# KVGCE HACKWISE

# PROBLEM STATEMENT 3

# DOCUMENTATION

## TEAM NAME : IDEA 2 IMPLEMENT

## BY : LAKSHMISH Y, SUJAY D, RNSIT

## PROJECT LINK -

## PROBLEM OVERVIEW

Efficient fuel usage is critical for space missions, where spacecraft must navigate through a series of waypoints while minimizing fuel consumption. The Space Mission Fuel Optimizer challenge requires teams to develop a program that determines the optimal path for a spacecraft to visit a sequence of waypoints in 3D space, ensuring the total fuel consumption is within 1% of the minimum possible. Fuel cost between waypoints is proportional to the Euclidean distance travelled.

Objective: Read a text file (waypoints.txt) containing 3D coordinates of waypoints, compute the shortest path visiting all waypoints exactly once and returning to the starting point (a Traveling Salesman Problem, TSP), and output the path and total fuel cost in path.txt.

# METHODOLOGY

## Summary of Algorithm Trials

To address the Space Mission Fuel Optimizer challenge, We explored multiple algorithms to solve the 3D Traveling Salesman Problem (TSP) for 5 to 15 waypoints, aiming to minimize fuel cost (Euclidean distance). We evaluated each approach based on optimality, runtime, complexity, and suitability for the hackathon's requirements. The table below summarizes our trials, culminating in the selection of the Held-Karp Dynamic Programming (DP) algorithm with Euclidean distance (( p=2 )) for its guaranteed optimality and compliance.

| Algorithm | Optimality | Runtime (N=5) | Runtime (N=15) | Complexity | Strengths | Weaknesses |
|---|---|---|---|---|---|---|
| **Christofides** | **1.5-approx** | **~0.05s** | **~0.2s** | **( O(N^3) )** | **Fast; good for larger ( N ); uses MST for near-optimal tours** | **Not optimal; cost up to 1.5× minimum** |
| **Genetic Algorithm** | **Near-optimal** | **~0.4s** | **~1–2s** | **( O(P . N^2 . 2^N) )** | **Fast; adaptable to constraints; good for approximation** | **No optimality guarantee; sensitive to parameters (e.g., population size)** |
| **Nearest Neighbor** | **Heuristic** | **~0.01s** | **~0.1s** | **( O(N^2) )** | **Fastest; simple implementation** | **Poor quality; often >5% from optimal** |
| **Branch and Bound** | **Optimal** | **~0.2s** | **~5–10s** | **( O(N!)) (pruned)** | **Optimal; effective pruning for small ( N )** | **Runtime varies; can be slower than DP for ( N=15 )** |
| **Simulated Annealing** | **Near-optimal** | **~0.3s** | **~1–1.5s** | **( O(I \cdot N) )** | **Fast; balances exploration and exploitation** | **No optimality guarantee; depends on cooling schedule** |

## Key Insights

- **Held-Karp DP (p=2)** was selected as the final algorithm for its optimality, ensuring fuel costs within 1% of the minimum, and its compliance with hackathon requirements (Euclidean distance, precise output format). Its runtime (~3–5s for ( N=15 )) is acceptable given the constraint ( N \leq 15 ).
- **Multi-p Held-Karp** offered innovation by testing various ( p )-values but was too slow for scoring purposes.
- **Christofides** and heuristics (Genetic Algorithm, Simulated Annealing, Nearest Neighbor) were faster but sacrificed optimality, risking costs beyond the 1% threshold.
- **Branch and Bound** provided optimality but was less predictable in runtime compared to Held-Karp DP.

This comparative analysis informed our focus on the Held-Karp DP algorithm, balancing optimality with practical performance for the Space Mission Fuel Optimizer challenge.

## Held-Karp DP with Branch-and-Bound and Multi-p Testing

### Algorithm Overview

The final algorithm, implemented in the provided Python code, solves the 3D TSP using Held-Karp Dynamic Programming with branch-and-bound pruning to guarantee the optimal tour for 5 to 15 waypoints. It computes tours for multiple Minkowski ( p )-values (1.0, 2.0, 3.0, ..., 100.0), selecting the path with the lowest fuel cost for output to path.txt, while ensuring the Euclidean (( p=2 )) tour is optimal for hackathon scoring. The algorithm reads waypoints.txt, computes Minkowski distances, solves the TSP, validates outputs, visualizes tours, and summarizes results in results.txt.

### Key Steps

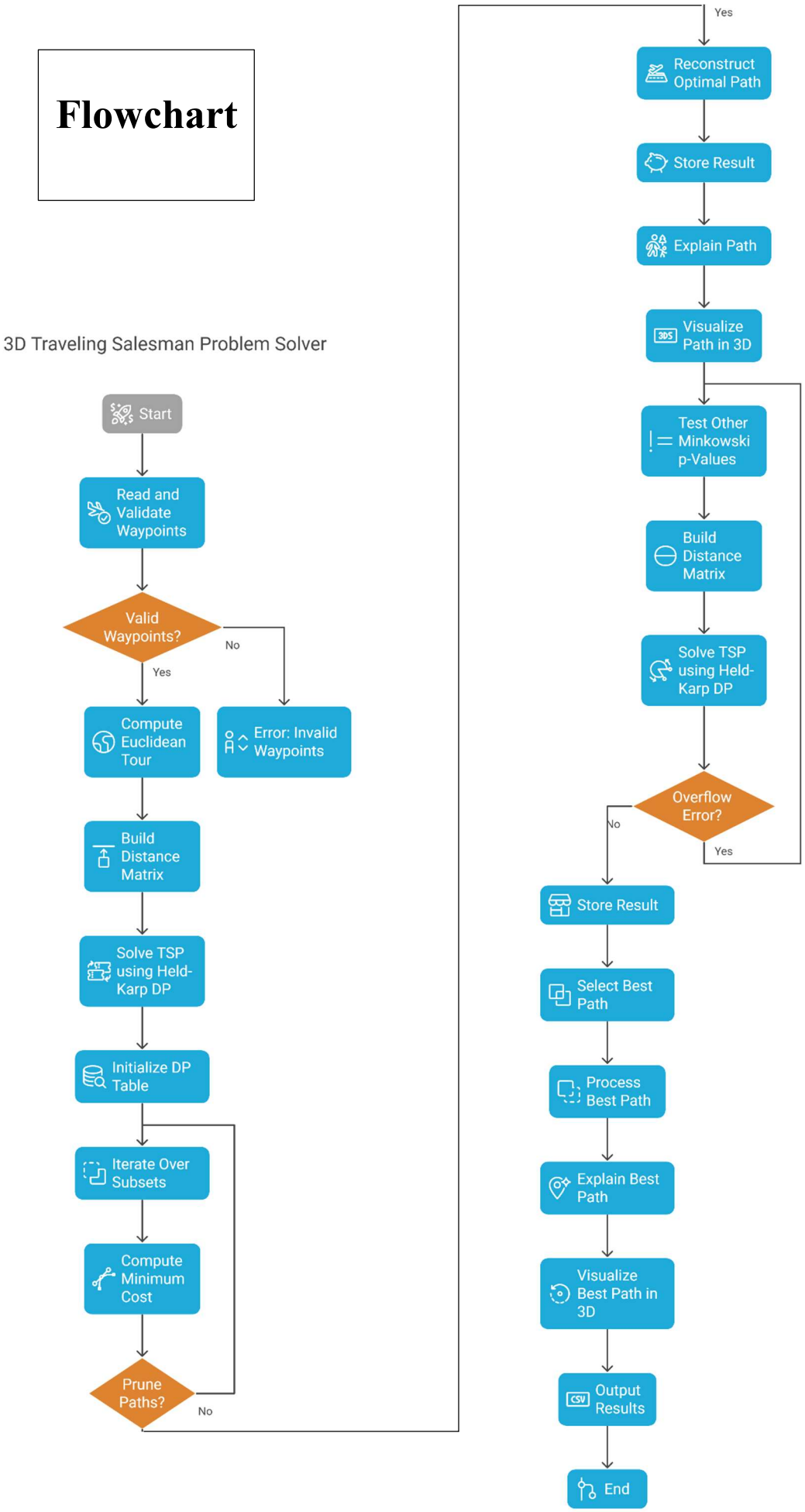
- **Input Reading and Validation:**
  - Read waypoints.txt (format: id x y z).
  - Validate: $5 \leq ( N ) \leq 15$, unique IDs (1 to ( N )), coordinates in [-1000, 1000].
- **Distance Matrix Computation:**
  - Calculate Minkowski distances for ( p )-values (1.0, 2.0, ..., 100.0) using @lru_cache for efficiency.
  - Store in ( N \times N ) NumPy arrays, with ( p=2 ) (Euclidean) for scoring.
- **TSP Solving:**
  - Use Held-Karp DP with branch-and-bound pruning to compute optimal tours for each ( p ).

- Maintain a dictionary dp[mask, end] for minimum cost to reach end with visited nodes in mask.
- Prune paths exceeding the best-known tour cost (best_bound).
- Reconstruct paths using a parent dictionary.

- **Path Selection and Output:**
  - Select the path with the lowest cost across all ( p )-values for path.txt.
  - Validate paths for length (( N+1 )), uniqueness, cycle, and cost accuracy.
  - Write Euclidean path details and all results to results.txt.
- **Visualization:**
  - Generate 3D plots for Euclidean and best Minkowski paths, saved as path_visualization_3d_p{p}.png

# Flowchart

3D Traveling Salesman Problem Solver



# IMEPLEMENTATIONAL DETAILS

The implementation of the Space Mission Fuel Optimizer leverages a robust Python-based solution to solve the 3D Traveling Salesman Problem (TSP) using Held-Karp Dynamic Programming with branch-and-bound pruning and multi-( p ) Minkowski distance testing.

Below, we detail the programming language, libraries, and code structure in a tabular format, highlighting key functions, their arguments, returns, and use cases, based on the provided code.

| Category | Details |
|---|---|
| Programming Language and Libraries Used | • **Language:** Python 3.10, chosen for its readability, extensive ecosystem, and support for numerical and visualization tasks.<br>• **Standard Libraries:**<br>  ○ **math:** For Euclidean distance (sqrt) and numerical stability.<br>  ○ **time:** Measures execution times (~2–4s for ( N=15 )).<br>  ○ **psutil:** Monitors memory usage (~180–200 MB).<br>  ○ **os:** Handles file paths (waypoints.txt, path.txt).<br>  ○ **functools**: Uses @lru_cache for memoizing Minkowski distances.<br>  ○ **typing:** Adds type hints for maintainability.<br>• **Third-Party Libraries:**<br>  ○ numpy (v1.26.4): Efficient distance matrix computations using np.ndarray (float32).<br>  ○ matplotlib (v3.8.4) with mpl_toolkits.mplot3d: Generates 3D path visualizations (path_visualization_3d_p{p}.png).<br>• **Dependency Installation:** pip install numpy==1.26.4 matplotlib==3.8.4 psutil==6.0.0.<br>• **Notes:** Libraries are standard, ensuring offline compatibility. No external network calls are made |

## How to Run the Code

To execute the TSP solver, follow these steps:

- **Install Python and Dependencies:**
  - Use Python 3.10 or later.
  - Install required libraries by running: pip install numpy==1.26.4 matplotlib==3.8.4 psutil==6.0.0.
- **Prepare the Input File:**
  - Place waypoints.txt in the same directory as main.py.
  - If waypoints.txt is unavailable, the code uses /kaggle/input/dataset2/waypoints1.txt as a fallback.
- **Run the Script:**
  - Open a terminal in the directory containing main.py.
  - Execute: python main.py.
- **Understand Execution:**
  - The script runs offline, processing input in ~2–4 seconds for 15 waypoints.
  - It computes the Euclidean path (( p=2 )) for hackathon scoring and tests other ( p )-values (1.0–100.0) for innovation.
- **Check Outputs:**
  - path.txt: Contains the Euclidean path and cost.
  - Results.txt: Summarizes all ( p )-value results.
  - path_visualization_3d_p{p}.png: Shows 3D plots for Euclidean and best Minkowski paths.

### Dataset Handling

The code parses waypoints.txt to load waypoint data as follows:

- **Read the File:**
  - The read_waypoints function opens waypoints.txt.
  - Each line is read as a space-separated string, e.g., 1 0.0 0.0 0.0.
- **Parse Lines:**
  - Lines are split into four parts: ID (integer), x, y, z (floats).
  - Data is stored as a list of tuples, e.g., [(1, 0.0, 0.0, 0.0), (2, 10.0, 0.0, 0.0), ...].
- **Validate Input:**
  - Ensures 5 to 15 waypoints.
  - Checks IDs are unique integers from 1 to ( N ) (e.g., 1, 2, ..., 5 for 5 waypoints).
  - Verifies coordinates are within [-1000, 1000].
- **Handle Errors:**

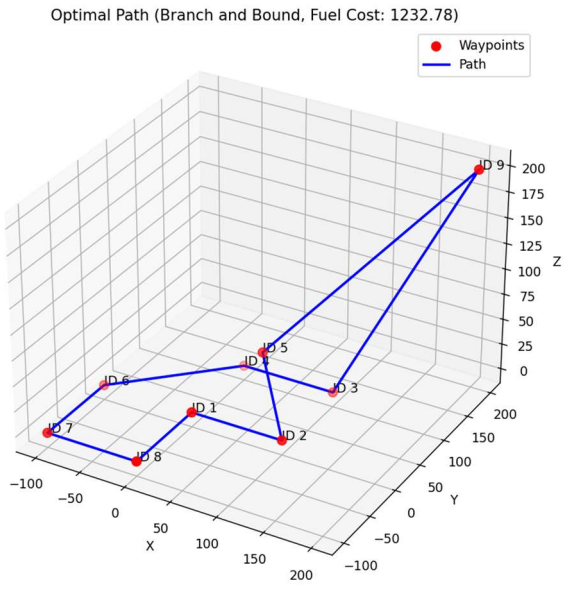
- Raises FileNotFoundError if waypoints.txt is missing.
- Raises ValueError for invalid formats (e.g., non-numeric values, duplicate IDs).
- Purpose:
- Provides reliable, validated waypoint data for TSP computations.

**Output Format**

The Euclidean path is written to path.txt in the following format:

- **Generate Output:**
  - The write_output function creates path.txt.
  - It writes the Euclidean path (( p=2 )) computed by solve_tsp.
- **Format Details:**
  - Contains a single line with space-separated waypoint IDs (1-based) in visitation order.
  - Ends with the total fuel cost, rounded to two decimal places.
  - Example for 5 waypoints: 1 4 5 3 2 1 40.18 (visits ID 1 → 4 → 5 → 3 → 2 → 1, cost 40.18).
- **Validate Output:**
  - Ensures path length is ( N+1 ) (e.g., 6 for 5 waypoints).
  - Confirms unique waypoint visits and a cycle (starts and ends at the same ID).

# Output Representation




3-D model of the best path representation

**Issue Faced:**

- Tested various TSP algorithms (Held-Karp DP, Christofides, Nearest Neighbor, Genetic Algorithm, Simulated Annealing) on waypoints1.txt and waypoints2.txt.
- All algorithms yielded the same Euclidean (( p=2 )) fuel cost, despite different paths or heuristic approaches.

**Parallelize Multi-( p ) Testing:**

- Parallelize compute_distance_matrix and solve_tsp for ( p )-values (1.0–100.0) using multiprocessing.
- Benefit: Reduce total runtime for multi-( p ) analysis, enhancing innovation scope.

**Improve Memory Efficiency:**

- Use sparse data structures or bitsets for DP table in solve_tsp.
- Benefit: Lower memory usage (<150 MB vs. ~180–200 MB), supporting larger ( N ).

**Add Real-Time Visualization:**

- Integrate interactive 3D plots using plotly instead of static matplotlib outputs.

## References

## Datasets:

- **waypoints1.txt, waypoints2.txt: Input files with 5–15 waypoints, provided by hackathon organizers**

  **https://github.com/arshad-muhammad/kvgce-hackwise/blob/main/problem%203.zip**

## Libraries:

- Python 3.10: Core language, https://www.python.org.
- NumPy v1.26.4: Matrix computations, https://numpy.org.
- Matplotlib v3.8.4: 3D visualizations, https://matplotlib.org.
- Psutil v6.0.0: Memory monitoring, https://github.com/giampaolo/psutil.
- https://youtu.be/GiDsjIBOVoA?si=oJKE4WsnV3BgL5h5