Project 4 - Group 6
Members: Kelsey Helms, Jay Steingold, Johannes Pikel
Class: CS325 - 400
Date: 2016.11.26
Due Date: 2016.12.02

Three possible methods to solve Traveling Salesman

1. Nearest Insertion Method

Nearest Insertion chooses the first vertex in the list(arbitrary) to start and the city is added twice to the tour. Then each city remaining in the list of cities is placed in the position in the tour that minimizes the total weight, which is the position in which the city is closest to both of its neighboring cities. This continues until all of the cities have been added to the tour.

2. Nearest Neighbor Method

The Nearest Neighbor algorithm chooses an arbitrary vertex (city) to start, this city is then added to the solution set. From this arbitrary vertex the closest neighbor is found amongst all other cities and this next city is added to the solution. As each city is added to the solution set it is marked as visited. Then the loop continues for every city as it is added to the solution, for all unvisited cities, until all cities have been added into the solution set.

There are some methods to improve on nearest neighbor. One of which is performing a double ended search where the first and last city in the solution are both compared to all other unvisited cities. The nearest city is then either prepended or appended to the solution.

Another method known as repetitive nearest neighbor performs the same search but starts the search from all possible starting cities.

The last improvement combines the double ended search with a weighted distance matrix, such that the cities that are the farthest away from other cities are given a slightly better proportional distance, so that they are added into the solution set a bit sooner rather than left to the very end. In this way the overall solution can be improved, because it may not find the least distance from each city to the next it will find the least overall average from each city to the next, thereby reducing the overall total distance travelled. Combined with double ended search this gives the nearest neighbor a good improvement over finding an approximate solution.

3. Genetic Heuristic

A Genetic algorithm uses a pool known as the population. The population is a random set of tours. The tour of cities in this case is the genetic makeup of that particular member of the population. Each tour is given a fitness level or in this case the total distance for that particular tour. A pair of tours are chosen, known as the parents based on certain criteria, for instance in a tournament selection, a the best fit (least total distance) parent from a subset of the total population is chosen. Two parents are chosen and their tour or genetic makeup is used to produce an offspring. This is done by a two point combination, such that a starting and ending point in the tour is chosen and the first parent provides the cities before and after these points, while the second parent fills in the remaining cities not already in the offspring. There is a small chance that the offspring will mutate further in order to keep a good level genetic diversity in the population. Mutation involves simply swapping a pair of randomly chosen cities in the tour. The offspring is added to a new population, once the new population is the size of the old population then the old population is removed and one complete Generation has passed. This cycle continues until all generations have passed. In this way the fitness parents produce offspring that should converge to the fittest tour, or in the case the tour with the least total distance.

Project 4 - Group 6
Members: Kelsey Helms, Jay Steingold, Johannes Pikel
Class: CS325 - 400
Date: 2016.11.26
Due Date: 2016.12.02

Research: We referenced quite a number of different sources to identify potential algorithms we discussed and implemented to see their performances on the problem set. An article that was useful as it is a good discussion of the different possible approximation, exact and other heuristic algorithms is found here in this paper.

We also used specific online resources to better understand and research our understanding of the pseudocode. A useful paper for the improved version of the nearest neighbor came from this scholarly paper published online.

Another excellent source we found from presentations specifically on the Traveling Salesman Problem. There are many but this one was very useful in it description and examples provided.

We made sure to cite our sources in the code as we implemented it.

Implementation: Of the three methods described above we chose to implement a version of all three. We decided to do this, so that we could see them in action and get a good understanding of how they performed, giving us more options to choose from when running the example instances as well as the instances from the competition.

We chose the nearest insertion, nearest neighbor and a genetic algorithm, because their implementation seemed fairly straight forward from the research we did. Other heuristics had what seemed like extremely complex decision trees that may not have lead to accurate results if even a minor mistake had been made.

So far we got the best results from the improved version of the double ended nearest neighbor implementation. It did not finish tsp_example_3.txt in under 3 minutes but it did provide a good ratio.

Pseudocode

Nearest Insertion:

```
List_of_cities = read_in_file(FILE_NAME)
totalDistance = 0
Tour = [list_of_cities[0], list_of_cities[0]]
list_of_cities.remove(list_of_cities[0])
#For each city in the list, find the cities it best fits between
For city in list_of_cities:
    minDistance = infinity
    minCity = tour[0]
    For i in length of tour - 2
        D1 = getCityDistance(city, tour[i])
        D2 = getCityDistance(city, tour[i + 1])
        if(d1 + d2 < minDistance):
            minCity = tour[i]
            minDistance = d1 + d2
    #Add the city between the cities that minimizes the total distance
    Half1 = half of tour up until the index of minCity + 1
    Half2 = half of tour from end until index of minCity + 1
```

```
        Tour = half1 + city + half2
    #Get the distance between cities in the tour
    For i to length of the tour - 1
        d = getCityDistance(tour[i], tour[i + 1])
        totalDistance += d

    #Remove final city in tour because of verification format
    tour.pop()

    #Extract city numbers only for the sake of verification
    Cities = []
    For all the cities in tour:
        cities.append(city[0])
```

Nearest Neighbor (double ended and weighted distance matrix):

```
let list_of_cities be an array of arrays of cities with x and y coordinates, bool for visited
let nearest_distance be infinity
let array be the distance_matrix
let distset be an empty set

#calculate the distance matrix
#find the two nearest starting cities


for city_1 in list_of_cities
    for city_2 in list of cities
        compute the euclidian distance from city_1 to city_2
        array[city_1][city_2] = euclidian distance
        if city_1 is not city_2
            if their euclidian distance is < nearest_distance
                nearest_distance = euclidian distance
                starting_city = city_1
                nearest_city = city_2

#calculate the distance of each city to all others naming its value as distset
for city_1 in list_of_cities
    distset appen 0
    for city_2 in list of cities
        distset[index city_1] = distset[city_1 value] + array[city_1 value][city_2 value]
let min_distset = distset[0]
let max_distset
```

```
#calculate the minimum, maximum and average of distances of each city
for city_1 in list_of_cities:
        if distset[city_1 value} < min_distset
                min_distset = distset[city_1 value]
        if distset{city_1 value] > max_disset
                max_distset = distset[city_1 value]

average_distset = (max_distset + min_distset) / 2

#calculate a new matrix from the combination of the distset and the matrix

for city_1 in list_of_cities
        for city_2 in list_of_cities
                array[city_1][city_2] = ((length(list_of_cities) * array[city_1][city_2]+distset[city_2])/2

#perform the nearest neighbor search as a double ended search with the
#modified distance matrix
let sol_of_cities be an empty set          # will store the solution
let opt_distance be 0                      #store the optimal distance of the solution
let nearest_dist = infinity

set starting_city visited          #the two cities nearest to each other
set nearest_city visited
add starting_city and nearest_city to sol_of_cities
let the opt_distance be the distance between city_1 and city_2

city_1 = nearest_city
city_3 = starting_city

while the sol_of_cities does not contain all cities
        nearest_dist = infinity
        nearest_dist_2 = infinity

        for city_2 in list_of_cities
                #get the modified distance from the cities at the front and end of the solution
                if city_1 is not city_2 and city_2 has not been visited
                        dist_to = array[city_1][city_2]
                        if dist_to < nearest_dist
                                nearest_dist = dist_to
                                nearest_city = city_2
                if city_3 is not city_2 and city_2 has not been visited
```

```
                        dist_to_2 = array[city_3][city_2]
                        if dist_to_2 < nearest_dist_2
                                nearest_dist_2 = dist_to_2
                                nearest_city_2 = city_2


        #prepend or append the nearest city depending whether it it closer to the first or last
        if nearest_city and nearest_city_2 have not been visited
                if nearest_dist < nearest_dist_2
                        append nearest_city to the sol_of_cities
                        add euclidian distance of city_1 and nearest_city to opt_distance
                        set nearest_city to visited
                        set city_1 to nearest_city
                else
                        prepend nearest_city_2 to sol_of_cities
                        add euclidian distance of city_3 and nearest_city_2 to opt_distance
                        set nearest_city_2 to visited
                        set city_3 to nearest_city_2

add the euclidian distance from the last city in the solution to the first city in the solution to opt_distance

write to file the sol_of_cities and the opt_distance
```

Genetic algorithm:

```
let GENERATIONS = 3500
let POPULATION = 200
let MUTATE_PROB = 5

for the size of the POPULATION
        population append a random tour of cities

let distance_matrix be a matrix of the distances from each city to all other cities

for the number of GENERATIONS
        for each individual in the population
                fitness_val = the tour distance of individual
                weight_population append (individual, fitness_val)

        let population = empty set

        for number of POPULATION / 2
                father, mother = the lowest tours of a randomly chosen subset of the population
```

#the choice is weighted with a probability factor as well

child_1 = a randomly chosen 2 point crossover of the father and mother
#child_1 receives the cities in order from the father before and after the points. The mother's cities fill in the gap in they are not part of the child_1

father, mother = the lowest tours of a randomly chosen subset of the population
child_2 = a randomly chosen 2 point crossover of the father and mother

add child_1 and child_2 to the population

In the remaining population find the tour with the lowest total distance
Write to file the tour with the lowest total distance and the tour of cities itself

Best tours and the time it took (No time limit)

nearest_neighbor with a double ended search and a weighted distance matrix **(nearest_dblimp2.py)**
produced the best tour for **tsp_example_1.txt** with a time of 0.095102072
Total distance = 125592
Ratio to optimal = 125592 / 108159 = 1.16

nearest_neighbor repetitive search with a short circuit (**nearest_neighbor.py)**
designed to stop when a current tour distance is greater than a previous optimal

```
flip3 ~/CS325/project4/nearest_dblimp2 197% python nearest_dblimp2.py tsp_example_1.txt
Time taken:  0.0951020717621
flip3 ~/CS325/project4/nearest_dblimp2 198% python tsp-verifier.py tsp_example_1.txt tsp_exam
ple_1.txt.tour
Each item appears to exist in both the input file and the output file.
('solution found of length ', 125592)
```

produced the best result for **tsp_example_2.txt** with a time of 12.10570502
Total distance = 2975
Ratio to optimal = 2975 / 2579 = 1.15

```
flip3 ~/CS325/project4/nearest_neighbor 183% python nearest_neighbor.py tsp_example_2.txt
Time taken:  12.1057050228
flip3 ~/CS325/project4/nearest_neighbor 184% python tsp-verifier.py tsp_example_2.txt tsp_exa
mple_2.txt.tour
Each item appears to exist in both the input file and the output file.
('solution found of length ', 2975)
```

nearest_neighbor with a double ended search and a weighted distance matrix **(nearest_dblimp2.py)** produced the best tour for **tsp_example_3.txt** with a time of 523.851702
Total distance = 1863112
Ratio to optimal = 1863112 / 1573084 = 1.18

```
flip3 ~/CS325/project4/nearest_dblimp2 201% python nearest_dblimp2.py tsp_example_3.txt
Time taken:  523.851701975
flip3 ~/CS325/project4/nearest_dblimp2 202% python tsp-verifier.py tsp_example_3.txt tsp_exam
ple_3.txt.tour
Each item appears to exist in both the input file and the output file.
('solution found of length ', 1863112)
```

Best solution for the competition test instances. Time Limit 3 minutes

Our best solutions for the test cases primarily came from two algorithms the nearest_dblimp2 that uses the weighted distance matrix to give distant cities a proportionally better chance to get added to the solution and from the repetitive nearest neighbor.

We recorded the following distances and times for the test cases from the flip server (local PC times in parentheses).

nearest_dblimp2 (nearest neighbor with a weighted distance matrix, double ended search)

nearest_neighbor repetitive

test-input-1 : Distance: 5603, Time: 0.0159549713135 (0.038883754767) Algorithm: nearest_dblimp2

test-input-2 : Distance: 8011, Time: 0.97706413269 (0.34900) Algorithm: nearest_neighbor (repetitive)

test-input-3 : Distance: 14067, Time: 0.302833080292 (0.0891565018579) Algorithm: nearest_dblimp2

test-input-4 : Distance: 19711, Time: 68.2930510044 (43.836) Algorithm: nearest_neighbor (repetitive)

test-input-5 : Distance: 27967, Time: 2.66168093681 (1.1595871208) Algorithm:nearest_dblimp2

test-input-6 : Distance: 38871, Time: 9.37059497833 (4.67754220118) Algorithm:nearest_dblimp2

test-input-7 : Distance: 60336, Time: 55.8759510517 (28.5657620805) Algorithm:nearest_dblimp2

One screenshot is below of the actual runs for the above algorithms from the FLIP server.

flip3.engr.oregonstate.edu - PuTTY

```
flip3 ~/CS325/project4 156% python nearest_dblimp2.py test-input-1.txt
Time taken:  0.0159549713135
flip3 ~/CS325/project4 157% python tsp-verifier.py test-input-1.txt test-input-1.txt.tour
Each item appears to exist in both the input file and the output file.
('solution found of length ', 5603)
flip3 ~/CS325/project4 158% python nearest_neighbor.py test-input-2.txt
Time taken:  0.97706413269
flip3 ~/CS325/project4 159% python tsp-verifier.py test-input-2.txt test-input-2.txt.tour
Each item appears to exist in both the input file and the output file.
('solution found of length ', 8011)
flip3 ~/CS325/project4 160% python nearest_dblimp2.py test-input-3.txt
Time taken:  0.302833080292
flip3 ~/CS325/project4 161% python tsp-verifier.py test-input-3.txt test-input-3.txt.tour
Each item appears to exist in both the input file and the output file.
('solution found of length ', 14067)
flip3 ~/CS325/project4 162% python nearest_neighbor.py test-input-4.txt
Time taken:  68.2930510044
flip3 ~/CS325/project4 163% python tsp-verifier.py test-input-4.txt test-input-4.txt.tour
Each item appears to exist in both the input file and the output file.
('solution found of length ', 19711)
flip3 ~/CS325/project4 164% python nearest_dblimp2.py test-input-5.txt
Time taken:  2.66168093681
flip3 ~/CS325/project4 165% python tsp-verifier.py test-input-5.txt test-input-5.txt.tour
Each item appears to exist in both the input file and the output file.
('solution found of length ', 27967)
flip3 ~/CS325/project4 166% python nearest_dblimp2.py test-input-6.txt
Time taken:  9.37059497833
flip3 ~/CS325/project4 168% python tsp-verifier.py test-input-6.txt test-input-6.txt.tour
Each item appears to exist in both the input file and the output file.
('solution found of length ', 38871)
flip3 ~/CS325/project4 169% python nearest_dblimp2.py test-input-7.txt
Time taken:  55.8759510517
flip3 ~/CS325/project4 170% python tsp-verifier.py test-input-7.txt test-input-7.txt.tour
Each item appears to exist in both the input file and the output file.
('solution found of length ', 60336)
flip3 ~/CS325/project4 171%
```

Best solution for the competition test instances. Unlimited time

We did find one solution for test-input-5 that was better than the above with the repetitive nearest neighbor, but it ran well over the 3 minute time limit. This solution is labeled as test-input-5-alt.txt.tour

test-input-5 : Distance: 27128, Time: 350.63317 Algorithm:nearest_neighbor (repetitive)