Travelling Salesman Problem Using OR Tools

13 random cities have been chosen and New York has been assigned as the depot arbitrarily.

```
def create_data_model():
    #Stores the data for the problem.
    data = {}
    data['distance_matrix'] = [
        [0, 2451, 713, 1018, 1631, 1374, 2408, 213, 2571, 875, 1420, 2145,
1972],
        [2451, 0, 1745, 1524, 831, 1240, 959, 2596, 403, 1589, 1374, 357,
579],
        [713, 1745, 0, 355, 920, 803, 1737, 851, 1858, 262, 940, 1453,
1260],
        [1018, 1524, 355, 0, 700, 862, 1395, 1123, 1584, 466, 1056, 1280,
987],
        [1631, 831, 920, 700, 0, 663, 1021, 1769, 949, 796, 879, 586, 371],
        [1374, 1240, 803, 862, 663, 0, 1681, 1551, 1765, 547, 225, 887,
999],
        [2408, 959, 1737, 1395, 1021, 1681, 0, 2493, 678, 1724, 1891, 1114,
701],
        [213, 2596, 851, 1123, 1769, 1551, 2493, 0, 2699, 1038, 1605, 2300,
2099],
        [2571, 403, 1858, 1584, 949, 1765, 678, 2699, 0, 1744, 1645, 653,
600],
        [875, 1589, 262, 466, 796, 547, 1724, 1038, 1744, 0, 679, 1272,
1162],
        [1420, 1374, 940, 1056, 879, 225, 1891, 1605, 1645, 679, 0, 1017,
1200],
        [2145, 357, 1453, 1280, 586, 887, 1114, 2300, 653, 1272, 1017, 0,
504],
        [1972, 579, 1260, 987, 371, 999, 701, 2099, 600, 1162, 1200, 504,
0],
    ]  # yapf: disable
```

```
    data['num_vehicles'] = 1
    data['depot'] = 0
    return data
```

The *distance matrix* is an array whose i, j entry is the distance from location i to location j in miles, where the array indices correspond to the locations in the following order:

0. New York - 1. Los Angeles - 2. Chicago - 3. Minneapolis - 4. Denver - 5. allas - 6. Seattle - 7. Boston - 8. San Francisco - 9. St. Louis - 10. Houston - 11. Phoenix - 12. Salt Lake City

Since it is TSP, we take that number of vehicles =1. Also, these distances could have been taken up as an input, taken up from the Google Maps Distance Matrix API, or calculated using the Euclidian formula.

Creating the routing model: -

```
data = create_data_model()
manager = pywrapcp.RoutingIndexManager(len(data['distance_matrix']),
                                       data['num_vehicles'], data['depot'])
routing = pywrapcp.RoutingModel(manager)
```

The inputs to `RoutingIndexManager` are: number of rows of the distance matrix, number of vehicles, depot node.

Creating the distance feedback (given 2 nodes, this gives the distance): -

```
def distance_callback(from_index, to_index):
    #Returns the distance between the two nodes.
    #Convert from routing variable Index to distance matrix NodeIndex.
    from_node = manager.IndexToNode(from_index)
    to_node = manager.IndexToNode(to_index)
    return data['distance_matrix'][from_node][to_node]

transit_callback_index = routing.RegisterTransitCallback(distance_callback)
```

Cost of travel: -

```
routing.SetArcCostEvaluatorOfAllVehicles(transit_callback_index)
#here we have cost as the distance.
```

The Heuristic Approach: -

```
search_parameters = pywrapcp.DefaultRoutingSearchParameters()
search_parameters.first_solution_strategy = (
    routing_enums_pb2.FirstSolutionStrategy.PATH_CHEAPEST_ARC)

def print_solution(manager, routing, solution):
    #Prints solution on console.
    print('Objective: {} miles'.format(solution.ObjectiveValue()))
    index = routing.Start(0)
    plan_output = 'Route for vehicle 0:\n'
    route_distance = 0
    while not routing.IsEnd(index):
        plan_output += ' {} ->'.format(manager.IndexToNode(index))
        previous_index = index
        index = solution.Value(routing.NextVar(index))
        route_distance += routing.GetArcCostForVehicle(previous_index,
index, 0)
    plan_output += ' {}\n'.format(manager.IndexToNode(index))
    print(plan_output)
    plan_output += 'Route distance: {}miles\n'.format(route_distance)
```

This creates an initial route by repeatedly adding edges with the least weight that don't lead to a previously visited node (other than the depot). The function displays the optimal route and its distance, which is given by `ObjectiveValue()`

Solving and printing the solution: -

```
solution = routing.SolveWithParameters(search_parameters)
if solution:
    print_solution(manager, routing, solution)
```

We get the output as 7293 miles and the route as 0->7->2->3->4->12->6->8->1->11->10->5->9->8.

We can save the routes to a list array and print it: -

```
def get_routes(solution, routing, manager):
  # Get vehicle routes from a solution and store them in an array.
  # Get vehicle routes and store them in a two dimensional array whose
  # i,j entry is the jth location visited by vehicle i along its route.
  routes = []
  for route_nbr in range(routing.vehicles()):
    index = routing.Start(route_nbr)
    route = [manager.IndexToNode(index)]
    while not routing.IsEnd(index):
      index = solution.Value(routing.NextVar(index))
      route.append(manager.IndexToNode(index))
    routes.append(route)
  return routes
```

To display the route: -

```
routes = get_routes(solution, routing, manager)
# Display the routes.
for i, route in enumerate(routes):
  print('Route', i, route)
```

The Complete TSP Code

```
from __future__ import print_function
from ortools.constraint_solver import routing_enums_pb2
from ortools.constraint_solver import pywrapcp



def create_data_model():
    #Stores the data for the problem.
    data = {}
    data['distance_matrix'] = [
        [0, 2451, 713, 1018, 1631, 1374, 2408, 213, 2571, 875, 1420, 2145,
1972],
        [2451, 0, 1745, 1524, 831, 1240, 959, 2596, 403, 1589, 1374, 357,
579],
        [713, 1745, 0, 355, 920, 803, 1737, 851, 1858, 262, 940, 1453,
1260],
        [1018, 1524, 355, 0, 700, 862, 1395, 1123, 1584, 466, 1056, 1280,
987],
        [1631, 831, 920, 700, 0, 663, 1021, 1769, 949, 796, 879, 586, 371],
        [1374, 1240, 803, 862, 663, 0, 1681, 1551, 1765, 547, 225, 887,
999],
        [2408, 959, 1737, 1395, 1021, 1681, 0, 2493, 678, 1724, 1891, 1114,
701],
        [213, 2596, 851, 1123, 1769, 1551, 2493, 0, 2699, 1038, 1605, 2300,
2099],
        [2571, 403, 1858, 1584, 949, 1765, 678, 2699, 0, 1744, 1645, 653,
600],
        [875, 1589, 262, 466, 796, 547, 1724, 1038, 1744, 0, 679, 1272,
1162],
        [1420, 1374, 940, 1056, 879, 225, 1891, 1605, 1645, 679, 0, 1017,
1200],
        [2145, 357, 1453, 1280, 586, 887, 1114, 2300, 653, 1272, 1017, 0,
504],
        [1972, 579, 1260, 987, 371, 999, 701, 2099, 600, 1162, 1200, 504,
0],
    ]  # yapf: disable
    data['num_vehicles'] = 1
    data['depot'] = 0
    return data


def print_solution(manager, routing, solution):
    #Prints solution on console.
    print('Objective: {} miles'.format(solution.ObjectiveValue()))
    index = routing.Start(0)
    plan_output = 'Route for vehicle 0:\n'
    route_distance = 0
    while not routing.IsEnd(index):
        plan_output += ' {} ->'.format(manager.IndexToNode(index))
        previous_index = index
        index = solution.Value(routing.NextVar(index))
        route_distance += routing.GetArcCostForVehicle(previous_index,
index, 0)
    plan_output += ' {}\n'.format(manager.IndexToNode(index))
    print(plan_output)
    plan_output += 'Route distance: {}miles\n'.format(route_distance)
```

```
def main():
    #Entry point of the program.
    # Instantiate the data problem.
    data = create_data_model()

    # Create the routing index manager.
    manager = pywrapcp.RoutingIndexManager(len(data['distance_matrix']),
                                           data['num_vehicles'],
data['depot'])

    # Create Routing Model.
    routing = pywrapcp.RoutingModel(manager)


    def distance_callback(from_index, to_index):
        #Returns the distance between the two nodes.
        # Convert from routing variable Index to distance matrix NodeIndex.
        from_node = manager.IndexToNode(from_index)
        to_node = manager.IndexToNode(to_index)
        return data['distance_matrix'][from_node][to_node]

    transit_callback_index =
routing.RegisterTransitCallback(distance_callback)

    # Define cost of each arc.
    routing.SetArcCostEvaluatorOfAllVehicles(transit_callback_index)

    # Setting first solution heuristic.
    search_parameters = pywrapcp.DefaultRoutingSearchParameters()
    search_parameters.first_solution_strategy = (
        routing_enums_pb2.FirstSolutionStrategy.PATH_CHEAPEST_ARC)

    # Solve the problem.
    solution = routing.SolveWithParameters(search_parameters)

    # Print solution on console.
    if solution:
        print_solution(manager, routing, solution)


if __name__ == '__main__':
    main()
```