CS 325 Project 4 Report

Joseph Barlan
Jeff Mueller
Kelvin Watson

Description of Algorithm

The algorithm we implemented is based on the nearest neighbor approximation algorithm for the traveling salesman problem, and incorporates an added 2-optimization edge-switching routine to improve the approximation ratio $\rho(n)$ of the solution.

The nearest-neighbor algorithm accepts two arguments: the list of all cities, as well as the starting city, and starts by initializing several variables:

- A visited array holds the cities currently visited. It is initialized to contain just the starting city.
- Accordingly, the cost of the path is initialized to 0, and the number of visited cities to 1.
- The number of cities is stored in a variable called num_cities.
- The number of visited cities are stored in a variable called num_visited.

The following while loop continues to iterate until all of the cities have been visited. On each iteration, an adjacency list (dictionary) is created by calling the compute_distance_from_city_to_all_cities() subroutine using the last city visited. To conserve memory, only the distances from the last visited vertex are computed. This is in contrast to creating an adjacency list for every single city (vertex) at the beginning, which is space inefficient and can cause the operation to run out of memory when computing the approximation of the TSP for a large number of vertices. Once the adjacency list for the last-visited vertex is computed, we created a subroutine find_min_adjacent() to find the closest vertex (the nearest neighbor) to the current last visited vertex. Once the nearest neighbor is identified, the edge between it and the last visited vertex is added to the current cost of the path, and the nearest neighbor is added to the list of visited vertices. Note that when the number of visited cities is greater than 2, we remove the second-last-visited city from the cities list. This reduces the number computations performed by compute_distance_from_city_to_all_cities() by one on every iteration of the while loop, and results in time savings since visited cities no longer need to be considered for distance computation to find the nearest neighbor. This in turn, creates progressively smaller adjacency lists on each while loop iteration, again, reducing the number of computations required. Iterations continue until the number of visited cities has increased to the original number of cities, indicating that all of the cities have been visited. Finally, the distance from the last-visited city to the starting city is added to the cost, and the list of visited vertices and cost are both returned.

Once the nearest neighbor algorithm has completed execution, the 2-OPT() algorithm is called on the computed path to search for improvements that would result in a lower cost path. This 2-opt is based on an algorithm described in [1]. Specifically, the two_opt_switch() takes the current best route and distance, and computes a potential new edge by checking if the total of the distance of the path between a vertex i and i+1 and a vertex j to j+1 (where j is at least two vertices along the path away from i), i.e. the original from i to i+3, is a more economic path than a potential new path denoted by the sum of the distance between a vertex i and j and a vertex i+1 to j+1 (where again, j is at least two vertices along the path away from i along the original path). If it is found that the new path costs less than the original path, the best distance is updated, and the helper function 2OPTSWITCH is invoked to update the best route. This 2-optimization continues for every vertex i and corresponding vertex j (which starts at least two vertices away from i) until the best possible route and approximate path from the 2-opt is determined. Note that there is a different control flow for 2-opt for larger numbers of cities above 3000 to limit the time the algorithm runs. The original control loop will iterate n^2 until no improvements are found where n equals the number of cities. In the restricted control loop it will run 5000n or n^2 for 3000 =< n =< 5000. The longer the 2-opt algorithm runs, the greater the potential to improve on the best possible route and distance.

Pseudocode

The following is the pseudocode for the Nearest Neighbor TSP approximation algorithm, the 2-OPT algorithm, along with their helper (utility) functions:

```
function NEARESTNEIGHBORTSP(cities, starting_vertex):
   visited = [starting_vertex]
   num_cities = len(cities)
   cost = 0
   num_visited = 1
   while num_cities > num_visited:
       last_node = visited[-1]  #get the last city visited
       if num_visited > 2:
           second_last_node = visited[-2]
       #compute distances to all other cities from last node
       adjacency_last_node = compute_distance_from_city_to_all_cities(last_node, cities)
       #get the nearest neighbor to the last node
       neighbor, weight = find_min_adjacent(adjacency_last_node, visited)
       #Add the nearest neighbor and distance to the visited and cost parameters
       if neighbor == -1:
           raise ValueError, "No path found"
       else:
           cost += weight
           visited.append(neighbor)
           if num_visited > 2:
               del cities[second_last_node]
           num_visited += 1
   cost = cost + distance(cities, visited[0], visited[-1])
   return visited, cost

def 2OPT(cities, best_route, best_distance, unlimited=True):
   route_len = len(best_route)
   if unlimited:
       outer_range = route_len - 2
   else:
       outer_range = 100
   #to emulate a do while loop
   change_flag = True
```

```
    while change_flag:
        change_flag = False
        for i in range(outer_range):
            for j in range(i + 2, route_len - 1):
                org_edge = distance(cities, best_route[i], best_route[i+1]) + distance(cities, best_route[j],
best_route[j+1])
                  new_edge = distance(cities, best_route[i+1], best_route[j+1]) + distance(cities,
best_route[i], best_route[j])
                if( new_edge < org_edge):
                    best_distance -= (org_edge - new_edge)
                    best_route = two_opt_switch(best_route, i, j)
                    if unlimited:
                        change_flag = True
    return best_route, best_distance
```

Subroutines (helper functions)

```
#saves each vertex/city as a row [vertex num, x coord, y coord] in the cities 2D matrix
def read_vertices(filename, num_citities=0):
    cities = {}
    with open(filename, 'r') as f:
        for line in f:
            city = []
            for num in line.split():
                city.append(int(num))
            cities[city[0]] = [city[1], city[2]]
    return cities #returns: [ [v0, x0, y0], ..., [vn, xn, yn] ]

#returns dictionary of {city: distance, ...} corresponding to distance from given city parameter
def compute_distance_from_city_to_all_cities(city, cities):
    city_distances = {}
    dist = 0
    for to_city, coords in cities.iteritems():
            delta_x = cities[city][0] - coords[0]
            delta_y = cities[city][1] - coords[1]
            dist = int(round(math.sqrt(math.pow(delta_x,2) + math.pow(delta_y,2))))
            city_distances[to_city] = dist
    return city_distances

def find_min_adjacent(adjacent_dict, visited):
    #swap distance and vertex in tuple for easy sorting
    items = [(distance, vertex) for vertex, distance in adjacent_dict.items()]
    items.sort()
    for distance, vertex in items:
        if vertex not in visited:
            return (int(vertex), int (distance))
    return (-1, -1)

def distance(cities, a, b):
    dx = cities[a][0] - cities[b][0]
    dy = cities[a][1] - cities[b][1]
    dist = int(round(math.sqrt(dx*dx + dy*dy)))
    return dist

#Source: [2]
def 2OPTSWITCH(route, i, j):
    start = route[0:i+1]
    middle = route[i+1:j+1]
    middle = middle[::-1]
    end = route[j+1:]
    new_route = start + middle + end
    return new_route
```

Time Complexity Analysis

In the NEARESTNEIGHBORTSP algorithm, the while loop runs in time linear to the number of cities in the graph. As every city must be visited, this outer while loop runs in $\Theta(n)$ time. On each iteration of the while loop, the compute_distance_from_city_to_all_cities() subroutine is invoked, which runs in time linear to the number of cities as each city is connected to every other city. Because the number of cities is reduced on each NEARESTNEIGHBORTSP's outer while loop iteration, the running time of the compute_distance_from_city_to_all_cities() algorithm is bounded above by inputted number of cities, and so the running time is O(n). Also invoked in NEARESTNEIGHBORTSP is the find_min_adjacent() subroutine, which needs to find the closest neighbor to the current vertex. It first sorts the adjacency list by distance (smallest to largest), which is an O(nlgn) operation [3]. The outer for loop runs in O(n) time, checking through all of the vertices in the sorted list. The if vertex not in visited: condition is an O(n) operation [3]. Thus, O(nlgn)+O(n)*O(n) results in an overall time complexity of $O(n^2)$ for the find_min_adjacent() function. On average, we expect this algorithm to run in less time $O(n^2)$ since the outer for loop searches through an already sorted list and should therefore encounter closer vertices first. In the outer while loop of NEARESTNEIGHBORTSP, the del (delete) operation is an O(n) time operation due to the need to shift list element after a deletion [3]. The last operation in NEARESTNEIGHBORTSP is the addition of the cost of the distance from the last visited vertex to the source, which consists of several constant time operations (subtraction, list element access, and math operations [4]. Taken together, although the overall run time of NEARESTNEIGHBORTSP is $O(n^3)$, the algorithm is expected to run faster than this upper bound due to the sorting of the nearest neighbor vertices as described above.

The 2-OPT algorithm allows an option to limit its running time. If the unlimited argument is set to True (default argument), then the number of iterations of the outer for loop is bounded the number of cities in the path (minus 2). If the unlimited argument is

Best Tours for Example Instances and Time Required

Results were obtained by running algorithms on the FLIP OSU server.

	tsp_example_1	**tsp_example_2**	**tsp_example_3**
Time required (seconds)	0.08	1.44	318.4 (669.47 for 5000)
Computed total distance using NEARESTNEIGHBORTSP algorithm	150393	3210	1964948
Computed total distance after 2-optimization	**126642** (unlimited 2-OPT's)	**2920** (unlimited 2-OPT's)	**1962979** using 100 2-OPT's **1903095** using 5000 2-OPTs)
$\rho(n)$ approximation ratio	$\frac{126642}{108159} = 1.171$	$\frac{2920}{2579} = 1.132$	$\frac{1962979}{1573084} = 1.248$

for minimization problem = $\frac{approximate\ solution}{optimal\ solution}$			(using a limit of 100 2OPT operations)
Computed route after 2-Optimization	See Appendix I	See Appendix I	See tsp_example_3.txt.tour file

Best Tours for Competition Test Instances

	test-input-1.txt	test-input-2.txt	test-input-3.txt	test-input-4.txt	test-input-5.txt	test-input-6.txt	test-input.7.txt
Time required (seconds)	0.02	0.08	0.85	4.93	17.27	67.79	99.41
Computed total distance using NEARESTNEIGHBORTSP algorithm	5926	9503	15829	20215	28685	40933	63780
Computed total distance after 2-optimization	**5639**	**8746**	**13554**	**17940**	**26046**	**35715**	**58805** (using 5000 2-opt's)
Computed route after 2-Optimization	see corresponding .tour files						

Instructions for running code

Unzip all of the files into a directory. Use the following to run the program

python main.py inputfilename

The program will output the tour, cost, and time to the console and also produce an output file based on the project specifications named **inputfilename.tour** in the same folder.

Sources

[1] https://web.tuke.sk/fei-cit/butka/hop/htsp.pdf

[2] http://codereview.stackexchange.com/questions/72265/2-opt-algorithm-for-traveling-salesman for route

[3]https://www.ics.uci.edu/~pattis/ICS-33/lectures/complexitypython.txt

[4]http://introcs.cs.princeton.edu/python/41analysis/

Appendix I

tsp_example_1

Computed NearestNeighbor TSP route for tsp_example_1

[0, 75, 1, 2, 23, 22, 21, 25, 24, 46, 45, 44, 48, 47, 69, 68, 67, 50, 49, 52, 53, 54, 42, 43, 28, 29, 30, 31, 19, 20, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 37, 36, 35, 34, 40, 41, 60, 59, 58, 57, 63, 64, 62, 61, 55, 56, 51, 66, 65, 71, 72, 73, 39, 38, 32, 33, 27, 26, 4, 3, 74, 70]

Computed NearestNeighbor TSP route for tsp_example_1 after 2-optimization

[0, 75, 2, 1, 23, 22, 25, 21, 3, 4, 20, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 74, 15, 16, 17, 18, 37, 36, 38, 39, 61, 62, 63, 64, 73, 72, 71, 65, 66, 51, 56, 55, 57, 58, 59, 60, 41, 40, 34, 35, 33, 32, 19, 31, 30, 29, 26, 27, 28, 43, 42, 54, 53, 52, 50, 49, 48, 44, 24, 46, 45, 47, 69, 68, 67, 70]

tsp_example_2

Computed NearestNeighbor TSP route for tsp_example_2

[0, 2, 3, 279, 278, 4, 277, 276, 275, 274, 273, 272, 271, 16, 17, 18, 19, 20, 21, 128, 127, 126, 125, 30, 31, 32, 29, 28, 27, 26, 22, 25, 23, 24, 14, 13, 12, 11, 10, 8, 7, 9, 6, 5, 1, 242, 243, 241, 240, 239, 238, 231, 232, 233, 234, 235, 236, 237, 246, 245, 244, 247, 250, 251, 230, 229, 228, 227, 226, 225, 224, 223, 222, 219, 218, 215, 214, 211, 210, 207, 208, 209, 252, 253, 254, 255, 256, 257, 258, 259, 260, 261, 262, 263, 264, 265, 266, 138, 137, 136, 135, 134, 270, 269, 268, 267, 140, 139, 148, 141, 142, 143, 144, 145, 146, 147, 149, 150, 178, 151, 152, 156, 153, 155, 154, 129, 130, 131, 132, 133, 15, 248, 249, 206, 205, 204, 203, 202, 200, 199, 198, 197, 194, 195, 196, 201, 193, 192, 191, 190, 189, 188, 187, 185, 184, 183, 182, 181, 176, 177, 180, 179, 160, 159, 158, 157, 119, 120, 121, 122, 123, 124, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 60, 61, 118, 62, 63, 64, 65, 66, 67, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 116, 115, 114, 111, 110, 108, 104, 103, 102, 101, 100, 99, 98, 93, 94, 95, 96, 97, 92, 91, 90, 89, 109, 112, 88, 87, 113, 117, 59, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 68, 69, 105, 106, 107, 173, 174, 161, 162, 163, 164, 165, 166, 167, 168, 169, 170, 171, 172, 175, 186, 220, 217, 216, 213, 212, 221]

Computed NearestNeighbor TSP route for tsp_example_2 after 2-optimization

[0, 3, 279, 278, 248, 249, 256, 257, 258, 259, 261, 260, 4, 277, 276, 275, 274, 273, 15, 18, 17, 16, 271, 272, 262, 263, 264, 265, 266, 267, 268, 269, 270, 134, 135, 136, 137, 138, 140, 139, 148, 141, 142, 143, 144, 200, 202, 201, 196, 195, 192, 191, 190, 189, 188, 187, 185, 184, 183, 182, 181, 176, 177, 179, 180, 186, 193, 194, 197, 198, 199, 145, 146, 147, 149, 150, 178, 151, 152, 156, 153, 155, 154, 129, 130, 131, 132, 133, 19, 20, 21, 128, 127, 126, 124, 123, 122, 121, 120, 119, 157, 158, 159, 160, 175, 161, 162, 163, 164, 165, 166, 167, 168, 169, 170, 171, 172, 173, 174, 107, 106, 105, 104, 103, 102, 101, 100, 99, 98, 97, 96, 95, 94, 93, 92, 91, 90, 89, 109, 108, 110, 112, 88, 87, 116, 113, 111, 114, 115, 117, 118, 62, 63, 86, 85, 84, 83, 82, 81, 80, 79, 78, 77, 76, 75, 74, 73, 72, 71, 70, 67, 66, 65, 64, 69, 68, 58, 57, 56, 55, 54, 53, 52, 51, 50, 49, 48, 47, 46, 45, 44, 59, 61, 60, 43, 42, 41, 40, 39, 38, 37, 36, 35, 34, 33, 32, 31, 30, 125, 29, 28, 27, 26, 22, 25, 23, 24, 14, 13, 12, 11, 10, 8, 9, 7, 6, 5, 1, 2, 242, 243, 241, 240, 239, 238, 237, 236, 235, 234, 233, 232, 231, 246, 245, 244, 247, 250, 251, 230, 229, 228, 227, 226, 225, 224, 223, 218, 215, 214, 211, 210, 207, 208, 209, 252, 255, 254, 253, 206, 205, 204, 203, 212, 213, 216, 217, 220, 219, 222, 221]

tsp_example_3

See tsp_example_3.txt.tour for computed NearestNeighbor TSP route for tsp_example_3 after 100 two-optimization operations