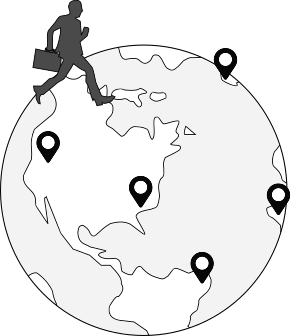
**Traveling Salesman**

**Branch and Cut Algorithm**



Sakibul Alam

*Emory ‘15*

*Rice ‘17*

Evan Toler

*Rice University ‘18*

Alexander Gardner

*Rice University ‘18*

Wendy Knight

*Rice University ‘18*

***Table of Contents***

1. *Introduction…………………………………………………………… 3*
2. *The Algorithm ……………………………………………………….. 4*
   1. *Minimum Cut*
   2. *Branch-and-Cut*
   3. *Upper Bound Heuristic*
3. *Computational Results ……………………………………………. 7*
4. *Concluding Remarks ……………………………………………… 8*
5. *Future Work ……………………………………………………….. 8*
6. *References ………………………………………………………. 10*

***Introduction***

The travelling salesman problem (TSP) is an extremely intuitive concept: visit every city in a list and return home in the most efficient way. Solving the TSP is pertinent because it outputs a shortest path that one should take when faced with this dilemma. The problem consists of a graph G = (V,E), where V corresponds to vertices that represent cities and E corresponds to the edges in the graph that represent paths. The goal of the problem is to begin at an origin city, find the shortest route to visit each city once, and then return back to the origin city. In general, the TSP is useful in transportation applications. For example, each edge will correlate to a specific cost that is trying to be minimized. This cost could be time, distance, or financial costs that it takes to travel from one city to another. In general, the travelling salesman problem will assist in scenarios such as mailmen scheduling routes, the President creating a campaign plan, or National Park enthusiasts in finding the quickest way to visit every park.

There are many different integer programming formulations to solve the traveling salesman problem, but specifically in this report, we will use the branch-and-cut technique to solve the “cut formulation” given below:

The TSP is constrained first by the fact that one must enter and exit each city once, and secondly, that there can only be one tour such that each city is visited. This concept of tour simply means that a person cannot teleport from one route to another without taking a path. If there are multiple tours in the graph, then we will denote each one as a subtour. Since we do not want subtours, we implement subtour elimination constraints into our formulation. Unfortunately, the number of subtour elimination constraints grows exponentially as cities and edges are added to the graph. Therefore, computing every subtour elimination constraint is complicated and inefficient.

Finding every subtour elimination constraint is computationally expensive--however, we can get the same optimal solution as the full problem without implementing every constraint. Our goal then becomes how to judiciously choose constraints such that we arrive at the true optimal solution. According to Cook et al. (2010), by using the branch-and-cut method, we will find cuts and add them to a relaxed LP formulation of TSP in place of the subtour elimination constraints.

***Algorithm***

In our branch-and-cut approach to the TSP, we initially solve an LP relaxation of the TSP without any subtour elimination constraints. After solving the relaxation, we add cuts to our feasible set that we find are necessary to remove subtours. Then, we re-solve the TSP with these added constraints, find an edge with a fractional decision variable, and branch on that variable. Then, we repeat until there is only one tour to get to each city. The general procedure for solving the TSP is:

1. Solve the LP relaxation of the TSP without subtour elimination constraints. In general, the optimal solution will not be integer-valued.
2. Find minimum cuts for every pair of nodes {s,t} in the graph, using as the weight for edge . Cuts with weight less than two will become subtour elimination constraints.
3. Add the cuts with weight less than two to the initial LP relaxation and re-solve it.
4. Choose a non-integer variable 0 < < 1 and branch from it. When we branch on a variable, we create two copies of the LP--one where and one where
5. In each branch, re-solve and return to step 2. Finish when is an integer solution with no subtours.

*Minimum Cut Algorithm*

A cut between two nodes s and t is a set of edges such that when F is removed from E, there is no path from node s to node t. The minimum cut between s and t is such a cut that has the smallest sum of edge costs for the edges in the cut. We can extend this concept to minimum cuts of an entire graph G. The minimum cut of G is the set of edges that, when removed from E, separate the nodes into two disconnected parts.

Recall the subtour elimination constraints from the full formulation:

The formulation requires that the minimum cut between two nodes s and t be at least 2 when we use x as the edge weights. If not, then there exists a subtour in the optimal solution to the reduced problem. Thus, we seek to find cuts with weight less than 2, and constrain that the sum of the decision variables corresponding to these edges be at least 2.

To find this minimum cut, we implement an algorithm outlined in Stoer and Wagner (1994). Given a graph G, edge weight w, and starting node a, we compute the minimum cut of G as follows:

MinimumCutPhase(G, w, a)

A ← {a}

While A add to A the most tightly connected vertex

Store the cut-of-the-phase and shrink G by merging the two vertices added last

MinimumCut(G, w, a)

While > 1

MinimumCutPhase(G, w, a)

If the cut-of-the-phase is lighter than the current minimum cut

Then store the cut-of-the-phase as the current minimum cut

Here, merging two vertices and means that the resulting graph has one node, call it , instead of nodes and , and the edge weight from to another node is the sum of the weight from to and the weight from to . Moreover, we define the most tightly connected vertex to A to be one for whom the sum of the weights of edges connecting to vertices in A is the maximum.

*Branch-and-Cut Concept*

As we traverse through the minimum cut algorithm, we mark any cuts with weights less than 2, and we add these subtour elimination constraints to the LP relaxation. After adding these cuts, the optimal solution of the TSP may have fractional optimal variables. Therefore, we apply the branch-and-cut technique to eventually force the variables to be strictly integer. The branching phase begins by forcing some variable to be integer by adding a bound or . Since the constraints of the LP relaxation imply that each variable will be between 0 and 1, branching on that variable forms two subcases--one where and one where . For each subcase, we construct a new LP relaxation. After resolving each LP relaxation, we get new optimal variables . As before, we now use this new graph to check for more valid cuts and generate subtour elimination constraints. After this process, we branch again on a different fractional variable. This process continues until the solution is infeasible, integer, or the optimal cost is less than the smallest previously calculated cost for an integer solution.



*Upper Bound Algorithm*

From basic linear and integer programming theory, we note that any feasible solution to a minimization problem is an upper bound on the optimal cost value. Thus, we can generate a naive feasible, integer solution at the very beginning of the main algorithm in order to compute an upper bound. With this bound, we no longer need to visit as many branches in the branching phase of our algorithm. Recall that one condition to terminate the branching phase is when a branch produces an optimal cost that is lower than the best cost for a previously found integer solution. Similarly, any branch that produces an optimal cost larger than our upper bound can never lead to the true optimal solution, and therefore it is not worth analyzing that branch.

To compute an integer feasible solution, we start at an arbitrary node, call it node 1, and add the edge incident to node 1 with the least edge weight. This connects node 1 to the next node, call it node 2. Then, we add an edge that connects node 2 to a node not already in the tour. From the valid edges, we add one with the least edge weight. We repeat this process until the last step, when the final edge is added between the final node and the starting node. This nearest neighbor solution is clearly integer feasible, and thus provides the upper bound that we are looking for.

***Computational Results***

*Computer Type/Code Language*

We implemented this algorithm in Python using Gurobi on a Macbook Pro with 2.6GHz i5, 8GB RAM @ 1600 MHz DDR3. The results and run times for a few test instances of the TSP are given in the following table:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Test Case | **Att48** | **Berlin52** | **Gr21** | **Hk48** | **St70** | **Ulysses22** |
| Number of Cities | 48 | 52 | 21 | 48 | 70 | 22 |
| Number of Edges | 1128 | 1326 | 210 | 1128 | 2415 | 231 |
| Optimal Distance | 10628 | 7542 | 2707 | 11461 | 675 | 7013 |
| Run Time (s) | 16.86714 | 1.798841 | 0.020761 | 11.11439 | 1598.799 | 0.148680 |
| NN Run Time (s) | 16.68480 | 1.763470 | 0.020891 | 10.93244 | 1589.454 | 0.142128 |

For every problem instance with less than 50 nodes, our algorithm returned the optimal solution in under 20 seconds. However, the larger examples like St70 took about 30 minutes to execute. As we discussed earlier, the number of subtour elimination constraints increases exponentially as the number of edges increases. Likewise, we notice that the runtime increases exponentially with the number of edges. As a result, large TSP instances will still require ample time to solve with our algorithm.

The Gr21 instance was by far the fastest to solve out of all our examples. Upon examining the output of our algorithm, we found that we achieved the optimal solution after just one iteration of the minimum cut phase. That is, no branching was ever necessary. This may indicate that the main factor slowing down our algorithm is the branching phase. This confirms our intuition that the branching algorithm is computationally expensive in terms of memory allocation, since there is usually a large queue of linear programming models to keep track of at any time.

The final row of the above table indicates the run time with the use of the nearest neighbor upper bound. It improves the run time for most instances by a small amount, but there are probably better upper bounds that we could take advantage of to further decrease our runtime.

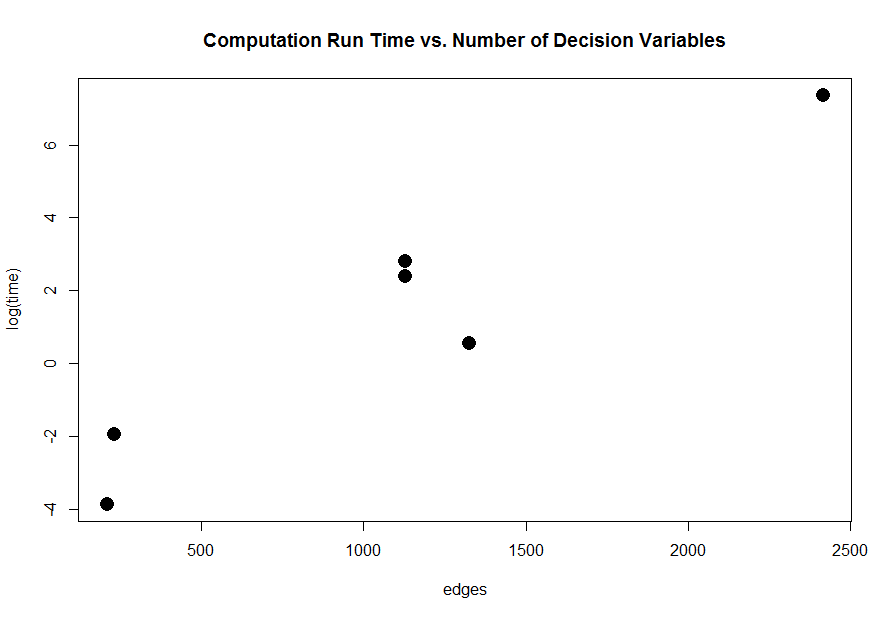
****

Figure 2: Run time results from test instances

***Concluding Remarks***

Overall, the traveling salesman problem has a variety of applications to optimize distances, costs, or time. However, the number of constraints in the formulation of the problem can vary depending on the algorithm used. Originally, if a city was added to the formulation, then the constraints exponentially increased ensuring to ensure no subtours, However, by taking minimum cuts along the edges connecting the cities, the formulation will be more efficient. Therefore, in our algorithm we implemented a cut formulation of the problem that involved a branch-and-cut method to find an optimal integer solution. However, our implementation of this algorithm is taxing on the memory. After adding an upper bound algorithm into our formulation, the results improved, but not significantly. In general, our cut formulation of the traveling salesman problem is efficient, but there are several areas to consider to enhance our algorithm.

***Future Work***

For future work, one could analyze how to reduce the memory in the algorithm. For example, in the branching process, our algorithm would be less taxing if we saved the constraints rather than the new models. By reducing the memory, the traveling salesman problem would decrease the running time, which would be useful when the formulation contains more cities and paths. Another way to improve our implementation is by changing our upper bound algorithm. One could look into using the Christofides Heuristic algorithm which will find approximate solutions within a range of error. In general, there are a few changes to examine to improve the running time of our traveling salesman problem.

Also, one could compare how changing a graph G = (V,E) into a Gomory-Hu tree G’ = (V, E’), affects the running time. According to Koutris Paraschos and Vasileios Syrgkanis (2014), a minimum cut in G’, is also a minimum cut in G. Moreover, there is exactly one minimum cut when the Gomory-Hu tree has n-1 distinct weights. On the other hand, actually constructing a Gomory-Hu tree requires another algorithm that may be too expensive when compared to the decrease in time in the minimum cut algorithm. In general, when considering these trees, one could evaluate how adding subtour elimination constraints found from the minimum cuts of the Gomory-Hu tree can affect the running time and efficiency of the travelling salesman problem.

***References***

Cook, William J., William H. Cunningham, William R. Pulleyblank, and Alexander Schrijver. "7. Traveling Salesman Problem." *Combinatorial Optimization*. New York: Wiley, 2010. N. pag. Print.

Paraschos, Koutris, and Vasileios Syrgkanis. "Gomory-Hu Trees: Theory and Applications." N.p., 30 Oct. 2014. Web. 18 Apr. 2017.

Stoer, Mechtild, and Frank Wagner. *A Simple Min Cut Algorithm*. Berlin: Freie U, Fachbereich Mathematik, 1994. Print.