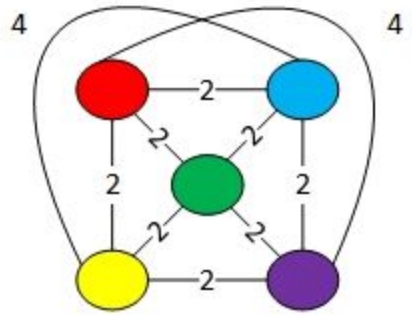
Christopher McLaughlin
Christiano Vannelli
Eugene Turgil

CS 325 Project 4: The Travelling Salesman Problem (TSP)

General Description of Methods:

1. The **Christofides Algorithm** is a method for finding approximate solutions to the Travelling Salesman Problem and guarantees that its solution will be within 3/2 of the optimal path length. It was named after Nicos Christofides who published the algorithm in 1976.

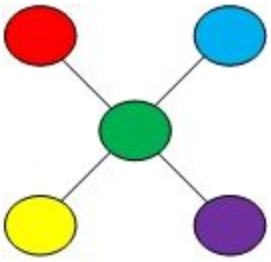
   Let G = (V, w), where G is a complete graph of V vertices, where function w assigns non-negative weights to the edges between vertices.



G

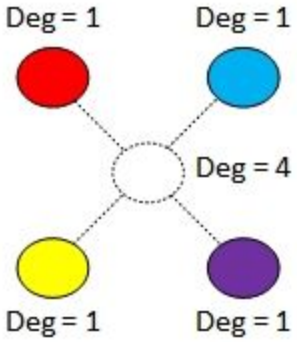
The algorithm can be divided into the following steps:

1. Create a minimum spanning tree T, from the graph G.

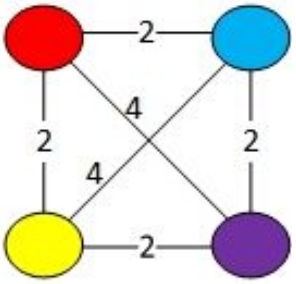


T

2. Let O be the set of vertices of odd degree within T. According to the handshaking lemma, O will contain an even number of vertices of odd degree.

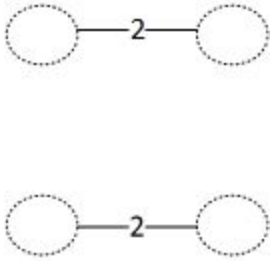


Vertices of O



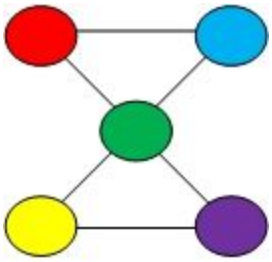
Subgraph with O

3. Find the minimum-weight perfect matching M in the subgraph obtained from the vertices within O.



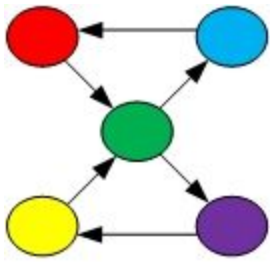
M

4. Combine the edges from T and M to create the multigraph H. Every vertex in H will have an even degree.



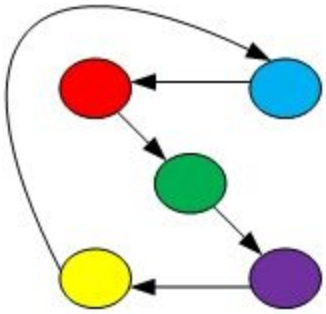
T U M

5. Determine the Eulerian Circuit in H.



Euler Tour

6. Use the Eulerian Circuit to create a Hamiltonian Circuit by skipping repeated vertices.



Hamiltonian Circuit

2. **Greedy Method #1** - Nearest Neighbor with ***Optimal*** Starting Vertex

   This approximation method for TSP takes a starting vertex and makes a greedy choice of choosing the nearest vertex as the next location to visit. This choice is repeated until all vertices have been visited. In order to further optimize the result; the method is repeated for each possible starting vertex and the tour with the minimum total path distance is selected as the the approximate shortest path.

3. **Greedy Method #2** - Nearest Neighbor with ***Fixed*** Starting Vertex

   This approximation method uses the same nearest neighbor principle as the greedy method from above. However, a fixed starting vertex is chosen and the method only produces a single tour. Since the results are not optimized further by considering

multiple starting vertices, we would expect to obtain a “less optimal” result with a faster execution time.

Implemented Approximation TSP Methods:

1. **Christofides algorithm**
    a. **Detailed Description** - This implementation utilizes a “City” class which stores x and y coordinates for the provided test file. Within this class are several methods such as returnX(), returnY(), returnNode() and distance(City). These return x and y coordinates, return the class object number and return the distance between two vertices.

    A CityList class was also created that exits to hold the complete list of City objects and contains the functions addCity(City), cityCount() and returnCity(index). These functions add a City object to CityList’s array, return the number of cities within the CityList’s array and return a specific City object by index value.

    Next, a Graph class was created that is initialized with a CityList object, creates a completed graph data structure and calculates the edge weights between all vertices. The “networkx” python package was utilized to create the complete graph by using the completeGraph() function. Within the Graph class, the methods returnGraph() and MST() exist, which return the complete graph object and calculate and return a graph object as the Minimum Spanning Tree respectively. The MST() function utilizes the networkx method minimum_spannning_tree(), which is based upon Kruskal’s algorithm.

    The getOdds() function is used with the MST graph as a parameter and returns an array of the vertex numbers which have odd degree. The networkx method .degree() is used in determining the degree number along side with a straightforward modulus operation.

    The PerfectMatch class is initialized with the list of odd vertices, the original complete graph and the MST graph. It creates a subgraph out of the list of odd vertices from the complete graph using the networkx method .subgraph(oddList) and also initializes a new multigraph with the .MultiGraph() method. The method edges() calculates the minimum perfect weight matching of the odd subgraph by utilizing the networkx function .max_weight_matching() but reversing all of the weights to negative values. The values are switched back to positive after the desired edges are obtained and the edges are added to the multigraph object along with the vertices and edges contained in the MST. This multigraph is returned by the edges() function.

The function Euler() is based upon the networkx method eulerian_circuit(), except that it is functional for multigraphs while the eulerian_circuit() function is not. The function performs an euler circuit by taking a multigraph object and utilizing a stack that contains its vertices. It continues to pop vertices out of the stack while following edges. It does this until no more vertices remain on the stack and the path has been output.

The class EulerCircuit is initialized with the results from the Euler() function and the CityList. It converts the results from the Euler() function into array format and contains the methods printList() and hamilton(). The printList() method simply prints the Euler Circuit, while the hamilton() method creates a hamilton path out of the Euler Circuit list. It does this by shortcutting; checking if vertices have been visited and skipping to the next vertex if so. Next, it calls upon the tspDistance() function which calculates the total path distance. Lastly, the path and total distance are returned.

b. ***Selection Discussion*** - We chose to implement this algorithm based on information from our research that showed that the method will produce within a 3/2 approximation factor; currently the most accurate approximation.

c. ***Pseudo Code*:**

class City: - Holds vertex data and returns distance between vertices

```
# initializes a city with it's line number, x, and y coordinates.
def __init__(self, number, x, y):
        self.node = number
        self.x ← x
        self.y ←  y

# returns x coordinate
def returnX(self):
        return self.x

# returns y coordinate
def returnY(self):
        return self.y

# returns city number
def returnNode(self):
        return self.node

# calculates the distance between vertices
def distance(self, City):
        dx ←  abs(self.returnX() - City.returnX())
        dy ←  abs(self.returnY() - City.returnY())

        distance ←  int(round(math.sqrt((dx * dx) + (dy * dy)), 0))

        return distance
```

class CityList: - Holds a list of the vertices and returns vertices

```
# holds the list of vertices
destinations ←  []

# function adds vertices
def addCity(self, City):
        self.destinations.append(City)

# returns the length of the list of vertices
def cityCount(self):
        return len(self.destinations)

# returns the vertex specified at index of the list
def returnCity(self, index):
```

```
return self.destinations[index]
```

class Graph: -Initializes a CityList into a complete graph, returns the graph and MST

```
    # initializes object containing a complete graph of the vertices with weights
        def __init__(self, cityList):
                self.size ← cityList
                self.cityList ← cityList
                self.graph ← nx.complete_graph(self.cityList.cityCount())

                for i ← 0 to i ← self.cityList.cityCount() do
                        for j ← 0 to j ← self.cityList.cityCount() do
                                if i != j then
                                        self.graph.edge[i][j]['weight'] ←
                self.cityList.returnCity(i).distance(self.cityList.returnCity(j))

                # returns the complete graph
                def returnGraph(self):
                        return self.graph

                # returns the minimum spanning tree
                def MST(self):
                        T ← nx.minimum_spanning_tree(self.graph)
                        return T
```

def getOdds(MST): - takes a MST as a parameter and obtains the odd degree vertices

```
                oddList ← []

                for i ← 0 to i ← len(MST.nodes() do
                        if (MST.degree(i) % 2 != 0) then
                                oddList.append(i)

                return oddList
```

```
class PerfectMatch: -Class helps convert the MST into a minimum matching multigraph

    def __init__(self, oddList, completeGraph, MST):
        self.MST ← MST
        self.oddList ← oddList
        self.completeGraph ← completeGraph
        self.oddGraph ← self.completeGraph.subgraph(oddList)
        self.multigraph ← nx.MultiGraph()
        self.multigraph.add_nodes_from(MST)
        self.mstEdges ← list(self.MST.edges())
        self.weights ← []

        for i ← 0 to i ← len(self.mstEdges)) do

            self.weights.append(self.MST[self.mstEdges[i][0]]
            [self.mstEdges[i][1]]["weight"])

        for i ← 0 to i ← len(self.mstEdges)) do
                self.multigraph.add_edge(self.mstEdges[i][0],
                self.mstEdges[i][1],weight=self.weights[i])

    # find the min weight matching of the odd vertices and appends
    # the edges to the MST in the form of a multigraph
    def edges(self):
        for v1, v2 in self.oddGraph.edges_iter() do
            self.oddGraph[v1][v2]["weight"] ← -(self.oddGraph[v1][v2]["weight"])

            edges ← nx.max_weight_matching(self.oddGraph, maxcardinality=True)
                for v1, v2 in self.oddGraph.edges_iter() do
                    self.oddGraph[v1][v2]["weight"] ← -(self.oddGraph[v1][v2]["weight"])

                    edgeList ← list(edges.items())

                    weights ← []

                    for i ← 0 to i ← len(edgeList) do
                        weights.append(self.oddGraph[edgeList[i][0]]
                        [edgeList[i][1]]["weight"])

                    for i ← 0 to i ← len(edgeList) do
                        for j ← 0 to j ← len(edgeList) do
                            if edgeList[i][0] == edgeList[j][1] then
                                self.multigraph.add_edge(edgeList[i][0],
                                    edgeList[i][1], weight=weights[i])
```

```
        return(self.multigraph)
```

```
# function calculates euler path.
# based upon (euler.py networkx python module function eulerian_circuit())
def Euler(G, source=None):
    g ← G.__class__(G) # copies graph structure

    # set up the starting vertex
    if source is None then
        v ← next(g.nodes_iter())
    else then
        v ← source

    degree ← g.degree
    edges ← g.edges_iter
    get_city ← itemgetter(1)

    stack ← [v]
    last_city ← None

    while stack do
        current_city ← stack[-1]
        if degree(current_city) == 0 then
            if last_city is not None then
                yield (last_city, current_city)
            last_city ← current_city
            stack.pop()
        else then
            random_edge ← next(edges(current_city))
            stack.append(get_city(random_edge))
            g.remove_edge(*random_edge)
```

```
# Calculates the total distance traveled in the hamiltonian circuit
def tspDistance(ham, cityList):
    total ← 0

    for i ← 0 to i ← len(ham) - 1) do
        total += cityList.returnCity(ham[i]).distance(cityList.returnCity(ham[i + 1]))

        total += cityList.returnCity(ham[len(ham) - 1]).distance(cityList.returnCity(ham[0]))

        return(total)
```

```
# Helps convert the euler circuit into a readable form. Uses list version of euler circuit to
make a hamiltonian path
class EulerCircuit:
        def __init__(self, cir, cityList):
                self.cityList ← cityList
                self.cir ← list(cir)
                self.root ← self.cir[0][0]
                self.path ← []

                self.path.append(self.root)

                for i ← 0 to i ← len(self.cir) do
                        self.path.append(self.cir[i][1])

        # prints the euler circuit
        def printList(self):
                print(self.path)

        # returns the hamilton path of the euler circuit
        def hamilton(self):
                visited ← np.zeros(self.cityList.cityCount())
                hamPath ← []

                for i ← 0 to i ← len(self.path) do
                        if visited[self.path[i]] == 0 then
                                hamPath.append(self.path[i])
                                visited[self.path[i]] ← 1

                distance ← tspDistance(hamPath, self.cityList)

                return(hamPath, distance)
```

d. ***Example Tours:*** (see .tour files for complete results)

Example Tours			
Case	Distance	Time (seconds)	Approx. Ratio
1	118275	0	1.09
2	2903	1	1.13
3	DNF	DNF	DNF

2. **Greedy Method #1** - Nearest Neighbor with ***Optimal*** Starting Vertex

    a. ***Detailed Description*** - This method uses a helper function that takes a set of vertices and a starting vertex to create a tour which is built using the nearest neighbor greedy choice. In order to do this, an array of unvisited vertices is populated with all possible vertices minus the starting vertex. The current vertex is set to the starting vertex and a loop is used find the unvisited vertex which has the minimum distance to the current vertex. This is calculated based on the rounded Euclidean distance description that was provided in the Project 4 requirements. The vertex that has the minimum distance to the current vertex is added to the tour and becomes the new current vertex. The distance between the two vertices is added to the total tour distance. This process is repeated until there are no remaining unvisited vertices. Once the last unvisited vertex is added, the distance between the last vertex and the starting vertex is added to the total tour distance, completing the tour. In order to further optimize the results, this method is repeated for each possible starting vertex. Once all possible starting vertices have been considered. The tour with the minimum total tour distance is selected as the approximate shortest path for this method.

    b. ***Selection Discussion*** - We attempted implementation of this algorithm based on information from our research that showed that the method will produce a shortest path that will be on average 25% larger than the optimal. This means that the algorithm should produce results close to our ratio of 1.25 (approximate/optimal) ratio that is a requirement for Project 4. This algorithm did not make the final cut into our tsp.py program however, as it ran too slowly for higher input values.

    c. *Pseudo Code*:

optimizeGreedyTSP() - Main Function

```
allPaths[] ←  empty
Tsp ←  (0, INF)
for vertex in vertices
        p ←  getgreedyTSP(vertices, vertex)
        allPaths.append(p)
 for path in allPaths
        If path < tsp
                tsp ←  path
return  tsp(path, pathDistance)
```

getGreedyTSP(vertices, startV) - Helper Function

```
pathDistance ←  0
```

```
unvisited[] ← All vertices minus startV
path[] ← empty
        path.append(startV)
        currV ← startV
        While unvisited != empty
                nearNB ← (0, INF)
                For vertex in unvisited
                        distNB ← getdistance(vertices[currV], vertices[vertex]
                        If distNB < nearNB
                                nearNB ← (vertex, distNB)
                path.append(nearNB vertex)
                unvisited.remove(nearNB vertex)
                pathDistance += distNB
                currV ← nearNB vertex
        pathDistance += getdistance(vertices[startV], vertices[nearNB])
return (path, pathDistance)
```

d. Example Tours: (see .tour files for complete results)

TSP Approx. Method:	tsp2Greedy.py	Nearest Neighbor with Optimal Starting Point			
Input File	**Verified**	**Exec. Time (sec)**	**Approximation**	**Optimal**	**App/Opt**
tsp_example_1.txt	YES	0	130921	108159	1.21
tsp_example_2.txt	YES	13	2975	2579	1.15
tsp_example_3.txt	NO	DNF	DNF	1573084	DNF

3. **Greedy Method #2** - Nearest Neighbor with ***Fixed*** Starting Vertex
   - ***a. Detailed Description -*** This method uses the same helper function as the previously described method. In order to speed up execution time for larger inputs, the starting vertex optimization was replaced with a fixed starting vertex (Vertex #1). Besides this modification, the remaining portions of the algorithm were kept the unchanged.

   - ***b. Selection Discussion -*** We selected to modify this algorithm based on the fact that we could not get Greedy Method #1 to calculate a shortest path in a reasonable amount of time on tsp_example_3.txt. When running our Greedy Method #1, we saw that it took approximately .5 seconds to calculate the nearest neighbor with an input of approximately 15,000 unvisited vertices. When only a single vertex is left unvisited, we assumed that the time to process would be 0. This meant that in order to process all unvisited vertices it would take approximately .25 sec x the total number of vertices to be visited (3,750 seconds for 15,000 vertices). Multiplying this by the 15,000 possible starting vertices used to optimize Greedy Method #1, we came up with an approximate execution time

of 56,250,000 seconds; equating to 651 days. This was unfeasible. By removing the starting vertex optimization we would be able to reduce our execution time by a factor of 15,000.

c. *Pseudo Code:*

optimizeGreedy2TSP() - Main Function

```
Tsp ← getgreedyTSP(vertices, 0)   #NOTE: 0 represents 1st vertex in list (fixed)
return tsp(path, pathDistance)
```

getGreedyTSP(vertices, startV) - Helper Function

```
pathDistance ← 0
unvisited[] ← All vertices minus startV
path[] ← empty
        path.append(startV)
        currV = startV
        While unvisited != empty
                nearNB ← (0, INF)
                For vertex in unvisited
                        distNB ← getdistance(vertices[currV], vertices[vertex]
                        If distNB < nearNB
                                nearNB ← (vertex, distNB)
                path.append(nearNB vertex)
                unvisited.remove(nearNB vertex)
                pathDistance += distNB
                currV ← nearNB vertex
        pathDistance += getdistance(vertices[startV], vertices[nearNB])
return (path, pathDistance)
```

d. *Example Tours:* (see .tour files for complete results)

TSP Approx. Method:	tsp2Greedy2.py	Nearest Neighbor with Single Fixed Starting Point (1st Vertex)			
Input File	**Verified**	**Exec. Time (sec)**	**Approximation**	**Optimal**	**App/Opt**
tsp_example_1.txt	YES	0	150393	108159	1.39
tsp_example_2.txt	YES	1	3210	2579	1.24
tsp_example_3.txt	YES	153	1964948	1573084	1.25

Example Tours			
Case	Distance	Time (seconds)	Approx. Ratio
1	118275	0	1.09
2	2903	1	1.13
3	1964948	153	1.25

Best Competition Tours (tsp.py)		
Case	Distance	Time (seconds)
1	5812	0
2	8255	0
3	13577	3
4	18826	10
5	25951	57
6	40933	2
7	63780	16

The competition cases 1 through 5 were performed using Christofides algorithm. Cases 6 and 7 were done using the Greedy Method with the fixed starting point so that they would complete within the 3 minute time frame. The memory constraint is too much for Christofides at high numbers of vertices. The tsp.py file along with the tspFunctions.py file decide which algorithm to utilize depending on the number of vertices read from file.