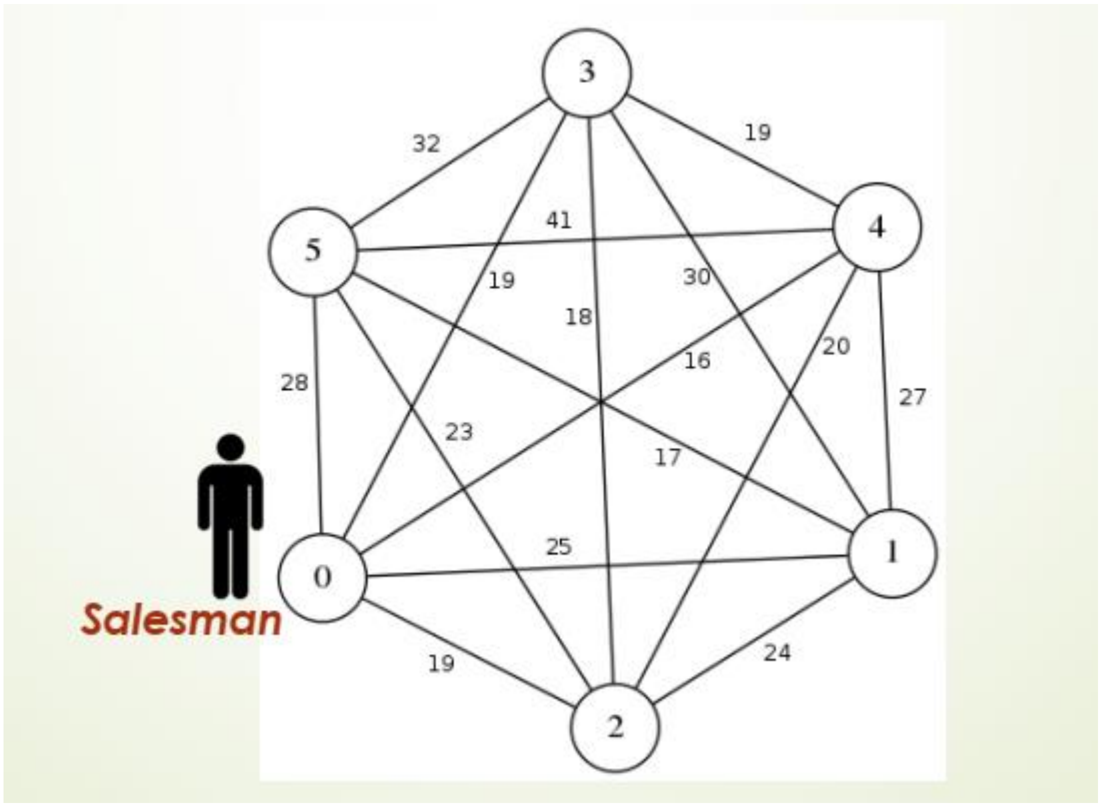
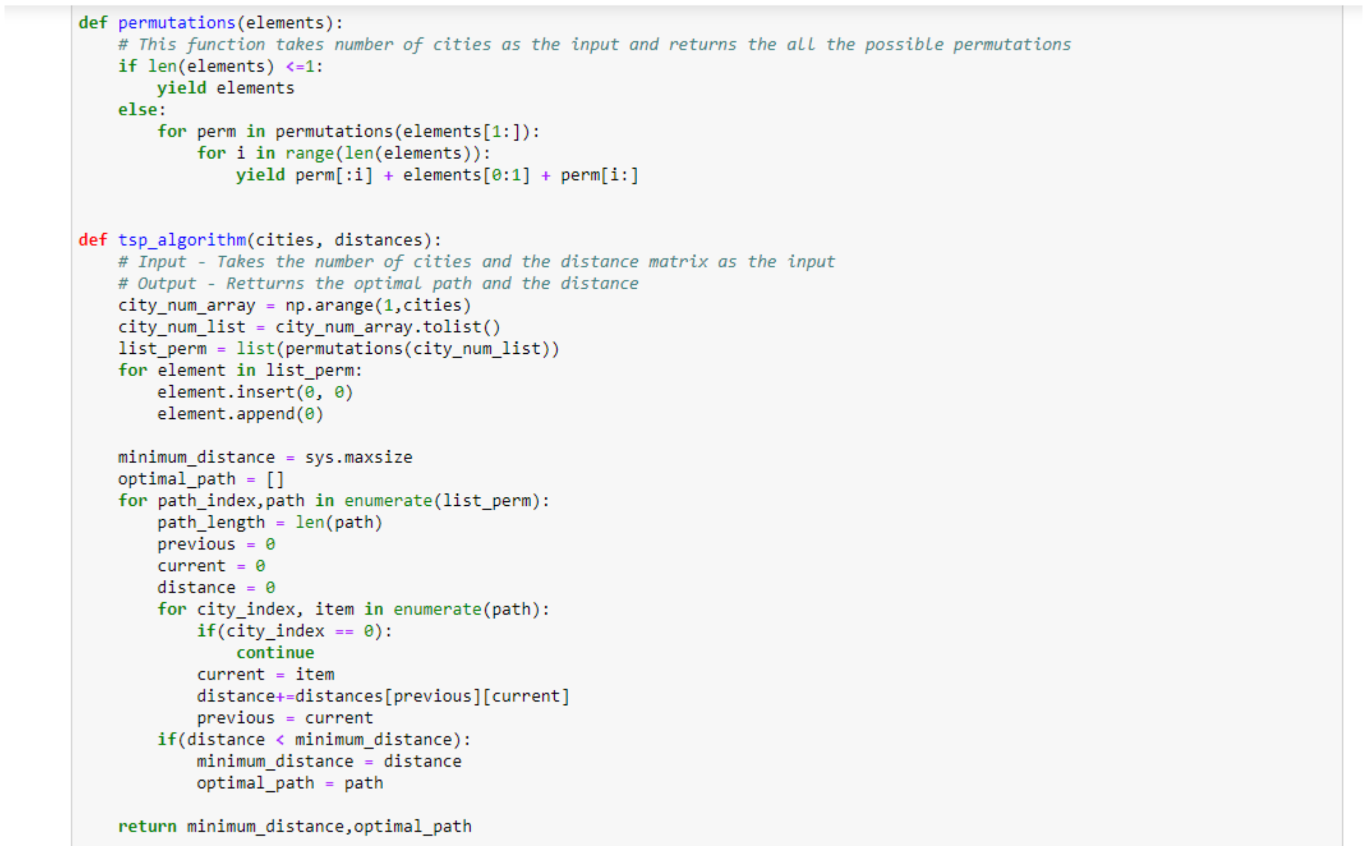
Travelling Salesman Problem



There are different algorithms to solve the travelling salesman problem. I selected the below algorithm

1. Considering city 0 as the starting and endpoint generate all the possible routes
2. Calculate the total distance of each and every route
3. Select the route with the minimum length as the optimal route

Python Code



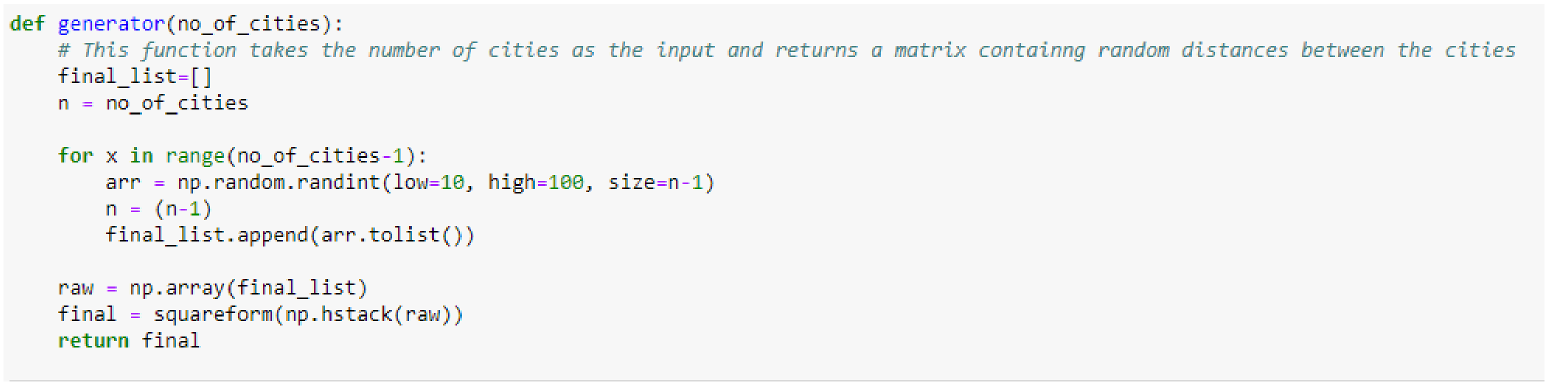
```python
def permutations(elements):
    # This function takes number of cities as the input and returns the all the possible permutations
    if len(elements) <=1:
        yield elements
    else:
        for perm in permutations(elements[1:]):
            for i in range(len(elements)):
                yield perm[:i] + elements[0:1] + perm[i:]


def tsp_algorithm(cities, distances):
    # Input - Takes the number of cities and the distance matrix as the input
    # Output - Retturns the optimal path and the distance
    city_num_array = np.arange(1,cities)
    city_num_list = city_num_array.tolist()
    list_perm = list(permutations(city_num_list))
    for element in list_perm:
        element.insert(0, 0)
        element.append(0)

    minimum_distance = sys.maxsize
    optimal_path = []
    for path_index,path in enumerate(list_perm):
        path_length = len(path)
        previous = 0
        current = 0
        distance = 0
        for city_index, item in enumerate(path):
            if(city_index == 0):
                continue
            current = item
            distance+=distances[previous][current]
            previous = current
        if(distance < minimum_distance):
            minimum_distance = distance
            optimal_path = path

    return minimum_distance,optimal_path
```

tsp_algorithm() – This is the implementation method of the algorithm. It takes number of cities and the distances between each city as inputs and returns the path with minimal distance. First it gets all the possible permutations by calling the permutations() method. Then for each route it calculates the total distance by using the distance matrix. Then it selects the optimal path with the minimum distance and returns the path and the distance.

permutations() – This is a helper method which generates all the possible paths that the salesperson can take. It will take the no. of cities as an input.

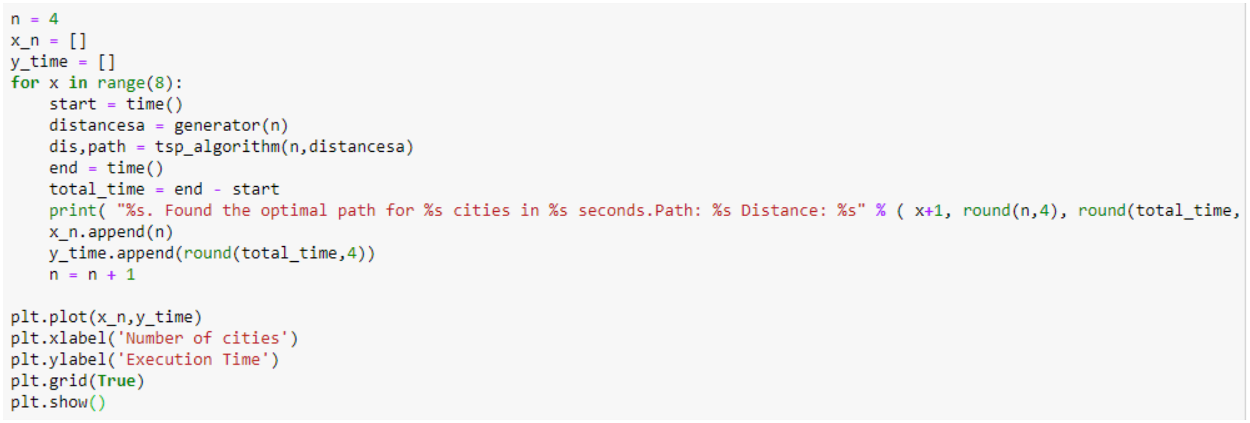
Execution time against number of cities

To demonstrate the variation of execution time against the no. of cities I used another helper method *generator()* to generate random distances between cities. This method takes no. of cities (n) as an input and returns a (n x n) matrix containing random distances.



```python
def generator(no_of_cities):
    # This function takes the number of cities as the input and returns a matrix containng random distances between the cities
    final_list=[]
    n = no_of_cities

    for x in range(no_of_cities-1):
        arr = np.random.randint(low=10, high=100, size=n-1)
        n = (n-1)
        final_list.append(arr.tolist())

    raw = np.array(final_list)
    final = squareform(np.hstack(raw))
    return final
```

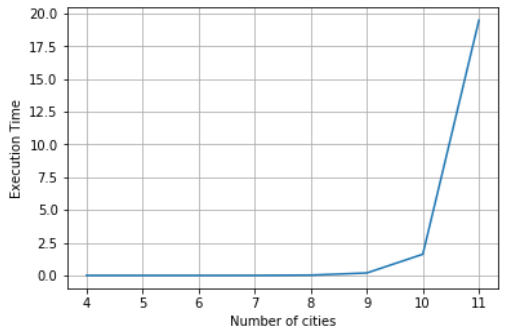
To visualize the execution time versus no of cities I used matplotlib library. Using a simple loop, I calculated the time taken to find the optimal path for different number of cities and plotted it in a graph.



```python
n = 4
x_n = []
y_time = []
for x in range(8):
    start = time()
    distancesa = generator(n)
    dis,path = tsp_algorithm(n,distancesa)
    end = time()
    total_time = end - start
    print( "%s. Found the optimal path for %s cities in %s seconds.Path: %s Distance: %s" % ( x+1, round(n,4), round(total_time,
    x_n.append(n)
    y_time.append(round(total_time,4))
    n = n + 1

plt.plot(x_n,y_time)
plt.xlabel('Number of cities')
plt.ylabel('Execution Time')
plt.grid(True)
plt.show()
```

Result

```
1. Found the optimal path for 4 cities in 0.001 seconds.Path: [0, 1, 2, 3, 0] Distance: 290
2. Found the optimal path for 5 cities in 0.0 seconds.Path: [0, 3, 4, 1, 2, 0] Distance: 184
3. Found the optimal path for 6 cities in 0.001 seconds.Path: [0, 4, 1, 2, 5, 3, 0] Distance: 236
4. Found the optimal path for 7 cities in 0.002 seconds.Path: [0, 2, 1, 5, 6, 3, 4, 0] Distance: 216
5. Found the optimal path for 8 cities in 0.022 seconds.Path: [0, 3, 4, 6, 1, 5, 2, 7, 0] Distance: 217
6. Found the optimal path for 9 cities in 0.1871 seconds.Path: [0, 5, 7, 6, 3, 1, 4, 8, 2, 0] Distance: 189
7. Found the optimal path for 10 cities in 1.6177 seconds.Path: [0, 2, 8, 9, 7, 4, 3, 6, 1, 5, 0] Distance: 325
8. Found the optimal path for 11 cities in 19.472 seconds.Path: [0, 7, 1, 9, 5, 8, 10, 4, 2, 3, 6, 0] Distance: 306
```



It was observed that when the no of cities increases, the execution time will also increase exponentially. When the no of cities exceeds 12 the program will crash and stop responding.