Assignment 1 - Greedy heuristics

Authors

- Michał Kamiński 151969
- Jan Indrzejczak 152059

Table of contents

Description of the problem

The travelling salesman problem (TSP) is a classic optimization problem. Given a list of cities and the distances between them, the task is to find the shortest possible route that visits each city exactly once and returns to the origin city. In this version of the problem, each city also has a cost of being visited, and we only need to select half of the cities.

As an input we received a list of coordinates of cities, along with the cost. To calculate the distance between cities we used Euclidean distance, and each city is represented as a number from 0 to n-1 (n-number of cities). The objective function is to find the route that minimizes the sum of distances between cities and the cost of visiting them.

Pseudocode of all implemented algorithms

Random algorithm

```
Function RandomAlgorithm(cost_matrix, points_cost, start_from):
    size := number of nodes in cost_matrix
    solution_size := ⌈(size / 2)⌉

    // Create a list of all nodes
    solution := [0, 1, 2, ..., size - 1]

    // Shuffle the nodes randomly
    Shuffle(solution)

    // Select the first solution_size nodes
    solution := solution[0 to solution_size - 1]

    RETURN solution
```

Nearest neigbor algorithm with adding the node at the end

```
Function NearestNeighborEndAlgorithm(cost_matrix, points_cost, start_from):
    size := number of nodes in cost_matrix
    solution_size := ⌈(size / 2)⌉

    // Initialize the start node
    IF start_from is provided:
        start_node := start_from
    ELSE:
        start_node := 0
    ENDIF

    // Initialize solution with the start node
    solution := [start_node]

    // Initialize visited array to mark nodes as visited
    visited := [false, false, ..., false] of length size
    visited[start_node] := true

    // Build the solution by adding the nearest node
    WHILE length of solution < solution_size:
        last_node := last node in solution

        nearest_node := None
        nearest_cost := Infinity

        // Find the nearest unvisited node
        FOR each node in 0 to size - 1:
            IF node is not visited:
                distance := cost_matrix[last_node][node] + points_cost[node]
                IF distance < nearest_cost:
                    nearest_cost := distance
                    nearest_node := node
```

```
                ENDIF
            ENDIF

        // Add the nearest node to the solution and mark it visited
        IF nearest_node is not None:
            Add nearest_node to solution
            visited[nearest_node] := true
        ENDIF
    ENDWHILE

    RETURN solution
```

Nearest neigbor algorithm with adding the node at any position

```
Function NearestNeighborEndAlgorithm(cost_matrix, points_cost, start_from)
    size := number of nodes in cost_matrix
    solution_size := ⌈(size / 2)⌉

    // Initialize the start node
    IF start_from is provided:
        start_node := start_from
    ELSE:
        start_node := 0
    ENDIF

    // Initialize solution with the start node
    solution := [start_node]

    // Initialize visited array to mark nodes as visited
    visited := [false, false, ..., false] of length size
    visited[start_node] := true

    current_cost := points_cost[start_node]

    WHILE length of solution < solution_size
        current_solution_length := length of solution

        min_cost := Infinity
        min_cost_node := None
        min_cost_position := 0

        FOR each node in 0 to size - 1
            IF node is not visited
                FOR each position in 0 to current_solution_length + 1
                    cost_after_insertion := current_cost + points_cost[node]

                    IF position == 0
                        cost_after_insertion := cost_after_insertion +
cost_matrix[solution[0]][node]
                    ELSE IF position == current_solution_length
```

```
                        cost_after_insertion := cost_after_insertion +
cost_matrix[solution[current_solution_length - 1]][node]
                    ELSE
                        cost_after_insertion := cost_after_insertion
                            - cost_matrix[solution[position]][position - 1]
                            + cost_matrix[solution[position - 1]][node]
                            + cost_matrix[solution[position]][node]
                    ENDIF

                    IF min_cost > cost_after_insertion
                        min_cost := cost_after_insertion
                        min_cost_node := node
                        min_cost_position := position
                    ENDIF
                ENDFOR
            ENDIF
        ENDFOR

        insert into "solution" value of "min_cost_node" at index
"min_cost_position"

        current_cost := min_cost
        visited[min_cost_node] = true

    ENDWHILE

    RETURN solution
```

Greedy Cycle

```
Function NearestNeighborEndAlgorithm(cost_matrix, points_cost, start_from)
    size := number of nodes in cost_matrix
    solution_size := ⌈(size / 2)⌉

    // Initialize the start node
    IF start_from is provided:
        start_node := start_from
    ELSE:
        start_node := 0
    ENDIF

    // Initialize solution with the start node
    solution := [start_node]

    // Initialize visited array to mark nodes as visited
    visited := [false, false, ..., false] of length size
    visited[start_node] := true

    current_cost := points_cost[start_node]

    WHILE length of solution < solution_size
```

```
        current_solution_length := length of solution

        min_cost := Infinity
        min_cost_node := None
        min_cost_position := 0

        FOR each node in 0 to size - 1
            IF node is not visited
                FOR each position in 0 to current_solution_length + 1
                    cost_after_insertion := current_cost + points_cost[node]

                    IF position == 0 OR position == current_solution_length
                        cost_after_insertion := cost_after_insertion
                            - cost_matrix[solution[0]][current_solution_length -
1]
                            + cost_matrix[solution[0]][node]
                            + cost_matrix[solution[current_solution_length - 1]]
[node]
                    ELSE
                        cost_after_insertion := cost_after_insertion
                            - cost_matrix[solution[position]][position - 1]
                            + cost_matrix[solution[position - 1]][node]
                            + cost_matrix[solution[position]][node]
                    ENDIF

                    IF min_cost > cost_after_insertion
                        min_cost := cost_after_insertion
                        min_cost_node := node
                        min_cost_position := position
                    ENDIF
                ENDFOR
            ENDIF
        ENDFOR

        insert into "solution" value of "min_cost_node" at index
"min_cost_position"

        current_cost := min_cost
        visited[min_cost_node] = true

    ENDWHILE

    RETURN solution
```

Resutls of computatiional experiments

TSPA

```
Results for Random Algorithm
Min cost: 225467
Max cost: 292965
```

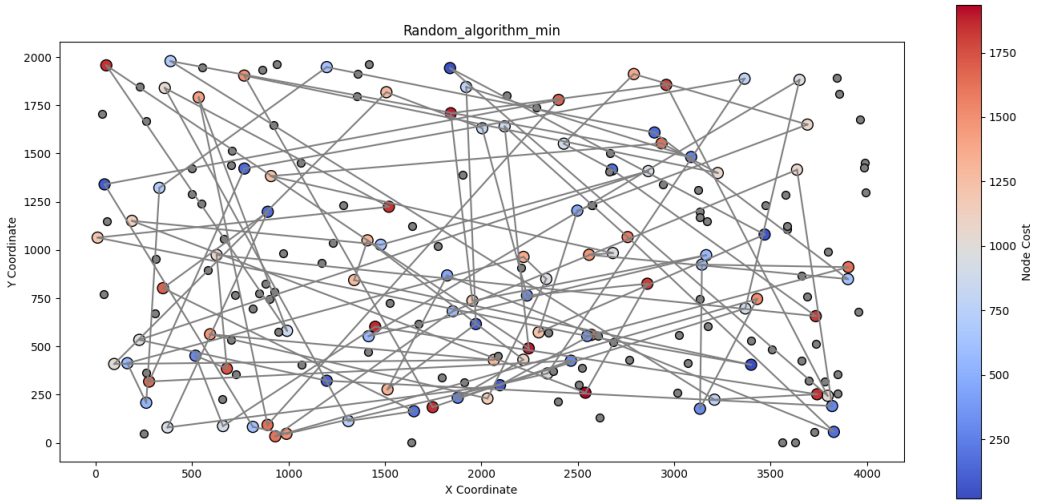
```
Average cost: 263940

Results for Nearest Neighbor with adding the node at the end algorithm
Min cost: 83182
Max cost: 89433
Average cost: 85109

Results for Nearest neighbor insert anywhere algorithm
Min cost: 71179
Max cost: 75450
Average cost: 73179

Results for Greedy cycle algorithm
Min cost: 71488
Max cost: 74410
Average cost: 72636
```

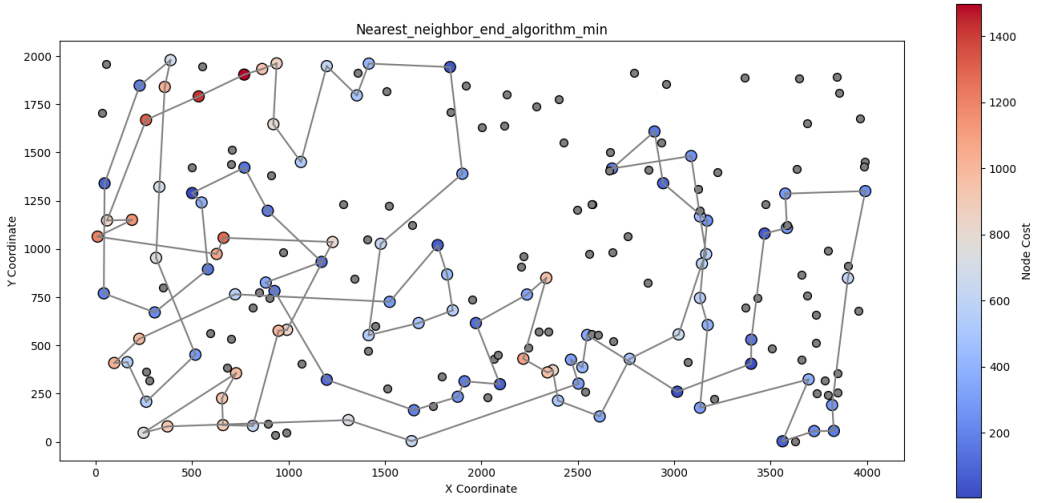
TSPB

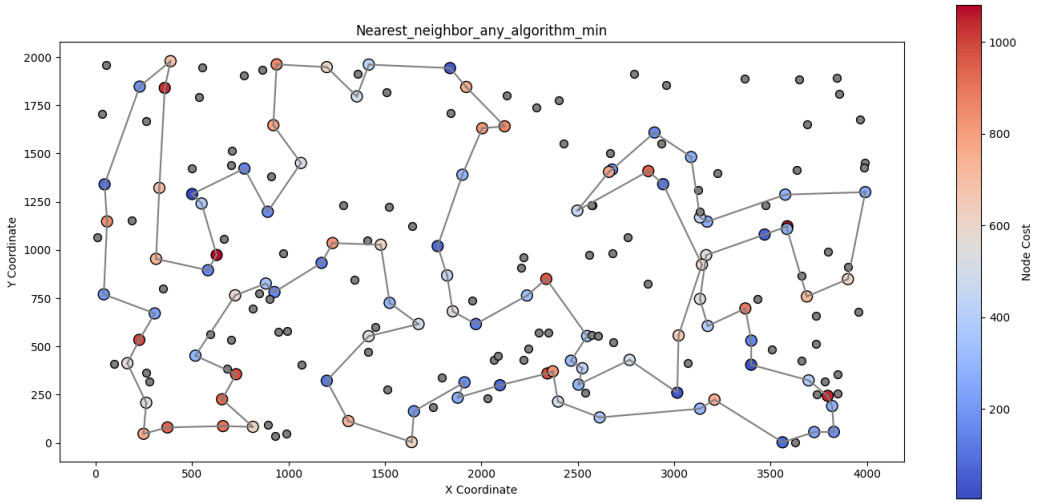
```
Results for Random Algorithm
Min cost: 193417
Max cost: 234394
Average cost: 213025

Results for Nearest Neighbor with adding the node at the end algorithm
Min cost: 52319
Max cost: 59030
Average cost: 54390

Results for Nearest neighbor insert anywhere algorithm
Min cost: 44417
Max cost: 53438
Average cost: 45870

Results for Greedy cycle algorithm
Min cost: 49001
Max cost: 57324
Average cost: 51401
```

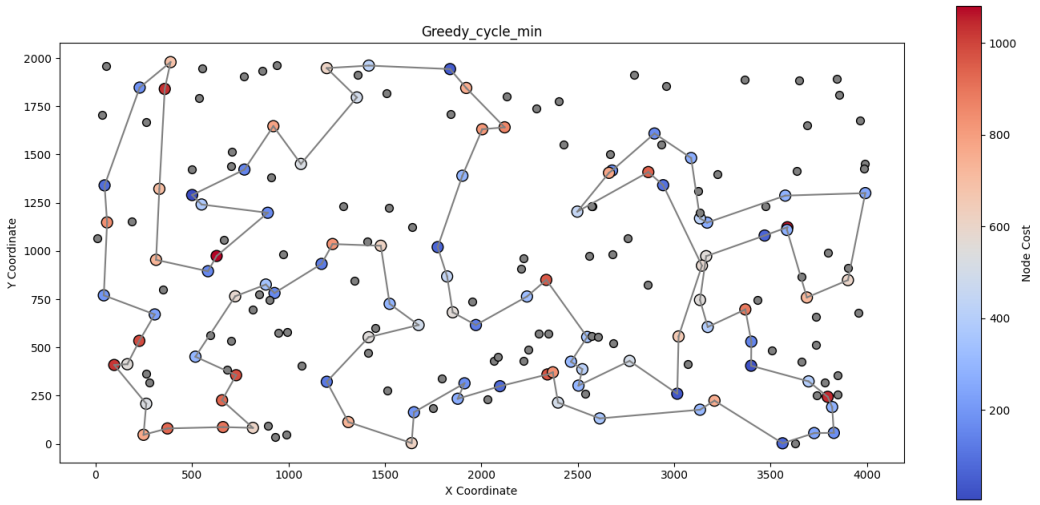
Plots of the results

TSPA

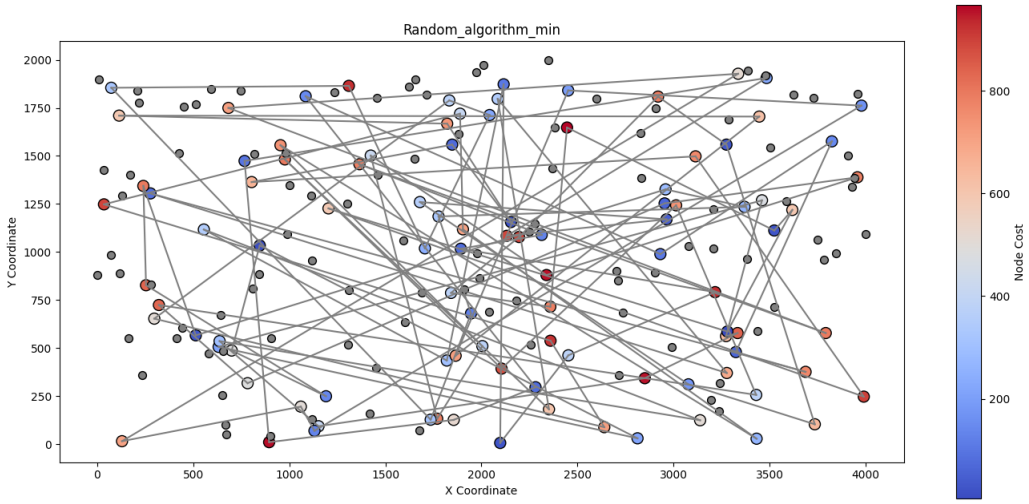

Random_algorithm_min
Y Coordinate
X Coordinate
Node Cost

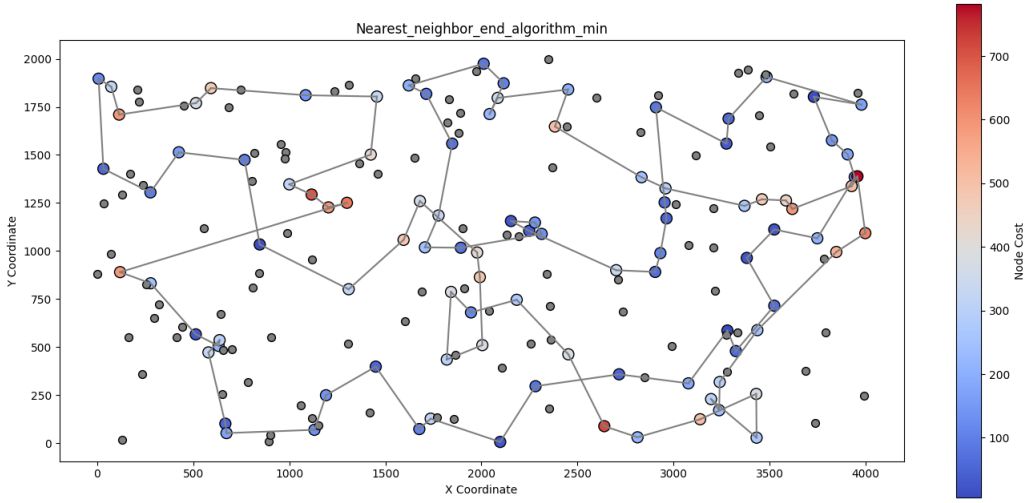

Nearest_neighbor_end_algorithm_min
Y Coordinate
X Coordinate
Node Cost

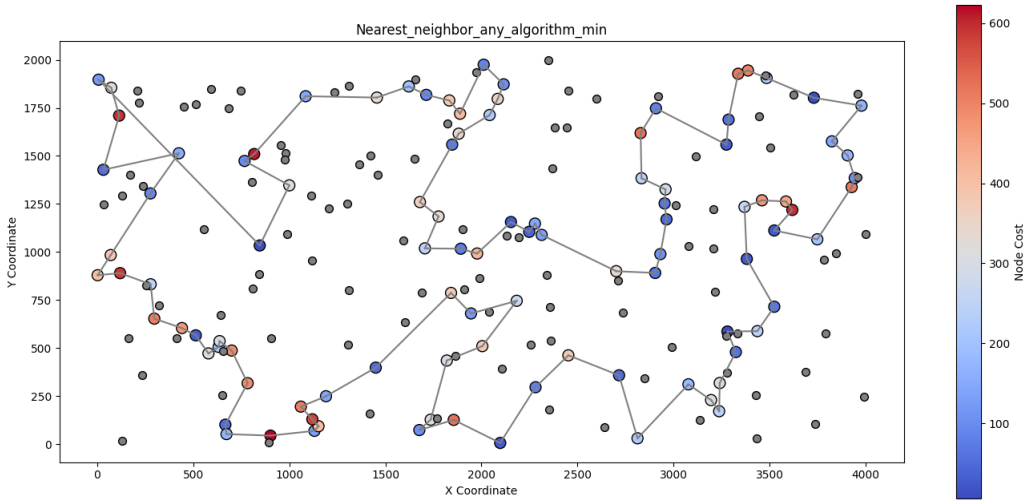

Nearest_neighbor_any_algorithm_min
Y Coordinate
X Coordinate
Node Cost

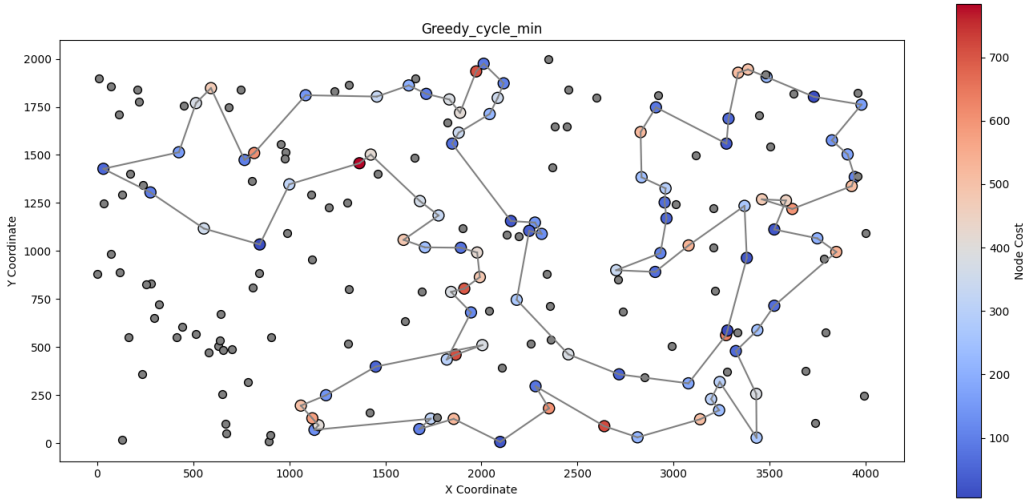

Greedy_cycle_min
Y Coordinate
X Coordinate
Node Cost

TSPB


Random_algorithm_min
Y Coordinate
X Coordinate
Node Cost


Nearest_neighbor_end_algorithm_min
Y Coordinate
X Coordinate
Node Cost





Best solutions as a list of nodes

TSPA

	Greedy cycle	Nearest neighbor any algorithm	Nearest neighbor end algorithm	Random algorithm
0	46	68	124	159
1	68	46	94	177
2	139	115	63	54
3	193	139	53	12
4	41	193	180	124
5	115	41	154	158

	Greedy cycle	Nearest neighbor any algorithm	Nearest neighbor end algorithm	Random algorithm
6	5	5	135	28
7	42	42	123	136
8	181	181	65	30
9	159	159	116	153
10	69	69	59	23
11	108	108	115	94
12	18	18	139	172
13	22	22	193	150
14	146	146	41	195
15	34	34	42	51
16	160	160	160	102
17	48	48	34	48
18	54	54	22	123
19	30	177	18	113
20	177	10	108	39
21	10	190	69	79
22	190	4	159	156
23	4	112	181	91
24	112	84	184	44
25	84	35	177	55
26	35	184	54	187
27	184	43	30	74
28	43	116	48	53
29	116	65	43	190
30	65	59	151	115
31	59	118	176	192
32	118	51	80	135
33	51	151	79	127
34	151	133	133	69

	Greedy cycle	Nearest neighbor any algorithm	Nearest neighbor end algorithm	Random algorithm
35	133	162	162	141
36	162	123	51	103
37	123	127	137	152
38	127	70	183	78
39	70	135	143	72
40	135	180	0	109
41	180	154	117	121
42	154	53	46	154
43	53	100	68	179
44	100	26	93	7
45	26	86	140	162
46	86	75	36	52
47	75	44	163	122
48	44	25	199	76
49	25	16	146	132
50	16	171	195	56
51	171	175	103	183
52	175	113	5	185
53	113	56	96	189
54	56	31	118	148
55	31	78	149	95
56	78	145	131	73
57	145	179	112	15
58	179	92	4	62
59	92	57	84	17
60	57	52	35	5
61	52	185	10	11
62	185	119	190	184
63	119	40	127	4

	Greedy cycle	Nearest neighbor any algorithm	Nearest neighbor end algorithm	Random algorithm
64	40	196	70	45
65	196	81	101	13
66	81	90	97	25
67	90	165	1	108
68	165	106	152	14
69	106	178	120	89
70	178	14	78	63
71	14	144	145	194
72	144	62	185	80
73	62	9	40	175
74	9	148	165	128
75	148	102	90	186
76	102	49	81	99
77	49	55	113	134
78	55	129	175	149
79	129	120	171	163
80	120	2	16	126
81	2	101	31	64
82	101	1	44	22
83	1	97	92	112
84	97	152	57	29
85	152	124	106	100
86	124	94	49	21
87	94	63	144	139
88	63	79	62	104
89	79	80	14	97
90	80	176	178	6
91	176	137	52	36
92	137	23	55	138

	Greedy cycle	Nearest neighbor any algorithm	Nearest neighbor end algorithm	Random algorithm
93	23	186	129	155
94	186	89	2	198
95	89	183	75	167
96	183	143	86	111
97	143	0	26	81
98	117	117	100	144
99	0	93	121	117

TSPB

	Greedy cycle	Nearest neighbor any algorithm	Nearest neighbor end algorithm	Random algorithm
0	51	40	16	76
1	121	107	1	135
2	131	100	117	26
3	135	63	31	15
4	63	122	54	97
5	122	135	193	120
6	133	38	190	58
7	10	27	80	83
8	90	16	175	114
9	191	1	5	24
10	147	156	177	136
11	6	198	36	18
12	188	117	61	154
13	169	193	141	37
14	132	31	77	134
15	13	54	153	79
16	161	73	163	82
17	70	136	176	71
18	3	190	113	137

	Greedy cycle	Nearest neighbor any algorithm	Nearest neighbor end algorithm	Random algorithm
19	15	80	166	49
20	145	162	86	13
21	195	175	185	92
22	168	78	179	14
23	29	142	94	146
24	109	45	47	156
25	35	5	148	132
26	0	177	20	145
27	111	104	60	138
28	81	8	28	150
29	153	111	140	107
30	163	82	183	61
31	180	21	152	8
32	176	61	18	11
33	86	36	62	95
34	95	91	124	55
35	128	141	106	117
36	106	77	143	32
37	143	81	0	197
38	124	153	29	5
39	62	187	109	73
40	18	163	35	121
41	55	89	33	187
42	34	127	138	31
43	170	103	11	130
44	152	113	168	180
45	183	176	169	88
46	140	194	188	186
47	4	166	70	78

	Greedy cycle	Nearest neighbor any algorithm	Nearest neighbor end algorithm	Random algorithm
48	149	86	3	54
49	28	95	145	165
50	20	130	15	131
51	60	99	155	21
52	148	22	189	3
53	47	185	34	141
54	94	179	55	81
55	66	66	95	64
56	22	94	130	33
57	130	47	99	35
58	99	148	22	85
59	185	60	66	48
60	179	20	154	178
61	172	28	57	4
62	166	149	172	108
63	194	4	194	45
64	113	140	103	175
65	114	183	127	60
66	137	152	89	155
67	103	170	137	87
68	89	34	114	199
69	127	55	165	100
70	165	18	187	126
71	187	62	146	29
72	146	124	81	168
73	77	106	111	125
74	97	143	8	174
75	141	35	104	93
76	91	109	21	118

	Greedy cycle	Nearest neighbor any algorithm	Nearest neighbor end algorithm	Random algorithm
77	36	0	82	176
78	61	29	144	22
79	175	160	160	185
80	78	33	139	167
81	142	138	182	41
82	45	11	25	104
83	5	139	121	12
84	177	168	90	53
85	82	195	122	183
86	87	145	135	75
87	21	15	63	163
88	8	3	40	139
89	104	70	107	62
90	56	13	100	39
91	144	132	133	28
92	160	169	10	90
93	33	188	147	46
94	138	6	6	91
95	182	147	134	77
96	11	191	51	147
97	139	90	98	124
98	134	51	118	148
99	85	121	74	113

Source code:

- Github repository

Conclusions

The best solutions have been checked with the solution checker.

Unsurprisingly, the Random algorithm turned out to be the worst. The second to last results were achieved by the Nearest neighbor with adding the node at the end algorithm.

In both datasets, the Nearest neighbor considering adding the node at all possible position turned out to yield the best solutions. Slightly worse results were obtained by the Greedy cycle algorithm. This was surprising, since Greedy cycle seems as a better heuristic. It considers the connection between the first and the last city, but this makes the algorithm consider much more compressed area because at all times it wanted to optimize the whole path including the last connection. This is clearly visible on the plots, each path encompassed by the Greedy cycle algorithm is much more compressed than Nearest neighbor.