Comparing Algorithms for Travelling Salesperson Problem

By Zahra, Kathleen and Isaac

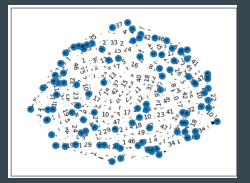
What is the Travelling Salesperson Problem?

- Given a list of locations, find the shortest distance to visit each one exactly once and return to the starting location
- Heavily studied since 1930s
- NP-Hard
- Some fast algorithms, however they are not guaranteed to give most optimal solution

Why is TSP important? (Motivation)

- Extremely useful for delivery based businesses
- Calculating most efficient route for deliveries can save money, gas, time
- Reduce carbon footprint
- Widely believed that there is no faster way to verify solution than O(n!)

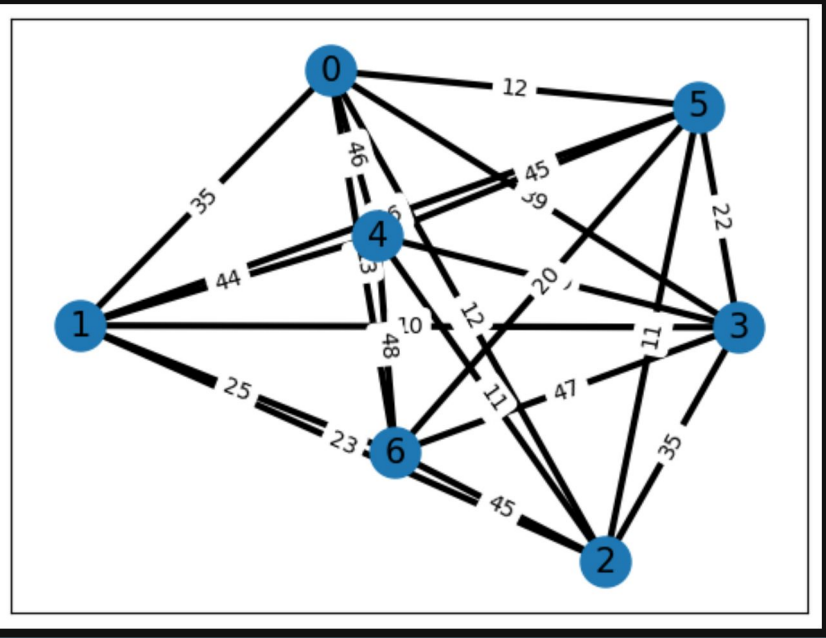
Generating random TSP graphs



```python
def node_generator(graph, n):
    # generates a graph with n nodes in it
    for i in range(n):
        graph.add_node(str(i))
```

```python
def edge_generator(graph):
    # adds edges between each node and every other node in the graph


    node_dist = [[0 for node in graph] for node in graph]
    # create matrix for hungarian algorithm
    # will return edge weights for nodes -
    # for example node_dist[1][2] == node_dist[2][1] and represents edge between nodes 1 and 2
    # diagonal x = -y is always 0

    for node1 in graph:
        node_dist[int(node1)][int(node1)] = INF
        for node2 in graph:
            if node1 != node2 and (node1, node2) not in gr.edges and (node2, node1) not in gr.edges:
                # avoid repeats
                x = np.random.randint(10, 50)
                # generate random weight for edges
                graph.add_edge(node1, node2, weight = x)
                node_dist[int(node1)][int(node2)] = x
                node_dist[int(node2)][int(node1)] = x
```

```python
pos = nx.spring_layout(gr, seed = 7)
nx.draw_networkx_nodes(gr, pos, node_size=500)
nx.draw_networkx_edges(gr, pos, width=3)
edge_labels = nx.get_edge_attributes(gr, 'weight')
nx.draw_networkx_edge_labels(gr, pos, edge_labels)
nx.draw_networkx_labels(gr, pos, font_size=15, font_family="sans-serif")
pass
```



Algorithms we're looking at:

- Hungarian Algorithm
- Nearest Neighbour
- Brute Force

What is the hungarian algorithm?

- Method that can be applied to a matrix to solve the assignment problem
- Assignment problem:

  The assignment problem consists of finding, in a weighted bipartite graph, a matching of a given size, in which the sum of weights of the edges is minimum.

How does this work on a matrix?

Step 1- Row Minimising

Work/ Job	1	2	3	4	5
1	∞	2	5	7	1
2	6	∞	3	8	2
3	8	7	∞	4	7
4	12	4	6	∞	5
5	1	3	2	8	∞

→

Work/ Job	1	2	3	4	5
1	∞	1	4	6	0
2	4	∞	1	6	0
3	4	3	∞	0	3
4	8	0	2	∞	1
5	0	2	1	7	∞

Step 1- Code

In [196]:

```
#step 1: find smallest element in each row of cost matrix and subtract it from each element in the row

def step_1(input_matrix):
    reduced_matrix = []
    for i in range(len(input_matrix)):
        new_row=[]
        row = input_matrix[i]
        smallest_element = 10000
        for k in range(len(row)):
            if row[k] < smallest_element:
                smallest_element = row[k]
        for j in range(len(row)):
            new_row.append(row[j] - smallest_element)
        reduced_matrix.append(new_row)

    return reduced_matrix
```

Step 2- Column Minimising

Work/ Job	1	2	3	4	5
1	∞	1	4	6	0
2	4	∞	1	6	0
3	4	3	∞	0	3
4	8	0	2	∞	1
5	0	2	1	7	∞

Work/ Job	1	2	3	4	5
1	∞	1	3	6	0
2	4	∞	0	6	0
3	4	3	∞	0	3
4	8	0	1	∞	1
5	0	2	0	7	∞

Step 2- Code

In [197]:

```
def transpose(prematrix):

    postmatrix =[[row[i] for row in prematrix] for i in range(len(prematrix[0]))]
    return postmatrix

#step 2: find smallest element in each column of cost matrix and subtract it from each element in the column

def step_2 (reduced_matrix):

    transposed_matrix = transpose(reduced_matrix)

    temp_matrix = step_1(transposed_matrix)

    intermediate_matrix = transpose(temp_matrix)

    return intermediate_matrix

```

Step 3- Assignment

Work/ Job	1	2	3	4	5
1	∞	1	3	6	0
2	4	∞	0	6	0
3	4	3	∞	0	3
4	8	0	1	∞	1
5	0	2	0	7	∞

Work/ Job	1	2	3	4	5
1	∞	1	3	6	[0]
2	4	∞	[0]	6	~~0~~
3	4	3	∞	[0]	3
4	8	[0]	1	∞	1
5	[0]	2	~~0~~	7	∞

Assignment- Code (methods)

```python
def single_value(input_row, minimum_values, i, crossed_off_values):

# this method determins whether there is a single value from minimum_values in the row

    row_zeros = []

    for j in range(len(input_row)):

            # if it is a 0 and it is not in the crossed off values, add it to the list

        if input_row[j] in minimum_values and [i,j] not in crossed_off_values:
            row_zeros.append([i,j])

    values = row_zeros[0]

    if len(row_zeros) == 1:
        return True, values
    else:
        return False, values
```

```python
def cross_off_coord(intermediate_matrix, crossed_off_values, values, assigned_values, minimum_values):

    p_coord = values[0]
    j_coord = values[1]


    for l in range(len(intermediate_matrix[p_coord])):

        if intermediate_matrix[p_coord][l] in minimum_values and [p_coord,l] not in assigned_values and \
        [p_coord, l] not in crossed_off_values:

            crossed_off_values.append([p_coord, l])


    for i in range(len(intermediate_matrix)):

        row = intermediate_matrix[i]

        for k in range(len(row)):

            #for k in range(len(intermediate_matrix)):

            if intermediate_matrix[k][j_coord] in minimum_values and [k, j_coord] not in assigned_values and \
            [k, j_coord] not in crossed_off_values:

                crossed_off_values.append([k, j_coord])
```

```python
def main_step(assigned_values, intermediate_matrix, crossed_off_values, minimum_values, passes):

    # prematurely coming out of this loop
     while len(assigned_values) != len(intermediate_matrix):

        for i in range(len(intermediate_matrix)):

            row = intermediate_matrix[i]

            boolean, values = single_value(row, minimum_values, i, crossed_off_values)

            # if it is a single value

            print(boolean, values)

            if boolean == True and passes < len(intermediate_matrix)*len(intermediate_matrix):

                # and it is not in assigned_values

                if values not in assigned_values:

                    #assign it
                    assigned_values.append(values)

                    # if you assign the coordinate, cross off the other zeros in the column

                    cross_off_coord(intermediate_matrix, crossed_off_values, values, assigned_values, minimum_values)



               # this loop should just be the case where there are multiple minimum values in the row
            elif boolean == False and passes >= len(intermediate_matrix)*len(intermediate_matrix):
                # assign the coordinate if it is not already assigned
                if values not in assigned_values:
                    assigned_values.append(values)


                    if values not in cross_off_values:
                        cross_off_coord(intermediate_matrix, crossed_off_values, values, assigned_values, minimum_values)

            passes += 1
            print(passes)
            print(assigned_values)
            print (crossed_off_values)


        return assigned_values, intermediate_matrix, crossed_off_values, minimum_values, passes
```

Assignment- Code Cont (put together)

```python
def assignment(intermediate_matrix):

    assigned_values = []
    crossed_off_values = []
    minimum_values = [0]
    passes = 0

    print (crossed_off_values)
    assigned_values, intermediate_matrix, crossed_off_values, minimum_values, passes = \
    main_step(assigned_values, intermediate_matrix, crossed_off_values, minimum_values, passes)

    #passes += 1

    print ("assigned values", assigned_values)
    true_false, route = check_route2(assigned_values)


    return route
```

```python
def hungarian_algorithm(input_matrix):

    reduced_matrix = step_1(input_matrix)
    print(reduced_matrix)

    intermediate_matrix = step_2(reduced_matrix)
    print(intermediate_matrix)

    route = assignment(intermediate_matrix)

    return route
```

An example

In [25]:

```
input_matrix = [
[ float('inf') , 2, 5, 7, 1],
[ 6, float('inf'), 3, 8, 2],
[ 8, 7, float('inf'), 4, 7 ],
[ 12, 4, 6,float('inf'), 5],
[1, 3, 2, 8, float('inf')]
]

hungarian_algorithm(input_matrix)
```

```
[[inf, 1, 4, 6, 0], [4, inf, 1, 6, 0], [4, 3, inf, 0, 3], [8, 0, 2, inf, 1], [0, 2, 1, 7, inf]]
[[inf, 1, 3, 6, 0], [4, inf, 0, 6, 0], [4, 3, inf, 0, 3], [8, 0, 1, inf, 1], [0, 2, 0, 7, inf]]
[]
assigned values [[0, 4], [1, 2], [2, 3], [3, 1], [4, 0]]
This is not a valid route for tsp.
The route is: [0, 4, 0]
```

Out[25]: [0, 4, 0]

How does this relate to tsp?

```python
def check_route2(assigned_values):

    route = [0]

    next_node = assigned_values[0][1]

    route.append(next_node)


    while route[-1] != 0 and len(route) < len(assigned_values):

        next_vertex = assigned_values[route[-1]][1]

        route.append(next_vertex)


    if len(route) == len(assigned_values) + 1 and route[-1] == 0:
        print ("This is a valid route for tsp!")
        print ("The route is:", route)
        return True, route

    else:
        print ("This is not a valid route for tsp.")
        print ("The route is:", route)
        return False, route
```

```python
def next_minimum(intermediate_matrix, minimum_values):

    next_min = 10000000000

    for i in range(len(intermediate_matrix)):
        for k in range(len(intermediate_matrix)):

            if intermediate_matrix[i][k]< next_min and intermediate_matrix[i][k] not in minimum_values:
                next_min = intermediate_matrix[i][k]

    minimum_values.append(next_min)

    return minimum_values
```

```python
    while true_false == False:

        print("boolean", true_false)

        minimum_values = next_minimum(intermediate_matrix, minimum_values)

        assigned_values = []
        crossed_off_values = []
        passes = 0

        assigned_values, intermediate_matrix, crossed_off_values, minimum_values, passes = \
        main_step(assigned_values, intermediate_matrix, crossed_off_values, minimum_values, passes)

        #passes += 1

        true_false, route = check_route2(assigned_values)


    return route
```

- Check_route method which checks the route
- If it is tsp valid, we have found out solution
- If not then we repeat the assignment steps including the next minimum values in the matrix
- Continue until there is a solution
- Particularly challenging putting it all together, I had to reorganise my code
- TSP section not complete
- If I could do it again...

Big O

- Finding the augmenting path (won't cover this) can take up to n^2
- You have to do this step n times
- So the Hungarian Algorithm has **O(n^3)**

- This is not the runtime for applying the hungarian algorithm to tsp, however tsp repeatedly uses the hungarian algorithm, so the big O of this applies to tsp must be larger than n^3

Nearest Neighbor Algorithm

- Greedy Algorithm (locally optimal choice)
- Tracking Three Lists

```
#starts at a city, and then always picks the closest neighbor
#passing in a city to start with (coordinate/vertex), and a graph of tuples (x,y)
def nearest_neighbor_algorithm(starting_city, all_cities):
    #starting with the first city, which will always be visited and part of the tour.
    current_city = starting_city
    visited_cities = []
    visited_cities.append(starting_city)
    tour_list = []
    tour_list.append(starting_city)
    comparison_counter = 0

    #all of the cities besides starting city are still unvisited at the start
    unvisited = []
    for city in all_cities:
        if city != starting_city:
            unvisited.append(city)
```

Nearest Neighbor Logic

```
#while there are still "cities" unvisited...
while len(visited_cities) < len(all_cities):
    nearest_neighbor = None
    nearest_distance = float('inf')
    #find a neighbor in a graph that's the smallest distance away
    for neighbor in unvisited:
        if neighbor not in visited_cities:
            distance = euclidean_distance(current_city, neighbor)
            comparison_counter += 1 #comparing current_city node to every other unvisited node
            if distance < nearest_distance:
                nearest_neighbor = neighbor
                nearest_distance = distance
    visited_cities.append(nearest_neighbor)
    tour_list.append(nearest_neighbor)
    current_city = nearest_neighbor
    unvisited.remove(current_city)

return print("tour list: ", tour_list)
```

Nearest Neighbor Example

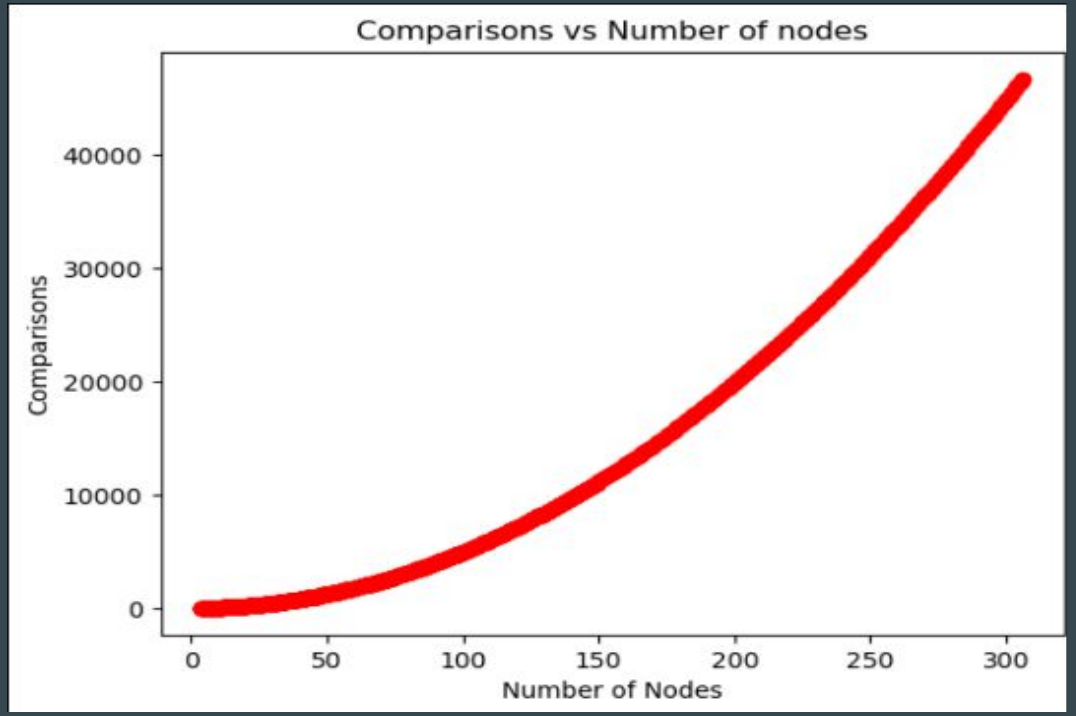
[38]:
```
group_of_cities = [
    (0,3),
    (1,14),
    (0,8),
    (0,7),
    (2,18)
]

starting_city = (0,3)
```

[39]:
```
nearest_neighbor_algorithm(starting_city, group_of_cities)
```

```
tour list:  [(0, 3), (0, 7), (0, 8), (1, 14), (2, 18)]
```

O((n^2)logn) Runtime

Fast, but has no guarantee to return the optimal tour list (path)

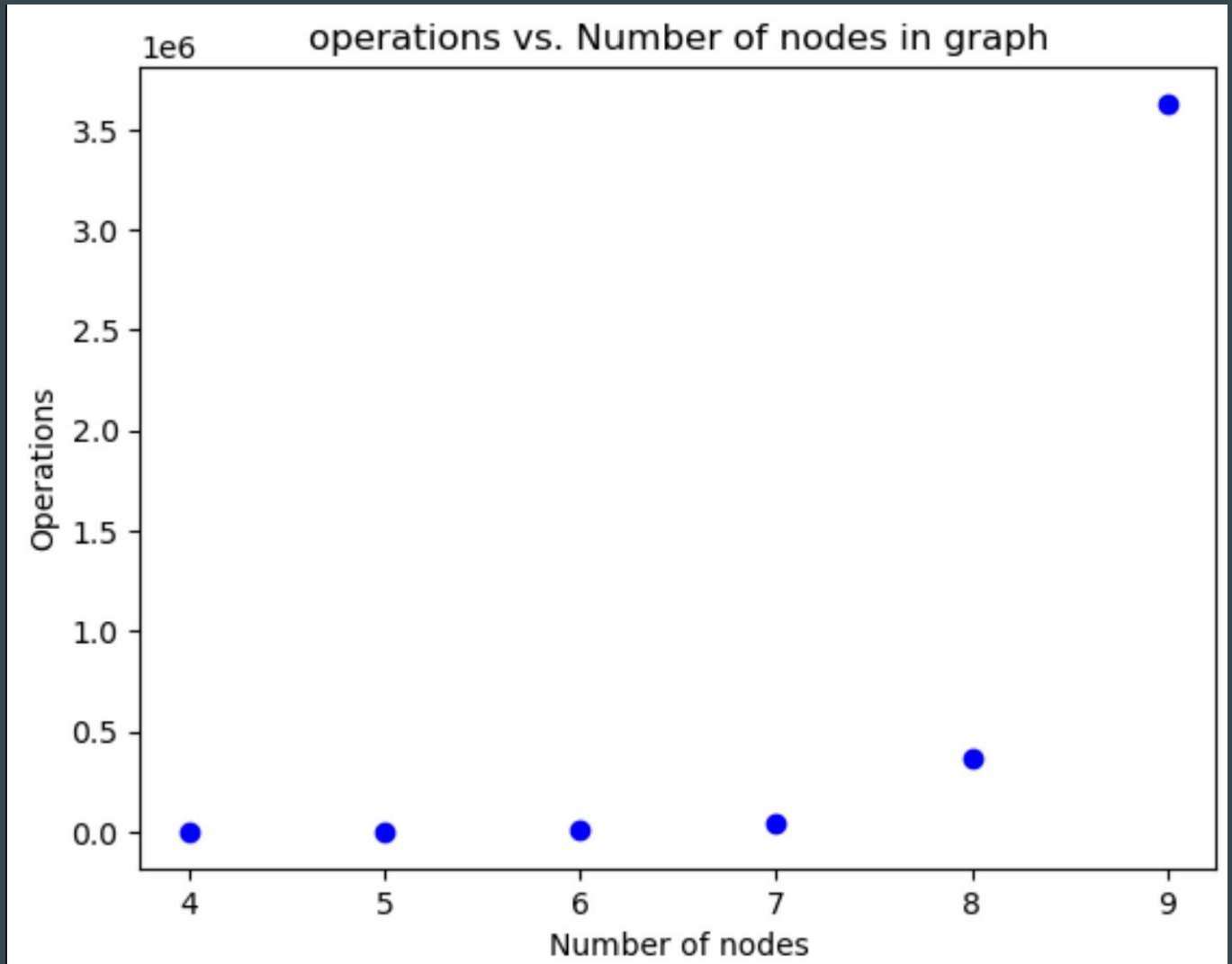


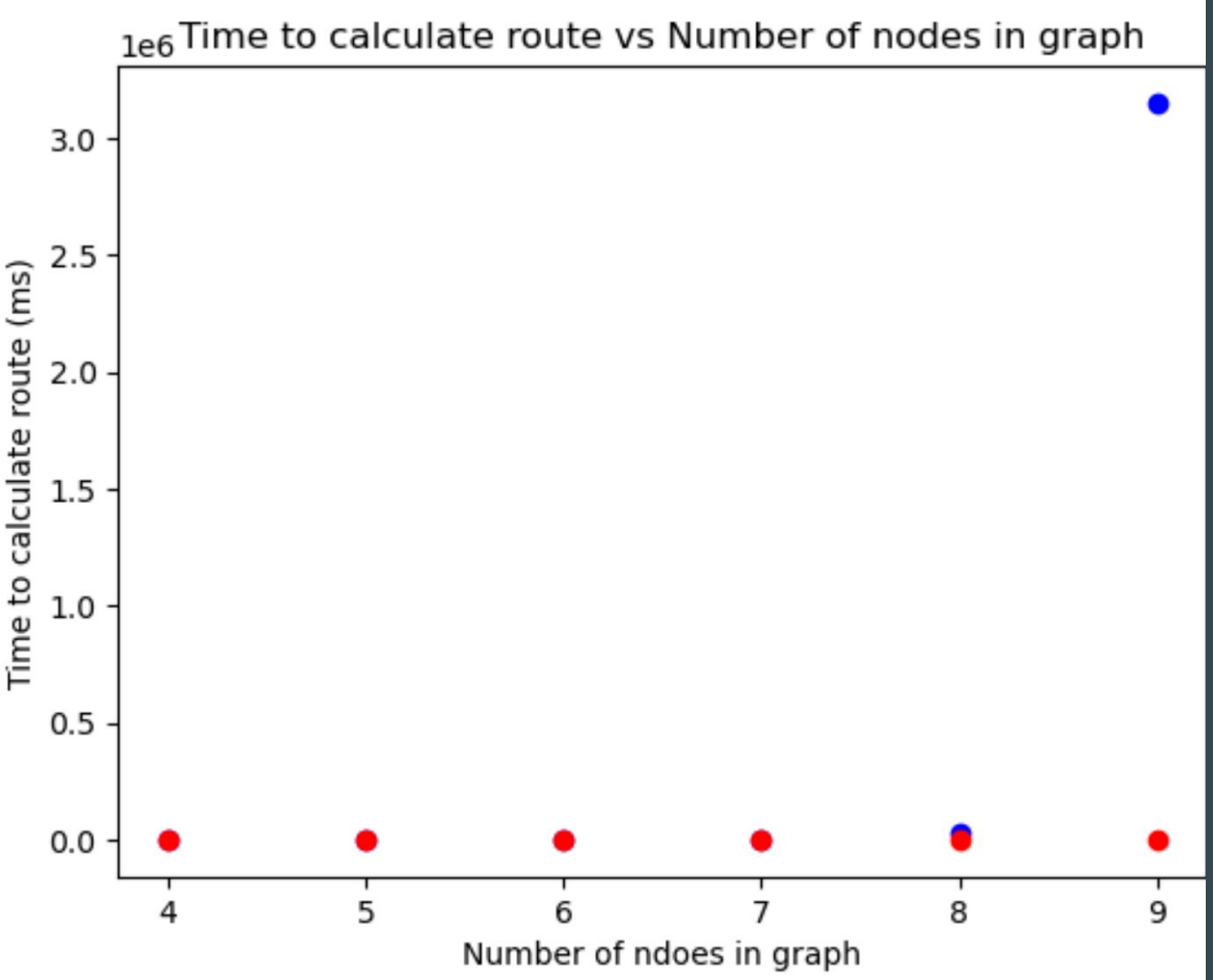

```
#find a neighbor in a graph that's the smallest distance away
for neighbor in unvisited:
    if neighbor not in visited_cities:
        distance = euclidean_distance(current_city, neighbor)
        comparison_counter += 1 #comparing current_city node to every other unvisited node
        if distance < nearest_distance:
            nearest_neighbor = neighbor
            nearest_distance = distance
```

Brute Force Algorithm

- Generate all possible routes that are valid within TSP
- Return route/s with smallest distance
- Pro - always returns shortest route if implemented correctly
- Con - extremely slow → O(n!)

```python
def brute_force_solver(graph):
    # make list of all possible routes
    routes = []
    # make list of the final weight of each route
    route_weights = []

    for node in graph:
        routes.append((node))
    # uses itertools to make list of every possible combination of nodes
    path_list = list(itertools.permutations(routes))

    # updates path list to add end point
    for path in path_list:

        index = path_list.index(path)
        # add start node as end node
        path_list[index] = path + (path[0], )
        route_weight = 0
        num_locations = len(path)

        for i in range(num_locations):
            # calculate total weight of route
            route_weight += graph.get_edge_data(path_list[index][i], path_list[index][i+1])['weight']
        route_weights.append((path_list[index], route_weight))

    # return route with smallest weight
    return tuple_sort(route_weights)[0]
```


operations vs. Number of nodes in graph
1e6
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
Operations
4
5
6
7
8
9
Number of nodes



Blue - Brute Force

Red - Nearest Neighbor

Questions?