Space Mission Fuel Optimizer

Overview Efficient fuel management is essential for space missions, where spacecraft must travel through multiple waypoints while minimizing fuel usage. The Space Mission Fuel Optimizer challenge tasks participants with creating a program that finds the most fuel-efficient route for a spacecraft to visit all given waypoints in 3D space. The total fuel cost must be within 1% of the absolute minimum possible, and is directly related to the Euclidean distance between points.

Objective : Read a file named waypoints.txt containing 3D coordinates of waypoints. Calculate the shortest possible round-trip route that visits each waypoint exactly once (solving a 3D Traveling Salesman Problem). Output both the optimal path and total fuel cost to a file named path.txt.

Methodology:

Algorithm Steps:

1. **Read input waypoints from file**
2. **Compute distance matrix**
3. **Initialize root node with starting city**
4. **Use priority queue for best-first search**
5. **Expand nodes (branch) and compute bound (lower cost estimate)**
6. **Continue until full tour with minimum cost is found**
7. **Write the optimal path and total cost to a file**

Pseudocode:

```
Function BranchAndBoundTSP():
   Read waypoints from file and store coordinates
   Create distance matrix using 3D Euclidean distance

   Initialize:
      Root path = [0]  // Start from city 0
      Visited = [True, False, False, ...]
      Bound = CalculateBound(path, visited)
      PriorityQueue ← [Node(path, cost=0, bound, visited)]

   While PriorityQueue is not empty:
      Node ← Extract node with lowest bound

      If Node contains all cities:
         FinalCost = Cost to complete tour
         If FinalCost < MinCost:
            Update MinCost and BestPath

      Else:
         For each unvisited city:
            Create new path with that city
            Update visited array
            Calculate cost and bound
            If bound < MinCost:
               Add new node to PriorityQueue

   Return BestPath and MinCost


Function CalculateBound(path, visited):
   Bound = Sum of distances in current partial path
   For each unvisited city i:
      Find minimum edge (i → j) where j is also unvisited
      Add that min edge to Bound
   Return Bound
```

Implementation Details:

Programming Languages and Library Used:

Python

- Matplotlib
- Math
- Heapq

Data set Handling:

1. waypoints = []

•Initializes an **empty list** to store each waypoint as a tuple.

2. with open("waypoints_test.txt") as f:

•Opens the file waypoints_test.txt in **read mode**.

•The with statement ensures the file is closed properly after reading.

3. for line in f:

•Iterates **line by line** through the file

4. line.strip().split()

•strip() removes any **leading/trailing whitespace or newline characters**.

•split() splits the line into separate values using **whitespace** as the delimiter.

•Example: "1 10.5 20.3 5.0" → ["1", "10.5", "20.3", "5.0"]

5. map(float, ...)

- Converts **all split values** to floats:
  - "1" → 1.0
  - "10.5" → 10.5
  - etc.

6. id, x, y, z = map(float, ...)

- Unpacks the 4 values into variables: id, x, y, and z as floats.

7. int(id)

- Converts the id value from float **back to int**, assuming it's a whole number (e.g., 1.0 –

8. waypoints.append((int(id), x, y, z))

- Adds the waypoint as a **tuple** (id, x, y, z) to the waypoints list.

OUTPUT FORMAT:

The output generated after solving the path optimization problem (such as the Traveling Salesman Problem) is a single line containing a sequence of waypoint IDs followed by the total path cost.

[ID_1 ID_2 ID_3 ... ID_n ID_1 TOTAL_COST]

ID_1 to ID_n: Integer values representing the ordered sequence of waypoints visited.

- **ID_1 (repeated at end):** The path returns to the starting point to complete the cycle.
- **TOTAL_COST (float):** The cumulative cost of the path, such as total distance or fuel used.

Challenges and Solution:

Challenge 1: Exact Solution is Computationally Expensive

•The **Traveling Salesman Problem (TSP)** is NP-hard, meaning that finding the exact shortest path becomes exponentially harder as the number of waypoints increases.

•Even with Branch and Bound, the solution space can be huge, especially when the number of nodes grows beyond 15–20.

Solution:

We implemented the **Branch and Bound algorithm** to systematically explore promising paths while pruning suboptimal ones. This reduced the number of paths evaluated and helped find the optimal solution more efficiently than brute force.

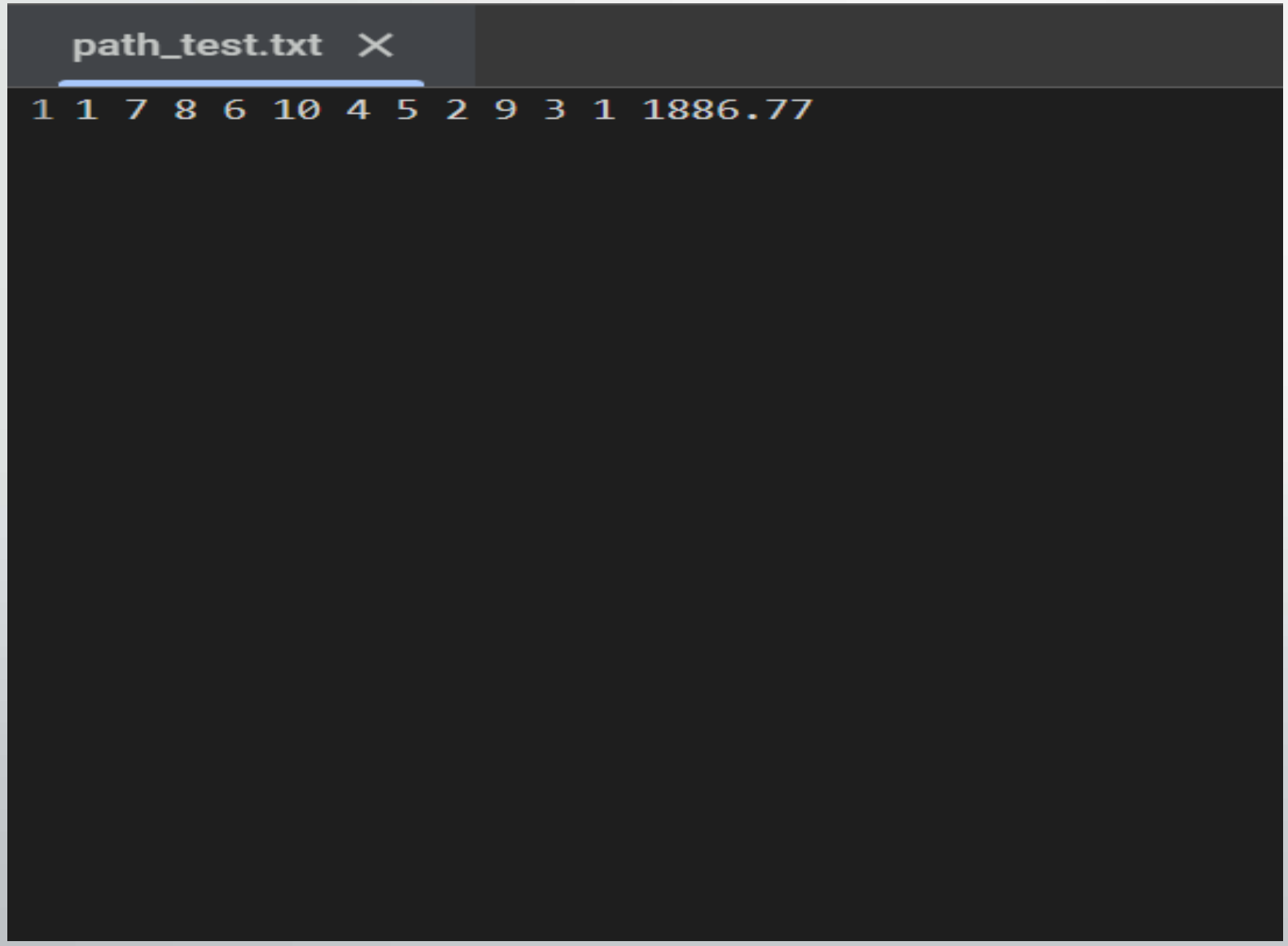
Challenge 3: Input Parsing and Path Reconstruction

•Reading raw waypoint data from a file and ensuring it's correctly fed into the algorithm (especially the distance matrix) can introduce bugs or inconsistencies.

Solution:

We used a clean parsing method to convert .txt files into a structured format (lists/tuples). The output path is reconstructed from the stored traversal and written in a compact format:

Test Case Result:



Reference:

- The Traveling Sales Man problem : when good enough beats perfect by Reducible.
- The traveling sales man series by Eddie woo.
- Traveling Sales Man Problem visualization by n sanity.

Libraries used:

- math
- hepq