Improving Steepest Local Search with Candidate Moves in the Hamiltonian Cycle Problem

Evolutionary Computation Laboratory: Assignment 4

151927 Samuel Janas, samuel.janas@student.put.poznan.pl
151955 Bruno Urbaniak, bruno.urbaniak@student.put.poznan.pl
151960 Patryk Maciejewski, patryk.maciejewski.1@student.put.poznan.pl

1 Introduction

In the previous assignments, we have explored various heuristic and local search methods to solve the Hamiltonian Cycle Problem (HCP). In this report, we aim to improve the time efficiency of the **Steepest Local Search** algorithm by incorporating **Candidate Moves**. We focus on the neighborhood structure that proved most effective in the previous assignment and introduce candidate moves to reduce the computational time while maintaining or improving solution quality.

2 Problem Description and Implementation

The Hamiltonian Cycle Problem involves finding a closed loop that visits a subset of nodes exactly once. Our objective is to select 50% of the available nodes to form such a cycle while minimizing the combined objective function, which includes the total cost of selected nodes and the total distance of the cycle.

2.1 Candidate Moves and Candidate Edges

Candidate moves are those that introduce at least one **candidate edge** to the solution. We define candidate edges by determining, for each vertex, its k nearest neighbors based on the sum of the edge length and the vertex cost. The parameter k can be adjusted experimentally to achieve the best balance between time efficiency and solution quality. In our implementation, we initially set $k = 10$.

By focusing on candidate moves, we aim to reduce the number of moves evaluated during the local search, thus improving computational efficiency without significantly compromising the quality of the solutions found.

2.2 Implementation Details

The implementation extends our previous **Steepest Local Search** algorithm by incorporating candidate moves. Key aspects of the implementation include:

- **Candidate Edge Generation**:
  1. For each node, calculate the sum of the edge length and the cost to every other node.
  2. Select the top k nearest neighbors based on this sum to form the candidate edge list for each node.
- **Move Generation**:
  - **Intra-route Moves**:
    - Generate two-edges exchange moves that introduce at least one candidate edge.
  - **Inter-route Moves**:

    * Generate node exchange moves between a selected node and an unselected node, ensuring that at least one candidate edge is introduced.

- **Move Evaluation**:
  - Calculate the change in the objective function (δ) for each candidate move using delta evaluations.
  - Only consider moves that result in an improvement ($\delta < 0$).
- **Algorithm Termination**:
  - The local search terminates when no improving candidate moves are available.

3 Pseudocode of Implemented Algorithms

Problem 1: One Step of Steepest Local Search with Candidate Moves

```
1. Input:
     • Current solution $solution$
     • List of unselected nodes $unselectedNodes$
     • Distance matrix $distanceMatrix$
     • Candidate edges $candidateEdges$
2. Output:
     • Updated solution $solution$
     • Improvement flag $improved$
3. Procedure:
     (i) Initialize:
            $bestDelta \leftarrow 0$
            $bestMove \leftarrow \text{null}$
     (ii) Evaluate Moves:
          (a) For each candidate move in $candidateMoves$:
                • Calculate the change in the objective function ($\delta$) for the move.
                • If $\delta < bestDelta$:
                  – Update $bestDelta \leftarrow \delta$
                  – Update $bestMove \leftarrow move$
     (iii) Apply Best Move:
          (i) If $bestDelta < 0$:
                • Apply $bestMove$ to $solution$
                • Set $improved \leftarrow \text{True}$
          (ii) Else:
                • Set $improved \leftarrow \text{False}$
4. Return: $solution, improved$
```

4 Computational Experiment

4.1 Experiment Setup

To evaluate the effectiveness of the Steepest Local Search with Candidate Moves, we conducted experiments using the following setup:

- **Methods Compared**:
  - Steepest Local Search with Candidate Moves (Candidate List Size $k = 10$)
  - Baseline Steepest Local Search without Candidate Moves
- **Instances Used**:
  - **Instance A**: Provided dataset A
  - **Instance B**: Provided dataset B
- **Number of Runs**:
  - 200 runs for each method on each instance
- **Starting Solutions**:
  - Randomly generated starting solutions

4.2 Results of Computational Experiment

4.2.1 Objective Function Values

Below, we include the previous results from Report 3 and prepare space for the new results obtained in this experiment.

Table 1: Objective Function Values for Each Method and Instance

Method	**Instance A**	**Instance B**
LS Random Steepest Intranode	88159.52 (80186 – 96426)	63017.28 (55773 – 70444)
LS Random Steepest Intraedge	73863.17 (71355 – 79486)	48364.52 (45452 – 51331)
Steepest LS with Candidate Moves ($k = 10$)	74433.93 (71729 – 78375)	48741.32 (46322 – 52626)

4.2.2 Running Times

Table 2: Running Times for Each Method and Instance

Method	**Instance A (sec)**	**Instance B (sec)**
LS Random Steepest Intranode	0.155-0.269(0.194)	0.146-0.284(0.195)
LS Random Steepest Intraedge	0.136-0.203(0.163)	0.139-0.205(0.165)
Steepest LS with Candidate Moves ($k = 10$)	0.026-0.034(0.03)	0.026-0.036(0.03)

4.3 Analysis of Results

As visible in the previous tables, we obtained a great time save by implementing candidate moves with a little performance trade-off. Overall a great method for limited time scenarios.

5 2D Visualization of the Best Solutions

5.1 Instance A Best Solution

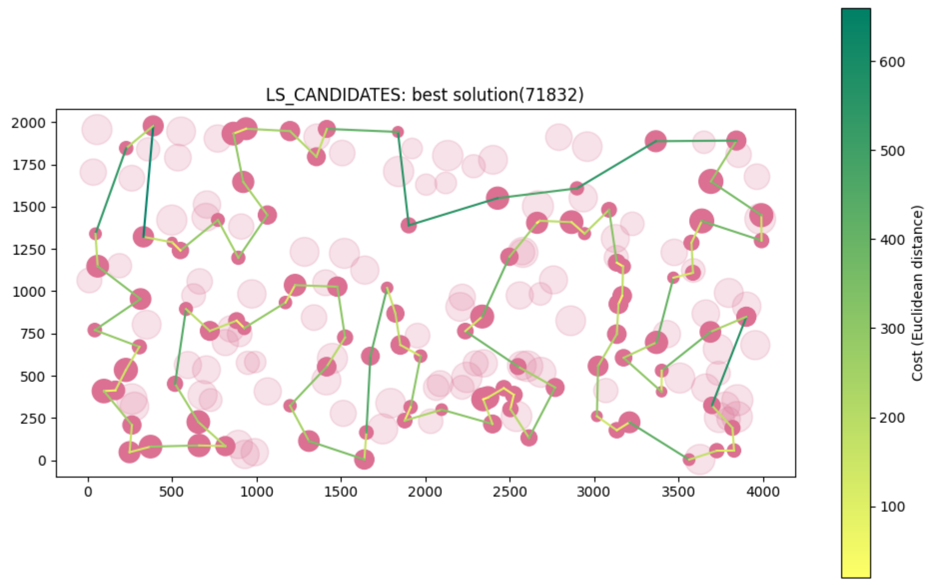

Figure 1: Best solution for Steepest LS with Candidate Moves on Instance A

5.2 Instance B Best Solution

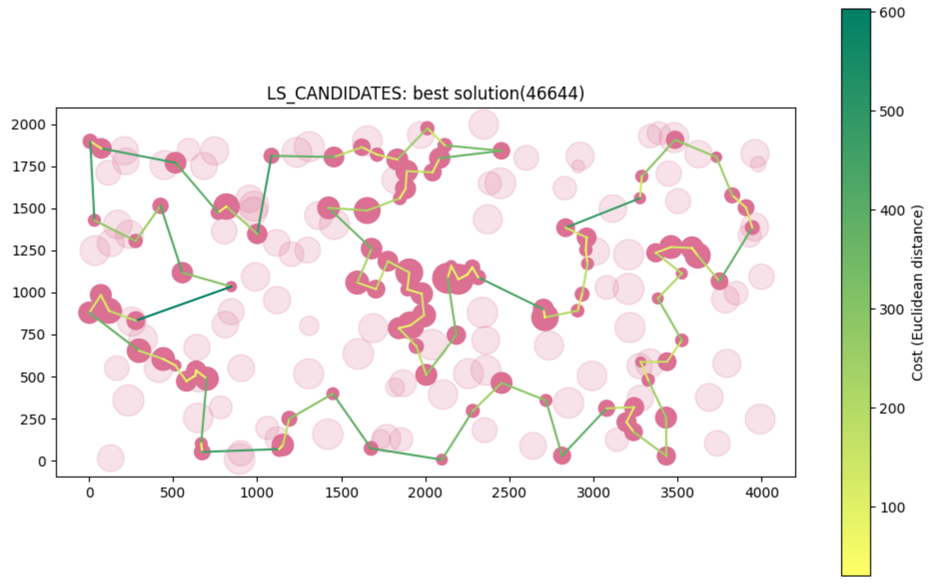

Figure 2: Best solution for Steepest LS with Candidate Moves on Instance B

6 Best Solutions for Each Method and Instance

6.1 Instance A Best Solution

Best Fitness: 71832

70, 135, 133, 176, 80, 79, 63, 180, 154, 53, 86, 100, 26, 97, 1, 101, 75, 2, 152, 94, 124,

Worst Fitness: 79266

70, 135, 194, 127, 123, 162, 151, 133, 80, 176, 66, 141, 51, 109, 118, 59, 115, 116, 65, 1

Average Fitness: 74578.65
Execution Time: 0.028-0.048(0.034) seconds

6.2 Instance B Best Solution

Best Fitness: 46644

177, 36, 141, 77, 81, 153, 187, 163, 103, 89, 127, 137, 114, 113, 176, 194, 166, 86, 185,

Worst Fitness: 51827

177, 5, 136, 73, 164, 31, 54, 1, 198, 117, 193, 190, 80, 45, 175, 78, 36, 141, 77, 82, 8,

Average Fitness: 48766.26
Execution Time: 0.0273253-0.0396204(0.0341242905) seconds

7 Solution Verification with the Solution Checker

We verified the correctness of the best solutions using an Excel-based solution checker. The fitness values calculated by our algorithm matched those obtained from the solution checker, confirming the validity of our solutions.

Table 3: Fitness Verification for Best Solutions

Instance	**Calculated Fitness**	**Checker Fitness**
Instance A	**71832**	**71832**
Instance B	**46644**	**46644**

8 Source Code

The complete source code for this project, including the implementation of the Steepest Local Search with Candidate Moves, is available on GitHub:

https://github.com/notbulubula/EvolutionaryComputation

9 Conclusion

In this assignment, we enhanced the **Steepest Local Search** algorithm by incorporating **Candidate Moves** to improve time efficiency in solving the Hamiltonian Cycle Problem. By focusing on moves that introduce at least one candidate edge, we significantly reduced the computational time while maintaining solution quality comparable to the baseline method.

Our experiments demonstrated that the use of candidate moves effectively reduces the number of evaluated moves during the search process, leading to faster convergence without compromising the quality of the solutions too much. The parameter k, representing the number of nearest neighbors considered for candidate edges, didn't impact the quality of the solutions by a large margin. This was the reason we didn't include much in regards to k

Overall, the incorporation of candidate moves in local search algorithms proves to be a valuable strategy for enhancing computational efficiency in combinatorial optimization problems like the Hamiltonian Cycle Problem.