CSE 202 Project
Winter, 2025
CSE 202 Project: Algorithm & Analysis

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

CSE 202 Project: Algorithm & Analysis

Graph Model of a Looped Mario Kart Race Track

We define a directed weighted graph $G = (V, E)$ that models a looped Mario Kart race track on a 3D grid. This construction enables precise pathfinding and race strategy optimization by rigorously accounting for spatial dimensions and terrain-dependent traversal speeds.

1. Vertex Set V

Let $X, Y, Z \in \mathbb{N}$ denote the grid dimensions along the track's length, width, and elevation, respectively. We define the vertex set as: $V = \{(x, y, z) \in \mathbb{Z}^3 \mid 0 \leq x < X, 0 \leq y < Y, 0 \leq z < Z\}$. For all Mario Kart graphs, we will assume that $X \geq 100, Y \geq 250, Z \geq 25$, so the minimum number of edges would be $100 \times 250 \times 25 = 625000$. We can see for this problem, we are dealing very large graph sizes due to the 3D nature of our problem.

Each vertex $u = (x_u, y_u, z_u)$ is endowed with a terrain type function: $T : V \to \{\text{Road}, \text{Grass}, \text{Boost}, \text{Obstacle}, \text{OutOfBounds}, \text{Item}\}$ which influences the local traversal speed. The terrain function T uses the following probability distribution to assign each vertex a terrain type:

- 60% normal tiles, 5% item tiles, 10% out of bounds, 10% grass, and 15% speed boost tiles.
- In Mario Kart, the item tile offers the player to acquire a random item, each of which can benefit or hurt the player's finish time. For now, we assume that each item is either a speed boost mushroom, or some item that negatively impacts the player which we define as an obstacle. Since we need the item tiles to be constant throughout our algorithm, we randomly assign with a uniform distribution the item type when a vertex is assigned as an item.
- We also need to define the starting point $s = (x, y, z)$ of the Mario Kart track. We assume that for our starting point s, $0 \leq x < \frac{X}{2}, 0.1Y < y < 0.9Y, 0 \leq z \leq Z$. We can randomly assign the location of our starting point using these bounds.

2. Edge Set E

Edges are introduced to enforce a directed, loop-like progression through the grid. For any vertex $u = (x, y, z) \in V$, we define the outgoing edges as follows:

2.1 Movement in the x-Direction

- Forward Movement: If $x + 1 < X$, include the edge $(u, (x + 1, y, z))$
- Backward Movement: If $x - 1 \geq 0$, include the edge $(u, (x - 1, y, z))$
- We don't add a forward movement between any nodes at $\left(\frac{x}{2}, y, z\right)$ and $\left(\frac{x}{2} + 1, y, z\right)$ if $y < 0.95Y$, and don't add a backward movement between any nodes at $\left(\frac{x}{2}, y, z\right)$ and $\left(\frac{x}{2} + 1, y, z\right)$ if $y > 0.05Y$.

2.2 Lateral Movement in the y-Direction

The lateral movement is contingent on the x-coordinate to enforce the loop:

- For $x < \frac{X}{2}$: If $y + 1 < Y$, include the edge $(u, (x, y + 1, z))$
- For $x \geq \frac{X}{2}$: If $y - 1 \geq 0$, include the edge $(u, (x, y - 1, z))$

2.3 Elevation Movement in the z-Direction

- Upward Movement: If $z + 1 < Z$, include the edge $(u, (x, y, z + 1))$

- Downward Movement: If $z - 1 \geq 0$, include the edge $(u, (x, y, z - 1))$

Thus, the edge set is formally given by: $E = \{(u, v) \mid u = (x, y, z) \in V, v$ is generated as above and $v \in V\}$

This scheme guarantees a circular track:
1. For $x < \frac{X}{2}$, racers advance in the y-direction toward increasing values.
2. For $x \geq \frac{X}{2}$, the lateral movement reverses, directing racers toward decreasing y values.
3. The combination of these directed moves mimics a closed-loop circuit without resorting to explicit wrap-around edges.

2.4 Visualization of Mario Kart Race Track Graph Flow

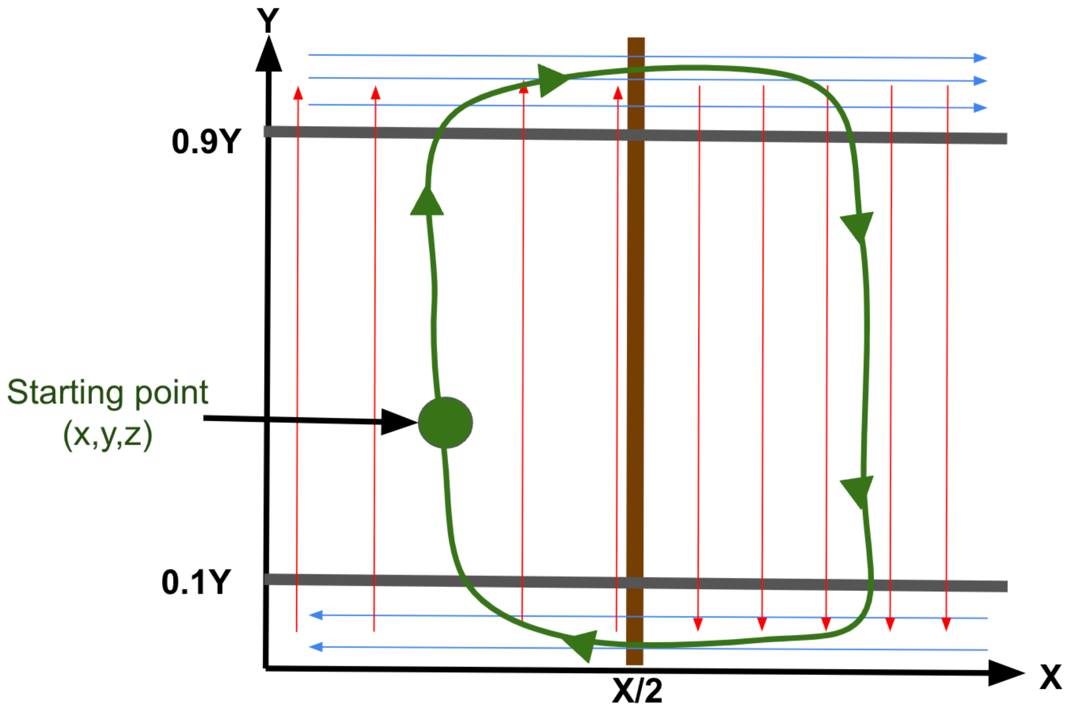

Figure 1: 2D Visualization of the edge flow in a Mario Race Track graph.

Figure 1 shows a 2D projection in the (X, Y) coordinate system of how the edge weights flow throughout the Mario Race track. With this representation, we can see how performing a full lap (highlighted by the approximate green path), around the track doesn't allow us to "cheat", where we skip portions of the race track to perform a lap faster.

3. Edge Weights

Each edge $(u, v) \in E$ carries a weight $w(u, v)$ representing the traversal time, defined by: $w(u, v) = \frac{1}{\text{speed}(T(u))}$

The speed depends on the terrain type at u. Example mappings include:
- $T(u) =$ Road: speed $= 1$ yielding $w(u, v) = 1$.
- $T(u) =$ Grass: speed $= 0.5$ yielding $w(u, v) = 2$.
- $T(u) =$ Boost: speed $= 2$ yielding $w(u, v) = 0.5$.
- $T(u) =$ Obstacle: speed $= \frac{2}{3}$ yielding $w(u, v) = 1.5$.
- $T(u) =$ OutOfBounds: speed $= 0.2$ yielding $w(u, v) = 5$.
- $T(u) =$ Mushroom: speed $= 2$ yielding $w(u, v) = 0.5$.

4.1 Graph Construction Complexity

-Time Complexity:
- Vertex Creation: Iterating through all grid cells yields O(XYZ).
- Edge Creation: Since each vertex generates a bounded number of edges (at most 6), this also results in O(XYZ).
- Weight Assignment: Computing the weight for each edge is O(XYZ).

The aggregate time complexity for graph construction is therefore: O(XYZ)

-**Space Complexity:**

- Storage for Vertices: Requires O(XYZ) space.
- Storage for Edges: Similarly, the space requirement is O(XYZ).

Thus, the total space complexity is: O(XYZ)

Baseline methods

Dijkstra

Dijkstra's Algorithm is a single-source shortest path algorithm that guarantees finding the minimum-cost path in a graph with non-negative edge weights. We use it to find the shortest lap time by computing the minimum time to return to the node directly behind the start while ensuring the racer completes the track.

1. Initialize the priority queue (min-heap) with the start node and a distance of 0.
2. Use a dictionary to store the shortest known distance to each node, initialized to infinity except for the start node.
3. Extract the node with the shortest distance from the priority queue.
4. Update the shortest known distance for its neighbors (relaxation step).
5. If the target node is reached, return the computed shortest distance.
6. Repeat until all reachable nodes have been processed.

```
Procedure DijkstraShortestLap:
  Input:
    - graph -> adjacency list representation where each node maps to a list of (neighbor, weight)
pairs
    - startNode -> (x, y, z) representing the start position

  Output:
    - minDistance -> minimum lap time required to reach the node directly behind start

  Initialize priority queue PQ
  PQ.push((0, startNode))  # (distance, node)

  Initialize shortestTime as a dictionary with default value ∞
  shortestTime[startNode] = 0

  Extract x, y, z from startNode
  targetNode = (x, y-1, z)  # Node directly behind the start

  While PQ is not empty:
    (currentTime, currentNode) = PQ.pop()  # Get node with smallest distance

    If currentNode == targetNode:
      Return currentTime  # Found the shortest lap distance

    For each (neighbor, weight) in graph[currentNode]:
      newTime = currentTime + weight

      If newTime < shortestTime[neighbor]:
        shortestTime[neighbor] = newTime
        PQ.push((newTime, neighbor))

  Return ∞  # No valid lap found
```

A*

A* Algorithm improves upon Dijkstra's by using a heuristic function to prioritize the search toward the goal, reducing unnecessary expansions. The heuristic function used is Euclidean distance to the node directly behind the start.

1. Initialize the priority queue (min-heap) with the start node and a cost of 0.
2. Use a dictionary to store the shortest known distance to each node, initialized to infinity except for the start node.
3. Use a heuristic function to estimate the remaining distance to the target.
4. Extract the node with the lowest total estimated cost f(n)=g(n)+h(n) from the priority queue.
   - g(*n*) = known cost from the start node to *n*.
   - h(*n*) = estimated cost from *n* to the target node (heuristic).
5. Update the shortest known distance for neighbors and push them into the queue with their estimated cost.
6. If the target node is reached, return the computed shortest distance.
7. Repeat until all reachable nodes have been processed.

HEURISTIC FUNCTION

```
Procedure Heuristic:
  Input:
    - node -> (x1, y1, z1) representing a vertex in the graph
    - target -> (x2, y2, z2) representing the destination node

  Output:
    - Euclidean distance between node and target

  Return sqrt((x1 - x2)² + (y1 - y2)² + (z1 - z2)²)
```

PSEUDOCODE

```
Procedure AStarShortestLap:
  Input:
    - graph -> adjacency list representation where each node maps to a list of (neighbor, weight)
pairs
    - startNode -> (x, y, z) representing the start position

  Output:
    - minDistance -> minimum lap time required to reach the node directly behind start

  Initialize priority queue PQ
  PQ.push((0, startNode))  # (f-score, node)

  Initialize shortestTime as a dictionary with default value ∞
  shortestTime[startNode] = 0

  Extract x, y, z from startNode
  targetNode = (x, y-1, z)

  While PQ is not empty:
    (_, currentNode) = PQ.pop()  # Get node with lowest estimated cost

    If currentNode == targetNode:
      Return shortestTime[currentNode]  # Found the shortest lap distance

    For each (neighbor, weight) in graph[currentNode]:
      newTime = shortestTime[currentNode] + weight
      If newTime < shortestTime[neighbor]:
        shortestTime[neighbor] = newTime
        fScore = newTime + Heuristic(neighbor, targetNode)
        PQ.push((fScore, neighbor))

  Return ∞  # No valid lap found
```

Improved Linear Time Algorithm

- We know that time complexity of the baseline methods are $O(n \log n)$ time, where $n = V + E$, in which V = # of vertices and E = # of edges. From our previous definition of the graph construction, we know that the Mario Kart track will have minimum dimensions of $100 \times 250 \times 25$, we will have at minimum 625000 vertices in our graph. Our baseline algorithms will run in minimum time $O(n \log(625000)) = O(24.16n)$ time. So if we can make use of some linear algorithm that is of time $O(an)$, where a is some constant, we can develop a faster function than our previous baselines.
- Let's say we made a new intermediate graph G' by adding $l(e) - 1$ many new vertices between each edge (u, v) in our original graph G. So now, instead of having an edge of length $l(e)$ between (u, v), we will have $l(e) - 1$ vertices in between u and v with unweighted edges (or just edge length of 1), meaning there will be $l(e)$ unweighted edges in between u and v. Therefore, the distance between u and v is still $l(e)$, so we maintain the properties of our original graph. Now since every edge has equal weight, we can run BFS on this graph to find the shortest track distance to the end node.
- The BFS algorithm will be more optimal than Dijkstra's or A* since are edge weights are bounded. We also know that the edge weights are bounded to the following numbers $[0.5, 1, 1.5, 2, 5]$ based on the tile types. To deal with non-integer edge weights, we can multiply every edge weight by two, giving us unique edge weights $[1, 2, 3, 4, 10]$ and then form G'. This won't affect the shortest path because the ratio between edge weights is still maintained, such that is a path p is distance d in G, it will have a distance $2d$ in G'.
- Since we want to find the shortest *lap* in our graph, we want to essentially find the minimum distance from our start node to the node directly behind it. So if our start node has coordinates (x, y, z), we will need to find the minimum distance $(x, y - 1, z)$. Based on our graph definition, we know that for the first half of a lap, we only have directed edges in y direction between vertices with a y value one greater, so it will be impossible for us to shortcut from (x, y, z) to $(x, y - 1, z)$, forcing us to traverse the track before reaching the coordinate $(x, y - 1, z)$.
- Let's assume worst case that somehow all of our vertices in the graph were out of bounds tiles, in this case our graph would be of size $O(9(V + E))$, since we would create $10 - 1 = 9$ edges and vertices in the new graph. Even though we are creating this larger graph, BFS will run $O(9(V + E))$, which is faster than the minimum time our baseline algorithm would run in $O(24.16n)$ time. We also know the percentage of each tile before hand, so the actual runtime of this algorithm would be $O(\alpha(V + E))$, where $\alpha < 9$ since only 10% all the vertices have type out-of-bounds.
- In addition, since this algorithm is strictly linear time, the gap between it and the baseline algorithm would increase as we increase the size of our Mario Kart graph, making it a more scalable algorithm.
- We define the pseudocode for this algorithm below:

```
procedure constructBFSGraph:
  Input: edges -> tuples (u, v, w), where u and v are vertices and w is the edge length between them
  Output: G_prime -> hash map where key = vertice and value = neighbors(vertice)

  Initialize G_prime such that it contains all vertices from G but they have no neighbors

  for u, v, w in edges:

    Get u and v from G', set them to G_prime_u and G_prime_v respectively
    currVertex = G_prime_u
    for (int i = 0; i < 2w - 1; i++):
      dummyVertex = Vertex()
      append dummyVertex to the list of neighbors for currVertex
      currVertex = dummyVertex
    append G_prime_u to the list of neighbors for currVertex

    return G_prime

procedure BFSShortestPath
  Input:
    - edges: tuples (u, v, w), where u and v are vertices and w is the edge length between them
    - startNode: node we start at for BFS
  Output:
    - minDistance: minimum lap distance to get back to the start node

  minDistance = infinity
  In G, set all nodes to be unexplored
  bfsQueue = [(startNode, 0)]
  behindNodes = all nodes
  G' = constructBFSGraph(edges)
  x, y, z = startNode.coordinate # Get the location of the start node to find the node behind

  while True:

    currentNode, distance = bfsQueue.deque()

    if currentNode.coordinate == (x, y-1, z):
      minDistance = (distance + 2*edge_length((x,y-1,z), (x,y,z))) / 2
      break

    currentNode.explored = True

    for nei in currentNode.neighbors:
      if nei.explored is False:
        bfsQueue.enqueue((nei, distance + 1))

  return minDistance
```

Time Complexity (Asymptotic Analysis)

In this section, we analyze the asymptotic time complexity of the algorithms used to solve the shortest path problem.

1. Graph Construction Time Complexity

The graph construction process consists of the following major steps:

1.1 Vertex Set Initialization

- Each tile on the 3D grid is mapped to a vertex.
- Given dimensions X, Y, Z, the total number of vertices is: $|V| = O(\text{XYZ})$
- Constructing this set requires iterating over all grid cells, leading to an $\boldsymbol{O(\text{XYZ})}$ time complexity.

1.2 Edge Creation

- Each vertex can have at most six edges (adjacent tiles in x, y, z directions).
- The total number of edges is at most: $|E| = O(\text{XYZ})$
- Constructing these edges also takes $\boldsymbol{O(\text{XYZ})}$ time.

1.3 Weight Assignment

- Edge weights are determined based on the tile type, requiring a constant-time operation for each edge.
- Since there are $O(\text{XYZ})$ edges, weight assignment takes $\boldsymbol{O(\text{XYZ})}$ time.

1.4 Final Complexity for Graph Construction
Since all operations run in $O(\text{XYZ})$ time, the total time complexity for constructing the graph is:

$$\boldsymbol{O(\text{XYZ})}$$

2. Baseline Algorithm Time Complexity
We analyze the time complexity of Dijkstra's Algorithm and A∗ when applied to the constructed graph.

2.1 Dijkstra's Algorithm Complexity
- Priority Queue Operations: Dijkstra's algorithm operates on a graph with V vertices and E edges. Using a binary heap, the complexity is: $\boldsymbol{O((\text{V+E})\log\text{V})}$
- Substituting $V = O(\text{XYZ})$ and $E = O(\text{XYZ})$:

$$\boldsymbol{O(\text{XYZ}\log(\text{XYZ}))}$$

2.2 A^* Algorithm Complexity
- A^* follows a similar $O((\text{V+E})\log\text{V})$ bound since it is essentially Dijkstra's algorithm with a heuristic.
- The **Euclidean heuristic calculation** takes $O(1)$ per node.
- Thus, A∗ also runs in: $\boldsymbol{O(\text{XYZ}\log(\text{XYZ}))}$.

3. Shortest Path Algorithm Complexity (Graph Expansion + BFS)

3.1 Graph Expansion Complexity
To transform the graph into an unweighted format for BFS:

- Each edge (u, v) of weight w is expanded into w intermediate vertices.
- The new Vertex Set V' and Edge Set E' in G' depend on the edge weights.

Given the tile type distributions:
- **60% normal tiles** (weight 2)
- **5% item tiles** (max weight 3)
- **10% out of bounds tiles** (weight 10)
- **10% grass tiles** (weight 4)
- **15% speed tiles** (weight 1)

The expected expansion factor α is computed as:

α = (0.6(2-1) + 0.05(3-2) + 0.1(10-1) + 0.1(4-1) + 0.15(1-1))

Evaluating the expression:

α = (0.6 + 0.05 + 0.9 + 0.3 + 0) = 1.85

This implies the new graph G' will have at most $\boldsymbol{O(1.85(XYZ))}$ vertices and edges.

Since constant multipliers are ignored in asymptotic notation, the graph expansion complexity simplifies to:

$$\text{O}(\alpha\ (XYZ)) = O(1.85(XYZ)) = O(XYZ)$$

3.2 Breadth-First Search (BFS) Complexity
BFS runs in $\boldsymbol{O(V' + E')}$ time. With $\boldsymbol{V' = O(XYZ)}$ and $\boldsymbol{E' = O(XYZ)}$, we obtain:

$$\boldsymbol{O(XYZ)}$$

Since BFS is optimal for unweighted graphs, it runs faster than Dijkstra's $O(XYZ\log(XYZ))$

Proof of Correctness

<u>Claim 1: The construction of G' preserves the shortest path distances of the original graph G</u>
Proof:

1. Base Case:

Consider a graph **G** with only a single edge (u,v) of weight w.

In **G'**, this edge is replaced by a path of w unweighted edges, each of weight 1, connecting u and v. The distance between u and v in G' is w, which matches with the Graph G.

Thus the base case holds

This can be understood using the figure below:

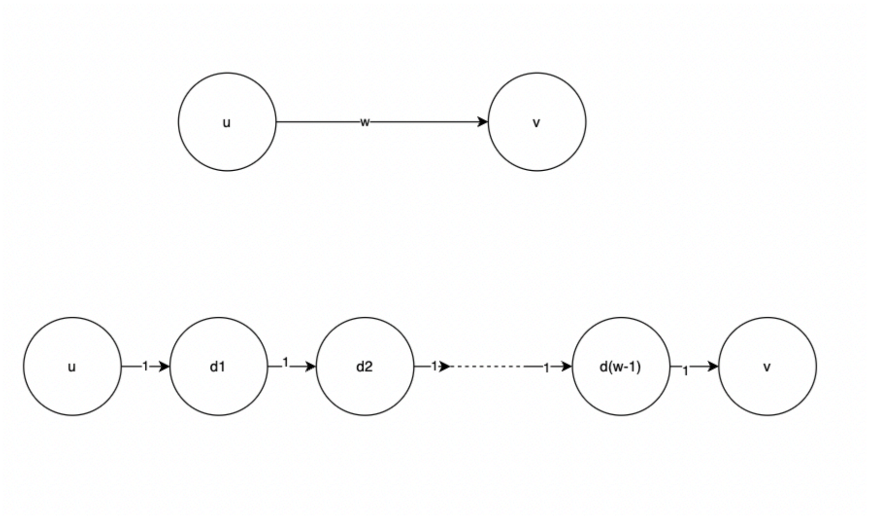

Figure 2: Base Case

2. Inductive Hypothesis

Assume that for a graph G with k edges, the construction of G' preserves the shortest path distances. That is, for any two vertices u and v in G, the shortest path distance between u and v in G is the same as the shortest path distance between u and v in G'.

3. Inductive Step

Consider a graph G with $k + 1$ edges. Let (u, v) be the additional edge with weight w .

By the inductive hypothesis, the shortest path distances for the first k edges are preserved in G' . This means that for any two vertices connected by the first k edges, the shortest path distances in G and G' are the same.

The additional edge (u,v) in G', is replaced with by a path of w unweighted edges.
- Add w−1 dummy vertices between u and v
- Connect u to the first dummy vertex, the dummy vertices to each other, and the last dummy vertex to v, using unweighted edges.

This ensures that the distance between u and v in G' is exactly w, which matches the weight of (u,v) in G

For any pair of vertices x and y in G, the shortest path distance between them in G' must consider the new path introduced by the additional edge (u,v).
- If the shortest path between x and y in G does not use the edge (u,v), then by the inductive hypothesis, the shortest path distance is already preserved in G'.
- If the shortest path between x and y in G does use the edge (u,v), then the distance in G' will include the w unweighed edges corresponding to (u,v). This ensures that the distance in G' matches the distance in G.

4. Conclusion

Since the shortest path distances are preserved for the first k edges (by the inductive hypothesis) and for the additional edge (u,v) (by the construction of G'), the shortest path distances are preserved for k+1 edges.

Thus by induction, the construction of G' preserves the shortest path distances for any graph G>

Claim 2: BFS on G' correctly computes the shortest path from the start node to the node directly behind it.
Breadth-First Search (BFS) is an algorithm that explores a graph level by level, starting from a given node (the "start node"). It explores all neighbors of the current node before moving on to neighbors of neighbors, and so on.

In an unweighted graph, the shortest path between two nodes is the path with the fewest edges. BFS guarantees that the first time a node is visited, it is via the shortest path (in terms of the number of edges). This is because BFS explores nodes in order of increasing distance from the start node.

Proof by Contradiction

Goal:

Assume that BFS does not correctly compute the shortest path from start node s to target node t and show that this leads to a contradiction.

Assumptions:

Suppose BFS does not find the shortest path from s to t. This means:
1. Either BFS does not find a path to t
2. Or, BFS finds a path to t that is not the shortest.

Case 1: BFS does not find any path to t

If BFS does not find a path to t in G'. then it means t is not reachable in G'. However by the construction of G', the distance between s and t is preserved form G. This claim was proved in Claim-1. Since t is reachable in G, it should also be reachable in G'.

This is a contradiction to the claim that BFS does not find a path to t.

Therefore, BFS will find a path to t

Case 2: BFS finds a path to t that is not the shortest.

Suppose BFS finds a path P from s to t with length L, but there exists a shorter path P' with length L' with $L' < L$.

But we know that in the property of BFS, it explores nodes level by level. This means it would have encountered t via path P' before encountering it via P. This is because P' has fewer edges compared to P as $L' < L$, so it would have explored t at an earlier level.

This contradicts the assumption that BFS finds P instead P' first.

Conclusion

It is clear that both the assumptions have been proved wrong. Therefore, BFS correctly computes the shortest path from s to t

Conclusions from Claim 1 and Claim 2:
From the above two claims we can say that - the shortest path obtained in our original graph G will be the same as the BFS graph G'

Claim 3: The algorithm correctly handles non-integer edge weights by scaling
Scaling Edge Weights:

The original edge weights are scaled by a factor of 2 to convert them into integers. This scaling preserves the ratio between edge weights, ensuring that the shortest path in G corresponds to the shortest path in G'.

Correctness of Shortest Path:

Let p be a path in G with distance d. After scaling, the distance of p in G' is 2d. Since all edge weights are scaled uniformly, the relative distances between paths are preserved. Thus, the shortest path in G remains the shortest path in G'.

Conclusion:

The algorithm correctly handles non-integer edge weights by scaling.

Claim 4: The algorithm forces traversal of the track before reaching the node directly behind the start node
Proof:

1. Graph Structure

The graph is constructed such that in the first half of the lap, all edges are directed in the y-direction. This means:
- You can only move forward along the y-axis ie. from (x, y, z) to $(x, y + 1, z)$
- You cannot move backward along the y-axis ie. from (x, y, z) to $(x, y - 1, z)$ in the first half of the lap.

This structure ensures that you must traverse the entire first half of the lap before you can reach the node directly behind the start node $(x, y - 1, z)$. It prevents you from taking a "shortcut" or looping back to $(x, y - 1, z)$ without completing the first half of the lap.

2. BFS Exploration

BFS explores nodes level by level, starting from the start node (x, y, z). It explores all neighbors of the current node before moving on to neighbors of neighbors, and so on.

Since the graph has directed edges in the y-direction in the first half of the lap, BFS will only explore nodes in the forward y-direction (eg. $y \to y + 1$) during the first half of the lap.

BFS cannot explore backward (eg. $y \to y - 1$) because there are no backward edges in the first half of the lap.

Conclusion

- The graph structure enforces that you must traverse the entire first half of the lap before reaching the node directly behind the start node $(x, y - 1, z)$
- BFS respects this structure by exploring nodes level by level and only moving forward along the y-axis in the first half of the lap.
- Therefore, the algorithm forces traversal of the track before reaching the node directly behind the start node.