OTTO Challenge

Ali Asl

July 14, 2024

1 Overview

These programs are designed to solve a routing problem for an OTTO robot navigating through a set of waypoints in a factory. The robot must start at a specified point, visit each waypoint in order, possibly skipping some waypoints with associated penalties, and end at a goal point. The goal is to minimize the total time taken, including travel time and penalties.

For this task, I employed two methods:

1. Dynamic programming
2. Integer programming (Branch and Cut).

However, other methods like A* can be also applied. The dynamic programming method has been implemented using both C++ and Python. The dynamic program does not need any mathematical modeling of the problem. However, the integer programming-based method needs to model the problem as an optimization problem, including binary decision variables. I first start by explaining the dynamic programming method and how to run the codes for both C++ and Python programs.

2 Dynamic Programming

The dynamic programming (DP) approach is used to find the optimal path through the waypoints. Here's how it works:

1. **Initialization**: The DP solver initializes a list of waypoints, including the start and goal points. It also sets up an array `dp` to store the minimum time required to reach each waypoint, initially setting all values to infinity except for the start point.
2. **Travel time calculation**: For each pair of waypoints, the travel time is calculated based on the Euclidean distance between them, divided by the robot's speed.
3. **Penalty calculation**: For each pair of waypoints, the penalty for skipping intermediate waypoints is summed.
4. **DP table update**: The algorithm iterates through each waypoint, updating the DP table by considering all possible previous waypoints. For each pair (j, i), the minimum time to reach i from j is calculated as the sum of:
    - The time to reach j (`dp[j]`).

- The travel time from j to i.
- The penalty for skipping any waypoints between j and i.
- The loading time at waypoint i.

This is represented by the formula:

$$\text{dp}[i] = \min(\text{dp}[i], \text{dp}[j] + t_{ji} + p_{ji} + \text{load\_time})$$

5. **Result extraction**: The minimum time to reach the goal point is found in the last entry of the DP table.

This method ensures that the solution is optimal by systematically considering all possible paths and choosing the one with the least total time.

2.1 C++

Components

1. **Main Program (`main.cpp`)**: Handles input selection, reads input data, and invokes the dynamic programming solver.
2. **Header File (`utils.h`)**: Declares the `waypoint` structure, the `read_input_file` function, and the `DP_Solver` class.
3. **Implementation File (`utils.cpp`)**: Implements the `read_input_file` function and the `DP_Solver` class methods.

Compilation Using a Makefile

1. In a terminal window, cd to the folder containing the files.
2. Build the project using the provided Makefile. This compiles both 'main.cpp' and 'utils.cpp, generating executables 'robot_program'.

```
make
```

To clean up generated files:

```
make clean
```

Running the Program

After compiling the program, you can run it using the following command:

```
./robot_program
```

If the program prompts for test case selection, you can enter:

- `s` for a small test case (`sample_input_small.txt`)
- `m` for a medium test case (`sample_input_medium.txt`)

- `l` for a large test case (`sample_input_large.txt`)

You can also provide input directly through standard input by piping a file or manual input. The commands for piping input files are:

```
cat sample_input_small.txt | ./robot_program
cat sample_input_medium.txt | ./robot_program
cat sample_input_large.txt | ./robot_program
```

You can also write the outputs in a file using the following commands:

```
cat sample_input_small.txt | ./robot_program | tee output_small.txt
cat sample_input_medium.txt | ./robot_program | tee output_medium.txt
cat sample_input_large.txt | ./robot_program | tee output_large.txt
```

2.2 Python

The Python version of the dynamic programming method is similar to that of the C++ version.

Components

1. **Main Program (`main.py`)**
2. **Dynamic Programming (`dynamic_programming.py`)**

Running the Program

You can run it using the following commands:

- Runtime test case selection:

  ```
  python3 main.py
  ```

- Providing input directly through standard input by piping a file or manual input:

  ```
  cat sample_input_small.txt | python3 main.py
  cat sample_input_medium.txt | python3 main.py
  cat sample_input_large.txt | python3 main.py
  ```

- Writing the outputs in a file:

  ```
  cat sample_input_small.txt | python3 main.py | tee output_small.txt
  cat sample_input_medium.txt | python3 main.py | tee output_medium.txt
  cat sample_input_large.txt | python3 main.py | tee output_large.txt
  ```

3 Integer Programming

The integer programming (IP) approach can also be used to solve this routing problem. The problem can be formulated as follows:

$$
\begin{aligned}
\min \quad & \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} x_{ij}\big(t_{ij} + p_{ij} + t_{load}\big) \\
& \text{s.t.} \\
& \sum_{j=2}^{n} x_{1j} = 1 \\
& \sum_{j=1}^{n-1} x_{jn} = 1 \\
& \sum_{j=1}^{k-1} x_{jk} = \sum_{j=k+1}^{n} x_{kj} \quad \forall k \in \{2, \ldots, n-1\}
\end{aligned}
\tag{1}
$$

Where:

- x_{ij} is a binary decision variable indicating whether the robot travels directly from waypoint i to waypoint j.
- t_{ij} is the travel time from waypoint i to waypoint j.
- p_{ij} is the penalty incurred for skipping intermediate waypoints between i and j.
- t_{load} is the loading time at each waypoint.

3.1 Explanation

- **Objective Function**: The objective is to minimize the total time, which includes travel time, penalty, and loading time.
- **Constraints**:
  - The robot must start its journey from the initial point (start point).

$$\sum_{j=2}^{n} x_{1j} = 1 \tag{2}$$

  - The robot must end its journey at the final point (goal point).

$$\sum_{j=1}^{n-1} x_{jn} = 1 \tag{3}$$

  - The robot must maintain continuity in its path, ensuring it arrives and departs from intermediate waypoints correctly.

$$\sum_{j=1}^{k-1} x_{jk} = \sum_{j=k+1}^{n} x_{kj} \tag{4}$$

This formulation ensures that the optimal path is chosen by evaluating all possible routes and selecting the one with the minimum total cost.

The integer programming model was implemented using the PuLP library, a Python library for linear programming. The Branch and Cut algorithm was employed to solve the IP problem. This method ensures that the optimal solution is found by exploring and pruning the solution space efficiently. Implementing Branch and Cut is complex and typically requires commercial solvers for effective performance.

3.2 Implementing the Mathematical Model

1. **Define the Problem**: Specify the problem as a minimization problem using `pulp.LpProblem`.
2. **Create Decision Variables**: Define binary decision variables x_{ij} to represent the robot's path between waypoints.
3. **Set the Objective Function**: Formulate the objective function to minimize the total cost, including travel time, penalties, and loading time.
4. **Add Constraints**: Incorporate the constraints to ensure the robot starts at the initial waypoint, ends at the final waypoint, and maintains continuity in its path.
5. **Solve the Problem**: Use the solver in PuLP to find the optimal solution. PuLP uses Branch and Cut internally to solve the integer programming problem.

Using specialized solvers like Gurobi or CPLEX can even yield better performance and faster results, especially for larger and more complex instances of the problem. These solvers are highly optimized for solving linear and integer programming problems and can handle larger datasets more efficiently.

Components

1. **Main Program (`main.py`)**
2. **Integer Programming (`integer_programming.py`)**

Running the Program

You can run it using the following commands:

- Runtime test case selection:

```
python3 main.py
```

- providing input directly through standard input by piping a file or manual input:

```
cat sample_input_small.txt | python3 main.py
cat sample_input_medium.txt | python3 main.py
cat sample_input_large.txt | python3 main.py
```

- Writing the outputs in a file:

```
cat sample_input_small.txt | python3 main.py | tee output_small.txt
cat sample_input_medium.txt | python3 main.py | tee output_medium.txt
cat sample_input_large.txt | python3 main.py | tee output_large.txt
```