**MCS 271 Project**

Gary Lao (21444849)

Wenting Zheng (52842020)

Caitlyn Zhang (68047615)

# 

**Contents**

[**1. Traveling Salesman Problem**](#_mx7xgz36udiq) **2**

[**2. Branch-and-Bound Depth-First Search**](#_8jc4xbbaz2w) **3**

[2.1 Data Structure(s):](#_dfsyxm70oogo) 3

[2.2 Algorithm (pseudo-code):](#_r4e5pxr9ddhn) 4

[2.3 Algorithm Explanation:](#_59ynrgt20zci) 5

[2.4 Assess Time/Space complexity:](#_qtcyvy9vcmz1) 5

[2.5 Experimental Results:](#_nmmvylmfjefq) 5

[2.6 Observation and Assessment:](#_fmcqs6elqe6o) 6

[**3. Stochastic Local Search with Simulated Annealing**](#_vznlc5a57xi1) **7**

[3.1 Data Structure(s):](#_e2grs3nsqqza) 7

[3.2 Algorithm (pseudo-code):](#_ss73lgatem9b) 7

[3.3 Algorithm Explanation:](#_16eytb7amvte) 8

[3.4 Assess Time/Space complexity:](#_z2envke39tvl) 9

[3.5 Experimental Results:](#_6p4s8dt90gwo) 9

[3.6 Observation and Assessment:](#_x47b20uu9fyo) 10

[**4. Reference**](#_4xud0exkmz4h) **12**

# **1. Traveling Salesman Problem**

The Traveling Salesman Problem is a combinatorial optimization problem that calculates the shortest possible route between two cities with the constraint of using a route that visits each city exactly once and returns to the original destination. The primary objective is to minimize the total cost of the tour. While this objective is a simple and straightforward statement, TSP is notoriously difficult to efficiently solve due to its classification as an NP-hard problem; meaning, in its worst-case scenario, no known polynomial time exists, which adds significant complexity to its solution.

The “cost” between two cities also does not necessarily have to be their physical distance; cost can also pertain to the cost of a plane ticket. In the case of flight cost, the cost from City A to City B can differ from the cost from City B to City A. Therefore, not all TSP is consistent with the Triangle Inequality Theorem which states that, in a triangle, the sum of the length of any two sides is greater than the third. However, for the sake of simplicity, our analysis only considers physical distance, specifically Manhattan Distance.

The Traveling Salesman Problem, initially formulated by W.R. Hamilton and Thomas Kirkman in the 1800s, remains unsolved. While seemingly manageable for a small number of cities, its complexity becomes exponential with larger instances, resulting in possible solutions for n cities. The Wheat and Chessboard Problem illustrates exponential growth, emphasizing the challenge of computing factorial solutions for TSP. Despite an approach reducing time complexity to , the computational demands remain high. Efficient algorithms are crucial due to the impracticality of computing all solutions, making finding a reasonably good solution a priority.

This document focuses on the AI component of the TSP solution, particularly the design and implementation of two distinct algorithms: Branch and Bound Depth First Search (BnB DFS) and Stochastic Local Search (SLS). Both algorithms offer different strategies for tackling the problem. BnB DFS systematically explores the search space by building a tree of possible solutions and pruning suboptimal paths using a problem-specific heuristic. On the other hand, SLS relies on a heuristic evaluation of the current solution and employs a combination of deterministic and probabilistic rules to explore the solution space, aiming to escape local optima and converge on a global optimum.

# **2. Branch-and-Bound Depth-First Search**

Branch-and-bound is a search algorithm that systematically explores the solution space of combinatorial problems. It builds a tree of subproblems, "branching" to explore different routes and "bounding" to eliminate non-promising routes based on a heuristic lower bound. The heuristic helps estimate the minimum cost from the current node to complete the tour, allowing the algorithm to prune branches that cannot lead to an optimal solution. Depth-First Search, on the other hand, is a graph traversal algorithm that starts at the root node and traverses as deeply as possible before backtracking and exploring another route.

Putting these two algorithms together, we get a Branch-and-Bound Depth-First search solution to the TSP problem. Using Depth First Search, this solution will traverse through the graph nodes and calculate the cost of the path, f(n) = g(n) + h(n), where g(n) is the known cost of the path from the start node to the current node and h(n) is the heuristic, the approximation of the unknown cost of the path from the current node to the end node. The Branch-and-Bound portion of this solution utilizes an upper bounded path cost that helps to prune sub-optimal routes. For each repetition, as long as f(n) is less than the upper bound, we continue to traverse. As soon as f(n) becomes larger than the upper bound, we prune and backtrack to another route. For the first traversal, the upper bound is set to infinity.

## 2.1 Data Structure(s):

The data structures we need for this implementation are a stack and a list: a stack for keeping track of the path/backtracing and a list for recording the visited cities.

## 2.2 Algorithm (pseudo-code):

function NN\_heuristic(graph, current\_city, visited, start\_city):

heuristic\_cost = 0

unvisited\_cities = graph.cities - visited

while unvisited\_cities is not empty:

nearest\_city = find the nearest city from current\_city in unvisited\_cities

heuristic\_cost += graph.distance(current\_city, nearest\_city)

current\_city = nearest\_city

remove nearest\_city from unvisited\_cities

heuristic\_cost += graph.distance(current\_city, start\_city) # back to the start point

return heuristic\_cost

function DFS (graph, current\_city, path, visited, current\_cost, start\_city, best\_cost, best\_path):

if all cities are visited:

complete\_path = (path.push(start\_city)).toList() # back to the start point

complete\_cost = current\_cost + graph.distance(current\_city, start\_city)

if complete\_cost < best\_cost:

best\_cost = complete\_cost

best\_path = complete\_path

return

for next\_city in all the neighbors of current\_city:

if next\_city is not visited:

g\_next = current\_cost + graph.distance(current\_city, next\_city)

h\_next = NN\_heuristic(graph, next\_city, visited)

f\_next = g\_next + h\_next

if f\_next < best\_cost:

path.push(next\_city)

visited.append(next\_city)

DFS(graph, next\_city, path, visited, next\_cost, start\_city)

path.pop()

visited.pop()

function BnB\_DFS(graph):

best\_cost = Infinity

best\_path = an empty list

start\_city = select any city from the graph

current\_tour = a stack with only the start\_city inside

visited = a list with only the start\_city inside

DFS(graph, start\_city, current\_tour, visited, 0, start\_city, best\_cost, best\_path)

return best\_path, best\_cost

## 2.3 Algorithm Explanation:

Note that, in our design, we implement the Nearest Neighbor as our heuristic algorithm. This pseudo-code provides a framework for the Branch-and-Bound DFS algorithm with the Nearest Neighbor heuristic.

*BnB\_DFS function*: This function initializes the search and sets the best tour distance, the upper bound, to infinity, meaning that no tour has been found yet. It then starts the DFS process from an initial city.

*DFS function*: It recursively explores further cities, extending the path as it goes. During the process, it checks if all cities are visited, then calculates the total cost of returning to the start city. If this is a new best, it updates the best path and cost. If there are still unvisited cities, for each unvisited city, it calculates the estimated total cost (current cost plus the estimate to complete the tour obtained from the Nearest Neighbor heuristic) and continues with DFS if this estimate is under the current best cost (Upper bound).

*NN\_heuristic function*: It estimates the cost to complete the tour from the given city using the Nearest Neighbor heuristic.

## 2.4 Assess Time/Space complexity:

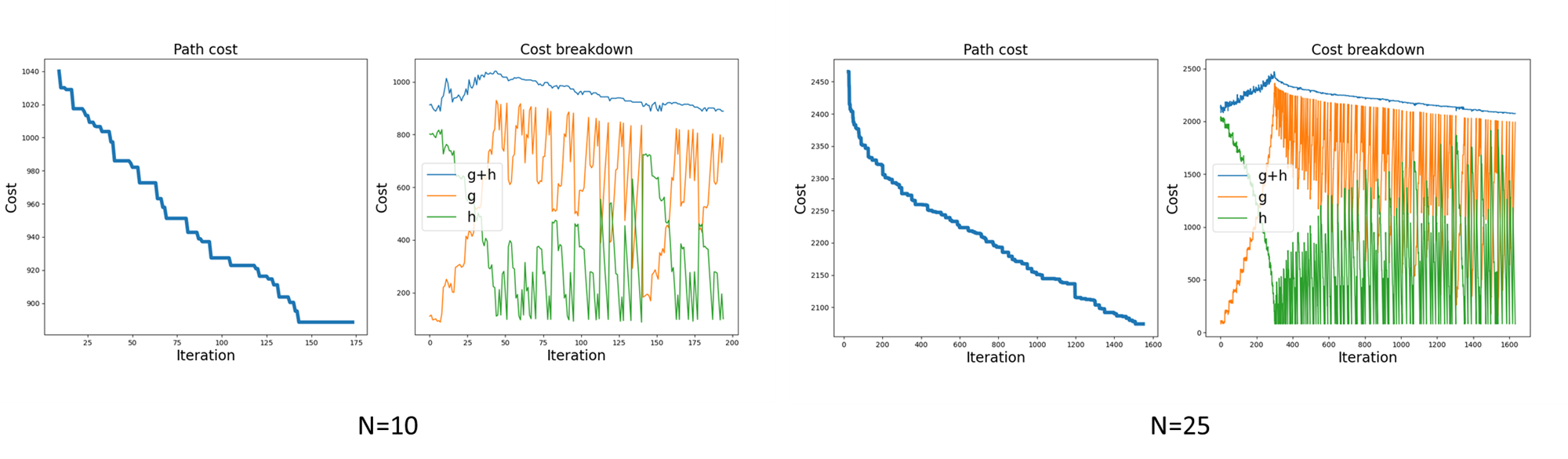
*Time complexity:* Although BnB may significantly reduce the actual time complexity via the effectiveness of pruning, this algorithm is still a DFS. In the worst case, this could be , where is the branching factor (average number of successors per state) and is the maximum depth of the search tree. It happens when the solution is at the deepest level or when there's no pruning possible.

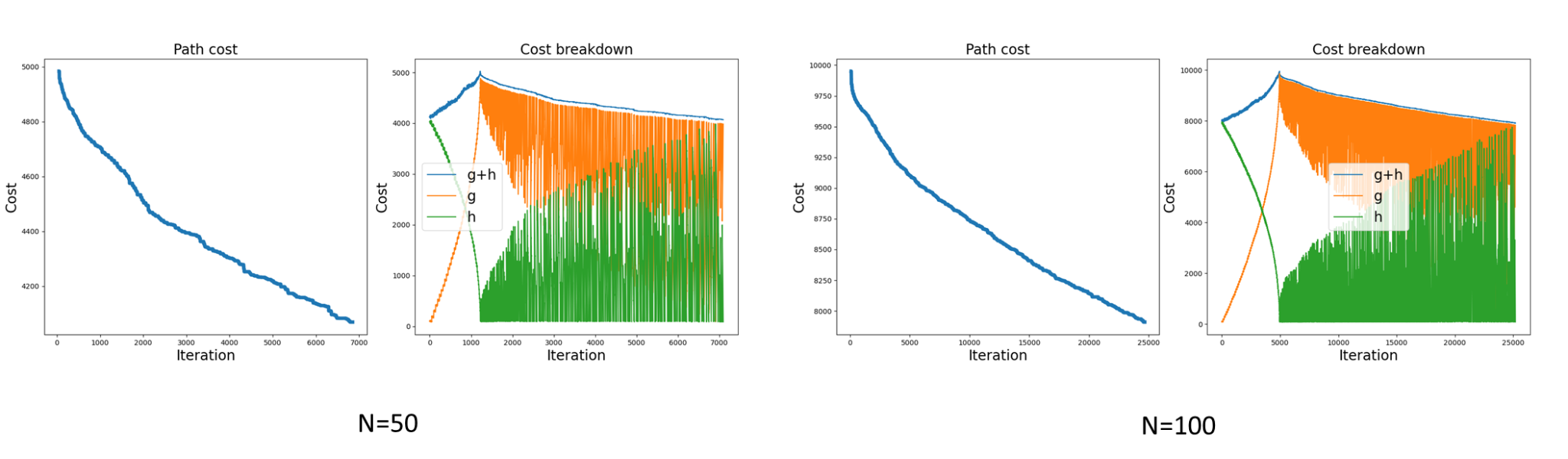
*Space complexity:* It can be less since it prunes branches that exceed the current best solution, but the worst-case scenario remains the same as regular DFS, that is, as stated in [3].

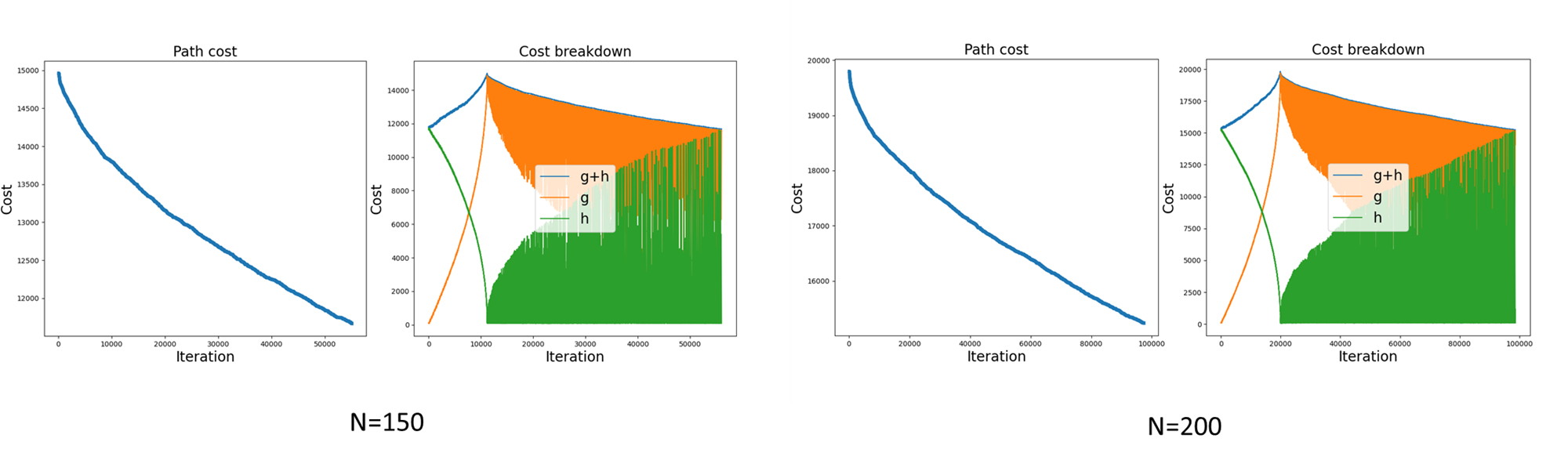
## 2.5 Experimental Results:

| N | Min Cost | Nodes Expanded | Runtime (s) |
| --- | --- | --- | --- |
| 10 | 888.546 | 174 | 0.008 |
| 25 | 2074.251 | 1550 | 0.723 |
| 50 | 4068.702 | 6850 | 24.632 |
| 100 | 7909.755 | 24660 | 520.781 |
| 150 | 11666.308 | 55024 | 4565.553 |
| 200 | 15234.619 | 97316 | 18498.395 |

***Table 1. Running results for BnB DFS with graph mean = 100, s.d. = 10***

****

****

****

***Figure 1. Detailed statistics of Table 1 on BnB DFS***

## 2.6 Observation and Assessment:

As the results shown above, we can see that the branch-and-bound depth-first search with the nearest neighbor heuristic can work well in some scenarios. The nearest neighbor heuristic often produces a reasonable path by always moving to the nearest unvisited city. Branch-and-Bound further explores and prunes branches in the search space, ensuring that the algorithm does not explore paths that are already longer than the best-known solution.

However, this approach only really works well for smaller instances of TSP, where the number of cities is not too large. For smaller-scale problems (N < 100), we achieve relatively precise results within a reasonable time frame. In cases where N is equal to or exceeds 100, we encounter prohibitive execution times due to the exponential increase in the search space. It can be computationally expensive for large instances, as the number of possible routes increases factorially with the number of cities. The depth-first nature of the search can also spend a lot of time exploring suboptimal branches.

The obvious area for improvement in our algorithm is thus the heuristic. KNN heuristic is less sophisticated than others (such as k-opt, Christofides algorithm, Lin-Kernighan Heuristicand, etc [7]) and thus allows for our algorithm to spend more time than necessary exploring the sub-optimal branches. Essentially, using a more robust heuristic algorithm could help us to prune/factor out sub-optimal paths much earlier, which will help with our program runtime.

For future improvement, it would also be helpful to leverage modern computational techniques such as parallel processing [8]. Working multithreading into our algorithm would also help in the runtime, especially in the case that we allow multiple threads to traverse different branches at the same time. Adding multithreading would increase the implementation difficulty of our algorithm especially in terms of isolating shared variables such as visited nodes, heuristic, etc. to allow for coherent and accurate read/writes. However, despite the increased implementation difficulty, the runtime of our algorithm would theoretically decrease due to parallel processing allowing for multiple branches to be explored at once.

# **3. Stochastic Local Search with Simulated Annealing**

Stochastic Local Search (SLS) is a general optimization technique used to find approximate solutions to combinatorial optimization problems. Within the realm of SLS, "simple SLS methods" permit exploration beyond local optima by considering worsening neighbors. In contrast to complex strategies, these methods rely on a single search step within a fixed neighborhood relation (Hoos & Stützle, 2015, section 54.3). Simulated Annealing (SA) is a prime example, employing a temperature-based strategy to accept moves to worse solutions. The acceptance of worsening solutions is governed by a temperature parameter, which is gradually reduced over the course of the optimization process according to a predefined schedule.

## 3.1 Data Structure(s):

We will use vectors to keep track of the current solution. The state space is a full tour and the objective function is the cost of the tour.

## 3.2 Algorithm (pseudo-code):

function Neighbors(solution):

neighbors = {}

for city in solution.nodes:

for other\_city in solution.nodes:

= copy(solution)

.swap(city, other\_city)

neighbors.add()

return neighbors

function SLS\_with\_SA(problem, schedule) returns solution state s :

= Random\_Solution(problem)

for t = 1 to do

T = schedule(t)

if T = 0 :

return

= randomly pick one from Neighbors(s)

= value() – value()

if < 0 **| |** rand() < :

s =

## 3.3 Algorithm Explanation:

*Neighbors function:* The algorithm starts by generating a random initial solution, denoted as , for the TSP problem. This function generates all possible neighbors of a solution by swapping two cities of the given solution to a different position.

*SLS\_with\_SA function:* The algorithm starts by generating a random initial solution, denoted as , for the TSP problem. The main loop iterates from 1 to a specified maximum number of iterations (t). The temperature T is determined by the cooling schedule function schedule(t).

Check for Termination: If the temperature T drops to zero, the algorithm terminates, and the current solution is returned.

Neighbor Generation: A neighboring solution is randomly chosen from the set of neighbors of the current solution . We generate the neighbor solution by swapping two cities in the path.

Evaluate the Energy (Line 6): The energy E is computed as the difference in cost between the new solution and the current solution .

Acceptance Criteria (Lines 7-9): If is better (lower cost) than the current solution , it is accepted. If is worse, its acceptance probability is determined by the . The acceptance rate drops as the temperature T decreases, allowing the algorithm to explore more at the beginning but converge towards the end.

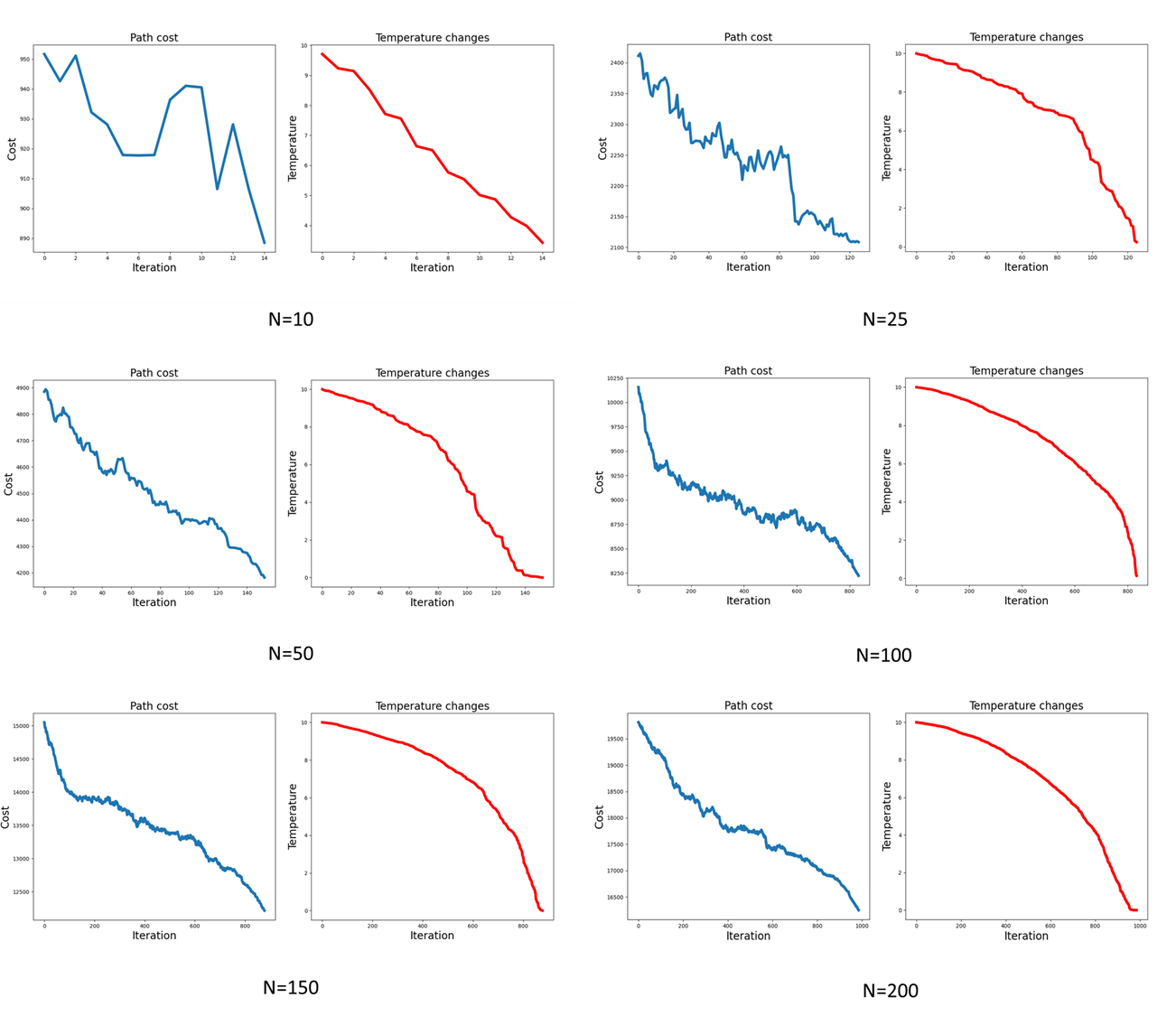
## 3.4 Assess Time/Space complexity:

A neighboring solution can be generated by swapping positions of two cities in the path. Therefore, the cost of this neighbor solution can be efficiently computed based on the cost of the current solution, eliminating the need to iterate through all cities. This allows for a constant-time computation of the cost, resulting in a time complexity in each iteration of SA. For the overall SA algorithm, given cities, we need time and space to compute the initial solution. If is the number of iterations of the for loop, the overall complexity becomes , with space complexity limited to as only one current solution and one neighbor solution need to be tracked.

## 3.5 Experimental Results:

| N | Min Cost | alpha | Iterations | Runtime (s) |
| --- | --- | --- | --- | --- |
| 10 | 888.5458 | 0.01 | 1000 | 0.0313 |
| 25 | 2101.8843 | 0.001 | 10000 | 0.3164 |
| 50 | 4157.0782 | 0.001 | 10000 | 0.3281 |
| 100 | 8235.2708 | 0.0001 | 100000 | 3.3093 |
| 150 | 12279.1477 | 0.0001 | 100000 | 3.459 |
| 200 | 16235.0629 | 0.0001 | 100000 | 3.613 |

***Table 2. Running results for SLS with graph mean = 100, s.d. = 10***

****

**Figure 2. Detailed statistics of Table 2 on SLS**

## 3.6 Observation and Assessment:

The SLS (Simulated Annealing) algorithm performs well for the 10-city instance, surpassing the BnB DFS. The success in small instances suggests that SLS can effectively navigate simple solution spaces and converge to high-quality solutions within a reasonable runtime. However, for larger instances (25, 50, 100, 150, and 200 cities), the minimum cost obtained from the SLS algorithm is not as competitive as the BnB DFS algorithm. This suggests that while SLS excels in navigating simpler solution spaces, it encounters difficulties exploring the intricate solution space systematically in larger problems, potentially getting stuck in local minima. Due to this nature, SLS may serve as a more valuable tool to quickly find suboptimal solutions rather than optimal solutions.

Besides performance, the runtime observation aligns with our analysis of the time complexity of SLS, expressed as O(N+M), where N is the number of cities and M is the number of iterations. This correlation is reflected in our results, indicating that as the number of cities increases, both the runtime and the complexity of the problem escalate. Furthermore, the runtime is notably contingent on the number of iterations. This high dependence on iterations is anticipated, particularly when N is relatively smaller than M. In this way, the runtime of the SLS outperforms the BnB DFS greatly.

SLS’s number of iterations is influenced by parameters such as initial temperature, alpha (cooling rate), and max iteration. In our performance evaluation, we maintain max\_iteration > actual\_iteration and set initial\_temperature to 10 for simplicity. By changing the value of alpha, we can adjust the number of iterations as needed. However, the performance of the SLS algorithm can be further enhanced through careful parameter tuning. This aspect could be explored in future work.

# **4. Reference**

1. ‘Dynamic programming treatment of the traveling salesman problem’, Richard Bellman, Journal of Assoc. Computing Mach. 9. 1962.
2. [The Traveling Salesman Problem (TSP)](https://www2.seas.gwu.edu/~simhaweb/champalg/tsp/tsp.html).
3. [Branch & Bound (B&B) and Constraint Satisfaction Problems (CSPs)](https://www.cs.ubc.ca/~mack/CS322/lectures/3-CSP1.pdf).
4. [Branch bound, summary of advanced topics](https://www.cs.ubc.ca/~hutter/teaching/cpsc322/2-Search7.pdf).
5. Hoos, H.H., Stützle, T. (2015). Stochastic Local Search Algorithms: An Overview. In: Kacprzyk, J., Pedrycz, W. (eds) Springer Handbook of Computational Intelligence. Springer Handbooks. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-662-43505-2\_54
6. [Local Search Algorithm](https://canvas.eee.uci.edu/courses/58458/files/folder/Slides?preview=23910339).
7. [11 Animated Algorithms for the Traveling Salesman Problem.](https://stemlounge.com/animated-algorithms-for-the-traveling-salesman-problem/)
8. Sergey Smirnov, Vladimir Voloshinov. (2017) Implementation of Concurrent Parallelization of Branch-and-bound algorithm in Everest Distributed Environment. Procedia Computer Science. https://doi.org/10.1016/j.procs.2017.11.163.