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.

The point is that if we are at vertex X in a backtracking algorithm and we find that $P(X) \leq$ 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 hence can be pruned away. A good bounding function is

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

These two features must be balanced.

This gives us a meta algorithm

Meta algorithm: BoundedBacktrack

```
global: X, OptP, OptX, C
input: l
if [x(0),...,x(l-1)] is feasible
   P = profit([x(0),...,x(l-1)]
   if P > OptP
```

```
        OptP = P
        OptX = [x(0),...,x(l-1)]
Compute C(l)
B = B([x(0),...,x(l-1)])
if B <= OptP
    return
for x(l) in C(l)
    BoundedBacktrack(l+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: Suppose you are given profits $p_0, \ldots, p_{n-1}$, weights $w_0, \ldots, w_{n-1}$, and capacity M as in the Knapsack problem; find the maximum value of

$$\sum_{i=0}^{n-1} p_i x_i$$

subject to

$$\sum_{i=0}^{n-1} w_i x_i \leq M$$

with the x_i rational numbers $0 \leq x_i \leq 1$. This is called the Rational Knapsack Problem. The key difference is that the x_i can be rational numbers rather than just 0 or 1. This change makes a big difference in how easily the problem can be solved. Now we can just take a greedy approach.

Algorithm: RationalKnapsack

```
input: p(0),...,p(n-1),w(0),...w(n-1),M

permute the indices so p(0)/w(0) >= p(1)/w(1) >= ... >= p(n-1)/w(n-1)
i=0
P=0
W=0
X=[0,0,...,0] (length n, indexed from 0 to n-1)
while W<M and i<n
    if W+w(i) <= M
        x(i)=1
        W = W+w(i)
        P = P+p(i)
        i = i+1
    else
        x(i) = (M-W)/w(i)
        W = M
        P = P+x(i)p(i)
        i = i+1
return P
```

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 $X = [x_0, \ldots, x_{l-1}]$ to the Knapsack problem, define

$$B(X) = \sum_{i=0}^{l-1} p_i x_i + \text{RationalKnapsack}(p_l, p_{l+1}, \ldots, p_{n-1}, w_l, w_{l+1}, \ldots, w_{n-1}, M - \sum_{i=0}^{l-1} w_i x_i)$$

The first sum captures the profit of the partial solution. If x_i in the rest were each 0 or 1 then we'd have the profit of a descendant of X. Allowing rational values may make a larger profit possible but can't make the profit worse, so

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

so B defines a bounding function. B is fast to compute since the rational knapsack algorithm is fast.

This gives the following algorithm

Algorithm: BoundedKnapsack

```
global: X, OptX, OptP, C
assume the p(i), w(i) are ordered so p(0)/w(0) >= p(1)/w(1) >= ... >= p(n-1)/w(n-1)

input: l, CurW
if l=n
    if sum(p(i)x(i), i=0..n-1) > OptP
        OptP = sum(p(i)x(i), i=0..n-1)
        OptX = [x(0),...,x(n-1)]
    return
else
    if CurW + w(l) <= M
        C(l) = {0,1}
    else
        C(l) = {0}
B = sum(p(i)x(i), i=0..l-1) + RationalKnapsack(p(l),...,p(n-1),w(l),...,w(l-1),M-CurW)
if B <= OptP
    return
for x(l) in C(l)
    BoundedKnapsack(l+1,CurW+w(l)x(l))
```

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 $p_i = 2w_i\epsilon_i$ 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 Travel-ling 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 and return home. Which order minimizes the cost.

We might as well assume that travel between any two cities is possible (though perhaps at high cost), so the formal problem statement is as follows:

TABLE 4.2
Size of state space trees for Algorithms 4.1, 4.3, and 4.9, on random instances with n weights

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

Given a complete graph $G = (V, E)$ with $V = \{1, \ldots, n\}$ and a cost function

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

Find a Hamiltonian cycle X of G such that

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

is minimized.

3.2 Naive backtracking solution

Without loss of generality we can take our cycles beginning at 0. We still get each cycle twice, once going around one way and once the other way. We'll represent the Hamiltonian cycles, and the partial solutions, as lists of vertices.

Algorithm: NaiveTravelingSalesman

```
global: X, OptC, OptX, C
input: l
if l=n
    C = cost([x(0),...,x(n-1)] (as a cycle)
    if C < OptC
        OptC = C
        OptX = [x(0),...,x(n-1)]
    return
if l=0
    C(l)={0}
if l=1
    C(l)={1,...,n-1}
else
```

```
    C(l)=C(l-1)-{x(l-1)}
for x(l) in C(l)
    NaiveTravelingSalesman(l+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 and edge costs as above, and given $x \in V$, $W \subseteq V$, $W \neq \emptyset$ define

$$b(x, W) = \min\{\text{cost}(x, y) : y \in W\}$$

Proposition. *Given a graph and edge costs as above. Let* $X' = [x(0), \ldots, x(n-1)]$ *be the ninimim cost Hamiltonian cycle which extends* $[x(0), \ldots, x(l-1)]$ *(with* $l < n$*). Then*

$$\textit{cost}(X') \geq \sum_{i=0}^{l-2} \textit{cost}(x_i, x_{i+1}) + b(x_{l-1}, Y) + \sum_{y \in Y} b(y, Y \cup \{x_0\})$$

where $Y = V \setminus \{x_0, \ldots, x_{l-1}\}$

Proof. For convenience let $x_n = x_0$. Then

$$\begin{aligned}
\text{cost}(X') &= \sum_{i=0}^{n-1} \text{cost}(x_i, x_{i+1}) \\
&= \underbrace{\sum_{i=0}^{l-2} \text{cost}(x_i, x_{i+1})}_{\substack{\text{cost of part} \\ \text{already chosen}}} + \underbrace{\text{cost}(x_{l-1}, x_l)}_{\substack{l<n \text{ so} \\ x_l \in Y}} + \underbrace{\sum_{i=l}^{n-1} \text{cost}(x_i, x_{i+1})}_{\{x_l, \ldots, x_{n-1}\} = Y} \\
&\geq \sum_{i=0}^{l-2} \text{cost}(x_i, x_{i+1}) + b(x_{l-1}, Y) + \sum_{y \in Y} b(y, Y \cup \{x_0\})
\end{aligned}$$

□

This bounding function is called the min cost bound. It gives an algorithm

Algorithm: MinCostBoundTravelingSalesman

```
global: X, OptC, OptX, C
input: l
if l=n
    C = cost([x(0),...,x(n-1)] (as a cycle)
    if C < OptC
        OptC = C
        OptX = [x(0),...,x(n-1)]
    return
if l=0
    C(l)={0}
else
    if l=1
        C(l)={1,...,n-1}
    else
        C(l)=C(l-1)-{x(l-1)}
    B = MinCostBound(x(0),...,x(l-1))
    if B >= OptC
        return
for x(l) in C(l)
    MinCostBoundTravelingSalesman(l+1)
```