Project 3: Traveling Salesman

Nathan Stauffer, Daniel Olivas, Joanna Lew
Project Group 7

Spring 2017

Introduction

The traveling salesman problem (TSP) asks the question: "Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?" We attempt to find a reasonable path such that the distance of such path is no greater than 1.25× the shortest possible route.

To do so, we have researched three algorithms: a greedy algorithm called Nearest Neighbor, a 2-approx algorithm similar to Christofides' algorithm, and a 2-opt heuristic algorithm (also known as Lin-Kernighan heuristics).

Nearest Neighbor

Description

Nearest Neighbor is a brute-force, greedy algorithm. The concept is simple. Choose an arbitrary starting city (vertex). Then, compare the distance from one's current city to every other city (edge). Travel to the nearest unvisited city, and repeat the process until all cities have been visited. It should be noted that this is a heuristic algorithm, not an exact one, and a study by D.S. Johnson noted the algorithm on average yields a path 25% longer than the shortest possible path [1].

Pseudocode

```
NEAR_NEIGH(vector<int> cities){
    // where vector cities is in format: [id0, x0, y0, id1, x1, y1, ...]
    // get the number of cities
    int num_cities = cities.size() / 3;

    // create a n*n sized vector with distances from city to city
    vector<int> distance_vec;
    for (i = 0; i < num_cities; i++)
```

```
        for (j = 0; j < num_cities; j++){
            distance = calculate distance using formula
            distance_vec.push_back(distance);
        }

    int start = 11;              // choose an arbitrary vertex as a starting point
    int min, index;
    int total_distance = 0;      // total distance traveled
    vector<int> visited_cities;  // list of visited cities (tour)
    visited_cities.push_back(start);

    // do while there are still unvisited cities
    while (visited_cities.size() < num_cities){
        min = 999999;
        index = 123456;

        // compare every possible path for a city
        // travel to nearest neighbor
        for (int i = 0; i < num_cities; i++){
            if (distance to next city < min && not already visited){
                min = distance to next city;
                index = next city;
            }
        }

        // update distance traveled and visited city
        total_distance += min;
        visited_cities.push_back(index);
        start = index;
    }

    // complete the loop & update distance
    total_distance += distance from last city to start city

    // print distance & tour to file
    fout << total_distance << "\n";
    for (auto& x : visited)
        fout << x << "\n";
    fout << "\n";

    return;
}
```

2-Approx Christofides′ Algorithm

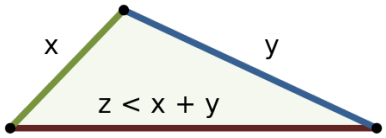
Description

Given a Metric TSP, we can make the following distance generalizations for all pair of vertices:

1. $d(x, y) \geq 0$
   Every edge, or distance from one vertex to another vertex, cannot be negative.

2. $d(x, y) = d(y, x)$
   The distance from city1 to city2 is the same forwards or backwards.

3. $d(x, y) + d(y, z) \geq d(x, z)$
   Given three vertices, the distance from one vertex to another, with the inclusion of another vertex in between, is always greater than equal to the direct distance. This is due to the triangle inequality.



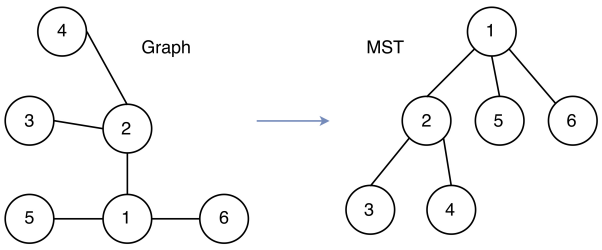
Our goal is to find a cycle and minimize the cost of that cycle. We define the cost, $c(S)$ to be the sum of the weight of all edges, where S is any set of edges. To do so, we first find a valid Hamiltonian cycle. This can be done by performing the following steps:

1. Turn the graph into a minimum spanning tree (MST)

2. Construct a path out of MST using depth first search (DFS)

3. Convert into cycle using triangle inequality to bypass repeated vertices

For example, given the following graph, we first convert it into a MST.



We can construct a path by using a DFS traversal. The resulting path is: $1, 2, 3, 2, 4, 2, 1, 5, 1, 6, 1$. We, then, use the triangle inequality to bypass repeated vertices. Doing so causes to path to become $1, 2, 3, 4, 5, 6, 1$, which is a valid cycle.

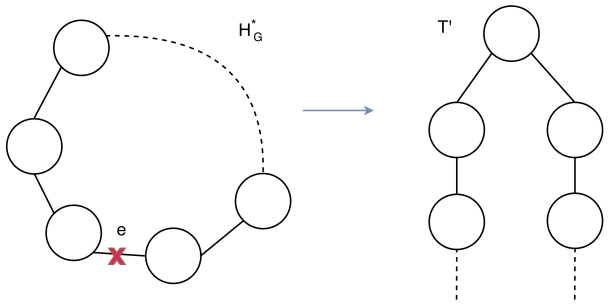
We define the MST as T, the path from the DFS as C (not to be confused with cost, which is lowercase c), and the cycle as C'. We get a 2-approximation by showing:

$$c(C') \leq c(C)$$
$$c(C) = 2\,c(T)$$

$$c(C') \leq 2\,c(T) \tag{1}$$

where $c(C) = 2\,c(T)$ is defined because every edge in DFS is being traversed twice (as a result of going down tree, then coming back up).

Given that some perfect Hamiltonian cycle exists, H_G^*, we want to show that T is bounded by H_G^*. We know H_G^* is just a cycle that goes through each vertex and goes back to the starting one, so we can take an edge, e, and delete it to get a spanning tree, which we call T.



We can conclude $c(H_G^* - e) \leq c(H_G^*)$ since the former is missing an edge, and therefore the weight must be less than the complete cycle.

We can also conclude $c(T) \leq c(H_G^* - e)$ since the cost of a minimum spanning tree is by definition less than or equal to the cost of a spanning tree. Putting them, we get the following inequality:

$$c(T) \leq c(H_G^* - e) \leq c(H_G^*) \tag{2}$$

Putting equations (1) and (2) together, we get the 2-approximation:

$$c(C') \leq 2\,c(H_G^*)$$

which means the cost of the cycle given by the algorithm can be no larger than double the cost of the perfect Hamiltonian cycle.

Pseudocode

```
// A utility function for PRIMS_MST to find the vertex with minimum key value
// from the set of vertices not yet included in MST
int minKey(int key[], bool mstSet[])
{
   // Initialize min value
```

```
   int min = infinity;
   int min_index;

   // Find the smallest key value
   for (int v = 0; v < V; v++)
     if (mstSet[v] == false && key[v] < min){
         min = key[v]
         min_index = v;
     }

   // Return index with smallest key value
   return min_index;
}

// Convert a graph into a MST using Prim's Algorithm
PRIMS_MST(vector<int> graph[V][V]){
     vector<int> parent(V);           // to store MST
     vector<int> key(V);              // Key values to pick min weight edge in cut
     vector<bool> MSTSet(V);          // set of vertices not yet included in MST

     // Initialize all keys as INFINITE
     for (int i = 0; i < V; i++){
        key[i] = infinite;
        mstSet[i] = false;
     }

     // Always include first 1st vertex in MST.
     key[0] = 0;    // Make key 0 so that this vertex is picked as first vertex
     parent[0] = -1; // First node is always root of MST

     // The MST will have V vertices
     for (int count = 0; count < V-1; count++)
     {
        // Pick the minimum key vertex from the set of vertices
        // not yet included in MST
        int u = minKey(key, mstSet);

        // Add the picked vertex to the MST Set
        mstSet[u] = true;

        // Update key value and parent index of the adjacent vertices of
        // the picked vertex. Consider only those vertices which are not yet
        // included in MST
        for (int v = 0; v < V; v++)

          // graph[u][v] is non zero only for adjacent vertices of m
          // mstSet[v] is false for vertices not yet included in MST
```

```
            // Update the key only if graph[u][v] is smaller than key[v]
            if (graph[u][v] && mstSet[v] == false && graph[u][v] < key[v])
               parent[v] = u, key[v] = graph[u][v];
        }

        // return the constructed MST
        return parent;
}

// find path with repeated vertices using DFS (Not Cycle)
DFS_PATH(graph[V][V], parent[V]){
        vector<int> cities = {0};
        int i = 0;
        vector<int> visited;
        visited.push_back(0);

        // build DFS
        while (visited.size() < num_cities){
            for (int i = 0; i < num_cities; i++){
                if (i not in visited && parent[i] in cities){
                    p_idx = find index of last instance of parent[i] in cities
                    insert i in cities[p_idx + 1]
                    insert parent[i] in cities[p_idx + 2]
                    visited.push_back(i);
                }
            }
        }
        return cities;
}

// Get a valid Hamiltonian Cycle from DFS
// by bypassing (deleting) repeated vertices
HAM_CYCLE(vector<int> DFS_path){
        vector<int> visited;
        for (int i=0; i < DFS_Path.size(); i++){
            if (DFS_path[i] not in visited){
                visited.push_back(DFS_path[i]);
            }
        }
        return visited;
}

// Calculate Distance and print Tour (Ham cycle)
CHRIS_2APX(vector<int> ham_cycle){
        int total_dist = 0;

        for (int i = 1; i < ham_cycle.size(); i++)
```

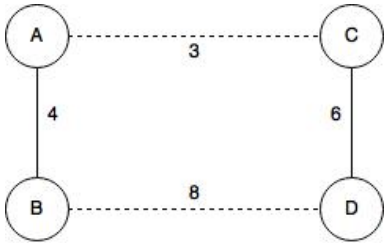
```
        total_dist += getDistance(i, i-1);          // use dist formula on [i, i-1]

    // print distance & tour to file
    fout << total_dist << "\n";
    for (auto& x : visited)
        fout << x << "\n";
    fout << "\n";
}
```

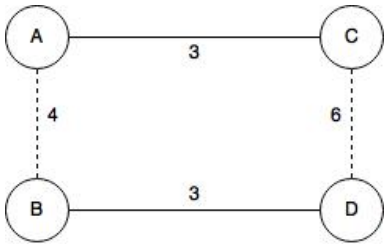
2-opt heuristic

The 2-opt heuristic is a technique that can be used - sometimes in combination with other techniques - to solve the TSP with reasonable accuracy. Essentially, the idea is fairly straightforward: for every set of four cities, there are two possible edge groupings.

Scenario 1: A and B are connected, and C and D are connected.



Scenario 2: A and C are connected, and B and D are connected.



*NOTE: Because of the triangle inequality principle, which states that the length of one side of a triangle can never be longer than the sum of the lengths of the other two sides, the scenario in which A and D are connected and B and C are connected can never yield optimal results for shortest length. This is why scenarios 1 and 2 are the only necessary options to consider for any set of four cities.

The 2-opt technique examines both scenarios and chooses the one in which the total edge weights are the smallest. In scenario 1, edges (A,B) and (C,D) yield a total weight of 10, while

the other two edges add up to 11. In scenario 2, edges (A,C) and (B,D) yield a total weight of 6, while the other two edges add up to 10.

There are many ways to implement the 2-opt heuristic and other techniques could be used in conjunction with it. For this particular city-to-city problem, here is the pseudocode for one possible approach, with a large portion of the logic borrowed from Martin Burtscher in a presentation found at http://on-demand.gputechconf.com/gtc/2014/presentations/S4534-high-speed-2-opt-tsp-solver.pdf

Pseudocode

```
Let d(a,b) denote the distance between city a and city b

do {
   minchange = 0
   for(i=0; i<cities-2; i++){
      for(j=i+2; j<cities; j++){
         change = dist(i,j) + dist(i+1,j+1) - dist(i,i+1) + dist(j,j+1)
            if(minchange > change){
               minchange = change
               //(i,j) and (i+1,j+1) are the ideal connections of the group
            }
            else{
               //(i,i+1) and (j,j+1) are the ideal connections
            }
         }
   }
} while minchange < 0
```

Implementation

We chose to implement Nearest Neighbor since it is the algorithm we felt we understood best. Although the results obtained using Nearest Neighbor were likely not as close to the optimal solution as the other two algorithms, we felt our understanding of Nearest Neighbor would allow us to more easily implement it, resulting in less coding errors and sparing us hours debugging. Also, as mentioned above, the research by Johnson showed that on average, Nearest Neighbor performs at 1.25× the optimum solution, so we thought it would provide a reasonable enough solution.

Results

Below are our results for the three example instances with no time limit.

File	Experimental Result	Optimal Result	Ratio	Time
Example 1	131,607	108,159	1.216	0.01 sec
Example 2	3,118	2,579	1.208	0.38 sec
Example 3	1,952,748	1,573,084	1.241	≈30 mins

Below are our results for the seven competition cases for both unlimited time and within 3 minutes.

File	Experimental Result	Time
Test-Input-1	6277	0.01 sec
Test-Input-2	9146	0.02 sec
Test-Input-3	15493	0.03 sec
Test-Input-4	20861	0.15 sec
Test-Input-5	28030	0.86 sec
Test-Input-6	40078	8.2 sec
Test-Input-7	63546	95.3 sec

References

[1] Johnson, D. S.; McGeoch, L. A. (1997). *The Traveling Salesman Problem: A Case Study in Local Optimization*. (PDF). In Aarts, E. H. L.; Lenstra, J. K. Local Search in Combinatorial Optimisation. London: John Wiley and Sons Ltd. pp. 215310.

[2] MIT OCW: Approximation Algorithms,
`https://www.youtube.com/watch?v=zM5MW5NKZJg`

[3] Wikipedia: Traveling Salesman Problem,
`https://en.wikipedia.org/wiki/Travelling_salesman_problem`

[4] GeeksForGeeks: Prim‘s Algorithm,
`http://www.geeksforgeeks.org/greedy-algorithms-set-5-prims-minimum-spanning-tree-mst-2/`

[5] A High-Speed 2-Opt TSP Solver: Martin Burtscher
`http://on-demand.gputechconf.com/gtc/2014/presentations/S4534-high-speed-2-opt-tsp-solver.pdf`