Evolutionary Computation

Assignment 5

Antoni Lasik 148287

Daniel Jankowski 148257

**Problem description**

The task involves analyzing three columns of integers, each row corresponding to a single node. The initial two columns designate the x and y coordinates, pinpointing the nodes' locations on a plane, while the third column specifies the cost associated with each node. The objective is to meticulously choose an exact half of the total nodes (in cases where the total node count is an odd number, the count of nodes to be selected is adjusted upward to the nearest whole number) to construct a Hamiltonian cycle, which is essentially a continuous loop that passes through each member of the selected set of nodes. The criterion for this selection is that the aggregate of the complete path's length and the cumulative cost of the chosen nodes should be as low as possible.

To quantify the distances between nodes, we employ the Euclidean distance formula, and the resulting figures are rounded off to the nearest integer in a standard mathematical fashion. Moreover, as part of the distance between nodes, we take into account the cost of the destination node. This ensures that cost has a significant impact on the final results.

In this report we implement the steepest version of a local search with deltas from previous iteration. While the results should be almost the same, the purpose of the modification is to reduce the runtime of a classical algorithm version.

**Pseudocode of implemented algorithms**

**calculate\_distance\_matrix(coords, costs):**

dist\_matrix = [][]

**FOR** i **IN** **RANGE**\(len(coords)):

**FOR** j **IN** **RANGE**(len(coords)):

dist\_matrix[i][j] = round(sqrt((coords[i].x – coords[j].x)\*\*2 + (coords[i].y – coords[j].y)\*\*2)

**RETURN** dist\_matrix

**objective\_function(solution, dist\_matrix, costs):**

total\_score = 0

n = len(solution)

**FOR** x in range(n):

total\_score += dist\_matrix[solution[x - 1]][solution[x]]

total\_score += costs[solution[x]]

**RETURN total\_score**

The methods for checking the applicability of a given move. The methods returns the following values:

* -1 if a move is not applicable and should be removed from LM
* 0 if a move is not applicable now, but shouldn’t be removed from LM
* 1 if a move is applicable and will be accepted

**is\_intra\_move\_applicable(solution, move):**

# for the intra moves changes affects only nodes that are inside a given solution, # therefore first we check if all nodes in edges, that a move introduced, are # present in the current solution

**FOR** edge **IN** move.added\_edges:

**IF** edge.source\_node **NOT IN** solution **OR**

edge.dest\_node **NOT IN** solution:

**RETURN -1**

# check if all edges that a move removed are present in a solution

all\_edges\_match = True

**FOR** edge **IN** move.removed\_edges:

reversed\_edge = edge[-1] # reverse the edge

**IF** edge **NOT IN** solution **AND** reversed\_edge **NOT IN** solution:

**RETURN -1**

**IF** edge **NOT** **IN** solution **AND** reversed\_edge **IN** solution:

all\_edges\_match = False

**RETURN 1 IF** all\_edges\_match **ELSE 0**

**is\_inter\_move\_applicable(solution, move):**

# added edges by a move have shape:(old\_node\_1, NEW\_NODE),(NEW\_NODE, old\_node\_2)

# therefore first we check if all old nodes are present in the current solution

**IF** move.added\_edges[0].source\_node **NOT IN** solution **OR**

move.added\_edges[1].dest\_node **NOT** **IN** solution:

**RETURN -1**

# if the node that will be inserted is not in the list of currently unselected # nodes, a move can’t be applied

**IF** move.added\_edges[0].dest\_node **NOT IN** unselected\_nodes:

**RETURN -1**

# check if all edges that a move removed are present in a solution

all\_edges\_match = True

**FOR** edge **IN** move.removed\_edges:

reversed\_edge = edge[-1] # reverse the edge

**IF** edge **NOT IN** solution **AND** reversed\_edge **NOT IN** solution:

**RETURN -1**

**IF** edge **NOT** **IN** solution **AND** reversed\_edge **IN** solution:

all\_edges\_match = False

**RETURN 1 IF** all\_edges\_match **ELSE 0**

To control moves that has been already discovered and evaluated the heap has been implemented using built-in python package. The heap stores information about discovered moves and their delta score, always sorted in a descending order by the score. We have used three basic methods:

* heap.add\_move() – add a new move to the heap
* heap.heappop() – take the first element from the heap (with the best score)
* heap.move\_exists() – check if a given move already exists in a heap

**two\_edges\_exchange(solution, heap, dist\_matrix):**

n = len(solution.nodes)

# iterate through the neighborhood

**FOR** i **IN RANGE(**n-2**):**

**FOR** j **IN RANGE(**i+2, n**):**

# construct a move

removed\_edges = (Edge(solution.nodes[j], solution.nodes[i+1]),

Edge(solution.nodes[j], solution.nodes[(j+1)%n]))

added\_edges = (Edge(solution.nodes[i], solution.nodes[j]),

Edge(solution.nodes[i+1], solution.nodes[(j+1)%n])

reversed\_edges = []

**FOR** k **IN RANGE(**i+1, j**):**

**ADD** Edge(solution.nodes[k], solution.nodes[k+1]) **TO** reversed\_edges

move = Move(removed\_edges, added\_edges, reversed\_edges)

# check if a move already exists in LM, if yes skip it

**IF** heap.move\_exists(move):

**CONTINUE**

**ELSE:**

# compute the delta score of a move

score\_delta = (

-dist\_matrix[solution.nodes[i]][solution.nodes[i+1]

-dist\_matrix[solution.nodes[j]][solution.nodes[(j+1)%n]]

+dist\_matrix[solution.nodes[i]][solution.nodes[j]]

+dist\_matrix[solution.nodes[i+1]][solution.nodes[(j+1)%n]]

)

**IF** score\_delta < 0:

heap.add\_move(move, score\_delta)

**two\_nodes\_exchange(solution, heap, dist\_matrix):**

n = len(solution.nodes)

index\_pairs = [(x, y) for x in range(n) for y in range(x+1, n)]

**FOR** (i, j) in index\_pairs:

# special case: the last and the first node are exchanged

if i == 0 and j == n-1:

removed\_edges = (Edge(solution.nodes[j], solution.nodes[0]),

Edge(solution.nodes[j-1], solution.nodes[j]),

Edge(solution.nodes[0], solution.nodes[1]))

added\_edges = (Edge(solution.nodes[j], solution.nodes[1]),

Edge(solution.nodes[j-1], solution.nodes[0]),

Edge(solution.nodes[0], solution.nodes[j]))

move = Move(removed\_edges, added\_edges)

if heap.move\_exists(move):

**CONTINUE**

**ELSE:**

score\_delta = (

-dist\_matrix[solution.nodes[j]][solution.nodes[0]]

-dist\_matrix[solution.nodes[j-1]][solution.nodes[j]]

-dist\_matrix[solution.nodes[0]][solution.nodes[1]]

+dist\_matrix[solution.nodes[j]][solution.nodes[1]]

+dist\_matrix[solution.nodes[j-1][solution.nodes[0]]

+dist\_matrix[solution.nodes[0][solution.nodes[j]]

**IF** delta\_score < 0:

heap.add\_move(move, delta\_score)

**ELIF** j == i + 1:

# case when nodes are adjacent

removed\_edges = (Edge(solution.nodes[j], solution.nodes[0]),

Edge(solution.nodes[j-1], solution.nodes[j]),

Edge(solution.nodes[0], solution.nodes[1]))

added\_edges = (Edge(solution.nodes[j], solution.nodes[1]),

Edge(solution.nodes[j-1], solution.nodes[0]),

Edge(solution.nodes[0], solution.nodes[j]))

move = Move(removed\_edges, added\_edges)

if heap.move\_exists(move):

**CONTINUE**

**ELSE:**

**Results**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Method | Instance A | Instance B | Instance C | Instance D |
| Random solution | 264,028(237,941-288,302) | 266,665(243,288-295,269) | 214,929(192,705-241,451) | 219,678(191,218-242,515) |
| Nearest Neighbor | 87,679(84,471-95,013) | 79,282(77,448-82,631) | 58,290(56,304-63,697) | 54,290.68(50,335-59,846) |
| Greedy Cycle | 76,711(75,136-80,025) | 70,464(67,896-76,096) | 55,859(53,020-58,499) | 54,931(50,288-60,208) |
| 2-regret GC | 116,772(106,734-124,404) | 116,871(104,997-125,925) | 68,444(63,247-72,558) | 68,585(62,852-74,184) |
| 2-regret with weighted sum | 76,980(74,708-82,990) | 73,067(67,490-80,001) | 53,795(50,158-58,173) | 52,930(46,549-62,321) |
| Greedy LS, random solution, two-edges + inter route | 77,014(74,663-79,803) | 69,990(67,877-74,141) | 50,998 (49,340-53,141) | 48,068 (45,336-51,629) |
| Greedy LS, random solution, two-nodes + inter route | 90,940(84,816-99,390) | 85,570(77,908-97,299) | 63,929 (58,135-70,886) | 62,175 (54,310-71,108) |
| Greedy LS, best solution from 2-regret with weighted sum, two-edges + inter route | 75,792 (74,221-79,688) | **71,266 (67,384-77,120)** | **52350,15(48,931-55,758)** | 51,013 (45,212-59,478) |
| Greedy LS, best solution from 2-regret with weighted sum, two-nodes + inter route | 75,932(74,344-79,315) | **71,839 (67,384-77,565)** | 52,638 (49,649-56,472) | 51,248(45,097-60,185) |
| Steepest LS, random solution, two-edges + inter route | 78,017 (74,874-82,619) | 71337.98(67,909-76,199) | 51,485 (49,235-53,755) | 48,225 (45,673-51,639) |
| Steepest LS, random solution, two-nodes + inter route | 92,714(84,218-103,034) | 87,666(79,356-97,895) | 65,679(59,604-73,386) | 64,162(54,716-75,351) |
| Steepest LS, best solution from 2-regret with weighted sum, two-edges + inter route | **75,728(74,091-79,220)** | **71,233 (67,384-77,057)** | 52,299 (49,098-5,5665) | **50,977(45,097-59,478)** |
| Steepest LS, best solution from 2-regret with weighted sum, two-nodes + inter route | 75,880(74,280-79,220) | **71,894(67,384-77,420)** | 52,607 (49,460-56,472) | **51,247 (45,097-60,185)** |

*\*the structure of values in the table: avg(min-max)*

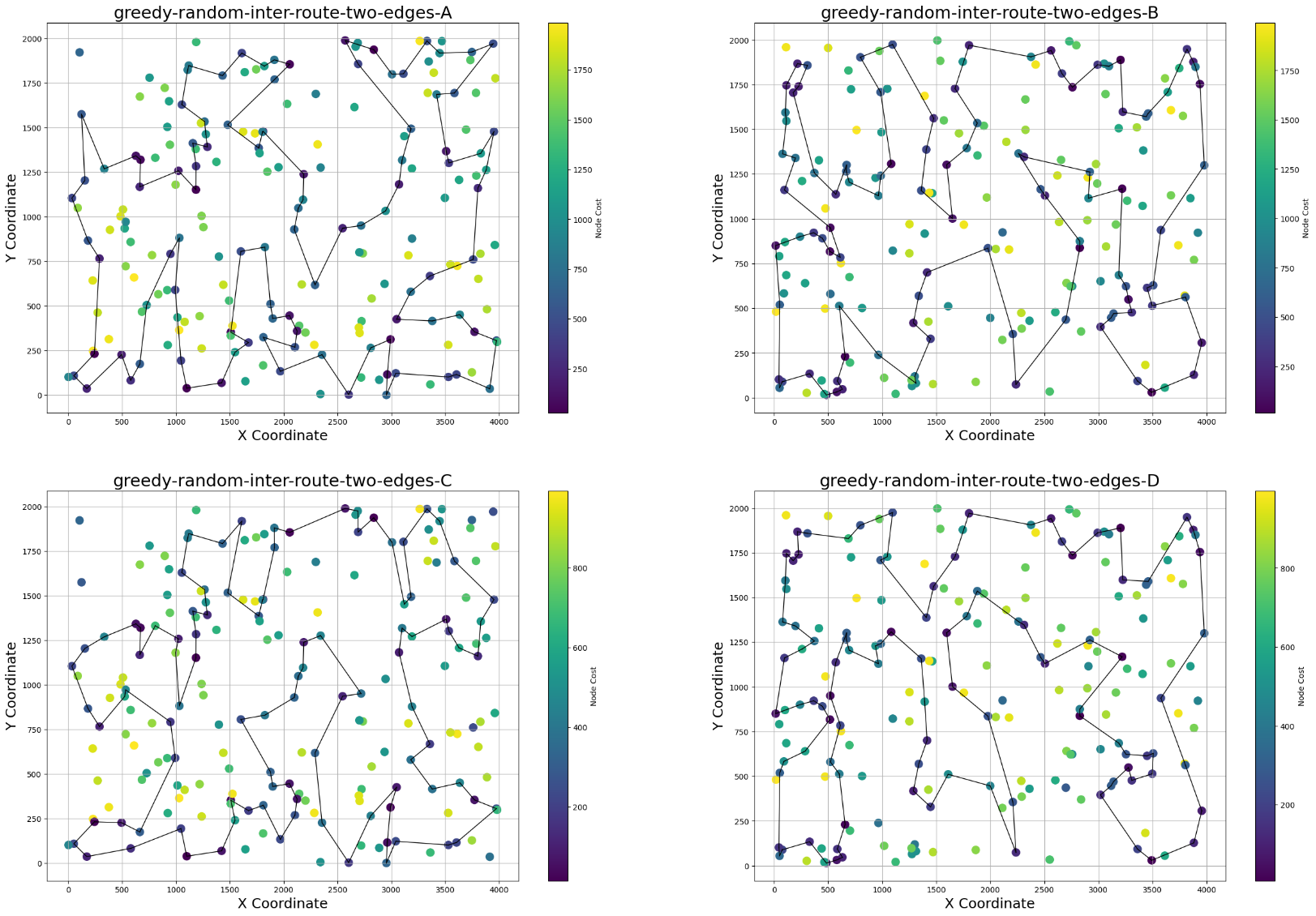
*\*\*in bold there are the best minimal solutions founded by a given method*

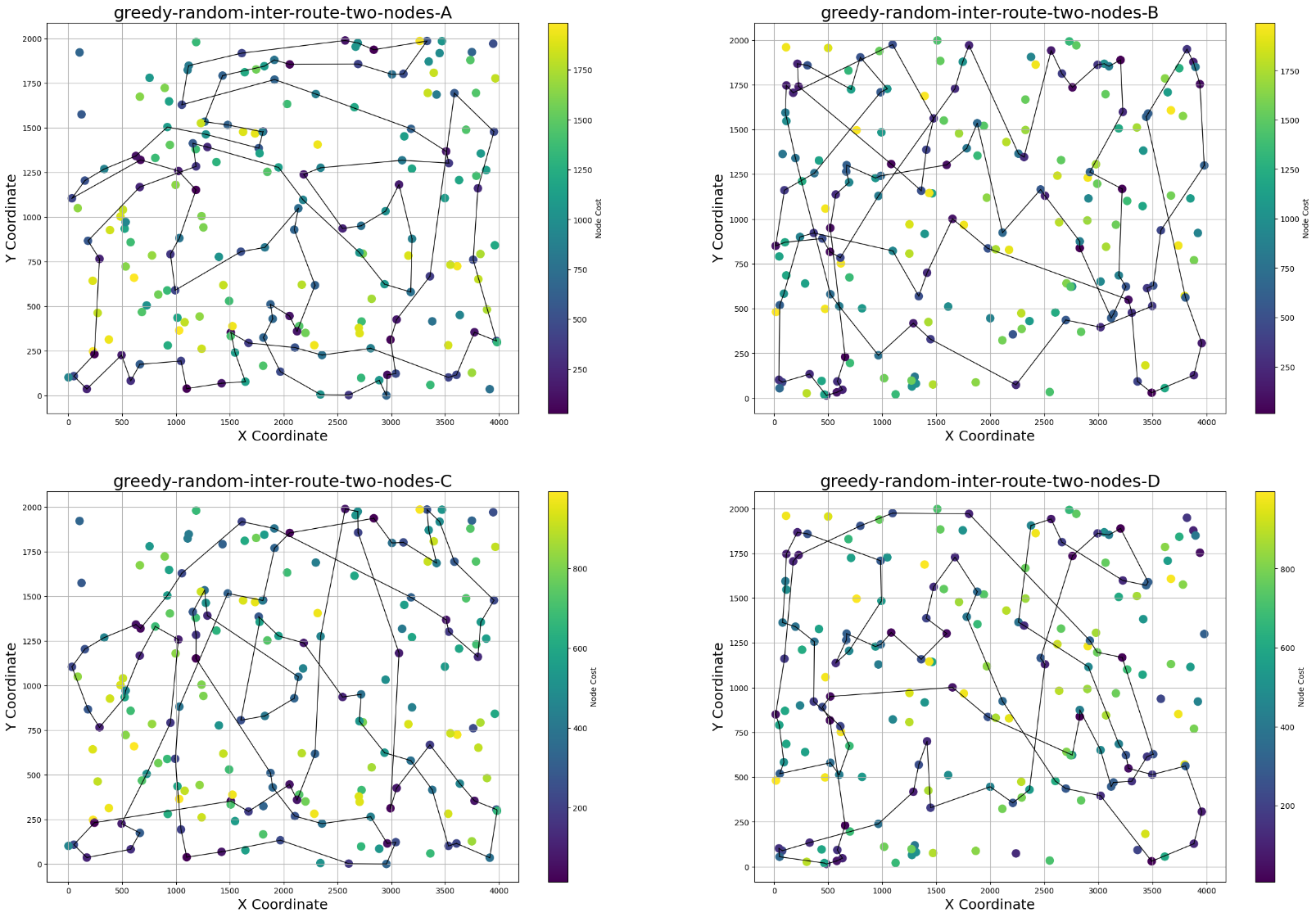
**Runtimes**

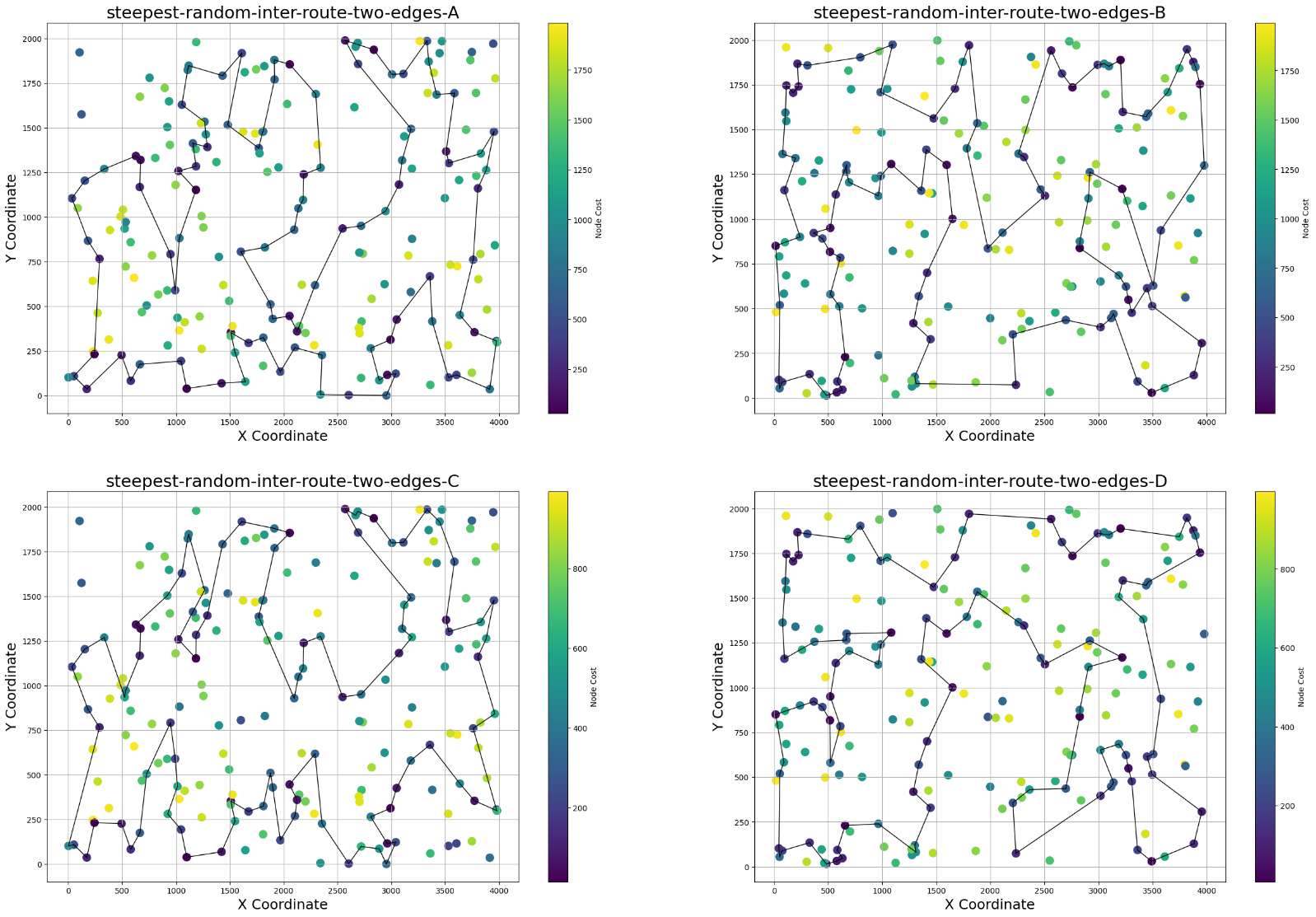
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Method | Instance A | Instance B | Instance C | Instance D |
| Greedy LS, random solution, two-edges + inter route | 1.56(1.06-2.63) | 1.95(1.19-3.48) | 1.25(0.77-2.28) | 1.18(0.72-1.99) |
| Greedy LS, random solution, two-nodes + inter route | 1.68(1.03-2.98) | 1.95(0.81-6.66) | 1.38(0.79-2.21) | 1.37(0.77-2.36) |
| Greedy LS, best solution from 2-regret with weighted sum, two-edges + inter route | 0.67(0.51-0.97) | 0.7(0.5-1.18) | 0.66(0.5-0.93) | 0.65(0.51-0.89) |
| Greedy LS, best solution from 2-regret with weighted sum, two-nodes + inter route | 0.63(0.46-0.89) | 0.69(0.53-1.15) | 0.68(0.49-1.24) | 0.67(0.54-1.18) |
| Steepest LS, random solution, two-edges + inter route | 5.46(4.47-7.46) | 5.64(4.51-7.16) | 5.41(4.72-6.54) | 5.64(4.76-6.88) |
| Steepest LS, random solution, two-nodes + inter route | 6.82(5.46-8.96) | 6.63(4.89-10.51) | 6.8(5.41-9.2) | 0.69(0.5-1.18) |
| Steepest LS, best solution from 2-regret with weighted sum, two-edges + inter route | 0.85(0.55-1.53) | 0.95(0.53-1.78) | 0.94(0.57-1.6) | 1(0.58-1.38) |
| Steepest LS, best solution from 2-regret with weighted sum, two-nodes + inter route | 0.88(0.58-1.71) | 0.83(0.53-1.57) | 0.89(0.5-1.58) | 1.03(0.67-1.5) |

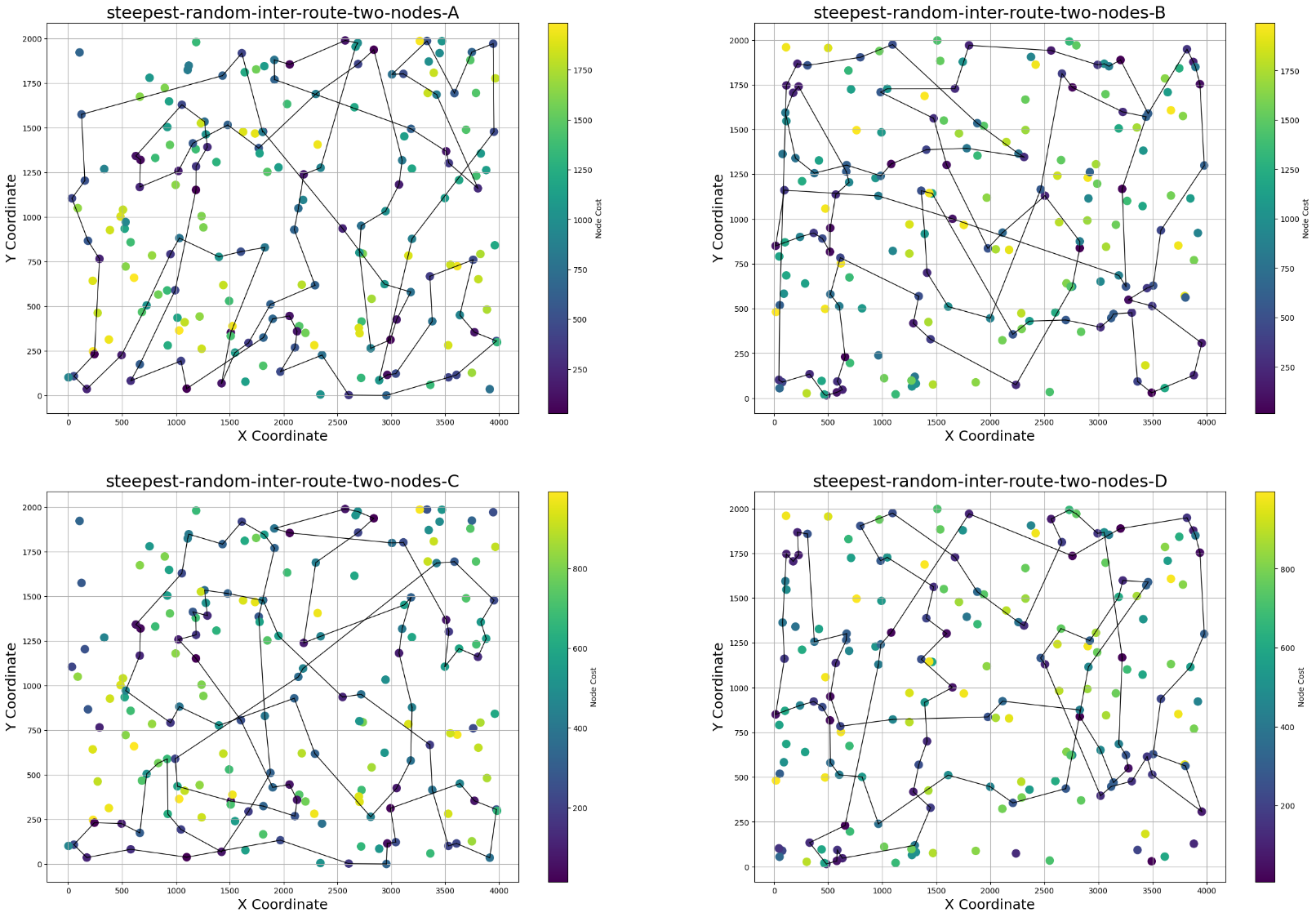
***\*****the format is: avg(min-max)*

***\*\*****all runtimes are provided in seconds.*

**

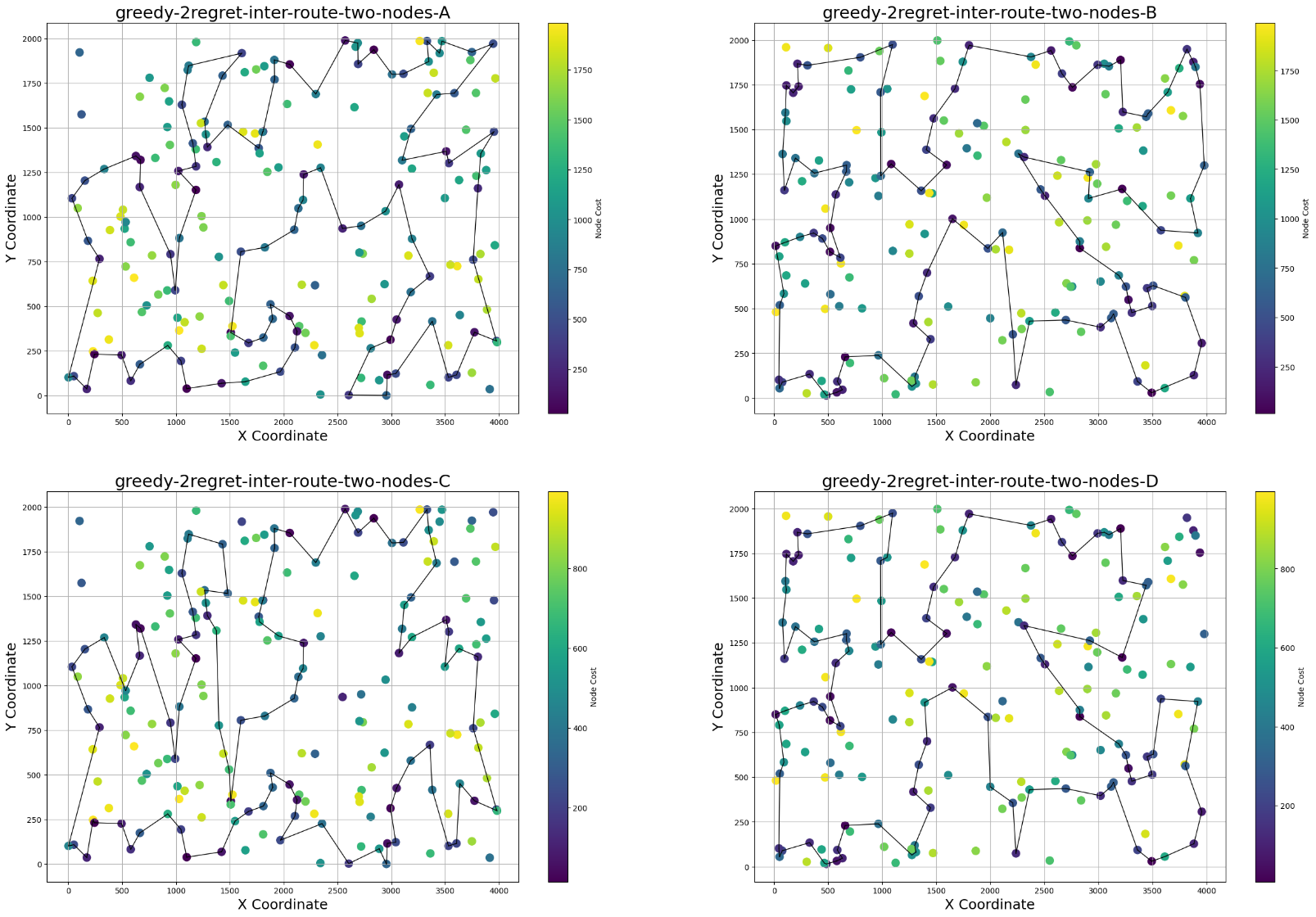






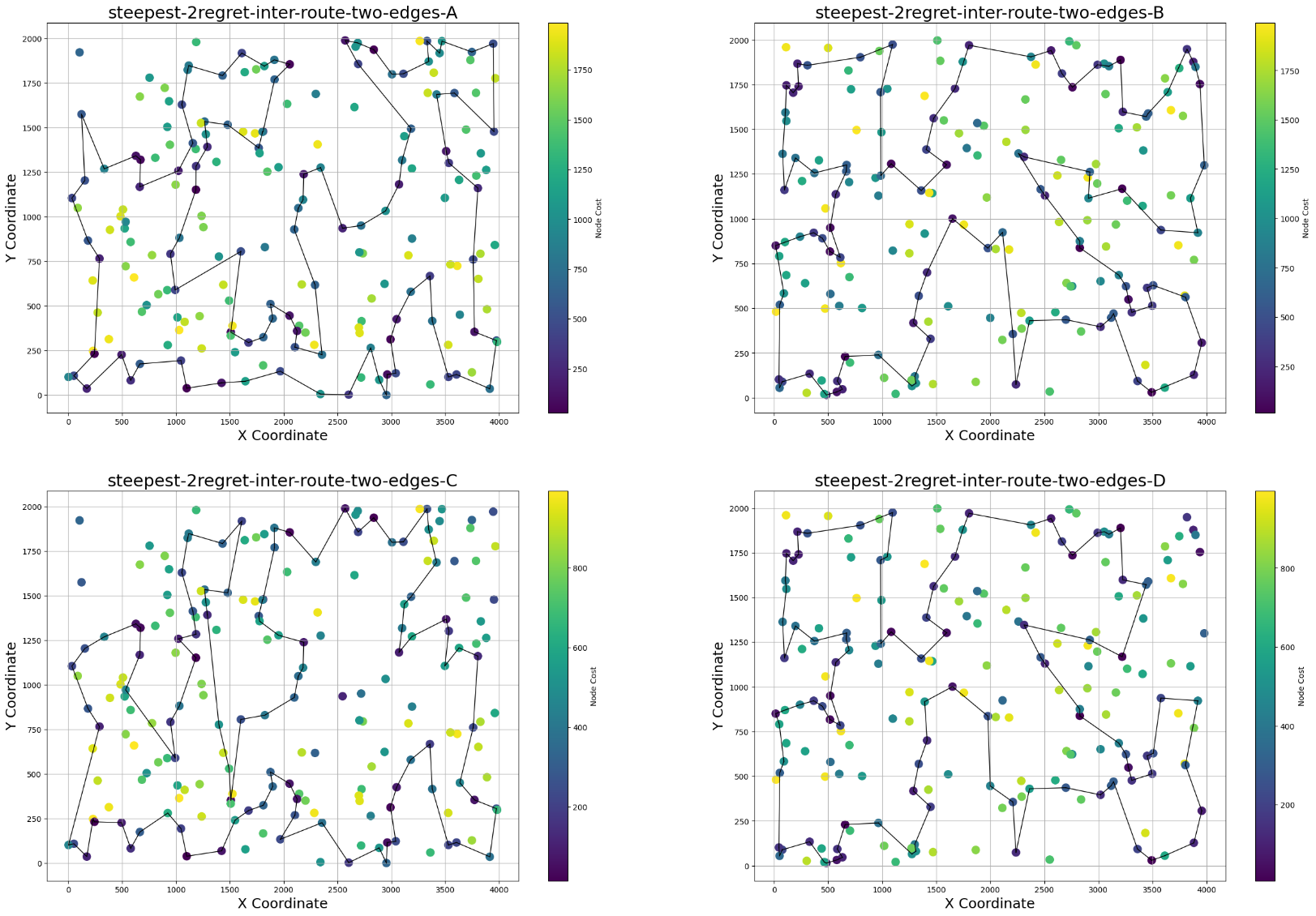
Obraz zawierający tekst, diagram, linia

Opis wygenerowany automatycznie



Obraz zawierający tekst, diagram, linia, pismo odręczne

Opis wygenerowany automatycznie



**Conclusions**

As we may expect, both methods performed well in comparison to the previously implemented algorithms. Even the configurations that started with a random solution at the beginning obtained paths on a similar level to other heuristic algorithms. It might be observed that the initial solution is not the only factor that affects the final results. It is clearly seen that a setup with two-nodes exchange is almost always worse than with two-edges exchange and it seems to be fair since changing the edges around a good solutions might be more valuable than exchanging nodes that could be far away each other.

Surprisingly, the greedy local search usually was almost as good as the steepest version. For one instance, the best solution has been even found by only the greedy algorithm. It’s due to the fact, that greedy method is able to explore different areas of the solutions space avoiding stacking in the local minima. Sometimes, from the greedy local search algorithm, it is valuable to choose quite worse solution that in the future will result in exploring better areas. Since the steepest version doesn’t have such option, choosing always the best option in each step may lead to be stacked in the local minima.

According to the runtimes, there is no surprise that steepest local search is much slower than the greedy version. Since the algorithm always has to explore the whole solution space, it is much more challenging approach form the complexity point of view than just choosing the first better path what a greedy version does. It is worth to notice, that much longer experiments occurred only for the configurations with a random solution as an initial one, because in such cases, many epochs were needed to find the result. When the steepest local search started from already a very good solutions it was enough to stack in a local (or maybe global) minimum after even 6 epochs.