Local Search with Candidate Moves

Students

- Patrick Molina, 157419
- Chihabeddine Zitouni, 158753

Code Repository

https://github.com/patrickmolina1/Evolutionary-Computation

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

- **Steepest Local Search**
    - Random Start + **Node Exchange**
    - Random Start + **Edge Exchange**

Computational experiment

CPU Specs

AMD Ryzen 7 5800H with Radeon Graphics

Cores	Threads	Clockspeed	Turbo Speed
8	16	3.2 GHz	4.4 GHz

Pseudocode

Algorithm: Steepest Local Search with Candidate Moves

Input: instance, startingSolutionType, intraRouteMoveType

Output: solution

1. Record startTime
2. Build candidateEdges map using buildCandidateEdges(instance, NUM_CANDIDATES)
3. Generate starting solution (random)
4. Initialize selectedNodes, cycle, selectedIds from starting solution
5. Initialize currentCost, currentDistance from starting solution
6. Set improved ← true
7. While improved:
   a. Set improved ← false
   b. Initialize bestDelta ← 0, bestDistanceDelta ← 0, bestMoveType ← null, bestMove ← null
   c. For each idx in cycle:
      i. Get nodeId ← cycle[idx]
      ii. Get neighbors ← candidateEdges[nodeId]
      iii. If neighbors is null, continue
      iv. Calculate prevIdx, nextIdx (with wraparound)
      v. Get prevNodeId ← cycle[prevIdx], nextNodeId ← cycle[nextIdx]
      vi. For each neighborId in neighbors:
         - If neighborId is selected (in selectedIds):
           INTRA-route candidate move:
           - Find neighborIdx in cycle
           - Skip if nodeId and neighborId are adjacent
           - Create move ← [idx, neighborIdx]
           - Calculate delta using calculateIntraDelta
           - If delta < bestDelta: update bestDelta, bestMove, bestMoveType ← "INTRA"
         - Else (neighborId not selected):
           INTER-route candidate move:
           - Calculate delta1 ← calculateInterDeltaDetailed for exchange (prevNodeId, neighborId)
           - If delta1.totalDelta < bestDelta: update bestDelta, bestDistanceDelta, bestMove ← [prevNodeId, neighborId], bestMoveType ← "INTER"
           - Calculate delta2 ← calculateInterDeltaDetailed for exchange (nextNodeId, neighborId)

- If delta2.totalDelta < bestDelta: update bestDelta, bestDistanceDelta, bestMove ← [nextNodeId, neighborId], bestMoveType ← "INTER"

   d. If bestDelta < 0 AND bestMoveType ≠ null:

      i. Set improved ← true

      ii. If bestMoveType equals "INTRA":

      - Apply applyIntraMove(cycle, bestMove, intraRouteMoveType)
      - Update currentDistance += bestDelt
      - Update currentCost += bestDelta

      iii. Else (bestMoveType equals "INTER"):

      - Extract selectedNodeId ← bestMove[0], nonSelectedNodeId ← bestMove[1]
      - Apply applyInterMove with selectedNodeId and nonSelectedNodeId
      - Update currentDistance += bestDistanceDelta
      - Update currentCost += bestDelta

8. Record endTime

9. Return solution with final selectedNodes, cycle, currentCost, currentDistance, execution time

Algorithm: Build Candidate Edges
Input: instance, k (number of candidates)
Output: candidateEdges map

1. Initialize candidateEdges ← empty map
2. Get n ← instance.nodes.size()
3. For each node in instance.nodes:
   a. Get id ← node.id
   b. Initialize neighbors ← empty list
   c. For each other in instance.nodes:
      i. If node.id equals other.id, continue
      ii. Calculate metric ← distanceMatrix[node.id][other.id] + other.cost
      iii. Add [other.id, metric] to neighbors
   d. Sort neighbors by metric (ascending)
   e. Initialize nearest ← empty set
   f. For i from 0 to min(k, neighbors.size()):
      i. Add neighbors[i][0] to nearest
   g. Put (id, nearest) into candidateEdges
4. Return candidateEdges

Results

Method	Instance A	Instance B
SteepestLS Candidate Node Exchange	98503.65 (89161 - 109877)	67747.81 (58473 - 79217)
SteepestLS Candidate Edge Exchange	77686.81 (73167 - 81696)	48517.50 (46027 - 52483)

SteepestLS Node Exchange	88108.59 (79789 - 97542)	62995.08 (55969 - 71020)
SteepestLS Edge Exchange	74038.50 (71587 - 77720)	48372.30 (45719 - 51098)

Running Times(ms)

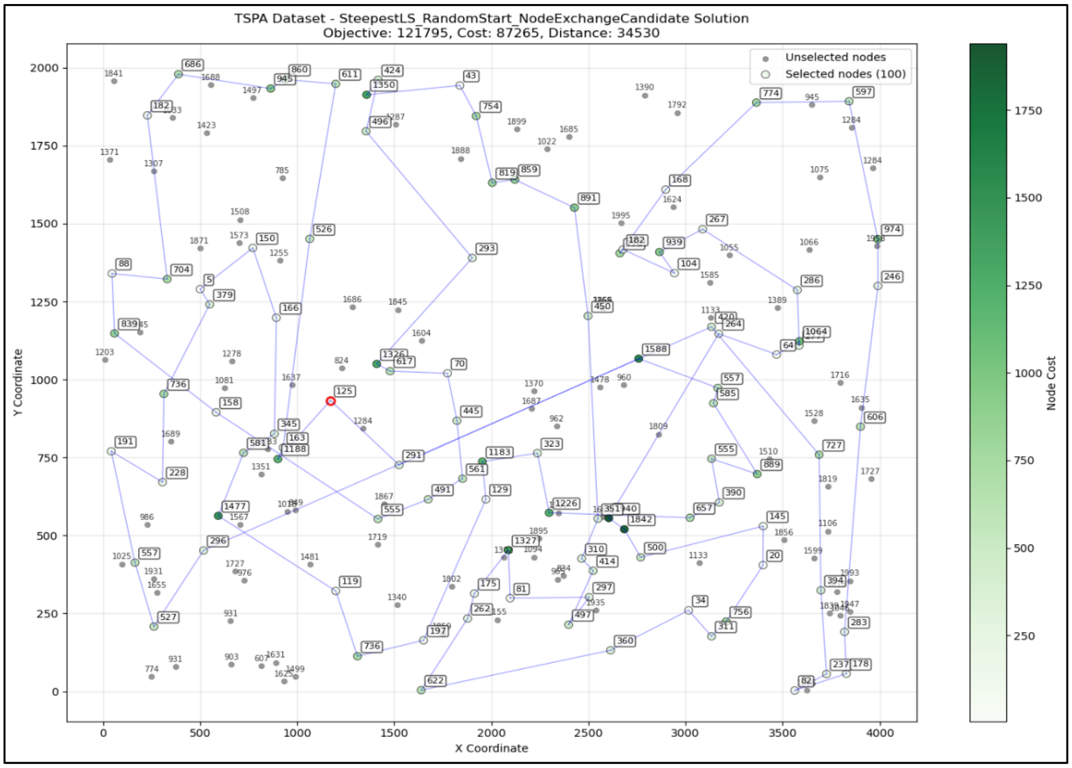
Method	Instance A	Instance B
SteepestLS Candidate Node Exchange	16.98 (13 - 85)	20.24 (16 - 99)
SteepestLS Candidate Edge Exchange	15.73 (13 - 22)	19.97 (17 - 28)

SteepestLS Node Exchange	257.92 (190 - 419)	266.45 (183 - 425)
SteepestLS Edge Exchange	222.91 (177 - 347)	228.37 (176 - 376)

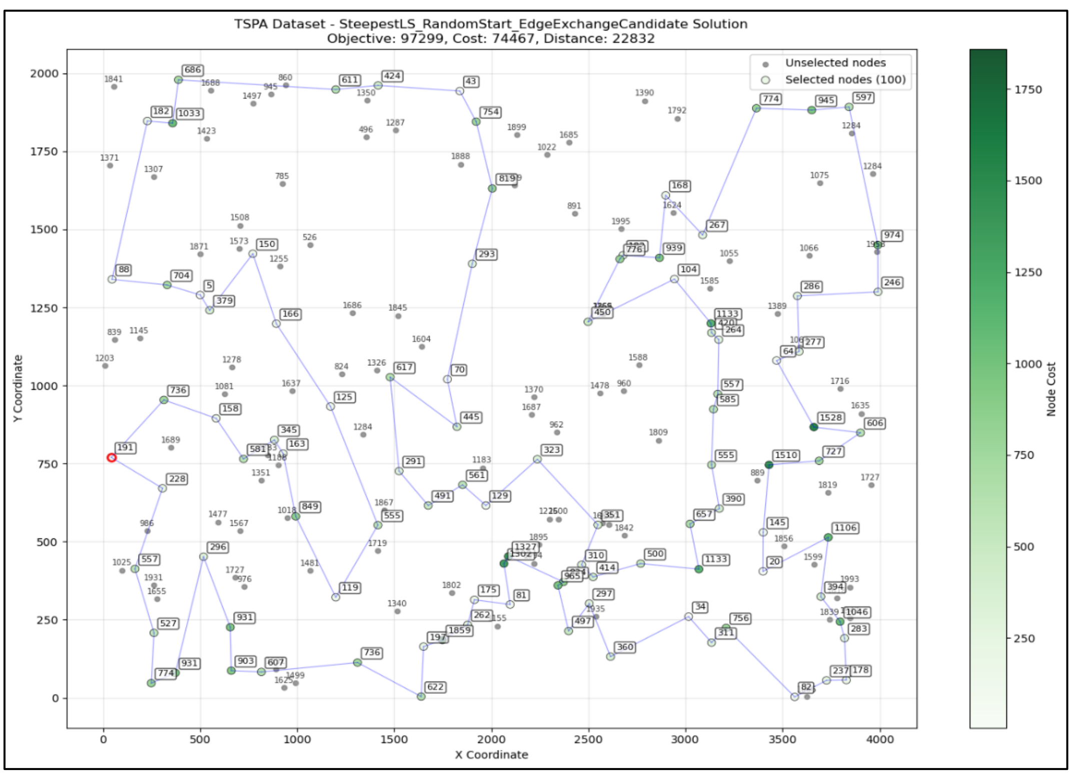
Best Solution

Instance A

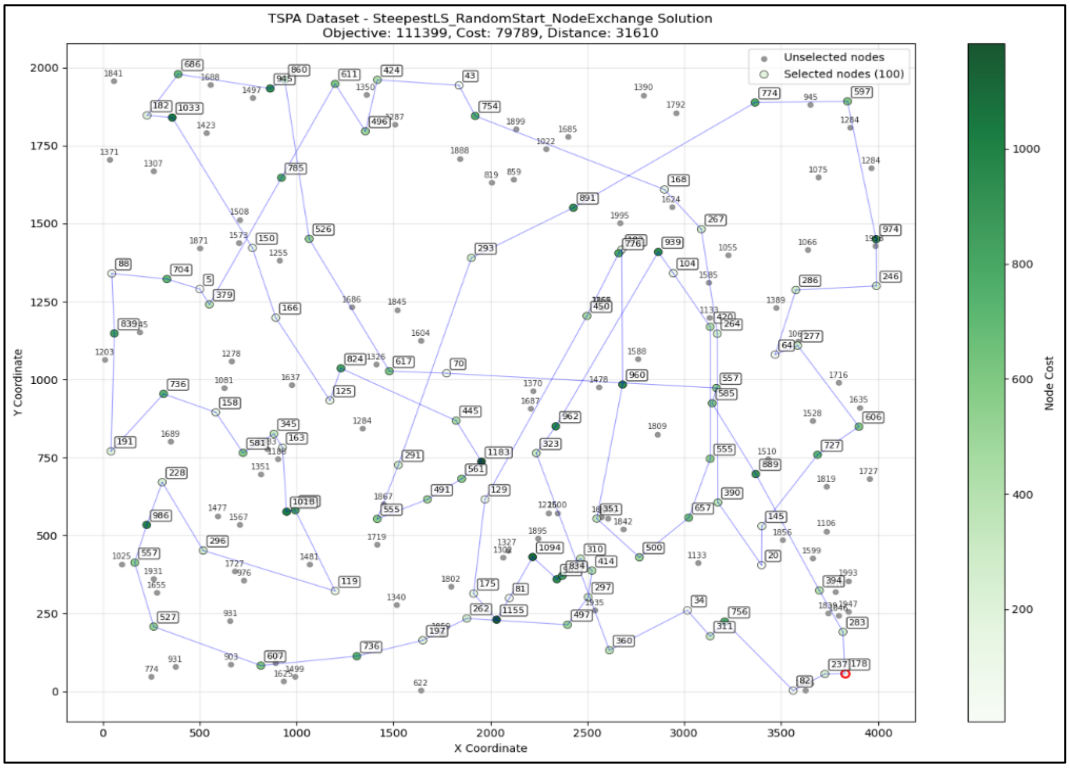
Steepest LS Candidate Node Exchange



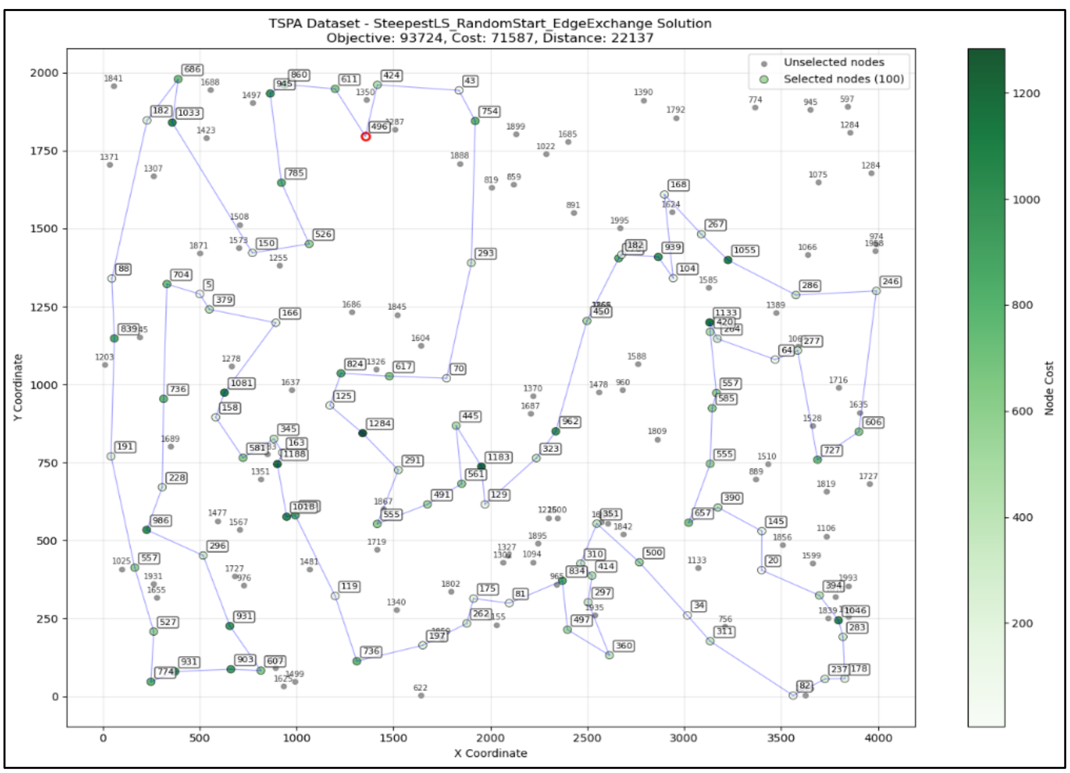
Steepest LS Candidate Edge Exchange



Steepest LS Node Exchange

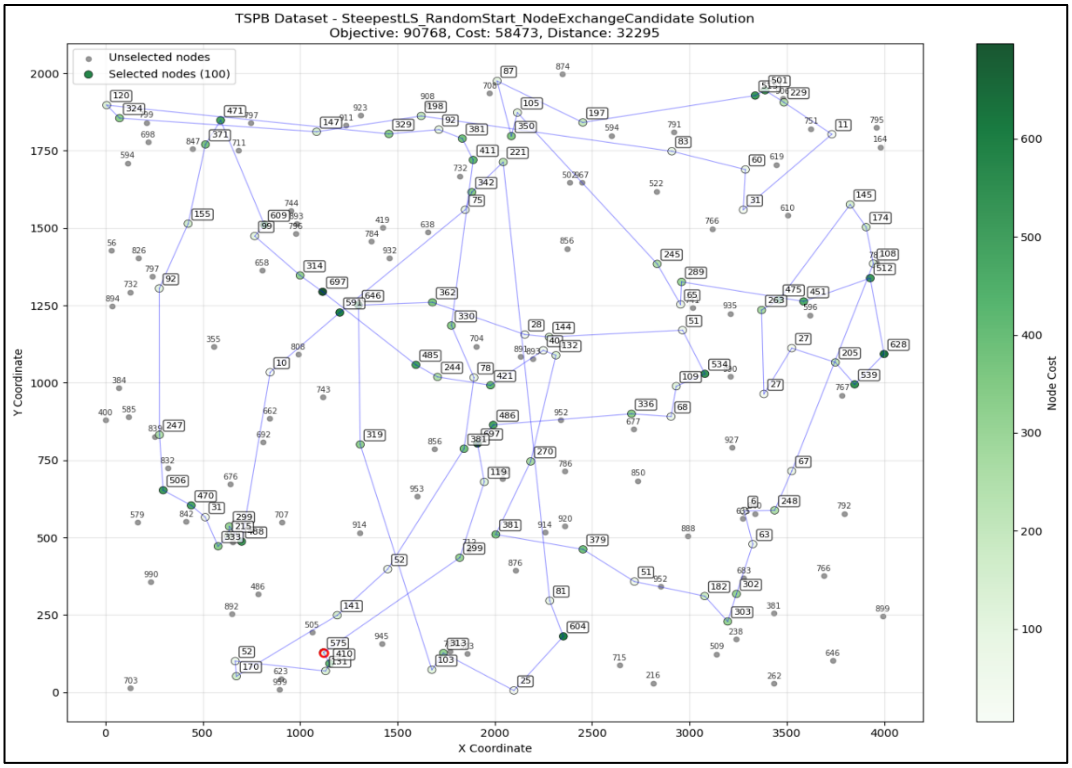


Steepest LS Edge Exchange

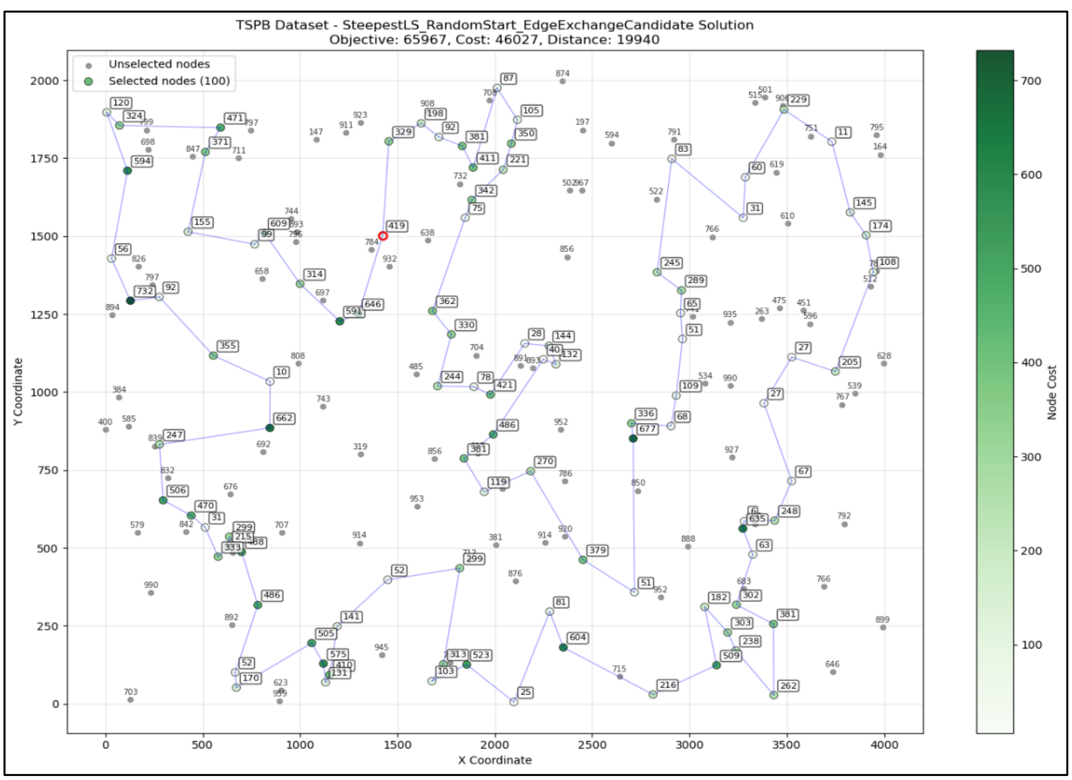


Instance B

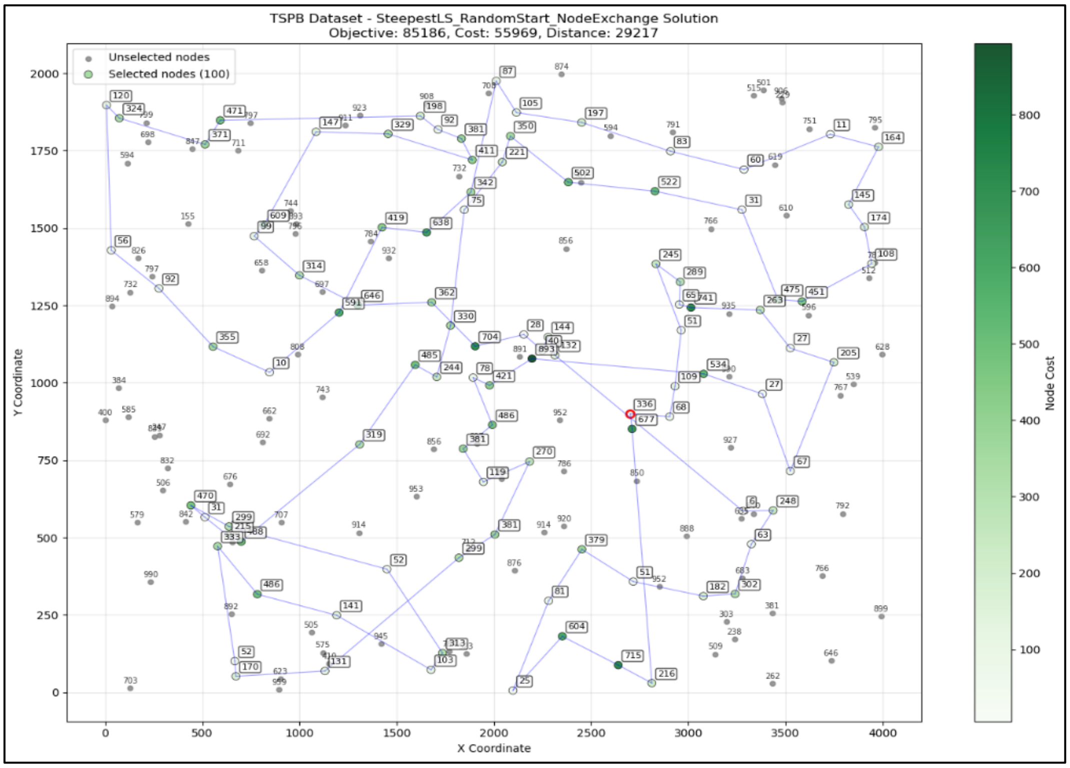
Steepest LS Candidate Node Exchange



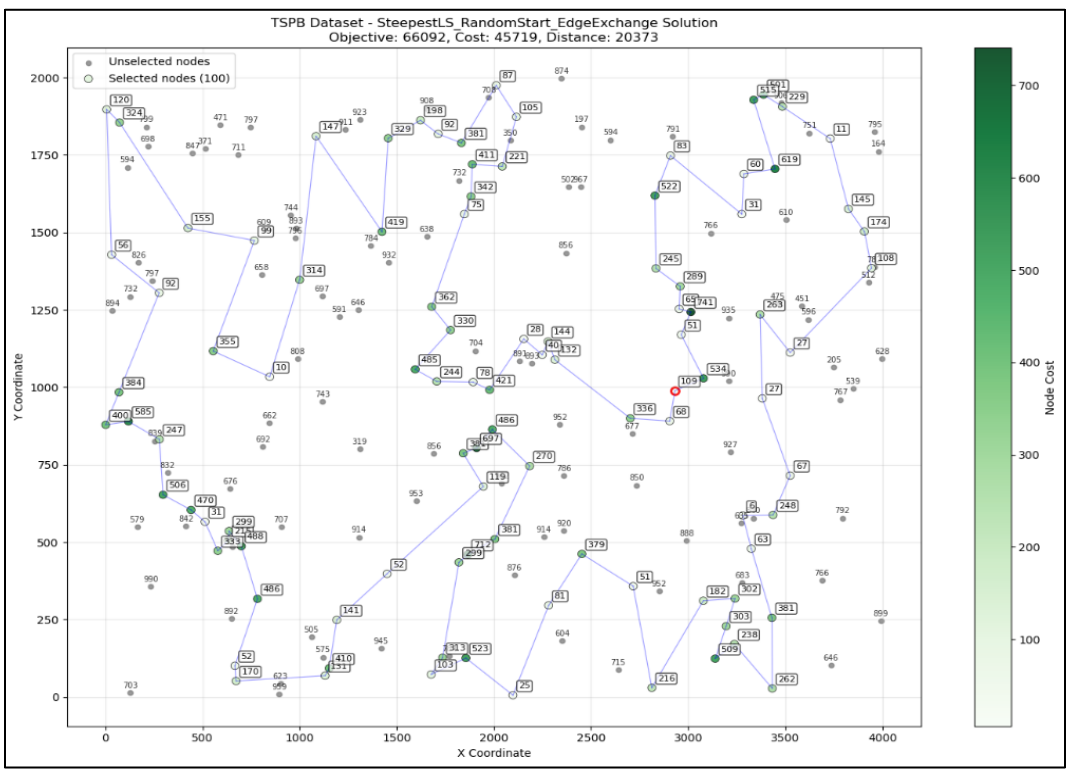
Steepest LS Candidate Edge Exchange



Steepest LS Node Exchange



Steepest LS Edge Exchange



Cycles

Instance A

Steepest LS Candidate Node Exchange

143-117-0-46-139-193-41-115-59-162-123-43-181-34-42-116-65-151-133-79-63-127-11-184-35-160-54-177-10-4-112-70-44-16-171-175-113-56-31-78-145-129-2-1-152-125-52-165-90-164-21-14-119-40-185-91-25-120-82-92-57-55-106-178-49-155-144-89-183-23-137-68-18-22-146-159-51-176-80-122-121-99-19-97-26-101-75-86-154-135-180-53-94-12-148-9-62-15-108-69

Steepest LS Candidate Edge Exchange

122-79-80-176-137-23-186-89-183-143-0-117-93-108-18-22-146-34-160-54-177-10-190-4-112-184-131-149-65-116-43-42-5-41-193-159-191-139-68-46-198-115-59-51-151-133-162-123-127-135-154-180-158-53-182-121-26-97-1-101-86-75-2-152-167-52-55-57-92-129-82-120-44-16-171-175-113-56-31-78-145-157-196-81-90-165-40-185-106-178-49-14-144-62-9-15-148-124-94-63

Instance B

Steepest LS Candidate Node Exchange

142-21-8-56-144-143-106-124-128-62-109-29-139-74-25-36-61-141-97-77-145-195-168-118-121-73-54-31-193-117-198-156-1-135-122-133-10-191-90-51-98-182-138-160-0-35-111-82-81-153-163-89-103-113-176-194-166-94-47-148-130-95-86-185-179-172-57-66-99-55-18-34-3-15-70-155-4-149-28-20-183-140-152-188-147-107-40-6-169-132-13-11-33-104-177-5-80-190-175-78

Steepest LS Candidate Edge Exchange

134-6-188-169-132-13-70-3-15-145-195-168-139-11-138-33-160-29-109-35-0-144-104-8-111-81-153-159-143-106-124-62-18-55-34-152-183-140-28-20-148-47-94-179-185-86-166-194-176-180-113-103-114-137-127-89-163-165-187-97-77-141-91-36-61-21-177-5-78-175-142-45-80-190-136-73-54-31-193-117-198-156-1-112-121-131-135-102-63-100-40-107-10-133-122-90-191-51-118-74

Conclusion

- The implementation of candidate move strategies represents a significant advancement in computational efficiency for steepest local search algorithms applied to the TSP variant with node selection costs. Testing **both Node Exchange** and **Edge Exchange with Candidate Moves** in both instances demonstrates that intelligent move filtering can maintain the solution quality of exhaustive local search while achieving substantial computational speedups.
- The results reveal that **Edge Exchange with Candidate Moves** typically produces superior solutions compared to **Node Exchange with Candidate Moves**, consistent with findings from previous assignments regarding neighborhood effectiveness. However, the critical achievement lies in the dramatic reduction of execution time while maintaining competitive solution quality relative to full neighborhood exploration.
- The requirement that moves must introduce at least one candidate edge prevents trivial exchanges while ensuring that the search explores meaningful modifications to the solution structure demonstrates that intelligent neighborhood restriction can achieve near-optimal solution quality at a fraction of the computational cost.