Better bounding for TSP

Contents

1 Recall

2 Even better bounding functions

2.1 A better bounding function for the Traveling salesman problem

In Lecture 20 we showed how to prune a backtrack search on an instance (G, M) of the Traveling Salesman problem, where $M = (M_{i,j})$ is a matrix of non-negative edge costs and $M_{i,j} = \infty$ if $\{i, j\}$ is not an edge of G. We used the bounding function defined for each partial solution $X = [x_1, \dots, x_m]$, $1 \le m \le n$ by

$$\text{MinCostBound}(X) = \sum_{i=1}^{m-1} M_{x_i,x_{i+1}} + b(x_m, V \setminus \{x_1, \dots, x_m\}) + \sum_{y \in V \setminus \{x_1,\dots,x_m\}} b(y, V \setminus \{x_2, \dots, x_m, y\})$$

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

Here we present an even better bounding function for TSP. (See Donald Kreher and Douglas Stinson, *Combinatorial Algorithms*, sections 4.6 and 4.7 for a reference on this material) The bounding function relies on an algorithm `Val` which inputs an $k \times k$ submatrix M' of M. It subtracts from each entry of M' the smallest element in its row to get matrix M'', Then it finds the smallest element in each column of M''. The output is the sum of all $2k$ of these smallest elements.

Algorithm: Val

```
input: M’  (k by k matrix)
local: M’’  (k by k matrix), rowmin, colmin, val, i, j
val = - infinity
for i from 1 to k
    rowmin := M’(i,1)
    for j from 2 to k
        if M’(i,j) < rowmin
            rowmin := M’(i,j)
    for j from 1 to k
        M’’(i,j) := M’(i,j) - rowmin
    val = val + rowmin
for j from 1 to k
    colmin = M’’(1,j)
    for i from 2 to k
        if M’’(i,j) < colmin
            colmin := M’’(i,j)
    val := val +colmin
return val
```

Definition. The value of an $k \times k$ submatrix M' of M is the output of $\mathrm{val}(M')$. That is,

$$\mathrm{val}(M') = \sum_{i=1}^{k} r_i + \sum_{j=1}^{k} c_j \tag{1}$$

where, for $1 \le i, j \le m$,

$$r_i = \min\{M'_{i,t} \mid 1 \le t \le k\} \quad \text{and} \quad c_j = \min\{M'_{t,j} - r_t \mid 1 \le t \le k\}. \tag{2}$$

Example. Suppose the rows and columns of M' are both indexed by vertices $V = 1, 2, 3, 4$ and the edge costs are given in the following table

edge	cost
12	3
13	5
14	8
23	2
24	7
34	8

. Then $M' = \begin{bmatrix} \infty & 3 & 5 & 8 \\ 3 & \infty & 2 & 7 \\ 5 & 2 & \infty & 6 \\ 8 & 7 & 6 & \infty \end{bmatrix}$. The row minima are 3, 2, 2, and 6. This leaves the reduced matrix

$$M'' = \begin{bmatrix} \infty & 0 & 2 & 5 \\ 1 & \infty & 0 & 5 \\ 3 & 0 & \infty & 4 \\ 2 & 1 & 0 & \infty \end{bmatrix}$$

The column minima of M'' are 1, 0, 0 and 4. The output is $\mathrm{val}(M') = (3+2+2+6) + (1+0+0+4) = 18$.

Using `Val()`, we may derive a bounding function for the Traveling salesman problem. Recall that the cost of a Hamiltonian cycle is the sum of the costs of the edges in the cycle.

Proposition. *Let (G, M) an instance of the Traveling salesman problem with vertex set $V = \{1, 2, \dots, n\}$. Let $X = [x_1, x_2, \dots, x_m]$ be any partial solution (a node) in the backtrack search tree. Let $\mathrm{cost}(X) = \sum_{i=1}^{m-1} M_{x_k i x_{i+1}}$ be the cost of the path in G represented by X. Let $Y = V - \{x_1, x_2, \dots, x_m\}$. Let M' be the submatrix of M obtained by selecting the rows in $\{x_m\} \cup Y$ and selecting the columns in $Y \cup \{x_1\}$. Then for any Hamiltonian cycle $X' = [x_1, x_2, \dots, x_m, x_{m+1}, \dots, x_n]$ which is a descendant of X we have*

$$\mathrm{cost}(X') \ge \mathrm{cost}(X) + \mathrm{val}(M')$$

Proof. Note that M' is a $n - m + 1$ by $n - m + 1$ matrix. Let us define $x_{n+1} = x_1$. Since $Y = \{x_{m+1}, x_{m+1}, \dots, x_n\}$, we may rewrite the definition of $\mathrm{val}(M')$ (equations (1) and (2) above) in terms of the matrix M.

$$\mathrm{val}(M') = \sum_{i=m}^{n} R_i + \sum_{j=m+1}^{n+1} C_j$$

where, for $m \le i \le n$ and $m + 1 \le j \le n + 1$,

$$R_i = \min\{M_{x_i,x_t} \mid m+1 \le t \le n+1\} \quad \text{and} \quad C_j = \min\{M_{x_t,x_j} - R_t \mid m \le t \le n\}.$$

We have

$$\begin{aligned}
\text{cost}(X') &= \text{cost}(X) + \sum_{i=m}^{n} M_{x_i,x_{i+1}} \\
&= \text{cost}(X) + \sum_{i=m}^{n} \big(R_i + (M_{x_i,x_{i+1}} - R_i) \big) \\
&\geq \text{cost}(X) + \sum_{i=m}^{n} \big(R_i + \min\{ (M_{x_t,x_{i+1}} - R_t) \mid m \leq t \leq n\} \big) \\
&= \text{cost}(X) + \sum_{i=m}^{n} (R_i + C_{i+1}) \\
&= \text{cost}(X) + \sum_{i=m}^{n} R_i + \sum_{j=m+1}^{n+1} C_j \\
&= \text{cost}(X) + \text{val}(M').
\end{aligned}$$

□

In other words, the function $B(X) = \text{cost}(X) + \text{val}(M')$ described above is a bounding function for the Traveling Salesman problem backtrack search.

Example. Consider the Traveling Salesman problem with $V = \{1, 2, 3, 4\}$ and cost matrix $M = M'$ from the previous example. The cost of each Hamiltonian cycle which is a descendant of the node $X = [x_1]$ with $x_1 = 1$ is as follows.

Hamiltonian cycle X'	cost(X')
$[1, 2, 3, 4]$	3+2+6+8 = 19
$[1, 2, 4, 3]$	3+7+6+5 = 21
$[1, 3, 2, 4]$	5+2+7+8 = 22

Here we have $M' = M$, so the bounding function evaluates to

$$B(X) = \text{cost}(X) + \text{val}(M') = 0 + 18$$

Here the bound is not tight, but it is pretty good!

Let us write the bounding function based on `Val`, and then the backtrack algorithm `ReduceBoundTravelingSalesman` using it.

Algorithm: ReduceBound

```
input:  m, X = [ x[1], ..., x[m] ]
global: n, Val(), M (an n by n cost matrix)
local:  Y, M', u, v, i, j

//   Build submatrix M'
Y := {1,2,...,n} - {x[1], ... , x[m]}
i := 1
for u in {x[m]} union Y
     j := 1
     for v in Y union {x[1]}
          M'(i,j) := M(u,v)
          j := j+1
     i = i+1
B := Val(M')
for i from 1 to m-1
     B :=  B + M(x[i],x[i+1])
return B
```

Algorithm: ReduceBoundTravelingSalesman

```
input: m
global: X, OptC, OptX, C, ReduceBound()
        M (n by n matrix of costs)
if m=1
    x[1] := 1
    C[1] := {2,3,...,n-1}
    OptC := infinity
else if m=n
    Cost := add( M[ x[i], x[i+1] ], i=1,2,...,n-1 ) + M[ x[n],x[1] ]
    if Cost < OptC
        OptC := Cost
        OptX := X
C[m] := C[m-1]-{x[m]}

B := ReduceBound(m,X)
if B >= OptC
    return

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

2.2 Branch and Bound

As we noticed in the knapsack problem, the order in which we recurse on the elements of C_ℓ can make a big difference. We can use the bounding function to make a good choice. When we are at node X in the state space tree, calculate $B(X')$ for every child X' of X. For a maximizing problem (like the knapsack problem), make the recursive calls in *decreasing* order of $B(X')$. For a minimizing problem (like the traveling salesman problem), make the recursive calls in *increasing* order of $B(X')$. This way we are more likely to get better pruning.

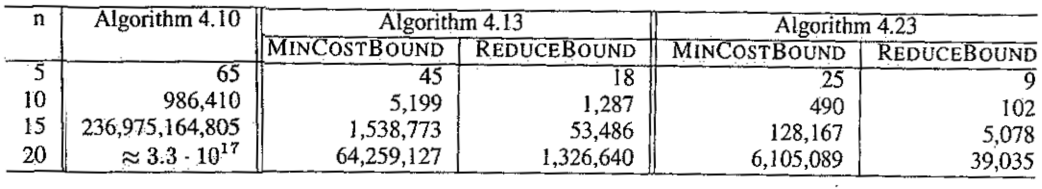
For the traveling salesman the algorithm will look as follows:

Algorithm: ReduceBoundTravelingSalesman

```
input: m
global: X, OptC, OptX, C, M (n by n matrix of costs), ReduceBound()
        B = [B[1], ..., B[n]] // B[m] is a list of lower bounds
                              // associated with the choice list C[m]
if m=1
    x[1] := 1
    C[1] := {2,3,...,n-1}
    OptC := infinity
else if m=n
    Cost := add( M[ x[i], x[i+1] ], i=1,2,...,n-1 ) + M[ x[n],x[1] ]
    if Cost < OptC
        OptC := Cost
        OptX := X
C[m] := C[m-1]-{x[m]}

B[m] := []
for x in C[m]
    B[m] := B[m], ReduceBound([x[1], ..., x[m], x]) // Append the bound to list B[m]
Sort B[m] into non-decreasing order B[m][1], B[m][2], ...
Sort C[m] into the corresponding order C[m][1], C[m][2], ...
for i from 1 to |C[m]|
    if B[m][i] < OptC
        x[m+1] := C[m][i]
        ReduceBranchBoundTravelingSalesman(m+1)
```

Kreher and Stinson ran the naive traveling salesman, the bounded traveling salesman with both the min cost bound and the reduce bound, and the branch and bound traveling salesman with both bounds, on random instances of the problem with edges costs random integers between 0 and 100. The results were (table from p134 and p143). Algorithm 4.10 is the naive backtrack, Algorithm 4.13 is the bounded backtrack and Algorithm 4.23 is the branch and bound algorithm. The values in the table are the sizes of the state space trees.



n	Algorithm 4.10	Algorithm 4.13		Algorithm 4.23	
		MINCOSTBOUND	REDUCEBOUND	MINCOSTBOUND	REDUCEBOUND
5	65	45	18	25	9
10	986,410	5,199	1,287	490	102
15	236,975,164,805	1,538,773	53,486	128,167	5,078
20	$\approx 3.3 \cdot 10^{17}$	64,259,127	1,326,640	6,105,089	39,035

3 Heuristic search

Sometimes even good bounding is too slow. How can we explore the state space faster, perhaps just finding a close to optimal solution rather than an optimal solution.

Definition.

- The universe $\mathcal{X}$ is a finite set of elements which are possible (not necessarily feasible) solutions
- $X \in \mathcal{X}$ is feasible if it satisfies the constraints of the problem
- $P(X)$ is the profit of X
- A neighbourhood function N is a function which maps each possible solution to a set of "nearby" solutions. That is,
$$N : \mathcal{X} \to 2^{\mathcal{X}}$$
where $2^{\mathcal{X}}$ is the set of all subsets of $\mathcal{X}$.
- Given N, a neighbourhood search is an algorithm, possible randomized, which takes a feasible solution $X \in \mathcal{X}$ and returns either a feasible solution $Y \in N(X) \setminus \{X\}$ or `fail`

The idea is that if we are at a feasible solution, then we consider its neighbourhood, and following the neighbourhood search algorithm we update X to Y (or we fail). We repeat this many times, with the hope of finding some close-to-optimal feasibile solutions along the way.

For the knapsack problem we could take $\mathcal{X} = \{0,1\}^n$ the set of all binary strings of length n. A reasonable neighbourhood function would be

$$N(X) = \{Y \in \mathcal{X} : d(X,Y) \leq c\}$$

for some fixed c, where $d(X,Y)$ is the Hamming distance between X and Y.

Possible neighbourhood search strategies include

- Find a feasible solution $Y \in N(X)$ such that $P(Y)$ is maximized. Fail if there are no feasible solutions in $N(X)$.
- Find a feasible solution $Y \in N(X)$ such that $P(Y)$ is maximized. If $P(Y) > P(X)$ return Y, otherwise fail.
- Choose a random solution in $N(X)$ return it if it is feasible, otherwise fail (or perhaps try a fixed number of times).
- Choose a random solution Y in $N(X)$ return it if $P(Y) > P(X)$, otherwise fail (or perhaps try a fixed number of times).

More sophisticated heuristic searches may use a combination of approaches.

3.1 Hill climbing

Hill climbing is the simplest heuristic search. Hill climbing is when, as in points 2 and 4 above, we always choose a solution which increases the profit. For many applications this "greedy approach" is too naive because it can get stuck in a local maximum which may be quite far from optimal. The basic shape of the algorithm is as follows

Algorithm: GenericHillClimbing

```
Select a feasible X in the universe
searching = true
while searching
    try to get a feasible solution Y in N(X) with P(Y)>P(X) (randomly or exhaustively)
    if Y = fail
        searching = false
    else
        X = Y
return X
```