Greedy heuristics

Students:

- Patrick Molina 157419
- ChihabEddine Zitouni 158763

Problem Description

The task is to select a subset of nodes and form an optimal Hamiltonian cycle minimizing the total cost and travel distance. Each node is defined by three attributes: **x-coordinate**, **y-coordinate**, and **cost**. Exactly **50% of the nodes** must be selected (rounded up if the total number is odd). The goal is to minimize the sum of the **cycle length** and the **total cost** of the selected nodes.

Distances between nodes are computed using the **Euclidean distance**, rounded to the nearest integer. A **distance matrix** is precomputed after reading each instance and used throughout the optimization process, allowing instances to be represented solely by distance values.

Methods

Greedy Heuristics

Four constructive heuristics are implemented and adapted to this problem:

1. **Random Solution** – Generates a feasible cycle by randomly selecting nodes.
2. **Nearest Neighbor (End Insertion)** – Adds the next node giving the best improvement when inserted only at the **end** of the path.
3. **Nearest Neighbor (Flexible Insertion)** – Adds the next node at any position (beginning, end, or inside) that best improves the objective function.
4. **Greedy Cycle** – Builds the cycle by repeatedly inserting nodes that minimize the total increase in distance and cost.

Here, "nearest" is understood as the **best improvement in the objective value**, not just geometric closeness. For each heuristic, **200 solutions** are generated from each starting node, and **200 random solutions** are also produced for comparison.

Pseudocode

1. `randomSolution(Instance instance)`

Purpose: Generates a completely random solution by randomly selecting half the nodes and creating a random tour

How it works:

- Randomly selects n/2 nodes from all available nodes
- Randomly shuffles the order of selected nodes to create a tour
- Calculates total cost (travel distance + node costs)

```
FUNCTION randomSolution(instance):
    n = total number of nodes
    numToSelect = ceil(n / 2)

    shuffled = shuffle all nodes randomly
    selected = first numToSelect nodes from shuffled

    order = IDs of selected nodes
    shuffle order randomly

    totalDistance = 0
    FOR each consecutive pair (i, i+1) in order (wrapping around):
        totalDistance += distance from order[i] to order[i+1]

    totalNodeCost = sum of costs of selected nodes
    totalCost = totalDistance + totalNodeCost

    RETURN Solution(selected, order, totalCost, totalDistance)
END FUNCTION
```

2. `nearestNeighborEndOnly(Instance instance)`

Purpose: Greedy construction heuristic that always adds the next node at the end of the tour, choosing the one that minimizes the objective increase

How it works:

- Starts with a random node
- Iteratively adds nodes at the end position only
- Selects the node that minimally increases the objective (distance increase + node cost)
- Considers the cycle closure (last node back to first)

```
FUNCTION nearestNeighborEndOnly(instance):
    startNode = random node from all nodes
    selected = [startNode]
    order = [startNode.id]
    remaining = all nodes except startNode
    numToSelect = ceil(n / 2)

    WHILE selected.size < numToSelect AND remaining not empty:
        lastNode = last node in selected
        firstNode = first node in selected
        bestCandidate = null
        minIncrease = infinity

        FOR each candidate in remaining:
            distToCandidate = distance[lastNode][candidate]
            distCandidateToFirst = distance[candidate][firstNode]
            distLastToFirst = distance[lastNode][firstNode]

            distanceIncrease = distToCandidate + distCandidateToFirst - distLastToFirst
            objectiveIncrease = distanceIncrease + candidate.cost

            IF objectiveIncrease < minIncrease:
                minIncrease = objectiveIncrease
                bestCandidate = candidate

        IF bestCandidate exists:
            append bestCandidate to selected
            append bestCandidate.id to order
            remove bestCandidate from remaining

    calculate totalDistance and totalCost
    RETURN Solution(selected, order, totalCost, totalDistance)
END FUNCTION
```

3. `nearestNeighborAllPositions(Instance instance)`

Purpose: Enhanced greedy construction that considers inserting the next node at any position in the tour, not just the end

How it works:

- Starts with a random node
- For each candidate node, tries inserting it at every possible position
- Selects the node-position combination that minimally increases the objective
- More sophisticated than `nearestNeighborEndOnly`

```
FUNCTION nearestNeighborAllPositions(instance):
    startNode = random node from all nodes
    selected = [startNode]
    order = [startNode.id]
    remaining = all nodes except startNode
    numToSelect = ceil(n / 2)

    WHILE selected.size < numToSelect AND remaining not empty:
        bestCandidate = null
        bestPosition = -1
        minIncrease = infinity

        FOR each candidate in remaining:
            FOR each position pos from 0 to order.size:
                prevNodeId = order[(pos - 1 + order.size) % order.size]
                nextNodeId = order[pos % order.size]

                distPrevToNext = distance[prevNodeId][nextNodeId]
                distPrevToCandidate = distance[prevNodeId][candidate]
                distCandidateToNext = distance[candidate][nextNodeId]

                distanceIncrease = distPrevToCandidate + distCandidateToNext - distPrevToNext
                objectiveIncrease = distanceIncrease + candidate.cost
```

```
                IF objectiveIncrease < minIncrease:
                    minIncrease = objectiveIncrease
                    bestCandidate = candidate
                    bestPosition = pos

        IF bestCandidate exists:
            insert bestCandidate at bestPosition in selected
            insert bestCandidate.id at bestPosition in order
            remove bestCandidate from remaining

    calculate totalDistance and totalCost
    RETURN Solution(selected, order, totalCost, totalDistance)
END FUNCTION
```

4. `greedyCycle(Instance instance, Node startNode)`

Purpose: Greedy cycle construction with a specified starting node, similar to `nearestNeighborAllPositions` but deterministic start

How it works:

- Starts with a given node (not random)
- Iteratively inserts nodes at positions that minimize objective increase
- Evaluates all positions between consecutive nodes in the current tour

```
FUNCTION greedyCycle(instance, startNode):
    selected = [startNode]
    order = [startNode.id]
    remaining = all nodes except startNode
    numToSelect = ceil(n / 2)

    WHILE selected.size < numToSelect AND remaining not empty:
        bestCandidate = null
        bestPosition = -1
        minIncrease = infinity

        FOR each candidate in remaining:
            FOR each position pos from 0 to order.size - 1:
                prevNodeId = order[pos]
                nextNodeId = order[(pos + 1) % order.size]

                distPrevToNext = distance[prevNodeId][nextNodeId]
                distPrevToCandidate = distance[prevNodeId][candidate]
                distCandidateToNext = distance[candidate][nextNodeId]

                distanceIncrease = distPrevToCandidate + distCandidateToNext - distPrevToNext
                objectiveIncrease = distanceIncrease + candidate.cost

                IF objectiveIncrease < minIncrease:
                    minIncrease = objectiveIncrease
                    bestCandidate = candidate
                    bestPosition = pos + 1

        IF bestCandidate exists:
            insert bestCandidate at bestPosition in selected
            insert bestCandidate.id at bestPosition in order
            remove bestCandidate from remaining

    calculate totalDistance and totalCost
    RETURN Solution(selected, order, totalCost, totalDistance)
END FUNCTION
```

5. `generateSolutions(Instance instance)`

Purpose: Generates a large set of diverse solutions using all available construction methods

How it works:

- Creates 200 random solutions
- For each node in the instance, generates one solution from each greedy method
- Returns a comprehensive collection of solutions for comparison or further optimization

```
FUNCTION generateSolutions(instance):
    solutions = empty list
```

```
    // Generate 200 random solutions
    FOR i from 0 to 199:
        solutions.add(randomSolution(instance))

    // Generate greedy solutions for each starting node
    FOR each startNode in instance.nodes:
        solutions.add(nearestNeighborEndOnly(instance))
        solutions.add(nearestNeighborAllPositions(instance))
        solutions.add(greedyCycle(instance, startNode))

    RETURN solutions
END FUNCTION
```

Results

In [ ]:
```python
import pandas as pd
import numpy as np
import os
import matplotlib.pyplot as plt


tspa_df = pd.read_csv("../../raw_data/TSPA.csv",  header=None)
tspb_df = pd.read_csv("../../raw_data/TSPB.csv",  header=None)
```

Instance A

In [4]:
```python
experiment_summary_a = pd.read_csv("../Results/TSPA/experiment_summary.csv")
```

In [5]:
```python
experiment_summary_a
```

Out[5]:

	Instance	Method	MinCost	MaxCost	AvgCost	NumSolutions	BestSolutionID
0	TSPA	RandomSolution	235000	300212	265387.29	200	128
1	TSPA	NearestNeighborEndOnly	90132	120393	103893.21	200	16
2	TSPA	NearestNeighborAllPositions	71488	74410	72528.80	200	60
3	TSPA	GreedyCycle	71488	74197	72639.63	200	18

Function to load each solution's method

In [6]:
```python
def load_solution(instance_name,method, solution_id):
    file_path = f"../Results/{instance_name}/{instance_name}_{method}_solutions.csv"
    if os.path.exists(file_path):
        df = pd.read_csv(file_path)
        return df[df['SolutionID'] == solution_id]
    else:
        print(f"File {file_path} does not exist.")
        return None
```

In [7]:
```python
solutions_A = pd.DataFrame()
for method in experiment_summary_a['Method'].unique():
    method_data = experiment_summary_a[experiment_summary_a['Method'] == method]
    best_solution_id = method_data['BestSolutionID'].iloc[0]
    print(f"{method}: Best Solution ID = {best_solution_id}")
    solution = load_solution("TSPA", method, best_solution_id)
    solutions_A = pd.concat([solutions_A, solution], ignore_index=True)

solutions_A
```

```
RandomSolution: Best Solution ID = 128
NearestNeighborEndOnly: Best Solution ID = 16
NearestNeighborAllPositions: Best Solution ID = 60
GreedyCycle: Best Solution ID = 18
```

Out[7]:

	SolutionID	TotalCost	NumNodes	TotalDistance	ObjectiveFunction	Cycle
0	128	235000	100	149144	384144	60-95-167-97-58-2-127-195-122-80-59-42-197-175...
1	16	90132	100	33374	123506	132-144-49-14-62-9-148-106-178-185-165-40-90-2...
2	60	71488	100	23578	95066	46-68-139-193-41-115-5-42-181-159-69-108-18-22...
3	18	71488	100	23578	95066	143-183-89-186-23-137-176-80-79-63-94-124-152-...

Parse the coordinates from tspa_df

In [8]:
```python
coords = tspa_df.iloc[:, 0].str.split(';', expand=True)
x_coords = coords[0].astype(int)
```

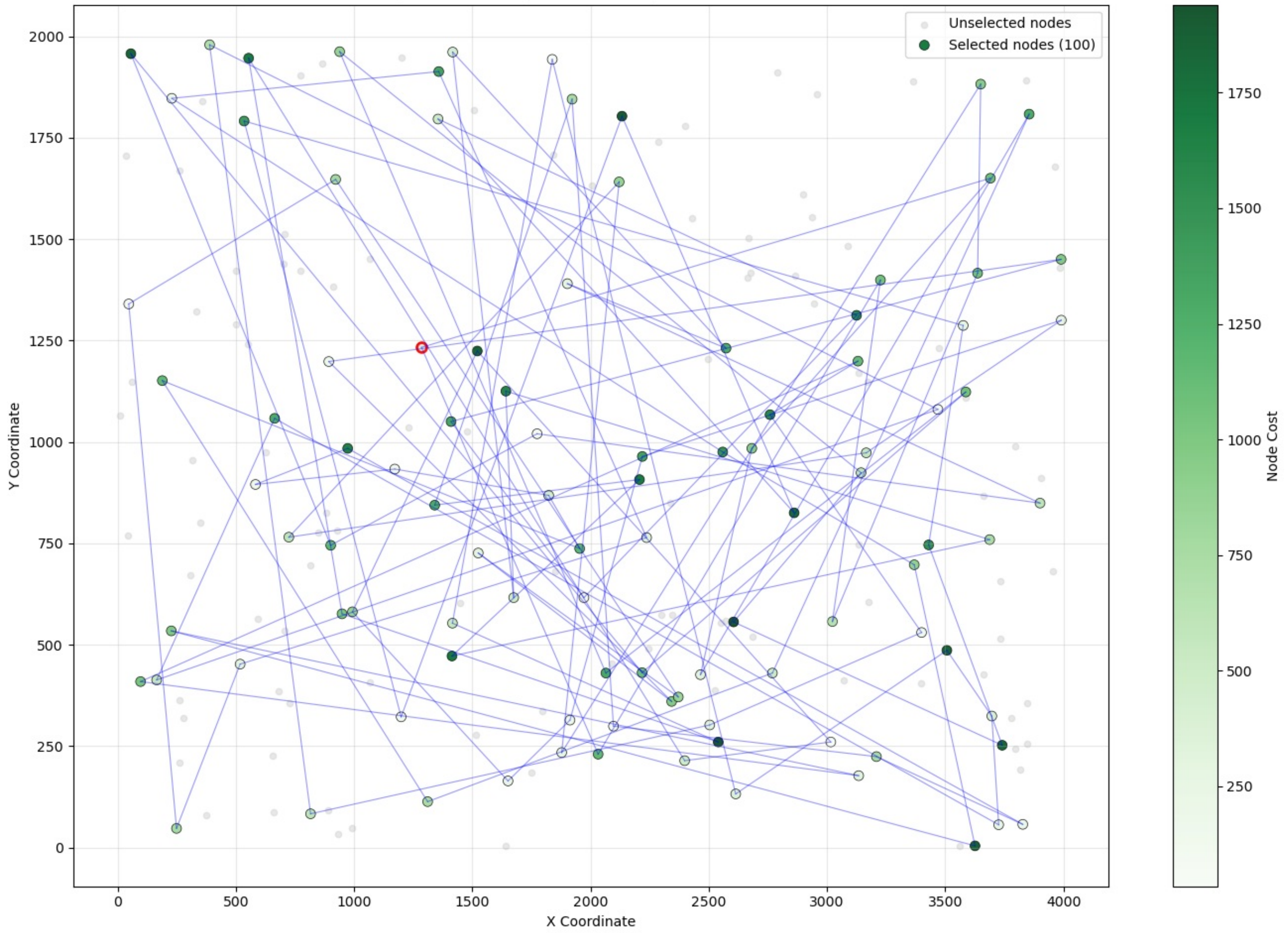
```
y_coords = coords[1].astype(int)
costs = coords[2].astype(int)
```

Graphs

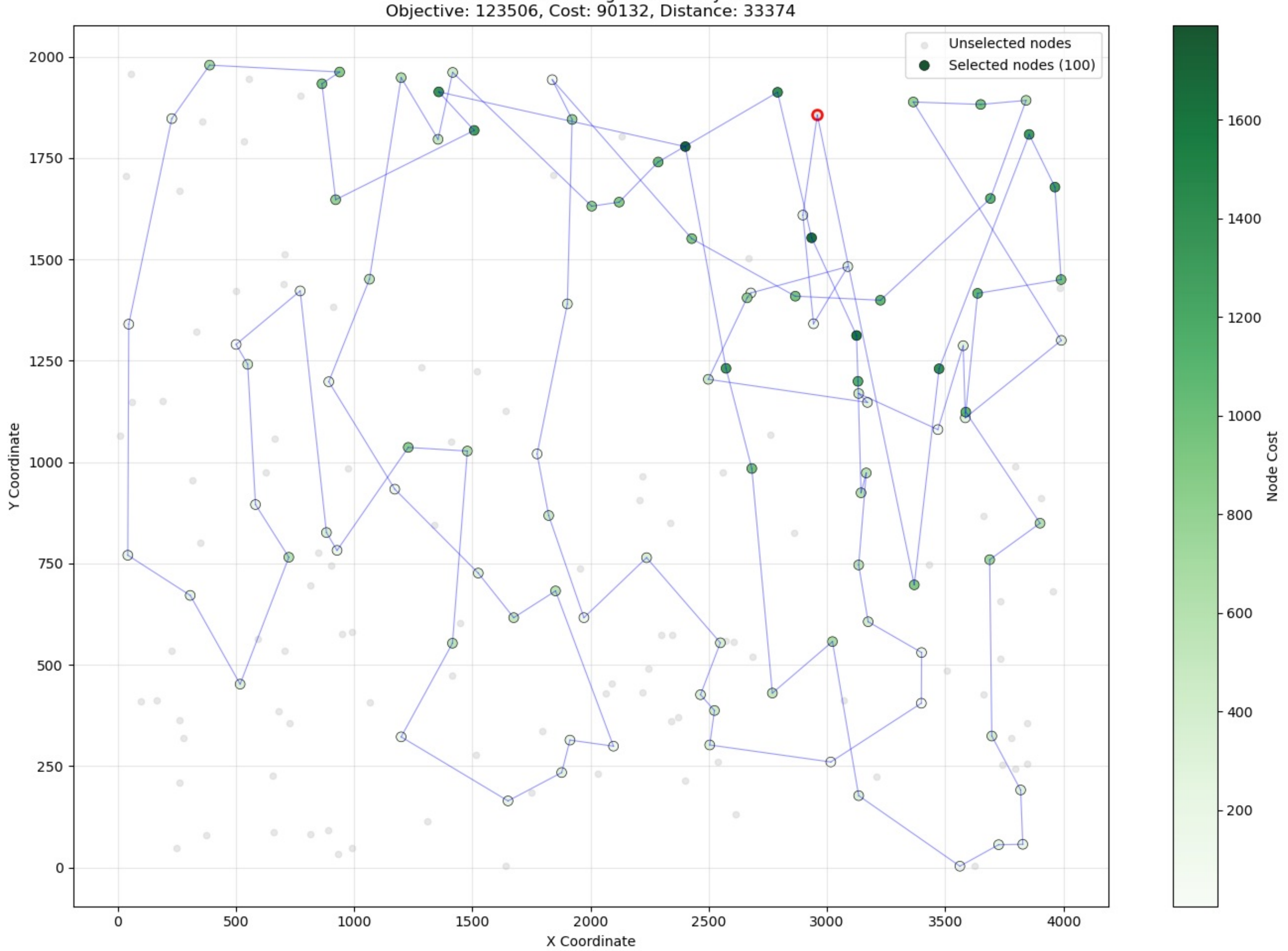
In [ ]:
```
def plot_solution(instance, method, solution_data):
    """
    Plot a TSP solution showing the cycle and nodes.

    Parameters:
    instance: str - 'TSPA' or 'TSPB'
    method: str - Method name or 'Checker'
    solution_data: dict or pandas.Series - Solution data containing cycle information
    """
    if instance == 'TSPA':
        df = tspa_df
    else:
        df = tspb_df

    coords = df.iloc[:, 0].str.split(';', expand=True)
    x_coords = coords[0].astype(int)
    y_coords = coords[1].astype(int)
    costs = coords[2].astype(int)

    if isinstance(solution_data, dict):
        cycle_str = solution_data['Cycle']
        objective = solution_data['ObjectiveFunction']
        total_cost = solution_data['TotalCost']
        total_distance = solution_data['TotalDistance']
    else:
        cycle_str = solution_data['Cycle']
        objective = solution_data['ObjectiveFunction']
        total_cost = solution_data['TotalCost']
        total_distance = solution_data['TotalDistance']

    cycle_nodes = [int(x) for x in cycle_str.split('-')]

    highlight_x = x_coords[cycle_nodes]
    highlight_y = y_coords[cycle_nodes]

    plt.figure(figsize=(14, 10))

    unselected_mask = ~x_coords.index.isin(cycle_nodes)
    plt.scatter(x_coords[unselected_mask], y_coords[unselected_mask],
               alpha=0.5, s=20, color='lightgray', label='Unselected nodes')

    selected_costs = costs[cycle_nodes]
    scatter = plt.scatter(highlight_x, highlight_y,
                         c=selected_costs, cmap='Greens',
                         alpha=0.9, s=50, label='Selected nodes (100)',
                         edgecolors='black', linewidths=0.5)

    first_node_idx = cycle_nodes[0]
    plt.scatter(x_coords[first_node_idx], y_coords[first_node_idx],
               c=costs[first_node_idx], cmap='Greens',
               alpha=0.9, s=50,
               edgecolors='red', linewidths=2)

    for j in range(len(cycle_nodes)):
        current_node = cycle_nodes[j]
        next_node = cycle_nodes[(j + 1) % len(cycle_nodes)]

        plt.plot([x_coords[current_node], x_coords[next_node]],
                [y_coords[current_node], y_coords[next_node]],
                'b-', alpha=0.4, linewidth=0.8)

    plt.colorbar(scatter, label='Node Cost')

    plt.xlabel('X Coordinate')
    plt.ylabel('Y Coordinate')
    plt.title(f'{instance} Dataset - {method} Solution\n'
              f'Objective: {objective}, '
              f'Cost: {total_cost}, '
              f'Distance: {total_distance}')
    plt.legend()
    plt.grid(True, alpha=0.3)
    plt.tight_layout()
    plt.show()
```

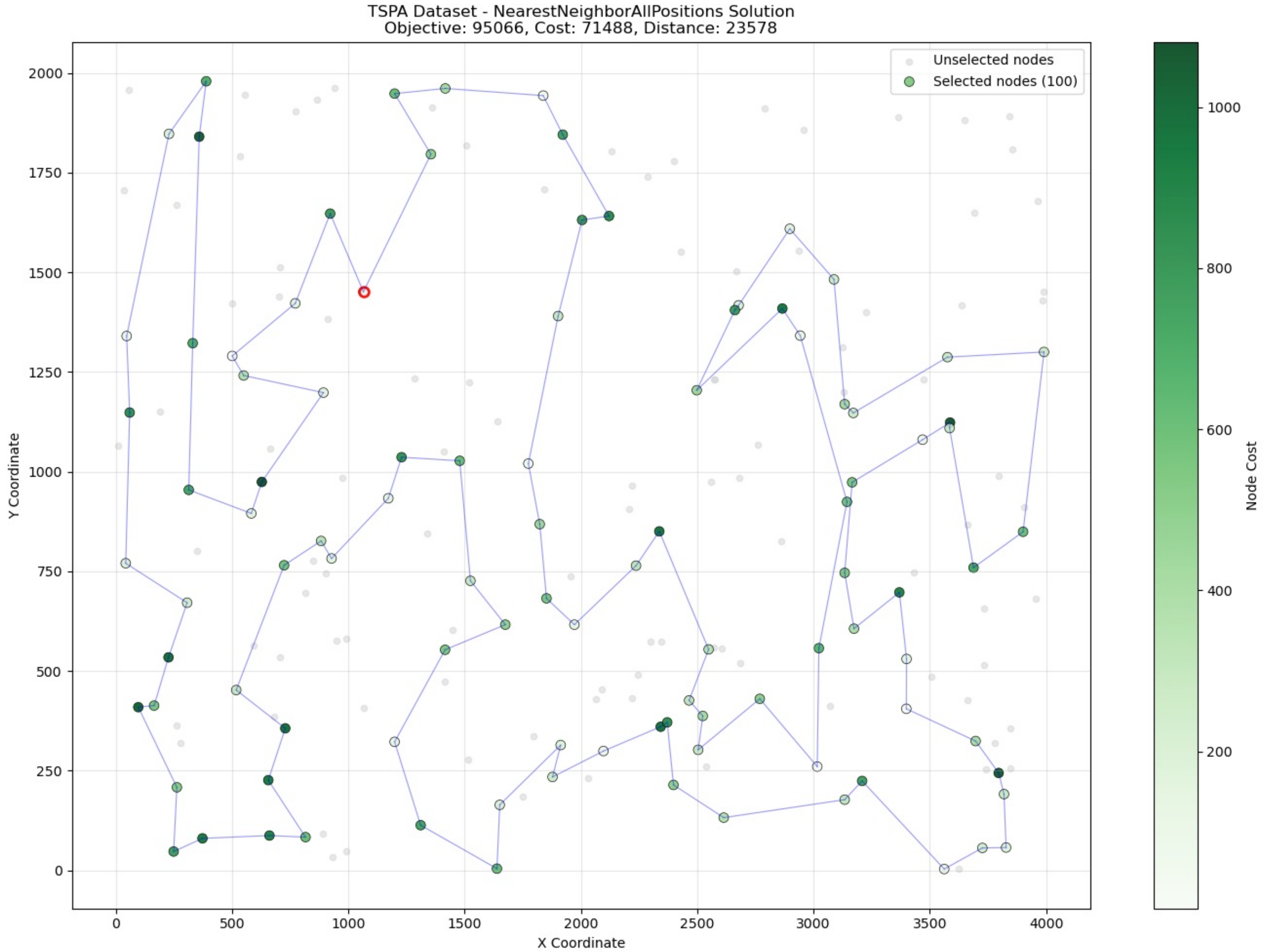
In [10]:
```
methods = ['RandomSolution', 'NearestNeighborEndOnly', 'NearestNeighborAllPositions', 'GreedyCycle']
for i, method in enumerate(methods):
    best_solution = solutions_A[solutions_A.index == i].iloc[0]
    plot_solution('TSPA', method, best_solution)
```

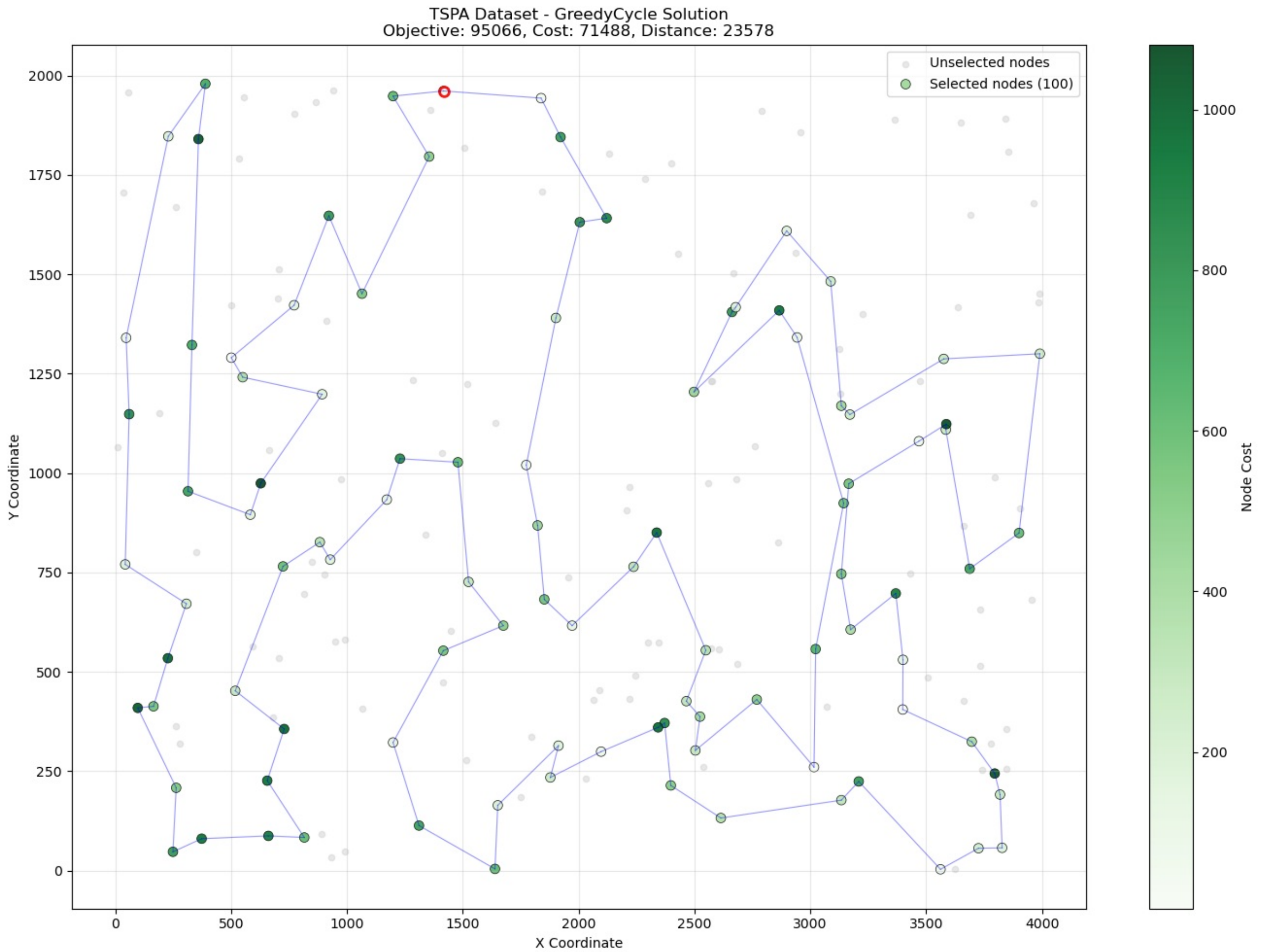
TSPA Dataset - RandomSolution Solution
Objective: 384144, Cost: 235000, Distance: 149144



TSPA Dataset - NearestNeighborEndOnly Solution
Objective: 123506, Cost: 90132, Distance: 33374




TSPA Dataset - NearestNeighborAllPositions Solution
Objective: 95066, Cost: 71488, Distance: 23578
Unselected nodes
Selected nodes (100)
X Coordinate
Y Coordinate
Node Cost


TSPA Dataset - GreedyCycle Solution
Objective: 95066, Cost: 71488, Distance: 23578
Unselected nodes
Selected nodes (100)
X Coordinate
Y Coordinate
Node Cost

Instance B

Load summary solution of TSPB

In [11]:
```
experiment_summary_b = pd.read_csv("../Results/TSPB/experiment_summary.csv")
```

In [12]:
```
experiment_summary_b
```

Out[12]:

	Instance	Method	MinCost	MaxCost	AvgCost	NumSolutions	BestSolutionID
0	TSPB	RandomSolution	188700	239698	213188.32	200	17
1	TSPB	NearestNeighborEndOnly	62606	77453	69681.57	200	22
2	TSPB	NearestNeighborAllPositions	49001	57078	51337.26	200	67
3	TSPB	GreedyCycle	49001	57324	51498.08	200	77

In [13]:
```
solutions_B = pd.DataFrame()
for method in experiment_summary_a['Method'].unique():
    method_data = experiment_summary_a[experiment_summary_a['Method'] == method]
    best_solution_id = method_data['BestSolutionID'].iloc[0]
    print(f"{method}: Best Solution ID = {best_solution_id}")
    solution = load_solution("TSPB", method, best_solution_id)
    solutions_B = pd.concat([solutions_B, solution], ignore_index=True)

solutions_B
```

```
RandomSolution: Best Solution ID = 128
NearestNeighborEndOnly: Best Solution ID = 16
NearestNeighborAllPositions: Best Solution ID = 60
GreedyCycle: Best Solution ID = 18
```

Out[13]:

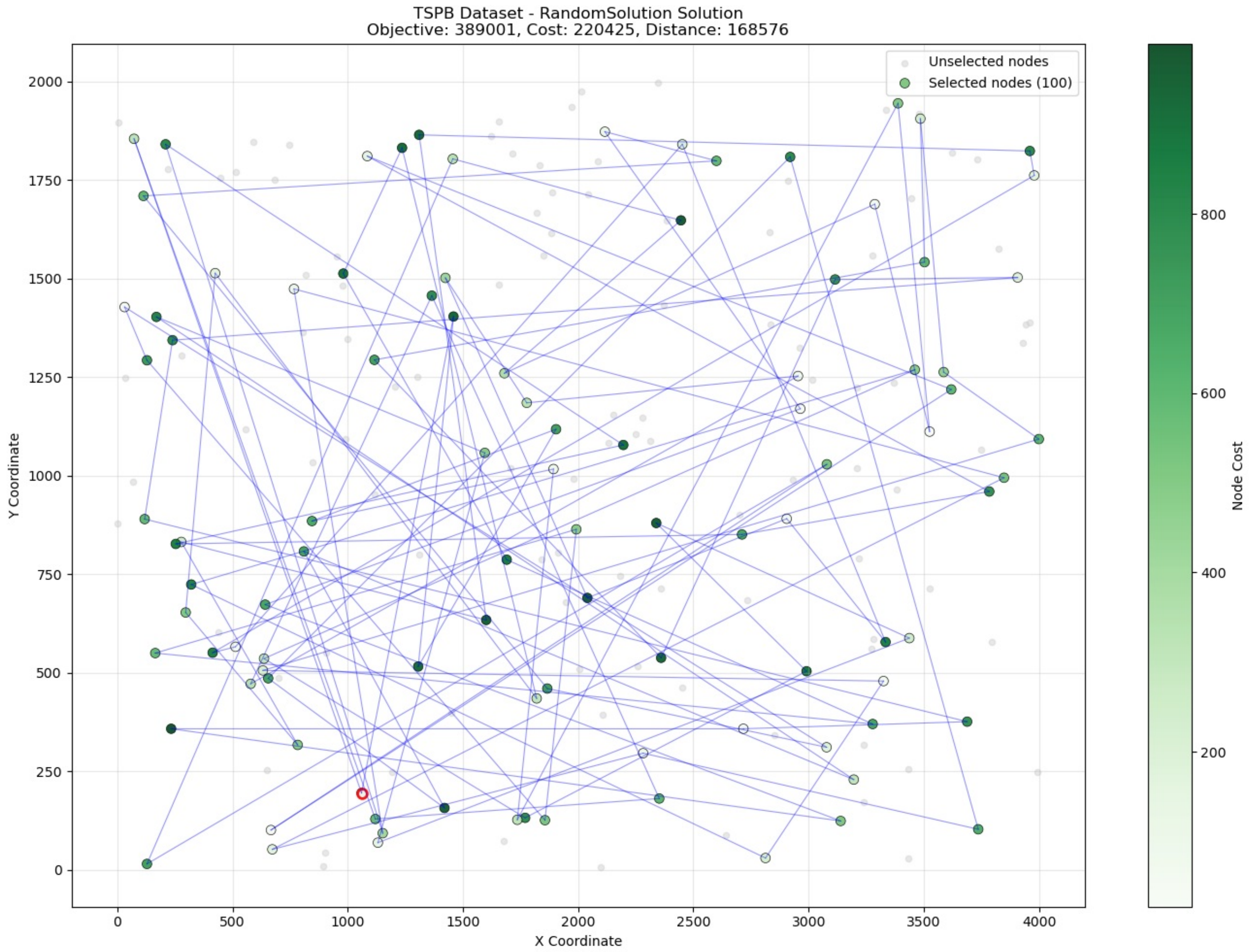
	SolutionID	TotalCost	NumNodes	TotalDistance	ObjectiveFunction	Cycle
0	128	220425	100	168576	389001	45-107-142-97-85-193-18-11-134-68-106-88-155-1...
1	16	66268	100	27424	93692	159-143-106-124-62-18-55-34-35-0-109-29-33-111...
2	60	51591	100	19535	71126	166-52-172-179-185-99-130-22-66-94-154-47-148-...
3	18	49891	100	20039	69930	126-195-168-29-109-35-0-111-81-153-163-180-176...

Parsing coordinates to plot

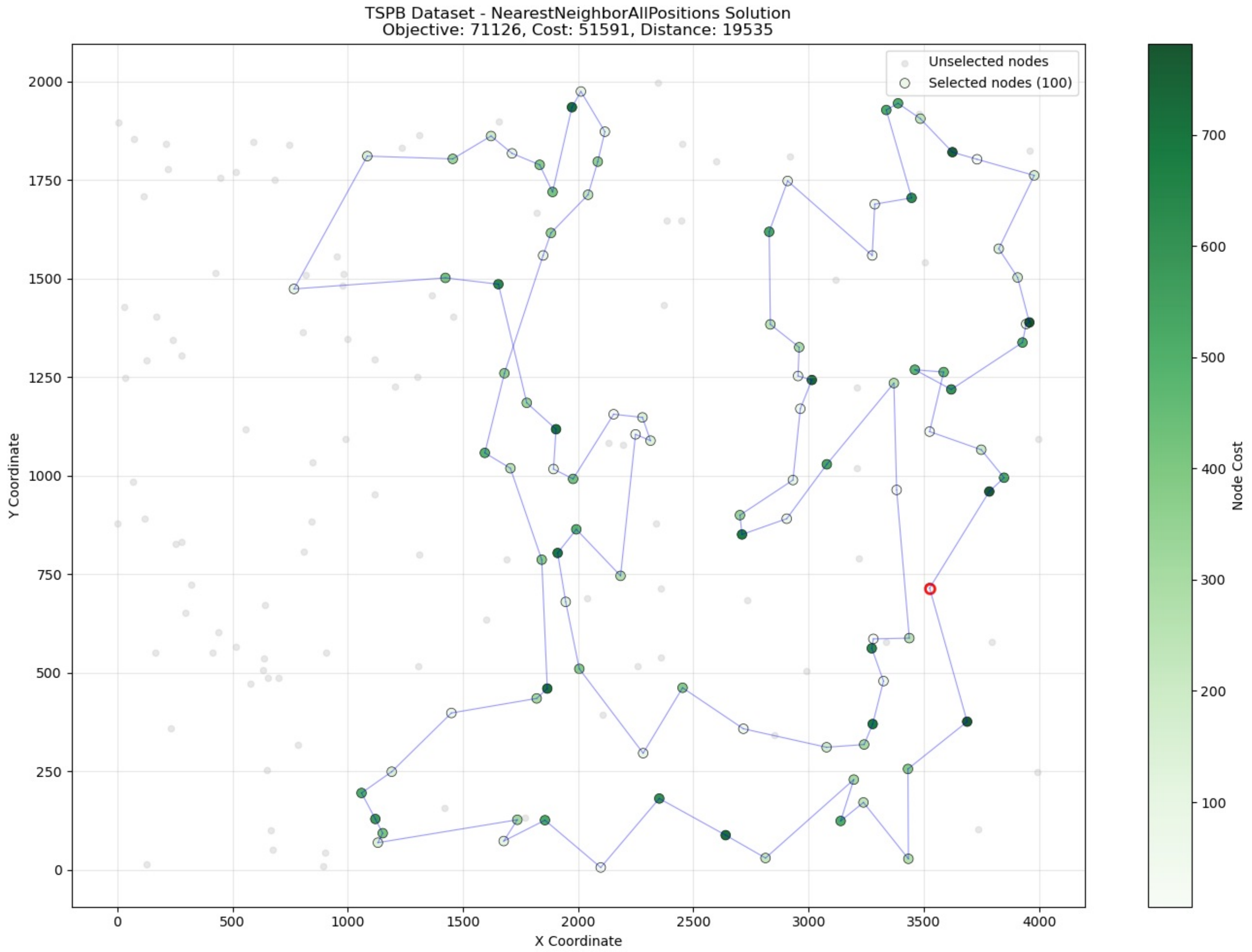
In [14]:
```
coords = tspb_df.iloc[:, 0].str.split(';', expand=True)
x_coords = coords[0].astype(int)
y_coords = coords[1].astype(int)
costs = coords[2].astype(int)
```

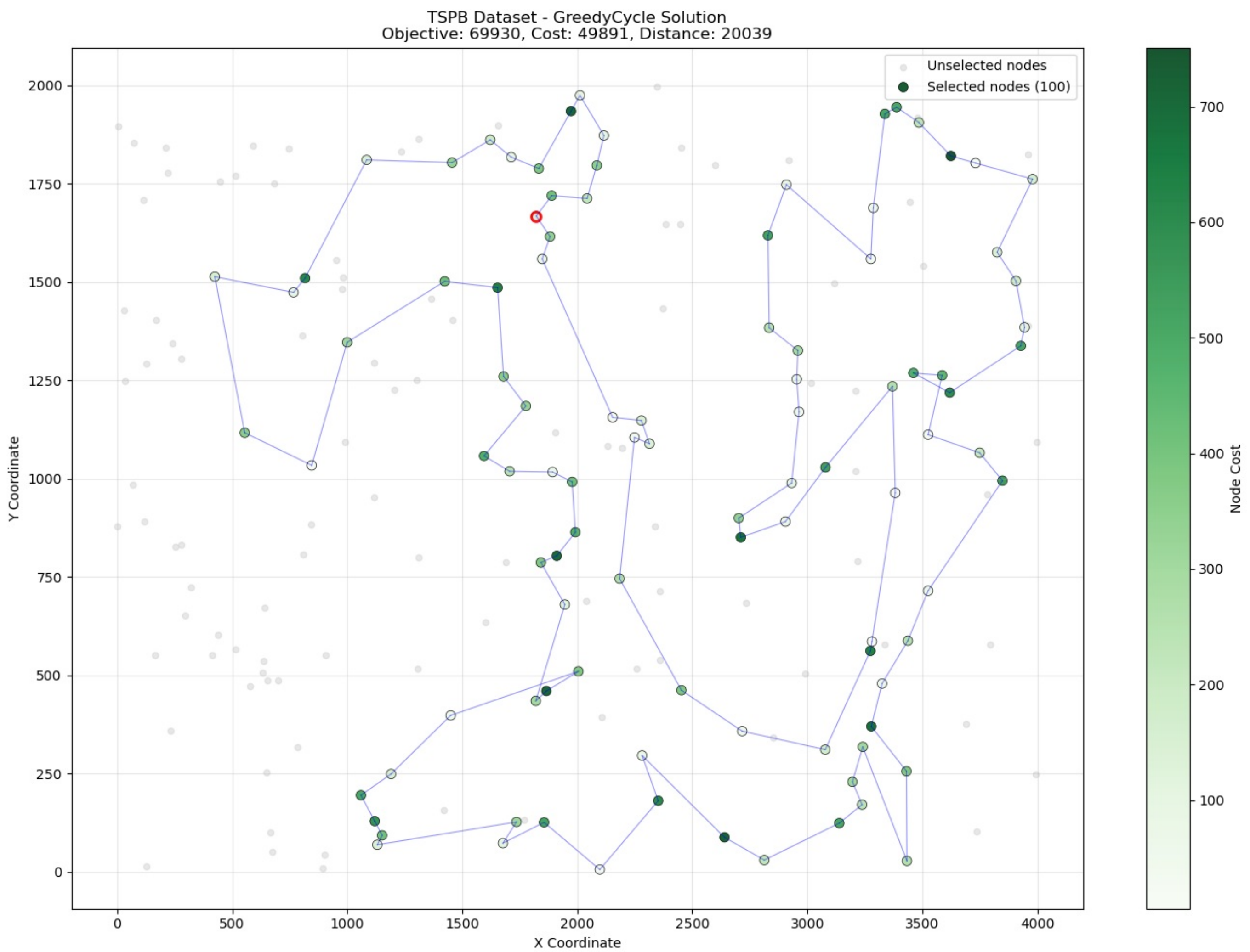
Graphs

In [15]:
```
for i, method in enumerate(methods):
    best_solution = solutions_B[solutions_B.index == i].iloc[0]
    plot_solution('TSPB', method, best_solution)
```


TSPB Dataset - RandomSolution Solution
Objective: 389001, Cost: 220425, Distance: 168576
Unselected nodes
Selected nodes (100)
X Coordinate
Y Coordinate
Node Cost


TSPB Dataset - NearestNeighborEndOnly Solution
Objective: 93692, Cost: 66268, Distance: 27424
Unselected nodes
Selected nodes (100)
X Coordinate
Y Coordinate
Node Cost





Solution Checker

Read the solution checker Excel file with multiple sheets

In [16]:
```
solution_checker_a = pd.read_excel("../Results/Solution checker.xlsx", sheet_name="TSPA")
solution_checker_b = pd.read_excel("../Results/Solution checker.xlsx", sheet_name="TSPB")
```

Extract solution in similar format as we had

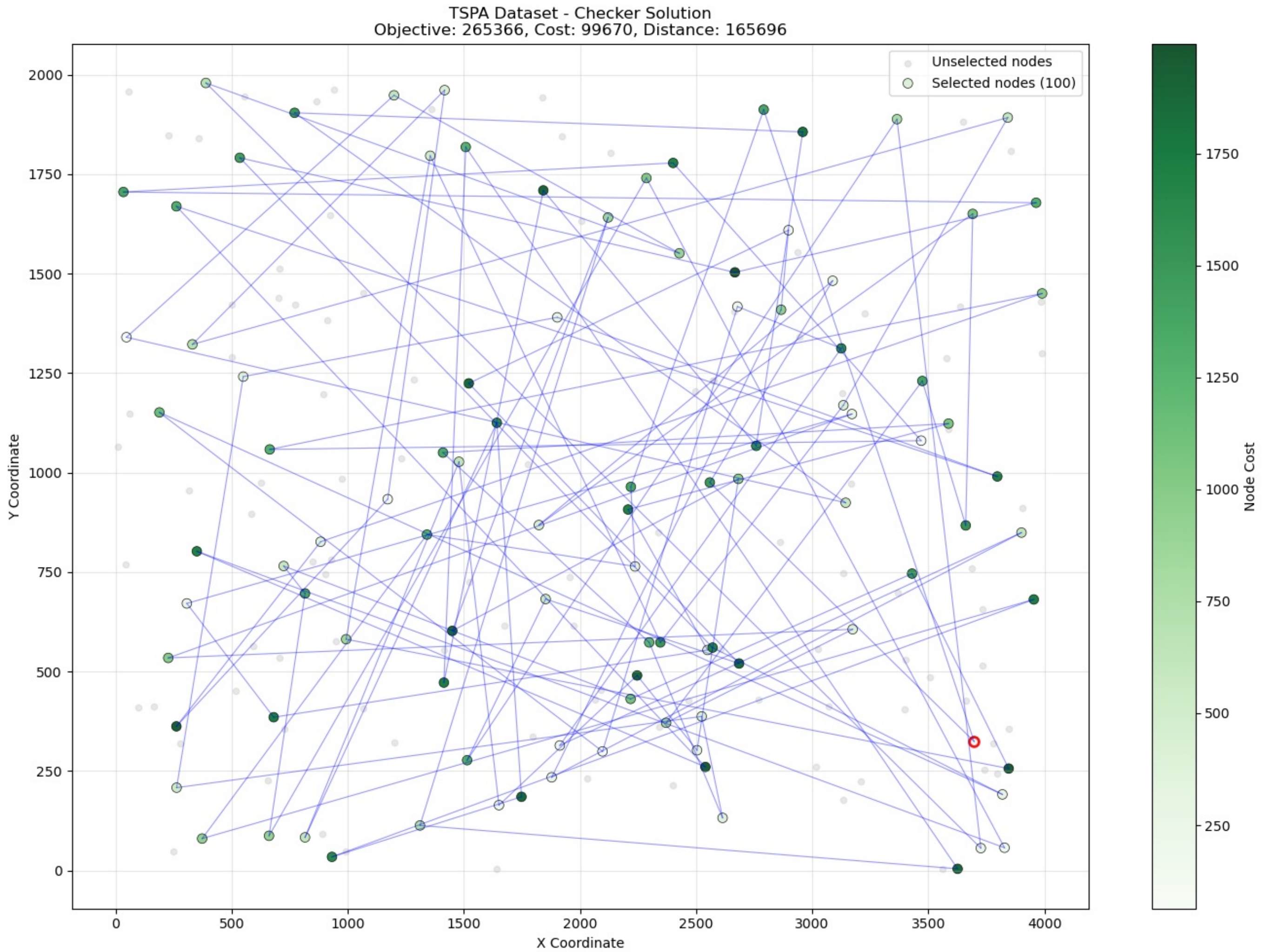
In [17]:
```
def extract_checker_solution(checker_df, instance_name):

    total_cost = checker_df.loc[0, 'cost.1']
    total_distance = checker_df.loc[0, 'Edge length']
    objective_function = checker_df.loc[0, 'Objective function']

    cycle_nodes = []
    for idx in range(1, len(checker_df)):
        node_id = checker_df.loc[idx, 'List of nodes']
        if pd.notna(node_id):
            cycle_nodes.append(int(node_id))

    cycle_str = '-'.join(map(str, cycle_nodes))

    checker_solution = {
        'Instance': instance_name,
        'Method': 'Checker',
        'TotalCost': int(total_cost),
        'TotalDistance': int(total_distance),
        'ObjectiveFunction': int(objective_function),
        'Cycle': cycle_str
    }

    return checker_solution

checker_solution_a = extract_checker_solution(solution_checker_a, 'TSPA')
checker_solution_b = extract_checker_solution(solution_checker_b, 'TSPB')

print("Checker Solution for TSPA:")
for key, value in checker_solution_a.items():
    print(f"{key}: {value}")

print("\nChecker Solution for TSPB:")
for key, value in checker_solution_b.items():
    print(f"{key}: {value}")
```

```
Checker Solution for TSPA:
Instance: TSPA
Method: Checker
TotalCost: 99670
TotalDistance: 165696
ObjectiveFunction: 265366
Cycle: 31-111-14-80-95-169-8-26-92-48-106-160-11-152-130-119-109-189-75-1-177-41-137-174-199-150-192-175-114-4-7
7-43-121-91-50-149-0-19-178-164-159-143-59-147-116-27-96-185-64-20-71-61-163-74-113-195-53-62-32-180-81-154-102-
144-141-87-79-194-21-171-108-15-117-22-55-36-132-128-76-161-153-88-127-186-45-167-101-99-135-51-112-66-6-156-98-
190-72-94-12-73-31

Checker Solution for TSPB:
Instance: TSPB
Method: Checker
TotalCost: 46383
TotalDistance: 162402
ObjectiveFunction: 208785
Cycle: 122-143-179-197-183-34-31-101-38-103-131-121-127-24-50-112-154-134-25-36-165-37-137-88-55-4-153-80-157-14
5-136-61-73-185-132-52-12-107-189-170-181-147-159-64-129-89-58-72-114-85-166-59-119-193-71-44-196-117-150-162-15
8-67-3-156-91-70-51-174-188-140-148-141-130-142-53-69-115-82-63-8-16-18-29-33-19-190-198-135-95-172-182-2-5-128-
66-169-0-57-99-92-122
```

Graphs

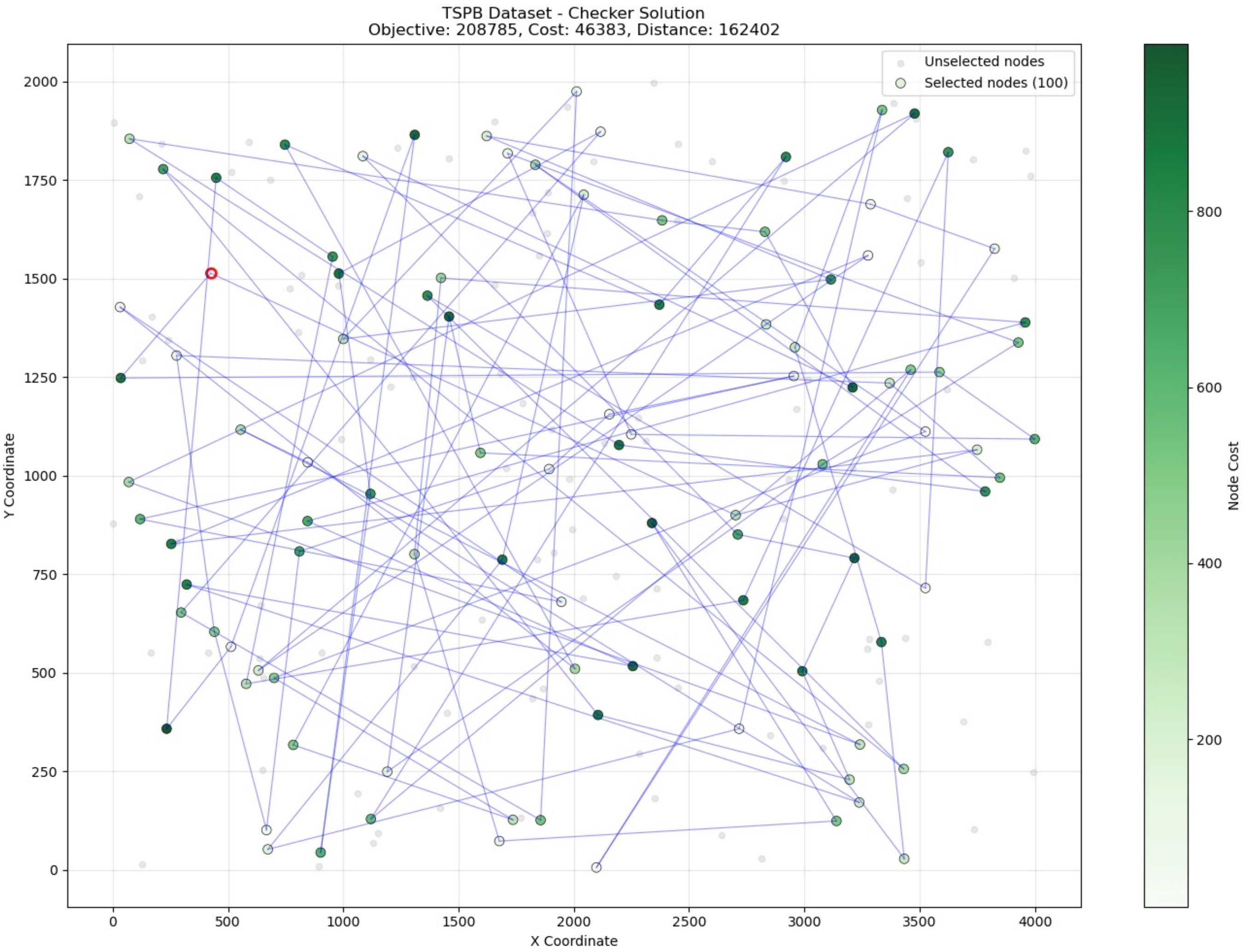
Instance A

In [18]:
```
plot_solution('TSPA', 'Checker', checker_solution_a)
```



Instance B

```
In [19]: plot_solution('TSPB', 'Checker', checker_solution_b)
```



Comparison of our solutions to the solution checker

Comparison Table: Heuristic Methods vs Solution Checker

Instance A (TSPA)

Method	Objective Function	Total Distance	Total Cost
Random Solution	384144	149144	235000
Nearest Neighbor (End Only)	123506	33374	90132
Nearest Neighbor (All Positions)	95066	23578	71488
Greedy Cycle	95066	23578	71488
Checker Solution	265366	165696	99670

Instance B (TSPB)

Method	Objective Function	Total Distance	Total Cost
Random Solution	389001	168576	220425
Nearest Neighbor (End Only)	93692	27424	66268
Nearest Neighbor (All Positions)	71126	19535	51591
Greedy Cycle	69930	20039	49891
Checker Solution	208785	162402	46383

Key Findings

1. Performance Comparison:

- The **Random Solution** method consistently produces the worst results, which is expected as it lacks any optimization strategy
- **Greedy heuristics** show significant improvement over random selection, demonstrating the value of constructive approaches
- Among the greedy methods, **Nearest Neighbor (All Positions)** and **Greedy Cycle** typically outperform the simpler **End Only** approach, as they explore more insertion possibilities

Conclusion: The comparison reveals that while our greedy heuristics produce feasible and competitive solutions efficiently, they consistently fall short of the checker solution's quality. This gap emphasizes the need for more sophisticated optimization techniques, such as local search operators or population-based metaheuristics, to approach optimal solutions. The foundation provided by these constructive heuristics, however, serves as an excellent starting point for more advanced optimization methods in subsequent iterations.

Link to the repository:

link GitHub Repository

Conclusions

This study implemented and evaluated four greedy heuristic methods for TSP. The implemented algorithms: Random Solution, Nearest Neighbor (End Only), Nearest Neighbor (All Positions), and Greedy Cycle demonstrate varying levels of performance across both test instances.

Algorithm Performance Hierarchy: The results clearly show a performance hierarchy where sophisticated greedy methods significantly outperform random selection. Greedy Cycle and Nearest Neighbor (All Positions) achieved the best results, producing identical solutions for TSPA and very competitive solutions for TSPB.

Why Nearest Neighbor (All Positions) and Greedy Cycle Outperform Other Methods:

- **Comprehensive Search Space Exploration**: Unlike the End Only variant that restricts insertions to the tour's end, both superior methods evaluate all possible insertion positions. This expanded search space allows them to find locally optimal placements that minimize the objective function increase at each step.
- **Reduced Greedy Trap Susceptibility**: By evaluating multiple insertion points, these algorithms are less likely to get trapped in poor local decisions early in the construction process. The End Only method, once it makes a suboptimal early choice, cannot recover by repositioning nodes.

The implemented greedy heuristics serve as effective constructive algorithms for this variant of TSP, providing good starting solutions.