CS 325 Final Project

Alexander Nead-Work, Alexander Rash, Lyell Read

Brute Force (DP)

Primitives[f1]:

```
def distance (a, b):
     return int(sqrt((a[0] - b[0])^2 + (a[1] - b[1])^2))

def in (a, b):
     # checks if a is in iterable b
     For x in b:
          If x == a:
               Return 1
     Return 0
```

Brute Force (DP) Method

```
     # Uses bottom up and memoize together
     # stored in format [v, [a .. b], w]
     #v = vertex, a .. b = remaining, w = total weight so far
     Global list o

     def p(v, r, r_w):
          #returns the value of the min path from v through all
vertices in list r
          If o contains element [v, r]:
               Return element from o

          Elif len(r) == 1:
               W = distance(v, r[0])
               O.append [v, r, w+r_w]
               Return [v, r, w+r_w]

          Else:
```

[1] Assumes the builtin int() function mimics the behavior of nearestint() from the homework prompt exactly - this is a fair assumption as most int() functions behave like this.

```
                T = min_by_elem_2[f2]( [p(k, {r}-k[f3], distance(v, k) + r_w)) for k in r][f4] )
                O.append(T)
                Return T

    def solve(v):
         S_v = random from (v)
         p(S_v, k, 0) for k in {v}-S_v
         Ret = p(S_v, {v}-S_v, 0)
         Ret[1] = path
         Ret[2] = cost
```

Brute Force Research
https://www.youtube.com/watch?v=XaXsJJh-Q5Y

Explanation: This pseudocode recursively tests out every possible node set out, and decides which is best. It uses a combination of the bottom up and memoize together. This algorithm is terribly slow and heavy in terms of memory used by calls to recursion, but it will find the best solution.

Christofides

```
    # Build the graph
    BuildGraph()
    graph = [][]
    for i in [0..num_vertices)
        for j in [0..num_vertices)
           graph[i][j] = graph[j][i] = euclid_distance(nodes[i], nodes[j])


    # Build the MST (Prim's)
    MST()
    multigraph = [][]
    key = Array()
    parent = Array()
    in_mst = Array()
    min_node = 0
```

[2] This compares based on third element in a list
[3] {} denotes set operations
[4] [] denotes a list comprehension in this context

```
key[0] = 0
parent[0] = -1

for i in [0..num_nodes)
    min_node = mst_min_key(num_nodes, key, in_mst)
    in_mst[min_node] = true

    for j in [0..num_nodes)
      if graph[min_node][j] and not in_mst[j] and
graph[min_node][j] < key[j]
           parent[j] = min_node
           key[j] = graph[min_node][j]

for i in [1..num_nodes)
    n2 = parent[i]
    if in_mst[i] and n2 is not -1
      multigraph[i].append(n2)
      multigraph[n2].append(i)



# Perfect Matching Graph
PerfectMatching(Array odd_vertices)
length = inf
saved_node = inf
i = 0

while odd_vertices.length > 0
    length = inf
    saved_node = inf

    for i in [1..odd_vertices.length)
      if graph[odd_vertices[0]][odd_vertices[i]] < length
           length = graph[odd_vertices[0]][odd_vertices[i]]
           saved_node = odd_vertices[i]

    multigraph[odd_vertices[0]].append(saved_node)
    multigraph[saved_node].append(odd_vertices[0])

    odd_vertices.remove(idx=0)
    odd_vertices.remove(val=saved_node)
```

```
# Euler Tour
current_vertex = 0
stack = Stack()
circuit = Array()

while graph[current_vertex].length > 0 || stack.length > 0:
    if graph[current_vertex].length is 0
      circuit.append(current_vertex)
      current_vertex = stack.pop()
    else
      stack.push(current_vertex)
      current_vertex = graph[current_vertex].pop()

# Build final tour (remove duplicates)
tour = Array()
for i in [0..circuit.length)
    if circuit[i] not in tour
      tour.append(circuit[i])

# `tour` contains the final circuit
```

Christofides research:
https://en.wikipedia.org/wiki/Travelling_salesman_problem
https://en.wikipedia.org/wiki/Christofides_algorithm
https://en.wikipedia.org/wiki/Metric_space
https://en.wikipedia.org/wiki/Minimum_spanning_tree
https://en.wikipedia.org/wiki/Matching_(graph_theory)
https://en.wikipedia.org/wiki/Induced_subgraph
https://en.wikipedia.org/wiki/Multigraph
https://en.wikipedia.org/wiki/Eulerian_path
https://en.wikipedia.org/wiki/Hamiltonian_path
http://www.math.uwaterloo.ca/tsp/book/contents.html
https://sites.math.washington.edu/~raymonda/assignment.pdf
https://www.youtube.com/watch?v=nICtId7Q59A
http://www.graph-magics.com/articles/euler.php
https://github.com/beckysag/traveling-salesman#traveling-salesperson-problem

Explanation: Christofides is an interesting algorithm, it uses a sequence of graph manipulations to end up with a valid path. First, christofides requires that we find the MST for the graph provided. This was easy as we had worked on it already in class. The next step is to collect all the nodes with an odd number of connected edges to them. Next, we must derive a perfect matching of the nodes in the graph. Ideally this matching would be minimal. Then, we must find an euler tour or euler circuit in the graph, and convert that to a hamiltonian path. From a hamiltonian path. This path is our final tour. Once there, we considered doing two-opt, but did not do it.

Nearest Neighbor

<u>Nearest Neighbor Heuristic Pseudocode</u>

```
Nearest Neighbor(Shortest path length, shortest path, vertexArray,
weightArray):
     For all vertices in the array:
          While not all nodes visited:
               #find the shortest connected edge with node not
               visited and with weight > 0
                    #set as current city
                    #set last city as visited
                    #add traveled edge weight to total path weight
```

<u>Language Specific Pseudocode</u>

```
     Read in file to nodes
     Solutions = []

     solution.append(nodes[0])
     nodes.remove(nodes[0])

     while nodes != []:

         current = solution[len(solution)-1]

         # find closest neighbor to current node
         nn = nodes[0]
         nd = large_int
         for node in nodes:
             if distance(node, current) < nd:
                 nn = node
                 nd = distance(node, current)

         # node now contains the closest node.
         solution.append(nn)
         nodes.remove(nn)
```

```
#print(solution, len(solution))

total_distance = 0
for x in range (0, len(solution)-1):
    total_distance += distance(solution[x], solution[x+1])
total_distance += distance(solution[0],
solution[len(solution)-1])
```

Nearest neighbor research:
http://digitalfirst.bfwpub.com/math_applet/asset/3/TSP_NN.html
https://en.wikipedia.org/wiki/Nearest_neighbour_algorithm
https://cse442-17f.github.io/Traveling-Salesman-Algorithms/
https://www2.seas.gwu.edu/~simhaweb/champalg/tsp/tsp.html

Description and research summary:

The nearest neighbor algorithm is a relatively fast method of solving the traveling salesman problem and could be considered the simplest method to implement. The algorithm attempts to select the shortest path by always traveling to the nearest vertex that has not previously been visited. The algorithm is considered to be pretty efficient as the work required does not increase very quickly as the number of vertices increases. The rate of increase for nearest-neighbor is on the order of n^2 which is considered pretty good.

Nearest-neighbor is an approximation algorithm. This means that it is fast but that at times it finds routes that are longer than the optimal route. The route found by the nearest neighbor can also be different depending on which vertex it starts at. The path found using the best verticy can be the best path, but this isnt always the case. In the worst case scenario, the best vertices will still be better than (N/2)-1 other possible paths.

The repetitive nearest-neighbor algorithm is a variation of the nearest-neighbor algorithm. In this variation, the nearest neighbor algorithm is run multiple times, using a different starting vertex for each. After all starting points are tested, the shortest path found is chosen as the answer.

Implementation Reasoning

After our initial research on tsp algorithms we wanted to implement Christofides. After difficulties getting a Christofides implementation to work, we chose to implement the nearest neighbor algorithm instead. We chose nearest neighbor because it can be one of the faster algorithms for solving the traveling salesman problem. Although the nearest neighbor will not always find the most optimal path, we were confident we could tune the algorithm to get our result closer. We initially implemented the algorithm in C with some tuning to speed up the algorithm. This still wasn't fast enough, most likely due to memory allocation. We looked to switch the implementation to a different language and eventually settled on Python. This new implementation was much faster, especially for the large inputs.

We implemented Nearest Neighbor using Python, and the built-in List and Tuple data structures to store the data. The theoretical runtime for Nearest Neighbor is O(V+E).

Results

Times were collected using the `time` command while running the script. All data was collected by using the Python Nearest Neighbor implementation.

Case	**Result**	**Time**
test-input-1	5926	0.071s
test-input-2	9503	0.093s
test-input-3	15829	0.242s
test-input-4	20215	0.623s
test-input-5	28685	1.473s
test-input-6	40933	5.676s
test-input-7	63780	35.391s
example1	150393	0.048s
example2	3210	0.261s
example3	1964948	5m22.946s