

Maze Pathfinding Implementation

Advanced Software Engineering (ACCB5011)

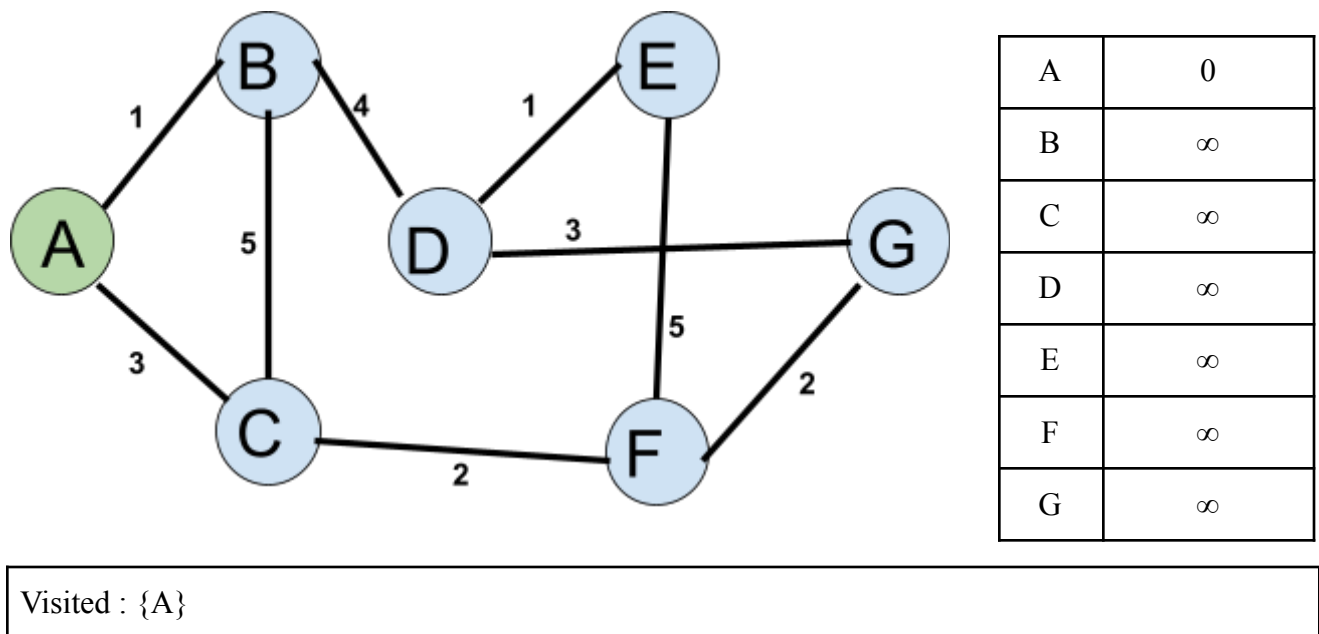
Jovanna De La Trinidad (2504811)

Dijkstra's Algorithm

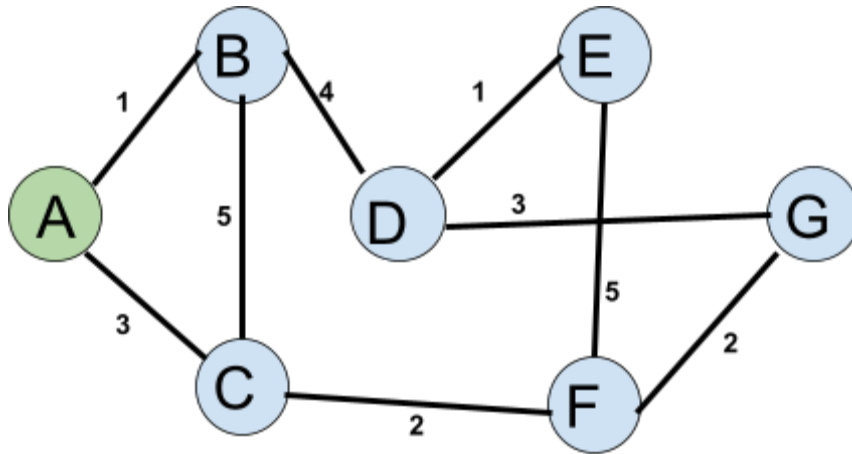
Dijkstra's algorithm is used to find the shortest path within a weighted undirected graph. The example below helps illustrate how the algorithm works.

The algorithm starts by selecting a starting node, in this case node A. A visited set is created to keep track of the nodes already processed so those nodes are not visited more than once. A distance map is also created to store the shortest known distance from the starting node to every other node in the graph(GeeksforGeeks, 2012).

Initially, all nodes are set to infinity, except for the starting node A, which is set to 0. Node A is then added to the visited set.



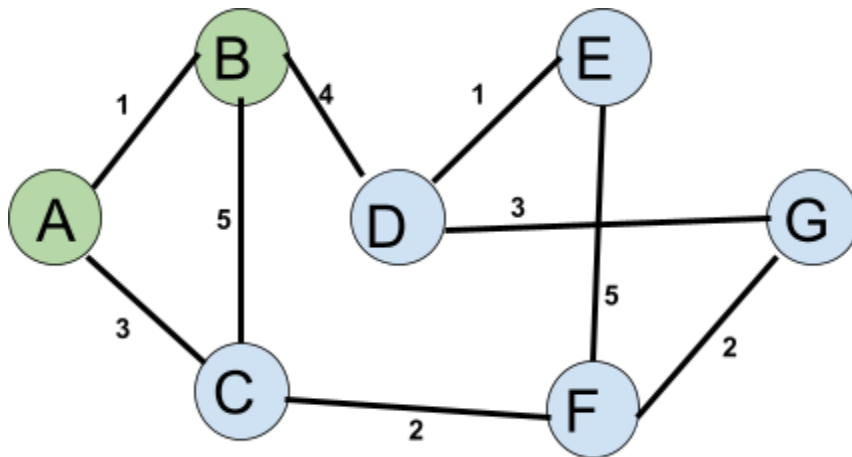
At Node A, the algorithm looks at its neighbors, B and C, and updates their distances based on the edge weights from A.



A	0
B	1
C	3
D	∞
E	∞
F	∞
G	∞

Visited : {A}

After analyzing the distance map, the algorithm selects the unvisited node with the smallest distance value. In this case, node B has the shortest distance, so B is then added to the visited set.

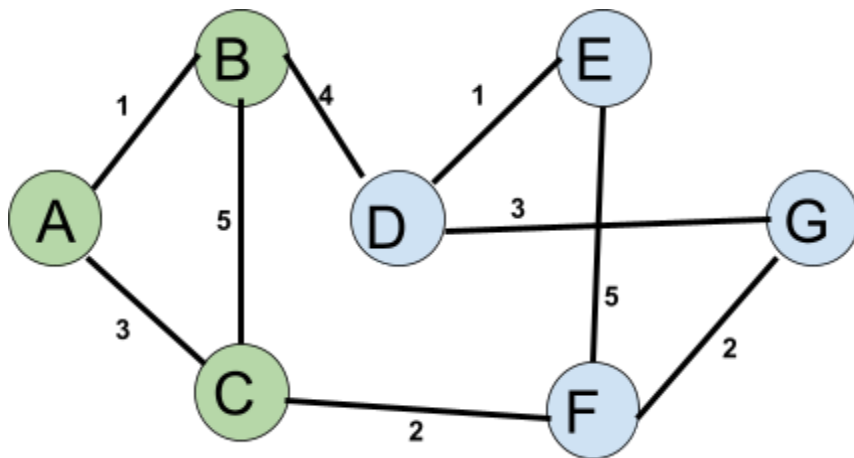


A	0
B	1
C	3
D	∞
E	∞
F	∞
G	∞

Visited : {A,B}

The algorithm then examines B's neighbors and updates their distances if a shorter path is found. For example, from B to C the weight is 5, but we already have the value 3 in node C from

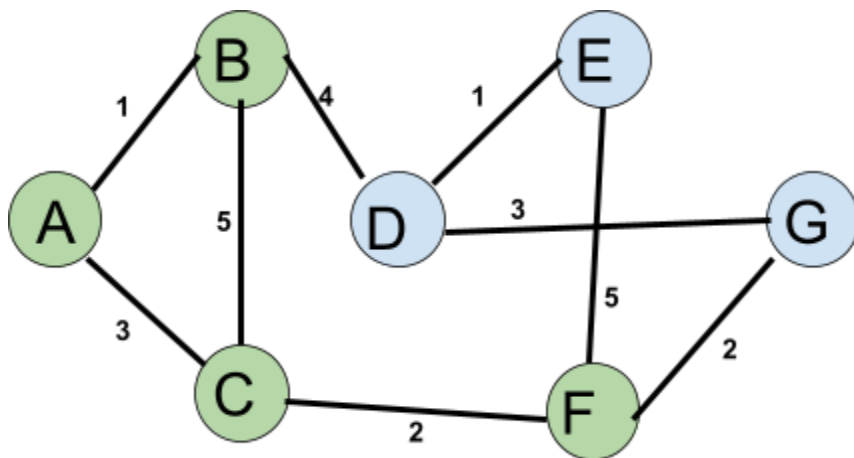
previously the distance from A to C so we leave as is. After analyzing the distance map, C is added to visited.



A	0
B	1
C	3
D	4
E	∞
F	∞
G	∞

Visited : {A,B,C}

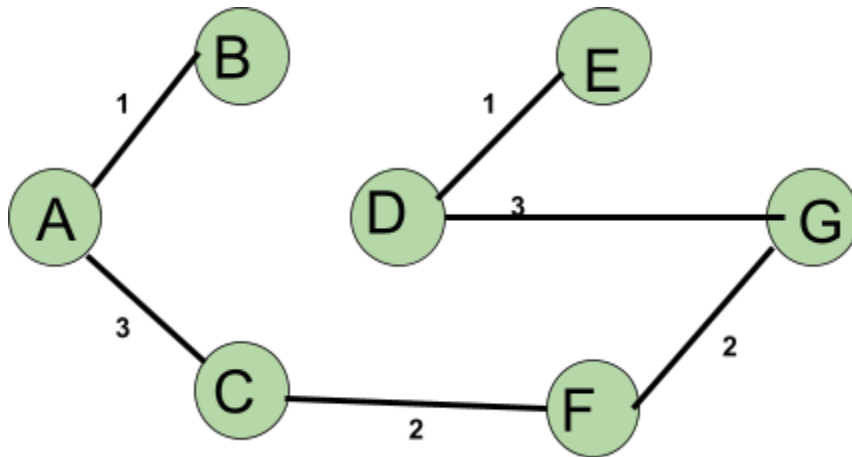
Repeating the process we update the distance map, setting F to 2. F has the smallest distances of an unvisited node and is therefore selected and added to the visited set.



A	0
B	1
C	3
D	4
E	∞
F	2
G	∞

Visited : {A,B,C,F}

After repeating the process until every node is visited we get the final graph below.



In the dijkstra algorithm, a priority queue is used to keep track of the node with the smallest current distance (GeeksforGeeks, 2012). Below is the pseudocode of Dijkstra's algorithm using a priority queue (GeeksforGeeks, 2012).

- A priority queue is created that stores a pair of distance and node
- A distance map is created for each node and initializes all distance value to infinity
- The start node is then set to a distance of zero and pushed into the queue
- While the queue is not empty the node with the smallest distance is popped called curr
- If a shorter path of curr neighbors is found, it is updated into distance and pushed into the queue
- This while loop is continued until all nodes are in visited

A* Algorithm

A* algorithm is a pathfinding algorithm that approximates the shortest path (GeeksforGeeks, 2016). It is one of the best and popular techniques used in path finding and graph traversals (GeeksforGeeks, 2016). This algorithm uses a heuristic approach into finding its path.

A* selects nodes based on the value : $f = g + h$

- g represents the cost to move from the starting point to given square on the grid (GeeksforGeeks, 2016)
- h represents the estimated cost from current node to target, calculated heuristically (GeeksforGeeks, 2016)

Common heuristic methods include Manhattan, diagonal, and Euclidean (GeeksforGeeks, 2016).

The algorithm begins by initializing an open list and a closed list. The starting node is placed in the open list with an initial cost of zero. While the open list is not empty, the node with the lowest f value is selected and removed from the open list.

The algorithm then generates all valid successors of this node and calculates their g, h, and f values. If a successor is already in the open or closed list with a lower f value, it is skipped.

Otherwise, it is added to the open list. The current node is then moved to the closed list.

This process continues until the target node is reached.

Difference Between Dijkstra's Algorithm and A* Search Algorithm

Dijkstra's algorithm uses a greedy approach and guarantees the shortest path to all nodes in the graph. However, it can be slower with a larger graph size (GeeksforGeeks, 2024).

A* uses a heuristic-based approach, which makes it faster and more efficient for pathfinding toward a specific target. While A* typically finds the shortest path it does not guarantee to find the shortest path (GeeksforGeeks, 2024).

Maze Automatic Implementation Process

BFS

For the automatic maze solver, Breadth-First Search (BFS) was chosen. When solving mazes, BFS or Depth First Search (DFS) are commonly used, but the use of BFS guarantees the shortest path in an unweighted grid (Luthfisauqi, 2024).

In the assignment description, the player always starts at the bottom-left of the grid and the target always starts at the top-right. In this 20x20 grid, with a wall barrier, the player always starts at (18,1) and the target is always (1,18).

A queue was created to manage the BFS traversal, along with a visited set to track visited cells using (row, column) pairs. A map called previous stores each cell's previous location, which will later be used to reconstruct the path.

The player's start location is added to the queue and marked as visited. The algorithm then loops while the queue is not empty or until the target is reached. In Figure 1, each cell's neighbor is checked to see if it is in bounds, not a wall, and not visited. If it is valid it is then added to the queue, visited and recorded into previous.

```

// loop through all neighbor of curr
for (const auto& d : ::distance) {
    int dr = d.first;
    int dc = d.second;
    pair<int, int> loc = { curr.first + dr, curr.second + dc }; // get location (row, col) within grid

    // check boundaries
    if (loc.first < 0 || loc.first >= 20 || loc.second < 0 || loc.second >= 20) {
        continue;
    }

    // if the location has already been visited, continue
    if (visited.find(loc) != visited.end()) {
        continue;
    }

    // GetGridSquare(column, row)
    auto cellLoc = grid->GetGridSquare(loc.second, loc.first);
    if (cellLoc == griditem::wall) {
        continue;
    }
    if (cellLoc == griditem::empty || cellLoc == griditem::target) {
        visited.insert(loc);
        previous[loc] = curr;
        q.push(loc);
    }
}

```

Figure 1. Checking Cells Neighbors

Path Reconstruction

After BFS is completed, the previous map is used to reconstruct the path. In Figure 2, the algorithm repeatedly looks up each cell's previous position and adds it to a path vector until the player's starting point is reached.

```

// find shortest path by using previous
std::vector<std::pair<int, int>> path;
auto step = target;
while (step != player) {
    path.push_back(step);
    auto it = previous.find(step);
    if (it == previous.end()) {
        return;
    }
    step = it->second;
}
path.push_back(player);

```

Figure 2. Finding the Shortest Path

This produces an array from target to player. In order to flip it, we utilize the reverse shown in Figure 3 to flip the vector from player to target.

```
// reverse order of path to start from player to target
std::reverse(path.begin(), path.end());
```

Figure 3. Reverse the Path

Player Movement

At this stage, the path exists, but the player does not yet move. The path is only a blueprint of the steps needed to reach the target.

To move the player, a movement function is created to move the player one step at a time based on the difference between current position and the previous positions, either moving the player up, down, left, or right.

```
~moveresults GridNavigatorAutomatic::movement(pair<int, int> curLoc, pair<int, int> prevLoc) {
    int dx = (curLoc.first - prevLoc.first);
    int dy = (curLoc.second - prevLoc.second);
    if (dx == -1 && dy == 0) { // move up
        return grid->MovePlayer(movedirection::up);
    }
    else if (dx == 1 && dy == 0) { //move down
        return grid->MovePlayer(movedirection::down);
    }
    else if (dx == 0 && dy == -1) { //move left
        return grid->MovePlayer(movedirection::left);
    }
    else if (dx == 0 && dy == 1) { // move right
        return grid->MovePlayer(movedirection::right);
    }
}
```

Figure 4. Movement Function

Inside the Navigate function, a loop iterates through the path starting at index $i = 1$ to avoid out of bounds. Each iteration moves the player one step along the path until the target is reached or the movement is complete.

```
// move player
for (int i = 1; i < path.size(); i++) {
    moveresults result = movement(path[i], path[i - 1]);
    if (result == moveresults::complete) {
        return;
    }
    if (grid->CheckComplete()) {
        return;
    }
}
```

Figure 5. Moving Player Loop for Path

References

GeeksforGeeks. (2012, November 25). *Dijkstra's Algorithm to Find Shortest Paths*.

GeeksforGeeks.

<https://www.geeksforgeeks.org/dsa/dijkstras-shortest-path-algorithm-greedy-algo-7/>

GeeksforGeeks. (2016, June 16). *A* Search Algorithm*. GeeksforGeeks.

<https://www.geeksforgeeks.org/dsa/a-search-algorithm/>

GeeksforGeeks. (2024, June 21). *Difference between Dijkstra's Algorithm and A* Search Algorithm*. GeeksforGeeks.

<https://www.geeksforgeeks.org/dsa/difference-between-dijkstras-algorithm-and-a-search-algorithm/>

Luthfisauqi. (2024, March 1). *Artificial Intelligence Search Problem: Solve Maze using Breadth First Search (BFS) Algorithm*. Medium.

https://medium.com/@luthfisauqi17_68455/artificial-intelligence-search-problem-solve-maze-using-breadth-first-search-bfs-algorithm-255139c6e1a3

Sambol, M. (2014). Dijkstra's algorithm in 3 minutes — Review and example [YouTube Video].

In *YouTube*. https://www.youtube.com/watch?v=_IHSawdgXpI