Evolutionary Computation

Assignment 3

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 are implementing local search algorithm in two versions – greedy and steepest. Both algorithms in each step will try to find a better result in the neighborhood of the current the best solution. Every epoch algorithm will create a new solutions using three methods: two-edges exchange, two-nodes exchange (these two are called intra moves, since they exchange only nodes that already exist in the solution) and the third one called intra-route-exchange, when we exchange one node from the current solution with the one from unselected ones.

A greedy version of the algorithm every step randomly choose the move to be taken at first and then in random order (starting at random index in a random direction) is looking for a better solution. Always the first better founded solution is accepted. In contrast, the steepest version of the algorithm is always exploring the whole neighborhood of each method, and choose only the best path among all methods.

Both methods are run with two types of starting solutions, as well as with different configuration of moves. In a single run a starting solution is always either a random generated one or the best solution founded by the greedy cycle 2-regret algorithm with weighted sum, presented in the previous assignment. Each moves configuration consists of inter and one of the intra moves.

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

Two_nodes_exchange(current_solution, score, distance_matrix, start_index=0, direction='right'):
    n = len(current_solution)
```

```
 new_solutions = []
total_moves = n * (n - 1) // 2  # Total number of possible swaps
//Defining all of the possible pairs for swaps
index_pairs = [(x, y) for x in range(n) for y in range(x+1, n)]

//We either enumerate to the left or right of the starting point
IF direction == 'left':
        index_pairs = index_pairs[::-1]
        start_index = total_moves - start_index - 1

FOR pair(i,j)  in index_pairs[:start_index]:
        temp = current_solution[:]
        temp_score = score
        score_delta=(
                //Subtracting the distances of the edges that will change
                -distance_matrix[current_solution[i - 1]][current_solution[i]]
                -distance_matrix[current_solution[j - 1]][current_solution[j]]
                -distance_matrix[current_solution[i]][current_solution[(i + 1) % n]]
                -distance_matrix[current_solution[j]][current_solution[(j + 1) % n]]

                //Adding the distances of the new edges
                +distance_matrix[current_solution[i - 1]][current_solution[j]]
                +distance_matrix[current_solution[j - 1]][current_solution[i]]
                +distance_matrix[current_solution[i - 1]][current_solution[j]]
                +distance_matrix[current_solution[j - 1]][current_solution[i]]
                )
        //performing the exchange of nodes
        temp[i], temp[j] = temp[j], temp[I]

        temp_score += score_delta
        *FOR GREEDY*

        //If the new score is better, return the new solution immediately
        if temp_score < score:

                RETURN temp, temp_score

//If no improvement is found, return None
RETURN None, None

        *FOR STEEPEST*

        ADD (temp, temp_score) to new_solutions

//return only the best solution from the neighborhood
sorted_solutions = sorted(new_solutions, key=lambda x: x[1])

RETURN sorted_solutions[0][0], sorted_solutions[0][1]



Two_edges_exchange(current_solution,  current_distance,  distance_matrix, start_index, direction:
str = "right"):
 new_solutions = []
//Setting the ranges for the direction of iteration
IF direction == "right":
        range_i = range(n - 2)
        range_j = lambda i: range(i + 2, n)
ELSE:  // direction == "left"
        range_i = range(n - 3, -1, -1)
        range_j = lambda i: range(n - 1, i + 1, -1)

count = 0
FOR i in range_i:
        FOR j in range_j(i):
                IF count >= start_index:
                 //Perform the two-edges exchange from this point by inverting the order of
                        nodes between these two points
                 new_solution = (current_solution[:i + 1]
```

```
                    + current_solution[i + 1:j + 1][::-1]
                    + current_solution[j + 1:])

                    score_delta = (
                        // subtracting the distances of edges between the nodes and the part to
                            be inverted
                        -distance_matrix[current_solution[i]][current_solution[i + 1]]
                        -distance_matrix[current_solution[j]][current_solution[(j + 1) % n]]
                        // adding the distances of edges between the nodes and inverted
                            part
                        +distance_matrix[current_solution[i]][current_solution[j]]
                        +distance_matrix[current_solution[i + 1]][current_solution[(j + 1) %
                            n]]
                        )
                    new_score = current_distance + score_delta
                    *FOR GREEDY*

                    //If the new score is better, return the new solution immediately
                    IF new_score < current_distance:

                        RETURN new_solution, new_score

                count += 1  //Increment the counter after checking the condition
        RETURN None
                    *FOR STEEPEST*
                    ADD (new_solution, new_score) to new_solutions

        //return only the best solution from the neighborhood
         sorted_solutions = sorted(new_solutions, key=lambda x: x[1])

        RETURN sorted_solutions[0][0], sorted_solutions[0][1]


Inter_route_exchange(current_solution, unselected_nodes, distance_matrix, costs, start_index=0,
direction="right")
        n_selected = len(current_solution)
        n_unselected = len(unselected_nodes)
        current_score = objective_function(current_solution, distance_matrix, costs)
        //Create all possible combinations of selected and unselected nodes
         all_combinations = [(i, j) for i in range(n_selected) for j in range(n_unselected)]
        all_scores = []
        method_best_score = float("inf")
        method_best_solution = None
        method_best_new_node = None
        method_best_old_node = None

        IF direction == "left":
                all_combinations = all_combinations[::-1]
        FOR i, j in all_combinations[start_index:]:
                selected_node = current_solution[i]
                new_node = unselected_nodes[j]
                new_solution = current_solution.copy()
                new_solution[i] = new_node
                prev_node_index = (i - 1) % n_selected
                next_node_index = (i + 1) % n_selected
                score_delta = (
                    // subtract the distances of edges that the selected node had
                     -distance_matrix[current_solution[prev_node_index]][selected_node]
                     -distance_matrix[selected_node][current_solution[next_node_index]]
                    // add the distances of edges that new_node will have
                    +distance_matrix[current_solution[prev_node_index]][new_node]
                    +distance_matrix[new_node][current_solution[next_node_index]]
                    // subtract the cost of the selected node and replace it with the cost of the new
                        node
                    -costs[selected_node]
                    +costs[new_node]
                     )
                new_score = current_score + score_delta
                *FOR GREEDY*
```

```
        IF new_score < current_score:

                //remove from unselected nodes the new inserted one
                REMOVE new_node from unselected_nodes

                //add to unselected the node that has been dropped
                ADD selected_node to unselected_nodes

                RETURN new_solution, new_score

    //If no better solution is found, return None
    RETURN None, None

        *FOR STEEPEST*
        ADD new_score to all_scores
        IF new_score < method_best_score:
                method_best_score = new_score
                method_best_solution = new_solution
                method_best_new_node = new_node
                method_best_old_node = selected_node

    RETURN method_best_solution, method_best_score, "inter-route-exchange",
                                    method_best_new_node, method_best_old_node
```

```
GreedyLocalSearch.run(
self,
start_solution, moves//moves is a string list of types of neighbourhood used for this search,
moves_prob //probabilities for the choice of the neighbourhood)

    //parent class LocalSearch has a dict for each type of neighbourhood that stores as values the
        outcome of the functions presented above

    self.moves_probs = moves_prob

    //We calculate the objective function for the starting solution
    IF start_solution is not None:
        self.current_solution = start_solution
        self.current_score = objective_function(start_solution, self.distance_matrix,
            self.costs)

    //Flag for the stopping condition
    progress = True
    WHILE progress:
        //Random selection of the kind of move we are doing
        chosen_move = np.random.choice(moves, p=moves_prob)

        //We have defined utility function to randomly select start_index from which we start
            iteration and the direction of this iteration
        start_index, direction = self.moves_utils[chosen_move]()

        //Here is the hidden call to the neighbourhood function from which we select the new
            solution and obtain new score
        new_solution, new_score = self.moves[chosen_move](start_index=start_index,
            direction=direction)

        IF new_solution is not None: // if solution is not None it means that there is improvement
            self.current_solution = new_solution
            self.current_score = new_score
        ELSE:// it means that this direction doesn't produce any better solutions

            we try different direction for the same method

            IF  new_solution is not None: // if it improved we don't change the progress flag
                self.current_solution = new_solution
```

```
                        self.current_score = new_score

                ELSE: //It means that this neighbourhood doesn't produce any better solutions

                        we change the neighbourhood method to the second one in the primary direction

                        IF new_solution is not None: // if it improved we don't change the progress
                                                                        flag
                        self.current_solution = new_solution
                        self.current_score = new_score
                        ELSE: //this means that this direction doesn't produce better solutions

                                we change the direction of the second method

                                IF  new_solution is not None: // if it improved we don t change the
                                                                                progress flag
                                        self.current_solution = new_solution
                                        self.current_score = new_score
                                ELSE: //there is no possible improvement so we change the flag
                                        progress = False
        RETURN self.current_solution, self.current_score, epoch_counte


SteepestLocalSearch.run(
        self,
        moves)

        IF start_solution is not None:
                self.current_solution = start_solution
                 self.current_score = objective_function(start_solution, self.distance_matrix,
                        self.costs)

        //Flag for stopping condition
        progress = True

        WHILE progress:
                best_move_solutions = []
                improvement = 0
                FOR move in moves: //For kind of moves passed to the function add to the solutions to the
                                list best_move_solutions
                        IF move == "inter-route-exchange":
                                //Here we choose the best solutions of the set of neighbours of the
                                        current solution
                                new_solution, new_score, method, new_node, old_node = self.moves[move]()
                                best_move_solutions.append((new_solution, new_score, method, new_node,
                                        old_node))
                        ELSE:
                                new_solution, new_score, method = self.moves[move]()
                                best_move_solutions.append((new_solution, new_score, method))

                //Choosing the best solution from the best neighbour from each kind of
                        neighbourhood
                best_solution = min(best_move_solutions, key=lambda x: x[1])

                IF best_solution[1] < self.current_score:
                        improvement = 1
                        // if the solution is from inter-route-exchange neighbourhood we change the set
                        of unselected solutions
                        IF best_solution[2] == "inter-route-exchange"
                                self.unselected_nodes.remove(best_solution[3])
                                self.unselected_nodes.append(best_solution[4])
                //Change the current solution
                self.current_solution = best_solution[0]
                self.current_score = best_solution[1]

                IF improvement == 0:
                        progress = False
                        RETURN self.current_solution, self.current_score
```

Results

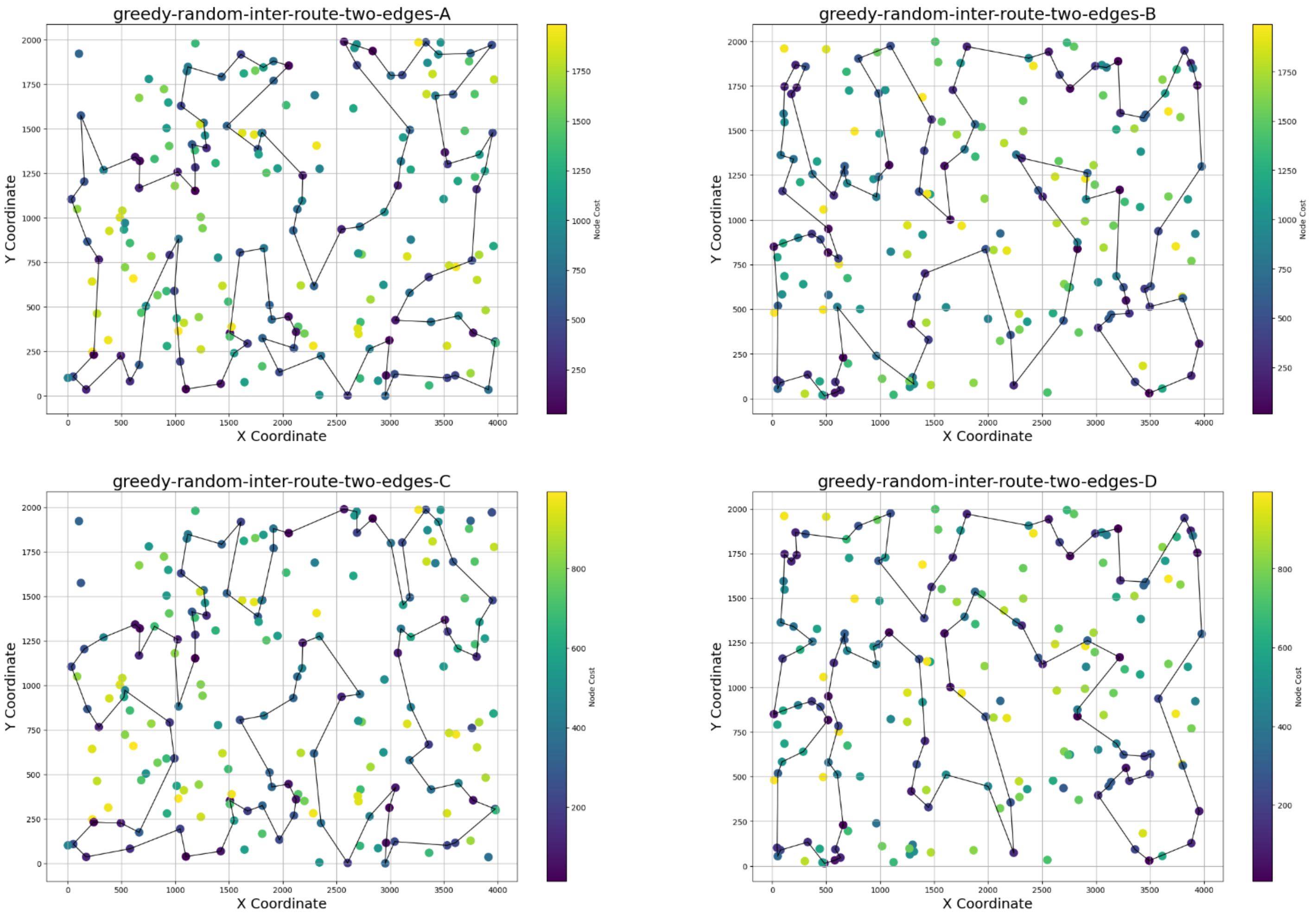
Method	Instance A	Instance B	Instance C	Instance D
Random solution	264,028(237,941-288,302)	266,665(243,288-295,269)	214,929(192,705-241,451)	219,678(191,218-242,515)
Nearest Neighbor	87,679(84,471-95,013)	79,282(77,448-82,631)	58,290(56,304-63,697)	54,290.68(50,335-59,846)
Greedy Cycle	76,711(75,136-80,025)	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*

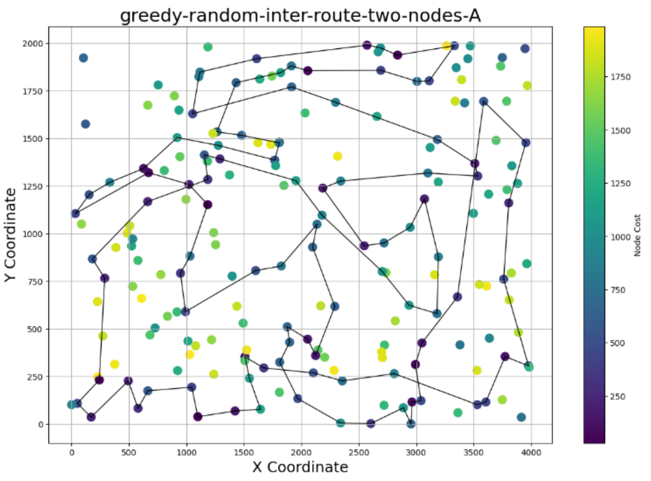
Runtimes

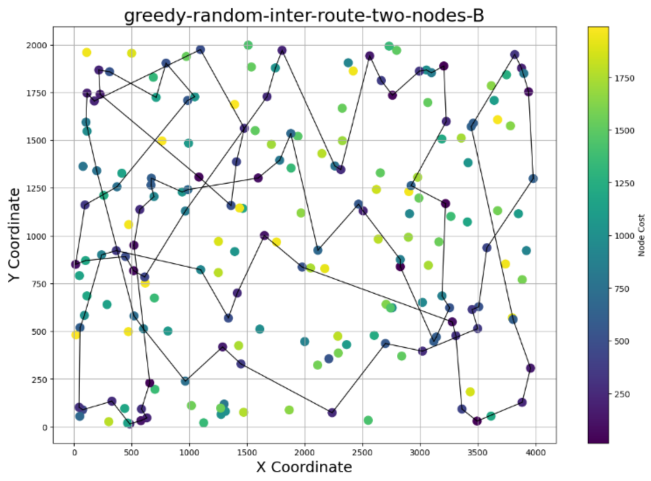
Method	Instance A	Instance B	Instance C	Instance D
Greedy LS, random solution, two-edges + inter route	1.56(1.06-2.63)	1.95(1.19-3.48)	1.25(0.77-2.28)	1.18(0.72-1.99)
Greedy LS, random solution, two-nodes + inter route	1.68(1.03-2.98)	1.95(0.81-6.66)	1.38(0.79-2.21)	1.37(0.77-2.36)
Greedy LS, best solution 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)

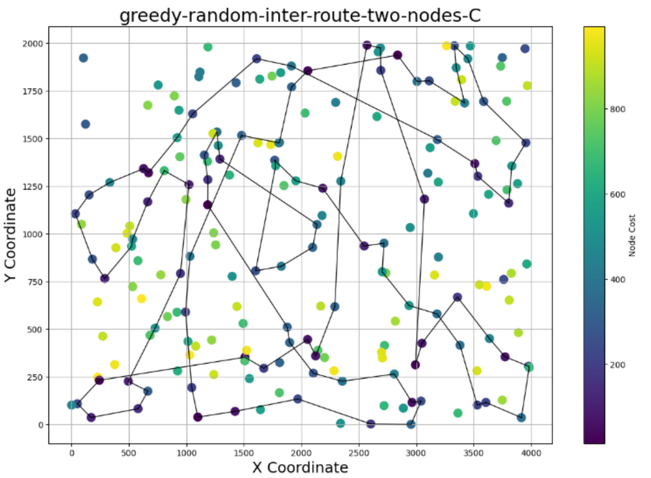
**the format is: avg(min-max)*
***all runtimes are provided in seconds.*

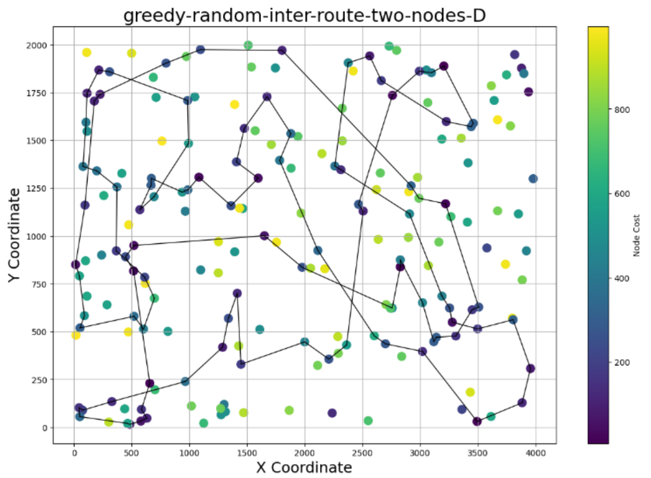


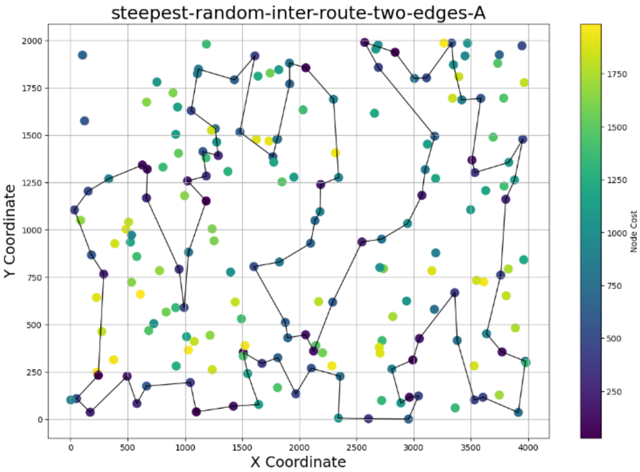
greedy-random-inter-route-two-edges-A
greedy-random-inter-route-two-edges-B
greedy-random-inter-route-two-edges-C
greedy-random-inter-route-two-edges-D
X Coordinate
Y Coordinate
Node Cost

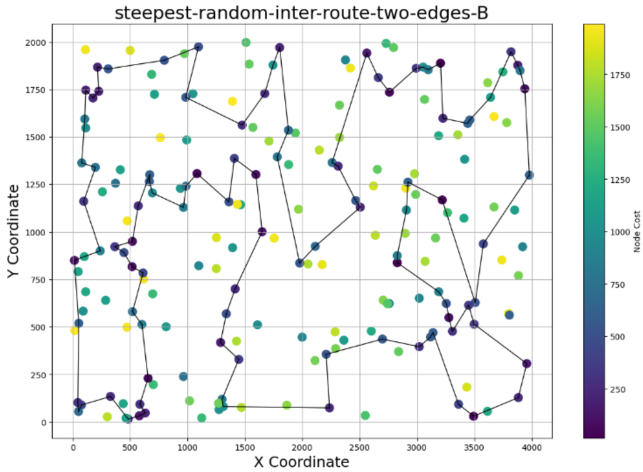

greedy-random-inter-route-two-nodes-A
Y Coordinate
X Coordinate
Node Cost

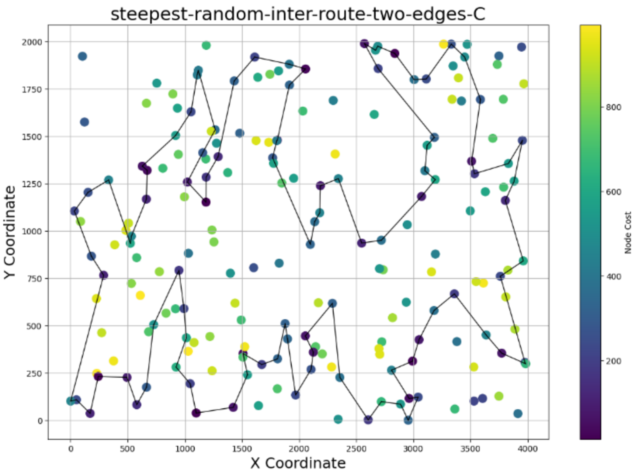

greedy-random-inter-route-two-nodes-B
Y Coordinate
X Coordinate
Node Cost

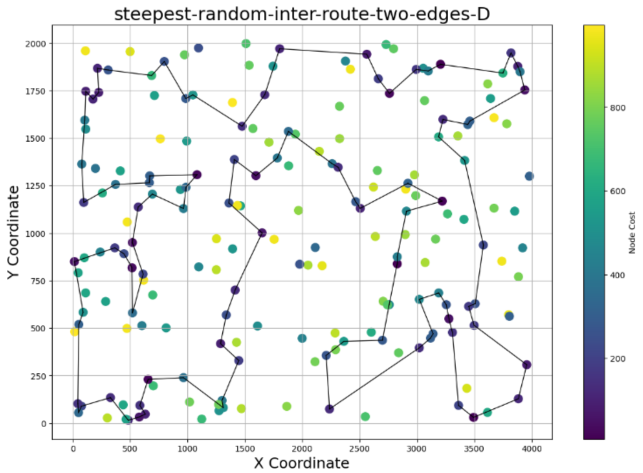

greedy-random-inter-route-two-nodes-C
Y Coordinate
X Coordinate
Node Cost

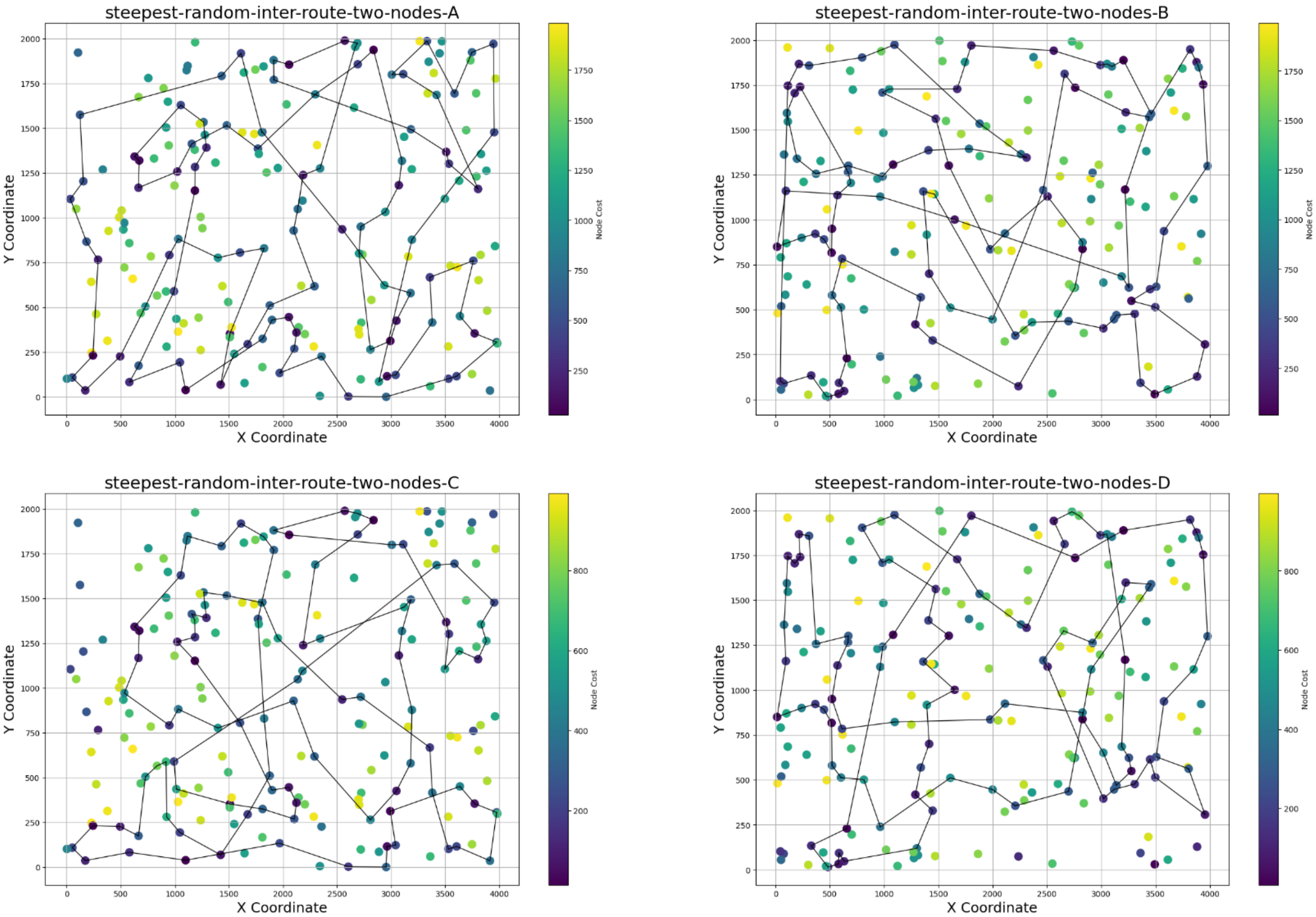

greedy-random-inter-route-two-nodes-D
Y Coordinate
X Coordinate
Node Cost

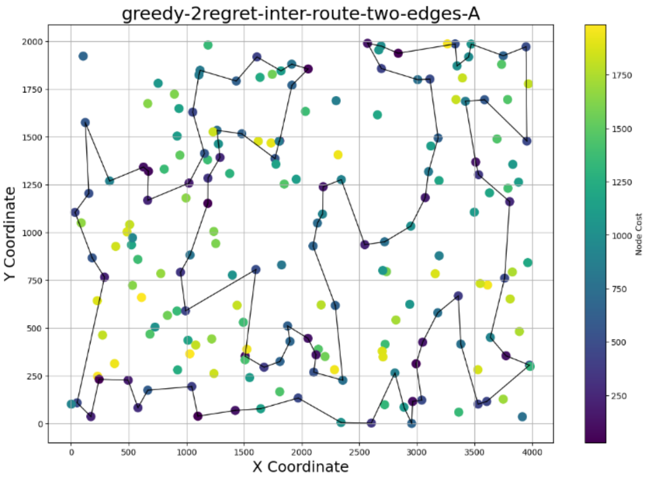

steepest-random-inter-route-two-edges-A
Y Coordinate
X Coordinate
Node Cost

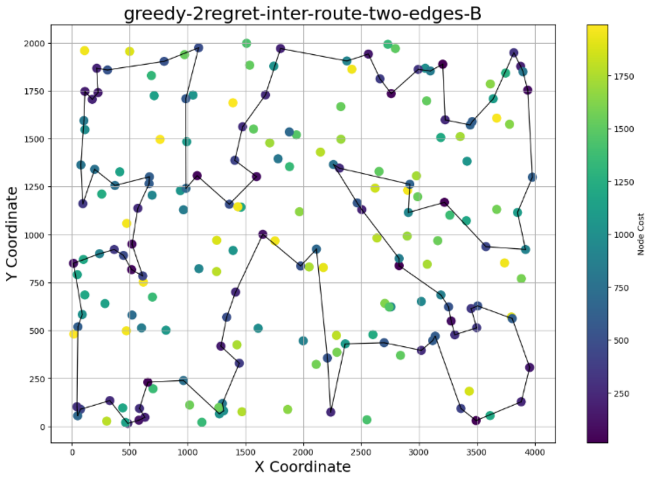

steepest-random-inter-route-two-edges-B
Y Coordinate
X Coordinate
Node Cost

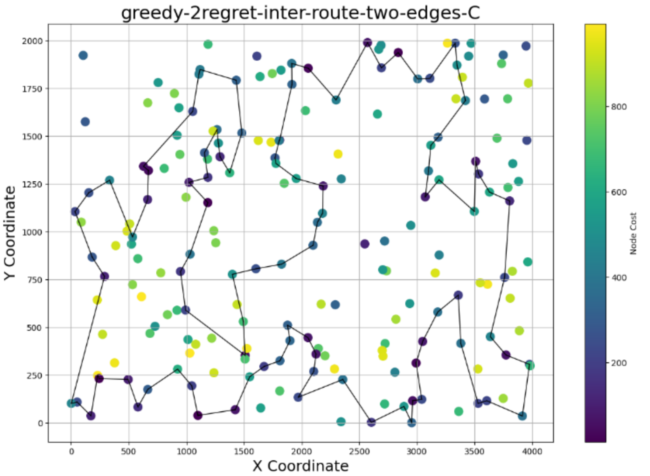

steepest-random-inter-route-two-edges-C
Y Coordinate
X Coordinate
Node Cost

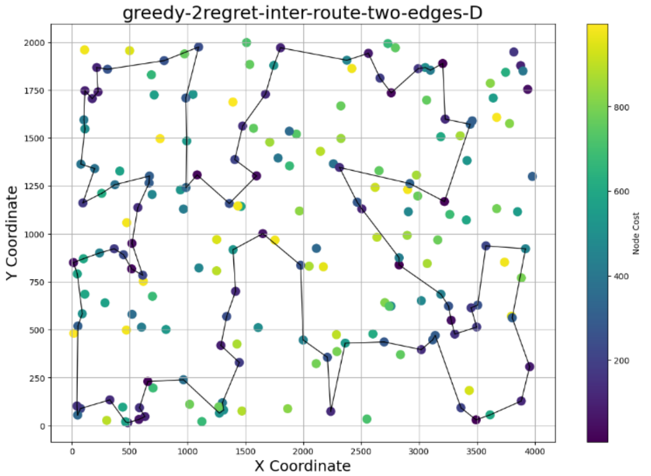

steepest-random-inter-route-two-edges-D
Y Coordinate
X Coordinate
Node Cost

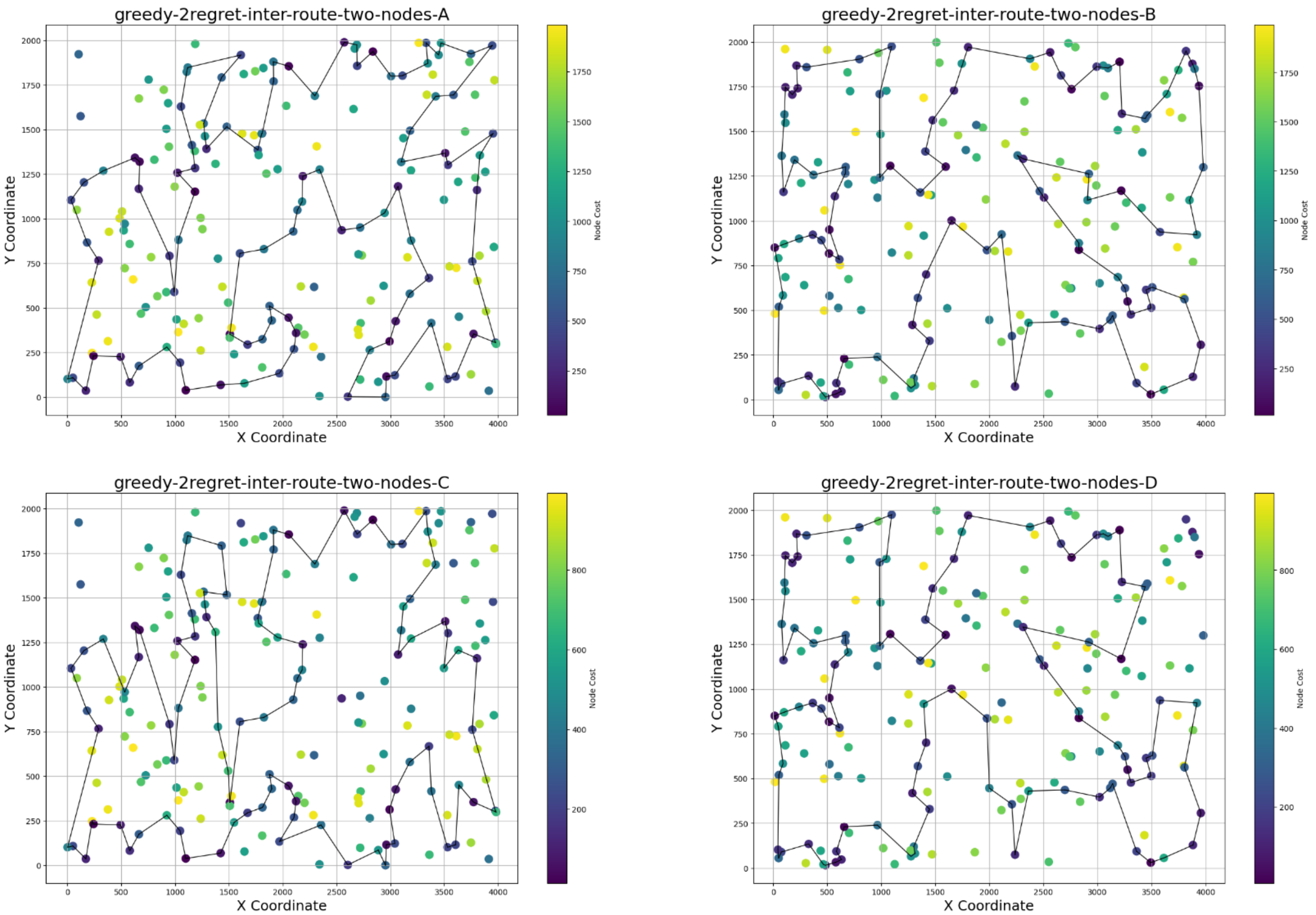

steepest-random-inter-route-two-nodes-A
steepest-random-inter-route-two-nodes-B
steepest-random-inter-route-two-nodes-C
steepest-random-inter-route-two-nodes-D
X Coordinate
Y Coordinate
Node Cost

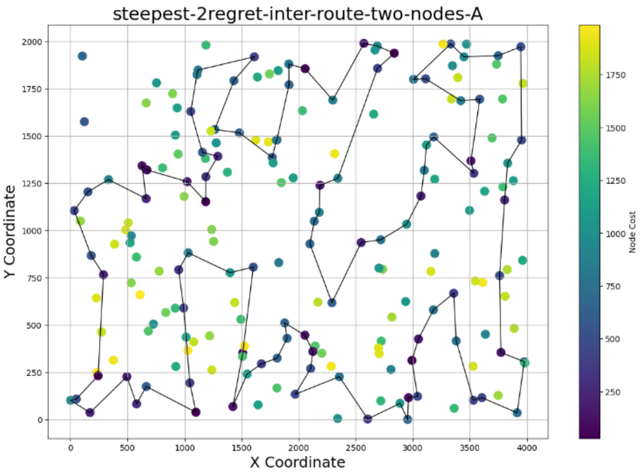

greedy-2regret-inter-route-two-edges-A
Y Coordinate
X Coordinate
Node Cost

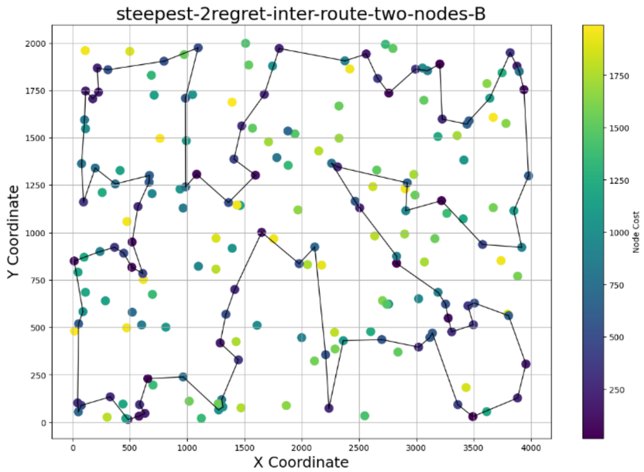

greedy-2regret-inter-route-two-edges-B
Y Coordinate
X Coordinate
Node Cost

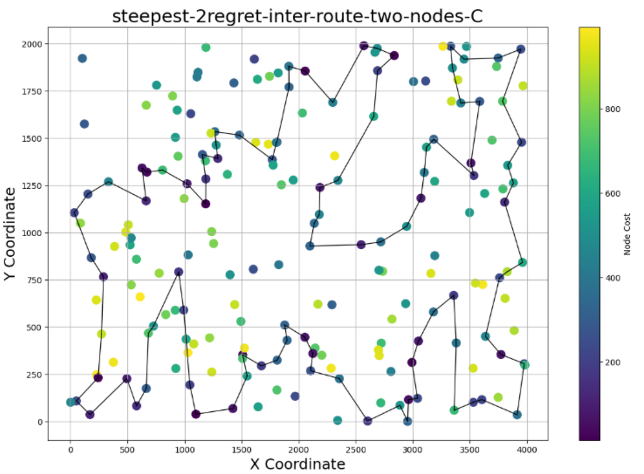

greedy-2regret-inter-route-two-edges-C
Y Coordinate
X Coordinate
Node Cost

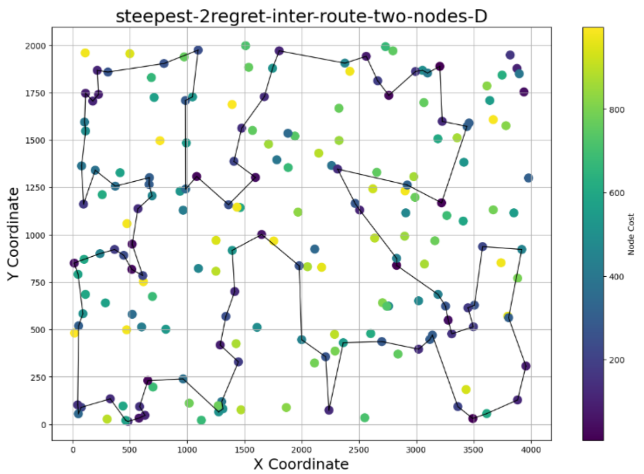

greedy-2regret-inter-route-two-edges-D
Y Coordinate
X Coordinate
Node Cost

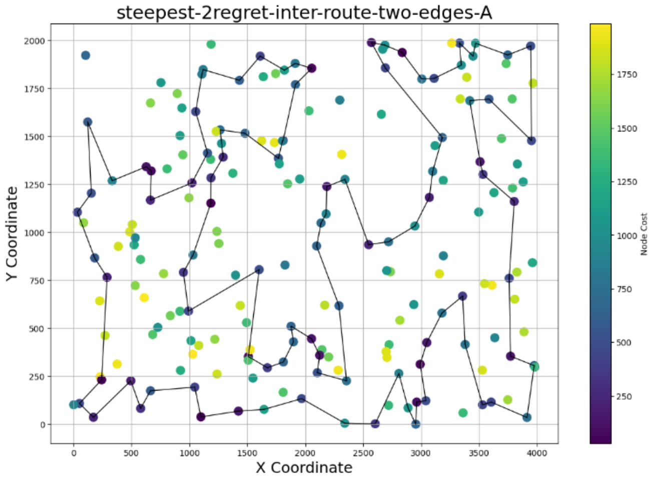

greedy-2regret-inter-route-two-nodes-A
Y Coordinate
X Coordinate
Node Cost
greedy-2regret-inter-route-two-nodes-B
Y Coordinate
X Coordinate
Node Cost
greedy-2regret-inter-route-two-nodes-C
Y Coordinate
X Coordinate
Node Cost
greedy-2regret-inter-route-two-nodes-D
Y Coordinate
X Coordinate
Node Cost

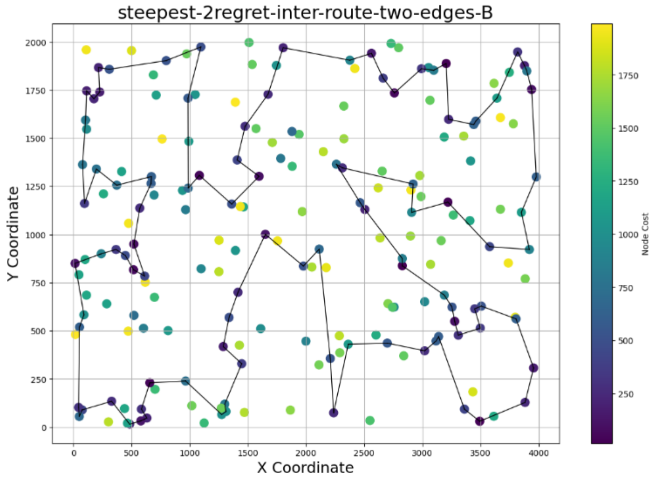

steepest-2regret-inter-route-two-nodes-A
Y Coordinate
X Coordinate
Node Cost
0
250
500
750
1000
1250
1500
1750
2000
0
500
1000
1500
2000
2500
3000
3500
4000
250
500
750
1000
1250
1500
1750

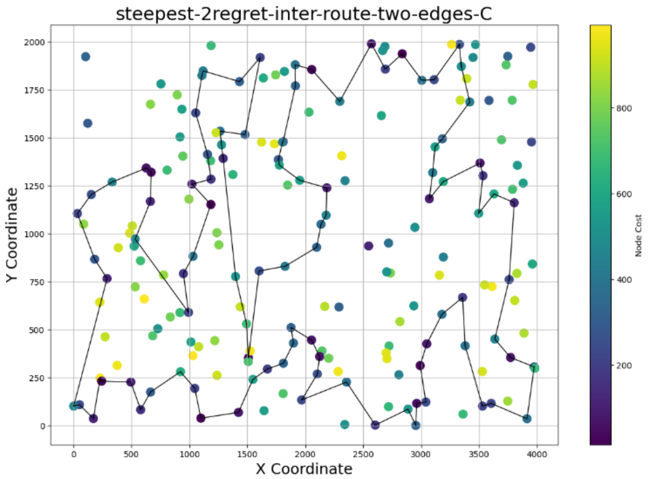

steepest-2regret-inter-route-two-nodes-B
Y Coordinate
X Coordinate
Node Cost
0
250
500
750
1000
1250
1500
1750
2000
0
500
1000
1500
2000
2500
3000
3500
4000
250
500
750
1000
1250
1500
1750

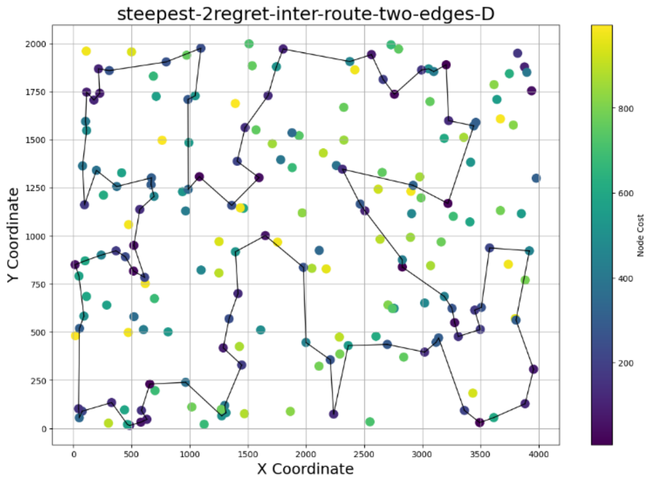

steepest-2regret-inter-route-two-nodes-C
Y Coordinate
X Coordinate
Node Cost
0
250
500
750
1000
1250
1500
1750
2000
0
500
1000
1500
2000
2500
3000
3500
4000
200
400
600
800


steepest-2regret-inter-route-two-nodes-D
Y Coordinate
X Coordinate
Node Cost
0
250
500
750
1000
1250
1500
1750
2000
0
500
1000
1500
2000
2500
3000
3500
4000
200
400
600
800


steepest-2regret-inter-route-two-edges-A
Y Coordinate
X Coordinate
Node Cost


steepest-2regret-inter-route-two-edges-B
Y Coordinate
X Coordinate
Node Cost


steepest-2regret-inter-route-two-edges-C
Y Coordinate
X Coordinate
Node Cost


steepest-2regret-inter-route-two-edges-D
Y Coordinate
X Coordinate
Node Cost

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.