

Notes on Branch and Bound

March 20, 2006

1 Branch and Bound

These notes follow the discussion of branch and bound algorithms in Computer Algorithms by E. Horowitz, S. Sahni and S. Rajasekaran.

These notes describe a mathematical model of the process of choosing the next node to expand. This model also includes a function for deciding when to abandon a search path before reaching the end of the path. Abandoning searches early attempts to minimize computational effort to find the minimal solution.

The basis of branch and bound algorithms is a ranking (or bound) function. The ranking function assigns a value to each node in the search graph. At each step, a branch and bound algorithm uses the ranking function, with the help of a priority queue, to decide which node to expand next. In contrast, the usual DFS and BFS exploration algorithms perform a blind search of the graph, guided only by the feasibility (or “partial criterion”) function.

Ideally, the ranking function, $\hat{c}(x)$, ranks nodes based on the cost of a minimal solution reachable from node x . The problem with this ranking function is that the minimal solution to be reached must be known ahead of time.

Instead, the ranking function uses an estimate, $\hat{g}(x)$, of the cost of a minimal solution reachable from node x . Using $\hat{g}(x)$ to rank nodes may require exploring unnecessary nodes in the graph—particularly if $\hat{g}(x)$ is not a good estimate. We saw the results of an imprecise estimate in the job assignment example covered in Lecture 25. In this example, the entire subtree for the assignment $a : 2$ and $b : 3$ is explored before the minimal solution is found in a different subtree. In this example, searching the $a : 2, b : 3$ subtree was fairly inexpensive; however, in other cases, this search may be costly.

Another element of the ranking function measures the cost of reaching a node from the root. As the search gets farther from the root node, the node falls in the ranking. The function $h(x)$ measures the cost of reaching node x from the root node. The final element is a function f that determines how much significance to give the cost of reaching node x .

After including $f()$ as a function of $h(x)$, the ranking function $\hat{c}(x)$ is

$$\hat{c}(x) = f(h(x)) + \hat{g}(x)$$

If $f(h(x)) \equiv 0$ then the algorithm makes long, deep searches of the graph. If $f(h(x)) > 0$ then the algorithm considers nodes close to the root before making long potentially fruitless forays into the graph.

A node is *live* if its subgraph might contain a minimal solution. Live nodes are potential candidates for exploration. A node is *dead* if its subgraph can not contain a minimal solution. If the estimated cost of a minimal solution reachable from x is less than the actual minimal solution reachable from x then a node can be killed when the estimated cost is greater than the least upper bound (aka, BSSF = “best solution so far”) on a solution. In symbols, if $c(x) \geq \hat{c}(x) \geq BSSF$ then an optimal solution is not reachable from x (assuming $BSSF$ is the least upper bound on a solution).

The job scheduling example in Lecture 25 used a least upper bound to kill nodes before searching their subtrees.

It should come as no surprise that tuning \hat{c} by adjusting f and \hat{g} requires empirical studies of representative data sets.

2 B and B for Traveling Salesperson Problem

Two branch and bound solutions to the TSP are described in this section. Each solution estimates the cost of a minimal tour using a *reduced cost matrix*. In the first solution, the nodes in a path through the search tree represent cities on a tour. In the second solution, nodes in the search tree represent the inclusion or exclusion of an edge between two cities in the optimal tour.

2.1 First Solution: Cities on a Tour

Suppose the distances between n cities are given in an $n \times n$ cost matrix (i.e., an adjacency matrix where entries represent edge cost). A row (or column) in this matrix is reduced if at least one entry in the row is 0 while the others are all positive. A matrix is reduced if all of its rows and columns are reduced. An example of constructing a reduced cost matrix is given in lecture 27.

The cost of a tour including a trip from city R to city S can be estimated by a reduced cost matrix A . Matrix A is constructed by:

1. set all entries of row R and column S to ∞ to avoid making a second visit from R or to S in an optimal tour.
2. set $A(S, 1)$ to ∞ to prevent making a premature trip from S to 1.
3. for each row or column that does not contain all ∞ , subtract each entry in each row by the smallest value in each row. Subtract each entry in each column by the smallest value in each column.

The estimated length of the shortest tour including a trip from R to S is given by

$$\hat{c}(S) = f(h(R) + 1) + \hat{c}(R) + A(R, S) + r$$

where r is the total value subtracted in step 3. The optimal values of f and h are determined empirically.

Lecture 27 includes an example of solving the TSP using the above cost estimate and tree construction.

2.2 Inclusion/Exclusion of Edges

In the first solution, trees were built by deciding which node to visit next. For example, the nodes in the i th level of the graph are the options for the i th city in a tour. A second solution can be built by deciding whether or not to include a trip between a particular pair of cities in each step. In this solution, the root node might represent the decision to include or exclude a trip between cities 6 and 8, for example. The left child represents the inclusion this trip and the right child represents the exclusion of this trip. The same distance cost estimate function \hat{c} is used in this solution. We explore this solution in Lecture 28.

An efficient implementation of this algorithm requires a policy for choosing which trip between cities to consider next in the search. A good policy is to choose cities R, S such that a trip between R and S has a high probability of being included in the optimal tour. One way of choosing R and S is to choose cities which result in the highest approximate cost for *not* including a trip between R and S in the tour. An example of this algorithm is given in Lecture 29.