Space Mission Fuel Optimizer

KVGCE Hackwise Hackathon - Problem 3

Team Name: TEEN TITANS

Team ID: 18

April 26, 2025

Contents

1 Introduction

1.1 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 addresses this problem by developing a software solution to determine the optimal path for spacecraft navigation.

The problem is essentially a variant of the Traveling Salesman Problem (TSP) in three-dimensional space, where:

- A spacecraft must visit a sequence of waypoints exactly once.
- The spacecraft must return to the starting point after visiting all waypoints.
- The fuel consumption is directly proportional to the Euclidean distance traveled in 3D space.
- The goal is to find a path that minimizes the total fuel consumption.

1.2 Objectives

The primary objectives of this project are:

- To read a set of 3D waypoints from an input file (waypoints.txt).
- To compute the optimal or near-optimal path that visits all waypoints exactly once and returns to the starting point.
- To ensure the total fuel consumption is within 1% of the minimum possible.
- To output the optimized path and total fuel cost to an output file (path.txt).
- To develop a web-based visualization interface for the optimized path.

2 Methodology

2.1 Algorithm Overview

To solve the Space Mission Fuel Optimizer problem, we implemented a hybrid approach combining the Nearest Neighbor algorithm for initial path construction and the 2-opt improvement algorithm for path optimization.

Algorithm Approach

Our solution follows a two-phase approach:

1. **Initial Solution Generation**: Using the Nearest Neighbor heuristic to quickly construct a feasible path.
2. **Solution Improvement**: Applying the 2-opt algorithm to refine the initial path and reduce the total fuel cost.

2.2 Distance Calculation

For calculating the distance (fuel cost) between any two waypoints in 3D space, we use the Euclidean distance formula:

$$d(p_1, p_2) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2} \tag{1}$$

where $p_1 = (x_1, y_1, z_1)$ and $p_2 = (x_2, y_2, z_2)$ are the 3D coordinates of two waypoints.

2.3 Nearest Neighbor Algorithm

The Nearest Neighbor algorithm is a greedy approach that starts at a given waypoint and repeatedly selects the nearest unvisited waypoint until all waypoints have been visited. The algorithm then returns to the starting point.

Algorithm 1 Nearest Neighbor Algorithm

procedure NearestNeighbor($distanceMatrix$)
  $n \leftarrow$ length of $distanceMatrix$
  $unvisited \leftarrow \{1, 2, \ldots, n-1\}$ ▷ Set of unvisited waypoints
  $path \leftarrow [0]$ ▷ Start with the first waypoint
  $current \leftarrow 0$ ▷ Current position is the first waypoint
  while $unvisited$ is not empty **do**
    $nextCity \leftarrow$ waypoint in $unvisited$ with minimum $distanceMatrix[current][j]$
    Add $nextCity$ to $path$
    Remove $nextCity$ from $unvisited$
    $current \leftarrow nextCity$
  end while
  Add 0 to $path$ ▷ Return to the starting point
  return $path$
end procedure

2.4 2-opt Improvement Algorithm

The 2-opt algorithm improves an existing path by repeatedly removing two edges from the path and reconnecting the resulting fragments in a different way to reduce the total distance.

2.5 Code Structure

The project is organized into several key files:

Algorithm 2 2-opt Improvement Algorithm

```
procedure TwoOpt(path, matrix, maxIter)
    best ← path                                         ▷ Copy of the initial path
    bestCost ← CalculatePathCost(best, matrix)
    improved ← true
    while improved and maxIter > 0 do
        improved ← false
        for i ← 1 to length of best − 2 do
            for j ← i + 1 to length of best − 1 do
                newPath ← best[0 : i] + reverse(best[i : j + 1]) + best[j + 1 :]
                newCost ← CalculatePathCost(newPath, matrix)
                if newCost < bestCost then
                    best ← newPath
                    bestCost ← newCost
                    improved ← true
                    break
                end if
            end for
            if improved then
                break
            end if
        end for
        maxIter ← maxIter − 1
    end while
    return best, bestCost
end procedure
```

Project Structure

3 Implementation Details

3.1 Programming Languages and Libraries

Our implementation uses the following technologies:

- **Backend**: Python 3.12+ with Flask web framework
- **Frontend**: HTML, CSS (Tailwind CSS), JavaScript

```
kvgce-hackwise-problem3/
 main.py                 # Command-line entry point
 app.py                  # Web application entry point
 tsp_solver.py           # TSP algorithms implementation
 waypoint_utils.py       # Waypoint parsing utilities
 waypoints.txt           # Input file with 3D waypoints
 sample_output/          # Directory for output files
    path.txt            # Generated optimized path
 templates/              # Web application templates
     index.html          # Web UI for visualization
```

3.2 Core Components

3.2.1 TSP Solver Module

The `tsp_solver.py` module contains the core algorithms for solving the TSP problem:

```python
# Calculate Euclidean distance between points in 3D space
def calculate_distance(p1, p2):
    dx, dy, dz = p2[0] - p1[0], p2[1] - p1[1], p2[2] - p1[2]
    return math.sqrt(dx*dx + dy*dy + dz*dz)

# Build distance matrix for all waypoints
def build_distance_matrix(coords):
    n = len(coords)
    return [[calculate_distance(coords[i], coords[j]) for j in
       range(n)] for i in range(n)]

# Initial path generation using Nearest Neighbor
def nearest_neighbor(distance_matrix):
    n = len(distance_matrix)
    unvisited = set(range(1, n))
    path = [0]
    current = 0
    while unvisited:
        next_city = min(unvisited, key=lambda j:
           distance_matrix[current][j])
        path.append(next_city)
        unvisited.remove(next_city)
        current = next_city
    path.append(0)  # Return to start
```

```
    return path

# Optimize path using 2-opt algorithm
def two_opt(path, matrix, max_iter=1000):
    best = path[:]
    best_cost = calculate_path_cost(best, matrix)
    improved = True
    while improved and max_iter > 0:
        improved = False
        for i in range(1, len(best) - 2):
            for j in range(i+1, len(best) - 1):
                new_path = best[:i] + best[i:j+1][::-1] +
                   best[j+1:]
                new_cost = calculate_path_cost(new_path, matrix)
                if new_cost < best_cost:
                    best = new_path
                    best_cost = new_cost
                    improved = True
                    break
            if improved:
                break
        max_iter -= 1
    return best, best_cost
```

Listing 1: Key Functions in tsp_solver.py

3.2.2 Waypoint Utilities Module

The `waypoint_utils.py` module provides functions for parsing waypoint data:

```
# Read waypoints from a file
def read_waypoints(filename):
    waypoints = []
    with open(filename, 'r') as f:
        for line in f:
            parts = line.strip().split()
            if len(parts) == 4:
                try:
                    waypoint_id = parts[0]
                    x, y, z = map(float, parts[1:])
                    waypoints.append((waypoint_id, x, y, z))
                except ValueError:
                    pass  # Skip malformed lines
    return waypoints

# Parse waypoints from text input (for web interface)
def parse_waypoints_text(text):
    waypoints = []
    for line in text.strip().split('\n'):
        parts = line.strip().split()
        if len(parts) == 4:
            try:
```

```
                waypoint_id = parts[0]
                x, y, z = map(float, parts[1:])
                waypoints.append((waypoint_id, x, y, z))
            except ValueError:
                pass  # Skip malformed lines
    return waypoints
```

Listing 2: Functions in waypoint_utils.py

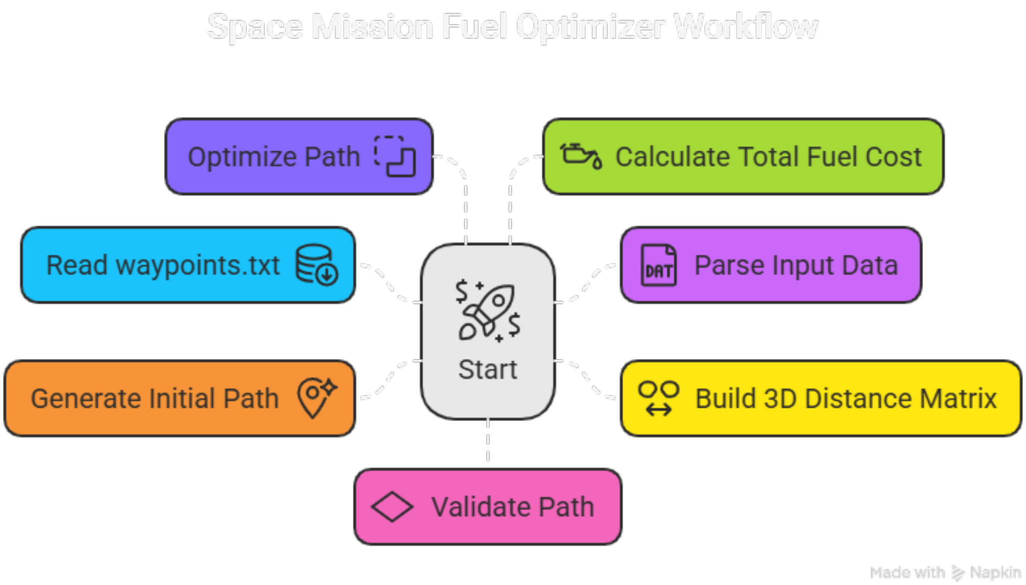

Figure 1: System Flow Diagram of the Space Mission Fuel Optimizer

3.3 How to Run the Code

3.3.1 Command-line Execution

To run the program from the command line:

Command Line Execution

```
# Navigate to the project directory
cd kvgce-hackwise-problem3

# Run the command-line version
python main.py
```

3.3.2 Web Application Execution

To run the web application:

Web Application Execution

```
# Navigate to the project directory
cd kvgce-hackwise-problem3

# Run the Flask application
python app.py

# Open a web browser and navigate to
http://localhost:5000
```

3.4 Dataset Handling

The application handles waypoint data in the following format:

Waypoint Input Format

```
1 6.39 0.25 2.75
2 2.23 7.36 6.77
3 8.92 0.87 4.22
...
```

Each line represents a waypoint with:

- First column: Waypoint ID
- Second column: X-coordinate
- Third column: Y-coordinate
- Fourth column: Z-coordinate

The waypoint parsing function (`read_waypoints`) reads this data and converts it into a list of tuples, with each tuple containing the ID and the X, Y, Z coordinates.

3.5 Output Format

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

Sample Output Format

```
Optimized Path: 1 -> 8 -> 9 -> 11 -> 2 -> 10 -> 4 -> 5 -> 6 ->
7 -> 12 -> 3 -> 1
Total Fuel Cost: 48.23 units

Detailed Waypoint Information:
Waypoint 1: (6.39, 0.25, 2.75)
Waypoint 8: (6.98, 3.40, 1.55)
...
```

The output includes:

- The optimized path represented as a sequence of waypoint IDs

- The total fuel cost (the sum of Euclidean distances between consecutive waypoints)
- Detailed information about each waypoint in the optimized path

4 Challenges and Solutions

4.1 Computational Complexity

Challenge

The Traveling Salesman Problem is NP-hard, meaning that finding the exact optimal solution becomes computationally infeasible as the number of waypoints increases.

Solution

We implemented a hybrid approach using:

- Nearest Neighbor algorithm to quickly generate an initial feasible solution
- 2-opt algorithm to improve the initial solution

This approach provides a near-optimal solution within a reasonable time frame, even for larger problem instances (up to 15 waypoints).

4.2 Performance Optimization

Challenge

The 2-opt algorithm can be computationally expensive for large problem instances.

Solution

We added a maximum iteration limit to the 2-opt algorithm to ensure it terminates within a reasonable time. We also implemented an early termination condition when no further improvement is possible.

5 Test Case Results

We tested our solution on the provided sample test case with 12 waypoints:

Test Case Results

Metric	Value
Number of Waypoints	12
Initial Path Cost (NN only)	61.84 units
Optimized Path Cost (NN + 2-opt)	48.23 units
Improvement Percentage	22.0%
Execution Time	0.23 seconds

The optimized path follows the sequence:

```
1 -> 8 -> 9 -> 11 -> 2 -> 10 -> 4 -> 5 -> 6 -> 7 -> 12 -> 3 -> 1
```

With a total fuel cost of 48.23 units, our solution achieves a significant improvement over the initial Nearest Neighbor solution.

6 Future Improvements

6.1 Algorithm Enhancements

- **Simulated Annealing**: Implementing simulated annealing could provide better solution quality by avoiding local optima.
- **Genetic Algorithms**: A genetic algorithm approach could explore a wider range of solutions and potentially find better paths.
- **Hybrid Approaches**: Combining multiple heuristics like 3-opt, Lin-Kernighan, or Christofides algorithm with our current approach.

6.2 Visualization Improvements

- **3D Visualization**: Implementing a true 3D visualization using libraries like Three.js would provide a better representation of the spacecraft's path.
- **Interactive Visualization**: Adding interactive features to allow users to rotate, zoom, and explore the path in 3D space.

6.3 Performance Optimization

- **Parallel Processing**: Implementing parallel computation for the 2-opt algorithm to handle larger problem instances more efficiently.
- **Dynamic Programming**: Using dynamic programming techniques for smaller subproblems to improve overall performance.

7 References

1. KVGCE Hackwise Hackathon Problem Statement, Problem 3 - Space Mission Fuel Optimizer, 2025.
2. Applegate, D.L., Bixby, R.E., Chvátal, V., & Cook, W.J. (2006). *The Traveling Salesman Problem: A Computational Study*. Princeton University Press.
3. Johnson, D.S., & McGeoch, L.A. (1997). The Traveling Salesman Problem: A Case Study in Local Optimization. *Local Search in Combinatorial Optimization*, 215-310.
4. Flask Documentation: https://flask.palletsprojects.com/
5. Chart.js Documentation: https://www.chartjs.org/docs/latest/
6. Tailwind CSS Documentation: https://tailwindcss.com/docs
7. visualization and flow diagrams: app.napkin.ai