Grace Feigh
Dominic Phan
Jared Breedlove

TSP Report

Three Different Methods/Algorithms for Solving TSP

2-OPT: The 2-OPT algorithm is an optimization algorithm for the Traveling Salesman Problem. It chooses to take a route that crosses over itself, and then reorders it so that it does not cross over.

It does this continually until no more improvements are made, so the run time of this algorithm really depends on the dataset. In the worst case, could take 2^n 2-opt moves if the dataset doesn't converge in some strange local optimum cases. Most the time it does converge though and an arbitrary value can be set to limit the amount of times the algorithm runs.

Pseudocode:

```
while (improvementIsMade) {
  best_distance = calculateTotalDistance(existing_route)
  for (i = 1; i < number of nodes eligible to be swapped - 1; i++) {
    for (k = i + 1; k < number of nodes eligible to be swapped - 1; k++) {
      new_route = 2optSwap(existing_route, i, k)
      new_distance = calculateTotalDistance(new_route)
      if (new_distance < best_distance) {
        existing_route = new_route
      } else {
        improvementIsMade = 0
      }
    }
  }
}


2optSwap(route, i, k) {
  take route[0] to route[i-1] and add them in order to new_route
  take route[i] to route[k] and add them in reverse order to new_route
  take route[k+1] to end and add them in order to new_route
  return new_route
}
```

Dynamic Programming: To solve The Traveling Salesman problem using Dynamic Programming, we notice in the process of solving the problem, we often have repeating subproblems, therefore, to find the optimal solution for all subpaths of length n, we'll be using already known optimal partial tours of length n-1.

First, we select a node from 0 to n to be our starting vertex (such as vertex 1). Then, we store the optimal value from 1 to each node. Meaning we store:
1) The set of visited nodes in the subpath (all subsets) - space 2^n
2) The index of the last visited node in the path - space n

These subproblems are solved in linear time, therefore the total time complexity for this algorithm is O((2^n)(n^2)).

More simply, if C(S,i) is the minimum cost path visiting every vertex in set S once, starting at 1 and ending at vertex i, then our
Base Case: when size of S<=2, C(S,i) = distance(1,i)
When size of S > 2:
C(S,i) = min(C(S-{i},j)+distance(i,j)}

Pseudocode:

```
C = a 2-D array, indexed by subsets(1,...,i)
if sizeof(S) <= 2, then C[S,i] = distance(1,i);

for m = 3...n. //subproblem size
    for each subset(1...i) of size m:
        for each vertex j belonging to S where j!= 1:
            C[s,i] = min(C[S-{i},j)+distance(i,j));
Return min(C[S,j]+distance(j,1));
```

Greedy: Firstly, in the simplest way, greedy algorithms pick a determined most valuable metric from the problem and use that as the end all be all deciding what to pick next. In the TSP problem, it is a little more complex than that, but choosing the closest city would give a decent efficiency return. For my implementation, it determines the closest city to the current city. The issues that arise here are if there's a few cities clustered near each other, there's no guarantee the path it leads you down on would be the most efficient

Pseudocode:

```
nearestNeighborGreedy(startCity, distanceTable, visitedArray, n, cost, path):
  originatingCity = startCity
  visited[startCity] = 1
  path.push(startCity)
  While (path.size() < n):
    Minimum = infinity
    newCity = infinity
    for (i = 0; i < n; i++):
      if (distanceTable[startCity][i] < minimum && visitedCities[i] == 0 && i != city):
        minimum = distanceTable[startCity][i];
        newCity = i;
    visited[newCity] = 1
    path.push(newCity)
    startCity = newCity
    cost += minumum
  totalCost += distanceGrid[newCity][originatingCity]
  path.push(originatingCity)
```

Description of Chosen Algorithm

The algorithm our group decided on is a Greedy Algorithm. The algorithm pulls the information to be sorted from a file provided by the user. The next step is it sorts based on the lowest cost. Then the algorithm outputs a file with the sorted results.

Why We Chose Greedy Algorithm

Our group decided on greedy algorithms because of two key benefits: algorithm simplicity and efficiency. One con of greedy algorithms is that they're giving you the most efficient route as long as going to the least expensive option provides the user with the best route. For this problem, the most efficient answer would have more caveats, but Greedy gave us an algorithm that met/exceeded the problem requirements and gave us a simple, efficient solution.

Pseudo Code

```
Pull information from file to run through algorithm
nearestNeighborGreedy(startCity, distanceTable, visitedArray, n, cost, path):
  originatingCity = startCity
  visited[startCity] = 1
  path.push(startCity)
  While (path.size() < n):
    Minimum = infinity
    newCity = infinity
    for (i = 0; i < n; i++):
      if (distanceTable[startCity][i] < minimum && visitedCities[i] == 0 && i != city):
        minimum = distanceTable[startCity][i];
```

```
        newCity = i;
    visited[newCity] = 1
    path.push(newCity)
    startCity = newCity
    cost += minumum
  totalCost += distanceGrid[newCity][originatingCity]
  path.push(originatingCity)
```

Output sorted information to file.

Best Tours for Example Instances
Example 1: 132174
Example 2: 3147
Example 3: 1930997

Best Solutions for the Test Instances

Competition 1: 6008 .027s
Competition 2: 8820 .009s
Competition 3: 15345 .017s
Competition 4: 19892 .044s
Competition 5: 27331 .144s
Competition 6: 40770 .536s
Competition 7: 62680 3.168s