CIVIL-557

Decision aid methodologies in transportation

Lab 4: TSP Heuristics

Tom Haering, Prunelle Vogler

Transport and Mobility Laboratory (TRANSP-OR)
École Polytechnique Fédérale de Lausanne (EPFL)

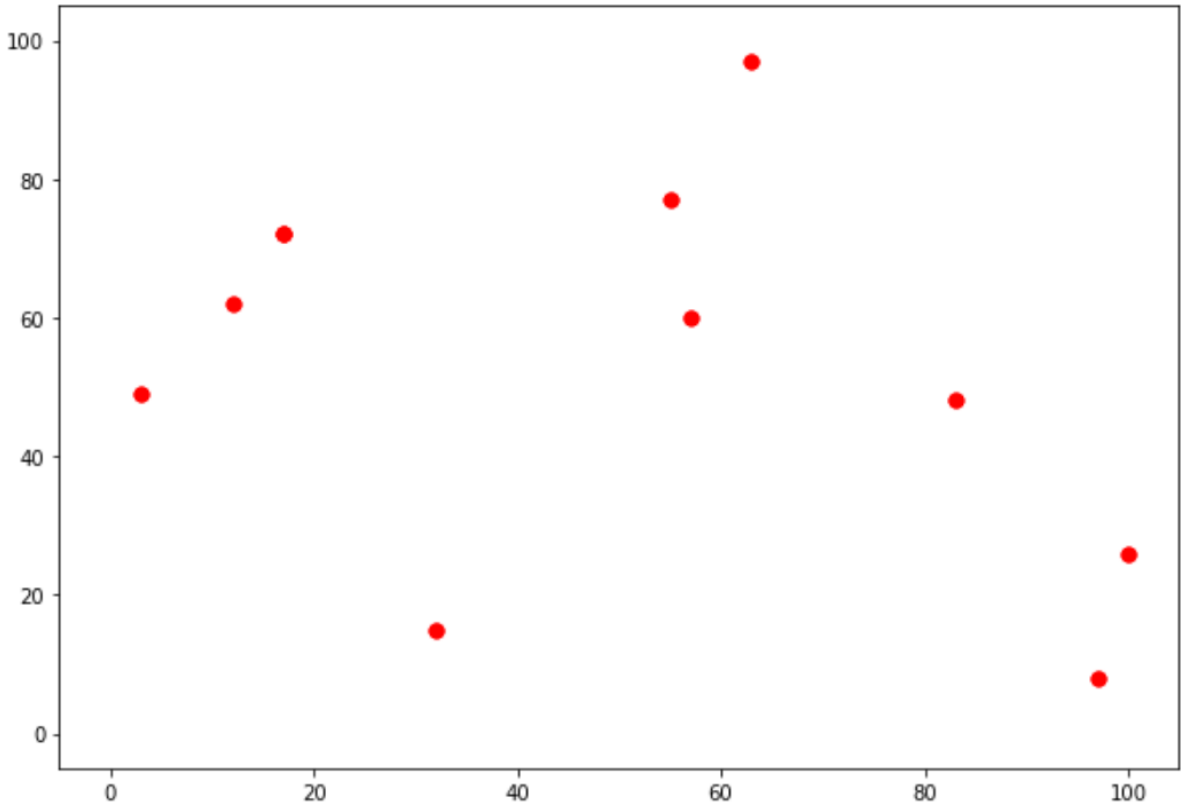
11.03.2025

Overview

- TSP recap
- Nearest Neighbour Heuristic
- 2opt Heuristic
- Random Insertion Heuristic
- Destroy and Rebuild Heuristic

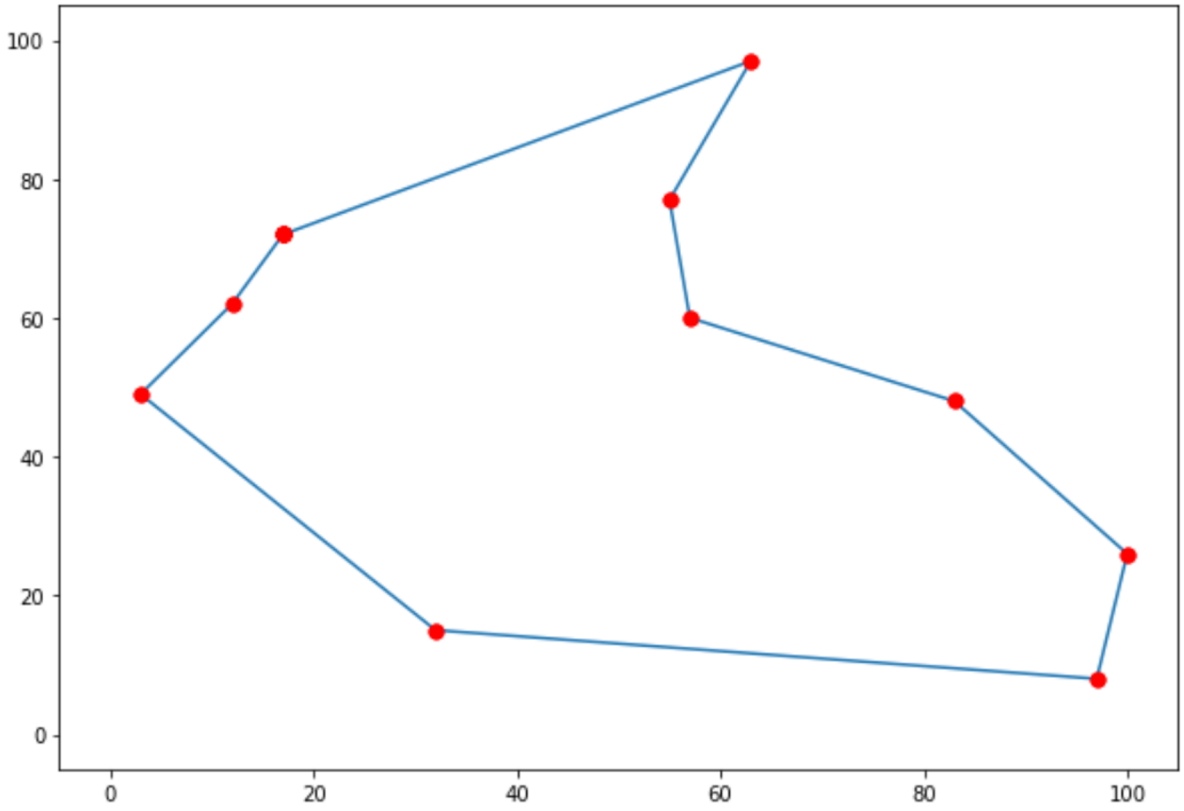
Traveling salesman problem

- **Input:** set of n **points** as (x, y) coordinates (or a *distance matrix*)
- **Goal:** find the **shortest tour** that **visits every point** and returns to the origin



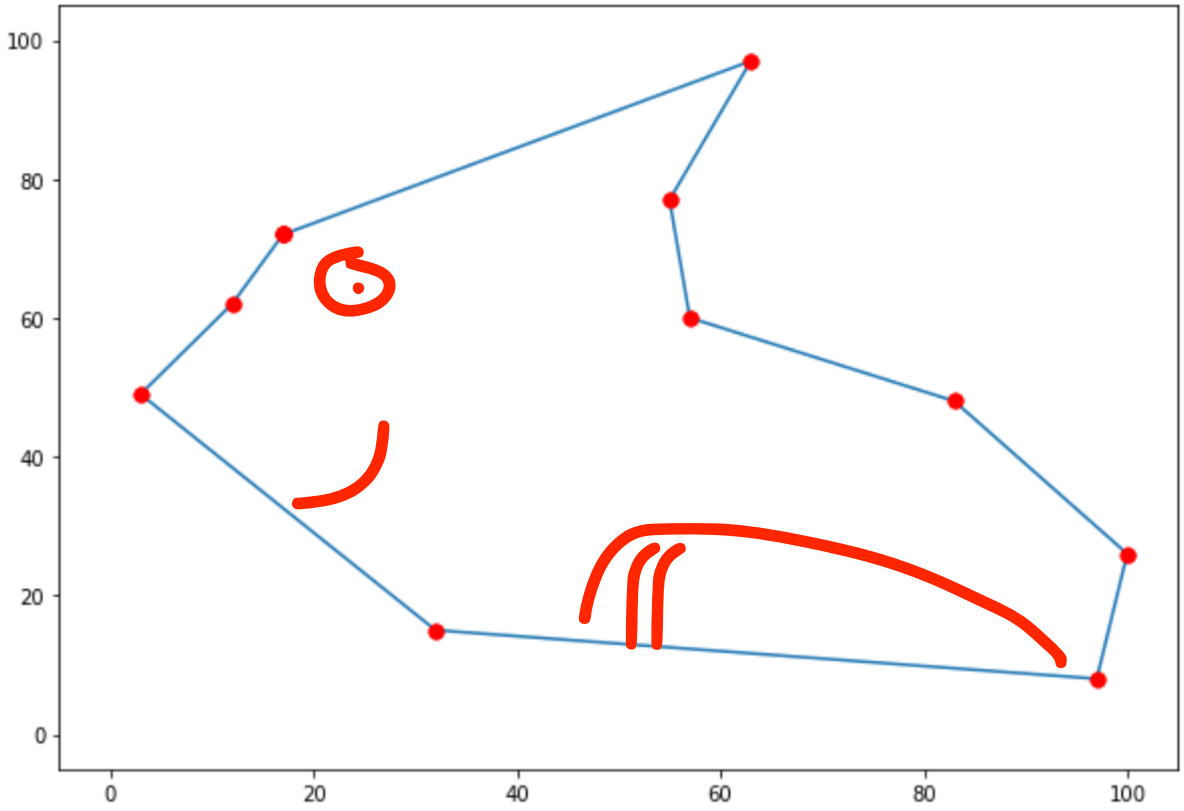
Traveling salesman problem

- **Input:** set of n **points** as (x, y) coordinates
- **Goal:** find the **shortest tour** that **visits every point** and returns to the origin



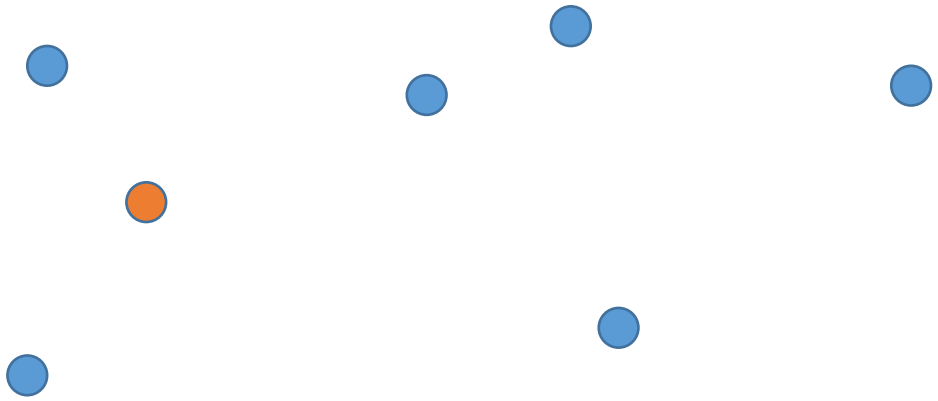
Traveling salesman problem

- **Input:** set of n **points** as (x, y) coordinates
- **Goal:** find the **shortest tour** that **visits every point** and returns to the origin



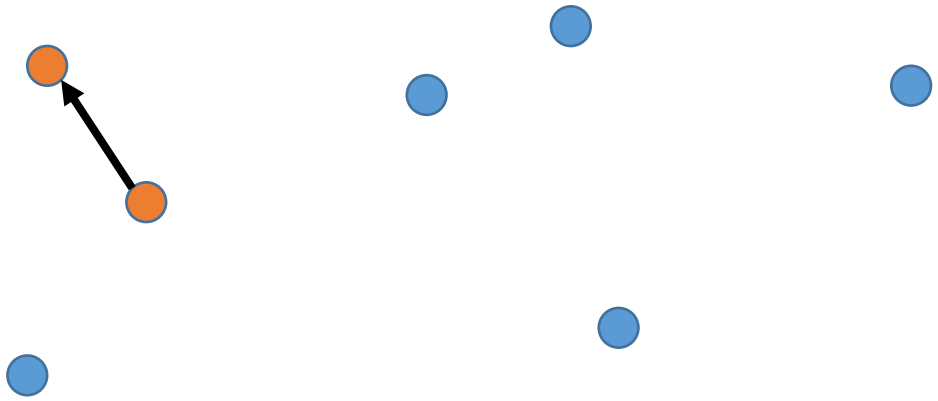
Nearest Neighbor Heuristic

- Start from city 0
- Find city that is closest to that city, add it to the route
- Continue until all cities are traversed
- At the end tie back to city 0



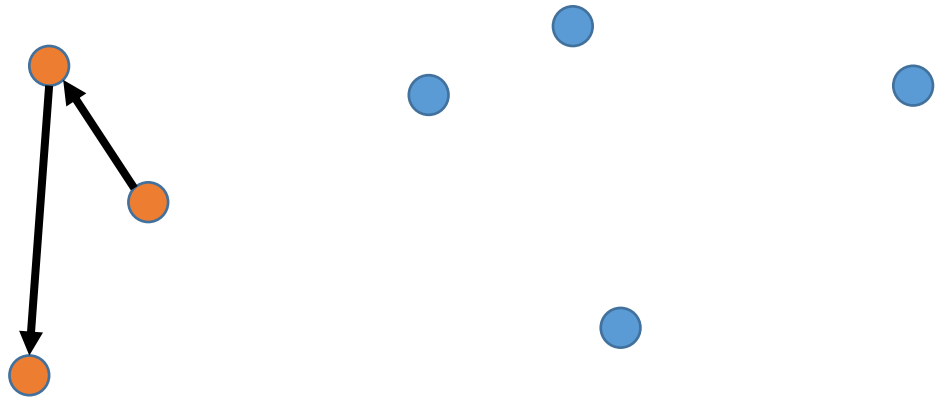
Nearest Neighbor Heuristic

- Start from city 0
- Find city that is closest to that city, add it to the route
- Continue until all cities are traversed
- At the end tie back to city 0



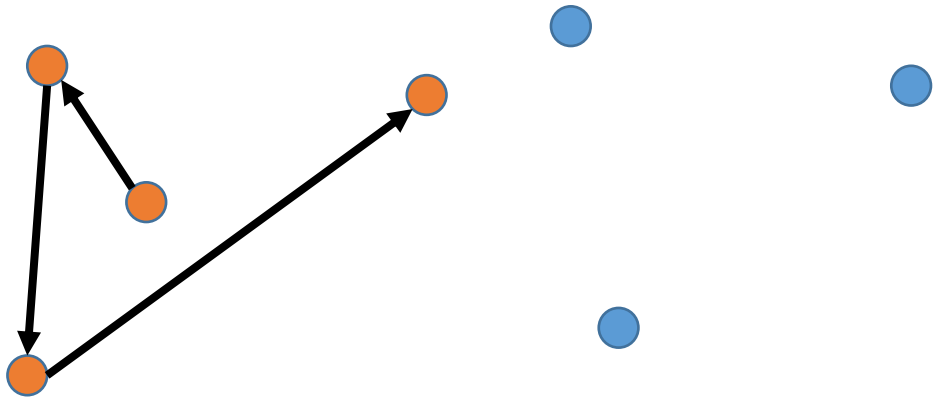
Nearest Neighbor Heuristic

- Start from city 0
- Find city that is closest to that city, add it to the route
- Continue until all cities are traversed
- At the end tie back to city 0



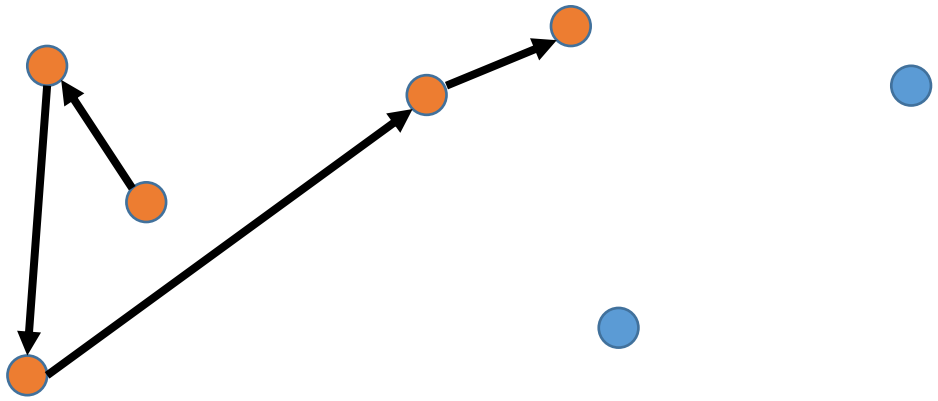
Nearest Neighbor Heuristic

- Start from city 0
- Find city that is closest to that city, add it to the route
- Continue until all cities are traversed
- At the end tie back to city 0



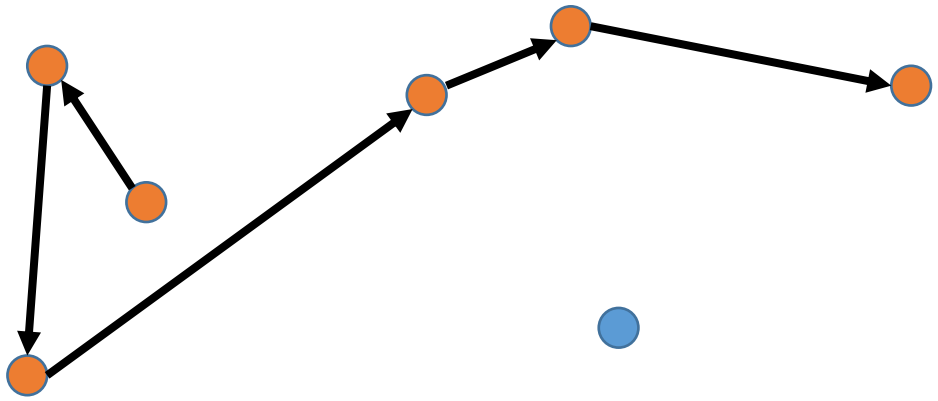
Nearest Neighbor Heuristic

- Start from city 0
- Find city that is closest to that city, add it to the route
- Continue until all cities are traversed
- At the end tie back to city 0



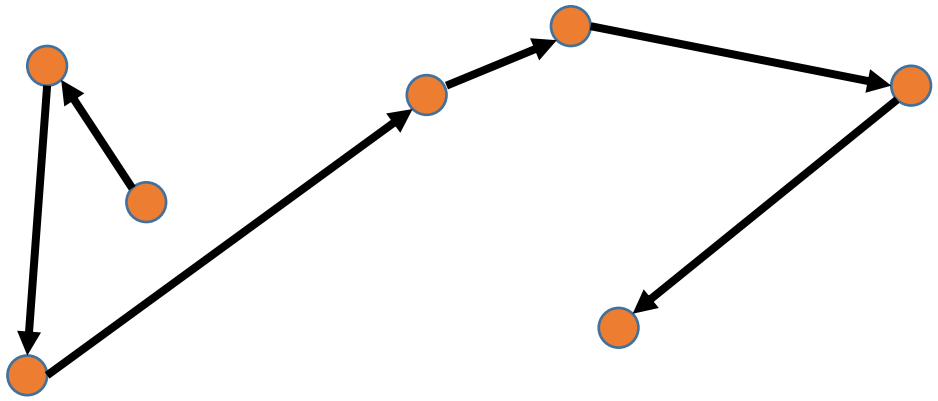
Nearest Neighbor Heuristic

- Start from city 0
- Find city that is closest to that city, add it to the route
- Continue until all cities are traversed
- At the end tie back to city 0



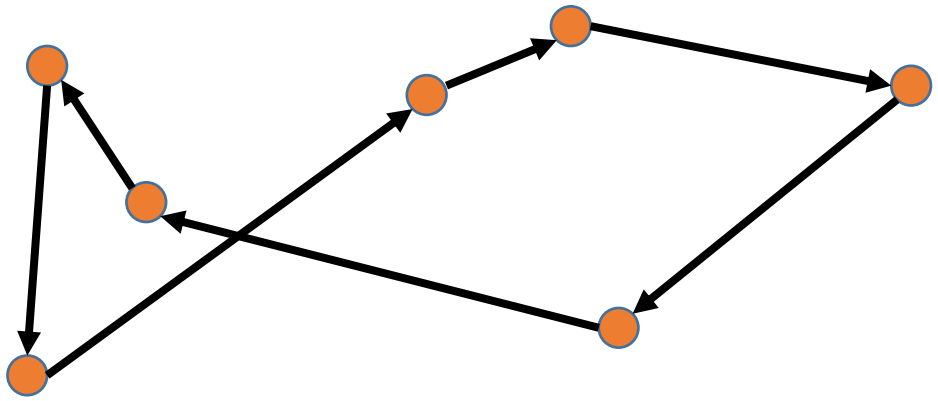
Nearest Neighbor Heuristic

- Start from city 0
- Find city that is closest to that city, add it to the route
- Continue until all cities are traversed
- At the end tie back to city 0



Nearest Neighbor Heuristic

- Start from city 0
- Find city that is closest to that city, add it to the route
- Continue until all cities are traversed
- At the end tie back to city 0

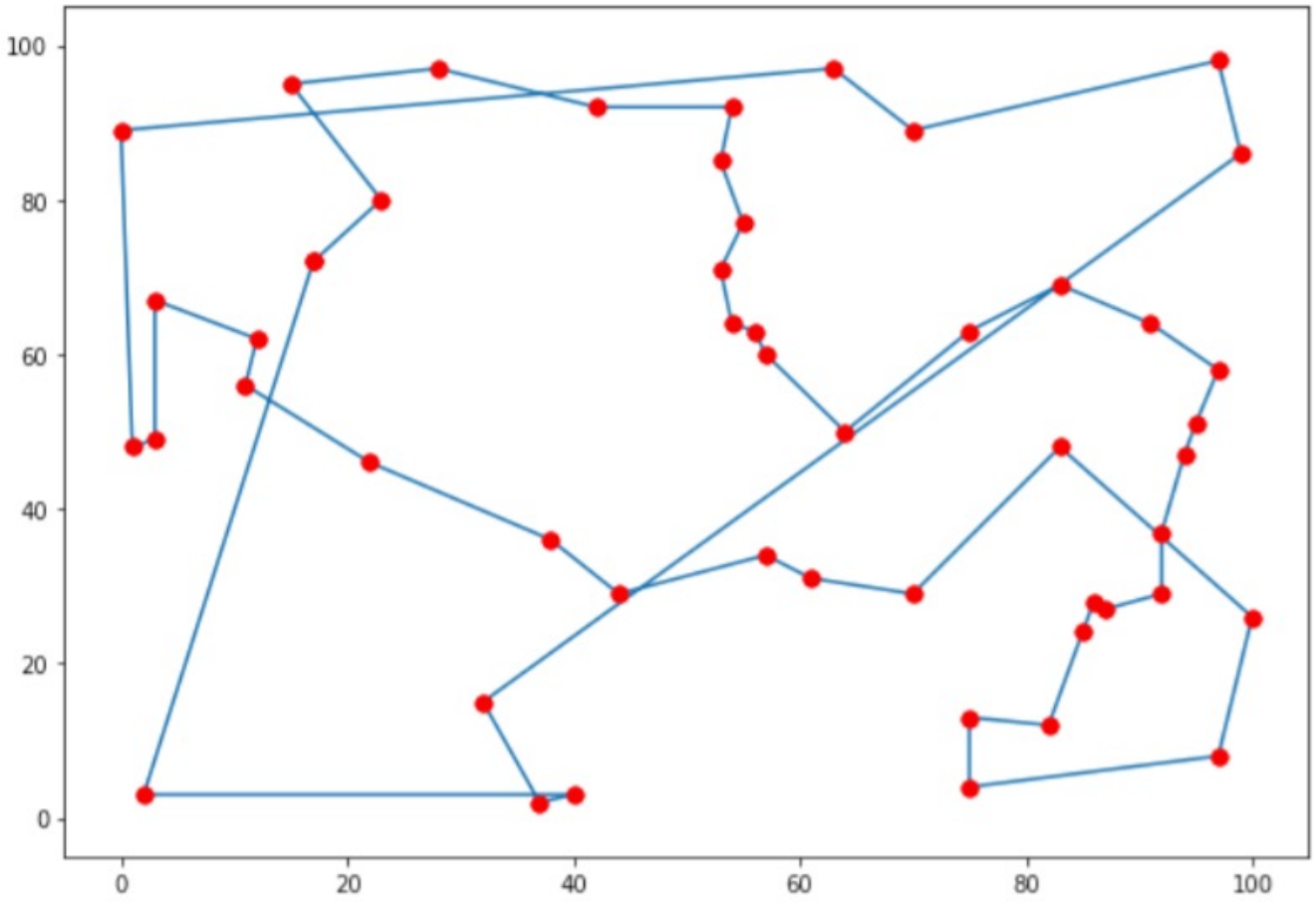


Nearest Neighbor Heuristic

```
num_cities = distance_matrix.shape[0]
route = [0]
unvisited_cities = set(range(1, num_cities))
while unvisited_cities:
    # take the city for which distance is minimized,
    # call it "nearest city"

    # append "nearest city" to route

    # remove "nearest city" from unvisited cities
route.append(0) # Return to starting city
return route
```

```
# distance between two cities i, j
distance_matrix[i, j]

# taking last element from route
route[-1]

# finding the element in a Set that minimizes a certain function:
min_element = min(Set, key=lambda element: function(element))

# add element to list
List.append(element)

# remove element from set
Set.remove(element)
```

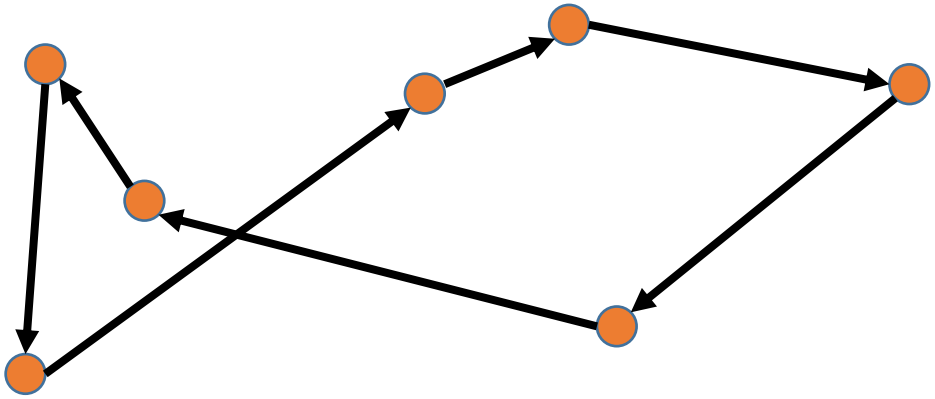
Nearest Neighbor Heuristic (N=50)




```
Length =  805.4974003341896
Solve time =  0.00044989585876464844
```

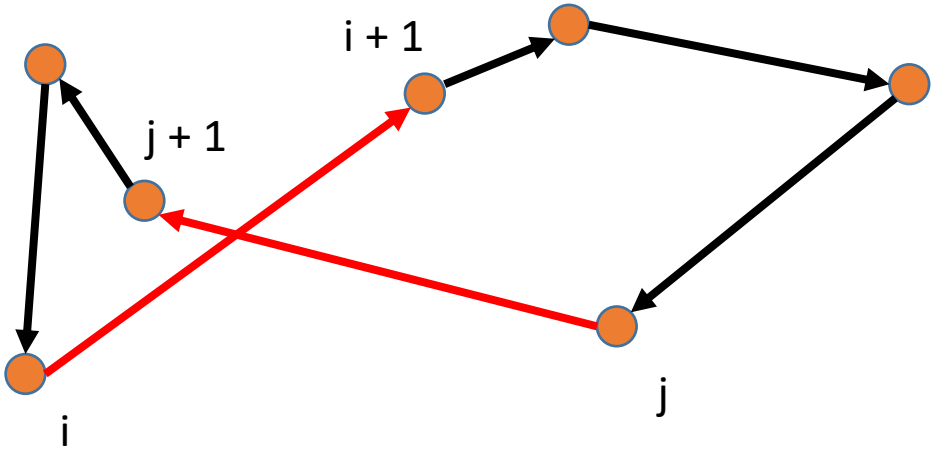
2opt Heuristic

- Start with a solution (ex. from nearest neighbor heuristic)
- Traverse all pairs of edges
- For every pair of edges, delete them, reverse the middle section, reconnect the reversed sequence to the other nodes
- If the new route is better, update the route, start again
- Else, continue looking for pairs of edges



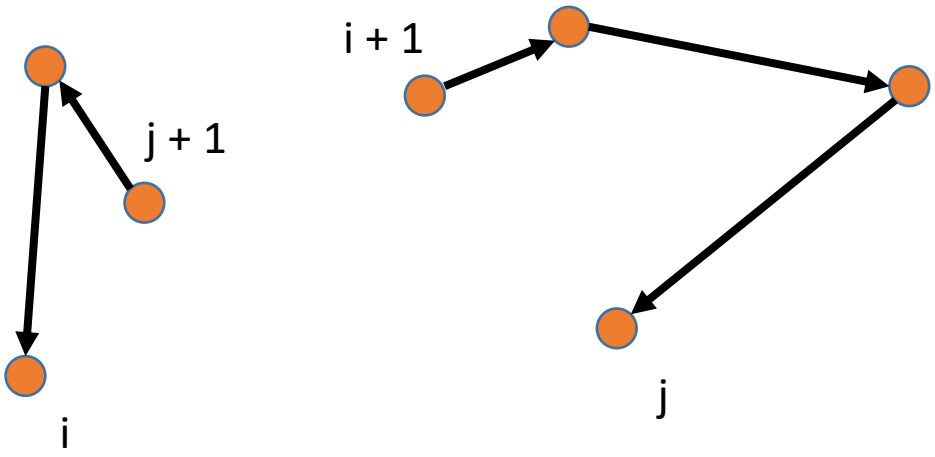
2opt Heuristic

- Start with a solution (ex. from nearest neighbor heuristic)
- Traverse all pairs of edges
- For every pair of edges, delete them, reverse the middle section, reconnect the reversed sequence to the other nodes
- If the new route is better, update the route, start again
- Else, continue looking for pairs of edges



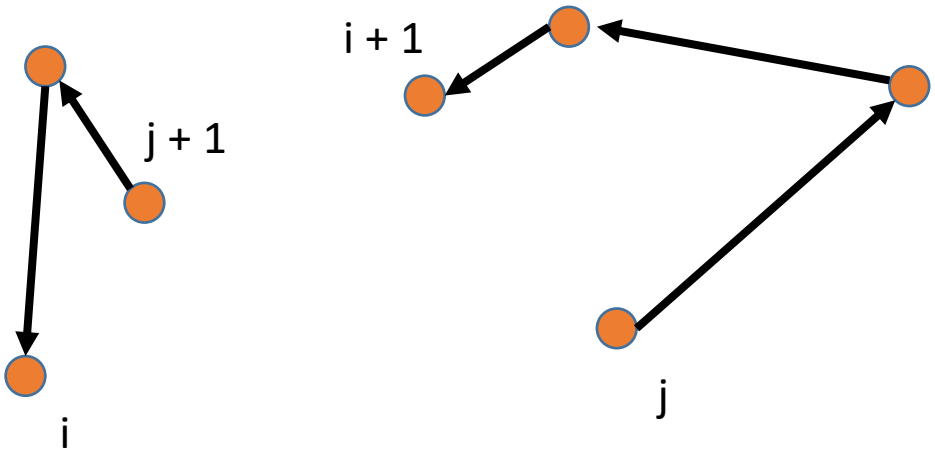
2opt Heuristic

- Start with a solution (ex. from nearest neighbor heuristic)
- Traverse all pairs of edges
- For every pair of edges, delete them, reverse the middle section, reconnect the reversed sequence to the other nodes
- If the new route is better, update the route, start again
- Else, continue looking for pairs of edges



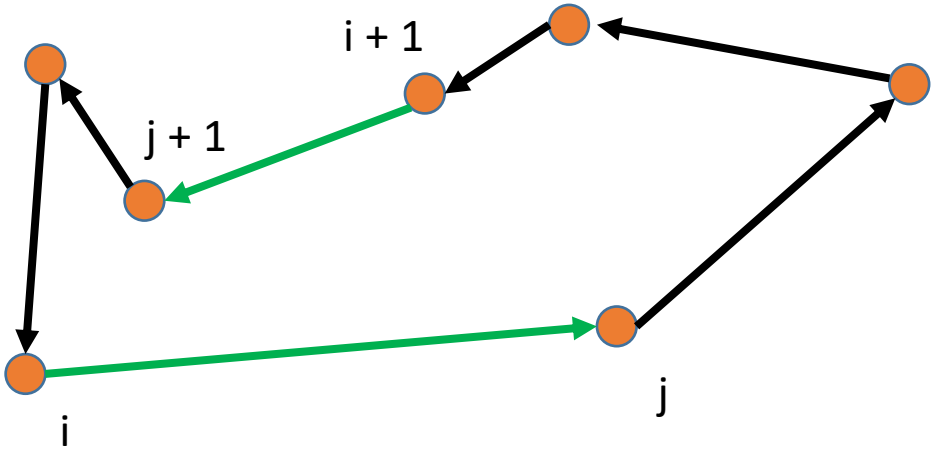
2opt Heuristic

- Start with a solution (ex. from nearest neighbor heuristic)
- Traverse all pairs of edges
- For every pair of edges, delete them, reverse the middle section, reconnect the reversed sequence to the other nodes
- If the new route is better, update the route, start again
- Else, continue looking for pairs of edges



2opt Heuristic

- Start with a solution (ex. from nearest neighbor heuristic)
- Traverse all pairs of edges
- For every pair of edges, delete them, reverse the middle section, reconnect the reversed sequence to the other nodes
- If the new route is better, update the route, start again
- Else, continue looking for pairs of edges

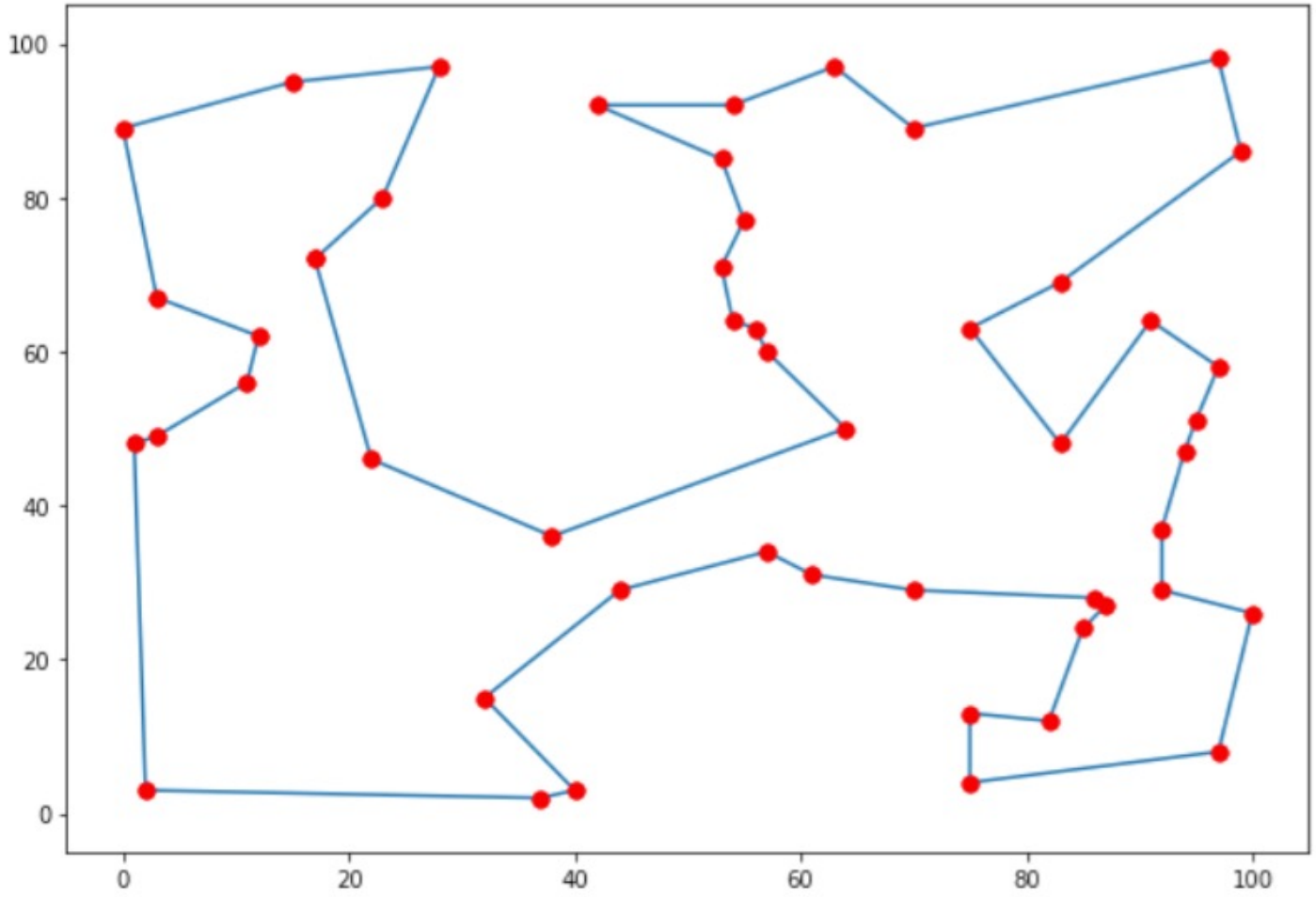


2opt Heuristic

```python
def reverse_segment(route, i, j):
    """Reverse the order of cities from index i to index j in a route."""
    new_route = route[:i] + route[i:j+1][::-1] + route[j+1:]
    return new_route

# Start with NN route
route = nearest_neighbour_tsp(distance_matrix)
size = len(route)
improved = True
best_distance = total_distance(route, distance_matrix)
while improved:
    best = total_distance(route, distance_matrix)
    improved = False
    # for all possible combinations of edges
    for i in range(size-2): # edge i to i+1
        for j in range(i+2, size-1): # edge j to j+1
            # Calculate gain: old edges - new edges
            # if gain > 0 then reverse the middle segment,
            # set improved to True, update "best_distance" variable
            # and exit the current j-loop (break)
return route
```

```python
# insert a list2 into another list1 at element i
new_list = list1[:i] + list2 + route[i+1:]

# invert a list1
new_list = list1[::-1]

# take all elements from index i to index j from a list1:
new_list = list1[i:j+1]

# exit a loop at any point
break

# shorter notation for x = x - y
x -= y
```

2opt Heuristic (N=50)

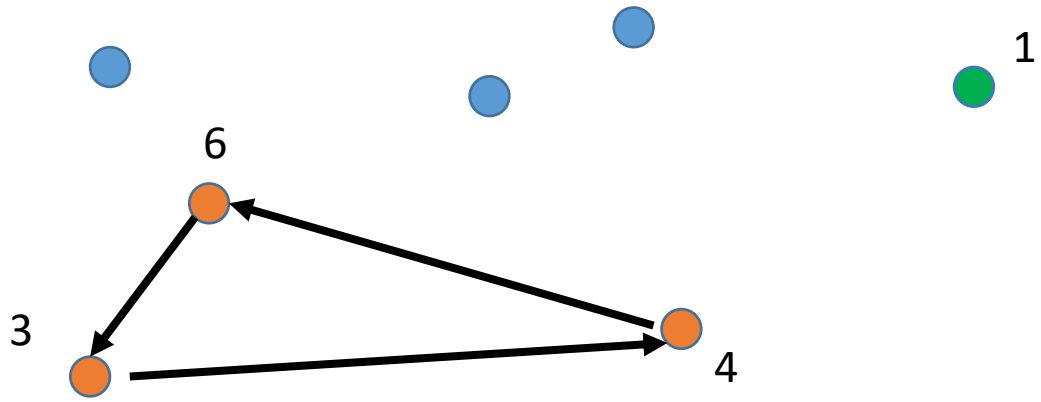



```
Length =  658.3492010827817
Solve time =  0.006730079650878906
```

Random insertion Heuristic

- Start with two randomly chosen cities
- Choose a random city from unvisited cities and compute the total length for every possible position where we could insert it in the current route
- Insert it where the total length is minimized

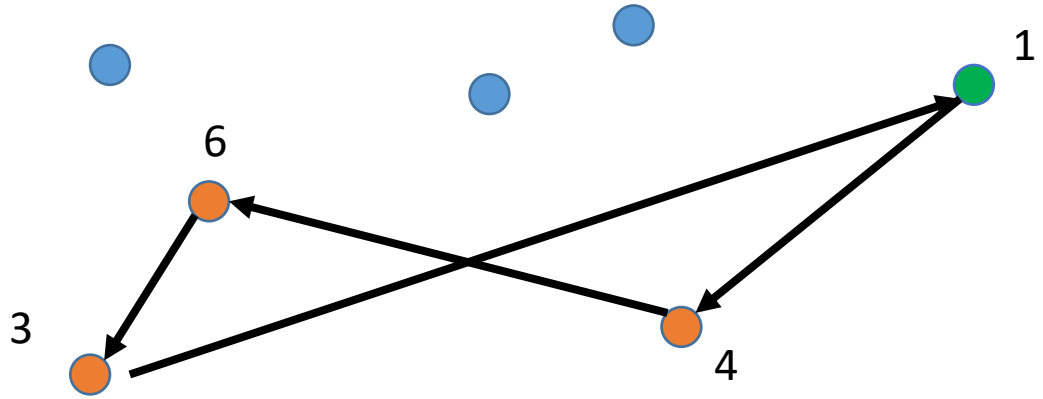
Route = [3, 4, 6, 3]



Random insertion Heuristic

- Start with two randomly chosen cities
- Choose a random city from unvisited cities and compute the total length for every possible position where we could insert it in the current route
- Insert it where the total length is minimized

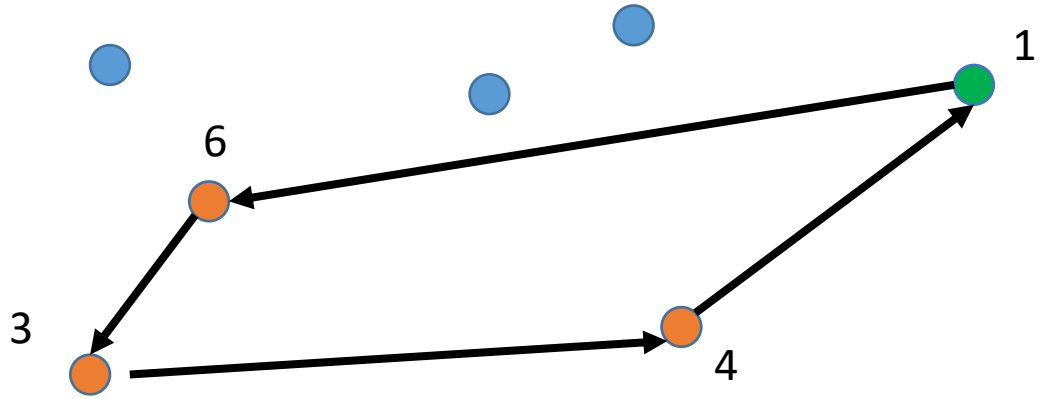
Route = [3, 4, 6, 3]
New = [3, 1, 4, 6, 3]



Random insertion Heuristic

- Start with two randomly chosen cities
- Choose a random city from unvisited cities and compute the total length for every possible position where we could insert it in the current route
- Insert it where the total length is minimized

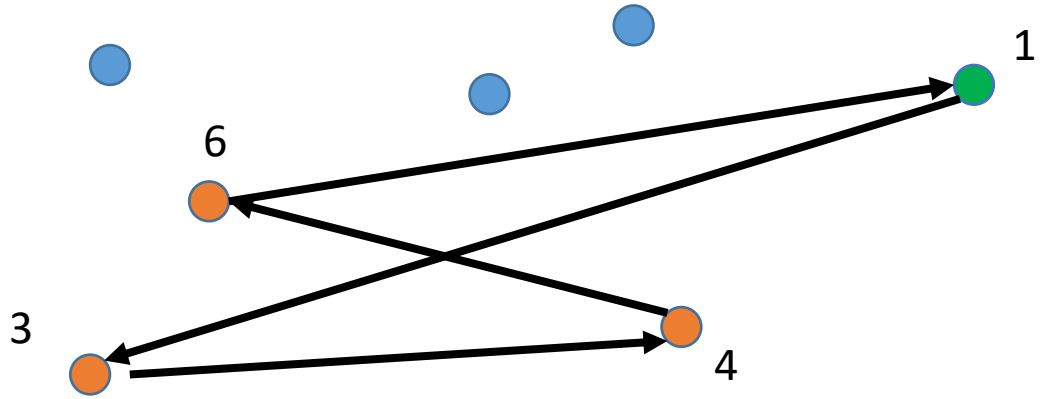
Route = [3, 4, 6, 3]
New = [3, 4, 1, 6, 3]



Random insertion Heuristic

- Start with two randomly chosen cities
- Choose a random city from unvisited cities and compute the total length for every possible position where we could insert it in the current route
- Insert it where the total length is minimized

Route = [3, 4, 6, 3]
New = [3, 4, 6, 1, 3]



Random insertion Heuristic

```python
# compute cost for inserting a city between two adjacent cities
def cost(i, j, city, distance_matrix):
    return distance_matrix[i, city] + distance_matrix[city, j] - distance_matrix[i, j]
```

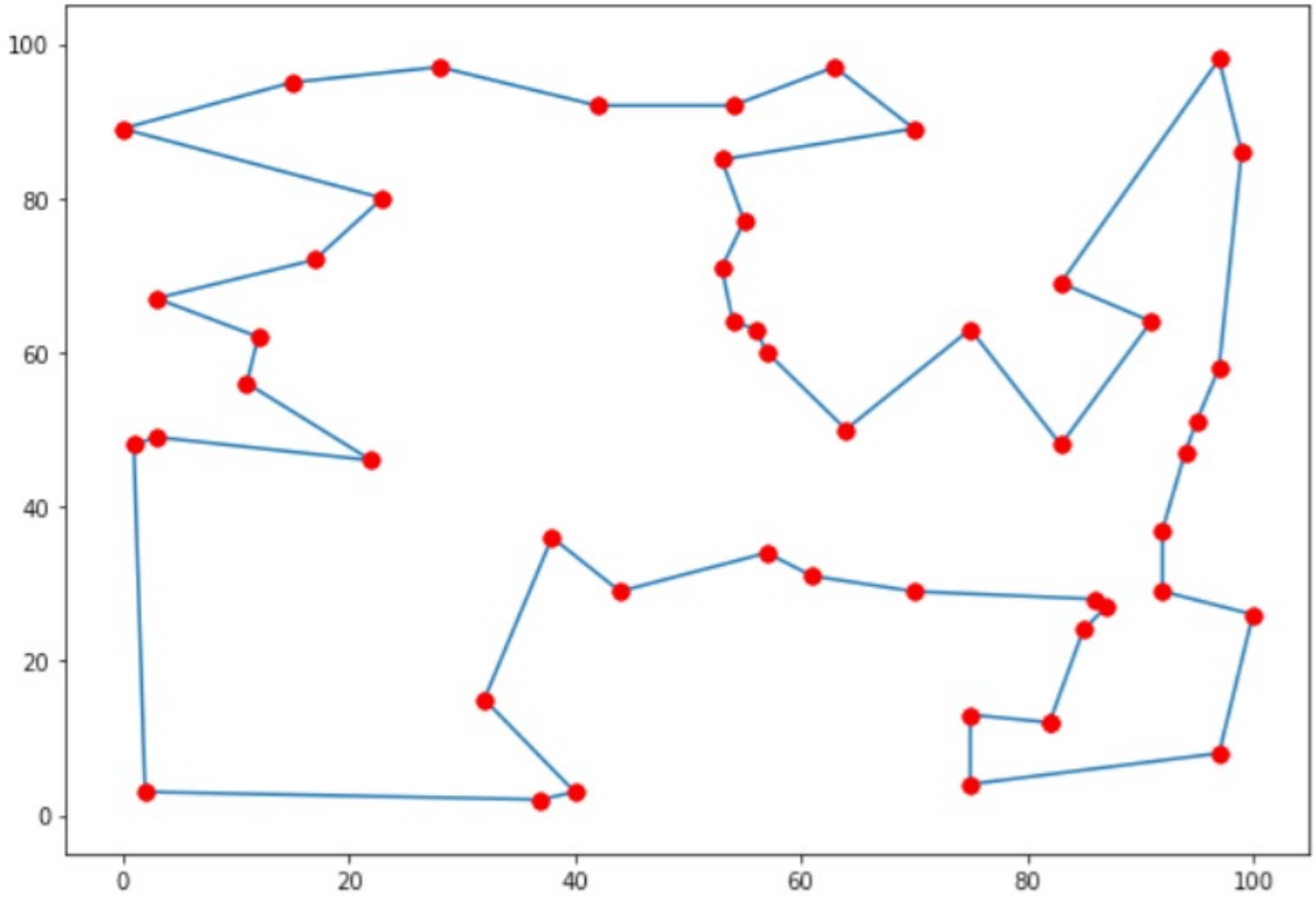
```python
random.seed(1)
num_cities = distance_matrix.shape[0]
unvisited_cities = set(range(num_cities))
start = random.choice(list(unvisited_cities)) # start from a random city
route = [start]
unvisited_cities.remove(start)
while unvisited_cities:
    # choose random city amongst unvisited ones
    # if the route so far contains 1 city we simply add the new city, else:
    # find insertion position that minimizes added cost when inserted
    # between i-1 and i, for all i in the route
    # insert the random city at the index
    # remove the random city from the unvisited cities
route.append(route[0])  # Return to starting city
return route
```

```python
# choose a random element from a set:
random_element = random.choice(list(set))

# find the index that minimizes a function over a list "List"
index = min(list(range(len(List))), key=lambda i: function(i))

# insert an element into a list at a specific index
List.insert(insertion_index, element)

# remove an element from a list
List.remove(element)
```

TRANSP-OR

EPFL

Random insertion Heuristic




```
Length =  654.9459660024927
Solve time =  0.0012331008911132812
```

Destroy and rebuild Metaheuristic

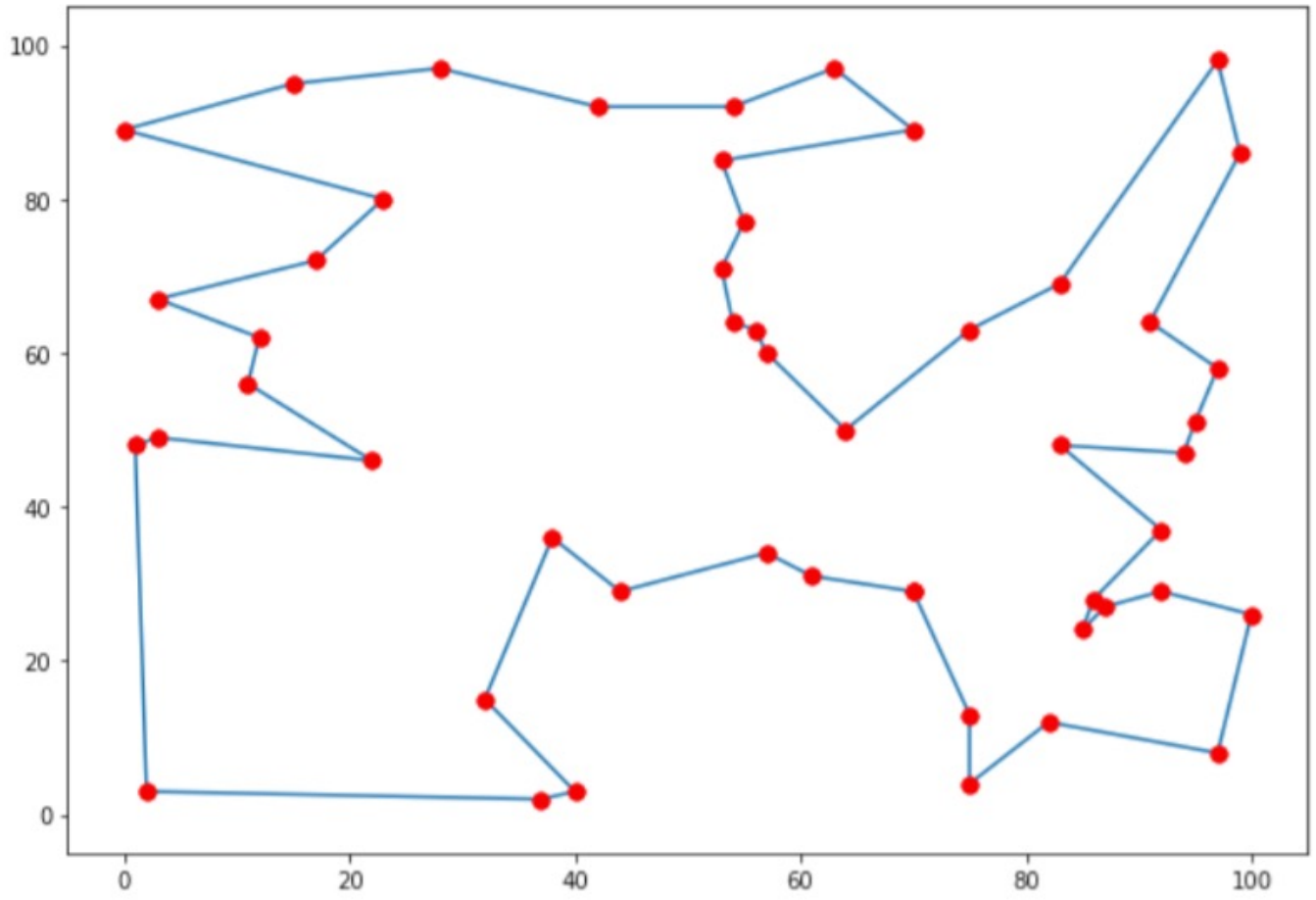
- Construct tour using random insertion
- Randomly remove half the cities of the tour
- Re-insert the removed cities using for example the random insertion heuristic
- Repeat M times, for example M = 10000

Destroy and rebuild Metaheuristic

```
num_cities = distance_matrix.shape[0]
half_num_cities = round(num_cities/2)
# create a first tour by using random insertion
route = random_insertion_tsp(distance_matrix)
best_tour = route
best_length = total_distance(route, distance_matrix)
for i in range(num_iterations):
    # remove randomly half of the cities
    # use random insertion method as before to fill up the tour to full length
    # (just copy paste the while loop)
    # check if these new found route is shorter than the best route so far
    # (using total_distance(route, distance_matrix))
    # if yes, update the variables "best_tour" and "best_length"
return best_tour
```

```
# take a random sample of size H from a list
sample = random.sample(List, H)

# list comprehension: define a list buy manipulating elements from another list
# example: list1 = all elements in list2 that are not in list3
list1 = [el for el in list2 if el not in list3]
```

Destroy and rebuild Metaheuristic (N=50, 1000 iter)




```
Length =  635.5025787588411
Solve time =  0.6271371841430664
```