CS 312

Traveling Sales Person with Branch & Bound

Overview:

In this project, you will implement a branch and bound algorithm to find solutions to the traveling sales person problem.

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. Note that as formulated here, the TSP problem is an optimization problem in so far as we are searching for the lowest cost complete simple tour.

We discuss several branch and bound solutions to the TSP in great detail in class. See the class schedule for links to the lectures and some short lecture notes. Additionally, we offer the following discussion of one approach:

Suppose you are given the following instance of the traveling salesperson problem for four cities in which the symbol "i" represents infinity.

```
i 5 4 3
3 i 8 2
5 3 i 9
6 4 3 i
```

The first step in a branch and bound solution is to find the 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 enter and leave each city. The lower bound is a lower bound because any tour must leave and enter each city exactly once.

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:

```
i 2 1 0
3 i 8 2
5 3 i 9
6 4 3 i
```

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:

```
i 2 1 0
1 i 6 0
5 3 i 9
6 4 3 i
```

The remaining two rows are reduced in similar fashion. 3 is subtracted from row three and 3 is subtracted from row 4. The final bound is 3 + 2 + 3 + 3 = 11 and the reduced matrix so far is:

```
i 2 1 0
1 i 6 0
2 0 i 6
3 1 0 i
```

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, 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:

```
i 2 1 0
0 i 6 0
1 0 i 6
2 1 0 i
```

The remaining columns are already reduced.

The next step is to decide which edge to include or exclude. We'll assume that you want to 1. minimize the left child and 2. maximize the right child. The left child represents the inclusion of an edge and the right child represents the exclusion of an edge.

The first decision to make is 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. We are trying to minimize the left side.

Next, find a way to 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 for each one and pick the entry that has the max. right child bound and min. 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:

```
i 2 1 0
i i i i
i 0 i 6
i 1 0 i
```

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). Not bad.

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:

```
i 2 1 0
i i 6 0
1 0 i 6
2 1 0 i
```

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 and edge at row i column j, look at all of the 0's in row i. If column x of row i contains a 0, look at all of the entires 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 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 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
0's 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) 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 a priority queue.

The next step is to pop the next-most promising node off the priority queue 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. If so, the new solution is the best solution. If the new solution is the current best, check the first node on the priority queue to see if a better solution can be found. If not, the algorithm is done. If the new solution is not an improvement over the previous best solution, continue with the next-best node (unless of course, the next-best node can't improve the previous best solution, in which case you are done).

Another important aspect of the algorithm is preventing early cycles and keeping track of the solution so far. They are related. As you add edges to a solution, 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 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 node in the priority queue.

To Do:

1. Download the project file distribution from the link on the schedule. We include a GDI-based 2-dimentional 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.

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. So 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 as well).

4. For this project, the performance experiment 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 can not improve the new best solution.

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 to collect data for the report.

5. Write a report as described below and turn in the report.

Report:

1. Describe your priority queue data structures. If you used the same data structure for both the pruning and non-pruning priority queues, then you only need to describe one data structure here. If you used some one else's priority queue, then you should describe their data structure.

2. Describe your priority queue pruning algorithm. When does it prune? Which nodes get pruned? How expensive is the pruning operation?

3. A table containing the following columns.

	Priority queue with Pruning				Priority queue without pruning		
Problem-cities	Running time (sec.)	Cost of best tour found (*=optimal)	Max.node count	Nodes Pruned	Running time (sec.)	Cost of best tour found (* = optimal)	Max. node count
1 - 20	4.2	156*	600	40	3.5	156*	620
2 - 20	2.8	58*	412	324	2.5	62	534
3 – 25	5.0	213	710	120	5.0	216	790

Note that the numbers in the above table are completely made up and may or may not have any correlation with reality.

4. Your table must include at least 10 rows of results for 10 different problems. Of those problems, at least 3 must run for 20 seconds (before timing out and returning the best solution found).

5. Make a clear concise statement about the effect of pruning the priority queue during branch and bound search. Your statement should include the impact on performance in time and space. Be careful to make a conclusion no stronger than your results.

6. Describe why you think you observed the behavior you summarized in the above statement.

Improvement:

Think of a way to improve your TSP B&B algorithm and implement your improvement. Determine if your improvement had the desired effect. Write a page or two (one page will certainly suffice) that

1. describes the purpose of your improvement
2. describes your improvement implementation
3. provides some empirical evidence to support a conclusion that your improvement either did or did not do what you thought it would do
4. explain the evidence in #3

Some possible improvements:

- Think of a better way to visualize the step by step performance of the algorithm. Maybe think of something that shows how the reduced cost matrix figures into the whole thing. The visualization should do something for each node 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 (include/exclude edge, all next edges, …)

Revised: 3/12/2007