CS 312: Algorithm Analysis

Traveling Salesperson Problem with Branch & Bound

Overview:

In this project, you will implement a branch and bound algorithm to find solutions to the traveling salesperson problem (TSP).

Objectives:

- To implement a branch and bound algorithm for finding solutions to the TSP
- To solve an NP-complete problem
- To further develop your ability to conduct empirical analysis

Background:

The TSP problem consists of the following:

Given: a directed graph with a cost associated with each edge.
Return: the lowest cost complete simple tour of the graph.

A complete simple tour is a path through the graph that visits every node in the graph exactly once and ends at the starting point, also known as a Hamiltonian cycle or Rudrata cycle. Note that as formulated here, the TSP problem is an optimization problem in so far as we are searching for the simple tour with minimum cost.

We branch and bound, as well as B&B solutions to the TSP, in great detail in lectures 32, 33, and 34. See the class schedule for links to the lectures and some short lecture notes (in the reading column) for guidance. Additionally, we offer the following discussion of one promising approach to generating child states (the “include-exclude” approach).

Bounding Function

Suppose we are given the following instance of the traveling salesperson problem for four cities in which the symbol "i" represents infinity.
One important element of a branch and bound solution to the problem is to define a bounding function. Our bounding function requires that we find a reduced cost matrix. The reduced cost matrix gives the additional cost of including an edge in the tour relative to a lower bound. The lower bound is computed by taking the sum of the cheapest way to leave each city plus any additional cost to enter each city. This bounding function is a lower bound because any tour must leave and enter each city exactly once, but choosing such edges may not define a solution, as we saw in class.

First, let's reduce row 1. The smallest entry in row 1 is the cheapest way to leave city A. A row is reduced by taking the smallest entry in the row, 3 in this case, and subtracting it from every other entry in the row. The smallest entry, 3 in this case, is also added to the lower bound. After reducing row 1, we have a bound of 3 and the following matrix:

\[
\begin{bmatrix}
1 & 2 & 1 & 0 \\
3 & 1 & 8 & 2 \\
5 & 3 & 1 & 9 \\
6 & 4 & 3 & 1 \\
\end{bmatrix}
\]

Next, we reduce row 2 by taking the smallest entry in row 2, 2 in this case, and subtracting 2 from each entry in row 2. We add 2 to the bound and obtain the following matrix:

\[
\begin{bmatrix}
1 & 2 & 1 & 0 \\
1 & 1 & 6 & 0 \\
5 & 3 & 1 & 9 \\
6 & 4 & 3 & 1 \\
\end{bmatrix}
\]

The remaining two rows are reduced in similar fashion. Lowest value 3 is subtracted from row 3, and 3 is likewise subtracted from row 4. The final bound is \(3 + 2 + 3 + 3 = 11\), and the reduced matrix so far is:

\[
\begin{bmatrix}
1 & 2 & 1 & 0 \\
1 & 1 & 6 & 0 \\
2 & 0 & 1 & 6 \\
3 & 1 & 0 & 1 \\
\end{bmatrix}
\]

Reducing the rows only accounts for the cheapest way to leave every city. Reducing the columns includes the cheapest way to enter every city. Column reduction is similar to row reduction. A column is reduced by finding the smallest entry in a column of the reduced cost matrix, subtracting that entry from every other entry in the column and adding the entry to the bound.

The smallest entry in the first column is 1 so we subtract 1 from each entry in column 1 and add 1 to the bound. The new bound is \(11 + 1 = 12\) and the new matrix is:
Include-Exclude Approach

Another important element of a branch and bound solution is to define the manner in which children states are expanded from a given state. In the “include-exclude” approach, we generate two children for every parent: the left child represents the inclusion of an edge and the right child represents the exclusion of an edge. The next step is to decide which edge to include or exclude. We'll assume that we want to
1. minimize the bound on the left (include) child
and
2. maximize the bound on the right (exclude) child.
Choosing an edge so as to maximize a bound on the right child can lead to more aggressive pruning; consequently, this can be a compelling approach to finding a solution.

We observe that it is advisable to avoid including edges that are non-zero in the reduced matrix. If a non-zero edge in the reduced matrix is included, then the extra cost of that edge (as contained in the reduced matrix) must be added to the bound on the left side. However, we are trying to minimize the bound on the left side.

Next, get the bounds of including or excluding an edge. In the reduced matrix above, there are 5 entries that contain 0. We'll compute a pair of bounds (one for include and one for exclude) for each 0-residual-cost edge and pick the one that has the maximum right child bound and the minimum left child bound.

Start with the 0 at entry (2,1). If the edge from city 2 to 1 is included in the solution, then the rest of row 2 and column 1 can be deleted since we will leave city 2 once and enter city 1 once. We get this matrix:

\[
\begin{array}{cccc}
1 & 2 & 1 & 0 \\
0 & 1 & 6 & 0 \\
1 & 0 & 1 & 6 \\
2 & 1 & 0 & 1 \\
\end{array}
\]

This matrix must be reduced. The cost incurred during the reduction is added to the bound on the left child. In this case, no rows or columns need to be reduced. So the bound on the left child is 12 (which is the bound on the parent).

Now for the right child. If the edge between 2 and 1 is excluded, then we just replace entry 2,1 in the matrix with an infinity. We now have:

\[
\begin{array}{cccc}
1 & 2 & 1 & 0 \\
1 & 1 & i & 1 \\
1 & 0 & 1 & 6 \\
1 & 1 & 0 & 1 \\
\end{array}
\]
This matrix must be reduced and the bound increased by the amount reduced. Only column 1 must be reduced, and it is reduced by subtracting 1 from each entry. The bound on the right child is then 12 + 1 = 13.

Stepping back for a minute, we have now determined that the bound on including edge 2,1 is 12 and the bound on excluding edge 2,1 is 13. Can we do better using a different edge? We'll answer that question by examining all of the other 0's in the matrix.

The easy way to examine the 0's is the following. To include an edge at row i column j, look at all of the 0s in row i. If column x of row i contains a 0, look at all of the entries in column x. If the 0 in row i of column x is the only 0 in column x, then replacing row i with infinities will force a reduction in column x. So add the smallest entry in column x to the bound on including the edge at row i and column j. Perform a similar analysis for the zeros in column j. This is the bound on including an edge.

To examine the bound on excluding an edge at row i and column j, add the smallest entries in row i and column j.

A complete examination of the 0 entries in the matrix reveals that the 0s at entries (3,2) and (4,3) give the greatest right-bound, 12+2, with the least left-bound, 12. You should verify this on your own.

So we'll split on either (3,2) or (4,3); it doesn't matter which.

After deciding which edge to split on, the next step is to do the split. Doing the split generates two new reduced matrices and bounds. These are then inserted into the priority queue.

Following the branch-and-bound algorithm from the lectures, the next step is to dequeue the most promising node and repeat the process. This continues until a solution is found. You know you've found a solution when you've included enough edges. When a solution is found, check to see if that solution improves the previous best solution (so far). If so, the new solution is the best solution so far. If the new solution is now the best solution so far, then the priority queue may be trimmed to avoid keeping unpromising states around. We iterate until the queue is empty or until time is exhausted.

Another important aspect of the algorithm is preventing early cycles and keeping track of the best solution so far. They are related. As you add edges to a partial solution (state), you'll need to keep track of which edges are part of the solution. Since a simple tour of the graph can't visit the same city twice, you'll need to delete edges from the state's residual cost matrix that might result in a city being visited twice. To do this, you'll need to know which cities have been included. The cities included in a partial solution will need to be stored (along with the matrix and bound) in each state in the priority queue.
To Do:

1. Download the project code distribution from the link on the schedule. We include a GDI-based 2-dimensional viewer that you can use. The problems are generated randomly by clicking a button and you can control the problem size.

2. Code up a TSP solver that uses branch and bound, following the pattern given in class.

3. Your solver should include a time-out mechanism so that it will terminate and report the best solution so far after 20 seconds of execution time. (Note that we aren’t concerned that you use exactly 20 seconds. Running a timer and checking the time every iteration through your branch and bound algorithm is sufficient if slightly imprecise. You can use timers and interrupts if you want to be more precise).

4. For this project, the performance requirement will focus on space rather than time. The branch and bound algorithm uses a priority queue to store the list of nodes waiting to be expanded. One way to decrease the size of the priority queue is to prune the priority queue each time a new best solution is found. After a new best solution is found, the priority queue can be pruned so that it no longer contains nodes that cannot improve the new best solution so far.
   a. Your solver must include a priority queue management mechanism to optionally prune the queue during execution as outlined above.
   b. You will also need a way to report the maximum size of your priority queue and the number of nodes that were pruned during the search.

Report:

Write up a paragraph in which you describe your priority queue pruning algorithm. When does it prune? Which nodes get pruned? How expensive is the pruning operation?

Improvement Ideas:

Some possible improvements:

- Think of a better way to visualize the step by step performance of the algorithm. Perhaps show how the reduced cost matrix evolves during the search process. The visualization should do something for each state pulled from the priority queue.
- Implement a better ad hoc solution to compute your initial BSSF.
- Implement a better (and more expensive?) feasibility test to prune the space earlier.
- Implement a different search strategy (try the strategy that you didn’t try in the main project: include/exclude edge, all next edges, ...)
- Implement a randomized version of your algorithm.

Revised: 4/28/2008