# CSCI 532 - Final Project: Algorithm to Implement “Ticket to Ride” Board Game

Brittany Boles and Redempta Manzi

December 9, 2024

## 1 Introduction

Over the years, a variety of algorithms have been explored to solve different board games, highlighting the complexity and strategic depth they offer [1, 2, 3]. Several games have been extensively studied, with strategies focusing on optimizing moves, analyzing patterns, and planning long-term plays. Modern board games such as *Ticket to Ride* [4] present unique challenges, as players must manage resources, optimize routes, and anticipate opponents' strategies. Techniques such as route planning and heuristic-based methods are often employed to evaluate the best moves, striking a balance between completing personal objectives and disrupting opponents' plans.

Ticket to Ride [4] is a popular board game created by Alan R. Moon[1] and realed in 2004. This game combines strategy, planning, and a touch of luck, appealing to players of all ages. It challenges players to build train routes across a map, connecting cities to complete specific destination tickets for points. Players must balance competing objectives, such as completing their routes, blocking opponents, and maximizing their resources' efficiency, including train cards and limited route spaces. The straightforward mechanics of the game, including the drawing of cards, the claiming of routes, and the drawing of new tickets, make it easy to learn, but the strategic depth ensures replayability. An algorithm such as Dijkstra can be used to efficiently solve some game's challenges, such as determining the shortest or most efficient route between cities by treating cities as nodes and route lengths as edge weights.

Dijkstra's algorithm [5] is a widely-used graph traversal technique for finding the shortest path between a starting node and all other nodes in a weighted graph. The algorithm assumes non-negative edge weights and iteratively explores the closest unvisited node, updating the shortest known distances to neighboring nodes. It maintains a priority queue to efficiently select the next node with the smallest tentative distance, ensuring that it will not be revisited once a node's shortest path is determined. Dijkstra's algorithm has applications in diverse fields, including routing in networks, navigation systems, and path optimization in games, making it a foundational tool in computer science and optimization. One of the many applications of this algorithm is Ticket to Ride.

In Ticket to Ride, Dijkstra's algorithm can enhance route optimization by identifying the shortest path between cities on the board. In this context, the algorithm treats cities as nodes and the routes connecting them as weighted edges, with weights representing the number of trains required to claim a route. Dijkstra's algorithm calculates the minimum ”cost” path to complete a specific route, ensuring that players use the least resources while connecting their target cities. This approach can also aid in strategic planning, such as prioritizing efficient routes to block opponents or adapting to changing board states. Implementing Dijkstra's algorithm provides players with a systematic way to maximize their score while minimizing resource expenditure. This short report outlines how Dijkstra's algorithm can enhance route optimization by identifying the shortest paths between cities on the board.

## 2 Related Work

Different strategies have been explored to improve winning strategies in ticket to ride game including *Graph Theory and Probability* [6]. This study examines mathematical interpretations of the game

[1]Ticket to ride author: `https://en.wikipedia.org/wiki/Alan_R._Moon`

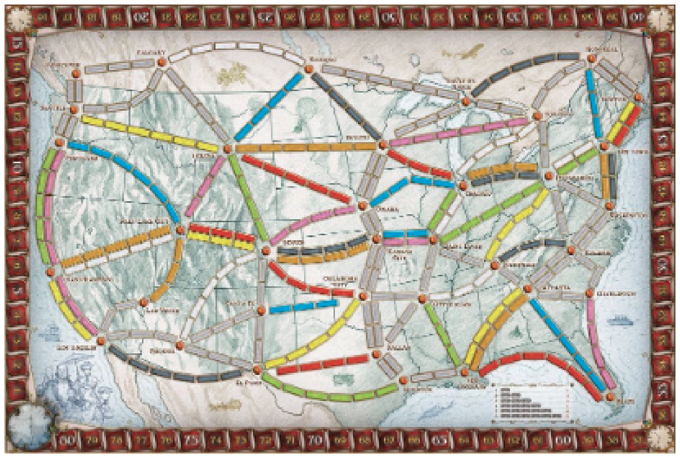


Figure 1: Ticket to Ride - US cities [6]

Ticket to Ride to improve player strategies and create a fairer scoring system. Researchers calculate the expected time for specific moves, arguing that rewards should reflect this time. They utilize effective resistance from electrical circuit theory and linear regression to analyze city pair connections. By applying two algorithms to the game board, they propose optimal strategies for city connections. The study compares these measures with simulated game outcomes to enhance gameplay fairness and strategy.

Various studies have explored Dijkstra's algorithm, including "shortest Distance between Building" [7]. In this paper, researchers focused on applying Dijkstra's algorithm to determine the shortest paths between buildings on a college campus. The implementation begins by extracting the coordinates of all campus buildings using Google Maps. With these coordinates, a custom script calculates the distances between the buildings. These distances are then used to construct a rough graph representing the buildings' spatial relationships. This graph generates an adjacency matrix, which forms the basis for creating a weighted undirected graph using an online graph creation tool. The resulting graph allows for efficient computation of the shortest paths between any two buildings, demonstrating the practical utility of Dijkstra's algorithm in real-world navigation scenarios.

## 3 Algorithm Implementation

The implementation of To effectively implement the algorithm '

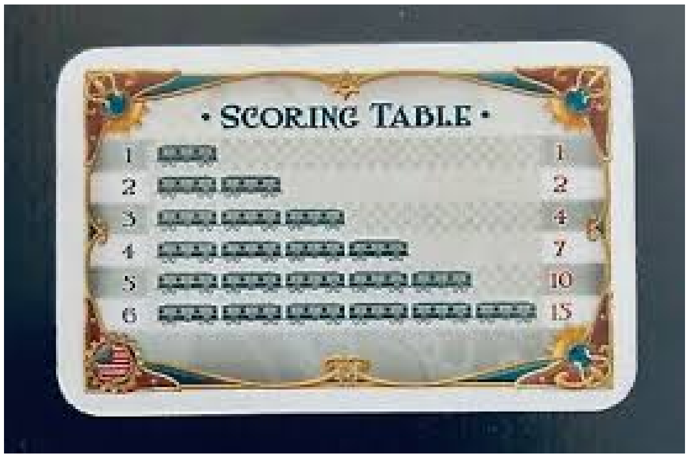
### 3.1 How to play "Ticket to Ride"

"Ticket to Ride"

### 3.2 Assumptions

To simulate the game, we began by making several assumptions and simplifications. These simplifications included assuming that each player had all the cards needed to lay their trains at any given moment. In the actual game, players typically draw one or two cards per turn, choosing from five face-up cards, the draw pile, or a combination of both. If the face-up cards include a "rainbow" card, players are limited to taking only one; otherwise, they can take two. Simulating this complex card-drawing mechanism proved challenging. The game's strategic depth further complicates matters, with players often debating whether to stockpile cards for future moves or immediately claim routes to avoid losing them to opponents. This creates a waiting game driven by luck, as players strive to collect the right colors of cards to connect cities. Since drawing cards is an action and actions can only occur once per turn, simulating the decision-making process for when to draw versus when to lay tracks became a larger issue than the shortest-path calculations we initially aimed to address. As a result, we chose to exclude this game rule from our simulation.

Additionally, we assumed players would draw and keep only one route at a time, even though in the actual game they draw three routes and choose to keep between one and three. Thirdly, we ignored longest train, an additional way to earn points, to keep it simple as we weren't basing any of the strategies off this card. Finally, we simplified further by assuming that players were not actively sabotaging each other's progress. As Redempta can now attest after playing Brittany, this is a ruthless, yet affective technique to help win the game. After multiple games, your opponents might be able to start predicting what your route is as you try to connect it. This is another factor we were unable to incorporate into a simulation.



• SCORING TABLE •

| | |
|---|---|
| 1 | 1 |
| 2 | 2 |
| 3 | 4 |
| 4 | 7 |
| 5 | 10 |
| 6 | 15 |

Figure 2: Scoring system, Image source: https://www.ebay.com/itm/254748377198 [6]

## 3.3 Implementation

To implement Dijkstra's algorithm for a Ticket to Ride path, the first step was to represent the game board as an undirected graph. Translating the board required us to simplify certain elements, such as ignoring the colors of trains needed for each path and focusing solely on the total cost, represented by the number of trains required to connect neighboring cities. For those unfamiliar with the game, it's worth noting that the board often includes multiple connections between the same pair of cities, enabling different players to claim parallel routes. These multiple paths were preserved in our graph representation to maintain this aspect of the game. To construct the graph, we manually entered the nodes corresponding to cities and the edges representing connections between them.

Given a board and route, we can implement an algorithm to find the path using the fewest trains. We noticed that because it's an undirected graph, it doesn't matter which city we start from in the route; the optimal path will be the same for both. We start with an empty priority queue that will be the shortest path from our source city for all cities. We also assign distance values to all cities starting at infinity. We then traverse through the cities, and if the distance of a city is reduced, we add another instance of a city in `priority_queue`. Even if there are multiple instances, we only consider the instance with minimum distance and ignore others. Using a priority queue reduces the time complexity from O($V^2$) to O((E+V)Log(V)).

Given the base Dijsktra algorithm, we simulated gameplay between two players. The two players took turns. We calculate the shortest path between the two cities in each player's turn in their route. We then figure out what track to lay next. We start at the origin city, and as we continue to play, we look for the next edge that's not weighted zero in our path. If an edge is weighted as zero, we have already placed a track there. This brings us to the next step; after figuring out what track is next, we need to actually "play" the track. To do this, we need a board for each player. We update the player's board to have an edge weight of zero, and the opponent's board to have an edge weight of infinity where we want to lay the track. We modified Dijkstra's algorithm to ignore edges with a cost of infinity, so it removes these tracks from the opponent's board. We then go back and forth between the players until one player uses all the trains assigned at the beginning. This is how the game is decided to be over in the game. This algorithm ends up taking O(PK(E+V)Log(V)), where K is the number of rounds that happen, and p is the number of players. Something of note is that when the

**Algorithm 1** Shortest Path Calculation using Dijkstra's Algorithm

```
function SHORTEST_PATH(board, routes, cities)
    $start\_node \leftarrow routes[$"city1"$]$
    $end\_node \leftarrow routes[$"city2"$]$
    Initialize:
    $distances \leftarrow \{city : \infty$ for each $city \in cities\}$
    $previous \leftarrow \{city :$ None for each $city \in cities\}$
    $distances[start\_node] \leftarrow 0$
    $priority\_queue \leftarrow [(0, start\_node)]$                ▷ (distance, city)
    while $priority\_queue$ is not empty do
        $(current\_distance, current\_city) \leftarrow$ HeapPop($priority\_queue$)
        if $current\_distance > distances[current\_city]$ then
            continue
        end if
        for all $neighbor \in cities$ do
            $connections \leftarrow board[current\_city, neighbor]$
            for all $connection \in connections$ do
                if $connection < \infty$ then                ▷ Only valid connections
                    $new\_distance \leftarrow current\_distance + connection$
                    if $new\_distance < distances[neighbor]$ then
                        $distances[neighbor] \leftarrow new\_distance$
                        $previous[neighbor] \leftarrow current\_city$
                        HeapPush($priority\_queue$, $(new\_distance, neighbor)$)
                    end if
                end if
            end for
        end for
    end while
    Reconstruct Path:
    $path \leftarrow []$
    $current \leftarrow end\_node$
    while $current \neq$ None do
        Append $current$ to $path$
        $current \leftarrow previous[current]$
    end while
    Reverse $path$
    if length($path$) $< 2$ or $distances[end\_node] = \infty$ then
        return (None, [])                ▷ No valid path
    end if
    return $(distances, path)$
end function
```

edge weights between the cities is 1, the number of rounds K equals the number of trains allotted to each player. Also, K is less than the number of edges in our version because, with each turn, a player lays down a track. In the actual game, players' turns can be used to draw train cards instead of playing their roads.

We keep track of points and tally them at the end to determine which player wins. Each time a player lays a track, we calculate the points earned by mapping the number of tracks to the associated points 2. We keep a running total of the number of points each player has. We also update points based on the completion or, in some cases, the inability to complete routes. The rules state that if a player finishes a route, they get X number of points; however, if a player is unable to finish the routes, they lose that many points. As the game continues, if a player completes one route, they draw another and continue trying to finish it. When a player runs out of tracks, we tally up all the points achieved so far.

We should note that in the real game, a player gets anywhere from one to three routes each time they "draw" new routes. We chose to consider only one at a time for simplicity. We felt this was valid as well because not completing a route has high consequences, so an acceptable strategy is to finish one at a time.

Originally, we chose the strategy of fewest tracks because, in the real game, playing tracks is costly. However, we implemented another strategy as well. This strategy minimized the number of turns it took to reach a city. Connecting two cities is one action. Only one action is allowed per turn. Since there is only one action per turn, we hypothesized that the number of turns would be more important than the number of tracks. To see how the two strategies compared in our simulation, we, in a two-player game, first tested whether going first had any benefit. We had two players find the minimum number of trains to place between routes and compared the number of wins out of 500. We found that player 1 won 552, player 2 won 522 times, and they tied 26 times. Next, we compared the traditional shortest path versus the shortest number of turns. We changed the weight of all the edges to one to update the algorithm to find the shortest number of turns. This made it so we visited the fewest cities. Player 1 played traditionally, and Player 2 played the shortest number of turns. Over 500 games, player 1 won 80 times, player 2 won 417 times, and they tied 3 times. Given our assumptions of unlimited resources to play trains, it was a better strategy to minimize the turns of trains.

An unintended consequence of choosing routes that minimize turns is that laying longer, more consecutive routes can get you more points in this game. This is due to the scoring schema for track placement (Fig. 2). Choosing the longer routes is also often a consequence of trying to minimize city visits. This, combined with being able to complete more routes before the game ends, leads to a winning strategy for our simulation. However, the number of tracks is important in the actual game. In our assumptions, we assume we have all the cards; in the game, though, we need color cards of the same type to pay for the tracks. Drawing cards is also an action, so minimizing tracks might also help minimize turns. Other complexities of the real game, such as sabotage from other players and drawing multiple routes, will likely change the winning strategy.

It is no surprise that adding more players adds to the time complexity. At the beginning of each player's turn, we have to recalculate the shortest path, then make a move and update the board. The time complexity thus becomes O(PK(E+V)log(V)), where P is the number of players and k is the number of rounds played. Note that this holds true even if the board changes and is expanded or shrunk. The number of rounds is bounded where K is less than or equal to the minimum of either edges divided by players or number of trains per person. Clearly (given our assumption that each turn a player places tracks), there cannot be more rounds than the number of edges a player can place.Further, we cannot possibly have more rounds than a player has trains, as that is when the game is over. In the case where each edge weight is 1, the number of rounds K is equal to exactly N. Linear time (O(V)) is added as well to update the board, trace back the path, find the next move in the path and keep tally of the score. However, Dijkstra still accounts for the time complexity as it is the highest cost. The storage costs increase with the number of players, as we need P copies of the board.

### 3.4 Future Work

While our simulation provides valuable insights into potential winning strategies, we also made many assumptions that reduce the reliability of our strategies. One example is adding the longest train into the simulation. After the game is finished, implementing this would involve using a variation of depth-first search to determine which player had the longest consecutive path. We would then add the points to which player had the longest path.

Further research could examine assisting decision-making in the game. Given 3 routes, how many should you keep? When should you draw cards rather than play? We could simulate many examples, perhaps incorporating machine learning to better understand what strategies win with the full set of game rules.

## 4 Conclusion

In this report, we demonstrate the application of Dijkstra's algorithm in the Ticket to Ride board game. The game is represented as an undirected graph, where the nodes are cities, and the edges represent the number of trains required to travel from one city to another. The game's goal is to gather as many points as possible by connecting cities, maximizing the number of trains used, and successfully linking the cities indicated on the players' cards. Since Dijkstra's algorithm is primarily used to calculate the shortest path from one node to another, our main objective in this experiment was to explore how implementing Dijkstra's algorithm could help identify the shortest routes between the cities specified on the cards.

We found that the implementation of this algorithm was effective. In a two-player setup (which we designed the game to support), we could see who was better positioned to connect the cities with a shorter path. However, we recognize there may be better strategies than using Dijkstra's algorithm for maximizing points in the game, as there are several ways to gain points beyond merely connecting the cities indicated on the cards.

## References

[1] W.-C. Chang, Y.-D. Chiu, and M.-F. Li, "Learning kruskal's algorithm, prim's algorithm and dijkstra's algorithm by board game," in *Advances in Web Based Learning-ICWL 2008: 7th International Conference, Jinhua, China, August 20-22, 2008. Proceedings 7*. Springer, 2008, pp. 275–284.

[2] M. H. Winands, "Monte-carlo tree search in board games," in *Handbook of Digital Games and Entertainment Technologies*. Springer, 2017, pp. 47–76.

[3] C.-C. Chang, G.-D. Su, and C.-C. Lin, "An improved sudoku-based data hiding scheme using greedy method," *International Journal of Computational Science and Engineering*, vol. 23, no. 1, pp. 10–21, 2020.

[4] L. M. Traub, "Tickets to ride: selecting cargo for clathrin-regulated internalization," *Nature reviews Molecular cell biology*, vol. 10, no. 9, pp. 583–596, 2009.

[5] A. Javaid, "Understanding dijkstra's algorithm," *Available at SSRN 2340905*, 2013.

[6] R. T. Witter and A. Lyford, "Applications of graph theory and probability in the board game ticket to ride," in *Proceedings of the 15th International Conference on the Foundations of Digital Games*, 2020, pp. 1–4.

[7] G. Deepa, P. Kumar, A. Manimaran, K. Rajakumar, and V. Krishnamoorthy, "Dijkstra algorithm application: Shortest distance between buildings," *International Journal of Engineering & Technology*, vol. 7, no. 4.10, p. 974, 2018.