Evolutionary Computation

Assignment 5

Antoni Lasik 148287

Daniel Jankowski 148257

<https://github.com/JankowskiDaniel/evolutionary-computation/tree/AL/assignment5>

**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])

move = Move(removed\_edges, added\_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[i-1], solution.nodes[i]),

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

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

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

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

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

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

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

**CONTINUE**

**ELSE:**

score\_delta = (

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

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

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

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

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

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

**IF** score\_delta < 0:

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

**ELSE:**

# most common case, nodes are not adjacent

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

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

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

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

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

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

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

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

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

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

**CONTINUE**

**ELSE:**

score\_delta = (

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

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

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

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

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

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

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

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

**IF** score\_delta < 0:

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

**inter\_route\_exchange(solution, unselected, heap, dist\_matrix):**

n\_selected = len(solution)

n\_unselected = len(unselected)

index\_pairs = [(i, j) **FOR** i **IN** range(n\_selected) **FOR** j **IN** range(n\_uselected)]

**FOR** i, j **IN** index\_pairs:

selected\_node = solution.nodes[i]

new\_node = unselected.nodes[j]

new\_solution = solution.copy()

new\_solution[i] = new\_node

prev\_node\_index = (i-1)%n\_selected

next\_node\_index = (i+1)%n\_selected

removed\_edges = (Edge(solution.nodes[prev\_node\_index], selected\_node),

Edge(selected\_node, solution.nodes[next\_node\_index]))

added\_edges = (Edge(solution.nodes[prev\_node\_index], new\_node),

Edge(new\_node, solution.nodes[next\_node\_index]))

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

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

**CONTINUE**

**ELSE:**

score\_delta = (

-dist\_matrix[solution.nodes[prev\_node\_index]][selected\_node]

-dist\_matrix[selected\_node][solution.nodes[next\_node\_index]]

+dist\_matrix[solution.nodes[prev\_node\_index]][new\_node]

+dist\_matrix[new\_node][solution.nodes[next\_node\_index]]

-selected\_node.cost

+new\_node.cost

**IF** score\_delta < 0:

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

In our implementation, when a given move was accepted, we just removed edges that where provided to be removed by a move, and append edges that a given move introduced. However, this haven’t ensured us, that a new list of edges is in a proper order. Therefore, the following method has been implemented.

**order\_edges(edges):**

ordered\_edges = [edges.pop(0)]

**WHILE** edges:

last\_destination = ordered\_edges[-1].destination

**FOR** i **IN** **RANGE**(len(edges):

**IF** edges[i].source == last\_destination:

ordered\_edges.append(edges.pop(i))

**BREAK**

**RETURN** ordered\_edges

**apply\_move(solution, current\_score, move, delta\_score)**

**//TODO**

**run\_algorithm(start\_solution, dist\_matrix, costs, moves)**

current\_score = objective\_function(start\_solution, dist\_matrix, costs)

progress = True

**WHILE** progress:

**FOR** move\_method **IN** moves:

# perform a given move, called move was adding deltas to heap

# to which class has access, therefore it was enough to call a method

# to explore new moves

move\_method()

# a list for storing moves that won’t be accepted, but at the same time # shouldn’t be removed from LM

temp\_moves = []

progress = False

**WHILE** moves exists **IN** LM:

delta, move = heap.heappop() # take the best move

**IF** move.type == INTER\_ROUTE\_EXCHANGE:

applicability = is\_inter\_move\_applicable(solution, move)

**ELSE:**

applicability = is\_intra\_move\_applicable(solution, move)

**IF** applicability == -1:

**CONTINUE**

**ELIF** applicability == 0:

temp\_moves.append((score, move))

**ELSE:**

# Since each method was implemented inside a class, the # apply\_move method overwrites the current solution with a

# new one. It was enough to call a function

apply\_move(solution, current\_score, dist\_matrix, move, delta)

progress = True

**BREAK**

**FOR** move **IN** temp\_moves:

heap.add\_move(move)

**RETURN** solution, score

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 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, 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)** |
| Candidates LS, random solution, two-edges + inter route | 81,129(76,609-86,447) | 73,977(69,300-80,189) | 51,588(49,120-54,801) | 48,429(45,385-51,392) |
| 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) |
| Deltas from previous iteration, random solution, two-edges + inter route | 78,003(74,745-81,695) | 71,335(68,691-75,555) | 51,498(49,130-54,727) | 48,311(44,942-52,111) |
| Deltas from previous iteration, random solution, two-nodes + inter route |  |  |  |  |

**Results**

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

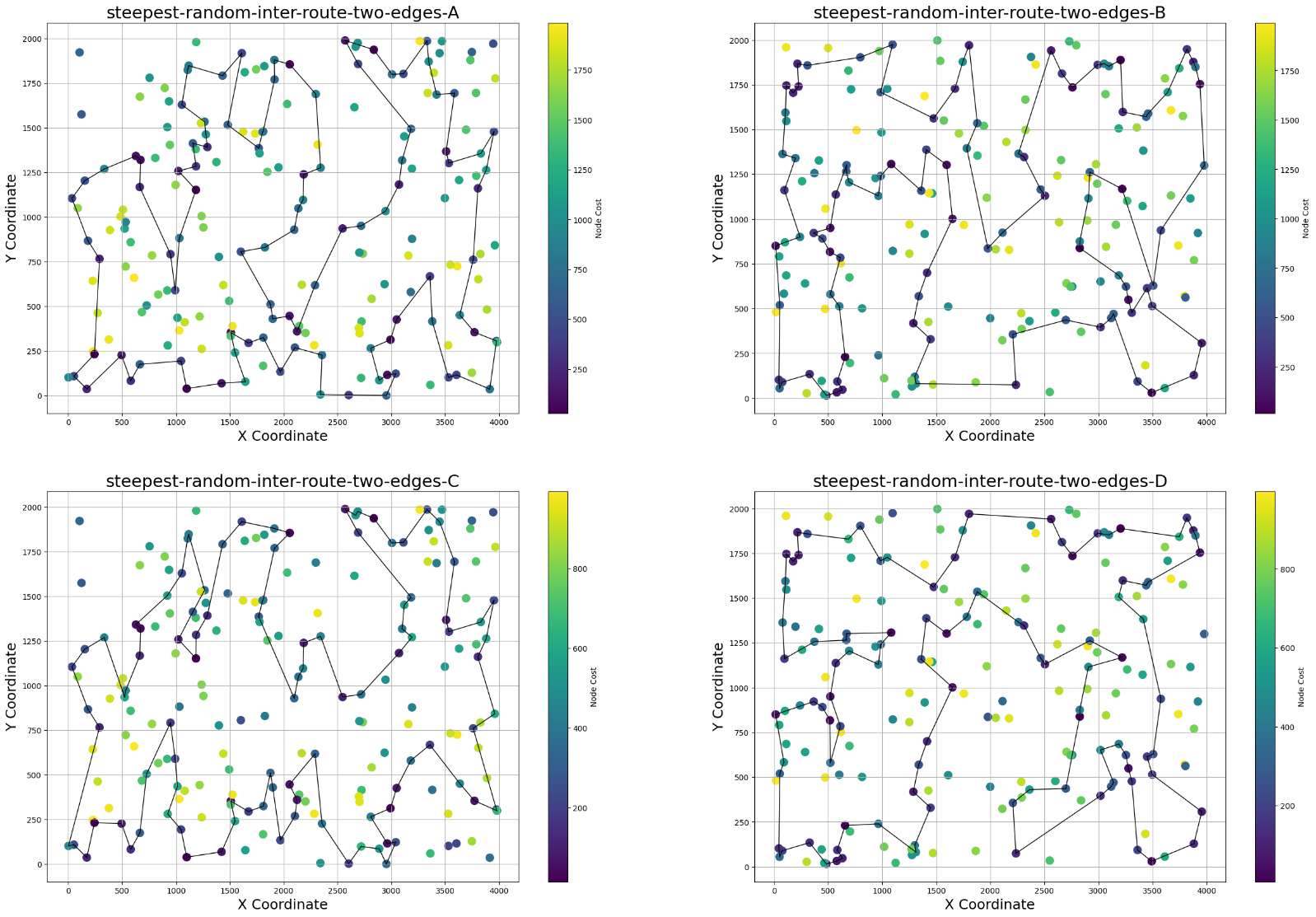
*\*\*in bold there are the best minimal solutions founded by a given method*

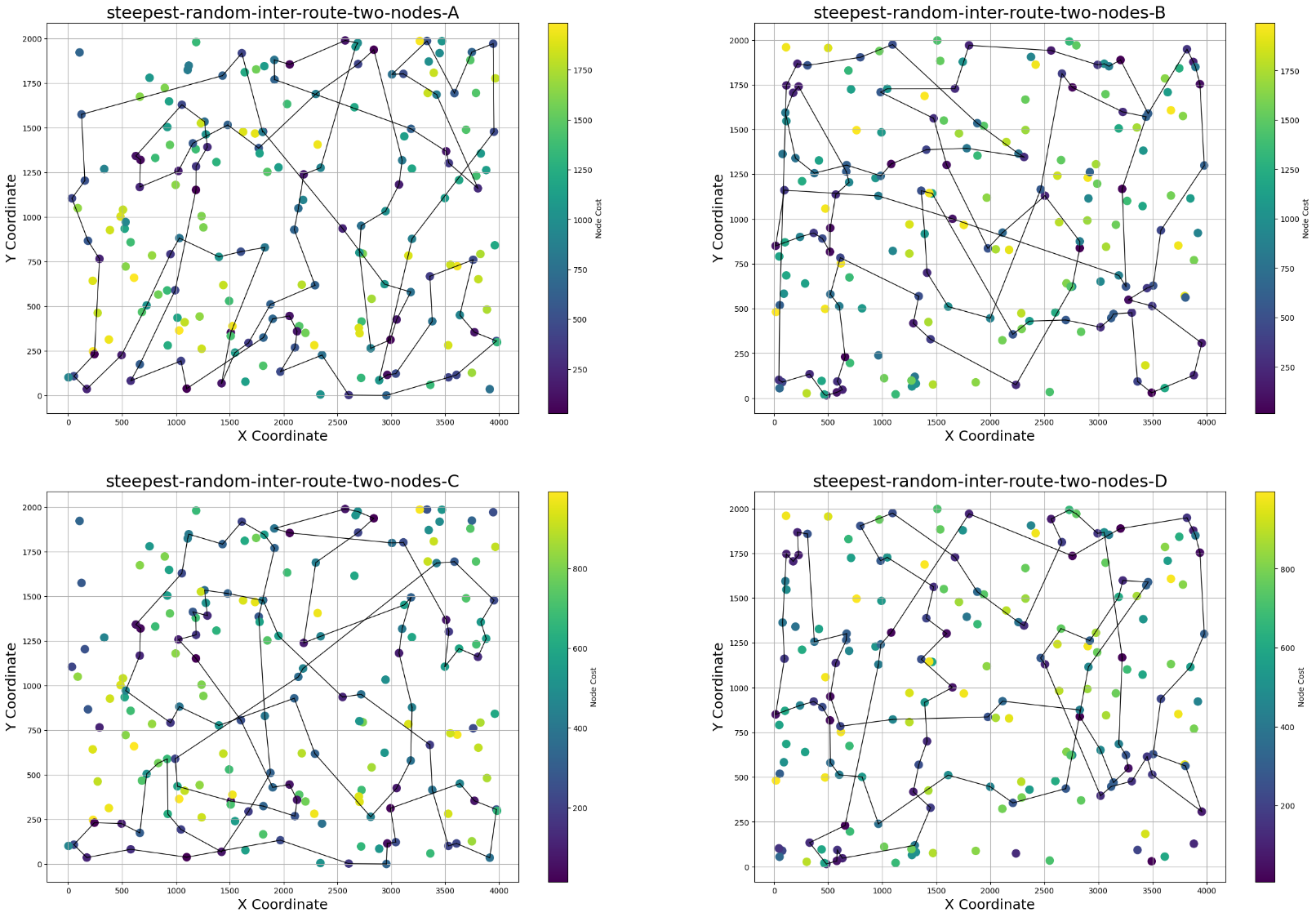
**Runtimes**

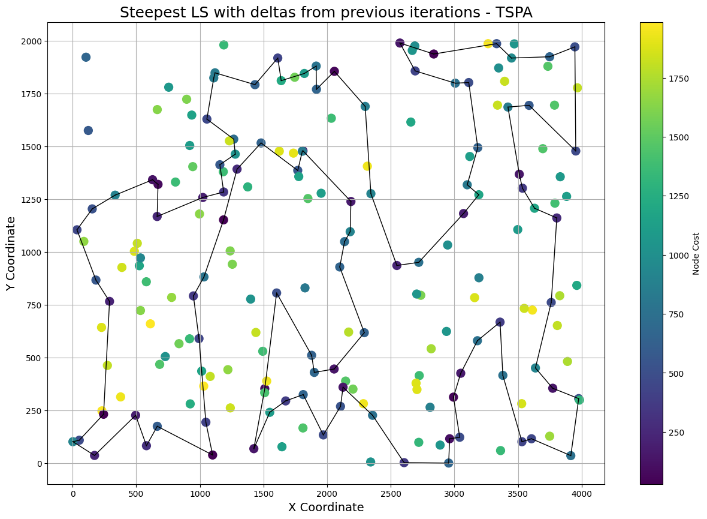
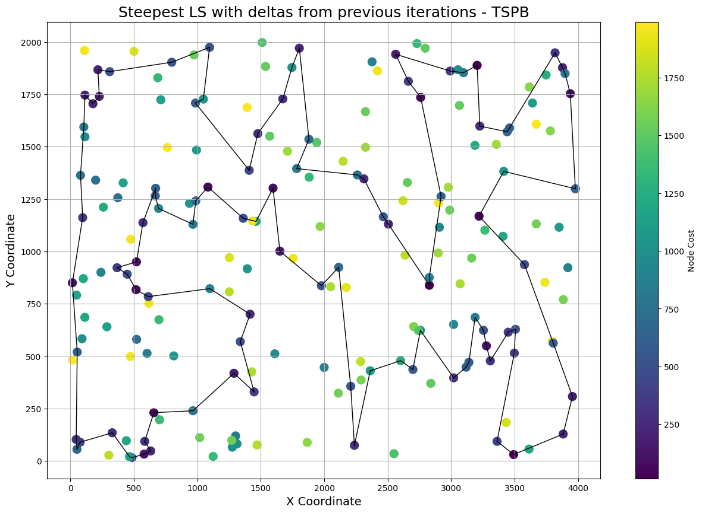
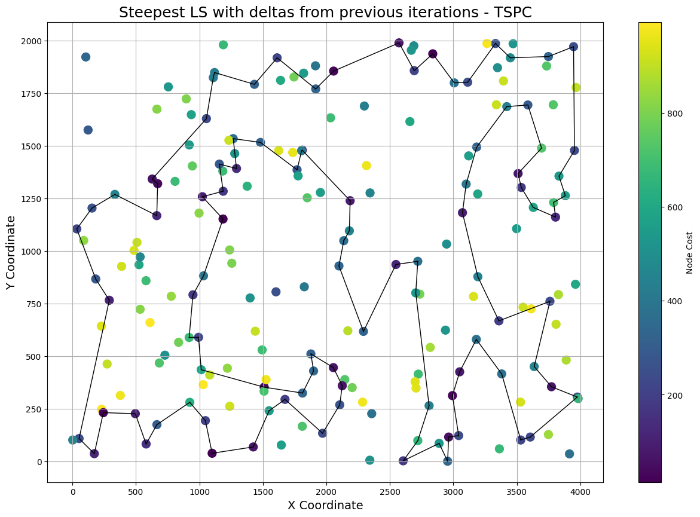
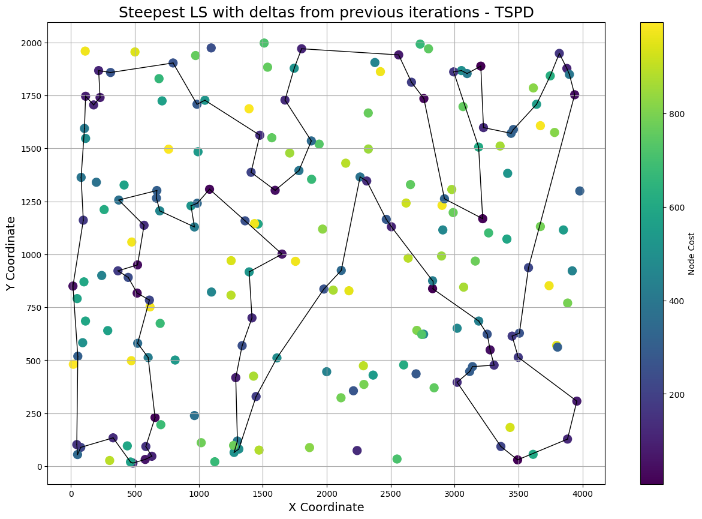
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Method | Instance A | Instance B | Instance C | Instance D |
| Greedy LS, random solution, two-edges + inter route | 1.56(1.06-2.63) | 1.95(1.19-3.48) | 1.25(0.77-2.28) | 1.18(0.72-1.99) |
| Greedy LS, random solution, two-nodes + inter route | 1.68(1.03-2.98) | 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, 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) |
| Candidate LS, random solution, two-edges + inter route | 4.43(3.95-6.41) | 4.52(3.99-5.70) | 4.53(3.83-6.44) | 4.58(4.06-5.88) |
| 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) |
| Deltas from previous iteration, random solution, two-edges + inter route | 20.10(16.65-24.66) | 21.24(17.95-25.17) | 20.80(16.46-23.18) | 21.83(17.51-23.97) |
| Deltas from previous iteration, random solution, two-nodes + inter route |  |  |  |  |

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

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







**Conclusions**

Results obtained by the modified version of a Steepest Local Search, where we store deltas from previous iterations, are very similar to the classical version of the algorithm. It was expected, since the main idea for finding a best solution didn’t changed. The size of the neighborhood was the same, however, the way how we explore it was different. Instead of evaluating each move, we were computing delta score only for new moves, and considering only those, that provided improvement with respect to the current solution.

The purpose of this change was to reduce the runtime in comparison to the classical version of the Steepest Local Search. Unfortunately, in our implementation, instead of reducing the runtime, we’ve observed significant increase of time needed to perform a single run of the algorithm. At first glance, there should be no such big difference in a runtime, because the complexity of both implementation looks very similar. After spending some time to investigate what caused such surprising behaviour, we’ve spotted the potential issue related not strictly to the algorithm and its complexity, but rather to the technical aspects of our implementation and the nature of Python language.

In the report 3, where we’ve implemented the classical version of the Steepest Local Search, all operations associated with move creation, such changing the nodes, edges or computing deltas, were conducted in the simplest possible way basing on nodes indices in a solution list. This time, to make the algorithm easier to implement, and the code itself more clear, we’ve decided to declared data classes for core components needed to perform the Local Search for a TSP problem. Therefore, several classes has been implemented such: Node, Edge, Solution (constructed from Nodes and Edges), Move (constructed from Nodes, Edges nested in additional objects like RemovedEdges or AddedEdges). The usage of objects instead of the simplest list-indices-based approach leaded to significant slowdown of the program. The main reason of a inefficiency was that operating on objects in Python programming language occurred to be visibly slower than on any other built-in structures. For example, according the Python official documentation, checking if an element exists in a hash table (in Python, we may consider a hash table as a dictionary) has an average complexity O(1), and the O(n) for the worst case, which depends on the hash function. We’ve compared the efficiency of storing some basic data types in a hash table, and it seemed that for our case the complexity of checking if, e.g. a Move object exists in a dictionary always converged to O(n). In general, we observed that even operations such declaring objects for components like Solution, Move, are much slower than just referencing appropriate indices.