Lin Kernighan Heuristic

Isabelle May
CPSC 450
Fall 2024

Summary

This project focuses on solving the Traveling Salesman Problem (TSP) using the Lin-Kernighan heuristic, a dynamic k-opt algorithm, and comparing it to the Bellman-Held-Karp Algorithm. Each algorithm was implemented in Java, utilizing graph-based data structures to represent nodes and edges. Performance tests included correctness validation on large graphs, comparisons of time efficiency, and testing on graphs of up to 2000 nodes. The Lin-Kernighan algorithm consistently outperformed the Bellman-Held-Karp Algorithm, achieving near-optimal solutions (within 1% of the optimal tour cost) and scaling significantly more efficiently for larger datasets.

1 ALGORITHM SELECTED

1.1 Problem and Algorithm

The problem addressed in this work is the Traveling Salesman Problem (TSP), a classic NP-hard optimization problem where the goal is to determine the shortest possible tour that visits every vertex in a graph exactly once and before returning to the starting point. The algorithm explored is the Lin-Kernighan heuristic, a dynamic k-opt algorithm. This heuristic iteratively improves an initial tour by breaking and reconnecting k edges to achieve shorter routes. The method dynamically determines the value of k based on the potential gain from further optimization, making it more adaptive in comparison to fixed-k algorithms, brute-force, and dynamic programming algorithms.

1.2 Sources

This project relied on three main sources. *Lin and Kernighan (1973)* introduced the dynamic k-opt method, which formed the foundation for implementing the Lin-Kernighan heuristic in my work [1]. *Idzenga (2023)* provided a comprehensive overview and a detailed analysis of TSP algorithms, offering valuable context for evaluating the performance of the Lin-Kernighan heuristic against other methods [2]. Additionally, *Keld Helsgaun's LKH-1.3 documentation* gives an in-depth explanation of the algorithm's structure and decision-making process, aiding in refining the implementation used in this project [3].

Additionally, *Hutchinson (2016)* and *Nguyen* provided detailed explanations of the Bellman-Held-Karp algorithm, emphasizing its dynamic programming approach and its role as a benchmark for solving TSP instances [5][6]. Further insights on both algorithms were gained from *Chen, Yang, and Li (2022)*, which explores the application of deep reinforcement learning to the TSP, introducing a modern approach that contrasts with traditional heuristic and exact methods [4]. Together, these sources informed both the theoretical understanding and practical implementations that shaped this project.

1.3 Lin Kernighan Pseudocode [2]:

Algorithm 1 Main loop
1: for t1 in vertices do
2: for t2 in neighbors of t1 do
3: broken ← [(t1, t2)]
4: added ← []
5: G∗ ← 0
6: k ← 0
7: if AddingEdge(broken, added, G∗, k) is "improved" then
8: restart MainLoop
9: end if
10: end for
11: end for

Algorithm 2 AddingEdge(broken, added, G∗, k)
1: i ← i + 1
2: t_{2i} ← last vertex in t
3: for t_{2i+1} in vertices do
4: if t_{2i+1} was already chosen or t_{2i+1} is in neighbors of t_{2i} then
5: continue
6: end if
7: y_i ← (t_{2i}, t_{2i+1})
8: added.append(y_i)
9: Calculate G_i
10: if G_i < G∗ and G∗ > 0 then
11: RelinkTour
12: return "improved"
13: else if G_i < G∗ and G∗ = 0 then
14: added.remove(y_i)
15: return "not improved"
16: end if
17:
18: if BreakingEdge(broken, added, G∗, k) is "improved" then
19: return "improved"
20: else if BreakingEdge(broken, added, G∗, k) is "no suitable edge" then
21: added.remove(y_i)
22: continue
23: else if BreakingEdge(broken, added, G∗, k) is "not improved" then
24: added.remove(y_i)
25: if backtracking is not allowed or max number of neighbors are considered then
26: return "not improved"
27: end if
28: end if
29: end for
30: return "not improved"

Algorithm 3 BreakingEdge(broken, added, G∗, k)
1: t_{2i+1} ← last vertex in t
2: xi = (t_{2i+1}, t_{2i+2}) ← suitable edge to break

3: if no suitable edge to break then
4: return "no suitable edge"
5: end if
6: broken.append(xi)
7: if $G_i-1 + |x_i| - |(t_{2i+2}, t_1)| > G*$ then
8: if greedy then
9: RelinkTour
10: return "improved"
11: end if
12: $G* \leftarrow G_i-1 + |x_i| - |(t_{2i+2}, t1)|$
13: $k \leftarrow i$
14: end if
15:
16: if max depth is reached then
17: if $G* > 0$ then
18: RelinkTour
19: return "improved"
20: else if $G* = 0$ then
21: broken.remove(x_i)
22: return "not improved"
23: end if
24: end if
25:
26: if AddingEdge(broken, added, G∗, k) is "improved" then
27: return "improved"
28: else if AddingEdge(broken, added, G∗, k) is "not improved" then
29: broken.remove(x_i)
30: return "not improved"
31: end if

1.4 Bellman-Held-Karp Pseudocode [6]:

1: function BellmanHeldKarp(G, V)
2: Initialize distance matrix dist[V][V] for all pairs of vertices
3: for each subset of vertices $S \subseteq V$ do
4: for each vertex $v \in S$ do
5: dist[S][v] ← minimum cost to reach vertex v from the subset S
6: end for
7: end for
8: for each subset of vertices $S \subseteq V$ do
9: for each vertex $v \in S$ do
10: for each vertex $u \in S \setminus \{v\}$ do
11: dist[S][v] ← min(dist[S][v], dist[S \ {v}][u] + cost(u, v))
12: end for
13: end for
14: end for
15: return dist[V][1] // Optimal solution: the shortest cycle distance
16: end function

1.5 Time Complexity

The Lin Kernighan Algorithm begins with an initialization step, where the initial random tour is constructed, taking O(V) time, where V represents the number of vertices. The main loop of the algorithm involves evaluating candidate edges for each vertex, performing dynamic k-opt moves, and optimizing through edge-breaking. The complexity per restart in this phase is $O(V^2 \cdot C \cdot D)$, where C is the number of candidate edges and D is the recursion depth. Assuming constant values for C and D, this complexity simplifies to $O(V^2)$. The recursive k-opt step explores all edges at the current depth, with a worst-case complexity of $O(V^2 \cdot D)$. Finally, the edge-breaking optimization step evaluates all edges, contributing $O(V^2)$ to the overall complexity [2].

Overall Complexity: $O(V^2 \cdot (C \cdot D+1)) \rightarrow O(V^2)$

The Bellman-Held-Karp algorithm starts with an initialization step, where the initial distance matrix is constructed, taking $O(V^2)$ time, where V is the number of vertices. The core of the algorithm involves dynamic programming, where subproblems are solved by recursively finding the shortest paths between subsets of vertices. In each recursive step, the algorithm evaluates all transitions between subsets, leading to a complexity of $O(V^2 \cdot 2^V)$, as it needs to compute distances for every subset of vertices and for each potential transition. The complexity is dominated by the number of subsets (2^V) and the number of vertices (V), making the Bellman-Held-Karp algorithm exponential in time complexity. [5]

Overall time complexity: $O(V^2 \cdot 2^V) \rightarrow O(2^V)$

2 IMPLEMENTATION

2.1 Setup

In my implementation of the Lin-Kernighan algorithm, I used Java as the programming language. The core data structures I utilized were ArrayLists, HashSets, and PriorityQueues, due to their ability to efficiently manage dynamic sets of objects such as tours, edges, and candidate edges. Overall, my implementation utilizes a dynamic and iterative refinement strategy.

Starting with a randomized tour, the algorithm progressively improves the solution by exploring local optimizations through k-opt moves. Each move involves breaking and reconnecting edges with the main goal of reducing the total tour cost. It uses a dynamic candidate edge selection process, utilizing a priority queue to identify promising edges based on weight. To avoid local minima, the algorithm incorporates global restarts and reinitializes the tour multiple times.

Recursive methods are then employed to explore deeper k-opt optimizations, with safeguards like gain thresholds and maximum recursion depths to balance performance and computational cost. Additionally, the algorithm includes edge-breaking optimization to refine solutions further. Combining iterative refinement, randomization, and systematic edge evaluations, I aim to balance computational efficiency with solution quality in my implementation to efficiently solve TSP instances, even if quite complex.

2.2 Deviations From Pseudocode

My implementation deviated from the pseudocode in several ways, particularly in terms of the structure and complexity of the recursive optimization process. In the pseudocode, the focus is on iteratively selecting candidate edges and applying k-opt moves, with a clear termination condition based on improvement thresholds or a maximum number of allowed moves. My final algorithm, however, introduces additional complexity with multiple layers of recursion, dynamic edge breaking, and optimization using priority queues for candidate selection. Furthermore, the pseudocode contains a simpler approach to tour improvement, while my algorithm incorporates more advanced techniques like caching edge weights and using sets for edge

management, making it more thorough and slightly more computationally expensive.

I also added specific termination conditions like recursion depth limits and gain thresholds for the dynamic k-opt search, as well as a backtracking mechanism with edge reversals for increased efficiency and better exploration of potential solutions. Additionally, I implemented edge-breaking optimizations and tour cost calculations to handle invalid or overflow edge weights.

2.3 External Libraries

The only external library used was the Java Collections Framework to handle dynamic data structures such as lists, sets, and queues.

2.4 Challenges

One of the most challenging aspects of the implementation was handling the recursive exploration of k-opt moves. This required careful management of recursion depth and edge reversals to prevent the algorithm from diving too deep into unpromising search paths while still exploring enough potential moves. Additionally, edge handling posed a significant challenge, as edge weights could potentially result in overflow or invalid values, such as NaN or infinite weights. To address this, I added error checks and validation in methods like calculateTourCost and calculateGain to ensure the algorithm wouldn't fail due to invalid data. Finally, performance optimization was necessary due to the growing complexity of the algorithm as the graph size increases. Optimizing the search for candidate edges and managing recursion depth were key strategies in mitigating performance concerns and improving the overall efficiency of the solution.

2.5 Unit Test Cases

2.5.1 Graph Data and Unit Test Setup

To ensure the correctness of the Lin-Kernighan algorithm implementation, 15 test cases were developed, each targeting specific aspects of the algorithm's behavior and performance. Below is a general overview of the test cases:

- Single Node Graph: Used to test the edge case of minimal input.
- Small Complete Graph: A 4-node complete graph to assess the behavior with multiple edges.
- Disconnected Graph: A graph with isolated components to test handling of disconnected graphs.
- Graph with Identical Weights: Used to evaluate the algorithm's behavior when all edges have the same weight.
- Graph with Negative Weights: Used to evaluate the algorithm's behavior when all edges have negative weights.
- Graph with Zero Weights: Used to evaluate the algorithm's behavior when all edges have a weight of 0.
- Moderate Size Graph: A 20-node cycle to simulate a more realistic, moderately sized graph.
- Large Graphs: Up to 100 nodes, testing how the algorithm scales with increasing size.
- Correctness: Verified that the algorithm computes the correct tour and compares it to a brute-force solution for small graphs.
- Performance: Evaluated the execution time for both small and large graphs. A special test compared the Lin-Kernighan algorithm's time against a Bellman-Held-Karp Algorithm to measure efficiency.

Table 1: Overview of Test Cases

Test Case	**Purpose**	**Graph Type**	**Expected Outcome**
Basic Graphs	Tests for graphs with small, simple configurations (1-2 nodes)	Single node, two nodes	Correct vertex/edge count, edge presence validation
Graph Structure (Complete /Disconnected)	Tests for complete and disconnected graphs	Complete, disconnected	Edge count matches graph structure, disconnected vertices have no edges
Edge Weight Variations	Tests for graphs with varying edge weights, including negative, zero, and identical weights	Varying weights	Correct edge weights applied, results consistent for differing weights
Moderate to Large Graphs	Tests larger graphs with up to 50 vertices, ensuring scalability	Moderate (20 nodes) to large (50 nodes)	Correct vertex/edge count, results consistent with graph structure
Path Accuracy: Optimal vs. Lin-Kernighan	Tests for path correctness, comparing Lin-Kernighan algorithm with brute force solution	TSP graphs (10-15 nodes)	Lin-Kernighan's result within tolerance of the optimal solution
Time Efficiency	Tests large graphs to assess time efficiency and algorithm performance	Large graphs (100 nodes)	Algorithm completes within time limits, results consistent with expected efficiency

2.5.2 *Scaling the Graph Data:*

- For the large graphs I ensured that the graph was fully connected with edges added to form a cyclic structure.
- Edge weights were assigned using random values or fixed values to simulate different scenarios, such as graphs with identical, negative, or zero weights.

3 PERFORMANCE TESTS

3.1 *Performance Testing Setup*

To evaluate the performance of the Lin-Kernighan algorithm, I created a test using graphs of varying sizes and complexities. This test involved creating both sparse and dense graphs, ranging from 200 to 2000 vertices. The second performance test compares the Lin-Kernighan and Bellman-Held-Karp algorithms on various sparse and dense graphs, ranging from 1 to 20 vertices, with their runtimes measured in milliseconds.

3.2 *Graph Data and Test Setup*

The performance tests aim to evaluate the performance of two algorithms—Lin-Kernighan and Bellman-Held-Karp—on both sparse and dense graphs. The program creates 2 different graphs with varying vertex counts, ranging from 1 to 2000, and uses two types of graph structures: sparse adjacency lists, where nodes are sparsely connected, and dense adjacency lists, where every pair of nodes is connected. Both graphs label edges with incremental weights to ensure consistency.

The first evaluation graph is set up to test the Lin-Kernighan algorithm on both sparse and dense adjacency lists ranging from 200-2000 vertices. The second evaluation graph is set up to test both algorithms by measuring their execution time and plotting them next to each other. This test is built to test sparse adjacency lists ranging from 1-20 vertices. Though 20 vertices might seem low compared to the first test, it is intentionally chosen to account for the computational complexity of the Bellman-Held-Karp algorithm, which becomes impractical for larger graphs due to its exponential time complexity [5][6]. Limiting the vertex count to 20 ensures the evaluation remains feasible while still providing meaningful insights into the algorithm's performance.

For both graphs, results are logged for each graph configuration and visualized using charts generated by the JFreeChart library. The program then saves the performance charts as PNG files for further analysis.

3.2.1 *Graph Structures*

Two primary types of graph structures are used to test the algorithms: sparse and dense adjacency lists. Sparse graphs are created by connecting vertices with a limited number of edges, ensuring a low edge density, while dense graphs involve connecting every possible pair of vertices, resulting in a high edge density. These structures are used to assess the performance of the Lin-Kernighan to evaluate its computational efficiency on varying levels of graph complexity. In contrast, the Bellman-Held-Karp algorithm is evaluated only on sparse graphs due to its exponential time complexity, which makes it infeasible for dense graphs with larger vertex counts.

3.3 *Environment*

Hardware: The tests were executed on a machine with the following specifications:

Processor: Intel i7-9700K

RAM: 16 GB DDR4

Disk: SSD storage

Operating System: Windows 10 (64-bit)

Java Version: OpenJDK 11

3.4 *Building and Running the Tests*

3.4.1 *Project Setup:*

- The tests were implemented using JUnit 5 for unit testing, ensuring the correctness of the algorithm.
- Maven was used to manage dependencies. The required dependencies were:
  - JUnit 5
  - Java Collections API
  - Custom classes for the graph representation (Graph, AdjList, EdgeLabeling)

3.4.2 *Instructions:*

1. Clone the Repository:
   git clone https://github.com/Gonzaga-CPSC-450-Fall-2024/final-project-ifmay.git
2. Run the Unit Tests: mvn test
3. Run the Performance Evaluation Tests:
   mvn compile exec:java

4 EVALUATION RESULTS

This section analyzes the tests results from both the conducted performance tests and the unit tests. The results highlight the Lin-Kernighan heuristic's efficiency, versatility, and accuracy in comparison to the Bellman-Held-Karp algorithm. Further, the tests demonstrate the heuristic's ability to find near-optimal solutions significantly faster than exact methods, like the Bellman-Held-Karp algorithm/

4.1.1 *Performance Test Results:*

Figure 1 depicts Lin-Kernighan algorithm's performance across various sized sparse and dense graphs ranging from 200-2000 vertices. Both sparse and dense graphs performed similarly with an increasing computational cost with the number of vertices. For instance, both sparse and dense graphs with 2000 vertices require approximately 4500 milliseconds to run, indicating that the difference in processing time between the two graph types is not significant at this scale. This consistency highlights the algorithm’s computational efficiency and scalability in handling varying graph densities without substantial deviations in execution time. These results suggest that the Lin-Kernighan heuristic effectively maintains its efficiency regardless of edge density, making it a reliable choice for solving TSP on both sparse and dense graphs within the tested range.

Figure 2 illustrates the performance of a Bellman-Held-Karp algorithm and the Lin-Kernighan Algorithm on sparse graphs. For smaller-scale TSP instances, both algorithms exhibit similar performance, with execution times remaining relatively low. However, as the number of vertices increases for the Bellman-Held-Karp algorithm, the time required for the algorithm to find the optimal solution grows exponentially. This evaluation

suggests that the Bellman-Held-Karp algorithm is impractical for larger-scale instances due to its exponential performance deterioration.

In contrast, the Lin-Kernighan algorithm maintains a more stable performance profile, providing good approximate solutions in a reasonable amount of time. This growth demonstrates the computational challenge of solving large-scale TSP instances using exact methods. As the number of vertices increases for the Lin-Kernighan Algorithm, the time required for the algorithm to find a near-optimal solution grows polynomially, making it a practical choice for large-scale TSP instances.

The Lin-Kernighan Algorithm handles larger sizes efficiently (up to 387,700% [1ms vs. 3878ms] for 20 vertex graphs) but since it is a heuristic, it may not always find the precise optimal solution. Additional testing was performed in the unit tests to ensure that accuracy was not sacrificed for efficiency in my implementation.

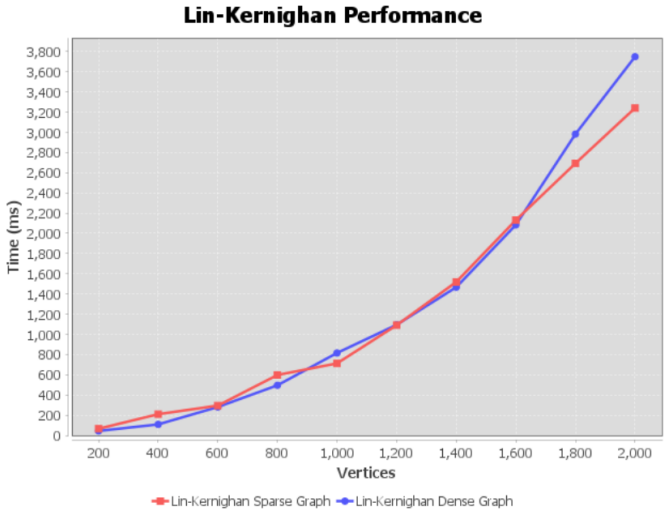

Figure 1: Lin-Kernighan Performance Graph

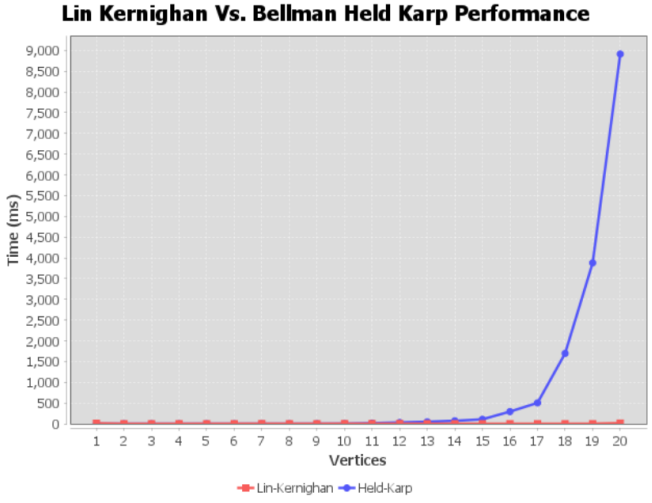

Figure 2: Algorithm Performance Comparison Graph

4.2 Unit Test Results:

The unit tests provide convincing evidence of the Lin-Kernighan heuristic's effectiveness in solving the Traveling Salesman Problem (TSP). The algorithm demonstrates both accuracy and efficiency across various graph sizes and edge weight configurations.

4.2.1 Accuracy

The testCorrectnessOnGraph test specifically highlights the heuristic's ability to find near-optimal solutions. In this test, the Lin-Kernighan algorithm consistently calculates tour costs within 1% of the optimal solution, often achieving the exact optimal solution.

4.2.2 Efficiency

The Lin-Kernighan heuristic exhibits polynomial time complexity, making it significantly more scalable than exact algorithms. For instance, the testVsExactTimeEfficiency test demonstrates that for a 15-node graph, the Lin-Kernighan algorithm completes execution in just 0 milliseconds, while the Bellman-Held-Karp algorithm requires 42 milliseconds. This significant performance difference becomes even more pronounced for larger graphs. The testTimeEfficiencyForLargeGraph test showcases the heuristic's ability to handle a 100-node graph in under 34 milliseconds.

4.2.3 Versatility

The unit tests cover a wide range of graph scenarios, including disconnected graphs (testDisconnectedGraph), graphs with identical, negative, or zero edge weights (testGraphWithIdenticalWeights, testGraphWithNegativeWeights, testGraphWithZeroWeights), and graphs with multiple edges between nodes (testMultipleEdgesWithVaryingWeights). This demonstrates the heuristic's functionality and adaptability to various unique problem instances.

Overall, the unit tests demonstrate Lin-Kernighan's ability to offer a practical and efficient solution for the Traveling Salesman Problem. Its ability to find high-quality solutions in a fraction of the time required by exact methods, combined with its versatility in handling various graph structures, proves its value as a tool for real-world applications.

5 REFLECTION

For this project, I focused on implementing and analyzing the performance of the Lin-Kernighan heuristic and Bellman-Held-Karp algorithm on different graph structures and in comparison to one another. I found the Lin-Kernighan algorithm particularly interesting because of its iterative nature and how it attempts to improve upon an initial solution by exploring local optimal solutions. This ties into class discussions about NP-completeness and time complexity vs. efficiency tradeoffs.

One of the challenges I faced while designing the Lin-Kernighan algorithm was optimizing the recursive k-opt approach. The complexity of the algorithm increases due to the multiple layers of recursion needed to explore different tour improvements. Ensuring that the recursion depth was both effective and efficient required extensive thought, as excessive recursion could quickly lead to performance deterioration.

Another challenge was Balancing the exploration of potential edge swaps with the depth limits was crucial for preventing over-exploration while still achieving meaningful improvements.

Additionally, caching edge weights for efficiency and managing the multiple restarts of the algorithm presented challenges in maintaining the best tour across iterations, especially when new tours were generated through randomization. The difficulty of fine-tuning parameters, such as the maximum recursion depth, candidate edge selection, and stopping conditions, added challenges to the design process, and reinforced the importance of carefully considering the trade-offs between exploration and performance for the Lin Kernighan Algorithm.

If I had more time, I would experiment with optimizing the algorithm further by parallelizing its steps to improve efficiency for larger datasets. I would also test it on a broader range of problem instances and explore utilizing machine learning techniques to predict better edge swaps [6]. Additionally, I would fine-tune parameters and implement more advanced visualizations to track the algorithm's progress and identify areas for further improvement.

6 RESOURCES

This project relied on foundational research and contemporary innovations to develop and analyze algorithms for solving the Traveling Salesman Problem (TSP). Lin and Kernighan's 1973 paper [1] introduced the Lin-Kernighan heuristic, which served as the theoretical foundation for the algorithm's design. The edge-swapping approach detailed in the paper directly influenced the algorithm's implementation. Idzenga's 2023 comparative study [2] provided valuable insights into the performance of various TSP algorithms, guiding the evaluation phase, and enabling a thorough comparison with an already established TSP algorithm.

Helsgaun's documentation [3] offered practical guidance for implementing my Lin-Kernighan heuristic, focusing on optimization techniques and common pitfalls, which were beneficial during testing and debugging. Additionally, Yang's IEEE publication [4] introduced deep reinforcement learning methods for TSP, offering a broader perspective on how AI-driven approaches compare to traditional heuristics. While this source did not directly inform my work, it provided me with a greater appreciation for the potential of machine learning techniques in solving TSP and expanded my understanding of alternative approaches to the problem. These resources collectively informed the development, implementation, and evaluation of the TSP algorithm.

Hutchinson et al. (2016) [5] and Nguyen [6] provided comprehensive overviews of the Bellman-Held-Karp algorithm, detailing its theoretical basis, implementation, and limitations. These resources were helpful in understanding the computational constraints of the Bellman-Held-Karp approach, particularly when evaluating its performance on sparse graphs.

7 REFERENCES

[1] *Lin, S., & Kernighan, B. W. (1973). An Effective Heuristic Algorithm for the Traveling-Salesman Problem.* Retrieved from Princeton University

[2] *Idzenga, R. (2023). "A Comparative Study of Algorithms for Solving the Traveling Salesman Problem". Bachelor's Thesis, University of Twente.* Retrieved from University of Twente.

[3] *Keld Helsgaun, "The Lin-Kernighan Heuristic for Solving the Traveling Salesman Problem", LKH-1.3 Documentation.* Retrieved From Roskilde Universitet.

[4] *Chen, X., Yang, L., & Li, Z. (2022). "Deep Reinforcement Learning for Solving the Traveling Salesman Problem". IEEE Transactions on Artificial Intelligence.* Retrieved from IEEE.

[5] Hutchinson, C., Pyo, J., Zhang, L., & Zhou, J. (2016). *Traveling Salesman Problem and Bellman-Held-Karp Algorithm*. Carnegie Mellon University. Retrieved from Carnegie Mellon University.

[6] Nguyen, Q. N. *Traveling Salesman Problem and Bellman-Held-Karp Algorithm*. Retrieved from Nagoya University.