Raport 1 - Greedy heuristics GITHUB

Eryk Ptaszyński: 151950
Eryk Walter: 151931

The **Traveling Salesman Problem (TSP)** is an optimization problem where the objective is to find the shortest possible route that visits a set of cities exactly once and returns to the starting city. In its traditional form, the TSP assumes that the cost of traveling between any two cities is known and fixed, and the salesman must visit all cities.

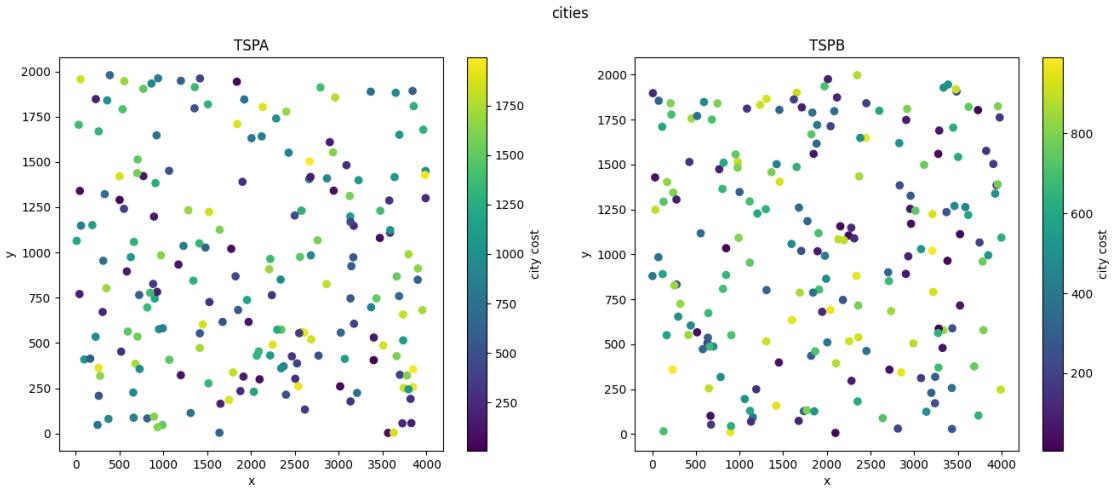
Table of Contents

Modified TSP Problem with Additional Constraints

1. **Additional City Cost**:
   In this modified version, each city has an associated **fixed cost** (besides the cost of travel). This city cost represents an additional expense incurred for visiting the city. Therefore, the total cost of the route is the sum of the travel costs between cities and the individual costs for each visited city. The objective becomes minimizing the total of both travel costs and city costs.

2. **Selection of Only 50% of Cities**:
   Another key modification is that the salesman is not required to visit **all** cities. Instead, the objective is to visit **exactly 50% of the available cities**. This creates a **sub-selection** problem where the salesman must decide which subset of cities to visit while minimizing the total cost (**travel + city cost**).

This visual representation provides an intuitive way to interpret the spatial relationships between cities, their associated fixed costs, and potential travel paths.



Combined TSPA and TSPB results table:

TOC

(all best solutions were checked with the solver)

Instance TSPA results table:

Algorithm	Min	Mean	Max	Time* (s)
random	235292	264415	301307	0.0158
greedy_tail	83182	85107	89433	0.0760
greedy_any_position	71868	73648	76178	0.8030
greedy_cycle	71706	72806	74533	0.5351
greedy_cycle_regret	109282	116771	127136	3.5260
greedy_cycle_weighted_regret	71224	72328	73816	3.4680

Instance TSPB results table:

Algorithm	Min	Mean	Max	Time* (s)
random	189578	212776	238786	0.0059
greedy_tail	52319	54390	59030	0.0216
greedy_any_position	44609	48553	57315	0.5879
greedy_cycle	48814	51477	57486	0.4997
greedy_cycle_regret	67391	73388	80062	3.5279
greedy_cycle_weighted_regret	47308	51154	55829	3.5244

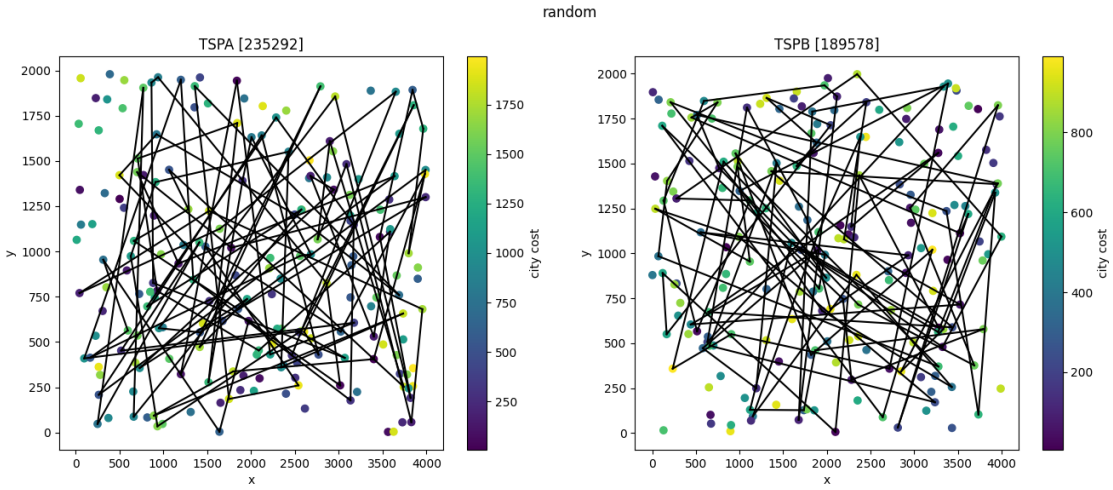
Time* - to solve all 200 instances

Solutions

Random

TOC

```
function randomSolution(problemInstance, availableCities, currentSolution):
   if currentSolution.path.size equals problemInstance.expectedSolutionLen:
      return currentSolution
   city <- random city from available cities
   append city to currentSolution
   currentSolution.cost += distance(currentSolution.last, city)
   remove city from availableCities
   randomSolution(problemInstance, availableCities, currentSolution)
```



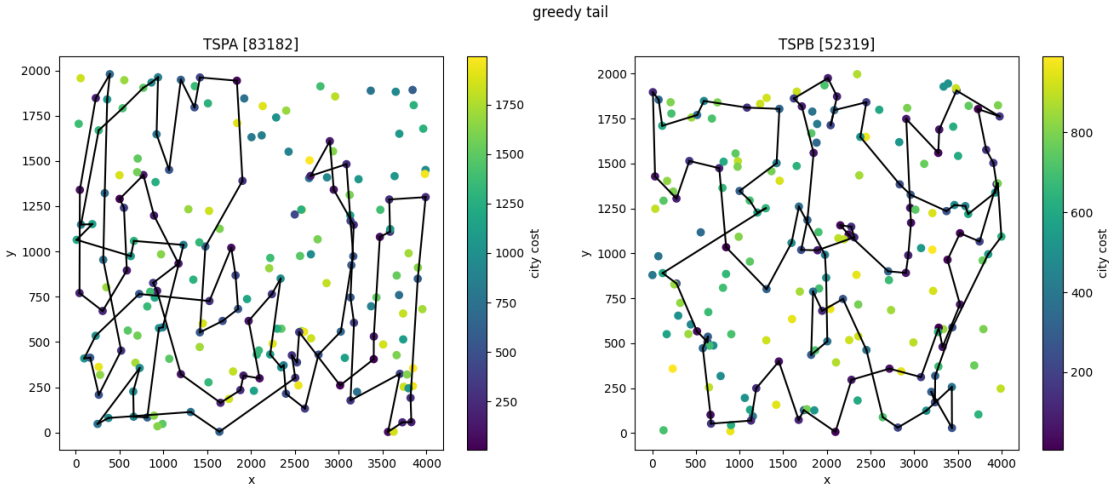
- TSPA random **best solution:** [1 141 62 63 194 55 129 40 144 182 176 45 65 159 23 192 126 198 169 125 101 74 193 100 154 4 41 181 21 22 5 32 123 136 15 96 80 14 16 104 158 151 20 135 145 179 92 120 168 37 86 53 76 0 58 17 139 85 48 184 134 13 42 117 106 68 67 30 152 34 57 46 119 49 138 51 175 107 26 109 9 84 116 191 12 35 64 185 60 187 8 2 121 180 54 50 178 94 3 118]
- TSPB random **best solution:** [140 70 191 63 92 77 120 117 148 95 185 149 139 177 134 167 53 37 183 75 194 14 154 180 187 51 168 13 31 40 30 78 172 190 195 17 44 38 96 45 159 67 155 58 136 16 113 28 182 179 48 64 137 74 152 138 4 198 80 11 192 112 91 126 9 49 68 141 22 79 151 57 19 100 119 7 20 71 115 121 160 90 97 82 60 101 86 1 25 59 130 176 114 153 56 46 105 106 174 124]

Greedy tail

TOC

```
function greedyAppendSolution(problemInstance, availableCities, currentSolution,
initialCity):
   if currentSolution.path.size equals problemInstance.expectedSolutionLen:
      return currentSolution
   city <- initialCity if not null, else take the city such that the
distance(currentSolution.last, city) is minimal
   append city to currentSolution
   currentSolution.cost += distance(currentSolution.last, city)
```

```
    remove city from availableCities
    greedyAppendSolution(problemInstance, availableCities, currentSolution)
```

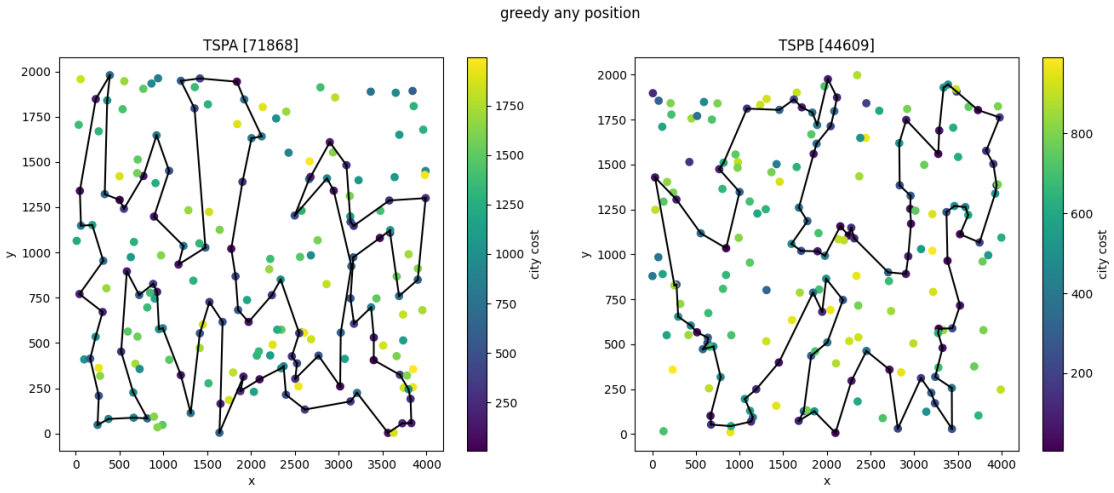


- TSPA greedy append **best solution:** [124 94 63 53 180 154 135 123 65 116 59 115 139 193 41 42 160 34 22 18 108 69 159 181 184 177 54 30 48 43 151 176 80 79 133 162 51 137 183 143 0 117 46 68 93 140 36 163 199 146 195 103 5 96 118 149 131 112 4 84 35 10 190 127 70 101 97 1 152 120 78 145 185 40 165 90 81 113 175 171 16 31 44 92 57 106 49 144 62 14 178 52 55 129 2 75 86 26 100 121]
- TSPB greedy append **best solution:** [16 1 117 31 54 193 190 80 175 5 177 36 61 141 77 153 163 176 113 166 86 185 179 94 47 148 20 60 28 140 183 152 18 62 124 106 143 0 29 109 35 33 138 11 168 169 188 70 3 145 15 155 189 34 55 95 130 99 22 66 154 57 172 194 103 127 89 137 114 165 187 146 81 111 8 104 21 82 144 160 139 182 25 121 90 122 135 63 40 107 100 133 10 147 6 134 51 98 118 74]

Greedy any position

TOC

```
function greedyAtAnyPositionSolution(problemInstance, availableCities,
currentSolution, initialCity):
   if currentSolution.path.size equals problemInstance.expectedSolutionLen:
      currentSolution.cost = calculateCost(currentSolution)
      return currentSolution
   city <- initialCity if not null, else:
      for cityInPath in currentSolution.path:
         find city such that the distance(cityInPath, city) is minimal
      take pair (cityInPath, city) such that the distance(cityInPath, city) is
minimal
   insert city after cityInPath
   remove city from availableCities
   greedyAtAnyPositionSolution(problemInstance, availableCities, currentSolution)
```

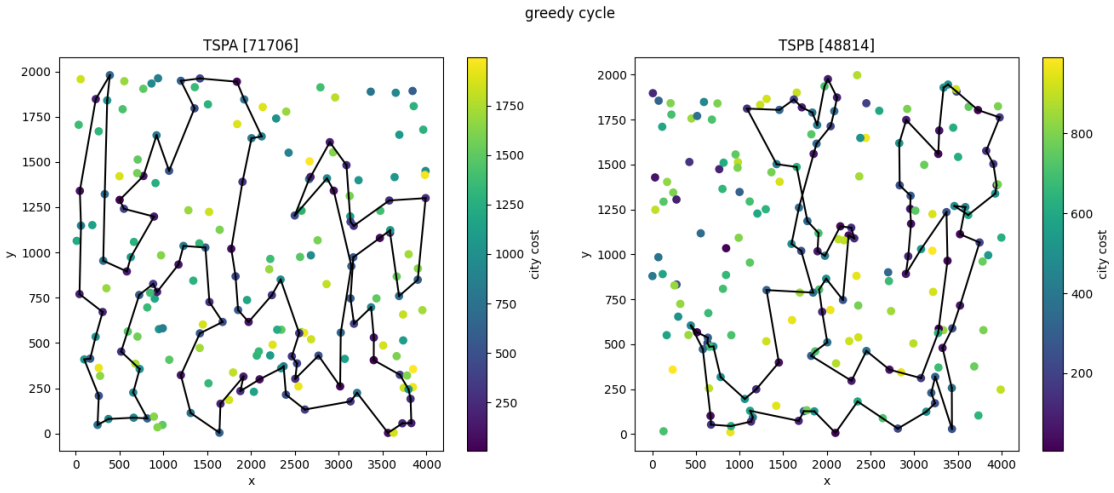


- TSPA greedy at any position **best solution:** [118 51 176 137 183 89 23 186 143 117 93 140 0 80 151 162 133 63 79 94 124 53 97 26 100 152 1 2 120 44 25 78 16 171 175 113 56 31 145 179 92 129 57 185 106 52 55 178 49 102 14 62 9 148 144 40 119 81 196 165 90 101 86 75 180 154 135 70 123 112 4 84 127 59 65 149 131 116 43 42 181 160 54 30 177 10 190 184 34 193 159 22 146 18 108 41 139 46 68 115]
- TSPB greedy at any position **best solution:** [10 133 122 90 51 121 117 198 1 38 27 31 73 193 190 80 175 78 142 45 5 177 36 61 91 141 77 81 153 187 163 89 103 114 127 165 137 176 166 194 86 185 95 130 99 62 124 106 143 0 35 109 29 33 160 144 8 82 21 104 111 138 182 11 139 168 195 145 3 155 15 70 169 132 13 188 6 147 18 55 34 152 183 140 20 28 149 4 148 60 47 94 66 179 113 54 135 63 40 107]

Greedy cycle

TOC

```
function greedyCycleSolution(problemInstance, availableCities, currentSolution,
initialCity):
   if currentSolution.path.size equals problemInstance.expectedSolutionLen:
      return currentSolution
   city <- initialCity if not null, else:
      for (city1, city2) in sliding pairs from currentSolution.path
         for middleCity in availableCities:
            take minimal triplet (city1, middleCity, city2) such that
distance(city1, middleCity) + distance(middleCity, city2) - distance(city1, city2)
is minimal
   insert middleCity after city1
   currentSolution.cost += distance(city1, middleCity) + distance(middleCity,
city2) - distance(city1, city2)
   remove middleCity from availableCities
   greedyCycleSolution(problemInstance, availableCities, currentSolution)
```



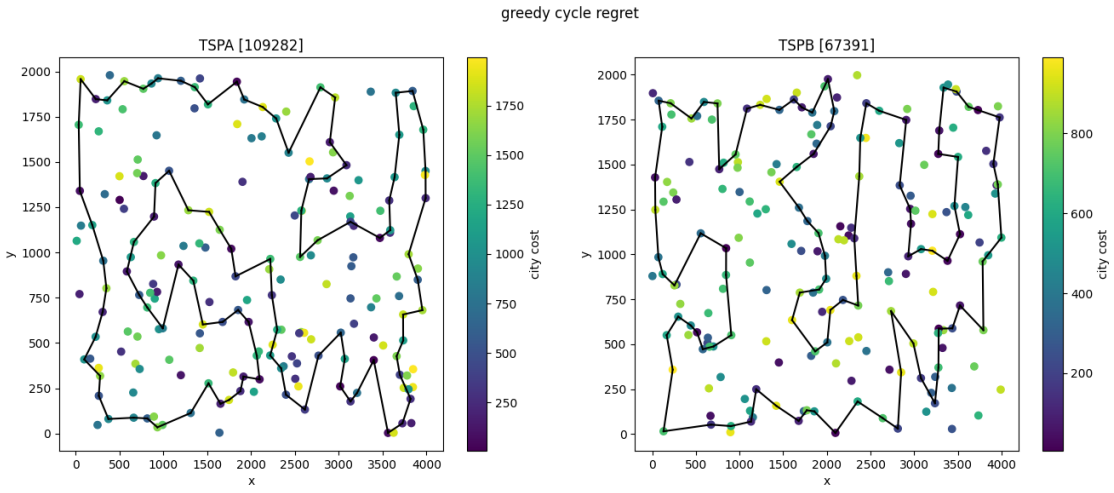
- TSPA greedy cycle **best solution:** [0 46 68 139 193 41 115 5 42 181 159 69 108 18 22 146 34 160 48 54 30 177 10 190 4 112 84 35 184 43 116 65 59 118 51 151 133 162 123 127 70 135 180 154 53 100 26 86 75 44 25 16 171 175 113 56 31 78 145 179 92 57 52 185 119 40 196 81 90 165 106 178 14 144 62 9 148 102 49 55 129 120 2 101 1 97 152 124 94 63 79 80 176 137 23 186 89 183 143 117]
- TSPB greedy cycle **best solution:** [80 162 175 78 142 36 61 91 141 97 187 165 127 89 103 137 114 113 194 166 179 185 99 130 22 66 94 47 148 60 20 28 149 4 140 183 152 170 34 55 18 62 124 106 128 95 86 176 180 163 153 81 77 21 87 82 8 56 144 111 0 35 109 29 160 33 49 11 43 134 147 6 188 169 132 13 161 70 3 15 145 195 168 139 182 138 104 25 177 5 45 136 73 164 31 54 117 198 193 190]

Greedy cycle regret

TOC

```
function getInsertionsAndCosts:
    for edge in current_cycle:
        for city in unvisited_cities:
            Costs[edge][city] = additionalCost(city, edge)
    return Costs

function GreedyCycleRegret:
    while cycle.len < 200:
        costs = getInsertionsAndCosts
        2_best_inserts = for each city find 2 best inserts
        regrets = for each city take (second best value - best value)
        best_insert = insert with max regret
        update cycle with best_insert
```



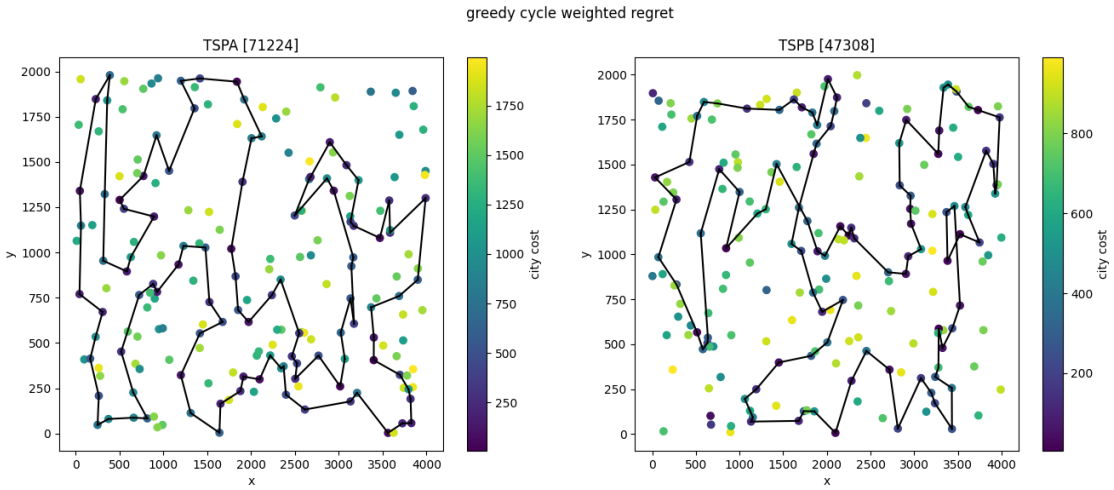
- `TSPA greedy cycle` **best solution:**
  `[178,128,111,37,9,102,14,144,132,73,15,114,83,89,183,153,170,117,93,140,36,67,69,18,134,20,22,195,181,192,160,48,30,104,177,190,4,112,156,127,194,135,6,154,180,53,136,63,79,133,45,72,59,149,77,43,42,96,115,198,46,60,141,66,176,80,12,94,189,121,100,86,75,2,129,82,120,44,25,78,16,171,113,31,38,157,17,98,81,174,90,27,71,164,7,95,39,165,119,185]`
- `TSPB greedy cycle` **best solution:**
  `[176,194,166,48,52,57,154,47,60,20,59,28,4,140,183,9,130,185,86,110,128,124,62,18,34,152,184,155,189,69,35,37,41,111,68,82,87,171,157,56,144,160,49,11,139,2,43,168,145,15,70,132,169,188,6,192,147,71,90,115,10,44,17,107,100,63,92,38,16,197,131,121,112,173,73,193,117,198,156,42,196,108,80,162,175,5,7,36,79,91,141,97,146,187,186,119,129,163,127,26]`

Greedy cycle weighted regret

TOC

```
function weightedRegret:
    return weight * best_value - (1 - weight) * (second_best - best_value)

function GreedyCycleWeightedRegret:
    while cycle.len < 200:
        costs = getInsertionsAndCosts
        2_best_inserts = for each city find 2 best inserts
        weightedRegrets = for each city take weightedRegret
        best_insert = insert with min weighted regret
        update cycle with best_insert
```



- `TSPA greedy cycle` **best solution:**
  `[9,148,102,49,178,106,52,55,57,92,129,2,152,97,1,101,100,53,180,154,135,70,127,123,162,149,65,116,43,42,184,35,84,112,4,190,10,177,54,48,160,34,181,146,22,18,108,69,159,41,193,139,68,140,93,117,143,183,89,23,137,0,46,115,59,118,51,151,133,176,80,79,63,94,189,26,86,75,120,44,25,16,171,175,113,56,31,78,145,179,196,81,90,185,40,165,138,14,144,62]`
- `TSPB greedy cycle` **best solution:**
  `[126,195,168,29,109,35,0,111,81,153,163,176,106,124,62,18,55,34,170,152,140,183,95,86,185,22,99,9,199,28,20,60,148,47,94,66,57,172,179,166,194,113,103,89,127,165,187,146,77,141,91,61,36,175,78,142,45,5,177,21,82,8,104,144,160,33,138,11,139,43,134,74,118,98,51,90,121,131,135,102,63,100,40,107,72,122,133,10,115,147,192,6,188,169,132,70,3,15,145,13]`

Conclusions(#conclusions)

TOC

Performance of Algorithms:

- The `random` algorithm consistently performed the worst in terms of both minimum and mean costs for both TSPA and TSPB instances. This is expected as the `random` approach does not utilize any heuristic to minimize the travel cost.
- The `greedy_tail` algorithm showed significant improvement over the `random` approach, achieving much lower costs. This indicates that even a simple greedy heuristic can substantially improve the solution quality.
- The `greedy_any_position` algorithm further improved the results, especially in the TSPA instance, demonstrating the benefit of considering multiple insertion points rather than just appending to the end.
- The `greedy_cycle` algorithm performed similarly to `greedy_any_position`, but with slightly better results in some cases. This suggests that forming cycles can be an effective strategy for minimizing travel costs.
- The `greedy_cycle_regret` and `greedy_cycle_weighted_regret` algorithms generally provided the best results, particularly in terms of minimum costs. These algorithms take into account the regret of not choosing the second-best option, which helps in making more informed decisions.

Execution Time:

- The `random` algorithm was the fastest, as expected, due to its simplicity.
- The `greedy_tail` and `greedy_any_position` algorithms had moderate execution times, with `greedy_any_position` being slower due to the additional computations for multiple insertion points.
- The `greedy_cycle` algorithm had a similar execution time to `greedy_any_position`, indicating that the cycle formation does not add significant overhead.
- The `greedy_cycle_regret` and `greedy_cycle_weighted_regret` algorithms were the slowest, reflecting the additional complexity of calculating regrets and weighted regrets. However, the improved solution quality may justify the longer execution times.

Overall Effectiveness:

- The results indicate that more sophisticated greedy heuristics, such as `greedy_cycle_regret` and `greedy_cycle_weighted_regret`, provide the best balance between solution quality and computational effort.
- For applications where execution time is critical, simpler heuristics like greedy_tail may be preferred, as they still offer substantial improvements over random solutions with relatively low computational cost.
- For applications where solution quality is paramount, the `greedy_cycle_regret` and `greedy_cycle_weighted_regret` algorithms are recommended despite their higher computational cost.