CS5800 Project - Road Trip Path Optimizer

Angela Tao, Chanthavary Prak and Jaee Oh

December 7, 2024

1 Introduction

Going on a road trip is a very popular way of traveling, which offers opportunities for adventure, exploration, and memorable experiences. However, planning an optimal road trip route involves balancing multiple considerationstime efficiency, scenic beauty, traffic conditions, weather, and personal preferences. Navigating these variables without a reliable tool can make trip planning time-consuming and inefficient.

Pathfinding algorithms, such as Dijkstras Algorithm, have been widely applied to solve optimization problems across various domains, from transportation networks to logistics and robotics. The problem of planning a road trip, where users need to find the best route between two cities while accounting for multiple weighted factors, provides an ideal use case for such algorithms.

In addition, advancements in data collection and availability of traffic and weather data have made it possible to design systems that dynamically respond to real-world conditions. However, real-time data integration can introduce significant complexity in algorithm design and implementation, making it a challenging component for short-term projects. By focusing on static data for this project, we aim to simulate these real-world variables in a controlled environment to demonstrate the feasibility of a path optimization tool.

In this project, we planned and proposed this Road Trip Path Optimizer to not only provide a practical demonstration of algorithmic application in travel planning but also serve as a stepping stone for future projects that might integrate real-time data and more advanced features. This project highlights the importance of computational techniques in solving everyday challenges, combining technical rigor with user-centric design to enhance the travel experience.

2 Defined Question

Our central question is: "How can we optimize road trip routes to balance user preferences, including travel time, scenic appeal, real-time traffic, and weather conditions?" Our project aims to answer this question by building an algorithm that evaluates multiple real-time factors and user-defined preferences to determine the most optimal route for road trips. This involves balancing both objective criteria (like time and traffic) and subjective criteria (like scenic value).

3 Why This Question Matters to Us

3.1 Angela

Our project focuses on developing a Road Trip Path Optimizer that uses real-time data to help users find the best route based on various criteria such as travel time, scenic views, weather, best food route, traffic conditions, etc. As someone who loves road trips, I often struggle to find routes that balance my desire for scenic beauty, a nice place to eat, and with the need for time efficiency. This tool would make trip planning easier and enhance my travel experience.

3.2 Chanthavary

The idea of optimizing road trip routes is relevant to me because they are about more than just reaching a destinationtheyre about creating memorable experiences along the way. Balancing practical factors like travel time and traffic with personal preferences, like scenic routes, can make travel both efficient and enjoyable. With this project, we want to develop an algorithm that simplifies and personalizes trip planning, allowing people to tailor their journeys and achieve a balance between efficiency and enjoyment.

3.3 Jaee

Pathfinding algorithm is one of the most common algorithms that we can find in real life. We even unconsciously use such algorithms when we are walking to the campus on a daily basis. However, in a situation of a road trip, picking an inefficient route may result in much bigger consequences in contrast to losing a few minutes by taking a wrong turn when you are walking to the campus. We may lose some extra money for being late in certain places, or miss a beautiful sunset that everyone wanted to see. Therefore, a road trip is a good case to apply a pathfinding algorithm for a better experience.

4 Analysis

The goal of this project was to develop an optimized road trip route using a pathfinding algorithm that balances multiple factors such as travel time, scenic appeal, traffic conditions, and weather. Below, we outline the approach and methods we used to gather data and solve the problem.

4.1 Problem Understanding

We began by defining the problem: how to optimize road trip routes based on multiple factors like time efficiency, scenic appeal, traffic conditions, and weather. The objective was to create an algorithm capable of calculating the best route by considering both objective (time, traffic) and subjective (scenic appeal/points of interest) factors. This directly addresses the central question posed in our project proposal.

4.2 Graph Construction

To represent the road network, we constructed a graph where each node corresponds to a city, and each edge represents a road segment connecting two cities. The weight of each edge reflects the travel time between cities, which we calculated based on several factors such as distance, speed limit, traffic, weather, and the presence of traffic lights. This graph structure is ideal for applying pathfinding algorithms like Dijkstras, as it provides an efficient way to represent the network and process route calculations.

4.3 Dijkstras Algorithm Implementation

We chose Dijkstras algorithm to compute the shortest path between two cities, given the weights we assigned to each road segment. The algorithm uses a priority queue (min-heap) to explore the shortest path by updating the travel time (weight) for neighboring nodes as it proceeds through the graph. Dijkstras algorithm is well-suited for this project because it guarantees the shortest path in terms of travel time, based on our custom weights.

4.4 Weight Calculation

A critical part of our solution was implementing a weight calculation function to compute the cost of traveling each road segment. The weight for each edge was determined by the distance between cities, the speed limit, and additional multipliers that accounted for traffic conditions, weather, and the presence of traffic lights. This approach allowed us to simulate real-world conditions without relying on dynamic real-time data. The weight function ensures that the optimization algorithm considers multiple factorsboth objective, such as time-related considerations, and subjective, like scenic appeal, which could be integrated into future versions.

4.5 Data Collection and Static Graph

For the purpose of this project, we used static data, meaning that the cities, routes, distances, speed limits, traffic conditions, weather, and traffic light presence were predefined manually. We created a graph with realistic travel connections, such as routes from Boston to Hartford and from Hartford to New York, adjusting travel times based on the factors listed above. This static approach allowed us to focus on algorithm design without the complexity of integrating dynamic real-time data at this stage.

4.6 Path Calculation and Output

Once Dijkstra's algorithm computes the shortest path, we retrieve the route from the algorithms result, which gives us the sequence of cities to travel through in the optimal order. The output is the optimized route along with the total travel time, accounting for all the factors we've integrated.

5 Conclusion

The solution to our problem lies in leveraging Dijkstras algorithm to compute the shortest path between a starting city and a destination while factoring in user-defined preferences such as traffic conditions, weather, speed limits, and the presence of traffic lights. By optimizing travel time through a balance of objective criteria (like time and traffic) and subjective considerations (such as scenic value, which could be added in future iterations), our project demonstrates the feasibility of a pathfinding tool tailored to road trip planning.

This prototype effectively simulates real-world conditions using static data, providing a controlled environment to explore the algorithm's potential. However, as a foundational step, this solution is limited in its current form and serves as a springboard for future enhancements. Addressing these limitationssuch as incorporating real-time data, improving scalability, and refining user-centric featureswill be essential for broader real-world applications.

With these improvements, the Road Trip Path Optimizer could evolve into a dynamic tool capable of enriching travel experiences while providing practical solutions to everyday challenges.

5.1 Weaknesses and Limitations

While the current project successfully demonstrates the use of Dijkstras algorithm for pathfinding, it has several limitations that need to be addressed for real-world application and scalability: Static Data: The current graph is based on static data for distances, speed limits, traffic conditions, and weather. In a real-world application, this data would need to be dynamic, fetched from real-time sources like traffic APIs, weather services, and GPS systems. Scalability: The solution is optimized for relatively small graphs (like a few cities). For larger datasets or more complex graphs, Dijkstras algorithm could become inefficient due to its time complexity. Exploring parallelized solutions or algorithms like A* search could improve scalability for larger problems. Simplistic Scenic Values: Factors like scenic beauty are subjective and hard to quantify accurately. Real-Time Adjustments: The current algorithm doesn't adjust for real-time events like accidents, road closures, or detours. Adding these factors would make the optimizer more accurate and dynamic.

5.2 Future Research

To address these limitations, future work will focus on: Real-Time Data Integration: Future work should focus on integrating real-time traffic and weather data to make the route optimizer dynamic. User-Centric Customization: Allow users to prioritize factors like scenic appeal or rest stops. Multi-Objective Optimization: Incorporating additional factors, such as scenic value, would require a more sophisticated multi-objective optimization approach, where the user can weigh different preferences according to their needs. Algorithm Enhancement: Research into more efficient algorithms for large-scale graphs (such as A* search)

or parallel computation strategies would be beneficial for handling larger road networks.

As weve seen, Dijkstras algorithm provides a solid foundation for optimizing road trip routes by considering both practical and subjective factors. While the current prototype demonstrates the feasibility of this approach, there are several areas for improvement to make it more dynamic, scalable, and user-centric.

The work presented here is just the beginningfuture iterations of the Road Trip Path Optimizer will benefit from real-time data integration, scalability improvements, and a more refined user experience. As we continue to innovate, the potential for this tool to enhance travel experiences grows, offering travelers a smarter, more personalized journey.

6 Individual Reflections

6.1 Angela

I have always find applying algorithm into real life problem very difficult and also different from thinking and coding in text book problems. This project gives me experience in actually implementing the Dijkstras algorithm into solve this road trip problem. Even though due to time constraint, we were not able to finish and implement all the features we planned to originally, but I definitely learned a lot in this process. From Graph theory to programming the algorithm out in Google Colab, I can see my improvement of thought process and also ability to transform ideas to algorithm and code. I hope that in the future we can finalize this project with all of our ideas possible, and can definitely see myself using the skills i learnt in the process.

6.2 Chanthavary

This project gave me practical experience in applying Dijkstras algorithm to solve a real-world problem. I learned how to model cities and routes using graphs and how to factor in real-world complexities like traffic and weather into an optimization problem. The experience also deepened my understanding of algorithm design, especially in terms of time complexity and the use of efficient data structures like priority queues. This project is valuable to me, not only because it strengthens my technical skills in graph theory and algorithm design, but also because it applies these concepts to a problem with real-world significance. I can see myself using these skills in future projects.

6.3 Jaee

The hardest challenge of this project was not coding Dijkstras algorithm itself, but how to find relevant data that can represent a real life situation. There was a lot of live traffic data online, but they were all commercialized data that required us to buy licenses to use them. This let us create our own sample data to be used in this project, but then there was a question of whether our format of data is close enough to the commercially available data or not. This realization once again made me realize some benefits of open source softwares and creating standard formatting for certain types of data. Moreover, object-oriented coding style was very suitable for Dijkstras algorithm, and it is suitable for many different algorithms due to its readability.

7 Appendix

7.1 Appendix A. Links to Sources

All of our relevant links are stored in the Github repo.

7.2 Appendix B. Source Code

```
# Importing necessary libraries
import heapq
import folium


# Graph structure
class Graph:
    def __init__(self):
        self.edges = {}

    def add_edge(self, from_node, to_node, weight):
        if from_node not in self.edges:
            self.edges[from_node] = []
        self.edges[from_node].append((to_node, weight))


# Dijkstra's algorithm implementation
def dijkstra(graph, start_node, end_node):
    pq = []
    heapq.heappush(pq, (0, start_node))  # (cumulative weight,
                                         current node)
    shortest_paths = {start_node: (None, 0)}  # node: (previous
                                         node, cumulative weight)

    while pq:
        current_weight, current_node = heapq.heappop(pq)

        # If we've reached the destination
        if current_node == end_node:
            break

        for neighbor, weight in graph.edges.get(current_node, []):
            new_weight = current_weight + weight
            if (
                neighbor not in shortest_paths
                or new_weight < shortest_paths[neighbor][1]
            ):
                shortest_paths[neighbor] = (current_node,
                                                    new_weight)
                heapq.heappush(pq, (new_weight, neighbor))

    # Reconstruct the shortest path
    path, total_weight = [], shortest_paths.get(end_node, (None,
                                         float("inf")))[1]
    node = end_node
    while node:
        path.append(node)
        next_node = shortest_paths[node][0]
```

```
        node = next_node
    path.reverse()

    return path, total_weight


# Function to calculate weight of an edge
def calculate_weight(
    distance,
    speed_limit,
    traffic_multiplier=1,
    weather_multiplier=1,
    traffic_lights=False,
):
    base_time = distance / speed_limit  # Base time (hours)
    weight = base_time * traffic_multiplier * weather_multiplier
    if traffic_lights:
        weight *= 1.1  # Add 10% penalty for traffic lights
    return weight


# Create the graph
def create_static_graph():
    graph = Graph()

    # Add edges (Example: Boston -> Hartford -> New York), modify
                                    with static data
    # Format: (distance in miles, speed limit in mph, traffic
                                    multiplier, weather
                                    multiplier, traffic lights)
    routes = [
        ("Boston", "Providence", 50.9, 65, 1.2, 1, False),
        ("Boston", "Worcester", 47.4, 65, 1.2, 1, False),
        ("Springfield", "Hartford", 26.9, 65, 1.2, 1, False),
        ("Worcester", "Springfield", 52.6, 65, 1.2, 1, False),
        ("Worcester", "Hartford", 62.5, 65, 1.2, 1, False),
        ("Worcester", "Providence", 39.5, 65, 1.2, 1, False),
        ("Providence", "Hartford", 86.4, 55, 1.2, 1, False),
        ("Providence", "New Haven", 103.0, 55, 1.2, 1, False),
        ("Hartford", "New Haven", 39.0, 55, 1.2, 1, False),
        ("New Haven", "New York", 81.6, 55, 1.2, 1, False),
    ]

    for from_city, to_city, distance, speed_limit, traffic, weather
                                    , lights in routes:
        weight = calculate_weight(distance, speed_limit, traffic,
                                        weather, lights)
        graph.add_edge(from_city, to_city, weight)
        graph.add_edge(to_city, from_city, weight)  # Assuming
                                        roads are bidirectional

    return graph


if __name__ == "__main__":
    # Main function to demonstrate the optimizer
    graph = create_static_graph()
```

```
start_city = "Boston"
destination_city = "New York"

# Calculating shortest route
path, total_time = dijkstra(graph, start_city, destination_city
                                    )

# Displaying the results
results = {
    "Optimized Path": " -> ".join(path),
    "Total Travel Time (hours)": f"{total_time:.2f}",
}
print(results)

# Main function to demonstrate the optimizer
graph = create_static_graph()
start_city = "Boston"
destination_city = "New York"

# Calculating shortest route
path, total_time = dijkstra(graph, start_city, destination_city
                                    )

# --- Visualization using folium ---
# Sample coordinates (replace with actual coordinates for
                                    cities)
city_coordinates = {
    "Boston": (42.3601, -71.0589),
    "Providence": (41.8240, -71.4128),
    "Worcester": (42.2626, -71.8023),
    "Springfield": (42.1015, -72.5898),
    "Hartford": (41.7637, -72.6851),
    "New Haven": (41.3083, -72.9279),
    "New York": (40.7128, -74.0060),
}

# Create the map centered around the starting city
map_osm = folium.Map(location=city_coordinates[start_city],
                                    zoom_start=7)

# Add markers for each city on the path
for city in path:
    if city in city_coordinates:
        folium.Marker(location=city_coordinates[city], popup=
                                            city).add_to(map_osm)

# Draw the route on the map
for i in range(len(path) - 1):
    if path[i] in city_coordinates and path[i + 1] in
                                        city_coordinates:
        folium.PolyLine(
            locations=[city_coordinates[path[i]],
                                                city_coordinates[
                                                path[i + 1]]],
            color="blue",
            weight=2.5,
            opacity=1,
```

```
            ).add_to(map_osm)

    # Save the map as html.
    map_osm.save("src/Boston_NY.html")
```