Assignment 2A - Hill Climbing

In this assignment, you will implement the necessary data structures and subroutines to run hill climbing on traveling salesman problems.

You are given four files

1. This Notebook - You will be running the cells in this file.
2. sa_utils.py. You should not change anything in this file.
3. tsp_utils.py. You will be writing most of the code in this file.
4. berlin52.tsp. A TSP file, downloaded from http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/

TODO

Enter your information below.

Name: ...
CWID: ...

In [1]:
```
import numpy as np

import pandas as pd

from matplotlib import pylab
import matplotlib.pyplot as plt
pylab.rcParams['figure.figsize'] = (10.0, 8.0)

from sa_utils import Node
from sa_utils import hill_climbing
```

In [2]:
```
from tsp_utils import City, TSPNode, read_cities, subsample_cities, create_initial_
from tsp_utils import plot_cities, plot_path, compare_sols
```

Reading the file

TODO: Implement the `read_cities` function in `tsp_utils.py`. This function should read a given TSP file from http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/ Check out the documentation file. Your function should support only the EUC_2D types of files. See `berlin52.tsp` as an example.

`read_cities` should accept a string (filename) and return a dictionary of the City objects, where the key is the City name and the objects are City objects with the correct coordinates.

In [3]:
```
# Run. It should show the dictionary.
```

```
all_cities = read_cities('berlin52.tsp')
TSPNode._cities = all_cities

all_cities
```

Out[3]:
```
{'1': City: 1 (565.00 575.00),
 '2': City: 2 (25.00 185.00),
 '3': City: 3 (345.00 750.00),
 '4': City: 4 (945.00 685.00),
 '5': City: 5 (845.00 655.00),
 '6': City: 6 (880.00 660.00),
 '7': City: 7 (25.00 230.00),
 '8': City: 8 (525.00 1000.00),
 '9': City: 9 (580.00 1175.00),
 '10': City: 10 (650.00 1130.00),
 '11': City: 11 (1605.00 620.00),
 '12': City: 12 (1220.00 580.00),
 '13': City: 13 (1465.00 200.00),
 '14': City: 14 (1530.00 5.00),
 '15': City: 15 (845.00 680.00),
 '16': City: 16 (725.00 370.00),
 '17': City: 17 (145.00 665.00),
 '18': City: 18 (415.00 635.00),
 '19': City: 19 (510.00 875.00),
 '20': City: 20 (560.00 365.00),
 '21': City: 21 (300.00 465.00),
 '22': City: 22 (520.00 585.00),
 '23': City: 23 (480.00 415.00),
 '24': City: 24 (835.00 625.00),
 '25': City: 25 (975.00 580.00),
 '26': City: 26 (1215.00 245.00),
 '27': City: 27 (1320.00 315.00),
 '28': City: 28 (1250.00 400.00),
 '29': City: 29 (660.00 180.00),
 '30': City: 30 (410.00 250.00),
 '31': City: 31 (420.00 555.00),
 '32': City: 32 (575.00 665.00),
 '33': City: 33 (1150.00 1160.00),
 '34': City: 34 (700.00 580.00),
 '35': City: 35 (685.00 595.00),
 '36': City: 36 (685.00 610.00),
 '37': City: 37 (770.00 610.00),
 '38': City: 38 (795.00 645.00),
 '39': City: 39 (720.00 635.00),
 '40': City: 40 (760.00 650.00),
 '41': City: 41 (475.00 960.00),
 '42': City: 42 (95.00 260.00),
 '43': City: 43 (875.00 920.00),
 '44': City: 44 (700.00 500.00),
 '45': City: 45 (555.00 815.00),
 '46': City: 46 (830.00 485.00),
 '47': City: 47 (1170.00 65.00),
 '48': City: 48 (830.00 610.00),
 '49': City: 49 (605.00 625.00),
 '50': City: 50 (595.00 360.00),
 '51': City: 51 (1340.00 725.00),
 '52': City: 52 (1740.00 245.00)}
```

Subsample Cities

TODO: Complete the implementation of the `subsample_cities` function in `tsp_utils.py`. The arguments are

- `cities`: the dictionary of the cities
- `number_of_cities`: the number of cities in the subsample
- `random_seed`: the random seed used to create the subsample

It should return a new dictionary of cities.

In [4]:
```
# Run

subsample_size = 10
subsample_seed = 2

cities = subsample_cities(all_cities, number_of_cities=subsample_size, random_seed=

cities
```

Out[4]:
```
{'15': City: 15 (845.00 680.00),
 '26': City: 26 (1215.00 245.00),
 '40': City: 40 (760.00 650.00),
 '46': City: 46 (830.00 485.00),
 '16': City: 16 (725.00 370.00),
 '45': City: 45 (555.00 815.00),
 '48': City: 48 (830.00 610.00),
 '4': City: 4 (945.00 685.00),
 '14': City: 14 (1530.00 5.00),
 '6': City: 6 (880.00 660.00)}
```

Implement TSPNode

TODO: Complete the implementation of the `TSPNode` class. You need to implement

- `expand` This should create all possible children of this node, where a child is a swap of two neighbor cities. A state is an ordered list of city names to visit. Remember that the last city travels back to the start city and hence they are also neighbors.
- `value` This is the negative of the cost of the state. The cost of the state is the sum of the distances between the neighor cities. The distance between two neighbors is the Eucledian distance (square root of the sum of the squares of the differences).

In [5]:
```
# Run

tsp_node = TSPNode(sorted(list(cities.keys())))
tsp_node
```

Out[5]:
```
TSPNode: 14-15-16-26-4-40-45-46-48-6
```

In [6]:
```
# Run
```

```
children_nodes = tsp_node.expand()
len(children_nodes)
```

Out[6]: 10

In [7]:
```
# Run

children_nodes
```

Out[7]:
```
[TSPNode: 15-14-16-26-4-40-45-46-48-6,
 TSPNode: 14-16-15-26-4-40-45-46-48-6,
 TSPNode: 14-15-26-16-4-40-45-46-48-6,
 TSPNode: 14-15-16-4-26-40-45-46-48-6,
 TSPNode: 14-15-16-26-40-4-45-46-48-6,
 TSPNode: 14-15-16-26-4-45-40-46-48-6,
 TSPNode: 14-15-16-26-4-40-46-45-48-6,
 TSPNode: 14-15-16-26-4-40-45-48-46-6,
 TSPNode: 14-15-16-26-4-40-45-46-6-48,
 TSPNode: 6-15-16-26-4-40-45-46-48-14]
```

In [8]:
```
tsp_node.value()
```

Out[8]: -4315.526779689034

Create a random start state

In [9]:
```
# Run
initial_seed = 5

initial_node = create_initial_node(cities, random_seed=initial_seed)

initial_node
```

Out[9]: TSPNode: 4-48-26-46-40-16-15-6-45-14

In [10]:
```
# Run
initial_node.value()
```

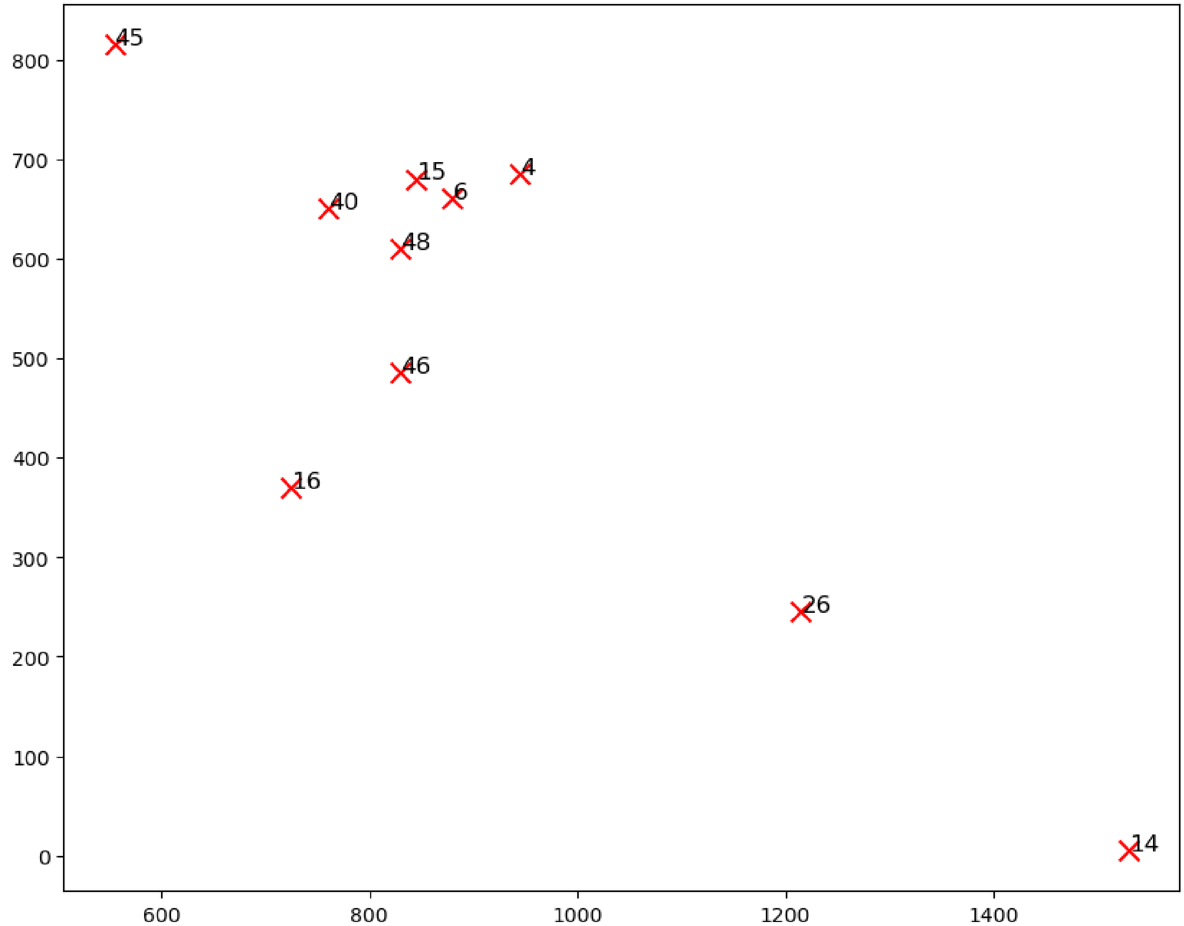
Out[10]: -4480.278437200555

Implement Plot Functions

TODO: Implement

- `plot_cities` Given an matplotlib axes, a dictionary of cities, and a state, it should plot the cities in the state. The cities should be plotted at their coordinates.

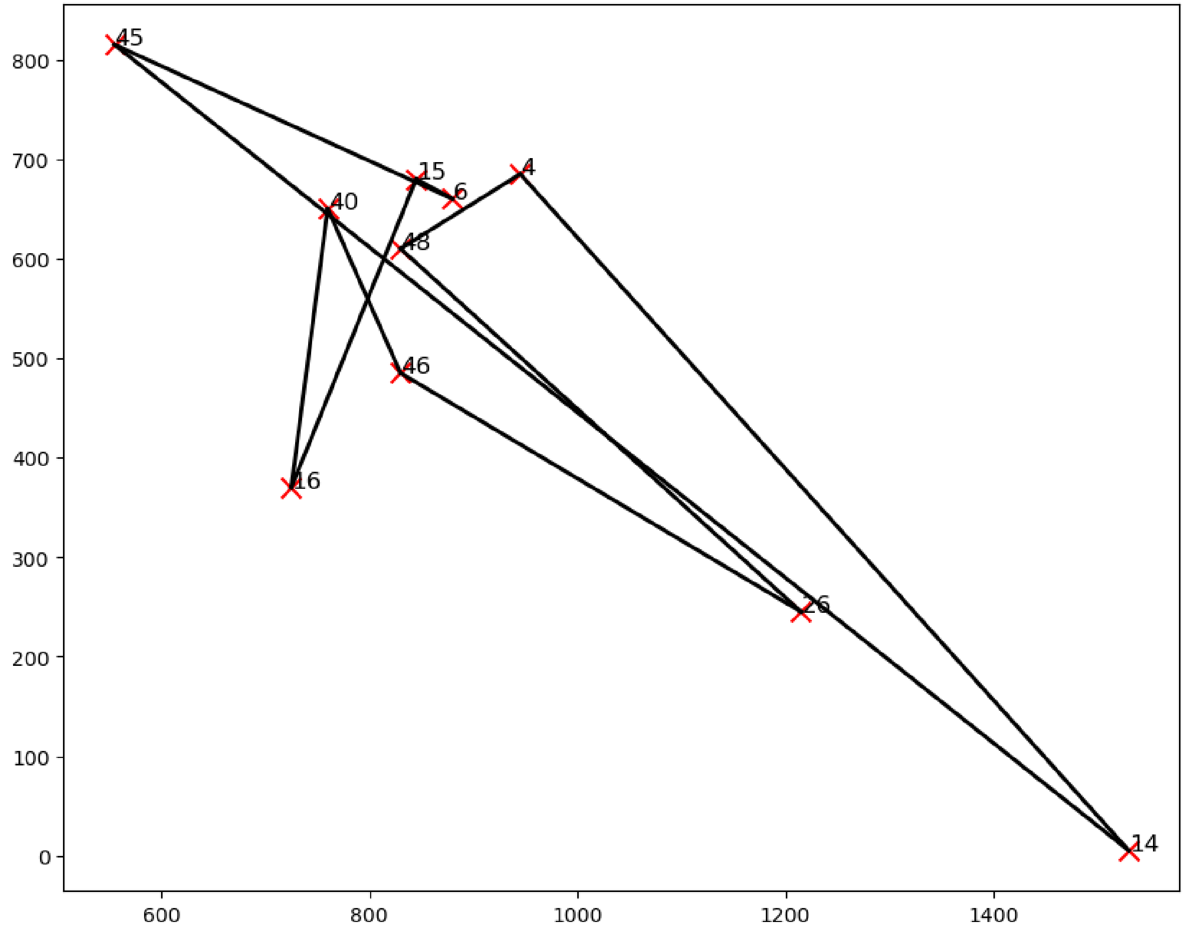
In [11]:
```
# Run
fig, ax = plt.subplots()
plot_cities(ax, all_cities, initial_node.state)
```



TODO: Implement

- `plot_path` Given an matplotlib axes, a dictionary of cities, and a state, it should plot the edges between the cities.

In [12]:
```
# Run
fig, ax = plt.subplots()
plot_cities(ax, all_cities, initial_node.state)
plot_path(ax, cities, initial_node.state)
```



Run Hill Climbing

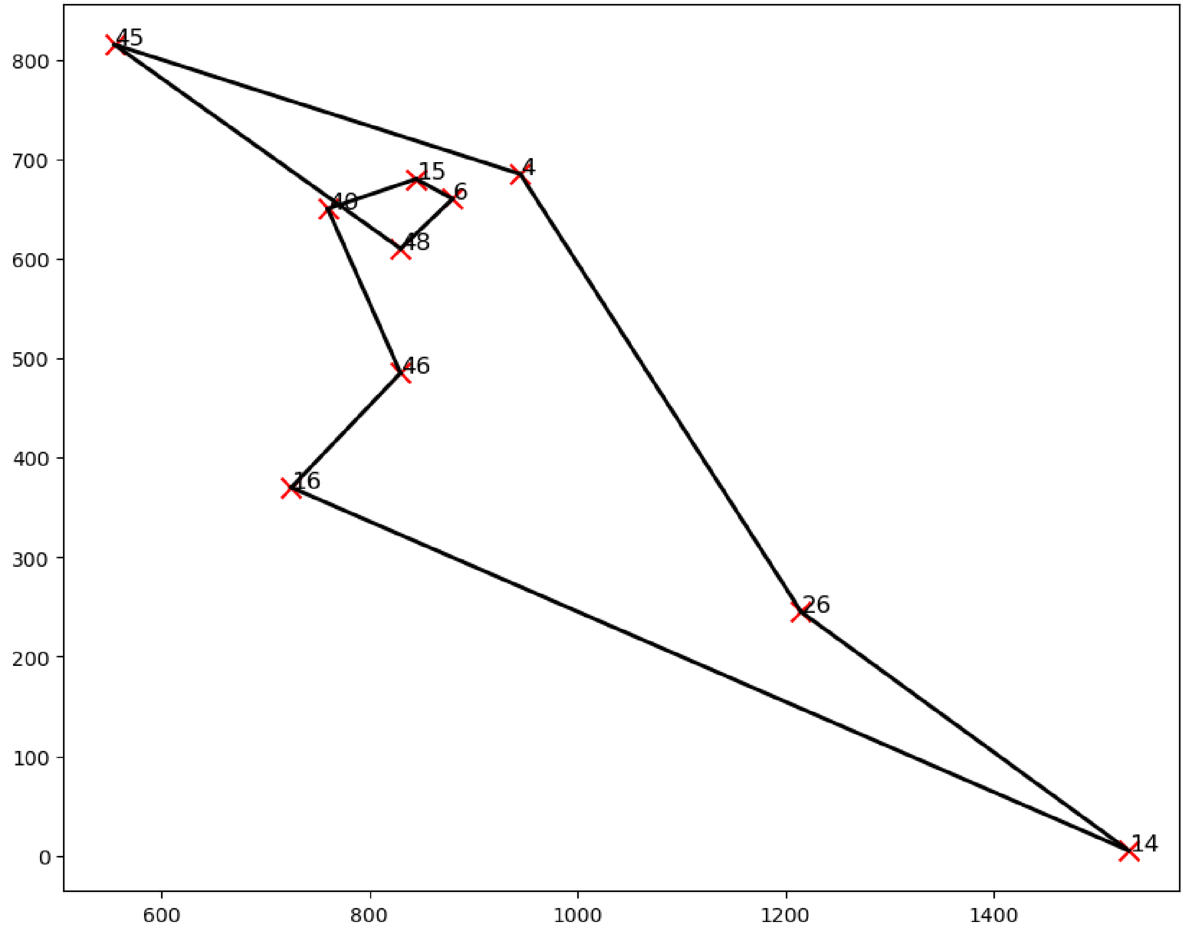
In [13]:
```
# Run
hc_sol_node = hill_climbing(initial_node)
hc_sol_node
```

Out[13]: TSPNode: 4-26-14-16-46-40-15-6-48-45

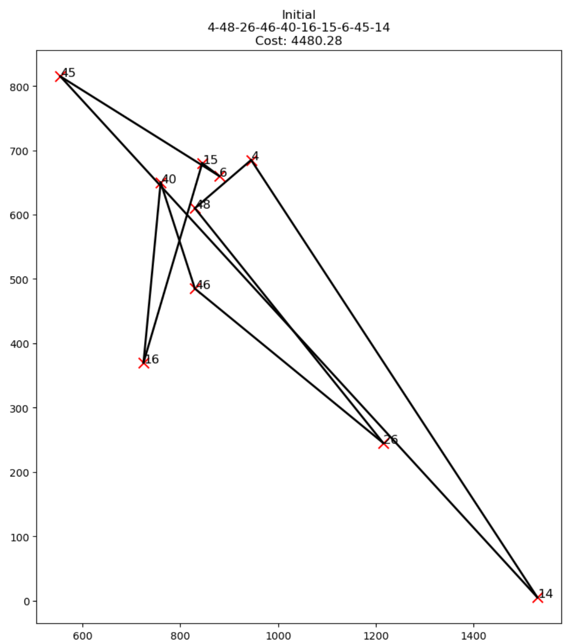
In [14]:
```
hc_sol_node.value()
```

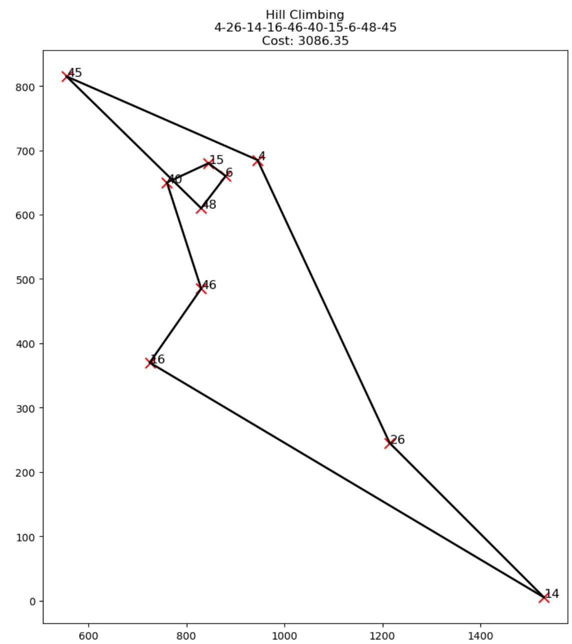
Out[14]: -3086.348120526929

In [15]:
```
# Plot the solution
fig, ax = plt.subplots()
plot_cities(ax, all_cities, hc_sol_node.state)
plot_path(ax, all_cities, hc_sol_node.state)
```



In [16]:
```
# Run
compare_sols((("Initial", initial_node), ("Hill Climbing", hc_sol_node)), all_citie
```





Simulations 1 - Subsample of 10

1. Create multiple subsamples of cities, of size 10 each
2. Create multiple initializations for each.
3. Run HC for each.
4. Present the initial and the HC results as a table.

Use

- subsample_size = 10
- subsample_seeds of `[0, 1, 2, 3, 4]`
- initial_seeds = `[11, 12, 13, 14, int(last_three_digits_of_your_CWID)]`

In [17]:

```
# TODO - Write code and run the simulation
```

In [18]:

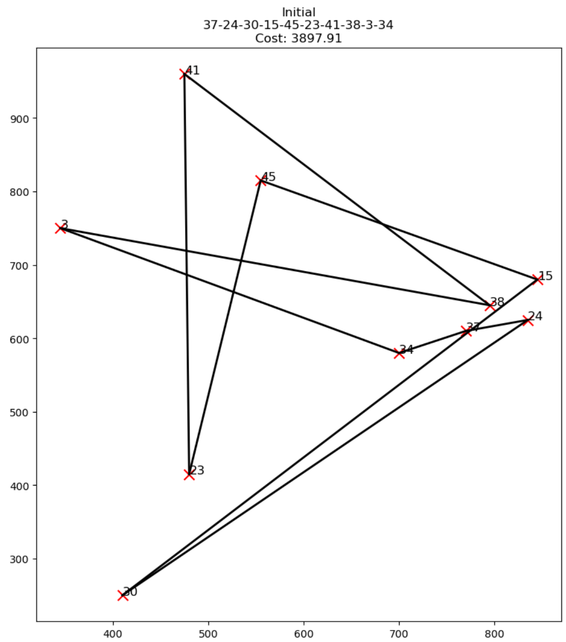
```
CWID='A12345678'
subsample_size = 10
subsample_seeds = range(0, 5)
initial_seeds = [11, 12, 13, 14, 15, int(CWID[6:])]

# TODO - Complete the code. It finally should display a table.
```

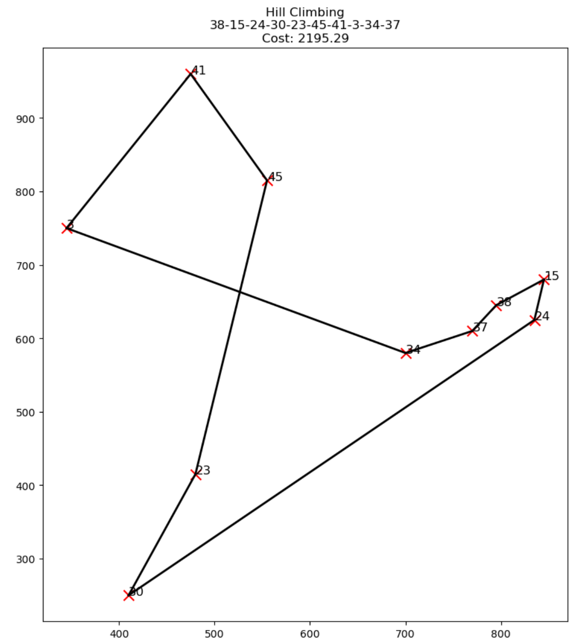
Out[18]:

	SubSeed	InitSeed	Initial	Hill Climbing
0	0	11	-6910.854964	-4845.283431
1	0	12	-4805.420205	-4578.810278
2	0	13	-6281.931261	-6230.336304
3	0	14	-6547.397559	-4943.598730
4	0	15	-6567.276372	-3994.034490
5	0	678	-6190.634115	-4844.239004
6	1	11	-3897.908590	-2195.289900
7	1	12	-3851.562313	-3140.493238
8	1	13	-3343.303102	-2746.034130
9	1	14	-2431.764939	-2159.997566
10	1	15	-2541.940049	-2390.651920
11	1	678	-4140.186452	-2790.398381
12	2	11	-4544.428908	-4089.549443
13	2	12	-4312.788187	-3211.747659
14	2	13	-4733.314832	-3118.205023
15	2	14	-3813.323874	-3334.947124
16	2	15	-4393.977833	-2850.467881
17	2	678	-4209.656929	-3197.775979
18	3	11	-4799.996881	-3990.085683
19	3	12	-5107.349706	-3764.067382
20	3	13	-4792.187279	-3898.438303
21	3	14	-4842.838365	-4342.710220
22	3	15	-4777.704743	-4060.666300
23	3	678	-5073.051494	-4474.981729
24	4	11	-6450.980927	-4993.819117
25	4	12	-7169.922143	-5886.325861
26	4	13	-6146.594024	-4957.369556
27	4	14	-4637.973404	-3997.536911
28	4	15	-6147.884236	-6078.827882
29	4	678	-5981.492319	-3960.525148

In [19]:

```
# Pick subsample seed and initial seed, and visualize the initialization and the HC

compare_sols((("Initial", initial_states[1][11]), ("Hill Climbing", hc_states[1][11
```





Simulations 2

Repeat the above simulation for

- subsample of 20
- subsample of 30
- subsample of 40

Finally, repeat it for the full set of cities (i.e., no subsampling, only random initialization).

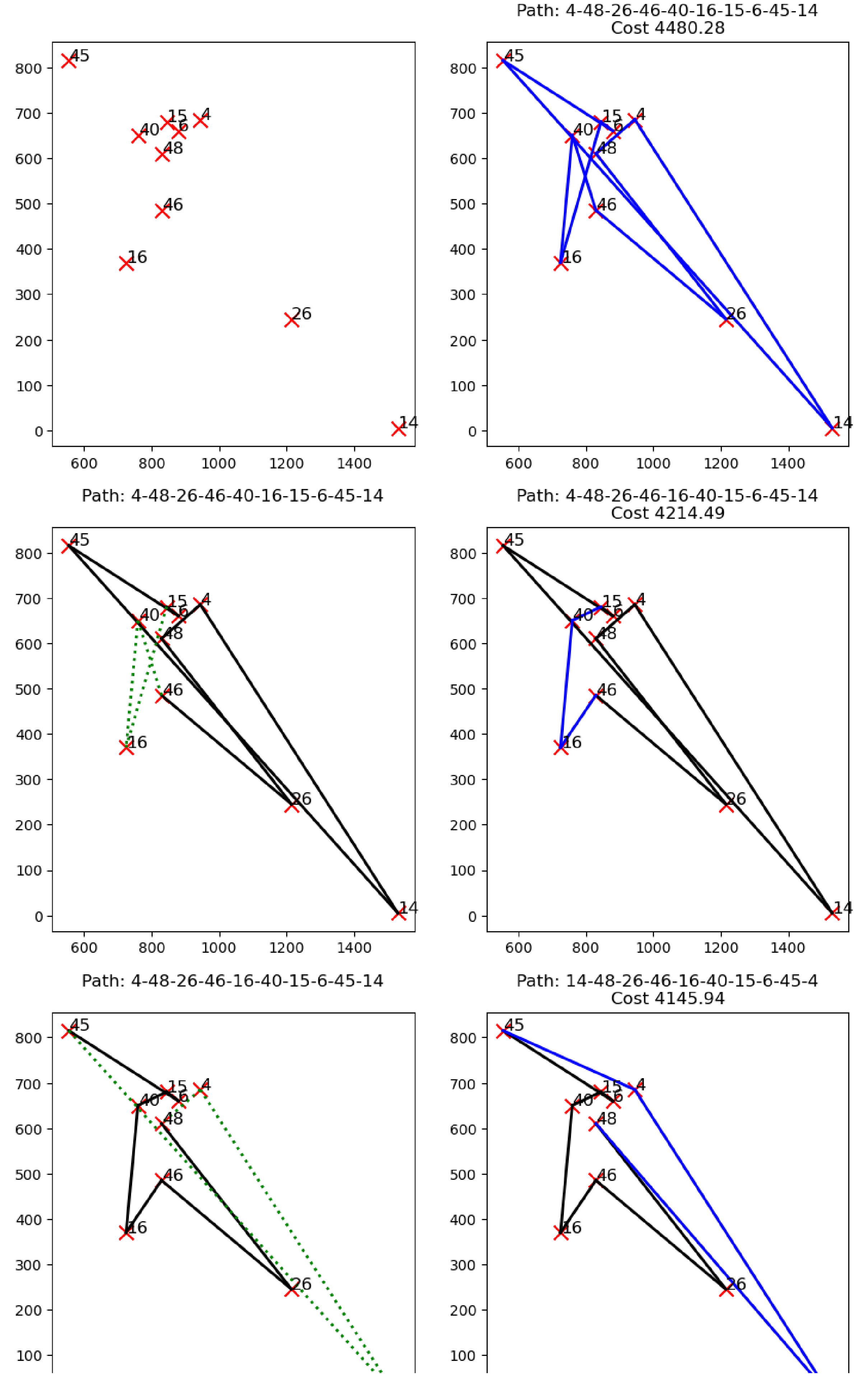
Optional (for fun only - no extra credit)

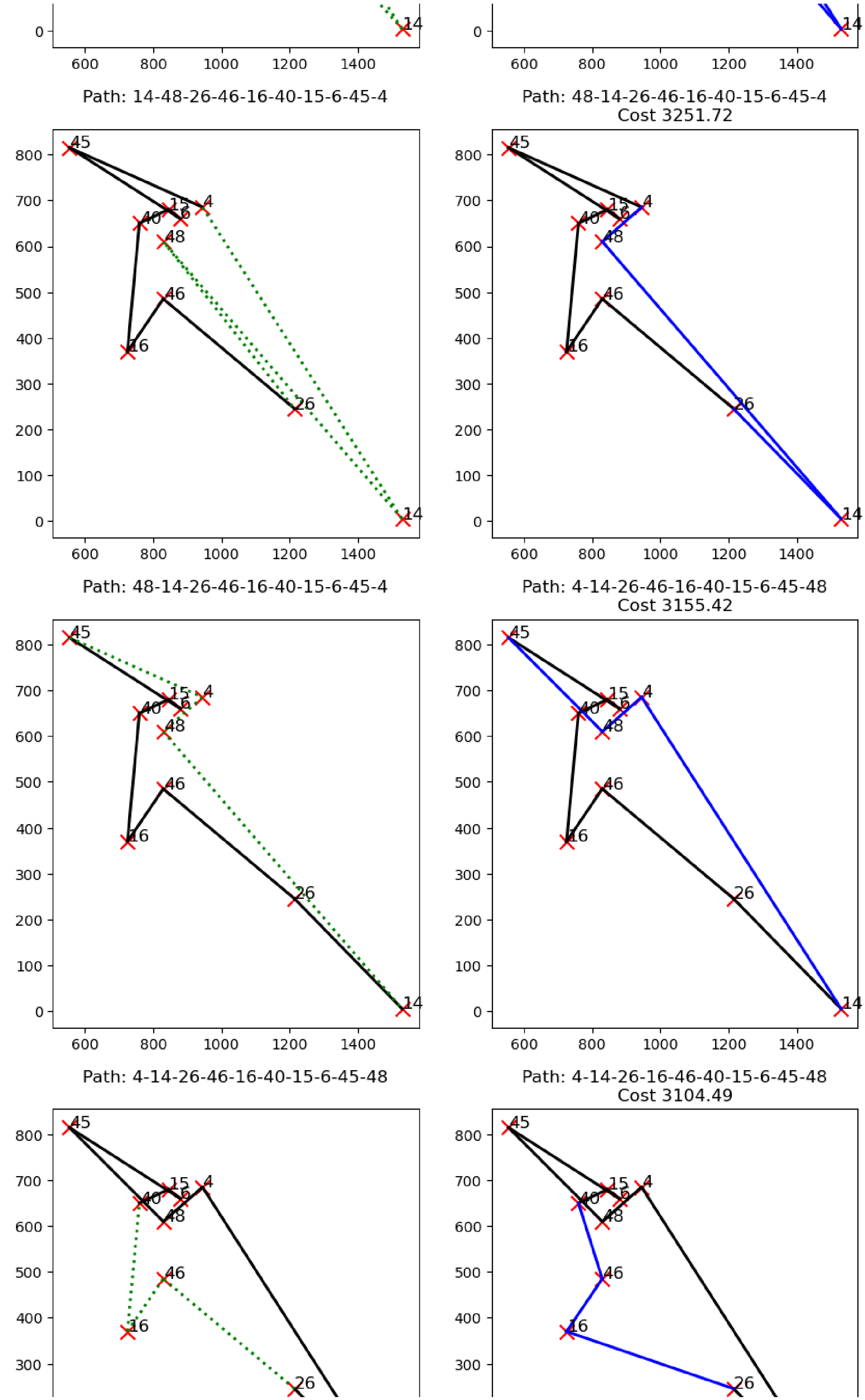
Given a Hill Climbing solution, trace the path to the initial state (using the `path()` function) and visualize the differences between each successsor state.

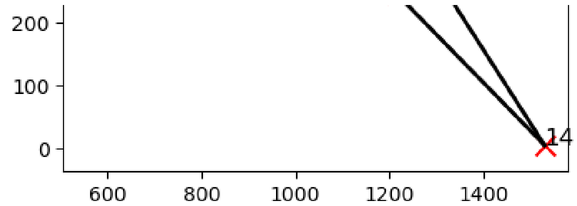
Here is a simple one. I'm sure you can come up with fancier ones.

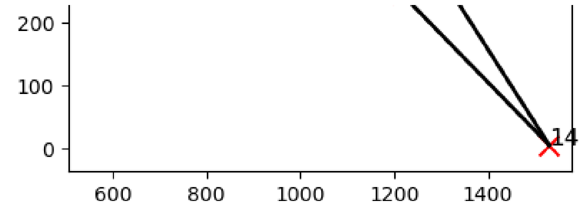
In [21]:

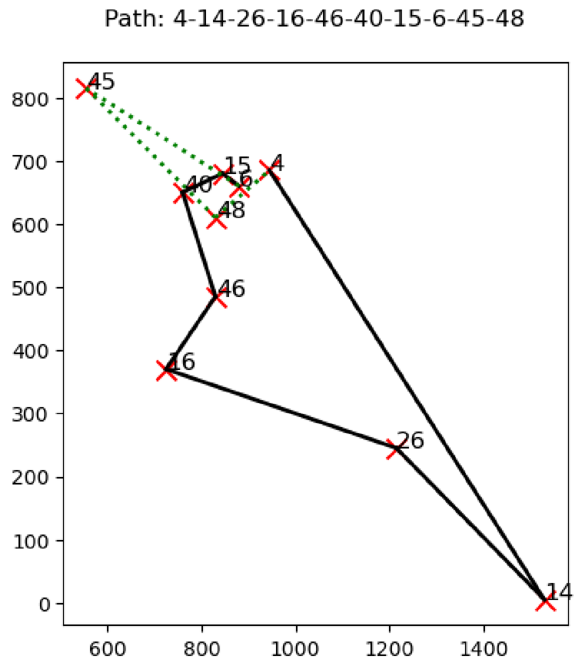
```

```

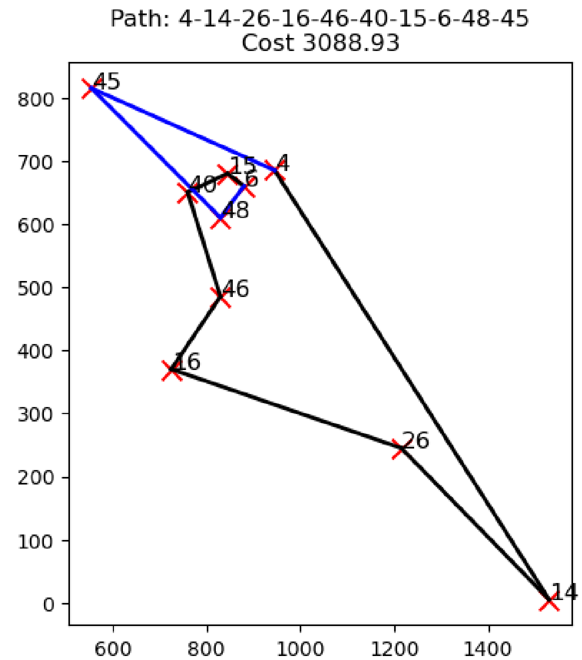

Path: 4-48-26-46-40-16-15-6-45-14
Cost 4480.28
Path: 4-48-26-46-40-16-15-6-45-14
Path: 4-48-26-46-16-40-15-6-45-14
Cost 4214.49
Path: 4-48-26-46-16-40-15-6-45-14
Path: 14-48-26-46-16-40-15-6-45-4
Cost 4145.94

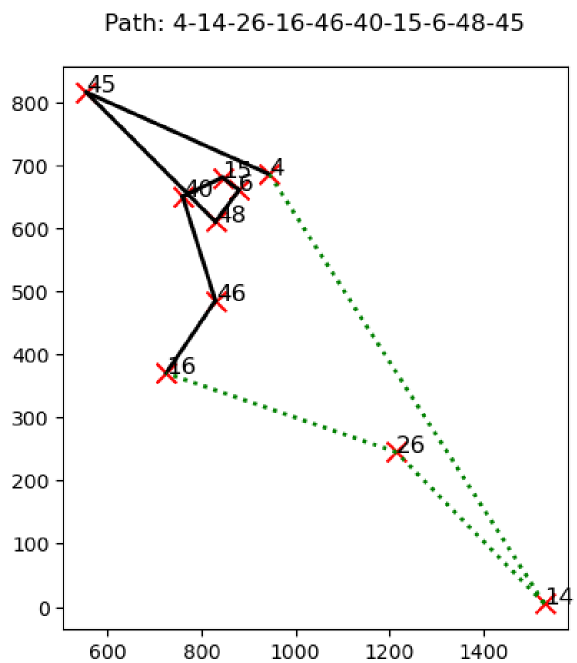

Path: 14-48-26-46-16-40-15-6-45-4
Path: 48-14-26-46-16-40-15-6-45-4
Cost 3251.72
Path: 48-14-26-46-16-40-15-6-45-4
Path: 4-14-26-46-16-40-15-6-45-48
Cost 3155.42
Path: 4-14-26-46-16-40-15-6-45-48
Path: 4-14-26-16-46-40-15-6-45-48
Cost 3104.49

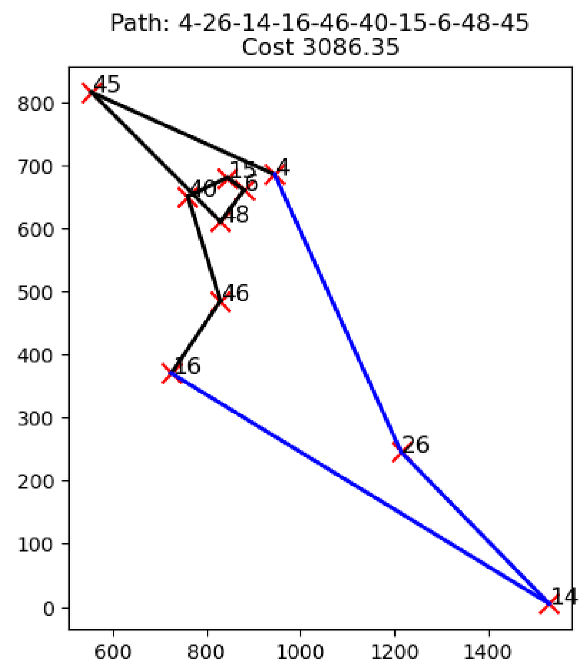

14


14


Path: 4-14-26-16-46-40-15-6-45-48
45
15
4
40
6
48
46
16
26
14


Path: 4-14-26-16-46-40-15-6-48-45
Cost 3088.93
45
15
4
40
6
48
46
16
26
14


Path: 4-14-26-16-46-40-15-6-48-45
45
15
4
40
6
48
46
16
26
14


Path: 4-26-14-16-46-40-15-6-48-45
Cost 3086.35
45
15
4
40
6
48
46
16
26
14

In [ ]: