

Assignment 1 - Tree-based Search

Ta Quang Tung - 104222196

Contents

Instructions	1
Prerequisite	1
Command syntax	1
Output format	2
Further notes	2
Introduction	2
Problem definition	2
Search terminology	2
Search algorithms	4
Breadth-first search	4
Depth-first search	4
Iterative deepening depth-first search	5
Greedy best-first search	5
A* search	5
Iterative deepening A* search	5
Implementation	6
Breadth-first search	6
Depth-first search	7
Iterative deepening depth-first search	7
Greedy best-first search	8
A* search	9
Iterative deepening A* search	10
Features	11
Research	11
Procedurally generating the map with Perlin noise	11
Identifying “islands” and picking the start and goal positions	12
Conclusion	13
Acknowledgements	13
References	13

Instructions

Prerequisite

This program is written in C# and requires dotnet **7.0** to run. Please install it [here](#).

Command syntax

To run the program using the map given by the assignment, run:

```
search RobotNav-test.txt <search_method>
```

where search_method is one of bfs, dfs, gbfs, as, ids, idas.

To run the program using a custom input map file, run:

```
search <file_name> <search_method>
```

Note: If you want to use your own input map files, please put them in the input folder on the same level as the search.bat file.

To run the program with a randomly generated map, run:

```
search rangen <method> <width> <height>
```

where width and height specifies the width and height of the map. These dimensions must be integers greater than or equal to 5. The map will be automatically printed to a text file in the input/ folder for future reuse.

Output format

If the input command is correct, the program should produce the following on the console:

- A line indicating the name of the input map file (rangen if the map is randomly generated), the search method, and the number of nodes created in the search tree.
- A colored map showing the walls, empty tiles, start position, goals, and the path discovered by the search algorithm.

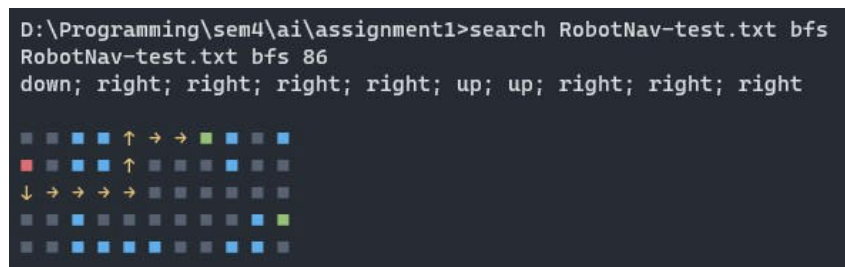


Figure 1: The program's output.

Further notes

As noted previously, if you wish to use your own input map files, you have to put them in the input folder, which is on the same level as the search.bat file. When specifying files in the command, you do not need the input/ prefix. The program will automatically look for the input file in that folder.

Due to some console's buffer width limit, maps that are too large might not be drawn correctly.

Introduction

The purpose of this report is to discuss the robot navigation problem and the search algorithms I have implemented to solve it. Additionally, it will also cover the extension feature I have developed for the program, which is procedural map generation.

Problem definition

The robot navigation problem takes place in a $M \times N$ ($M > 1, N > 1$) grid where M is the number of columns and N is the number of rows. A cell (x, y) on the grid ($0 \leq x < M, 0 \leq y < N$) can either be an empty cell or a wall. One of the empty cells is chosen as the starting position of the robot, and N ($N \geq 1$) other empty cells are chosen as the goal positions. The task is to find a path from the starting position to one of the goal positions while avoiding walls.

Search terminology

The robot navigation problem can be solved with a search algorithm. This section presents a list of terms that are important to the discussion of search algorithms in the sections that follow. Many of these definitions have been taken from (Russell and Norwig 2009).

- **State:** A state captures the state or condition of the world at any given moment. The information stored in a state should only be relevant to the search problem. For this problem, we define a state

as capturing the robot's position at any given moment. For example, (1, 1) is a state. Other pieces of information such as the goal positions and wall cells are stored elsewhere even though they are also relevant to the problem.

- **Action:** An action is something the agent can do to achieve a certain effect. For this problem, we limit the actions of the robot to going Up, Down, Left, or Right. An action is valid at a given state if it does not cause the robot to land on a wall cell or outside the map. As per the requirements, when all else is equal, the actions should be executed in the order of Up, Left, Down, and Right. Each action also has a cost (sometimes called **step cost**), which indicates how expensive it is to execute. We assign all actions a cost of 1 for this problem.
- **Search tree:** Search aims to consider several possible action sequences to see which ones lead to the goal. These action sequences form a search tree with the root as the initial state.
- **Node:** A node is a unit of the search tree. Nodes are connected by actions to form the search tree. We consider each node to store a state, an action that allows the state to be reached from a previous state, a parent node, and other information related to the search algorithm. The root node, which is the first node of the search tree, contains the initial state. It does not have an action and a parent node.
- **Path:** A path can be thought of as a sequence of actions that string together a sequence of states. In the search tree, a path can be visualized as the edges running along a sequence of connected nodes.
- **Path cost:** Since a path consists of a sequence of actions, the path cost is simply the sum of its actions.
- **Solution:** A solution for the robot navigation problem is a path that takes the robot from the start to one of the goals. In this program, the solution is implicitly stored in a node that contains the goal state. Because nodes contain actions and are linked to their parents (except the root node), one can trace back the sequence of actions from the goal node to the root node.
- **Search algorithm:** A search algorithm solves a search problem, which in this case is the robot navigation problem. At any given state, the search algorithm must find valid actions, apply these actions to generate new states and choose which new state to explore until it finds the goal state.
- **Expansion:** As the search algorithm works to find the solution, it implicitly constructs a search tree consisting of nodes. At each point during the execution, the algorithm has to select a node (which contains a state) to explore. It will identify the valid actions at that state, and then apply the actions to generate new nodes (which contain the new states) in a process called expansion.
- **Frontier:** The frontier contains the nodes available for expansion at any point of the algorithm. How the search algorithm chooses which node from the frontier to explore next defines its strategy.
- **Explored set:** The explored set contains the nodes that the search algorithm has expanded. It is used in situations where there can be redundant paths that cause infinite loops
- **Branching factor:** The branching factor b of the search problem is the maximum number of child nodes a node can have in the tree.

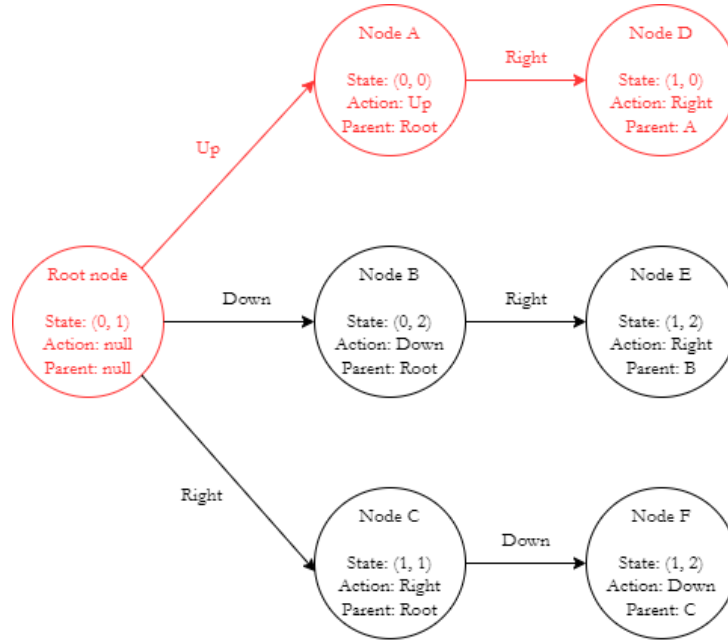


Figure 2: A partial search tree of the robot navigation problem. The red portion of the tree represents a path consisting of actions Up and Right. Its path cost is 2.

Search algorithms

A total of 6 search algorithms have been implemented for this program. Three of these (breadth-first search, depth-first search, and iterative deepening depth-first search) are uninformed search algorithms. The other three (greedy best-first search, A* search, and iterative deepening A* search) are informed (heuristic) search algorithms.

The heuristic used by the informed strategies is the Manhattan distance heuristic. The Manhattan distance L between two points (x_1, y_1) and (x_2, y_2) is expressed as: $L = |x_1 - x_2| + |y_1 - y_2|$. The Manhattan heuristic for each position of the robot navigation problem is the smallest Manhattan distance between that position and one of the goals.

Breadth-first search

Breadth-first search (BFS) selects nodes to expand in the order they were put into the frontier. Nodes that come in first get expanded first. In other words, nodes that are shallowest in the search tree get expanded first. This can be achieved by using a Queue for the frontier.

Due to the shallow-first nature of BFS, it is guaranteed to produce the optimal solution if all step costs are equal, which is actually the case for this problem. However, its major drawbacks are its space and time complexities, both of which are $O(b^d)$, where d is the depth of the solution (Russell and Norwig 2009).

Depth-first search

Depth-first search (DFS) works opposite to BFS, expanding the nodes that come in last first. In other words, nodes that are deepest in the search tree get expanded first. This can be achieved by using a last-in-first-out queue (also known as a Stack) for the frontier.

Because DFS will keep going down to the maximum depth of the search tree until it finds the first goal node, the solution it produces may not be optimal. In fact, while testing this program, DFS produces arguably the most convoluted solution out of all search algorithms. Additionally, its time complexity is no better than BFS, standing at $O(b^m)$ where m is the maximum depth of the tree. Despite this, its saving grace is a modest space complexity of $O(bm)$ (Russell and Norwig 2009).

Iterative deepening depth-first search

Iterative deepening depth-first search (IDDFS) repeatedly calls depth-limited search with an incremental depth limit until it finds a solution or discovers that there is no solution at all. Depth-limited search is a modified version of depth-first search which terminates at a certain depth limit. Depth-limited search can return a solution, a failure, or a cutoff indicating that no solution is found within the given depth limit. Upon receiving this output, IDDFS can return a solution/failure or increment the depth limit and run depth-limited search again with the new limit.

IDDFS combines the benefits of BFS and DFS. Like BFS, it completely explores one layer of nodes in the search tree after the next, allowing it to produce an optimal solution if all path costs are equal. Like DFS, its space complexity is a modest $O(bd)$, where d is the depth of the shallowest goal node (Russell and Norwig 2009).

One drawback of IDDFS is that it can generate nodes on the same layer of the search tree multiple times because of the repeated calls to depth-limited search, which can be rather wasteful.

Greedy best-first search

Greedy best-first search (GBFS) expands nodes with the smallest heuristic value first. It assumes that if the heuristic value is small, the node will be closer to the goal and can reach it faster. This can be achieved by making the frontier a priority queue with the priority being the heuristic.

GBFS is not optimal because it does not take into consideration the actual path costs. A node can have a small heuristic value yet cost much more to reach. Additionally, the time and space complexities of GBFS depend greatly on the heuristic.

A* search

A* search expands nodes with the smallest f -cost first. The f -cost of a node is the sum of its path cost and heuristic value. This can be achieved by making the frontier a priority queue with the priority being the f -cost.

A* search is optimal provided that the heuristic function $h(n)$ is consistent, which means that for every node n , child node n' , and action a that takes n to n' : $h(n) \leq \text{cost}(a) + h(n')$ (Russell and Norwig 2009). The Manhattan distance heuristic used in this program is a consistent heuristic, so A* is always optimal.

A* search's drawback is its space complexity.

Iterative deepening A* search

Iterative deepening A* (IDA*) search aims to overcome the space drawback of A* by applying the idea of iterative deepening depth-first search. Instead of using the depth limit as the cutoff for the search, IDA* uses the smallest f -cost that exceeded the cutoff limit in the previous iteration.

The table below compares the six implemented search algorithms in terms of their time performance (measured in terms of how many nodes have been created) and optimality running on a 100×50 randomly generated map. The choice of search algorithm will depend on time, space, and optimality requirements. If an optimal solution is desired, depth-first search and greedy best-first search cannot be used. The best uninformed search algorithm is IDDFS, even though it might run for longer than BFS. IDDFS helps overcome the space limitations of BFS for very large problems. The best informed search algorithm is IDA*, running at almost the same time as A* while solving its space limitations. However, it is worth noting that the optimality of A* (and by extension IDA*) is dependent on the heuristic being consistent.

Search algorithm	# nodes created	Optimal?
Breadth-first search	3434	Yes
Depth-first search	7750	No
Iterative deepening depth-first search	214,611	Yes
Greedy best-first search	183	No
A* search	455	Yes
Iterative deepening A* search	455	Yes

Table 1: Comparison among different search strategies in terms of the number of nodes created (time complexity) and optimality.

Implementation

As discussed in the previous section, search algorithms differ mainly in terms of how they choose the next node from the frontier to expand. However, the actual code implementations of the algorithms may vary in regard to the method of keeping track of explored nodes, the generation new nodes, etc. This section of the report presents the code implementation of each search algorithm, written in pseudocode form. The actual program is written in object-oriented C#, but pseudocode will keep things concise.

Breadth-first search

```

function BreadthFirstSearch(problem) returns Result (Solution/Failure) {
    node = Node(problem.InitialState)
    if (node is goal) return Solution(node)

    frontier = new Queue of Nodes
    frontier.Enqueue(node)

    explored = new Set of States

    while (true) {
        if (frontier is empty) return Failure

        node = frontier.Dequeue()
        explored.Add(node.State)

        actions = problem.GetActions(node.State)

        foreach (action in actions) {
            child = Node(node, action)

            if (child.State not in explored AND child.State not in frontier) {
                if (child is goal) return Solution(child)
                else frontier.Enqueue(child)
            }
        }
    }
}

```

This implementation is taken from (Russell and Norwig 2009). The check (child.State not in explored AND child.State not in frontier) prevents exploring repeated states. Some implemen-

tations define explored as a set of nodes instead of states. For BFS, since everything we care about a node is its enclosed state, it makes no difference whether the set keeps track of nodes or states.

Depth-first search

```
function DepthFirstSearch(problem) returns Result (Solution/Failure) {
  node = Node(problem.InitialState)
  if (node is goal) return Solution(node)

  frontier = new Stack of Nodes
  frontier.Push(node)

  explored = new Set of States

  while (true) {
    if (frontier is empty) return Failure

    node = frontier.Pop()

    if (node.State in explored) continue

    explored.Add(node.State)

    actions = problem.GetActions(node.State)

    foreach (action in actions) {
      child = Node(node, action)

      if (child.State not in explored) {
        if (child is goal) return Solution(child)
        else frontier.Push(child)
      }
    }
  }
}
```

The implementation of DFS differs from BFS in two aspects. First, the frontier uses a Stack instead of a Queue to allow LIFO access. Second, the “is in frontier” check of the child node has been removed. During testing, it was found that having this check interferes with the required action order of Up, Left, Down, Right. To prevent infinite loops, a node is checked to see if it has been explored immediately after it is removed from the frontier. If the node has been explored, it is skipped.

Iterative deepening depth-first search

```
function IterativeDeepening(problem) returns Result (Solution/Failure) {
  limit = 1

  while (limit < infinity) {
    result = DepthLimitedSearch(problem, limit)
    if (result is not Cutoff) return result;
    limit = limit + 1
  }
}

function DepthLimitedSearch(problem, limit) returns Result (Solution/Failure/Cutoff) {
  node = Node(problem.InitialState)
  if (node is goal) return Solution(node)
```

```

frontier = new Stack of Nodes
frontier.Push(node)

explored = new Set of Nodes

deepestNode = 0

while (true) {
    if (frontier is empty and deepestNode > limit) return Cutoff
    else if (frontier is empty and deepestNode <= limit) return Failure

    node = frontier.Pop()
    deepestNode = max(deepestNode, node.Depth)

    if (node in explored) continue

    explored.Add(node)

    if (node.Depth <= limit) {
        actions = problem.GetActions(node.State)

        foreach (action in actions) {
            childNode = Node(node, action)

            if (childNode not in explored) {
                if (childNode is goal) return Solution(childNode)
                frontier.Push(childNode)
            }
        }
    }
}

```

The IDDFS function calls depth-limited search with an incremental depth limit starting from 0 to positive infinity. The function terminates when depth-limited search returns a solution or failure.

The depth-limited search function is a modified version of DFS. The function returns a failure if no solution is found within the current depth limit and there are no nodes beyond the limit. It returns a cutoff if there are potential goal nodes beyond the limit. The key difference from DLS is that the explored set keeps track of nodes instead of states. Because now the depth of a node is of interest, it no longer suffices to just keep track of the explored states. If we only allow states to be explored once, we might discard a goal path containing previously explored states within a cutoff limit.

Greedy best-first search

```

function GreedyBestFirstSearch(problem) returns Result (Solution/Failure) {
    node = Node(problem.InitialState)

    if (node is goal) return Solution(node)

    frontier = new Priority Queue of Nodes
    frontier.Enqueue(node, heuristic(node))

    explored = new Set of States

    while (true) {
        if (frontier is empty) return Failure

```



```

node = frontier.Dequeue()
explored.Add(node.State)

actions = problem.GetActions(node.State)

foreach (action in actions) {
    childNode = Node(node, action)

    if (childNode.State not in explored AND childNode.State not in frontier) {
        if (childNode is goal) return Solution(childNode)
        else frontier.Enqueue(childNode, heuristic(childNode))
    }
}
}

```

The implementation of GBFS is the same as that of BFS, the only difference being the replacement of a queue with a priority queue.

A* search

```

function AStarSearch(problem) returns Result (Solution/Failure) {
    node = Node(problem.InitialState)

    frontier = new Priority Queue of Nodes
    frontier.Enqueue(node, heuristic(node))

    explored = new Set of States

    while (true) {
        if (frontier is empty) return Failure

        node = frontier.Dequeue()

        if (node is goal) return Solution(node)

        explored.Add(node.State)

        actions = problem.GetActions(node.State)

        foreach (action in actions) {
            childNode = Node(node, action)

            if (childNode.State not in explored AND childNode.State not in frontier) {
                frontier.Enqueue(childNode, childNode.pathCost + heuristic(childNode))
            }
        }
    }
}

```

A* is implemented in almost the same way as BFS, with the obvious difference being the priority queue. However, a less obvious difference is the position of the goal check. Here, the goal is checked when a node is selected for expansion, not when it is created. This helps avoid situations in which the function returns immediately when the goal node is generated even though there is a better goal.

Some might question the use of the explored set to store states instead of nodes. After all, in this algorithm, we are not only interested in a node's state but also its f -cost. One might reason that if we only

allow a state to be checked once, we might miss better paths that lead to that state. This is true if the heuristic is not consistent. However, because the program uses a consistent heuristic, the first obtained path to any state is guaranteed to be optimal, thus making other paths to the same state redundant.

Iterative deepening A* search

```
function IterativeDeepeningAStarSearch(problem) returns Result (Solution/Failure) {
    cutoffLimit = 0

    while (cutoffLimit < infinity) {
        result = AStarLimited(problem, cutoffLimit)
        if (result is not Cutoff) return result
    }
}

function AStarLimited(problem, cutoffLimit) returns Result (Solution/Failure/Cutoff) {
    node = Node(problem.InitialState)

    frontier = new Priority Queue of Nodes
    frontier.Enqueue(node, heuristic(node))

    minExceed = heuristic(node);

    while (true) {
        if (minExceed <= cutoffLimit and frontier is empty) return Failure
        if (minExceed > cutoffLimit and frontier is empty) {
            cutoffLimit = minExceed
            return Cutoff
        }

        node = frontier.Dequeue

        if (node is goal) return Solution(node)

        actions = problem.GetActions(node.State)

        foreach (action in actions) {
            childNode = Node(node, action)

            if (childNode.State is not in frontier) {
                fCost = childNode.PathCost + heuristic(childNode)

                if (fCost > cutoffLimit) {
                    if (minExceed <= cutoffLimit) minExceed = fCost
                    else minExceed = min(minExceed, fCost)
                }

                if (fCost <= cutoffLimit) frontier.Enqueue(childNode, fCost)
            }
        }
    }
}
```

IDA* simply combines the techniques of IDDFS and A* search.

Features

As part of this assignment, all six search algorithms described above have been implemented. A batch file containing the terminal commands to run the program correctly is also included. Please refer to the Instructions section for more information on how to run the application.

Additionally, as an extension to the base requirements, this program also features the option to randomly (or more accurately, procedurally) generate a map. Details on how this has been achieved is described in the Research section.

Research

The extension implemented for this program is the procedural generation of maps with Perlin noise. This extension allows new maps to be created on the fly whenever the program is run. Overall, this feature consists of three steps:

1. Using Perlin noise to generate walls and empty cells.
2. Identifying blocks of connected empty cells (or “islands”) using BFS.
3. Finding the biggest island and picking the start and goal positions.

Procedurally generating the map with Perlin noise

Perlin noise is an algorithm developed by Ken Perlin in 1983 to generate computer textures. In addition to its original purpose, the algorithm has been used to procedurally generate terrains in video games and randomize textures.

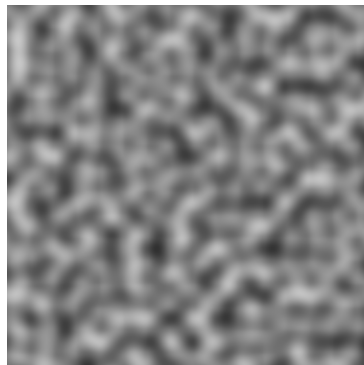


Figure 3: The noise pattern generated by Perlin's algorithm.

The algorithm can be used for many dimensions, but for our use case, we will focus on 2D noise. The algorithm begins with a two-dimensional grid in which each grid intersection is associated with a random vector.

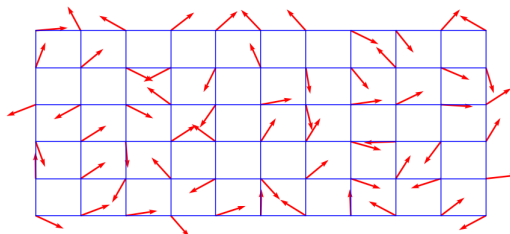


Figure 4: The 2D grid. Each red vector is associated with an intersection point.

To calculate the Perlin noise value for any given point (x, y) on the grid, we first have to identify which grid cell that point belongs to. For example, the point $(9.12, 3.6)$ belongs to cell $(9, 3)$. For each corner of that cell, we then calculate the offset vector to (x, y) , which is a vector from that corner to (x, y) .

We then take the dot product of that offset vector with the random vector associated with the corner to obtain a scalar value. Repeating this for all corners we get 4 scalar values for one cell. Finally, we interpolate between these values to get the final noise value.

With the Perlin grid set up and all of its operations defined, we need to think about how we can generate our robot map. Because the input of the Perlin noise function is two numbers representing a point on the Perlin grid, we need to find a way to map our robot map coordinates to a pair of coordinates on the Perlin grid. There are many ways to do this. One could simply use the robot map coordinates as the coordinates on the Perlin grid directly, but it is common to multiply the robot map coordinates with a frequency value f ($0 < f < 1$). A random offset input can be further added to the resulting coordinates to randomize the position of the Perlin grid to sample from. The result of the noise function is a real number, often between -1 and 1 or 0 and 1. Our program chooses the range 0 and 1. After obtaining this value for each cell of our robot map, we decide if the cell is a wall if the value exceeds a certain threshold value. The result of this process looks like this:

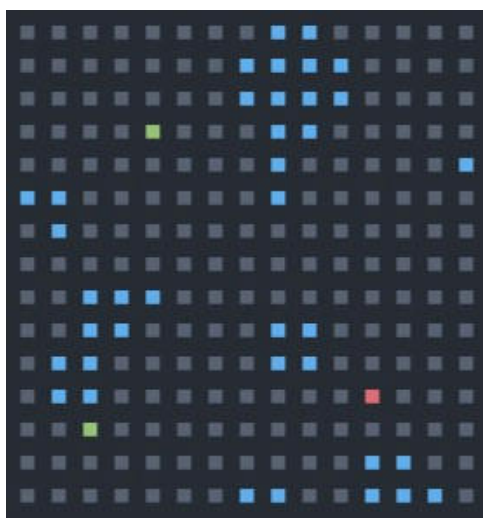


Figure 5: The procedurally generated map. Blue cells are walls, the red cell is the start position and the two green cells are the goals.

Notice how the wall cells in this map form noticeable clusters and not dispersed randomly. This is the benefit of using Perlin noise. The map makes sense and resembles a real environment.

Identifying “islands” and picking the start and goal positions

Having generated the walls and empty cells, one may be tempted to immediately pick the start and goal positions from the empty cells at random. However, our procedurally generated map may contain a region that is enclosed entirely within a wall. If we pick a start position in this region and the goal positions outside or vice versa, the problem will have no solution. Of course, this is an entirely valid map, but what insight would we gather from trying to solve an impossible map? Therefore, it is important we ensure that a solution exists when the map is generated.

To do this, we will need to identify “islands”, or blocks of connected empty cells. Cells are considered to be connected if they share one of the four cardinal sides (up, down, left, right). In the map above, all the gray cells form a single island; however, some maps can contain multiple islands separated by walls.

The way we identify these islands is to use breadth-first search. We first start with a 2D boolean matrix representing the map where a cell is false if it is an unvisited empty cell and true if it is a visited cell or a wall. We then loop through every value in the matrix. If we encounter an unvisited cell, we run BFS on that cell. BFS will explore its surrounding cells, then the surrounding cells of its surrounding cells, etc. until it is out of neighboring cells to explore. The exploration procedure returns a list of cell

positions all belonging to a cluster or “island”. Additionally, as it is executing, the procedure will mark unvisited cells as visited in the 2D matrix.

The result of this process is a list of “islands”. As the last step of map generation, we identify the largest island (because it is more interesting to play with) and randomly pick a start position and non-overlapping goal positions.

Conclusion

The report has described my efforts to solve the robot navigation problem as part of Assignment 1. I have covered six different search strategies implemented in the program, including their operations, strengths, and weaknesses. For a problem like the robot navigation problem, I would argue that the best search algorithm is IDA*. Not only does it find the optimal solution but it also has one of the strongest time and space performances out of all the algorithms discussed. However, I will need to experiment further to explore the limits of IDA*.

This assignment has also given me the opportunity to explore Perlin noise, which has been used as part of my extension. Originally created for generating computer graphics, it has found application in many other areas, one of which being procedural terrain generation which is directly related to the robot navigation problem. This extension leaves plenty of room for experimentation with map generation, as changing the frequency and offset inputs of the noise function can result in interesting map configurations.

Acknowledgements

The following section lists the resources that have helped me complete this program.

- *Perlin Noise: A Procedural Generation Algorithm - Raouf's Blog* - This article has helped me understand the operations of Perlin noise and develop my Perlin noise implementation.
- *Understanding Perlin Noise - Adrian's Soapbox* - This article has helped me understand the operations of Perlin noise.
- *Artificial Intelligence: A Modern Approach (Third Edition)* by Stuart J. Russell and Peter Norvig - This book is where I get the relevant terms and definitions from. It has also helped me develop a conceptual understanding of the different search algorithms and aided me in their implementation.

References

Russell, S., & Norvig, P. (2009). *Artificial Intelligence: A Modern Approach (Third Edition)*.