

Robot Navigator

Alexander Small

Table of Contents

Instructions for running the solution.....	2
Introduction.....	3
The Robot Navigation Problem.....	3
Introduction to Search Algorithms.....	4
Search Algorithms.....	6
Breadth-First Search (BFS).....	6
Depth-First Search (DFS).....	6
Greedy Best-First Search (GBFS).....	8
A-Star Search (A*).....	8
CUS1 – Random Frontier Sorting (Uninformed Search).....	8
CUS2 – Informed Search.....	8
Implementation.....	9
Breadth-First Search.....	11
Depth-First Search.....	11
Greedy Best-First Search.....	12
A* Search.....	12
Custom Search 1 (Random Uninformed Search).....	13
Features / Bugs.....	13
Features:.....	13
Bugs:.....	13
Missing Features.....	13
Research Initiative – Random Solvable Maze Generator.....	13
How to run the Random Maze Generator.....	14
How Random Maze Generation Works.....	15
Conclusion.....	18
Glossary.....	18
References.....	18

Instructions for running the solution

To run the **RobotNavigator** solution, open the project file (RobotNavigator.sln) in Visual Studio Community Edition and compile it. Navigate to the compiled **RobotNavigator** directory. The folder contents should look like this:

This PC > Desktop > RobotNavigator

Name	Date modified	Type	Size
RobotNavigator.deps.json	4/04/2023 12:50 PM	JSON File	1 KB
RobotNavigator.dll	8/04/2023 11:14 PM	Application extens...	17 KB
RobotNavigator.exe	8/04/2023 11:14 PM	Application	171 KB
RobotNavigator.pdb	8/04/2023 11:14 PM	Program Debug D...	18 KB
RobotNavigator.runtimeconfig.dev.json	14/03/2023 6:29 PM	JSON File	1 KB
RobotNavigator.runtimeconfig.json	14/03/2023 6:29 PM	JSON File	1 KB

Copy the path of the **RobotNavigator** directory to your clipboard. Open **Command Prompt (cmd)** and access the directory by pasting the statement:

```
cd <RobotNavigator folder directory>
```

```
C:\> Command Prompt

Microsoft Windows [Version 10.0.19045.2728]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Alex>cd C:\Users\Alex\Desktop\RobotNavigator

C:\Users\Alex\Desktop\RobotNavigator>
```

Once you have access to the **RobotNavigator** directory within **Command Prompt**, you can use RobotNavigator by typing:

```
RobotNavigator <problem file to search> <search method>
```

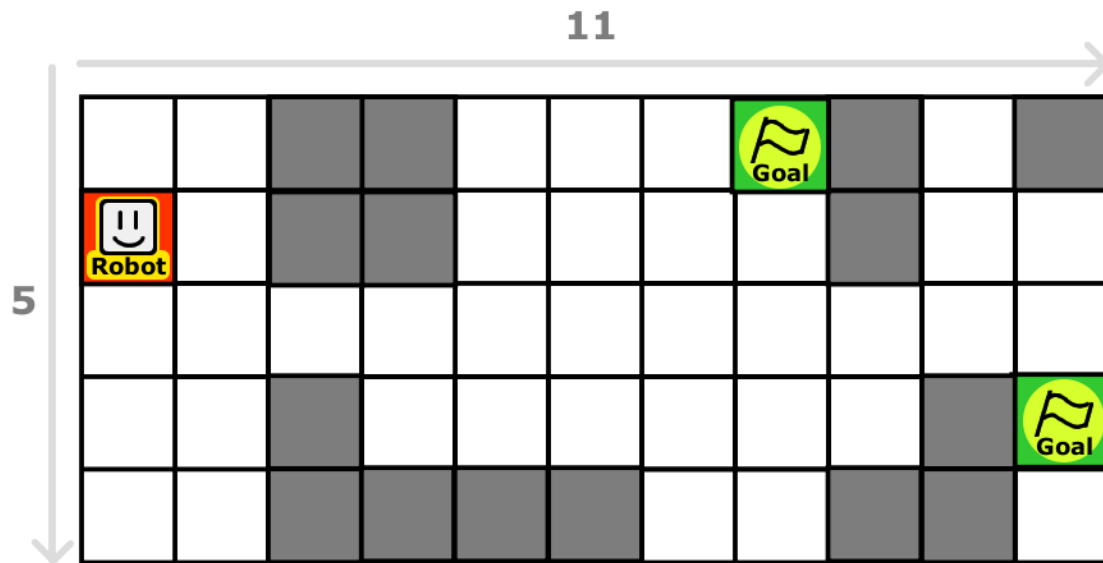
For example, typing: `RobotNavigator RobotNav-test.txt DFS` will search through the problem file **RobotNav-test.txt** using a **Depth-First Search (DFS)**.

```
C:\Users\Alex\Desktop\RobotNavigator>RobotNavigator RobotNav-test.txt DFS

RobotNav-test.txt DFS 46
up; right; down; down; right; right; down; right; up; up; up; right; down; down; down; right; up; up; up; right;
```

Introduction

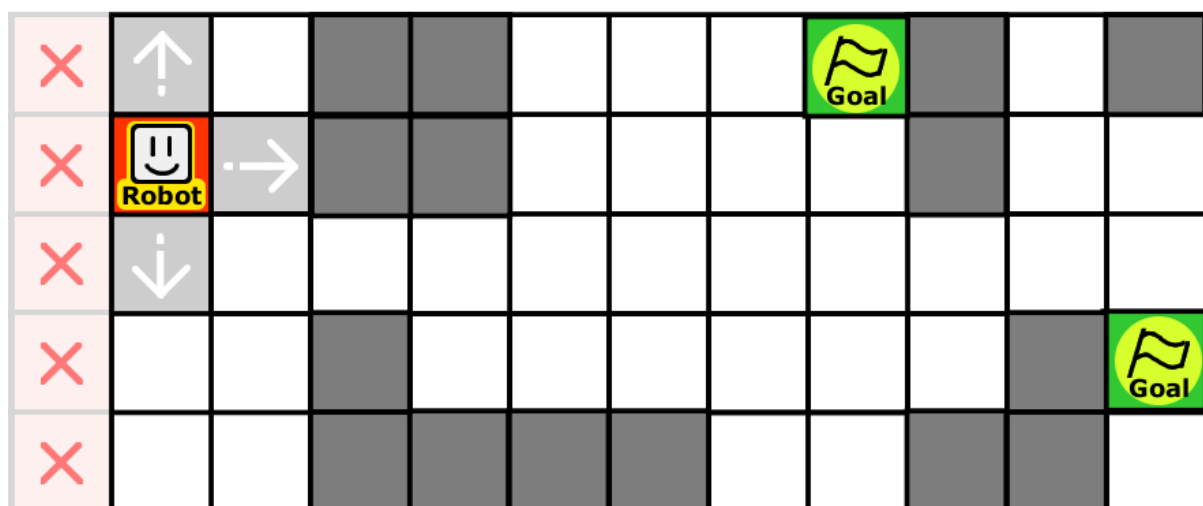
The Robot Navigation Problem



The Robot Navigation problem consists of a robot (pictured in the red square), goal spaces, and walls (the grey squares). These objects exist within a 2D world of square tiles.

The world has a fixed size. In this example the world is 11 squares long and 5 squares tall, so the world is 11x5. The world's height and width must both be at least 1 square.

The robot's initial location must be an **empty** square in the map (i.e., not a wall or a goal space).

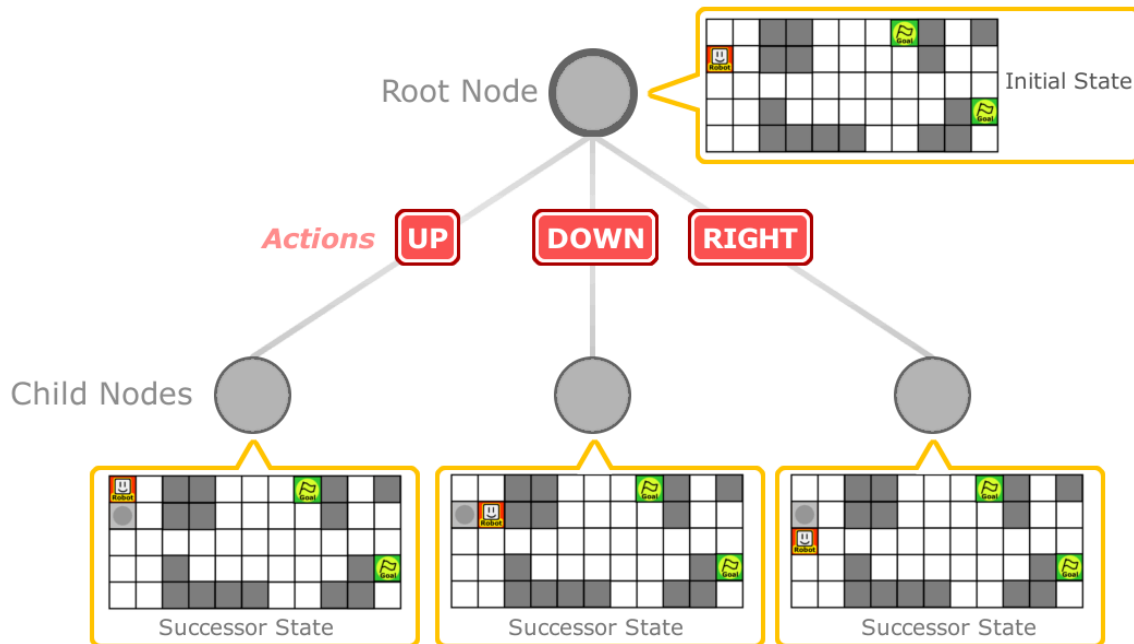


The robot can move in four cardinal directions: up, left, down, and right. The robot can only move one square at a time. The robot can't move into a square where there is a wall (the grey squares) or a square outside of the map's boundaries.

The robot **must get to a goal space to satisfy the navigation problem**. The aim of the Robot Navigation problem is to find the **optimal solution** – the shortest possible sequence of moves which will take the robot to a goal space. It is worth noting that multiple optimal solutions exist.

Introduction to Search Algorithms

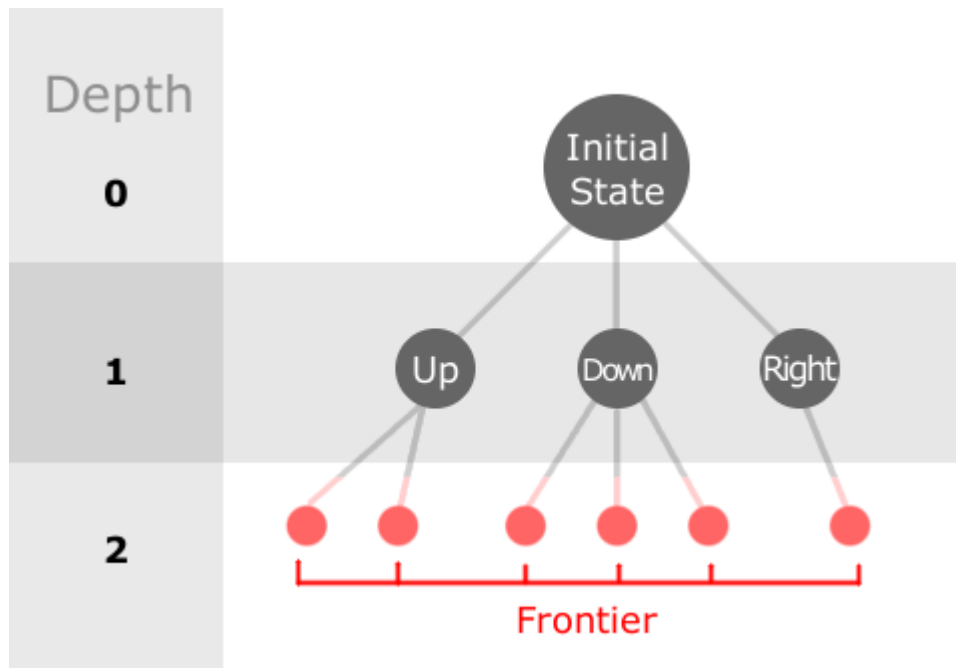
To solve the Robot Navigation problem, we utilise **Search Algorithms**.



The diagram above depicts a **search tree**. Search algorithms use **search trees** to explore many states of a world by sequentially performing actions to generate branches in the search tree. Search trees only stop searching through possible world states once they find a state within the tree that satisfies their **goal condition**. Once the goal is reached, the search tree stops and returns the list of actions which led to this goal state. The method of checking whether a state meets the goal condition is called a **goal test**.

We will be using a **search tree** to solve the Robot Navigator problem. To begin with, we get the **initial state** of the Robot Navigator problem (seen in the first example) and turn it into a **Node**. Nodes are objects which encapsulate a **state of the world** – essentially a snapshot of the world at a certain time.

We then **expand** this node using its **successor function** to generate a list of **child nodes** based on **all legal actions from the given state**. For our initial state in this example, the legal actions for the robot are to move **Up**, **Down**, or **Right**.



Nodes within a search tree also have a **depth**, which represents how many actions it took to reach that node.

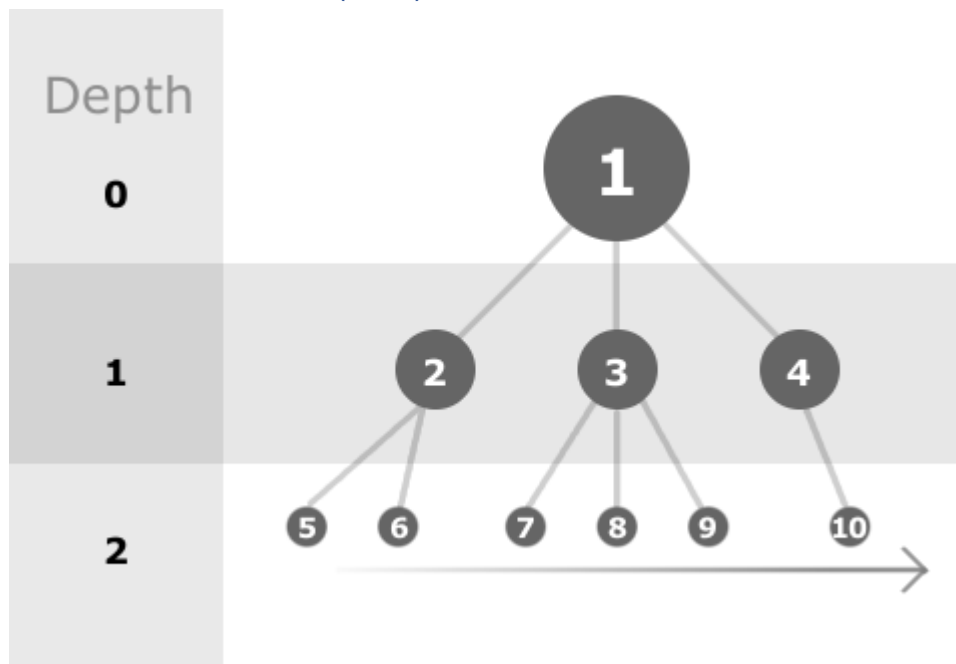
Nodes in a search tree which **have not been expanded** (and therefore don't have children) are part of the **frontier** – a queue of nodes waiting for expansion. The order in which we choose to expand nodes in the frontier depends on our **Search Strategy**.

Some search strategies **compare the state of every node in the frontier** to decide which nodes are most promising and expand those nodes first. These are called **informed** search strategies. In informed search strategies, a node's desirability is calculated using an **evaluation value**. The node with the lowest **evaluation value** is typically expanded first before all others.

Search strategies which do not consider the states of nodes are called **uninformed** search strategies.

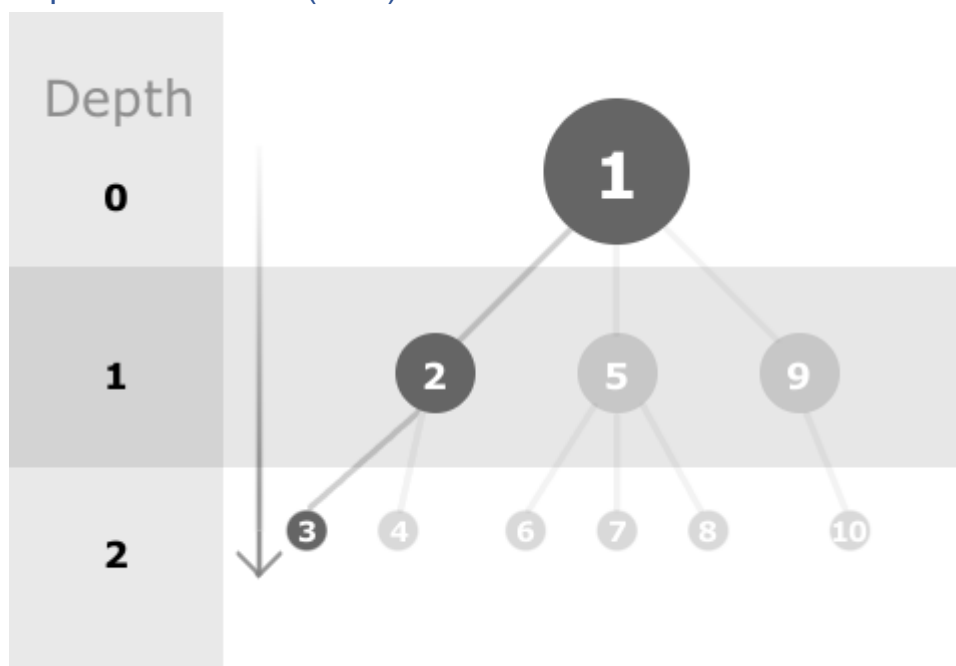
Search Algorithms

Breadth-First Search (BFS)



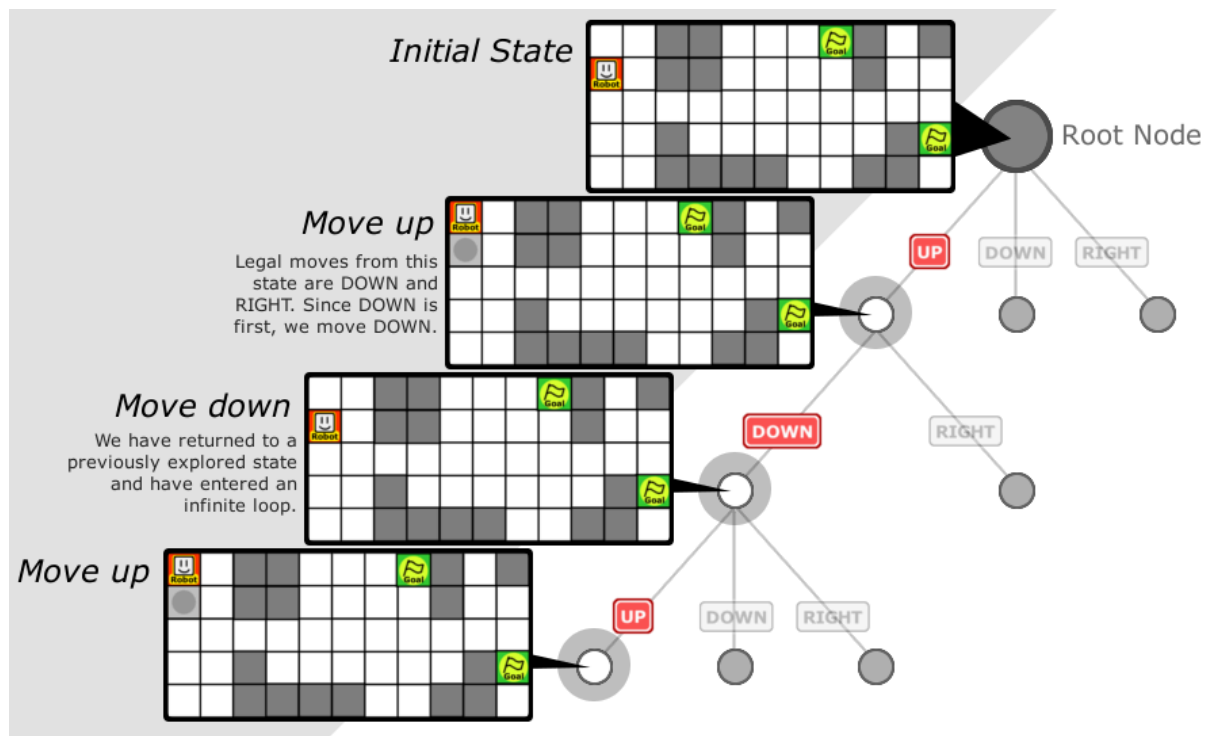
A breadth-first search explores nodes in the frontier in the order that they are added to the frontier. Child nodes are sequentially added to the end of the frontier when a parent node is expanded. This is also known as a **First In First Out** queue. The search is **horizontal**, that is, **all nodes of the current depth are explored before the search goes deeper**. This is why it is called breadth-first.

Depth-First Search (DFS)

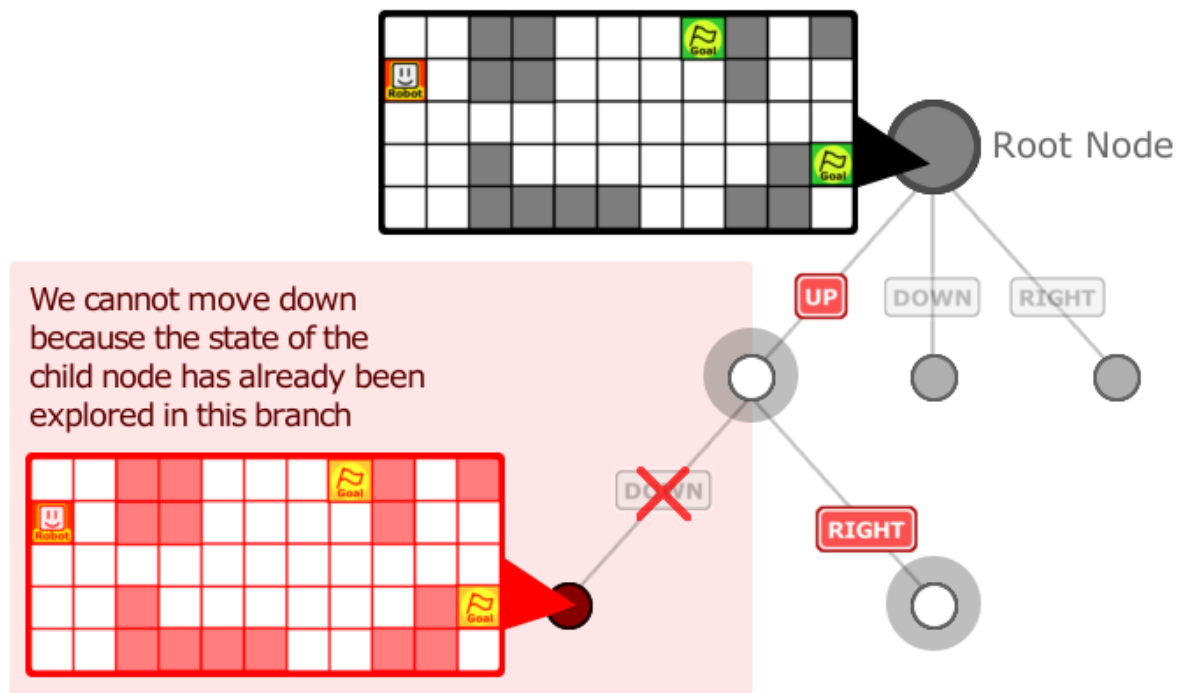


A depth-first search will always expand the **deepest node** in the current frontier. It will **burrow** as deep into individual branches of the search tree as possible. Once a branch has been explored fully, the DFS will then find the next deepest node in the frontier to expand.

An issue with **Depth-First Search** is that it may enter **infinite loops** if nodes within the search tree lead to previously explored nodes. If this occurs, the search will continue indefinitely, never finding a solution node and consuming more and more memory.



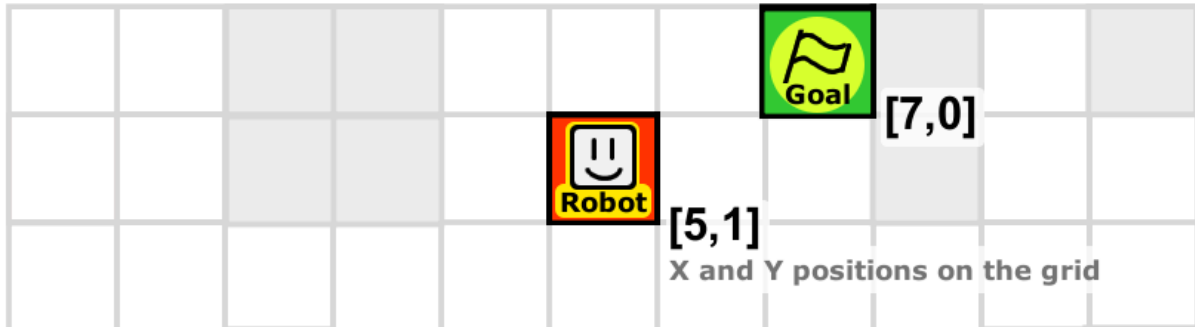
A solution to this issue is to add **repeated state checking**. This prevents the search from exploring nodes which have states which were already discovered within the current branch.



Greedy Best-First Search (GBFS)

This is an **informed search** where a node's **heuristic value** is used as its **evaluation value** to decide which node in the frontier to expand first.

The heuristic value in RobotNavigator is based on the Manhattan distance between the player and the closest goal tile.



$$\begin{aligned}\text{Manhattan Distance} &= \text{abs}(x_2 - x_1) + \text{abs}(y_2 - y_1) \\ &= \text{abs}(7 - 5) + \text{abs}(0 - 1) \\ &= 2 + 1 \\ &= 3\end{aligned}$$

The GBFS search will expand the node in the frontier with the **lowest heuristic value**.

A-Star Search (A*)

A* is an informed search algorithm where the **evaluation value used for nodes is a combination of the node's path cost and its heuristic value**. The evaluation value is calculated as follows:

$$f(n) = g(n) + h(n)$$

- n = A node within the search tree
- f(n) = The **evaluation value** for node n
- g(n) = The **path cost** for node n
- h(n) = The **heuristic value** for node n

NOTE: In RobotNavigator, a node's **path cost** is **identical to its depth**.

CUS1 – Random Frontier Sorting (Uninformed Search)

CUS1 is an uninformed search algorithm which **randomly chooses nodes in the frontier for expansion**. The search will produce different results every time it is run, so the path to the goal node and which goal node was found will vary. The algorithm uses **repeated state checking** to prevent infinite loops.

CUS2 – Informed Search

Sadly, I did not get the time to implement a custom informed search for my RobotNavigator solution.

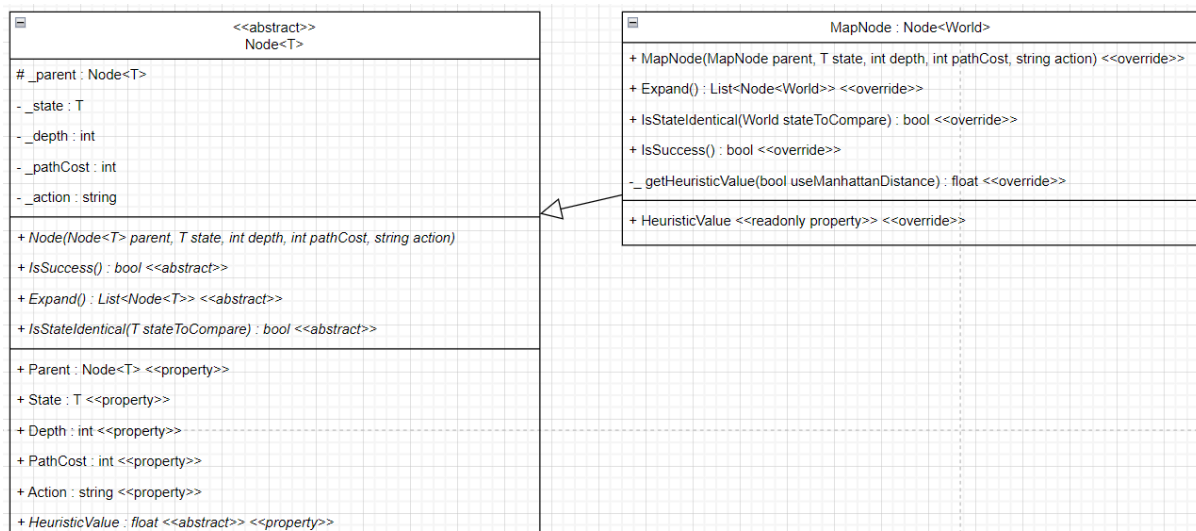
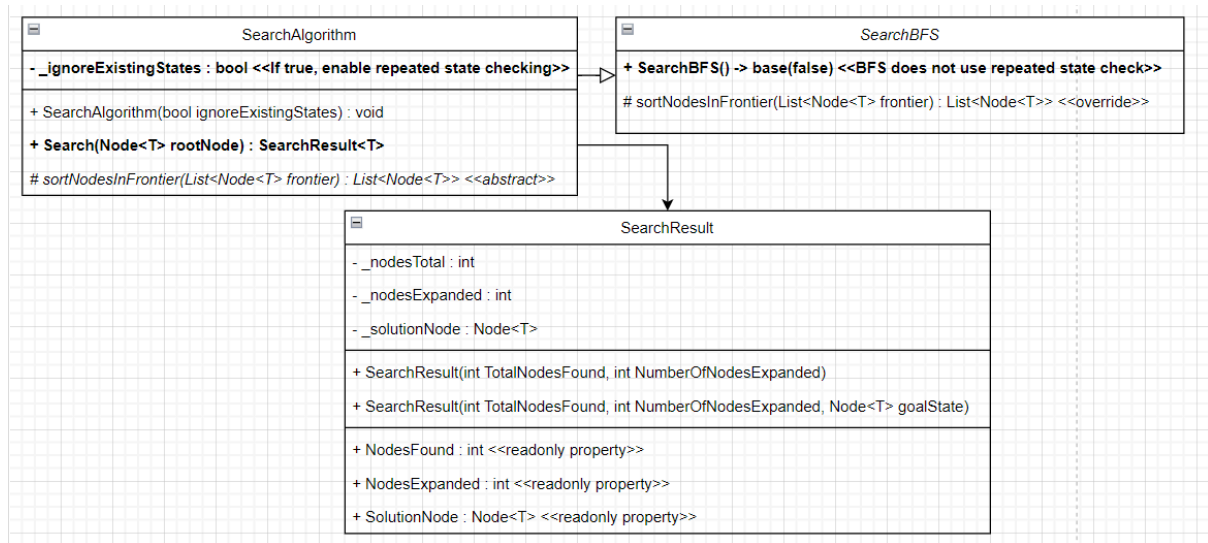
Implementation

Every search algorithm in my solution inherits from the generic abstract class **SearchAlgorithm**.

Children of **SearchAlgorithm** do not change the **Search** function; they only modify the **sortNodesInFrontier** function, which sorts the list of nodes in the frontier. Nodes at the top of the frontier are chosen for expansion in the search function.

When the **Search** function completes, it returns a **SearchResult** object which contains information about the search, such as the total number of nodes discovered and expanded, and the **solution node** object. A path of nodes from the root node to the solution node can be generated by getting all the parent nodes of the solution node.

Here are some UML diagrams to visualise the class structure for these classes:



Here is some pseudocode to demonstrate how the Search function works:

```
public function Search(rootNode : Node<T>) returns SearchResult

    solutionNode = null
    nodesExpanded = 0
    nodesDiscovered = 0
    currentDepth = 0

    // This dictionary is used for repeated state checking.
    // It stores the state and depth of every expanded node.
    knownStates = new Dictionary<T state, int depth>()

    frontier = new List<Node<T>>()

    // Expand the initial state and add all children to the frontier

    foreach (Node<T> childNode in rootNode.Expand())
        frontier.Add(childNode)
        nodesDiscovered++

    while (frontier.Count > 0)

        if (frontier.Count > 1) then frontier = sortNodesInFrontier(frontier)

        // Set currentNode to the first node in the Frontier list
        currentNode = frontier[0]

        if (currentNode.isSuccess) then
            solutionNode = currentNode
            break
            // We found the solution node, so stop executing this while loop

        currentDepth = currentNode.Depth
        knownStates.Add(currentNode.State, currentDepth)

        // Remove all nodes in knownStates which have a higher depth than the current depth
        foreach (state in knownStates)
            if (state.Depth > currentDepth) then knownStates.Remove(state)

        // Expand the current node and add all its children to the frontier
        // UNLESS those children are repeated states and we are checking for repeated states
        foreach (Node<T> childNode in currentNode.Expand())

            if (not knownStates.has(childNode.state) or repeatedStateCheck = false) then
                frontier.Add(childNode)
                nodesDiscovered++

    nodesExpanded ++
```

Breadth-First Search

`_ignoreExistingStates = False`

```
protected override List<Node<T>> sortNodesInFrontier(List<Node<T>> frontier)
{
    return frontier;
}
```

The Breadth-First Search does not sort nodes in the frontier, so the sorting method simply returns the frontier.

Depth-First Search

`_ignoreExistingStates = True`

```
protected override List<Node<T>> sortNodesInFrontier(List<Node<T>> frontier)
{
    List<Node<T>> sortedFrontier = frontier;

    int maxDepth = 0;

    // Set maxDepth to the Depth of the deepest node(s) in the frontier.
    foreach (Node<T> node in frontier)
    {
        maxDepth = Math.Max(maxDepth, node.Depth);
    }

    // Add all the nodes with the highest depth into the list desirableNodes.
    List<Node<T>> desirableNodes = new List<Node<T>>();

    foreach (Node<T> node in frontier)
    {
        if (node.Depth == maxDepth) desirableNodes.Add(node);
    }

    // The first node of this list is the best node.
    // It has the best action priority (UP before LEFT before DOWN before RIGHT) and was created the earliest.
    Node<T> bestNode = desirableNodes[0];

    // Move this node to the top of the frontier.
    sortedFrontier.Remove(bestNode);
    sortedFrontier.Insert(0, bestNode);

    return sortedFrontier;
}
```

Greedy Best-First Search

`_ignoreExistingStates = True`

```
protected override List<Node<T>> sortNodesInFrontier(List<Node<T>> frontier)
{
    List<Node<T>> sortedFrontier = frontier;

    float minimumHeuristicValue = 999999999999;

    // Get the lowest heuristic value of all nodes in the search tree
    foreach (Node<T> node in frontier)
    {
        minimumHeuristicValue = Math.Min(minimumHeuristicValue, node.HeuristicValue);
    }

    // Add all of the nodes with the lowest heuristic value into a list
    List<Node<T>> desiredNodes = new List<Node<T>>();
    foreach (Node<T> node in frontier)
    {
        if (node.HeuristicValue == minimumHeuristicValue) desiredNodes.Add(node);
    }

    // The first node of this list is the best node.
    Node<T> bestNode = desiredNodes[0];

    // Move this optimal node to the top of the frontier.
    sortedFrontier.Remove(bestNode);
    sortedFrontier.Insert(0, bestNode);

    return sortedFrontier;
}
```

A* Search

`_ignoreExistingStates = True`

```
2 references
protected override List<Node<T>> sortNodesInFrontier(List<Node<T>> frontier)
{
    List<Node<T>> sortedFrontier = frontier;

    float minimumHeuristicValue = 999999999999;

    // Get the lowest heuristic value of all nodes in the search tree
    foreach (Node<T> node in frontier)
    {
        float nodeHeuristicValue = node.HeuristicValue + node.PathCost;
        minimumHeuristicValue = Math.Min(minimumHeuristicValue, nodeHeuristicValue);
    }

    // Add all of the nodes with the lowest heuristic value into a list
    List<Node<T>> desiredNodes = new List<Node<T>>();
    foreach (Node<T> node in frontier)
    {
        float nodeHeuristicValue = node.HeuristicValue + node.PathCost;
        if (nodeHeuristicValue == minimumHeuristicValue) desiredNodes.Add(node);
    }

    // The first node of this list is the best node.
    Node<T> bestNode = desiredNodes[0];

    // Move this optimal node to the top of the frontier.
    sortedFrontier.Remove(bestNode);
    sortedFrontier.Insert(0, bestNode);

    return sortedFrontier;
}
```

Custom Search 1 (Random Uninformed Search)

`_ignoreExistingStates = False`

```
protected override List<Node<T>> sortNodesInFrontier(List<Node<T>> frontier)
{
    List<Node<T>> sortedFrontier = frontier;

    Random rng = new Random();

    // Set ITEM to a random int range from 0 to the length of the frontier list
    int Item = rng.Next(0, sortedFrontier.Count - 1);

    // Select node ITEM from the frontier
    Node<T> item = sortedFrontier[Item];

    // Place that node at the front of the frontier
    sortedFrontier.Remove(item);
    sortedFrontier.Add(item);

    return sortedFrontier;
}
```

Features / Bugs

Features:

- Working Random Solvable Maze Generator.
- BFS, DFS, GBFS, A*, and CUS1 search algorithms implemented correctly.

Bugs:

- BFS takes a very long time to find a solution node.
- Random Maze Generation sometimes hangs indefinitely if you generate a maze larger than approximately 12x12. This is likely due to the random uninformed search algorithm **CUS1** being unable to find the goal node.
- The Random Maze Generator sometimes generates mazes that are trivially easy to solve if the random player position is placed very close to a goal tile.

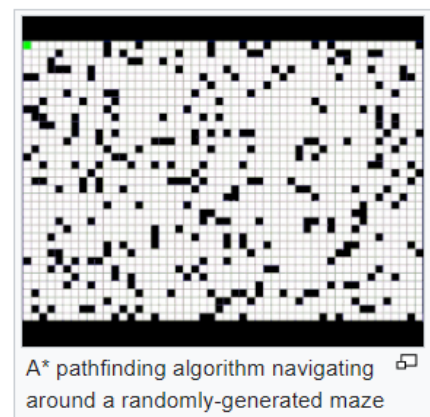
Missing Features

- There is no CUS2 (custom informed search) algorithm.

Research Initiative – Random Solvable Maze Generator

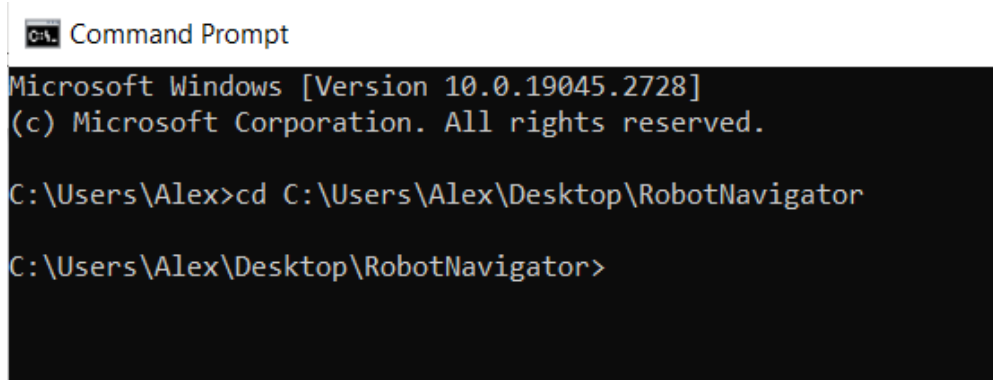
For my research extension task, I created **random maze generation** within the RobotNavigator program. This allows the RobotNavigator program to generate **random solvable mazes** of a specified width, height, and number of goal squares.

I was inspired by a Wikipedia page where I saw an A* search algorithm pathfinding around a randomly generated maze ([Wgullyn, 2021](#)).



How to run the Random Maze Generator

1. Open RobotNavigator in CMD



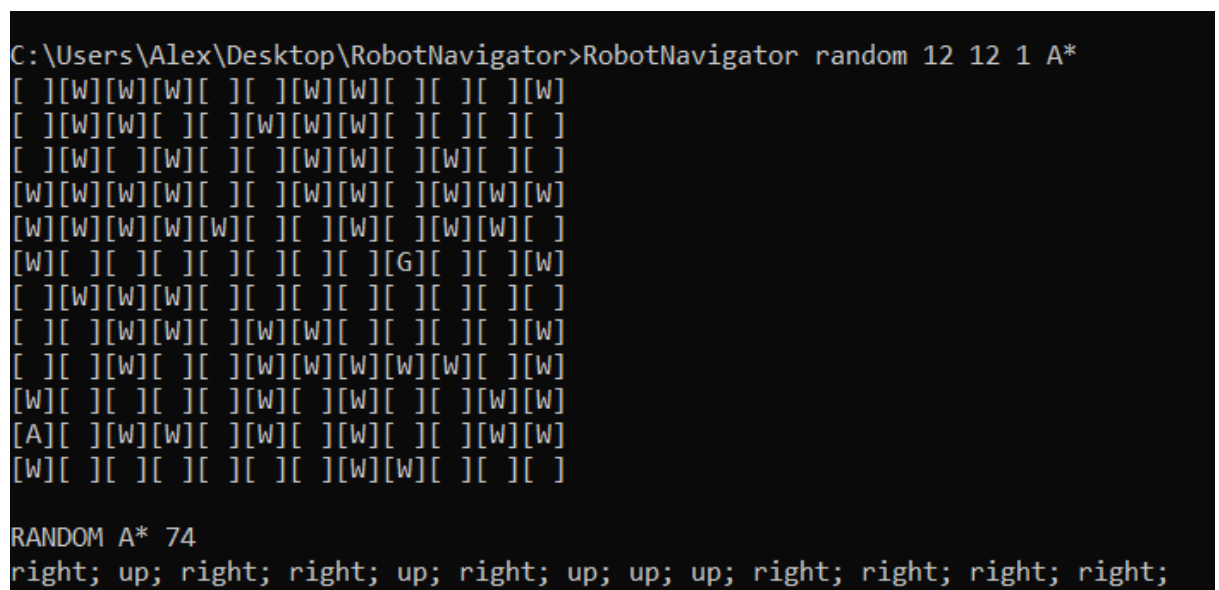
```
Command Prompt
Microsoft Windows [Version 10.0.19045.2728]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Alex>cd C:\Users\Alex\Desktop\RobotNavigator

C:\Users\Alex\Desktop\RobotNavigator>
```

2. Use the following command:

```
RobotNavigator random <maze width> <maze height> <number of goal squares> <search method>
```



```
C:\Users\Alex\Desktop\RobotNavigator>RobotNavigator random 12 12 1 A*
[ ][W][W][W][ ][ ][W][W][ ][ ][ ][W]
[ ][W][W][ ][ ][ ][W][W][W][ ][ ][ ][ ]
[ ][W][ ][W][ ][ ][W][W][ ][W][ ][ ][ ]
[W][W][W][W][ ][ ][W][W][ ][W][W][W]
[W][W][W][W][W][ ][ ][W][ ][W][W][ ][ ]
[W][ ][ ][ ][ ][ ][ ][ ][G][ ][ ][W]
[ ][W][W][W][ ][ ][ ][ ][ ][ ][ ][ ][ ]
[ ][ ][W][W][ ][W][W][ ][ ][ ][ ][W]
[ ][ ][W][ ][ ][W][W][W][W][W][ ][W]
[W][ ][ ][ ][ ][W][ ][W][ ][ ][W][W]
[A][ ][W][W][ ][W][ ][W][ ][ ][W][W]
[W][ ][ ][ ][ ][ ][ ][W][W][ ][ ][ ][ ]

RANDOM A* 74
right; up; right; right; up; right; up; up; up; right; right; right; right;
```

This will generate a random maze of dimensions [width, height] with the **specified number of goal squares**.

The random maze layout will be printed in text format and solved using the specified search algorithm.

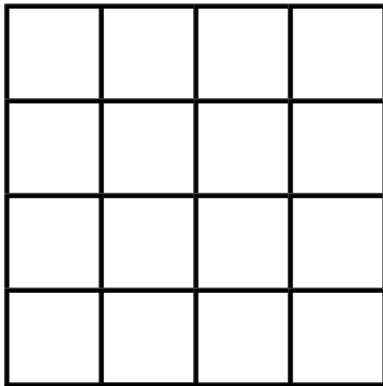
KEY:

[A]	Agent Starting Position
-----	-------------------------

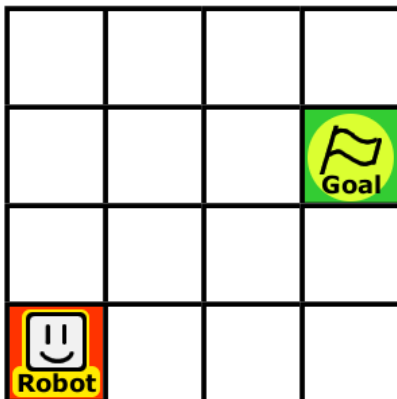
[W]	Wall Square
[G]	Goal Square
[]	Empty Square

How Random Maze Generation Works

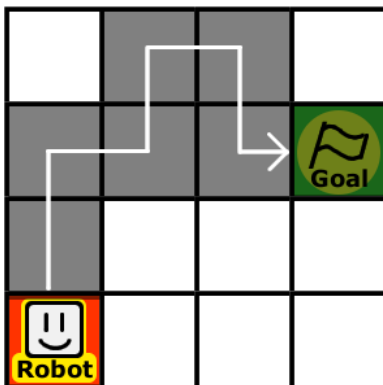
1. Get maze dimensions and number of goals – for example **[4,4] with 1 goal**
2. Generate an empty maze of the specified dimensions



3. Place the player agent and goal tiles at random positions within the maze



4. Generate a random path from the player to one of the goal squares.

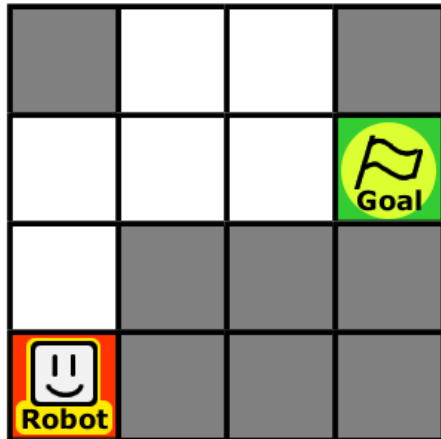


To achieve this, we use the **CUS1** search algorithm to find a path between the player and a goal node ([see Search Algorithms – CUS1](#)). **CUS1** is an uninformed search which **randomly selects nodes from the frontier for expansion**. This means that the path from the player to a

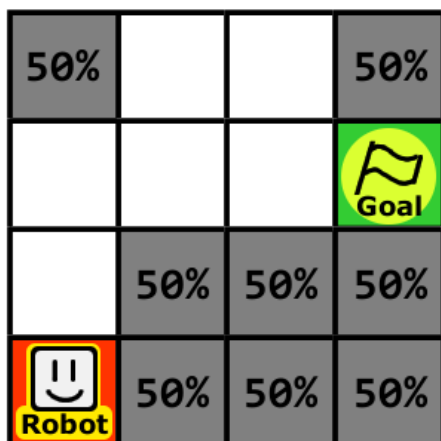
goal node will be randomised and not linear. We need to ensure that there is a clear path from the player to at least one goal square for the random maze to be solvable, otherwise there is a possibility that all goal squares could be unreachable, and the agent would never be able to solve the maze.

5. Now that we have a **guaranteed path from the player to a goal square**, we can fill in the remaining tiles with walls.

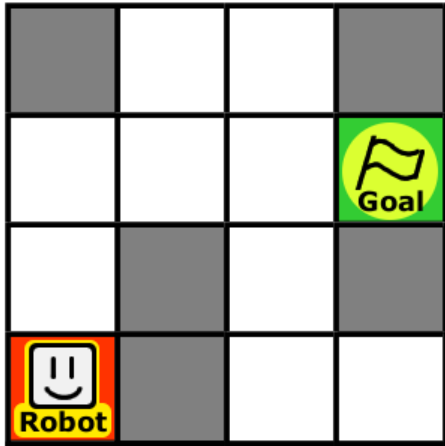
While we could simply fill in **all** non-path tiles with walls, it would make mazes very easy for the agent to solve, because there would already be a carved-out path from the agent's starting position to a goal tile. There would be no paths in the maze that lead away from the goal, or dead ends.



Instead, we give each non-path tile a **50% chance** of becoming a wall, like so:



Which may generate a result like this:



Conclusion

The best search algorithm for the RobotNavigator problem is A*. It uses heuristic values to find the optimal path to the goal, so it doesn't waste time exploring paths which do not lead to the goal. A* is a better heuristic search than GBFS because it considers a node's path cost as part of the evaluation value in addition to its heuristic value. This prevents the search algorithm from becoming stuck exploring suboptimal branches of the search tree when the heuristic value appears optimal. The BFS algorithm could have better performance if repeated states were pruned but it may yield different results. Search algorithms which employ repeated-state checking are generally time and space efficient, particularly if they use a heuristic value.

Glossary

- **Search Tree** – a data structure where states of the world are encapsulated as nodes and placed in a tree structure
- **Node** – the fundamental unit of a search tree. Has a depth, a state, and an action.
- **Child Node** – a node of the search tree which has a parent.
- **Leaf Node** – a node of the search tree which has not been expanded
- **Branch** – a group of nodes within the search tree which
- **Goal State** – a state which meets the goal conditions
- **State Space** – all possible states
- **Uninformed Search Algorithms** – search algorithms which do not utilise the states of nodes
- **Informed Search Algorithms** – search algorithms which utilise the states of nodes to inform their search
- **Heuristic Values** – a value used to calculate the desirability of a state
- **Successor Function** – a function which returns all possible child nodes from a parent node

References

Wgullyn (2021) *A* pathfinding algorithm navigating around a randomly-generated maze*, *Wikimedia*. Available at: <https://commons.wikimedia.org/wiki/File:Astarpathfinding.gif> (Accessed: April 21, 2023).