CSE 202 Project
Winter, 2025
CSE 202 Project: Implementation

Submitted by:
Shresth Grover
Isheta Bansal
Ronit Chougule
Balaaditya Mukundan
Akshay Gopalkrishnan

CSE 202 Project: Project Implementation

Recap of Problem

The project models a Mario Kart race track as a computational problem by representing the track as a 3D weighted graph. The key elements of the problem are:

1. 3D Grid Representation:
   - The track is modeled as a 3D grid with dimensions X, Y, Z (length, width, elevation)
   - Minimum dimensions of 100×250×25, resulting in at least 625,000 vertices
   - Each grid cell/vertex has coordinates (x, y, z)
2. Terrain Types:
   - Each vertex is assigned a terrain type: Road (60%), Grass (10%), Boost (15%), Obstacle (10%), OutOfBounds (5%), Item (5%)
   - Each terrain type affects the kart's speed (traversal time)
3. Edge Weights:
   - Edges represent possible movements between vertices
   - Weights represent traversal time based on terrain type:
     - Road: weight = 1
     - Grass: weight = 2
     - Boost: weight = 0.5
     - Obstacle: weight = 1.5
     - OutOfBounds: weight = 5
     - Mushroom Item: weight = 0.5
4. Special Edge Construction:
   - The graph is constructed to ensure a lap-like flow:
     - For $x < X/2$: y movement is only in the increasing direction
     - For $x \geq X/2$: y movement is only in the decreasing direction
   - Special boundary rules prevent shortcuts or "cheating."
5. Objective:
   - Find the shortest path (minimum time) to complete one lap
   - Start at a random point (x, y, z) and finish at the point directly behind it (x, y-1, z)

Recap of Algorithm

The project implements and compares two algorithms:

1. Baseline: Dijkstra's Algorithm (O(n log n) complexity):
   - Traditional implementation using a priority queue/min-heap
   - Guarantees finding the optimal shortest path
   - Inefficient for very large graphs due to priority queue operations

2. A* Algorithm (O(n log n))
   - Uses heuristics (Euclidean distance) to prioritize expansion towards the goal.
   - More efficient than Dijkstra's but still scales as
   - O((V+E)logV).
   - Better for large graphs than Dijkstra, but not as efficient as BFS.
3. Improved: BFS with Edge Weight Scaling (O(α(n + m))) complexity):
   - Transform the weighted graph into an unweighted graph by:
     - Multiplying all weights by 2 to handle non-integer weights
     - Adding dummy vertices between edges proportional to their weights
   - Run standard BFS on the transformed graph.
   - This approach maintains the same optimal path as Dijkstra's but runs in linear time.
   - Near-linear time complexity: O(α(n+m)), where α is the graph expansion factor.

The key innovation was transforming the weighted graph into an equivalent unweighted representation by scaling edge weights and introducing intermediate nodes, allowing BFS to approximate shortest paths while improving computational efficiency. This approach is particularly valuable for very large graphs, as it reduces the reliance on expensive priority queue operations while preserving optimal path properties.

Implementation Environment Setup & Testing

This section outlines the experimental setup, the libraries and classes implemented and the methodology used for testing and evaluating the project.

Experimental Setup

The primary objective of our project was to implement and compare two shortest path algorithms on a randomly generated 3D grid-based graph. The algorithms we evaluated were:

- Dijkstra's Algorithm
- BFS with Edge Weight Scaling

To simulate a realistic environment, we created a randomized 3D grid graph, where:

- Nodes represent points in a 3D space.
- Edges represent possible paths between nodes, with weights assigned based on the terrain or node type.

We ran multiple iterations of the experiment, each time generating a new randomized graph and measuring the performance of both algorithms in terms of shortest path length and execution time. The results were visualized in a 3D plot for further analysis.

Libraries and Classes Implemented

- **Programming Language**
  - The project was implemented using Python 3, leveraging its robust libraries for graph processing, numerical computation, and visualization.
- **External Libraries Used**
  - **numpy** - Used for numerical computations, including random graph generation and matrix operations.
  - **matplotlib** - Used for visualizing the 3D graph and the computed shortest paths.
  - **seaborn** - Used for generating statistical visualizations of algorithm performance
  - **collections.deque** - Used for an optimized queue implementation in BFS.
- **Custom Classes Developed**
  - **Graph:**

    The Graph class represents a 3D grid-based graph, where nodes correspond to specific points in the 3D space. It is responsible for generating these nodes, categorizing them into different types, and establishing weighted edges between them. The class provides essential methods to reset node attributes and initialize random connections, ensuring flexibility in different experimental setups.

- **Node**

  The Node class represents an individual node in the 3D space. Each node stores key information, including its coordinates, type, neighboring nodes, and attributes related to shortest path calculations. These attributes facilitate the execution of pathfinding algorithms by maintaining relevant details about each node's position and connectivity within the graph

- **MyHeap**

  The MyHeap class is a custom implementation of a binary min-heap. It is designed to efficiently manage and retrieve the node with the smallest distance value. This enables fast extraction of the highest-priority node which is critical in Dijkstra's Algorithm.

Testing & Evaluation

To validate our implementation, we conducted extensive testing under different conditions.

- **Generating Randomized Graphs**

  We created multiple 3D grid graphs of varying sizes to test the scalability and robustness of our implementation. Each node within these graphs was assigned a randomly generated weight, simulating different terrain types and varying traversal difficulties

- **Selecting Random Start Nodes**

  For each test iteration, we randomly select a start node within the 3D graph.

- **Recording Performance Metrics**

  To measure and compare the efficiency of the two algorithms, we collected key performance metrics. These included the Shortest Path Length, verifying whether both algorithms produced the same optimal path length; Execution Time, assessing which algorithm performed faster across different graph sizes. These metrics provided insights into the trade-offs between accuracy, speed, and resource utilization.

- **Visualizing the Results**

  To better understand the results, we utilized visualization techniques to analyze the computed paths and performance metrics. The 3D graph structure was plotted with clearly defined nodes and edges, while the paths calculated by each algorithm were highlighted using different colors to distinguish between them.

Average Time Complexity Analysis on Varying Mario Kart Graph Sizes

Algorithm scalability is a critical factor in determining its applicability to large-scale problems. As input size increases, the computational efficiency of an algorithm can significantly impact its usability, especially in real-time or resource-constrained environments. In this section, we analyze how our BFS-based shortest path algorithm and Dijkstra's algorithm scale with increasing graph sizes. Our goal is to understand how our algorithm's execution time grows as the number of vertices increases in the graph.

Experimental Setup

To evaluate the scalability of our BFS-based shortest path algorithm and the baseline, we conducted controlled experiments by varying the size of the graph and measuring execution times. The goal was to systematically increase the number of vertices and observe how each algorithm's runtime grows in response.

Graph Structure and Scaling Methodology

Our graph is modeled as a 3D grid-based structure where each vertex represents a point on the track, and edges represent possible movement paths. The total number of vertices in the graph is determined by the product of its dimensions:

$$\text{Graph Size} = X \times Y \times Z$$

where X represents the length of the graph, Y represents the width, and Z represents the elevation variations in the 3D space. To assess scalability, we varied X, Y, and Z across different experiments, increasing the total number of nodes exponentially.

Graph Sizes Used in Experiments

We tested the algorithms on the following graph sizes:

	Graph Dimensions ($X \times Y \times Z$)
1	50 × 250 × 10
2	100 × 500 × 25
3	150 × 750 × 35
4	200 × 1000 × 50

These sizes were chosen to start with a small-scale graph for baseline performance. We gradually increase the graph size to observe computational scaling trends. Finally, we ensure to test a very large graph to measure worst-case execution times.

Observations & Analysis

The collected execution time data reveals clear trends in the scalability as the graph size increases. This section analyzes the growth patterns, performance differences, and theoretical alignment of these results.

1. BFS vs Baseline Performance Trends

The recorded execution times are as follows:

Graph Size (X × Y × Z)	BFS Time (sec)	Dijkstra Time (sec)
50 × 250 × 10	1.06027	4.51661
100 × 500 × 25	15.72480	69.21898
150 × 750 × 35	65.56814	249.43162
200 × 1000 × 50	278.94271	699.91091

From this, we observed that BFS consistently runs faster than Dijkstra's algorithm across all graph sizes. The performance gap widens significantly as the graph size increases. Dijkstra's algorithm scales much worse, increasing in execution time by a larger factor compared to BFS. This confirms that BFS handles large graphs better, making it a more efficient choice for large-scale applications.

2. Growth Rate Analysis

Execution time data was collected and plotted against the total number of nodes in the graph. The resulting graph provides a visual representation of how each algorithm scales. The graph below plots execution time as a function of the number of nodes:

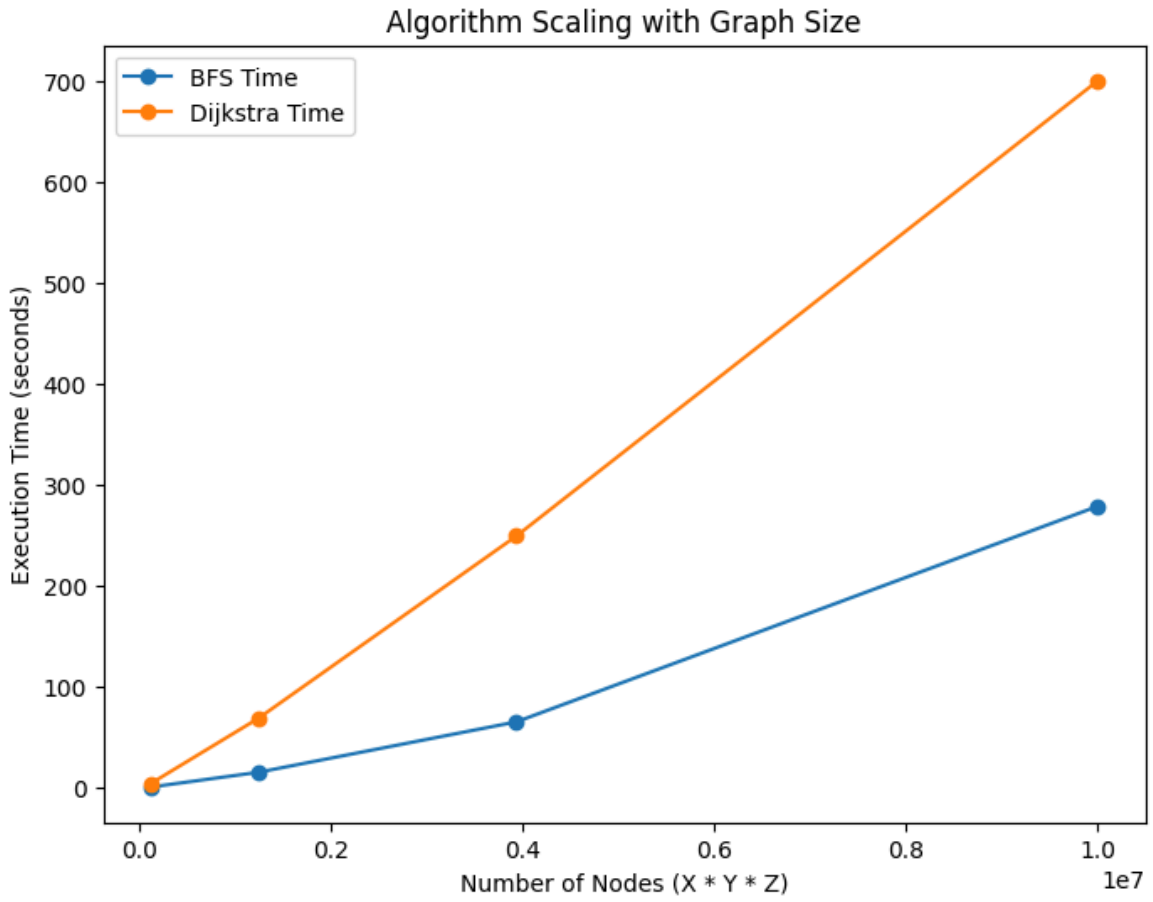

Figure 1: Algorithm Scaling Graph

Observing the slopes of both curves:

- BFS exhibits a near-linear growth pattern $O(n)$, where execution time increases proportionally to the number of nodes.
- Dijkstra's algorithm follows a superlinear trend $O(n \log n)$, confirming the expected impact of priority queue operations.

This is evident from the increase in time where BFS time increases by approximately 4x when the graph grows. Dijkstra's time increases by over 10x, reflecting the added computational cost of heap operations. As graph size increases, BFS maintains a more predictable and manageable growth in execution time, whereas Dijkstra's algorithm experiences severe performance degradation.

Experiment: Runtime Comparison of Dijkstra's Algorithm vs. BFS on a Constant 3D Graph with Weighted Nodes

Objective

The goal of this experiment is to compare the runtime performance of Dijkstra's algorithm and a modified BFS approach on the same graph configuration, designed to simulate a 3D environment with heterogeneous node types and variable edge weights. We use the minimum Mario Kart graph size, `100 x 250 x 25`, which totals 625,000 nodes for time analysis. For each trial, we still randomly initialize the tiles in our graph such that the nodes in the graph are different each iteration.

Algorithms Compared

Dijkstra's Algorithm

- Utilizes a custom min-heap.
- Tracks node distances and predecessors.
- Terminates upon reaching the goal.
- Measures total runtime and final path cost.

Breadth-First Search (Modified)

- Simulates variable edge weights using chains of dummy nodes.
- Applies standard BFS on the transformed graph.
- Scales weights by 2 to maintain integer steps.
- Reconstructs shortest path from dummy node trail.

Experiment Protocol

- Repeated over 15 randomized graphs.
- For each run:
  - New graph and start node are initialized.
  - Both algorithms are executed from the same start.
  - Runtime and path costs are recorded.
  - Results are visualized via plots.

Observations

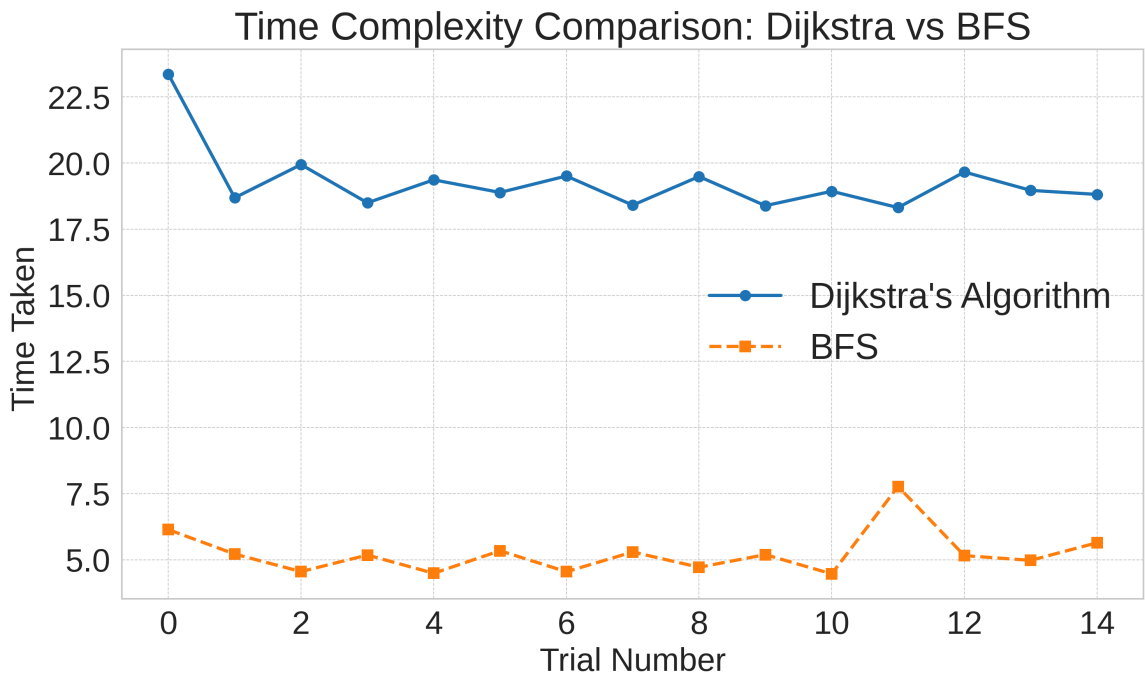

Figure 2: Algorithm Runtime Comparison

Execution Time

- **BFS** outperforms Djikstra in raw runtime, easily outperforming Dijkstra for every iteration.
- Djikstra is slower due to its O((E)log(V)) runtime, while we can see :
- Runtime plots across 15 runs show:
  - Dijkstra has stable performance, and besides one outlier iteration the time taken ranges from approximately $[18.75, 20]$. The BFS algorithm has a time taken range $[5, 7.5]$.

- BFS time varies more, especially with obstacle-dense graphs in which the optimal path may require us to explore longer paths, as traversing through slower tile types will require us to traverse through more edges. However, the time remains relatively consistent, which is what we expect since we run experiments on a consistent graph size.

Visualization of Shortest Path Answers

To verify that our algorithm implementation is following the graph flow constraints such that it completes a lap around a Mario Kart track without "cheating", we visualize the 3D path generated by our algorithm. In addition, we also want to verify that our algorithm properly works and returns the same shortest path distance as our baseline algorithm. To keep track of the final paths, we keep track of previous vertices throughout both algorithms, so at the end we can reconstruct the full path.

We randomly initialize four graphs using our graph reconstruction procedure to initialize a Mario Kart race track with a starting point. With these graphs, we run both Dijkstra's and our efficient BFS algorithm from the starting point to find the fastest lap. The visualization of the paths collected from both of these algorithms is shown in Figure 3. We can verify the correctness of our algorithm and graph construction in the following ways:

Correct Minimum Lap Distance

- In each of the graphs, we can see that the BFS algorithm achieves the same minimum lap distance as Dijkstra's. Therefore, we don't sacrifice any correctness for efficiency with this algorithm, as we can find the same optimal paths as Dijkstra's but in $O(n)$ instead. In the path visualizations, you will notice that the Dijkstra paths do differ from the BFS paths slightly. While we do end up with paths that have the same distance as Dijkstra's, it is not guaranteed to return the same overall path due to the order in which we can visit vertices in BFS. It is very likely in our case that BFS finds a different optimal path than Dijkstra's since we are dealing with very large graph sizes, which increases the chance for a larger amount of optimal paths in a Mario Kart race track.

Correct Graph Flow

- One of the key aspects of the Mario Kart graphs that we must simulate is that to get back to the starting point, a player must complete a full lap. To enforce this, we added rules for edges such that the only way to complete a lap back to the starting point was by completing some approximate path around our whole Mario Kart 3D graph. Therefore, we need to verify that our graph construction prevents our algorithms from taking some unwanted shortcuts that would lead to a shortest path that is much smaller than expected.
- We can see that the paths visualized follow this lap-like nature we expect, in which we have to traverse almost fully in the x and y directions to make it back to the starting point. In particular, you will notice the following about paths generated which reflect the rules we defined for edge construction:
  - From $\left[0, \frac{X}{2} - 1\right]$, the vertical flow is only in the $+y$ direction, and from $\left[\frac{X}{2}, X\right]$ the vertical flow is only in the $-y$ direction.
  - In order to cross from the $\left[0, \frac{X}{2} - 1\right]$ region to the $\left[\frac{X}{2}, X\right]$ region, the y value must be greater than $0.95Y$. To cross back from the $\left[\frac{X}{2}, X\right]$ region into $\left[0, \frac{X}{2} - 1\right]$, our y values must be less than $0.05Y$. Otherwise, there are no constraints on the edge flow in the x direction.
  - There are no constraints in the z-dimension, so a path can flow freely in this dimension throughout a path.

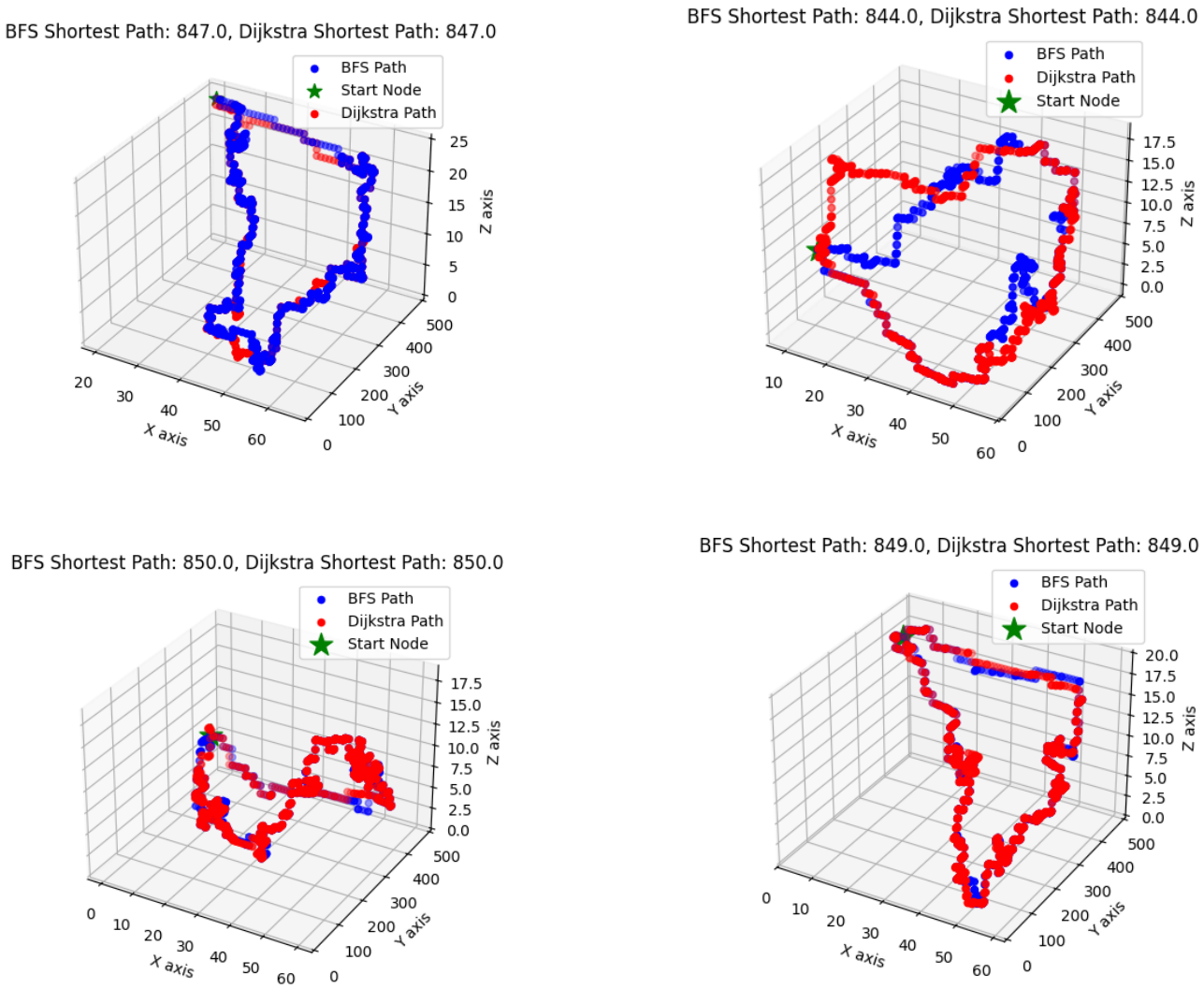

Figure 3: Example Mario Kart paths from proposed algorithm in randomly generated graphs

Conclusions

- From our experiments for finding efficient paths for Mario Kart, we were able to conclude the following about our solution and problem formulation:
  - In comparison to classic graph algorithms like Dijkstra's, we saw how our modified BFS approach to find the fastest lap around the Mario Kart track was more efficient and scalable. We noticed that the gap between the baseline algorithm and our proposed solution increased with larger graph sizes, as the time complexity algorithm grows in order $O(n)$ while the baseline grows in order $O(n \log n)$.
  - Our BFS algorithm maintains correctness and is able to find paths with same distance as Dijkstra's, proving that our algorithm, while more efficient, still holds complete correctness.
  - From visualizations of paths generated by our algorithm, we can confirm that our edge flow defined in our graph formulation does force any path to perform a circular-like lap similar to how Mario Kart is played.
- Limitations
  - Our algorithm assumes that the Mario Kart graph maintains the same properties while a player performs a lap around the track. However, in the actual gameplay, there are some changing elements like the item received at a item tile that we don't account for in our algorithm.
  - This algorithm also assumes that there aren't any other players in our environment, but in real Mario Kart games one often has to compete with other players who can affect the course a player takes on a Mario Kart track.