Evolutionary Computation
lab1. Greedy heuristics - report

Piotr Balewski 156037 and Lidia Wiśniewska 156063

1. Problem Description

The problem involves a set of nodes, each defined by three attributes:

- **x** and **y** coordinates representing the node's position in a 2D plane,
- **cost** associated with the node.

The objective is to select exactly **50% of the nodes** (rounded up if the total number of nodes is odd) and construct a **Hamiltonian cycle** through the selected nodes. The aim is to **minimize the sum** of:

1. The **total length** of the Hamiltonian cycle, and
2. The **total cost** of the selected nodes.

Node-to-node distances are computed as **Euclidean distances**, rounded to the nearest integer. After reading an instance, a **distance matrix** is generated and used exclusively by the optimization algorithms, allowing problem instances to be defined solely by distance data without direct access to coordinates.

2. Pseudocodes of implemented algorithms

Evaluate function was used in all algorithms (where distance is calculated as euclidean distance rounded to the nearest integer):

```
procedure EVALUATE(path, instance, form_cycle)

   total_distance ← 0

   for i from 0 to |path| - 2 do

      total_distance ← total_distance + distance(path[i], path[i+1])

   end for


   if form_cycle = True then

      total_distance ← total_distance + distance(path[|path|-1], path[0])

   end if
```

```
    total_cost ← sum of node costs for all nodes in path

    return total_distance + total_cost

end procedure
```

Random solution:

```
procedure RANDOM_SOLUTION(instance)

    n ← number of nodes in instance

    k ← ceil(n / 2)

    nodes ← list of all node indices

    shuffle(nodes)

    selected ← first k nodes from nodes

    value ← evaluate(selected, instance, form_cycle = True)

    return (selected, value)

end procedure
```

Nearest end:

```
procedure NEAREST_END(instance, start)

    selected ← [start]

    remaining ← set of all nodes except start

    while |selected| < ceil(n / 2) do

        last ← last node in selected

        best_node ← None

        best_value ← ∞
```

```
        for each j in remaining do
            value ← distance(last, j) + cost(j)
            if value < best_value then
                best_value ← value
                best_node ← j
            end if
        end for

        append best_node to selected
        remove best_node from remaining
    end while

    total_value ← evaluate(selected, instance, form_cycle = True)
    return (selected, total_value)
end procedure
```

Nearest any:

```
procedure NEAREST_ANY(instance, start)
    selected ← [start]
    remaining ← set of all nodes except start

    while |selected| < ceil(n / 2) do
        best_node ← None
        best_pos ← None
        best_value ← ∞
```

```
        for each j in remaining do
            for each insertion position p in [0 .. |selected|] do
                trial ← insert j at position p in selected
                value ← evaluate(trial, instance, form_cycle = False)
                if value < best_value then
                    best_value ← value
                    best_node ← j
                    best_pos ← p
                end if
            end for
        end for

        insert best_node at position best_pos in selected
        remove best_node from remaining
    end while

    total_value ← evaluate(selected, instance, form_cycle = True)
    return (selected, total_value)
end procedure
```

Greedy cycle:

```
procedure GREEDY_CYCLE(instance, start)
    remaining ← set of all nodes except start

    // Find best second node
    best_second ← None
```

```
best_value ← ∞
for each j in remaining do
    value ← distance(start, j) + cost(j)
    if value < best_value then
        best_value ← value
        best_second ← j
    end if
end for

selected ← [start, best_second]
remove best_second from remaining

while |selected| < ceil(n / 2) do
    best_node ← None
    best_pos ← None
    best_eval ← ∞

    for each j in remaining do
        for each edge position (pos) in [0 .. |selected| - 1] do
            trial ← insert j between selected[pos] and selected[pos + 1]
            value ← evaluate(trial, instance, form_cycle = True)
            if value < best_eval then
                best_eval ← value
                best_node ← j
                best_pos ← pos + 1
            end if
```

        end for

    end for

    insert best_node at position best_pos in selected

    remove best_node from remaining

  end while

  total_value ← evaluate(selected, instance, form_cycle = True)

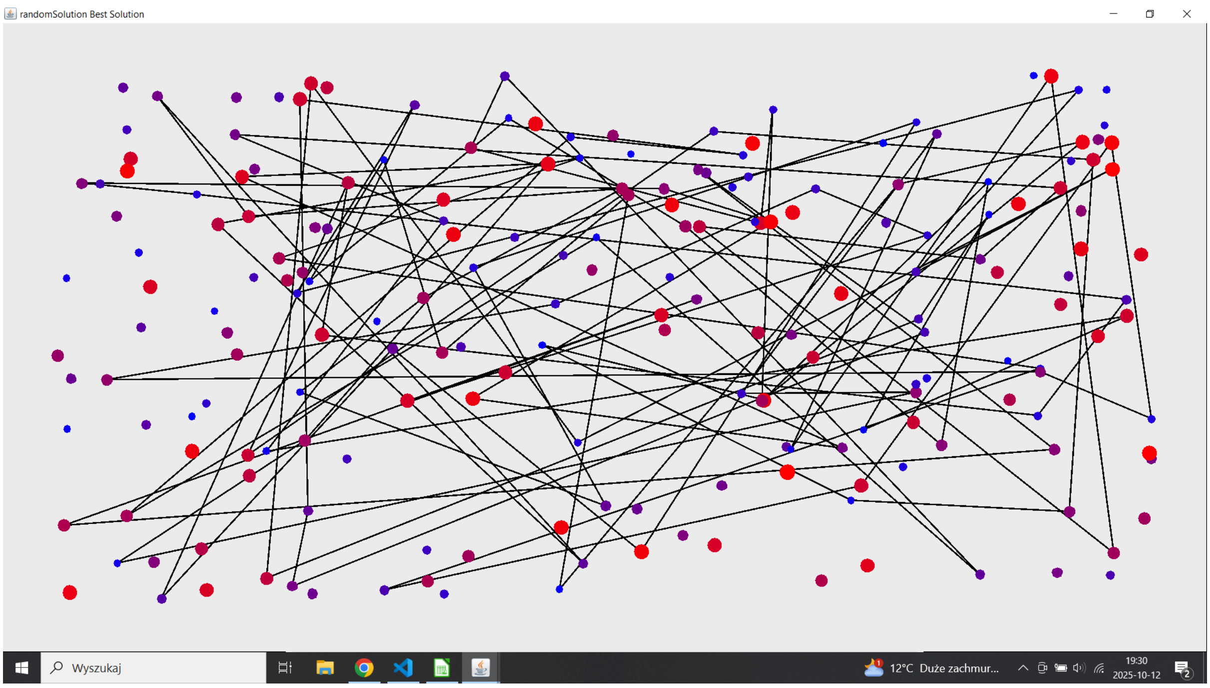
  return (selected, total_value)

end procedure

3. Results for instance A

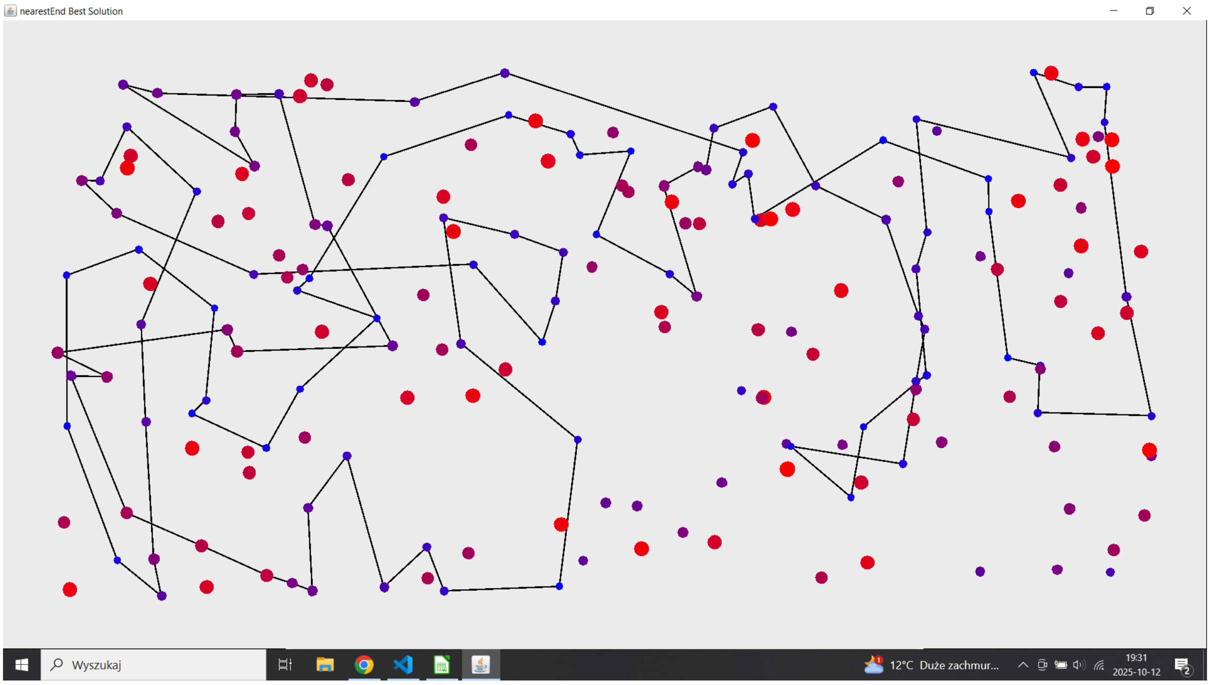
Random solution -> min: 237322 max: 291386 avg: 262994,82

best solution: [154, 101, 126, 68, 140, 52, 100, 39, 20, 124, 60, 92, 2, 80, 195, 119, 90, 50, 155, 117, 40, 77, 180, 28, 89, 190, 166, 136, 72, 115, 23, 151, 44, 83, 118, 108, 24, 197, 165, 81, 184, 30, 121, 125, 141, 102, 11, 173, 198, 18, 3, 148, 25, 62, 88, 58, 19, 179, 38, 84, 162, 139, 187, 49, 145, 55, 36, 156, 137, 188, 57, 171, 116, 127, 191, 86, 85, 95, 144, 176, 33, 78, 138, 189, 135, 65, 123, 109, 182, 183, 82, 13, 37, 75, 74, 194, 70, 21, 63, 199]

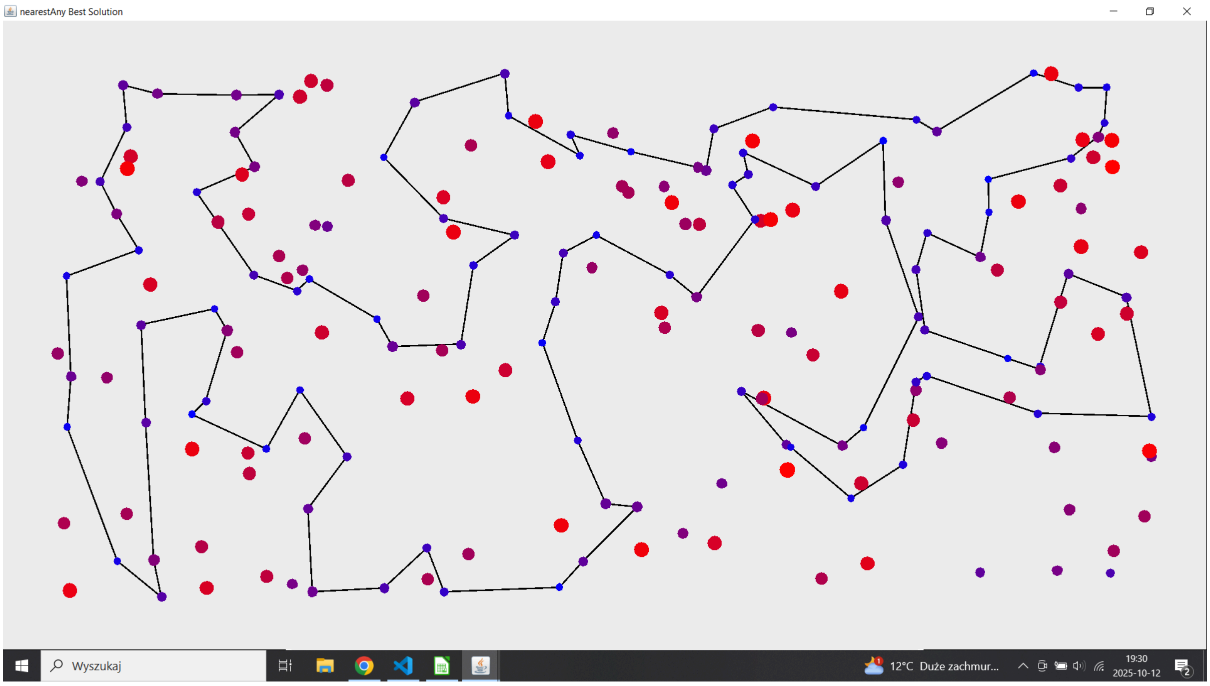


Nearest end -> min: 83182 max: 89433 avg: 85108,51

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]



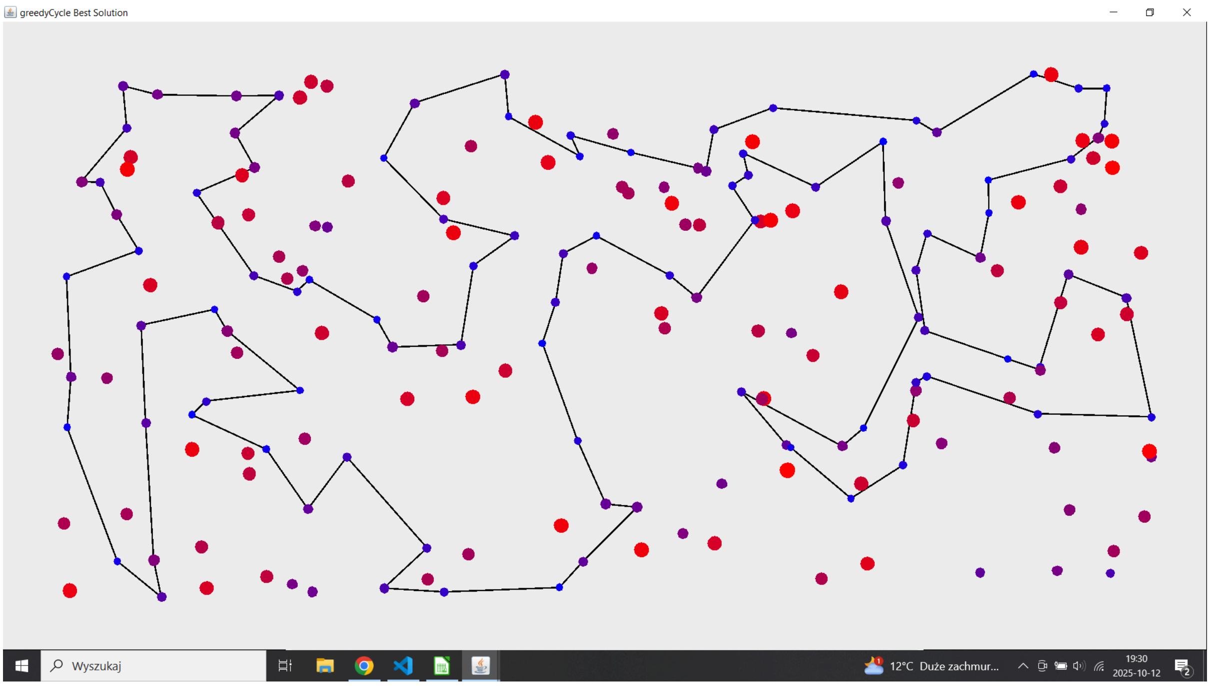
Nearest any -> min: 71179 max: 75450 avg: 73178,55



best solution: [68, 46, 115, 139, 193, 41, 5, 42, 181, 159, 69, 108, 18, 22, 146, 34, 160, 48, 54, 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, 0, 117, 93]

Greedy cycle -> min: 71488 max: 74410 avg: 72646,38

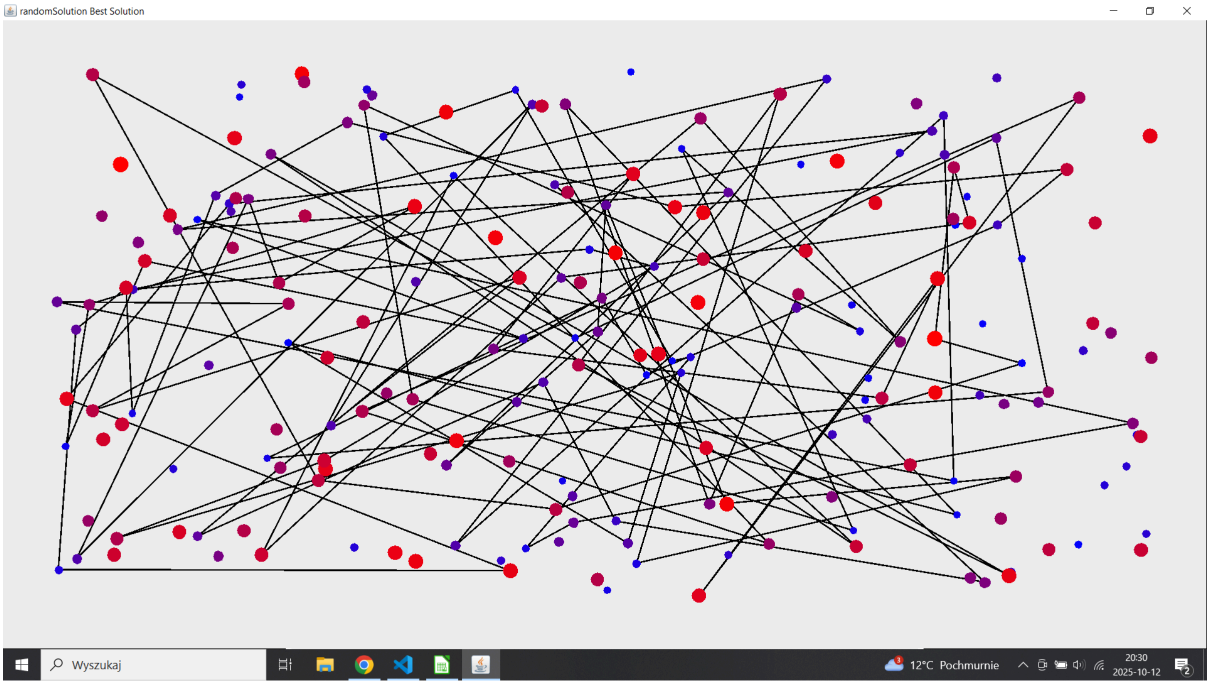
best solution: [0, 117, 143, 183, 89, 186, 23, 137, 176, 80, 79, 63, 94, 124, 152, 97, 1, 101, 2, 120, 129, 55, 49, 102, 148, 9, 62, 144, 14, 178, 106, 165, 90, 81, 196, 40, 119, 185, 52, 57, 92, 179, 145, 78, 31, 56, 113, 175, 171, 16, 25, 44, 75, 86, 26, 100, 53, 154, 180, 135, 70, 127, 123, 162, 133, 151, 51, 118, 59, 65, 116, 43, 184, 35, 84, 112, 4, 190, 10, 177, 30, 54, 48, 160, 34, 146, 22, 18, 108, 69, 159, 181, 42, 5, 115, 41, 193, 139, 68, 46]



4. Results for instance B

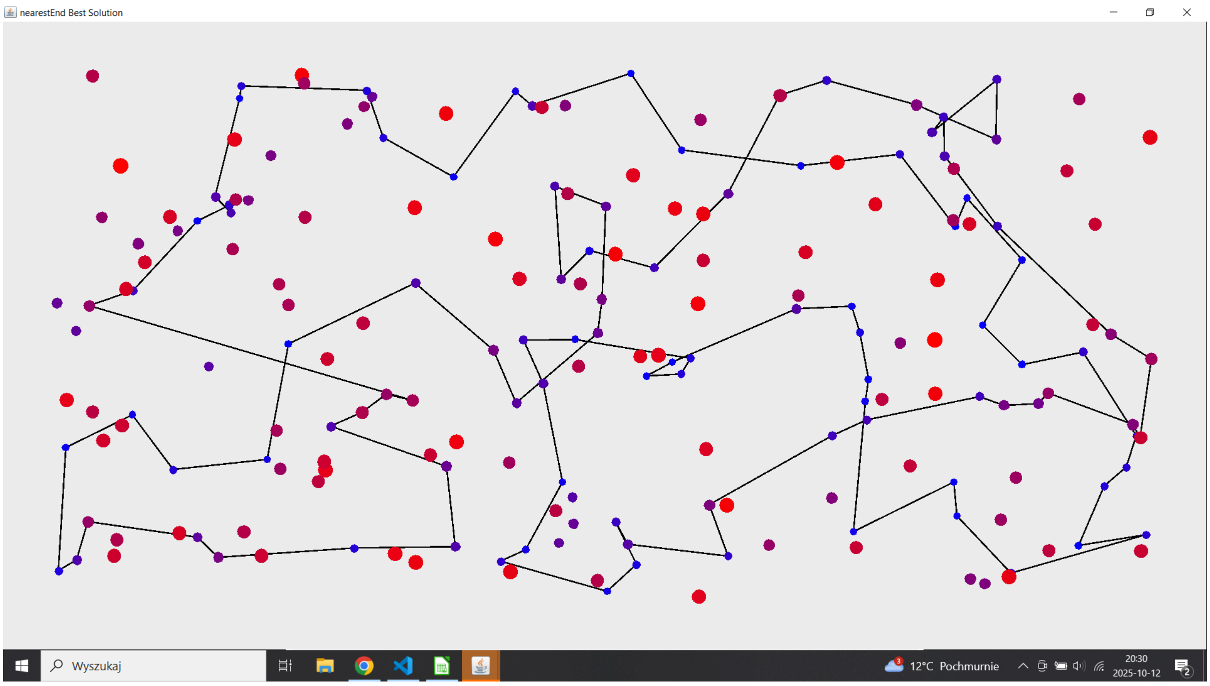
Random solution -> min: 185591 max: 240630 avg: 212627,14

best solution: [15, 121, 140, 91, 189, 159, 124, 142, 74, 152, 77, 128, 97, 133, 11, 145, 149, 21, 58, 182, 83, 64, 84, 75, 98, 184, 5, 36, 0, 35, 169, 13, 66, 117, 138, 24, 63, 16, 123, 19, 73, 107, 61, 51, 157, 102, 112, 27, 183, 127, 6, 187, 198, 81, 134, 160, 82, 167, 9, 3, 146, 49, 101, 136, 53, 177, 115, 104, 88, 26, 110, 185, 126, 71, 108, 33, 114, 22, 90, 144, 29, 194, 76, 50, 45, 193, 135, 197, 38, 40, 65, 92, 31, 89, 1, 8, 111, 120, 72, 109]



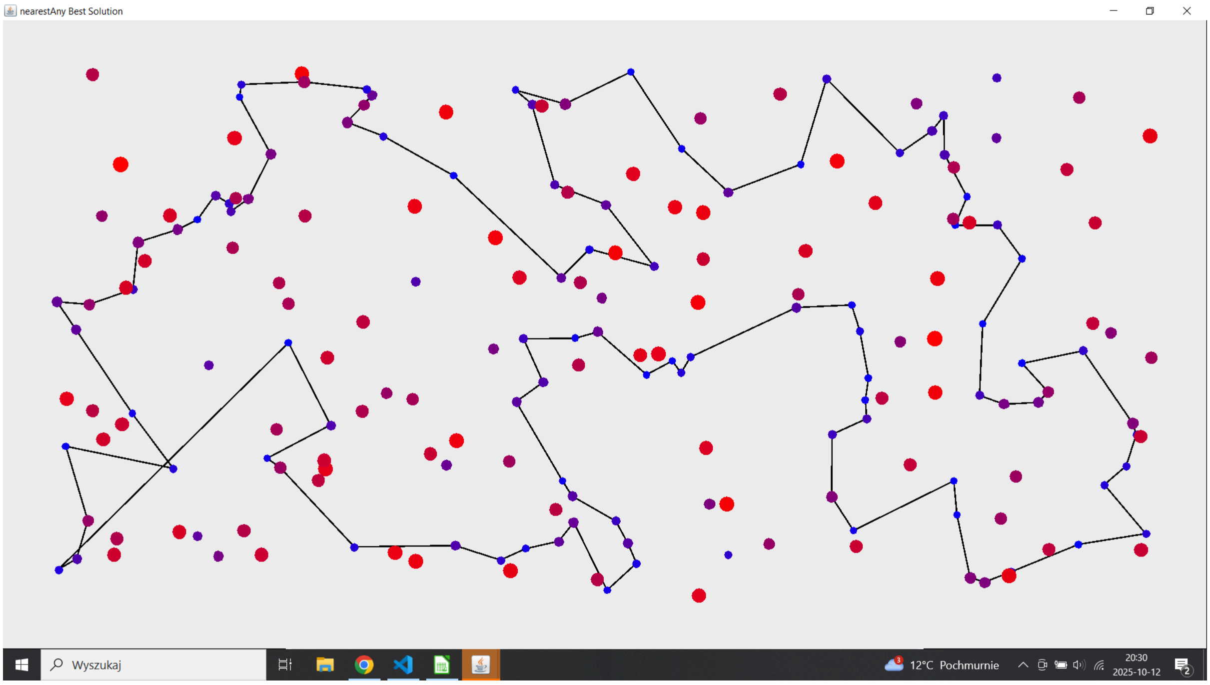
Nearest end -> min: 52319 max: 59030 avg: 54390,43

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]

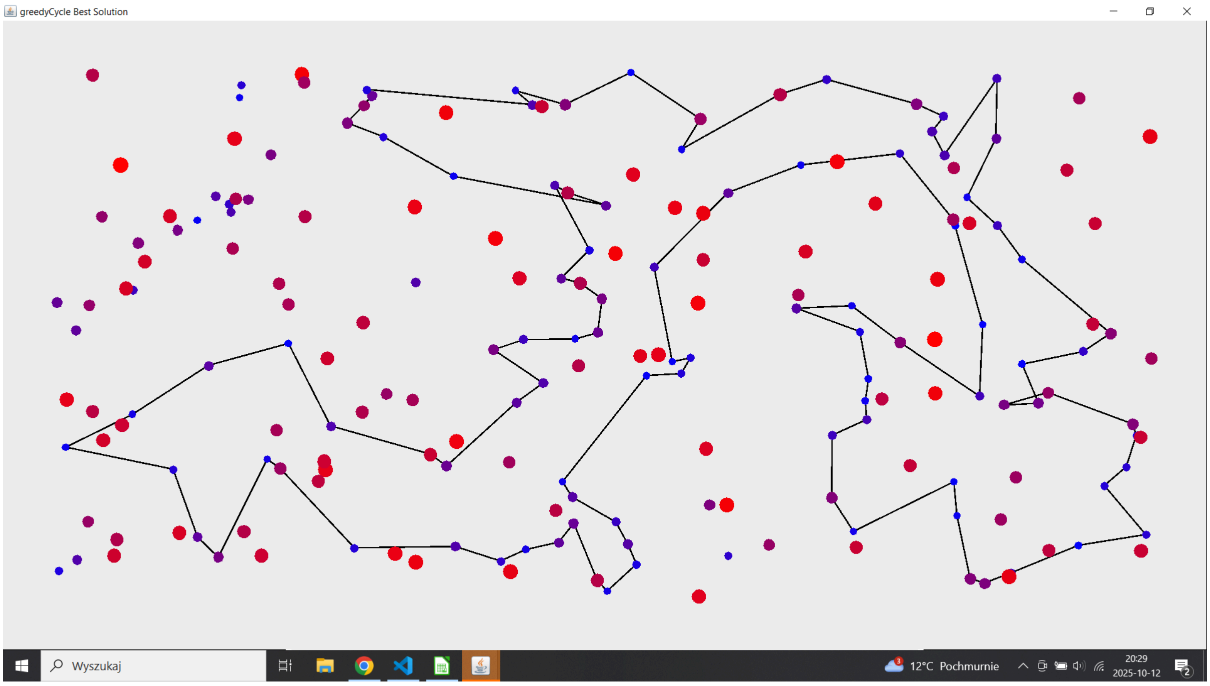


Nearest any -> min: 44417 max: 53438 avg: 45870,26

best solution: [40, 107, 100, 63, 122, 135, 38, 27, 16, 1, 156, 198, 117, 193, 31, 54, 73, 136, 190, 80, 162, 175, 78, 142, 45, 5, 177, 104, 8, 111, 82, 21, 61, 36, 91, 141, 77, 81, 153, 187, 163, 89, 127, 103, 113, 176, 194, 166, 86, 95, 130, 99, 22, 185, 179, 66, 94, 47, 148, 60, 20, 28, 149, 4, 140, 183, 152, 170, 34, 55, 18, 62, 124, 106, 143, 35, 109, 0, 29, 160, 33, 138, 11, 139, 168, 195, 145, 15, 3, 70, 13, 132, 169, 188, 6, 147, 191, 90, 51, 121]



Greedy cycle -> min: 49001 max: 57324 avg: 51400,64



best solution: [85, 51, 121, 131, 135, 63, 122, 133, 10, 90, 191, 147, 6, 188, 169, 132, 13, 161, 70, 3, 15, 145, 195, 168, 29, 109, 35, 0, 111, 81, 153, 163, 180, 176, 86, 95, 128, 106, 143, 124, 62, 18, 55, 34, 170, 152, 183, 140, 4, 149, 28, 20, 60, 148, 47, 94, 66, 22, 130, 99, 185, 179, 172, 166, 194, 113, 114, 137, 103, 89, 127, 165, 187, 146, 77, 97, 141, 91, 36, 61, 175, 78, 142, 45, 5, 177, 82, 87, 21, 8, 104, 56, 144, 160, 33, 138, 182, 11, 139, 134]

All the best solutions were checked with the solution checker.

5. Source code

https://github.com/PBalewski/Evolutionary-Computation/tree/main/lab1

6. Conclusions

The experiments compared four constructive greedy heuristics for solving a modified Traveling Salesman Problem with node costs. All algorithms were evaluated on two benchmark instances (A and B), with performance measured in terms of the total objective value (cycle length + node costs).

The results show clear differences in solution quality between the methods.

- **Random Solution** performed the worst in all cases, producing highly variable and suboptimal results due to its lack of any selection logic.

- **Nearest End** provided significantly better and more consistent results, confirming that greedy extension from one end is an effective baseline approach.

- **Nearest Any** achieved the **best results overall**, consistently yielding the lowest total costs. Allowing node insertion at any position in the current path improves solution flexibility and helps avoid poor early decisions.

- **Greedy Cycle** performed similarly to Nearest Any but slightly worse on average, suggesting that while considering full cycle insertions is beneficial, the additional constraints may sometimes limit optimization potential.

Across both instances, the ranking of methods was consistent:
Nearest Any < Greedy Cycle < Nearest End < Random Solution
(where “<” means “produces lower total cost”).

The visualization of best solutions confirmed that both Nearest Any and Greedy Cycle produce well-structured, compact tours that effectively balance spatial proximity and node cost minimization. In contrast, Random Solution paths appeared irregular and inefficient.

Overall, the study demonstrates that even simple greedy heuristics can generate high-quality approximate solutions for combinatorial optimization problems when properly designed. The **Nearest Any** approach proved to be the most effective heuristic in this experiment.