**Traveling Salesperson Problem – Using Search Algorithms**

Will B. Crawford

CECS

Speed School of Engineering

University of Louisville, USA

[wbcraw02@louisville.edu](mailto:wbcraw02@louisville.edu)

1. **Introduction**

In order to study the Traveling Salesperson Problem, or T.S.P., depth and breadth first search algorithms were used. In this problem, a directed graph with specific connections were used in order to work best with the algorithms. The above two algorithms will perform significantly better than a brute force approach. This is because the cities are connected using specific values and in the above algorithms not all permutations of the best path will be generated. Instead, the algorithm will methodically go through the cities by either going through all connections of a city or by going through the deepest node in the graph.

1. **Approach**

The algorithms used in this approach were slightly modified versions of the original breadth and depth first searches. The only modifications made are, instead of blindly visiting each node, it will calculate the distance from the current node to the next node in the current state, and if that value is less than the distance already calculated, then, it will go to the next node. In *figure 2* and *figure 3*, the algorithms of DFS, or Depth First Search, are implemented through recursion and storing the distances of a node *n* and the distance it is from the source node. This algorithm and the algorithm in *figure 4* utilize this array to determine at each node *n* what the previous node is and the distance it is from the source. Using this method, the algorithm will not visit nodes that will increase the cost if it has already visited that node already.

Further optimizations can be made to enhance the performance of the algorithms, for instance. Djikstra’s algorithm can be used instead of BFS to save on computing time, and to always process any node once. Another optimization can be made to use A\* search to compute a heuristic using Euclidean distance to make use of an informed search algorithm.

1. **Results**

The algorithms performed very well, as you can see in *figure 1*, DFS, or Depth First Search, and BFS, or Breadth First Search, were very close in running times. In order to keep the algorithms strictly the same, only optimizations can be made are through space complexities, by implementing custom libraries for linked lists and queues. Also, optimizations can be made to utilize multi-threading.

* 1. **Data** (Describe the data you used.)

The data used was generated by a tool that randomizes Travelling Salesperson Problem nodes, the tool can be found in the references. In the following section, there are the outputs to all the given TSP files.

* 1. **Results** (Numerical results and any figures or tables.)

*Figure 1:*

|  |  |
| --- | --- |
| *Time* | *Algorithm* |
| 13.14 | *DFS* |
| 9.619 | *DFS* |
| 16.249 | *DFS* |
| 9.164 | *BFS* |
| 11.744 | *BFS* |
| 14.614 | *BFS* |

*Figure 2:*

void Graph::start\_dfs(int root, int goal) {

std::pair<int, double> \*distances;

distances = new std::pair<int, double>[V];

for (int i = 0; i < V; i++)

distances[i] = std::make\_pair(i, INFINITY);

distances[0].second = 0;

distances = dfs(root, goal, distances);

this->print\_distances(distances);

delete[] distances;

}

*Figure 3:*

std::pair<int, double>\* Graph::dfs(int root, int goal,

std::pair<int, double>\* distances) {

if (root == goal)

return distances;

for (auto itr = graph[root].begin(); itr != graph[root].end(); ++itr) {

double distance = distances[root].second + cities[root].distance(&cities[\*itr]);

if (distance < distances[\*itr].second) {

distances[\*itr].second = distance;

distances[\*itr].first = root;

distances = dfs(\*itr, goal, distances);

}

}

return distances;

}

*Figure 4:*

void Graph::start\_bfs(int root, int goal) {

std::pair<int, double>\* distances;

distances = new std::pair<int, double>[V];

std::list<int> queue;

distances[root] = std::make\_pair(0, 0);

queue.push\_back(root);

for (int i = 0; i < V; i++)

if (i != root)

distances[i] = std::make\_pair(i, INFINITY);

while (!queue.empty()) {

root = queue.front();

queue.pop\_front();

if (root == goal)

continue;

for (std::list<int>::iterator i = graph[root].begin(); i != graph[root].end(); ++i) {

double distance = distances[root].second + cities[root].distance(&cities[\*i]);

if (distance < distances[\*i].second) {

distances[\*i].second = distance;

distances[\*i].first = root;

queue.push\_back(\*i);

}

}

}

this->print\_distances(distances);

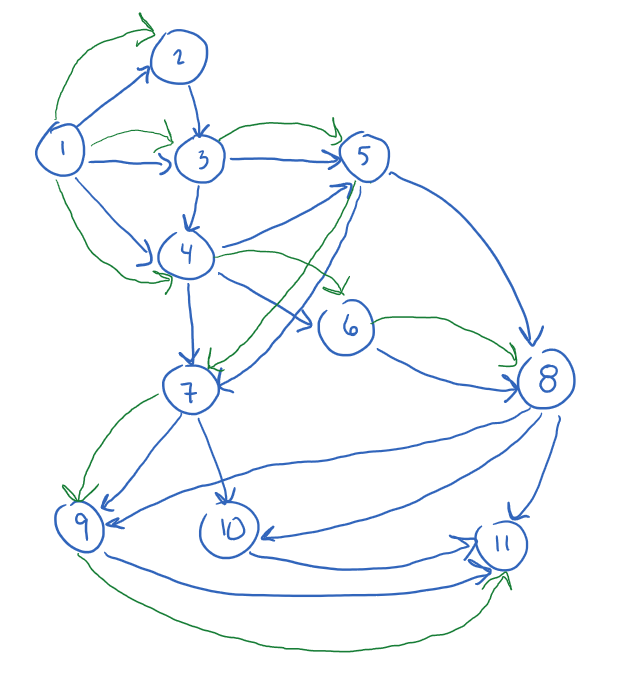
delete[] distances;

}

*Figure 5:*

**

*Figure 6:*

**

1. **Discussion**

As you can see in *figures 5* and *6*, the same best path was generated using both algorithms. Using this enhanced version depth and breadth first searches yields the same results and significantly faster than using any brute force methodology. The only limitations this algorithm has is when negative weight cycles are introduced into the graph, because the calculations will get stuck when encountering any negative edges. The running time of both algorithms are *O(V\*E)* where *V* is the number of vertices and *E* is the number of edges in the graph. These algorithms will be efficient when the graph size is not very large.

1. **References**

- http://www.math.uwaterloo.ca/tsp/