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)*

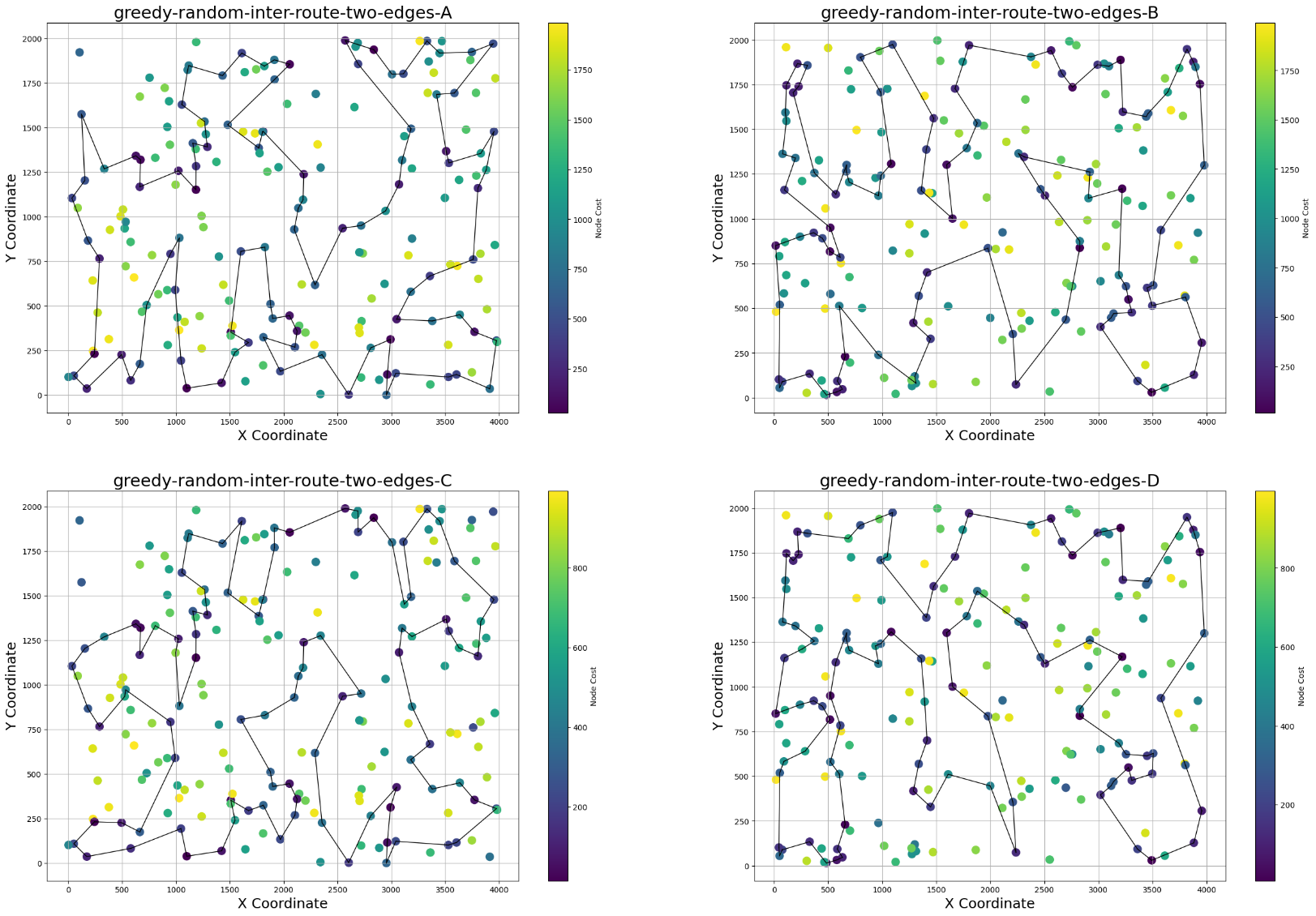
*\*\*in bold there are the best minimal solutions founded by a given method*

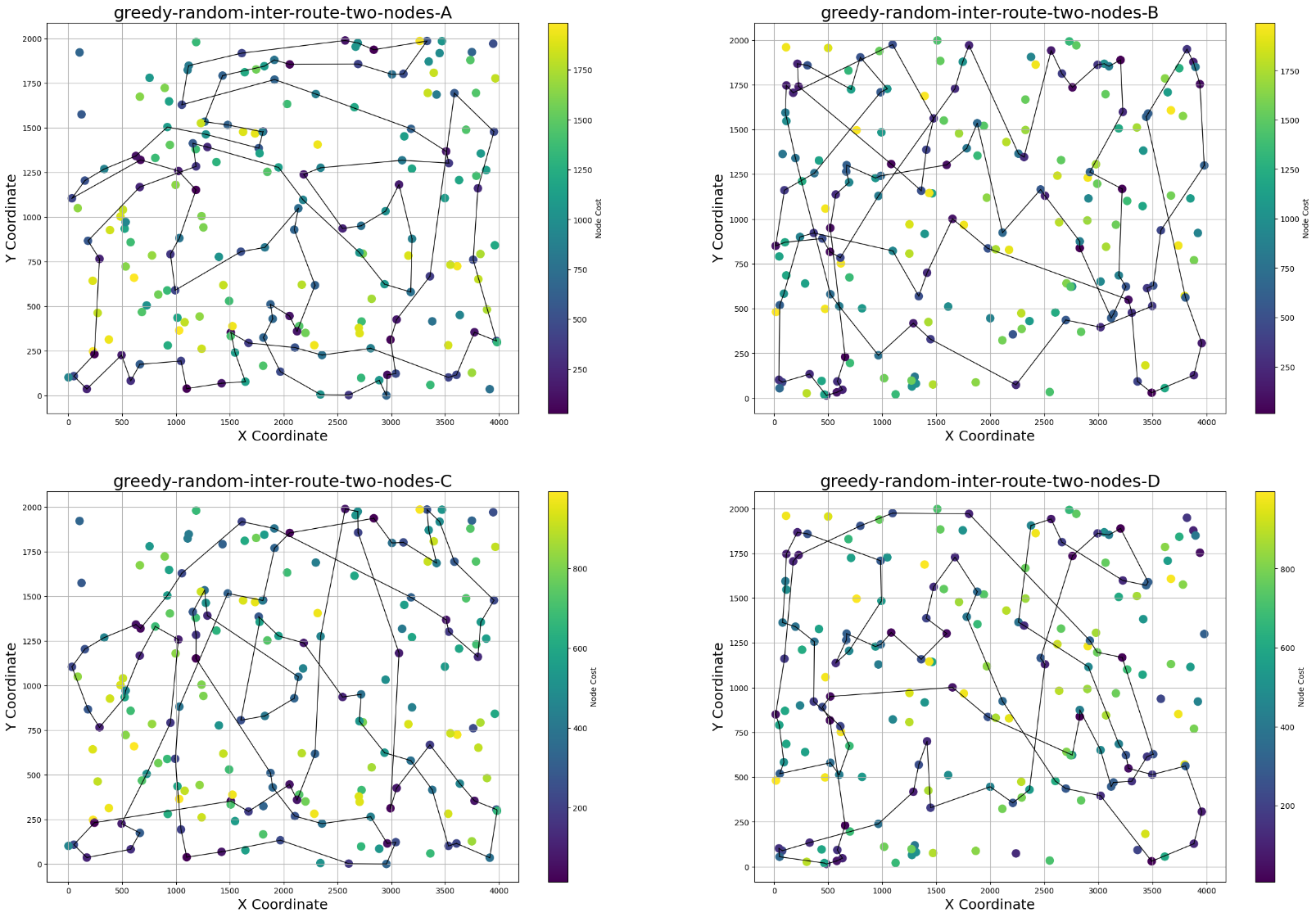
**Runtimes**

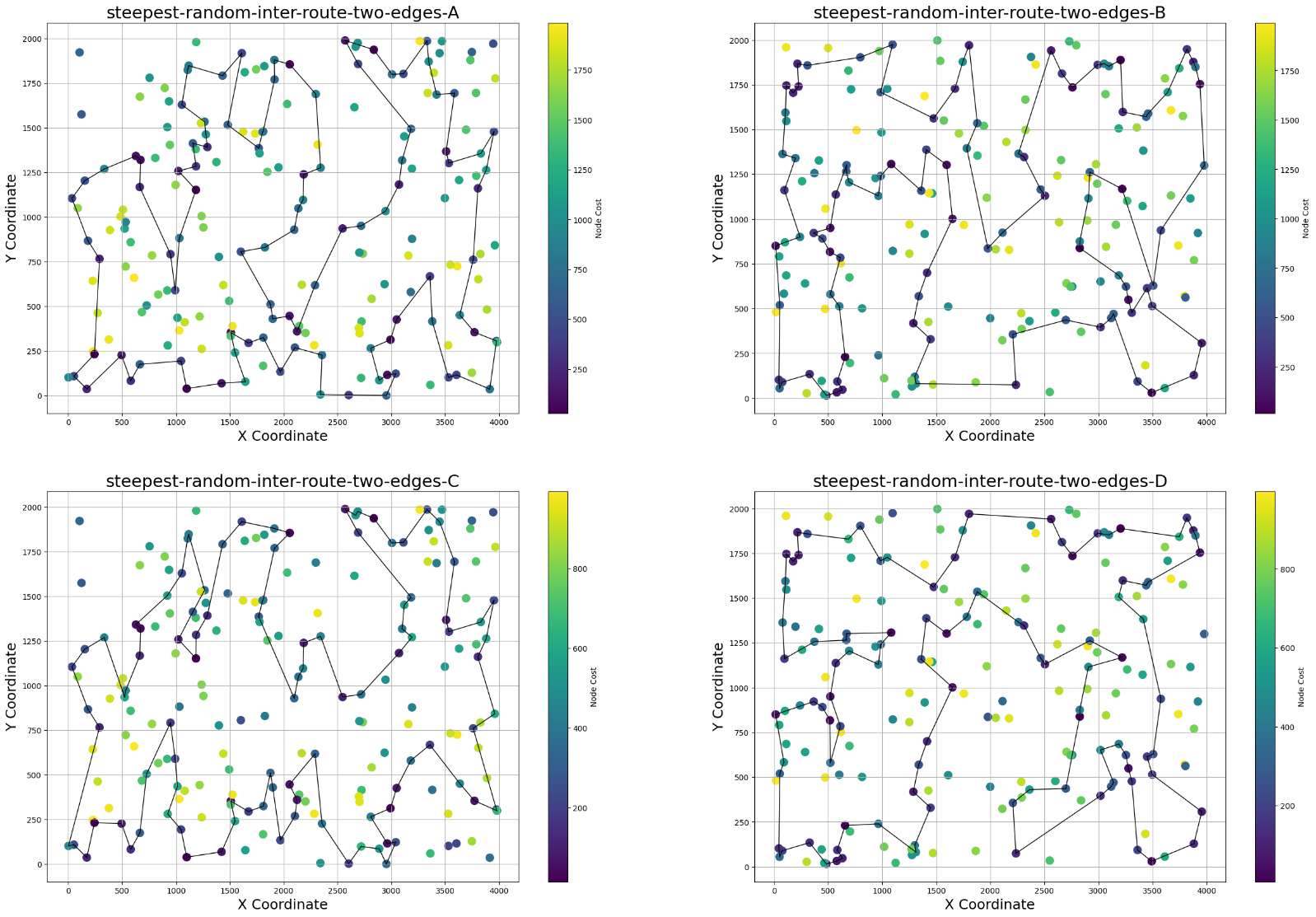
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Method | Instance A | Instance B | Instance C | Instance D |
| Greedy LS, random solution, two-edges + inter route | 1.56(1.06-2.63) | 1.95(1.19-3.48) | 1.25(0.77-2.28) | 1.18(0.72-1.99) |
| Greedy LS, random solution, two-nodes + inter route | 1.68(1.03-2.98) | 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) |

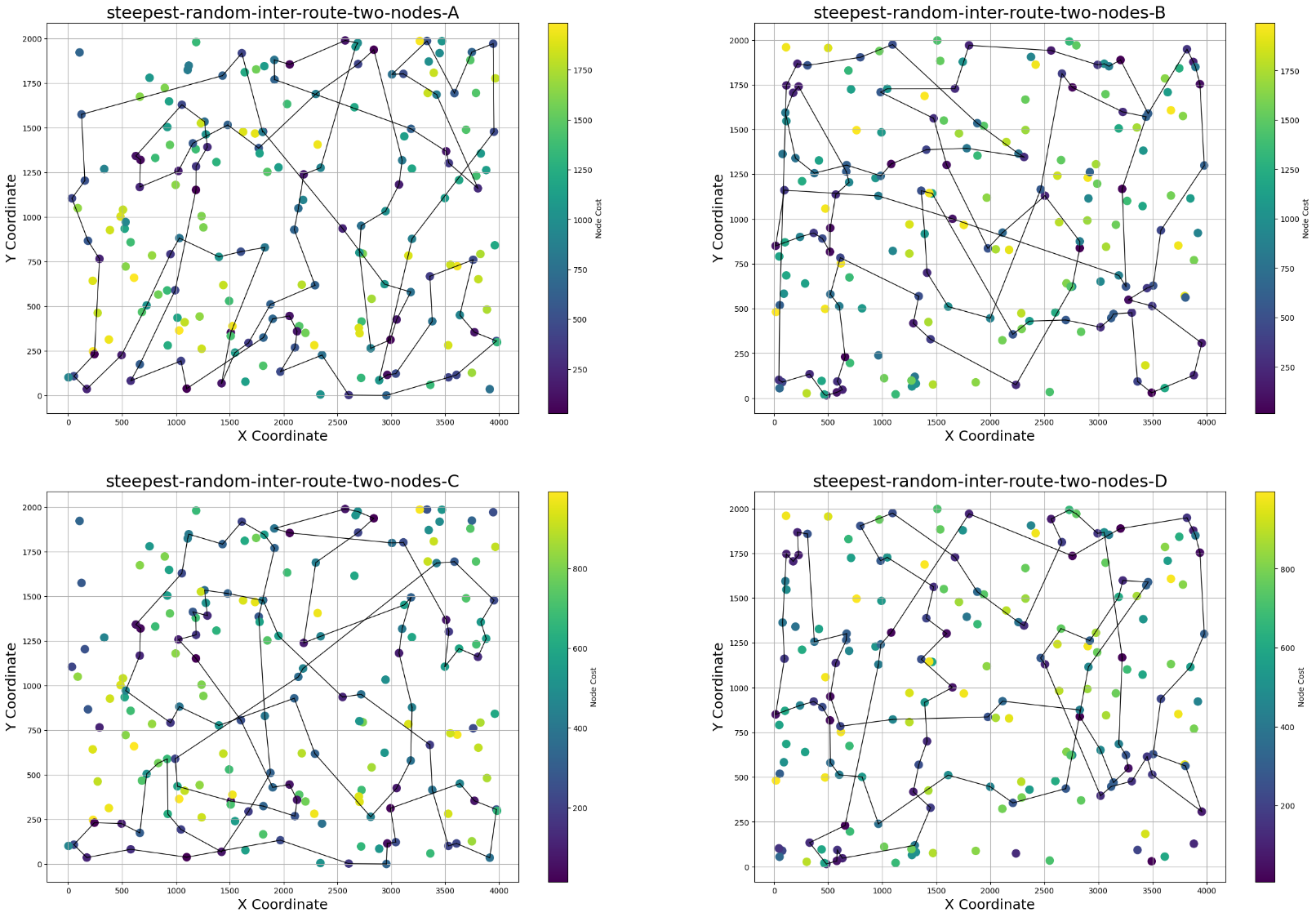
***\*****the format is: avg(min-max)*

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

**

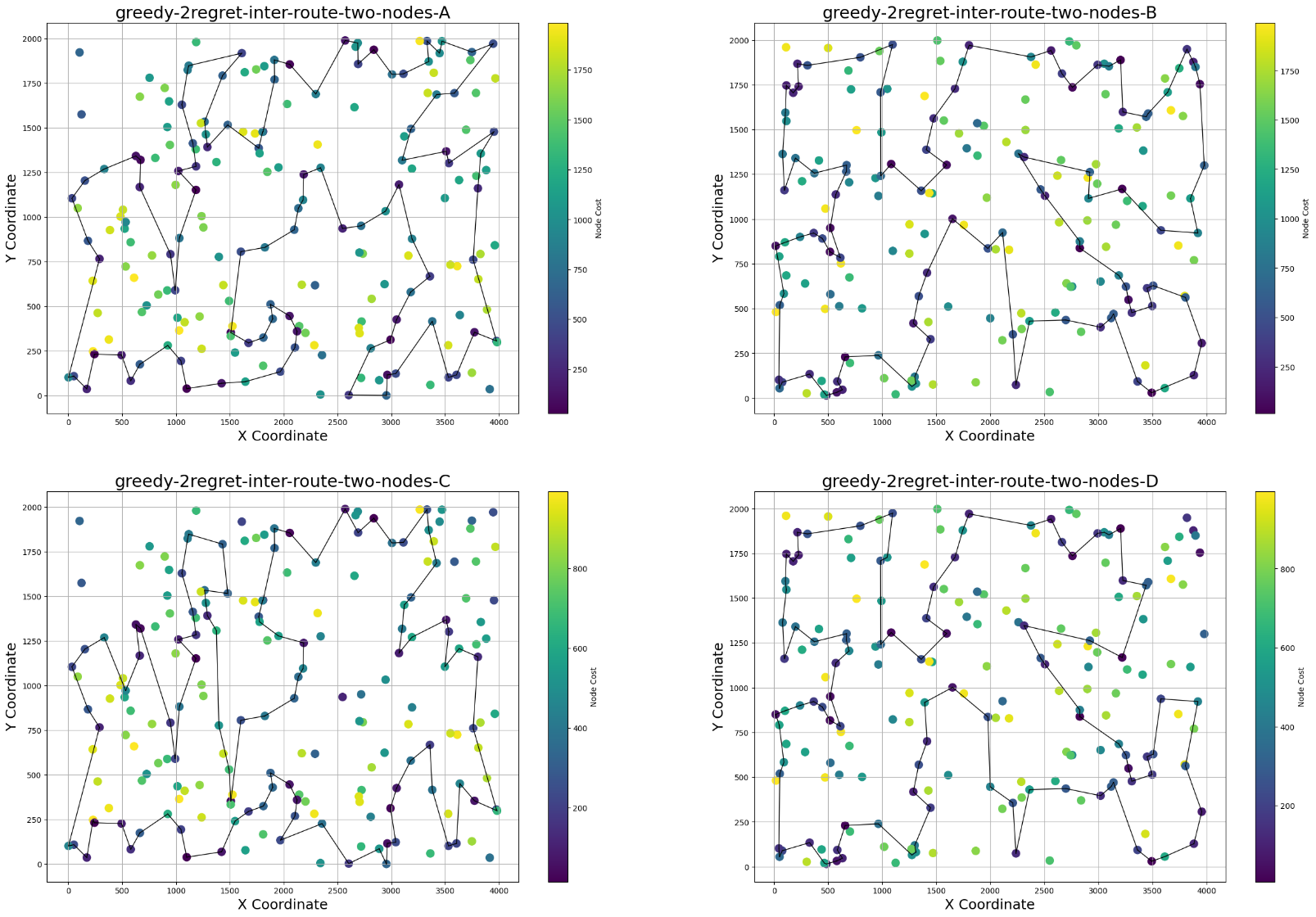






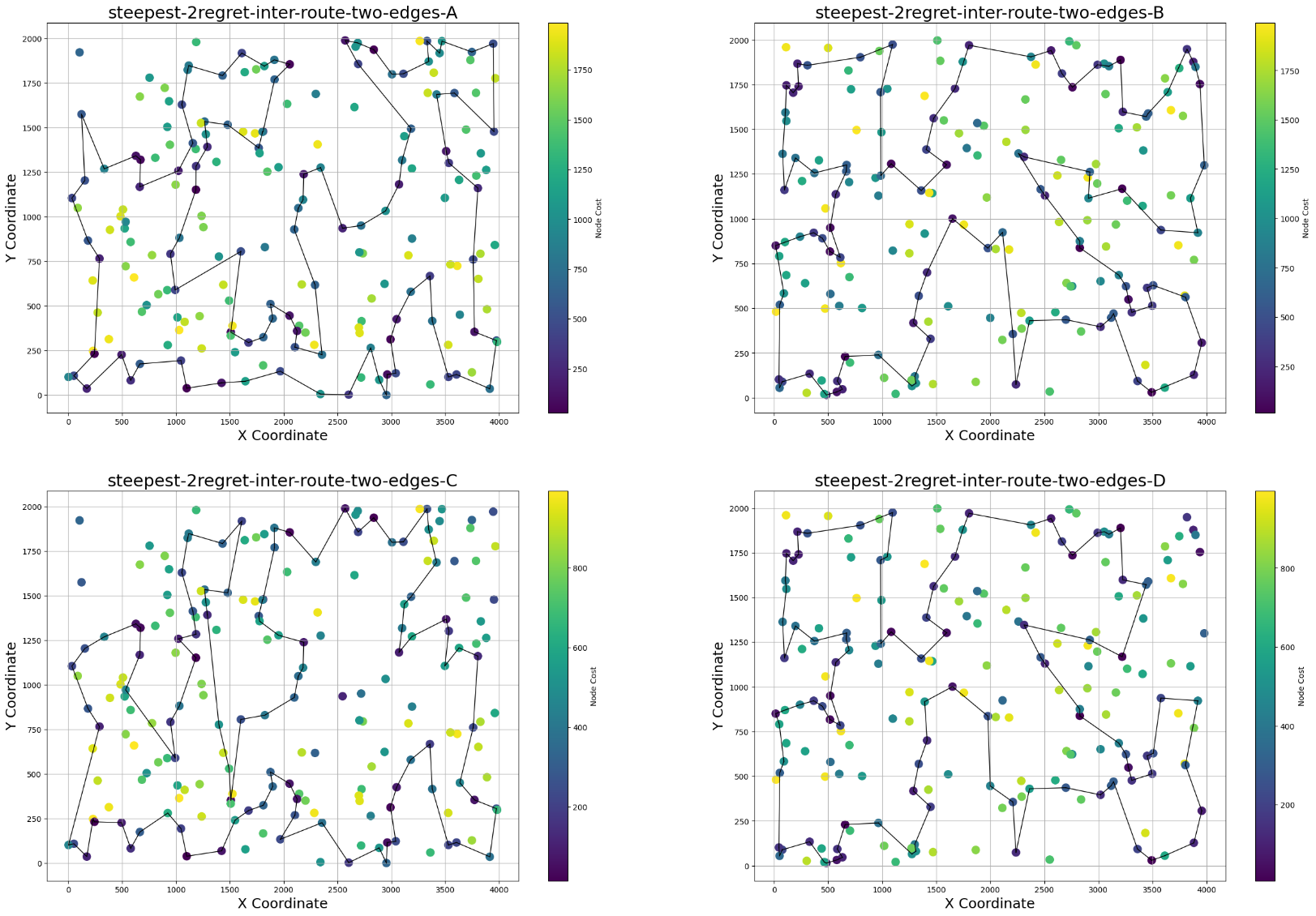
Obraz zawierający tekst, diagram, linia

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.