

Student Name: Shaun James Davies

Student ID Number: 2317135

Programme: Digital Degree Apprenticeship (DDA) Computing -
Software Development

Module: Advanced Software Development

Module Code: ACCB5011

Assignment Title: Maze Pathfinding Implementation.

Submission Deadline: 22/04/2026

*excluding abstract, figures, appendices and other sections as per the IEEE referencing standard.

Contents

Introduction	3
Report.....	3
A* & Dijkstra's Algorithm	3
Reflection	5
Implementation	8
Conclusion.....	29
References	29
Appendix 1	31

Introduction

Fundamental to computational problem solving, pathfinding is concerned with determining an efficient route between two points, or “two vertices”, within an environment which is typically represented by a “graph” [1]. This provides a solution to applicable problems such as GPS navigation, “robotics, logistics and crowd simulation”. Within these environments, it’s likely that the most efficient path to be “static, dynamic or real-time” which increases the complexity [2]. Detailed by Figure 1, a variety of efficient route pathfinding algorithm types exist and are used. Proceeding with the aforementioned techniques, Figure 1 highlights the types pathfinding algorithms namely static, dynamic, time dependant and hierarchical [1].

Discussed in the following text is two popular pathfinding algorithms which is the prioritised emphasis of the following report and implementations. The implementations are analysed to produce efficient node exploration and access to the maze grid.

Report

A* & Dijkstra’s Algorithm

The authors of “A Comprehensive Study on Pathfinding Techniques for Robotics and Video Games” discuss how pathfinding consists of two stages: graph generation and pathfinding algorithms. Graph generation involves modelling a graph of vertices, also known as nodes, and edges, also known as connections. A technique of generating these graphs in industry is “skeletonization” which prepare structures that are suitable for algorithmic search to create the most efficient path whilst “avoiding collisions” and “obstacles” [2]. In summary, pathfinding is a useful and “fundamental component of many important applications”. Alternatively, “video games” are an applicable use of pathfinding “generally represented by a graph” that includes real-time performance constraints. These environments can become large and the pathfinding graphs equally can grow in size and “finding paths can... be a complex task” [3].

Pathfinding is a key aspect to many computing processes such as the problem of finding the shortest path in a grid or maze. Dijkstra’s algorithm and A* are commonly used pathfinding methods in grid or maze navigation.

Dijkstra’s algorithm, known as a “uniform cost search”, developed by Edger Dijkstra with the purpose of finding the shortest path that is most “optimal” in terms of “least weight” to a final location between multiple nodes and their associated weights [4]. Conceptually, the algorithm works through the process highlighted in Figure 2 which shows a typical process of this algorithm. Initially, it sets the distance from the starting node to an infinity large figure to ensure the following distances result less than this value. Every node starts as unvisited and the previous nodes are generated to reconstruct the shortest path later in the code. It proceeds to set the start node distance to 0 and then begins the search from the start node. This algorithm runs until

there is no nodes left to explore or if the goal has been reached. In summary, the code repeatedly selects the nearest unvisited node updating the distances to its neighbouring nodes and continues until the shortest path to the goal is found. Effectively behaving like a priority queue ordered by minimum distance, where the next node is chosen as “the not visited vertex... with the smallest distance” [5]. Unlike A*, exploring all nodes in Dijkstra can increase search time [6]

Applying this to the assignment, the maze is represented as a grid where each cell is a node and movements between adjacent locations form edges.

The A* algorithm, created by Hart et al., is a pathfinding algorithm that finds the shortest path from a start node to a goal node by an informed search algorithm. Differently to Dijkstra's, a cost is generated using two values: the cost from the start to the current node, detailed as “ $g(n)$ ”, and an estimate of the cost from the current node to the goal detailed as “ $h(n)$ ”. These combine to form a “search cost $f(n) = g(n) + h(n)$ ” [7]. An example of this can be found in Figure 3 equation 3 [8]. Traditional A* algorithms may not be sufficient for increasingly complex pathfinding tasks however variants of this algorithm can increase its performance. For example, using a “squared heuristic distance” and “bidirectional A*” can improve searches [9]. Common heuristics in searching use case, such as grid mazes, is the Manhattan distance. This describes an estimate of how far a node is from the goal node using only horizontal, vertical and no diagonal movements. An example of this is in Figure 3 equation 2 [8].

Dijkstra time complexity of $O(|V|^2)$ in its basic implementation where V is the count of the nodes or number of vertices in the graph. Fredman and Tarjan 1984 proposed a Fibonacci heap which reduces the time complexity to $O(|E| + |V|\log|V|)$. [10]

The A* algorithm has the worst case time complexity of $O(b^d)$ where b is the branching factor and d is the depth of the solution. Typically, in graph based representations this can be denoted as $O(E\log V)$ when implemented with a priority queue. The worst case space complexity is $O(b^d)$. As shown in Figure 4 some implementations have a time complexity of A* is $O(E)$ with a space complexity of $O(V)$ however this reflects observed performance rather than theoretical worst behaviour [11].

In summary, Dijkstra's algorithm employs the time complexity of breadth first search and A* employs depth first search using a “heuristic function” to calculate cost [7].

In conclusion, this details pathfinding and its function across industry and two summaries of efficient algorithms such as Dijkstra's and A* algorithm. In comparison, the search strategy of Dijkstra's algorithm is a uninformed uniform cost search exploring locations by distance. Contradictory, the A* is an informed search using heuristics to navigate towards the goal.

Reflection

Basic Dijkstra approach

Initially, the original implementation of a Dijkstra algorithm accessed the grid once costing 400 access points. It is a basic Dijkstra algorithm that requires a full view of the grid before finding the path. This part of the full implementation focused on achieving correctness rather than how the path was generated.

Interestingly, this version of Dijkstra's algorithm uses a full view of the maze once despite the alternative view method being the less costly method.

```
// Read the whole grid once.  
griditem** maze = grid->GetGrid();
```

```
Enter choice:  
1: Manual mode (for learning the environment).  
2: Automatic mode, single run (for testing of your code).  
3: Benchmarking mode, 10,000 runs of automatic mode.  
3  
Minimum accesses: 400  
Mean accesses: 400  
Maximum accesses: 400  
Minimum moves: 34  
Mean moves: 48  
Maximum moves: 88  
Minimum wall hits: 0  
Mean wall hits: 0  
Maximum wall hits: 0  
  
C:\Users\shaun\Documents\Uni\SD\ASDPathfindingAssignment\ASDPathfindingAssignment\Debug\ASDPathfindingAssignment.exe (process 11948) exited with code 0 (0x0).  
Press any key to close this window . . .
```

Dijkstra Probed algorithm

This implementation aimed to improve the already viable basic Dijkstra by using its' skeleton to device a more cost efficient algorithm in terms of accesses. Instead of accessing the grid initially, this implementation used a probing version of Dijkstra that uses GetGridSquare instead of GetGrid. The Dijkstra part remains mostly the same. For example, storing shortest known distances, marking finalised nodes as visited and reconstructing the paths.

A difference is the first basic Dijkstra scans the full maze to find the player and target. The probing version hardcodes the player and target from the maze generation code. An interesting portion of the code is the local cache that was introduced and location of the squares were only accessed when required. This ensures the grid location is stored and reused. This reduces the grid accesses and is the backbone of the probed search. Once the shortest path to the target had been found, the path was reconstructed using the arrays and completed the path without deviation.

```
// Arrays store the state of the probed Dijkstra's algorithm.  
int known[SIZE][SIZE];
```

```

auto ensureKnown = [&](int x, int y)
{
    if (known[y][x] != 0)
    {
        return;
    }

    griditem square = grid->GetGridSquare(x, y);

    if (square == griditem::wall)
    {
        known[y][x] = 1;
    }
    else
    {
        known[y][x] = 2;
    }
};

```

```

Enter choice:
1: Manual mode (for learning the environment).
2: Automatic mode, single run (for testing of your code).
3: Benchmarking mode, 10,000 runs of automatic mode.
3
Minimum accesses: 202
Mean accesses: 289
Maximum accesses: 313
Minimum moves: 34
Mean moves: 48
Maximum moves: 88
Minimum wall hits: 0
Mean wall hits: 0
Maximum wall hits: 0

C:\Users\shaun\Documents\Uni\SD\ASDPATHfindingAssignment\ASDPATHfindingAssignment\Debug\ASDPATHfindingAssignment.exe (process 23660) exited with code 0 (0x0).
Press any key to close this window . . .

```

This produced a more efficient version of Dijkstra.

A* Algorithm with Manhattan Distance

This A* Algorithm was implemented to improve on the above Dijkstra algorithms. This prioritises directed search towards targets rather than following the Dijkstra algorithm's method of exploring cardinalities equally. For this algorithm, implementation of the Manhattan distance heuristic is sufficient for grids restricted to the movement in this scenario. This distance was derived from:

```

auto heuristic = [&](int x, int y)
{
    return abs(targetX - x) + abs(targetY - y);
};

```

Nodes that are closer to the target, dictated by this code, are prioritised. This reduces the number of nodes explored. This guarantees the shortest path, despite the previous reduction.

```
Enter choice:
1: Manual mode (for learning the environment).
2: Automatic mode, single run (for testing of your code).
3: Benchmarking mode, 10,000 runs of automatic mode.
3
Minimum accesses: 400
Mean accesses: 400
Maximum accesses: 400
Minimum moves: 34
Mean moves: 47
Maximum moves: 84
Minimum wall hits: 0
Mean wall hits: 0
Maximum wall hits: 0

C:\Users\shaun\Documents\Uni\SD\ASDPathfindingAssignment\ASDPathfindingAssignment\Debug\ASDPathfindingAssignment.exe (process 25108) exited with code 0 (0x0).
Press any key to close this window . . .|
```

A* Probed with Manhattan Distance

The above A* implementation uses a full grid view prior to exploring. This probed approach was implemented to reduce the amount of grid accesses by instead calling GetGridSquare, like the probed Dijkstra, only when required. The interesting portion of this code is that this performs a maze discovery incrementally instead of viewing the maze in its entirety.

```
auto ensureKnown = [&](int x, int y)
{
    if (known[y][x] != 0)
    {
        return;
    }

    griditem square = grid->GetGridSquare(x, y);

    if (square == griditem::wall)
    {
        known[y][x] = 1;
    }
    else
    {
        known[y][x] = 2;
    }
};
```

```

Enter choice:
1: Manual mode (for learning the environment).
2: Automatic mode, single run (for testing of your code).
3: Benchmarking mode, 10,000 runs of automatic mode.
3
Minimum accesses: 100
Mean accesses: 272
Maximum accesses: 313
Minimum moves: 34
Mean moves: 47
Maximum moves: 94
Minimum wall hits: 0
Mean wall hits: 0
Maximum wall hits: 0

C:\Users\shaun\Documents\Uni\SD\ASDPathfindingAssignment\ASDPathfindingAssignment\Debug\ASDPathfindingAssignment.exe (process 3824) exited with code 0 (0x0).
Press any key to close this window . . .|

```

Summary

Algorithm	Accesses (Min/Mean/Max)	Moves (Min/Mean/Max)	Wall Hits
A* (Probed)	100 / 272 / 313	34 / 47 / 94	0 / 0 / 0
A* (Manhattan Distance)	400 / 400 / 400	34 / 47 / 84	0 / 0 / 0
Dijkstra (Probed)	202 / 289 / 313	34 / 48 / 88	0 / 0 / 0
Dijkstra (Basic)	400 / 400 / 400	34 / 48 / 88	0 / 0 / 0

Implementation

Basic Dijkstra Algorithm

```

// Basic Dijkstra
void GridNavigatorAutomatic::Navigate()
{
    const int SIZE = 20; //Fixed size of the grid as per the provided grid
    code.
    const int INF = 1000000; // A large number representing infinity.

    // Read the whole grid once.
    griditem** maze = grid->GetGrid();

    int startX = -1, startY = -1; // Player's starting position
    int targetX = -1, targetY = -1; // Target's position

    // Find player and target.
    for (int y = 0; y < SIZE; y++) // iterate through the rows
    {
        for (int x = 0; x < SIZE; x++) // iterate through the columns
        {
            if (maze[y][x] == griditem::player) // if the current cell is the
player
            {

```

```

        startX = x; // store the player's x coordinate
        startY = y; // store the player's y coordinate
    }
    else if (maze[y][x] == griditem::target) // if the current cell is
the target
    {
        targetX = x; // store the target's x coordinate
        targetY = y; // store the target's y coordinate
    }
}

// If either the player or target is not found, exit the function.
Basically, if the starting or target positions are invalid, we cannot proceed
with navigation.
if (startX == -1 || startY == -1 || targetX == -1 || targetY == -1)
{
    return;
}

int dist[SIZE][SIZE]; //Stores shortest known distance from the start to
each cell in the grid.
bool visited[SIZE][SIZE]; //Tracks if a cell has been processed in
Dijkstra's algorithm.
int prevX[SIZE][SIZE]; //Previous node tracking
int prevY[SIZE][SIZE]; //Previous node tracking

for (int y = 0; y < SIZE; y++)
{
    for (int x = 0; x < SIZE; x++)
    {
        dist[y][x] = INF; // Initialise all distances to infinity.
        visited[y][x] = false; //No nodes have been visited yet.
        prevX[y][x] = -1; //marks previous coordinates unknown.
        prevY[y][x] = -1; //marks previous coordinates unknown.
    }
}

struct HeapNode //Declares a structure to represent a node in the priority
queue (min-heap) used in Dijkstra's algorithm. Each node contains the x and y
coordinates of a cell in the grid and the distance from the start cell to that
cell.
{
    // Variable declarations
    int x;
    int y;
    int distance;
};

```

```

HeapNode heap[SIZE * SIZE * 4]; // Creates an array to represent the min-
heap to pick the next closest location
int heapSize = 0; // Tracks valid entries in the heap.

auto heapPush = [&](int x, int y, int distance) //Defines a lambda that:
inserts a new square into the heap with its current distance from the start
{
    int index = heapSize; // Chooses the next free position at the end
of the heap
    heap[index].x = x; // Assigns the x coordinate of the new node.
    heap[index].y = y; // Assigns the y coordinate of the new node.
    heap[index].distance = distance; // Assigns the distance
    heapSize++;

    while (index > 0)
    {
        int parent = (index - 1) / 2; // Determines the index of
parent node.

        if (heap[parent].distance <= heap[index].distance) //
Condition: is the parent node's distance less than or equal to the current
node's distance? If yes, the heap property is satisfied, and we can break out
of the loop.
        {
            break;
        }

        HeapNode temp = heap[parent]; // Temporarily store the parent
node.
        heap[parent] = heap[index]; // Moves the current node up to
the parent's position.
        heap[index] = temp; // Moves the parent node down to the
current (child) node's position.

        index = parent;
    }
};

auto heapPop = [&]() -> HeapNode // Defines another lambda.
{
    HeapNode result = heap[0]; // Copies the root node which is the
current minimum distance
    heapSize--; // Decreases the size of the heap.
    heap[0] = heap[heapSize]; // Moves the last node in the heap to the
root position to fill gap left by removal

    int index = 0;

```

```

        // After removing the root of the heap, this loop restores the
min-heap property
        // By repeatedly comparing the current node with its children and
swapping it with the smallest child
        // until it is in the correct position (i.e. smaller than both
children).
        while (true)
        {
            int left = index * 2 + 1;
            int right = index * 2 + 2;
            int smallest = index;

            if (left < heapSize && heap[left].distance <
heap[smallest].distance)
            {
                smallest = left;
            }

            if (right < heapSize && heap[right].distance <
heap[smallest].distance)
            {
                smallest = right;
            }

            if (smallest == index)
            {
                break;
            }

            HeapNode temp = heap[index];
            heap[index] = heap[smallest];
            heap[smallest] = temp;

            index = smallest;
        }
        // Returns the node with the smallest distance (the root of the
heap).
        return result;
    };

    // Setting the start node distance to 0 and pushing it onto the heap to
begin the algorithm.
    // Essentially, this is the starting point of Dijkstra's algorithm and the
player.
    dist[startY][startX] = 0;
    heapPush(startX, startY, 0);
    // Defines movement directions
    int dx[4] = { 0, 0, -1, 1 };
    int dy[4] = { -1, 1, 0, 0 };

```

```

    // Main Dijkstra loop. Processes nodes from the priority queue. Algorithm
    begins expanding from the player's starting position.
    while (heapSize > 0)
    {
        HeapNode current = heapPop();
        int x = current.x;
        int y = current.y;
        // Skip visited nodes.
        if (visited[y][x])
        {
            continue;
        }
        // Marks current node visited.
        visited[y][x] = true;
        // Stops algorithm early once the target has been reached as shortest
        path has been found.
        if (x == targetX && y == targetY)
        {
            break;
        }
        // Explores nodes to attempt to find a shorter path.
        for (int i = 0; i < 4; i++)
        {
            int nx = x + dx[i];
            int ny = y + dy[i];
            // Skip out of bounds nodes.
            if (nx < 0 || nx >= SIZE || ny < 0 || ny >= SIZE)
            {
                continue;
            }
            // Skips walls
            if (maze[ny][nx] == griditem::wall)
            {
                continue;
            }

            int newDistance = dist[y][x] + 1;
            // If shorter path to the node has been found, add to the heap.
            if (newDistance < dist[ny][nx])
            {
                dist[ny][nx] = newDistance;
                prevX[ny][nx] = x;
                prevY[ny][nx] = y;
                heapPush(nx, ny, newDistance);
            }
        }
    }
}

```

```

    // Target is unreachable, distance is still infinity. No valid path
    exists, so exit the function.
    if (dist[targetY][targetX] == INF)
    {
        return;
    }
    //Prepares to reconstruct the shortest path.
    movedirection path[SIZE * SIZE];
    int pathLength = 0;

    int currentX = targetX;
    int currentY = targetY;
    // Reconstructs the path from the target back to the start using arrays.
    while (!(currentX == startX && currentY == startY))
    {
        int parentX = prevX[currentY][currentX];
        int parentY = prevY[currentY][currentX];

        if (parentX == -1 || parentY == -1)
        {
            return;
        }

        if (currentX == parentX && currentY == parentY - 1)
        {
            path[pathLength] = movedirection::up;
        }
        else if (currentX == parentX && currentY == parentY + 1)
        {
            path[pathLength] = movedirection::down;
        }
        else if (currentX == parentX - 1 && currentY == parentY)
        {
            path[pathLength] = movedirection::left;
        }
        else if (currentX == parentX + 1 && currentY == parentY)
        {
            path[pathLength] = movedirection::right;
        }

        pathLength++;
        currentX = parentX;
        currentY = parentY;
    }
    // Executes the moves in reverse order to navigate from the start to the
    target.
    for (int i = pathLength - 1; i >= 0; i--)
    {

```

```

        moveresults mr = grid->MovePlayer(path[i]);
        // If the target is reached, exit the loop.
        if (mr == moveresults::complete)
        {
            break;
        }
    }
}

```

Probed Dijkstra Algorithm

```

//Dijkstra probed
void GridNavigatorAutomatic::Navigate()
{
    const int SIZE = 20;
    const int INF = 1000000;

    // From the provided Grid code:
    // player starts at (1, 18)
    // target starts at (18, 1)
    const int startX = 1;
    const int startY = 18;
    const int targetX = 18;
    const int targetY = 1;

    // Arrays store the state of the probed Dijkstra's algorithm.
    int known[SIZE][SIZE];

    int dist[SIZE][SIZE];
    bool visited[SIZE][SIZE];
    int prevX[SIZE][SIZE];
    int prevY[SIZE][SIZE];
    // Initialises the arrays
    for (int y = 0; y < SIZE; y++)
    {
        for (int x = 0; x < SIZE; x++)
        {
            // Outer boundary is always wall.
            if (x == 0 || x == SIZE - 1 || y == 0 || y == SIZE - 1)
            {
                known[y][x] = 1;
            }
            else
            {
                known[y][x] = 0;
            }

            dist[y][x] = INF;
        }
    }
}

```

```

        visited[y][x] = false;
        prevX[y][x] = -1;
        prevY[y][x] = -1;
    }
}

// Start and target are known walkable cells.
known[startY][startX] = 2;
known[targetY][targetX] = 2;
// Defines node structure.
struct HeapNode
{
    int x;
    int y;
    int distance;
};
// Creates a min heap to process the lowest distance node next.
HeapNode heap[SIZE * SIZE * 4];
int heapSize = 0;
// Inserts a node into the above heap.
auto heapPush = [&](int x, int y, int distance)
{
    int index = heapSize;
    heap[index].x = x;
    heap[index].y = y;
    heap[index].distance = distance;
    heapSize++;

    while (index > 0)
    {
        int parent = (index - 1) / 2;

        if (heap[parent].distance <= heap[index].distance)
        {
            break;
        }

        HeapNode temp = heap[parent];
        heap[parent] = heap[index];
        heap[index] = temp;

        index = parent;
    }
};
// Removes the root of the heap (the node with the smallest distance) and
restores the heap property.
auto heapPop = [&]() -> HeapNode
{

```

```

        HeapNode result = heap[0];
        heapSize--;
        heap[0] = heap[heapSize];

        int index = 0;

        while (true)
        {
            int left = index * 2 + 1;
            int right = index * 2 + 2;
            int smallest = index;

            if (left < heapSize && heap[left].distance <
heap[smallest].distance)
            {
                smallest = left;
            }

            if (right < heapSize && heap[right].distance <
heap[smallest].distance)
            {
                smallest = right;
            }

            if (smallest == index)
            {
                break;
            }

            HeapNode temp = heap[index];
            heap[index] = heap[smallest];
            heap[smallest] = temp;

            index = smallest;
        }

        return result;
    };

    // The probed part of the Dijkstra's algorithm. Once a location is read,
    it is marked as known.
    auto ensureKnown = [&](int x, int y)
    {
        if (known[y][x] != 0)
        {
            return;
        }

        griditem square = grid->GetGridSquare(x, y);

```

```

        if (square == griditem::wall)
        {
            known[y][x] = 1;
        }
        else
        {
            known[y][x] = 2;
        }
    };

    // Initialises by setting start node distance to 0. Inserts into priority
queue.
    dist[startY][startX] = 0;
    heapPush(startX, startY, 0);
    // Cardinal movements.
    int dx[4] = { 0, 0, -1, 1 };
    int dy[4] = { -1, 1, 0, 0 };
    // Main Dijkstra's loop. Processes nodes from the priority queue.
Algorithm begins expanding from the player's starting position.
    // Removes the smallest distance from the heap (again and again), skips if
visited, finalises it otherwise.
    // Explores the closest locations.
    // Each location is probed if needed. Not if wall.
    // If a shorter path is found, it is updated and added to the heap.
    while (heapSize > 0)
    {
        HeapNode current = heapPop();
        int x = current.x;
        int y = current.y;

        if (visited[y][x])
        {
            continue;
        }

        visited[y][x] = true;

        if (x == targetX && y == targetY)
        {
            break;
        }

        for (int i = 0; i < 4; i++)
        {
            int nx = x + dx[i];
            int ny = y + dy[i];

            if (nx < 0 || nx >= SIZE || ny < 0 || ny >= SIZE)

```

```

        {
            continue;
        }

        ensureKnown(nx, ny);

        if (known[ny][nx] == 1)
        {
            continue;
        }

        int newDistance = dist[y][x] + 1;

        if (newDistance < dist[ny][nx])
        {
            dist[ny][nx] = newDistance;
            prevX[ny][nx] = x;
            prevY[ny][nx] = y;
            heapPush(nx, ny, newDistance);
        }
    }
}

// No valid path found if target is still INF.
if (dist[targetY][targetX] == INF)
{
    return;
}

// Prepares to reconstruct the shortest path.
movedirection path[SIZE * SIZE];
int pathLength = 0;

int currentX = targetX;
int currentY = targetY;
// Reconstructs the path from the target back to the start using arrays.
while (!(currentX == startX && currentY == startY))
{
    int parentX = prevX[currentY][currentX];
    int parentY = prevY[currentY][currentX];

    if (parentX == -1 || parentY == -1)
    {
        return;
    }

    if (currentX == parentX && currentY == parentY - 1)
    {
        path[pathLength] = movedirection::up;
    }
}

```

```

        else if (currentX == parentX && currentY == parentY + 1)
        {
            path[pathLength] = movedirection::down;
        }
        else if (currentX == parentX - 1 && currentY == parentY)
        {
            path[pathLength] = movedirection::left;
        }
        else if (currentX == parentX + 1 && currentY == parentY)
        {
            path[pathLength] = movedirection::right;
        }

        pathLength++;
        currentX = parentX;
        currentY = parentY;
    }
    // Executes the moves in reverse order to navigate from the start to the
    target.
    for (int i = pathLength - 1; i >= 0; i--)
    {
        moveresults mr = grid->MovePlayer(path[i]);

        if (mr == moveresults::complete)
        {
            break;
        }
    }
}

```

Basic A* Algorithm with Manhattan Distance

```

////A* with Manhattan Distance
void GridNavigatorAutomatic::Navigate()
{
    // Fixed values for the grid size and a large number representing
    infinity.
    const int SIZE = 20;
    const int INF = 1000000;

    // Read the whole maze once
    griditem** maze = grid->GetGrid();
    // Stores the coordinates of the player and target.
    int startX = -1, startY = -1;
    int targetX = -1, targetY = -1;

    // Find start and target positions
    for (int y = 0; y < SIZE; y++)

```

```

{
    for (int x = 0; x < SIZE; x++)
    {
        if (maze[y][x] == griditem::player)
        {
            startX = x;
            startY = y;
        }
        else if (maze[y][x] == griditem::target)
        {
            targetX = x;
            targetY = y;
        }
    }
}
// Stops if code failed to find valid start or target.
if (startX == -1 || targetX == -1)
{
    return;
}
// As the second A* algorithm, this creates the main A* tracking arrays.
int gScore[SIZE][SIZE];
int fScore[SIZE][SIZE];
bool closed[SIZE][SIZE];
int prevX[SIZE][SIZE];
int prevY[SIZE][SIZE];
// Initialises the arrays.
for (int y = 0; y < SIZE; y++)
{
    for (int x = 0; x < SIZE; x++)
    {
        gScore[y][x] = INF;
        fScore[y][x] = INF;
        closed[y][x] = false;
        prevX[y][x] = -1;
        prevY[y][x] = -1;
    }
}
// Defines the node structure.
struct Node
{
    int x;
    int y;
    int f;
};
// Creates an openList for nodes. Tracks nodes that are yet to be
evaluated.
Node openList[SIZE * SIZE * 4];

```

```

    int openCount = 0;
    // Defines the heuristic function for A* search. Uses Manhattan distance
    to estimate the cost from a node to the target.
    auto heuristic = [&](int x, int y)
    {
        return abs(targetX - x) + abs(targetY - y);
    };
    // Creates an openList for future searches
    auto pushNode = [&](int x, int y, int f)
    {
        openList[openCount].x = x;
        openList[openCount].y = y;
        openList[openCount].f = f;
        openCount++;
    };
    // Selects and removes the node using the lowest fscore.
    auto popLowest = [&]() -> Node
    {
        int bestIndex = 0;

        for (int i = 1; i < openCount; i++)
        {
            if (openList[i].f < openList[bestIndex].f)
            {
                bestIndex = i;
            }
        }

        Node best = openList[bestIndex];
        openCount--;

        openList[bestIndex] = openList[openCount];
        return best;
    };
    // Sets the start node cost to zero, finds total cost (Manhattan
    distance), places into openList.
    gScore[startY][startX] = 0;
    fScore[startY][startX] = heuristic(startX, startY);
    pushNode(startX, startY, fScore[startY][startX]);
    // Cardinal movements.
    int dx[4] = { 0, 0, -1, 1 };
    int dy[4] = { -1, 1, 0, 0 };
    // Main A* loop.
    // Takes lowest fscore from open list, skips if closed, marks closed
    otherwise, explores closest path, valid paths are updated if a shorter path
    have been found.
    while (openCount > 0)
    {

```

```

Node current = popLowest();
int x = current.x;
int y = current.y;

if (closed[y][x])
{
    continue;
}

closed[y][x] = true;

if (x == targetX && y == targetY)
{
    break;
}

for (int i = 0; i < 4; i++)
{
    int nx = x + dx[i];
    int ny = y + dy[i];

    if (nx < 0 || nx >= SIZE || ny < 0 || ny >= SIZE)
    {
        continue;
    }

    if (maze[ny][nx] == griditem::wall)
    {
        continue;
    }

    if (closed[ny][nx])
    {
        continue;
    }

    int tentativeG = gScore[y][x] + 1;

    if (tentativeG < gScore[ny][nx])
    {
        gScore[ny][nx] = tentativeG;
        fScore[ny][nx] = tentativeG + heuristic(nx, ny);
        prevX[ny][nx] = x;
        prevY[ny][nx] = y;
        pushNode(nx, ny, fScore[ny][nx]);
    }
}
}

```

```

    // If target has no gScore after search, then this algorithm did not find
    a path. A* did not find a valid route.
    if (gScore[targetY][targetX] == INF)
    {
        return;
    }
    // Preparation to reconstruct the path.
    movedirection path[SIZE * SIZE];
    int pathLength = 0;

    int currentX = targetX;
    int currentY = targetY;
    // Build the shortest path backwards from the target.
    while (!(currentX == startX && currentY == startY))
    {
        int parentX = prevX[currentY][currentX];
        int parentY = prevY[currentY][currentX];

        if (parentX == -1 || parentY == -1)
        {
            return;
        }

        if (currentX == parentX && currentY == parentY - 1)
        {
            path[pathLength] = movedirection::up;
        }
        else if (currentX == parentX && currentY == parentY + 1)
        {
            path[pathLength] = movedirection::down;
        }
        else if (currentX == parentX - 1 && currentY == parentY)
        {
            path[pathLength] = movedirection::left;
        }
        else if (currentX == parentX + 1 && currentY == parentY)
        {
            path[pathLength] = movedirection::right;
        }

        pathLength++;
        currentX = parentX;
        currentY = parentY;
    }
    // Creates the path in the correct forward order. Moves player.
    for (int i = pathLength - 1; i >= 0; i--)
    {
        moveresults mr = grid->MovePlayer(path[i]);
    }

```

```

        if (mr == moveresults::complete)
        {
            break;
        }
    }
}

```

Probed A* Algorithm with Manhattan Distance

```

// A* Algorithm Probed
void GridNavigatorAutomatic::Navigate()
{
    const int SIZE = 20;
    const int INF = 1000000;

    // From the maze generation code:
    // player starts at (1, 18)
    // target starts at (18, 1)
    const int startX = 1;
    const int startY = 18;
    const int targetX = 18;
    const int targetY = 1;

    // Arrays hold the search state for A*
    int known[SIZE][SIZE]; //What has been discovered about the maze so far.

    int gScore[SIZE][SIZE]; // The movement cost from start
    int fScore[SIZE][SIZE]; // Estimate total cost
    bool closed[SIZE][SIZE]; // Marks nodes that have been evaluated
    int prevX[SIZE][SIZE]; // Previous node tracking
    int prevY[SIZE][SIZE]; // Previous node tracking

    // Prepares the grid for probed A* search.
    // Marks walls immediately and resets so that search starts with no
    explored routes

    for (int y = 0; y < SIZE; y++)
    {
        for (int x = 0; x < SIZE; x++)
        {
            // Outer border is always wall
            if (x == 0 || x == SIZE - 1 || y == 0 || y == SIZE - 1)
            {
                known[y][x] = 1;
            }
            else
            {

```

```

        known[y][x] = 0;
    }

    gScore[y][x] = INF;
    fScore[y][x] = INF;
    closed[y][x] = false;
    prevX[y][x] = -1;
    prevY[y][x] = -1;
}
}

// Start and target are definitely accessible squares.
known[startY][startX] = 2;
known[targetY][targetX] = 2;
// Represents one node in the open list for A* search. Each node contains
its coordinates and its fScore.
struct Node
{
    int x;
    int y;
    int f;
};
// The open list is a simple array of nodes. Implemented with a simple
array.
Node openList[SIZE * SIZE * 4];
int openCount = 0;

// Heuristic function for A* search. Uses Manhattan distance to estimate
the cost from a node to the target.
auto heuristic = [&](int x, int y)
{
    return abs(targetX - x) + abs(targetY - y);
};
// Adds a node to the open list with its coordinates and fScore.
auto pushNode = [&](int x, int y, int f)
{
    openList[openCount].x = x;
    openList[openCount].y = y;
    openList[openCount].f = f;
    openCount++;
};
// Removes and returns the node with the lowest fScore from the open list.
// This is a linear search through the open list to find the node with the
lowest fScore.
auto popLowest = [&]() -> Node
{
    int bestIndex = 0;

```

```

        for (int i = 1; i < openCount; i++)
        {
            if (openList[i].f < openList[bestIndex].f)
            {
                bestIndex = i;
            }
        }

        Node best = openList[bestIndex];
        openCount--;
        openList[bestIndex] = openList[openCount];

        return best;
    };

    // This is the probed part of the A* search.
    // It checks if a square has been discovered yet. If not, it queries the
    grid to find out if it's a wall or a possible place to move to.
    // Once discovered, it marks the square as known.
    auto ensureKnown = [&](int x, int y)
    {
        if (known[y][x] != 0)
        {
            return;
        }

        griditem square = grid->GetGridSquare(x, y);

        if (square == griditem::wall)
        {
            known[y][x] = 1;
        }
        else
        {
            known[y][x] = 2;
        }
    };

    // Initialises the search from the start node.
    gScore[startY][startX] = 0;
    fScore[startY][startX] = heuristic(startX, startY);
    pushNode(startX, startY, fScore[startY][startX]);

    int dx[4] = { 0, 0, -1, 1 };
    int dy[4] = { -1, 1, 0, 0 };

    // Main A* search loop.
    // An iteration takes the node with the lowest fScore from the open list,
    skips if closed, marks it closed otherwise.

```

```

    // Then, it moves onto closest squares.
    // For each closest square, A* checks if it is a wall, closed, or unknown.
    // If unknown, it probes the grid to find out what it is.
    // The loop combines the path cost with the Manhattan estimate.
    while (openCount > 0)
    {
        Node current = popLowest();
        int x = current.x;
        int y = current.y;

        if (closed[y][x])
        {
            continue;
        }

        closed[y][x] = true;

        if (x == targetX && y == targetY)
        {
            break;
        }

        for (int i = 0; i < 4; i++)
        {
            int nx = x + dx[i];
            int ny = y + dy[i];

            if (nx < 0 || nx >= SIZE || ny < 0 || ny >= SIZE)
            {
                continue;
            }

            ensureKnown(nx, ny);

            if (known[ny][nx] == 1)
            {
                continue;
            }

            if (closed[ny][nx])
            {
                continue;
            }

            int tentativeG = gScore[y][x] + 1;

            if (tentativeG < gScore[ny][nx])
            {

```

```

        gScore[ny][nx] = tentativeG;
        fScore[ny][nx] = tentativeG + heuristic(nx, ny);
        prevX[ny][nx] = x;
        prevY[ny][nx] = y;
        pushNode(nx, ny, fScore[ny][nx]);
    }
}
}
// If target has no gScore after search, then this algorithm did not find
a path.
if (gScore[targetY][targetX] == INF)
{
    return;
}

// These variables are used to reconstruct the path from the target.
// A* stores the previous node for each node in the prevX and prevY
arrays. This loop traces back from the target to the start, building a path of
move directions.
movedirection path[SIZE * SIZE];
int pathLength = 0;

int currentX = targetX;
int currentY = targetY;

while (!(currentX == startX && currentY == startY))
{
    int parentX = prevX[currentY][currentX];
    int parentY = prevY[currentY][currentX];

    if (parentX == -1 || parentY == -1)
    {
        return;
    }

    if (currentX == parentX && currentY == parentY - 1)
    {
        path[pathLength] = movedirection::up;
    }
    else if (currentX == parentX && currentY == parentY + 1)
    {
        path[pathLength] = movedirection::down;
    }
    else if (currentX == parentX - 1 && currentY == parentY)
    {
        path[pathLength] = movedirection::left;
    }
    else if (currentX == parentX + 1 && currentY == parentY)

```

```

    {
        path[pathLength] = movedirection::right;
    }

    pathLength++;
    currentX = parentX;
    currentY = parentY;
}
// This portion of code executes the moves in reverse order to navigate
from the start to the target.
for (int i = pathLength - 1; i >= 0; i--)
{
    moveresults mr = grid->MovePlayer(path[i]);

    if (mr == moveresults::complete)
    {
        break;
    }
}
}

```

Conclusion

In conclusion, the proposed solutions highlight successful maze pathfinding solutions to the given maze to guarantee the shortest path. Dijkstra's algorithm is a strong choice that generates a reliable but less efficient pathfinding solution. This is because the basic Dijkstra explores nodes without a goal directed guidance. To increase efficiency, the A* algorithm uses a heuristic to direct search. The probed code implementations discover locations only when needed which reduce the views of the maze. Overall, all implementations use similar amount of moves, despite the difference in grid accesses. They determine equally good paths but with different access costs. Collisions with walls were non-existent with each implementation. The differences are related to how they search rather than the amount of errors. Amongst the implementations, the most successful algorithm is the A* probed algorithm with the lowest average grid access and so the most computationally efficient. It avoids unnecessary grid accesses like the a full grid view with the basic Dijkstra and A*. The A* probed implementation uses a heuristic guidance and probing which creates an advantage over the Dijkstra implementations.

References

- [1] A. Madkour and W. G. Aref, "A Survey of Shortest-Path Algorithms," *Data Structures and Algorithms*, 2017.

- [2] Z. A. Algfoor, M. S. Sunar and . H. Kolivand, "A Comprehensive Study on Pathfinding Techniques for Robotics and Video Games," *International Journal of Computer Games Technology*, no. 1, pp. 1-11, 2015.
- [3] N. R. Sturtevant and R. Geisberger, "A Comparison of High-Level Approaches for Speeding Up Pathfinding," *Sixth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, vol. 6, no. 1, pp. 76-82, 2010.
- [4] A. Fitro, "SHORTEST PATH FINDING IN GEOGRAPHICAL INFORMATION SYSTEMS USING NODE COMBINATION AND DIJKSTRA ALGORITHM," *International Journal of Mechanical Engineering and Technology (IJMET)*, vol. 9, no. 2, pp. 755-760, 2018.
- [5] H. B. I. C. A. K. M. A. Adel Ammar, "Relaxed Dijkstra and A* with linear complexity for robot path planning problems in large-scale grid environments," *Soft computing*, vol. 20, no. 10, pp. 4149-4171, 2016.
- [6] A. Candra and M. A. Budiman, "Dijkstra's and A-Star in Finding the Shortest Path: a Tutorial," *International Conference on Data Science, Artificial Intelligence, and Business Analytics (DATABIA)*, pp. 28-32, 2020.
- [7] H. T. J. G. T. F. A.R. Soltani, "Path planning in construction sites: performance evaluation of the Dijkstra, Ap , and GA search algorithms," *Advanced Engineering Informatics*, vol. 303, no. 4, p. 291, 2002.
- [8] M. M. Attoyibi, F. E. Fikrisa and A. N. Handayani, "The Implementation of A Star Algorithm (A*) In the Game Education About Numbers Introduction," *2nd International Conference on Vocational Education and Training (ICOVET 2018)*, vol. 242, pp. 234-237, 2019.
- [9] D. Foeada, "A Systematic Literature Review of A* Pathfinding," *5th International Conference on Computer Science and Computational Intelligence 2020*, pp. 507-514, 2021.
- [10] M. L. Fredman, "Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms," *Association for Computing Machinery*, vol. 34, no. 3, pp. 597-613, 1987.
- [11] A. M. Elshaer, "Exploring Algorithmic Efficiency of A-Star and Dijkstra for Optimal Route Planning in Green Transportation," *International Journal of Intelligent Transportation Systems Research*, vol. 23, no. 2, pp. 1097-1107, 2025.

Appendix 1

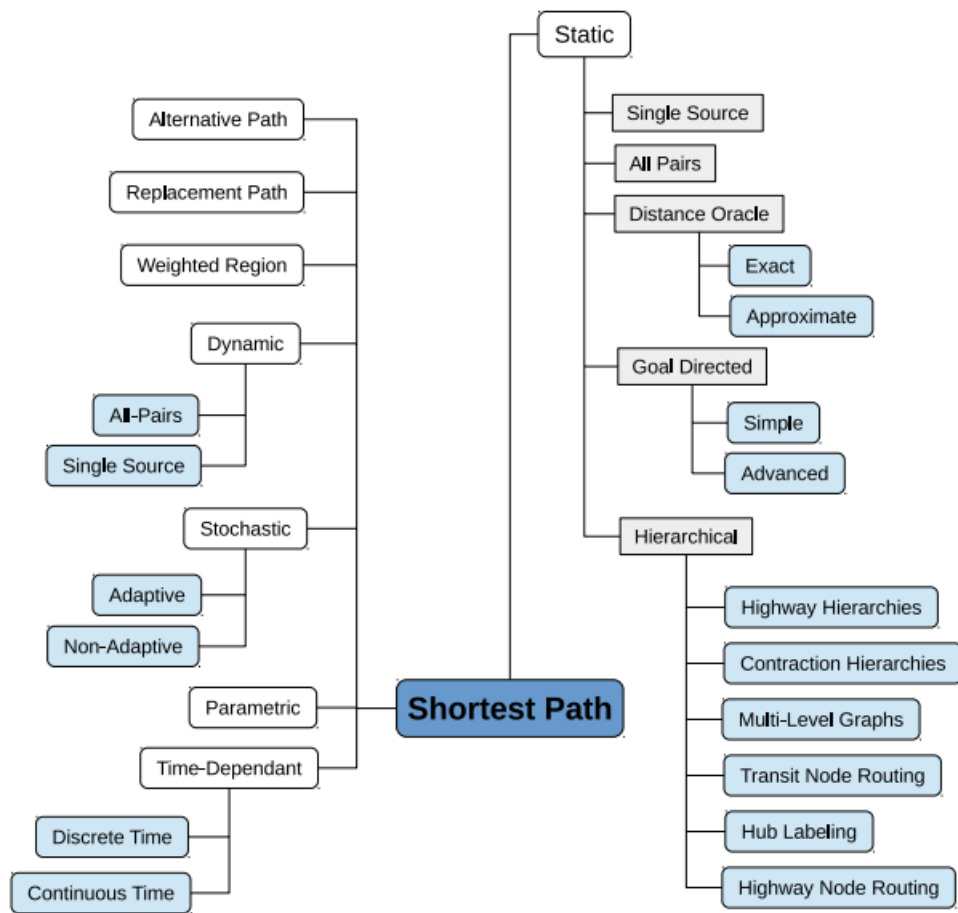


Figure 1: Taxonomy of shortest path algorithms [1].

```

input : Grid, Start, Goal
output : dist
1 for each vertex v in Grid do
2   dist(v)=infinity // Mark distance from
   start to each node v as not yet
   computed;
3   visited(v)=false // Mark all nodes as
   unvisited;
4   previous(v)=undefined // Previous node in
   optimal path;
5 end
6 dist(Start)=0;
7 insert Start into Q // Initially, only the
   start node is in the Queue;
8 while (Q is not empty) and (not visited(Goal)) do
9   u = not visited vertex in Q with smallest distance;
10  remove u from Q;
11  visited(u)=true // mark this node as
   visited;
12  for each neighbor v of u do
13    g = dist(u) + distEdge(u, v) // distance
       from Start;
14    if g ≤ dist(v) then
15      dist(v)=g;
16      previous(v)=u ;
17      if !visited(v) then
18        insert v into Q // Add unvisited
           node into the Queue to be
           processed;
19      end
20    end
21  end
22 end

```

Algorithm 1: Classical Dijkstra

Figure 2. Classical Dijkstra [5]

$$G(n) = G \text{ node} + 10(\text{straight}) \text{ atau } 14(\text{sideways})$$

.....Equation (1)

$$H(n) = |X(\text{target}) - X(n)| + |Y(\text{target}) - Y(n)|$$

.....Equation (2)

$$F(n) = G(n) + H(n) \text{..... Equation (3)}$$

Figure 3 – A* Equations [8]

	Dijkstra's	A-Star
Time Complexity	$O(V^2)$, finished in 0.22 s	$O(E)$, finished in 0.07 s
Space Complexity	$O(V + E)$, used (179241, 190961) bytes	$O(V)$, used (159287, 163919) bytes
Accuracy in Pathfinding	optimal or near-optimal paths	optimal or near-optimal paths

Figure 4 – Performance of Dijkstra and A Star [11]
