is full. An optimum solution will take the form

$$[x_1, x_2, \ldots, x_n] = [1, 1, \ldots, 1, x_i, 0, \ldots, 0]$$

for some index i and with $0 \leq x_i \leq 1$.

A similar paradigm works well for several other optimization problems; a good bounding function often arizes as the linear relazation of an integer linear program formulation of the problem.

Algorithm: RationalKnapsack

```
input: p[1],...,p[n],w[0],...w[n],M,
       assuming that  p[1]/w[1] >= p[2]/w[2] >= ... >= p[n])/w[n]
i := 0
P := 0
W := 0
[x[1],x[2],...,x[n]] := [0,0,...,0]
while W<M and i<n
   i := i+1
   if W+w[i] <= M
      x[i] := 1
      W := W+w[i]
      P := P+p[i]
x[i] := (M-W)/w[i]
return P+x[i]p[i]
```

This algorithm runs in linear time.

2.2 Bounded Knapsack

We can use the rational knapsack problem to give a bounding function for our original knapsack problem.

Definition. For a partial solution $X = [x_1, \ldots, x_m]$ to the Knapsack problem, define

$$B(X) = \sum_{i=1}^{m} p_i x_i + \text{RationalKnapsack}(p_{m+1}, p_{m+2}, \ldots, p_n, w_{m+1}, w_{m+2}, \ldots, w_n, M - \sum_{i=1}^{m} w_i x_i)$$

The first sum captures the profit of the partial solution. If each x_i in the solution of Rational Knapsack were 0 or 1, then we would have the profit of a descendant of X. Allowing rational values can only result in a larger profit than any descendant of X, so

$$B(X) \geq P(X).$$

So B defines a bounding function. Note that B is fast to compute since the rational knapsack algorithm is fast. This gives the following algorithm

Algorithm: BoundedKnapsack

```
input: m, CurW, CurP
global: OptX, OptP, C,
        X = [x[1],...x[n]]       // Assumption w[1]x[1] + ... w[m]x[m] <= M
        p[1],...,p[n],w[0],...w[n],M,
        assuming that  p[1]/w[1] >= p[2]/w[2] >= ... >= p[n])/w[n]
if m=0
     OptP = 0
if m=n
    if CurP > OptP
        OptP := CurP
        OptX := [x[1],...,x[n]]
    return
B := CurP + RationalKnapsack(p[m+1],...,p[n],w[m+1],...,w[n],M-CurW)
if B <= OptP
    return
if CurW + w[m+1] <= M
     C[m] = {0,1}
else
     C[m] = {0}

for x[m+1] in C[m]
    BoundedKnapsack(m+1,CurW+w[m+1]x[m+1], CurP+p[m+1]x[m+1])
```

n	Algorithm 4.1	Algorithm 4.3	Algorithm 4.9
8	511	332	52
8	511	312	78
8	511	333	72
8	511	321	74
8	511	313	57
12	8191	4598	109
12	8191	4737	93
12	8191	5079	164
12	8191	4988	195
12	8191	4620	87
16	131071	73639	192
16	131071	72302	58
16	131071	76512	168
16	131071	78716	601
16	131071	78510	392
20	2097151	1173522	299
20	2097151	1164523	104
20	2097151	1257745	416
20	2097151	1152046	118
20	2097151	1166086	480
24	33554431	19491410	693
24	33554431	18953093	180
24	33554431	17853054	278
24	33554431	19814875	559
24	33554431	18705548	755

Figure 1: From Kreher and Stinson p127

This algorithm is a substantial practical improvement over what we had before. Figure 1 shows some experimental data from Kreher and Stinson. Algorithm 4.1 is the naive Knapsack algorithm. Algorithm 4.3 is our first attempt at pruning (from last lecture). Algorithm 4.9 is the algorithm we just developed. The instances were generated by, for each i, randomly selecting w_i an integer between 0 and $1\ 000\ 000$ and then choosing $p_i = w_i\epsilon_i$ where ϵ_i is random in the interval $(0.9, 1.1)$, and

$$M = \frac{1}{2}\sum_{i=0}^{n-1} w_i$$

These choices were made to generate instances which are hard for the algorithm to solve.

3 Traveling Salesman Problem

The Travelling Salesman Problem is another classic optimization problem.

3.1 Problem statement

Suppose there are n cities and different costs to fly between them. You want to begin at your current city, visit all the other cities in some order and return home. Which order minimizes the cost?

We can assume that direct flights exist between any two cities (symmetry), by specifying an extremely high cost for flights that do not exist. We assume here that the cost of a flight between two cities does not depend on time or direction of travel. The formal problem statement is as follows:

A Hamilton cycle in a graph $G = (V, E)$ is a cycle $X = x_1e_1x_2e_2 \dots e_{n-1}x_ne_nx_1$ in which every vertex x_i in V appears exactly once, and each e_i is an edge connecting x_i to x_{i+1} (where $x_{n+1} = x_1$). We sometimes refer to X by the permutation $[x_1\, x_2\, \dots\, x_n]$, or by its edge set $\{e_1, e_2, \dots e_n\} \subseteq E$.

The (symmetric) Traveling Salesman Problem: Given a complete graph $G = (V, E)$ with $V = \{1, \dots, n\}$ and a cost function

$$\text{cost} : E \to \mathbb{Z}_{>0},$$

find a Hamiltonian cycle X of G which minimizes

$$\text{cost}(X) = \sum_{e \in X} \text{cost}(e),$$

Without loss of generality we can assume that cycles begin and end at $x_1 = 1$ (the "home city"). We'll represent each Hamiltonian cycle as a permutation $X = [x_1\, x_2\, \dots\, x_n]$. We store the costs of travel by a symmetric $n \times n$ matrix M where $\text{cost}(ij) = M_{i,j} = M_{j,i}$ and $M_{i,i} = \infty$, so

$$\text{cost}(X) = M_{x_n,x_1} + \sum_{i=1}^{n-1} M_{x_i,x_{i+1}}.$$

3.2 Naive backtracking solution

We can use a search tree to generate every permutation of V, where each partial solution is a list of vertices representing a path in G beginning at $x_1 = 1$. We will generate every Hamilton cycle twice, once in each direction. We begin the search by the procedure call `NaiveTravelingSalesman(1)`.

Algorithm: NaiveTravelingSalesman

```
global: cost(), OptC, OptX, C
        X=[x[1],...,x[n]]    // [x[1],...,x[m]] is a path from x[1] to x[m] in G
input: m
if m=1
    x[1] := 1
    C[1] := {2,3,...,n-1} // Omit the choice x[2]=n; cycles can be traversed in reverse
    OptC := infinity
else if m=n
    Cost := cost(X)
    if Cost < OptC
        OptC := Cost
        OptX := X
C[m] := C[m-1]-{x[m]}
for x[m+1] in C[m]
    NaiveTravelingSalesman(m+1)
```

3.3 Min cost bound

Now lets do better with a bounding function. In this problem we are minimizing cost rather than maximizing profit so the bounding function must be a lower bound on the cost.

Definition. Given a graph G and edge costs (M_{ij}), and given $x \in V$ and $W \subseteq V \setminus \{x\}$ where $W \neq \emptyset$ define

$$b(x, W) = \min\{M_{x,y} : y \in W\}.$$

Proposition. *Let $G = (V, E)$ be a graph with $V = \{1, 2, \ldots, n\}$ and edge costs $(M_{i,j})$. Let $X = [x_1, \ldots, x_m]$ be a partial solution for some $m < n$ (so X lists the vertices of a path in G). Let $X' = [x_1, \ldots, x_m, x_{m+1}, \ldots, x_n]$ be a minimum cost Hamiltonian cycle which extends X. Then*

$$\mathit{cost}(X') \geq \sum_{i=1}^{m-1} M_{x_i,x_{i+1}} + b(x_m, Y) + \sum_{y \in Y} b(y, Y \setminus \{y\} \cup \{x_0\})$$

where $Y = V \setminus \{x_1, \ldots, x_m\}$

Proof. For convenience let $x_{n+1} = x_1$. Let $i \in \{m, m+1, \ldots, n\}$. Since $m < n$, we have

$$x_{i+1} \in \begin{cases} Y & \text{if } i = m \\ Y \cup \{x_1\} \setminus \{x_i\} & \text{if } m+1 \leq i \leq n. \end{cases}.$$

so

$$M_{x_i,x_{i+1}} \geq \begin{cases} b(x_m, Y) & \text{if } i = m \\ b(x_i, Y \cup \{x_1\} \setminus \{x_i\}) & \text{if } m+1 \leq i \leq n. \end{cases}.$$

Now

$$\text{cost}(X') = \sum_{i=1}^{m-1} M_{x_i,x_{i+1}} + M_{x_m,x_{m+1}} + \sum_{i=m+1}^{n} M_{x_i,x_{i+1}} \tag{1}$$

$$\geq \sum_{i=1}^{m-1} M_{x_i,x_{i+1}} + b(x_m, Y) + \sum_{y \in Y} b(y, Y \cup \{x_1\} \setminus \{y\}) \tag{2}$$

□

The expression (1), called the min cost bound for the partial solution $X = [x_1, \ldots, x_m]$, can be coded into a procedure we call `MinCostBound(X,m)` (not shown). It gives an algorithm which is invoked with `MinCostBoundTravewrlingSalesman(1)`.

Algorithm: MinCostBoundTravelingSalesman

```
input: m (in {1,2,...,n})
global: cost(), X, OptC, OptX, C, MinCostBound()
if m=1
    x[1] := 1
    C[1] := {2,3,...,n-1}
    OptC := infinity
else if m=n
    Cost := cost(X)
    if Cost < OptC
        OptC = Cost
        OptX = X
C[m] := C[m-1]-{x[m]}

if MinCostBound(X,m) >= OptC
    return

for x[m+1] in C[m]
    MinCostBoundTravelingSalesman(m+1)
```