Robert Ottolia

Andrew Brown

Luke Konane

CS325 Winter 2016

Project 4: Traveling Salesman

**Problem Description**

Given a list of cities, along with their coordinates, find a path starting in one city traveling to all of the other cities, minimizing the distance traveled. Below are our attempts at an approximation solution to this problem description. We came up with two different algorithms in order to better understand the nature of the problem and see the different ways we could get ‘close’ to the optimal solution.

**Algorithm A: Greedy/Heuristic Approach**

**I. Overview**

The first function, slowTSP is used for inputs of less than 500 cities. It takes the list of cities, and makes two lists: visited, and mustVisit. The list mustVisit contains all cities, and the first one is moved to visited and popped from mustVisit. Then a while loop is set up to run as long as mustVisit contains any cities. Just inside the while loop, the shortestPath variable is set to infinity. As long as mustVisit is not empty, there are two for loops which will run inside the while loop. The first one loops over the visited array, and the second one over the mustVisit array. Inside the two loops, the distance is set to the distance from the visited[i] city and the mustVisit[j] city. The shortest distance found inside these two loops is taken by the shortestPath variable. So it is looping over the list^2 each time it finds a city to add to the tour.

The second algorithm is using the greedy heuristic approach. Meaning it is only an approximation of, and not the absolute optimal result. The function (greedyFasterTSP) receives a list of cities from the input in main, and marks them “mustVisit,” and a separate list for “visited” is created. A variable tourLength is created and set to zero. A while loop runs as long as mustVisit contains any cities. Inside the while loop, there is a for-loop that runs over the visited array. Then, another for loop inside that one which iterates over the mustVisit array. The “distance” variable is set to the distance between the city at visited[i] and mustVisit[j]. When the nearest city in the mustVisit array is found, it is popped from that array and added to the visited array. The tourLength is incremented in this case. The variable “current” is set to this most recent city.

So in simpler words: you take one city, and look at the distances to every other city. Take the nearest city found, and add it to the list of “visited” cities, which is the tour. Then the city you just added, do the same thing: compare distances to each city in the remaining list of cities. Since you are iterating over the list \* the list, this is n2.

What is the difference between the two?

The first algorithm is n3. It does a while loop, and for each iteration, it does TWO MORE loops just to find one city to add. Because each time it adds a city, it is comparing all cities in the tour (visited) to each city in the remaining list (mustVisit).

The faster algorithm doesn’t go this deep. It just has one nested loop, and each time it compares each city in the mustVisit array to ONLY ONE city. So there is a big difference in the two.

**II. Pseudocode**

greedyFasterTSP(cities)

mustVisit ← cities

visited ← [0]

current ← mustVisit[0]

visited[0] ← current

mustVisit[0] ←remove

while mustVisit:

shortestPath ← ∞

for 0 < len(mustVisit)

Do{

distance = distance between (current, mustVisit[i])

if distance < shortestPath

shortestPath = distance

nearestCity = mustVisit[i]

}

tourLength += shortestPath

visited ←append nearestCity

mustVisit ← remove nearestCity

current = nearestCity

tourLength += distance between visited[0] and last city in visited array

slowTSP(cities)

while mustVisit

for 0 < length(visited)

for 0 < length(mustVisit)

Do{

distance = distance between (visited[i], mustVisit[j])

if distance < shortestPath

shortestPath = distance

nearestCity = mustVisit[i]

}

tourLength += shortestPath

visited ←append nearestCity

mustVisit ← remove nearestCity

tourLength += distance between visited[0] and last city in visited array

**III. Results**

Three Example Instances:

|  |  |  |
| --- | --- | --- |
| filename | Best Tour | Time |
| tsp\_example\_1.txt.tour | 106817 | 0.059 |
| tsp\_example\_2.txt.tour | 2452 | 2.28 |
| tsp\_example\_3.txt.tour | 1964948 | 73.19899 |

Competition Test Instances:

|  |  |  |
| --- | --- | --- |
| Test Case | Tour Distance | Time (s) |
| test-input-1.txt.tour | 4949 | 0.0219 |
| test-input-2.txt.tour | 6974 | 0.126 |
| test-input-3.txt.tour | 11455 | 1.9019 |
| test-input-4.txt.tour | 15137 | 14.26 |
| test-input-5.txt.tour | 28685 | 0.405 |
| test-input-6.txt.tour | 40933 | 1.53 |
| test-input-7.txt.tour | 63780 | 9.194 |

**Algorithm B: Hill-climbing & 2/3-opt Swapping**

**I. Overview**

Given the input file of cities and their coordinates, the initial route is determined. The initial route is determined by picking the lesser of two routes: One which goes straight through the cities in the input text file and one that is generated randomly using the cities in the input text file.

The main part of the algorithm is a loop that terminates only when we have gone through a complete cycle of the route without being able to make it shorter. In our research we learned of a optimization technique called ‘Hill climbing’, whereby an arbitrary solution is attempted to be improved upon by changing a single element of the solution. In our algorithm, this technique is implemented when we pick a city in the route, and then pick a sample of the remaining cities and check if swapping routes between these cities results in a shorter route.

The nature of the ‘swapping’ that occurs between these randomly selected cities is as follows. A third city is chosen, between the current two, and then either a 2-opt swap or a 3-opt swap is performed randomly. The 2-opt swap in particular is well known for taking a route that crosses over itself and reordering so it does not, thereby decreasing the distance. The route is tested to see if it has been improved.

Before returning the improved route as a result, more random routes are made to see if they are even better. If it is, it is used as a starting point. In this way, we are picking a route and trying to make it shorter. If we are successful in making it shorter, it is tested against another random route to see if we can best it. In our testing this technique has worked quite well, with an optimal solution ratio of ~1.08, but was rather slow due to multiple nested loops.

**II. Pseudocode**

**cities** ← dictionary of cities parsed from input file (key is city; value is coordinates of city)

bestDistance, bestRoute ← bestPrelimaryRoute(cities) // Random or straight through

while route can be improved:

previous result ← bestDistance

for city in cities:

sampleCities ← random subsample of remaining cities

for city in sampleCities:

randomCity ← randomInt in sample range

if randomCity % 2 == 0:

perform 2-opt swap to see if results in shorter route

else

perform 3-opt swap to see if shorter route

if newDistance < bestDistance:

bestRoute = newRoute

bestDistance = newDistance

break out of both for loops for random comparison

else

continue for loop to find better route

// Continue execution here if find a better distance

randomRoute ← get another random route from cities

randomRouteDistance ← calculateRouteDistance(cities, randomRoute)

if randomRouteDistance < bestDistance:

bestRoute = randomRoute

bestDistance = randomRouteDistance

if bestDistance == prevResult:

break execution of while loop

else

continue while loop

calculateFinalRouteDistance(cities, bestRoute)

**III. Results**

|  |  |  |
| --- | --- | --- |
| filename | Best tour | Time Taken |
| tsp\_example\_1.txt.tour | **112337** | **25.8** |
| tsp\_example\_2.txt.tour | **2781** | **119.4** |
| tsp\_example\_3.txt.tour | **1698990** | **>5 min** |

As this algorithm was slower, we did not enter it into competition instance.