Robot Navigation Problem
COS30019

Cormac Collins
Student id: 100655400

Contents

Introduction:

1.1 Robot Navigation Problem

We have been presented with the typical NxM solve problem which requires an algorithm to find a given goal state within the given NxM amount of states, this is often viewed as tree (figure 1) or graph states(figure 2). Calculating the number of maximum potential states in a simple tree diagram version can be calculated as: $b^{\wedge d}$, *b* representing the average branching factor to the power of the d, the depth of the shallowest goal or the 'depth factor'. As the number of states or in search terms 'nodes' begins to grow, you can see the potential area to traverse becomes quite large.

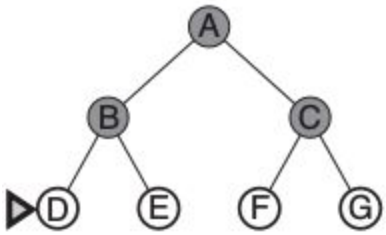


Figure 1: Tree diagram example of n*m, $2^{\wedge 2}$ = 4 states.
Russell & Norvig (2010, p82)

If either *b* or *d* grow excessively then we can see the task of reaching our goal become significantly harder. Additionally, if 'repeated states' were to occur, for example 'g' looped back to 'a' (figure 1) then the task become more problematic, a search algorithm without any heuristics to deal with this will likely enter an infinite loop, heuristics will discussed further later. This repeated state is present in any form of 'graph search'. In figure 2 is again an NxM problem, the start represented by the blue square and the end by the green square. As you can see, there are far more options for paths and more ways to enter infinite loops given the wrong search pattern.

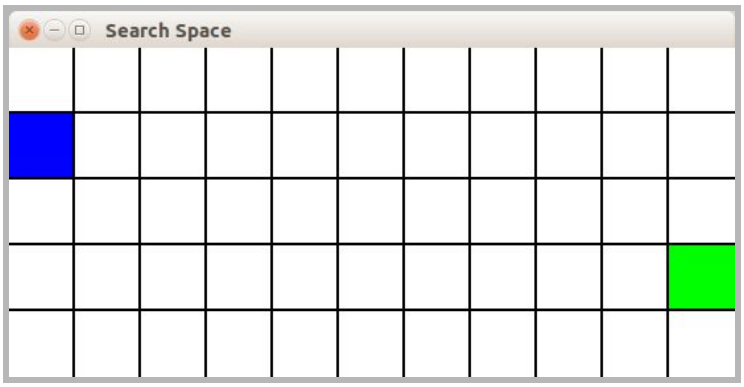


Figure 2: Graph diagram example of n*m, $2^{\wedge 2}$ = 4 states.

1.2 Setting up a problem

The important concept for applying search and AI algorithms to problems is knowing how to ‘set-up’ the problem, this approach may appear simple once working through the graph search, however it becomes an important skill when approaching more abstract non-visual state spaces.

Key search-problem information:

- Initial starting state
- Goal state
- Action type (i.e. are we taking a direction? Or making a calculation to solve a problem?)
- Path Cost (for informed searches - how to do we know which next state is a ‘better state’, if we are aiming for the ‘best’ solution, we would want the least path cost.

Nodes: Each search problem, whether a tree diagram, graph diagram or more abstract problem can all be constructed as a list of nodes. These nodes become the key individual representations of different states along the path (see figure 3 for graphical representation). A node generally needs the qualities of:

- Parent (the node that was used to reach the current node)
- It’s state (i.e. unique number, position in grid, amount calculated in mathematical formulae).
- A number of ‘Actions’ (For graph search this will be directions e.g. left or right)
- Path Cost: The value it has cost to reach this node from its parent
- Goal value: In informed searches only; this can be a value that represents a potential distance from the goal,

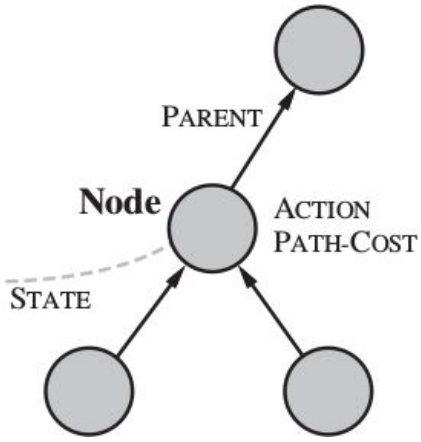


Figure 3: Search tree node. Russell & Norvig (2010, p79)

1.3 Traversing a problem

There are several key implementations to traversing Nodes in a search space (see glossary for definitions). These include:

- Frontier
- Explored Path
- State List

Search Algorithms:

2.1 Search Code Structure

The path finding code structure follows a simple OOP inheritance design for the search algorithms themselves, with an algorithm inheriting the primary tools needed for a search algorithm form the ‘SearchType’. Many of these functions were virtual for overriding between the different search requirements.

The problem structure itself, favoured composition, with the actual problem itself passed as a class containing the relevant information (i.e. goal state etc.) and traversal data was stored in vectors of ‘Path’ which contain the state Node (pointer) and the action taken to get there.

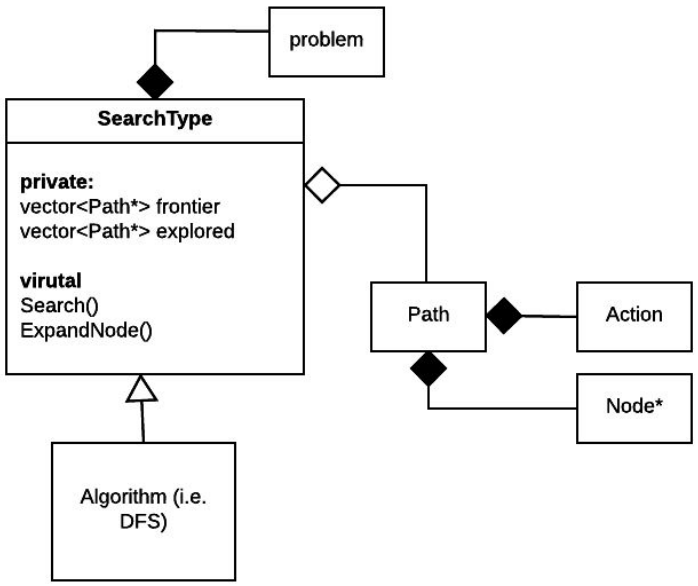


Figure 4: UML generalization of search algorithm code

The code follows a theoretical structure similar to that of Russell & Norvig (2010, p78) for reproducibility and proper implementation of Search standards. The generic structure for all algorithms is shown below in Figure 5:

```
Function Search(problem, searchNode)

    do
        ChildrenAndActions = ExpandNode(searchNode, problem)
        Frontier ← ChildrenAndActions
        Explored ← ChildrenAndActions

        // if uninformed
        NewPath ← PopFrontier()
        // else if informed
        NewPath ← HeurisiticSearch(Frontier)

        Solution = GoalCheck(problem, NewPath.Node)
        If Solution.Outcome == "success"
            Return Solution
    while(FrontierNotEmpty())

    Return Solution;
```

Figure 5: Generic search structure

The order and implementation will differ slightly between the different searches, particularly informed searches that must use their heuristic. The crux of 'Graph' exploration is performed within the ExpandNode() function. See the appendix for these base functions and an example of the search function implemented for A*.

2.1 Uninformed Search

Depth First Search (DFS):

Basic properties

LIFO (last in first out) frontier: When DFS reaches a node and Expands its children, it places them inside it's frontier. To retrieve the next node for searching it operates with a 'Last In First

Out' (LIFO) queue. In our Graph Search space DFS expands its children in the order of up, left, down, right (this is not necessarily specific to DFS implementation). Therefore, when it comes time to search a new node from our frontier, we will pop the next node off the queue using LIFO; which in figure 5 will be 'right' for our first. This in its entirety is how DFS searches.

Behaviour

DFS operates at $O(b^m)$, where m is the maximum depth, it may then in fact search every node in the graph to reach the goal. This is clearly very inefficient. As problems become larger, DFS will much more likely fail due to timing out or (in the case of an infinite space or infinite loops and no timeout clause) crash from excess memory consumption. The primary positive of DFS is that the memory complexity is kept linear at *O(bm).*

We can observe how DFS can quickly fail when a problem changes.

As we can see in the comparison from 'Search 1' to 'Search 2' we have made the map bigger and changed the goal location. Because of DFS's uninformed search pattern it simply keeps expanding nodes until it find what it wants. Also given some difficult walls, DFS will likely end up in an infinite loop.

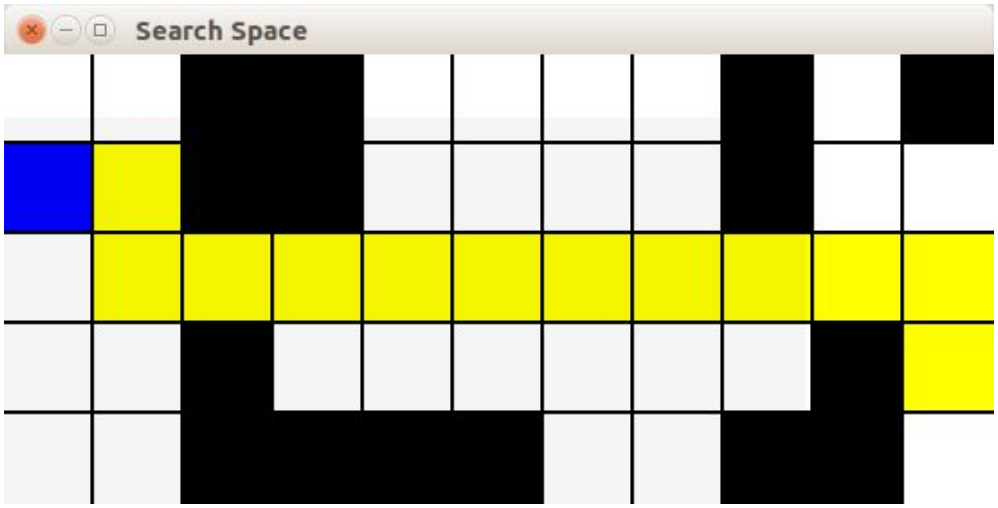


Figure 5: DFS Search 1 - **NxM = 11*5**
Nodes explored = 12

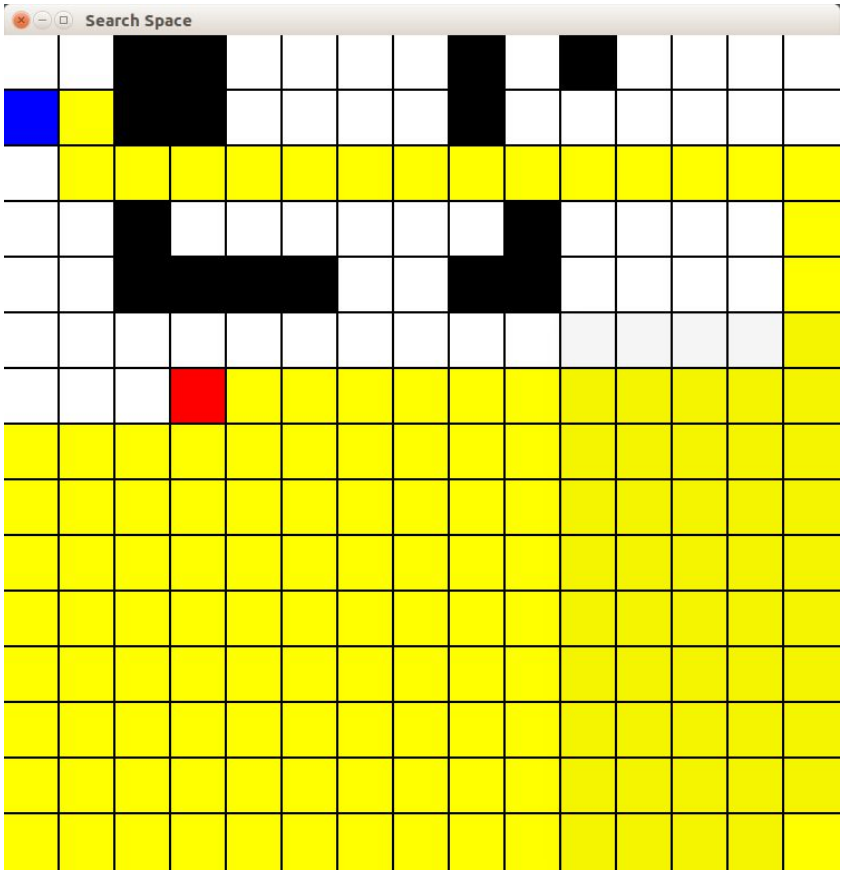


Figure 6: DFS Search 2 - **NxM = 15*15**
Nodes Explored = 150

Breadth First Search (BFS):

Basic properties

FIFO (first in first out) frontier: Unlike DFS, BFS uses a FIFO frontier when it reaches a node and Expands its children (see glossary), it then places them inside it's frontier. Unlike DFS, whose first choice from the frontier was important in deciding where it would next expand to, this does not matter for BFS because BFS will now expand every node in the frontier (Goal checking each time before expanding). If you were to look at the figure 1 tree diagram you could imagine that BFS looks at every node in each row as it traverses downwards.

Behaviour

BFS operates at $O(b^d)$, where b is the average branching factor and d is the depth of the solution. BFS will therefore always find the solution, however in guaranteeing a solution, BFS sacrifices both run time complexity (expanding all nodes at each depth) and space complexity

$O(b^d)$. If provided with an NP problem, BFS is incomplete if hardware capacity can not cope with the storage required for reaching the goal given d.

We can see in figure 7 the 'explored' nodes from BFS on the same 11*5 problem completed by DFS. BFS expands 39 nodes to find the solution, for a given problem size we can see the exponential rise in nodes explored/distance from goal as the problem size increases.

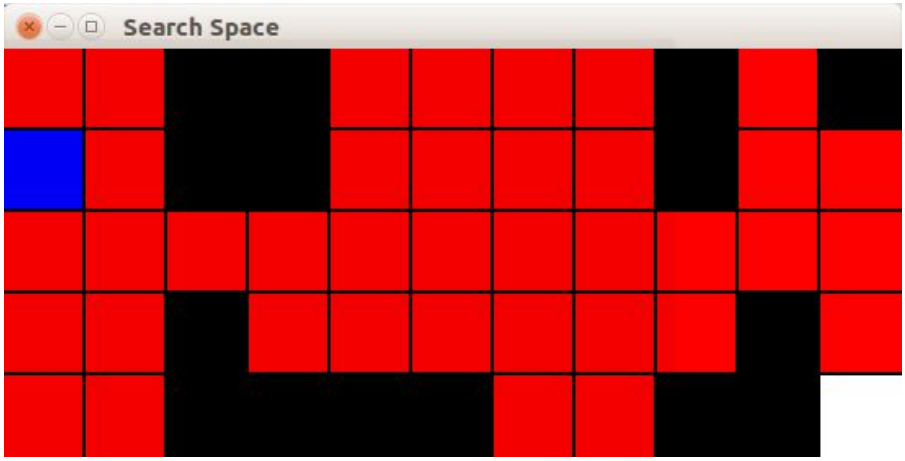


Figure 7: Search 1 - **NxM = 11*5**
Nodes Explored = 39

2.2 Informed Search

A*:

Basic properties

We have previously mentioned that informed searches differ due to their heuristic function, therefore in A*'s case, the frontier queue is not so important. A* works using a heuristic function for each new node:

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

Where g(n) represents the cost to reach a given node n, and h(n) represents the 'Goal Cost' which is the cost remaining to reach the goal. Given a graphical traversal problem such is ours, we will be looked for the lowest h(n), i.e. the shortest distance to the goal.

A* will always find the shortest path due to this heuristic, because for each iteration it will compare all f(n) of the nodes in its frontier and expand the 'best' one. However A* suffers a drawback for the capacity to operate like this, it must store a higher amount in its frontier and

explored path as it needs to check future node h(n) values against previous ones to absolutely know which is the best path.

Behaviour

A* actually operates at $O(b^d)$ the same as BFS, however importantly to note, this is the worst case. It is important to point out that our current NxM solution map/problem operates with step costs of 1 for every jump between nodes. If we were say, travelling through maps and using different distances between cities then A* becomes even more efficient, because, when BFS would expand each level of nodes to reach the chosen city, A* will only expand more nodes if their cost was better, allowing it to operate far below the worst case of $O(b^d)$. Thus, in the right problem, A*'s best case O(n) is much lower than its worst.

We can see in figure 8, to find the optimal path A* still searches the same amount as BFS, this is an example of the cases where A* gets close to its worst case of $O(b^d)$.

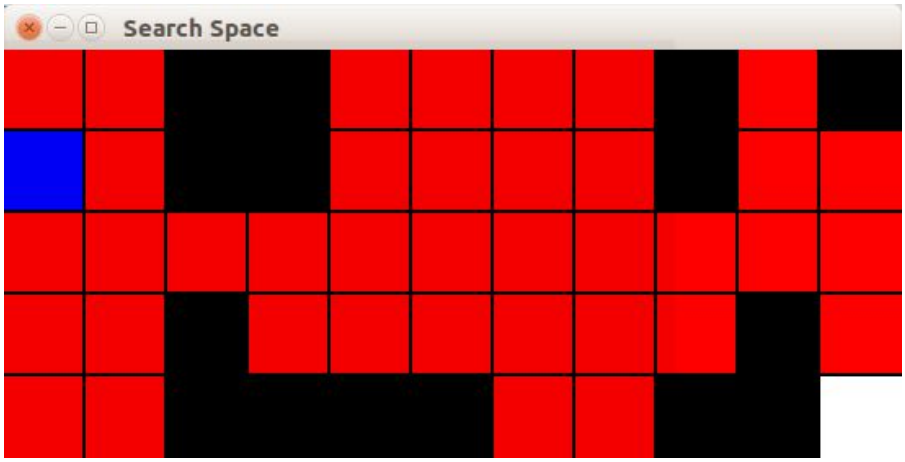


Figure 8: A* - **NxM = 11*5**
Nodes Explored = 39

However we can see in figure 9 when A* and heuristic search becomes optimal, we are both guaranteed to find the solution (given a finite depth) and can also find the best path in reasonable space. We can still see with this search pattern that as MxN becomes large, A* requires significant amounts of memory (this can be partially fixed in our future implementation MBA*).

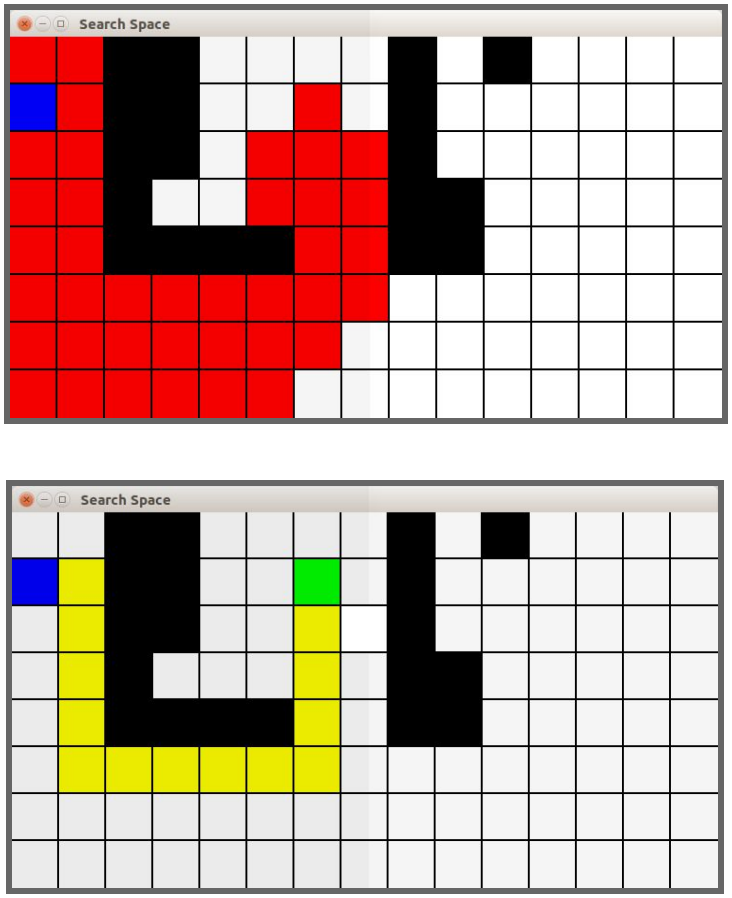


Figure 9: A* - **NxM = 15*8**
Nodes Explored = 39

Greedy Best First Search (GBFS):

Basic properties

GBFS follows a similar heuristic to A* however chooses not to check the path cost g(n). GBFS simply assess the best node children recently expanded and picks the best h(n) without regard to the path it has already taken g(n) - thus the name greedy.

f(n) = h(n)

Because of this disregard for previous paths GBFS is not admissible (see glossary).

Behaviour

GBFS operates at $O(b^m)$ where m is the maximum depth space. This exponential runtime occurs due to the potential that given its greedy nature and a graph search that might have ‘walls’ it could continue searching in the wrong direction for a long time - we will look at this in the below figures.

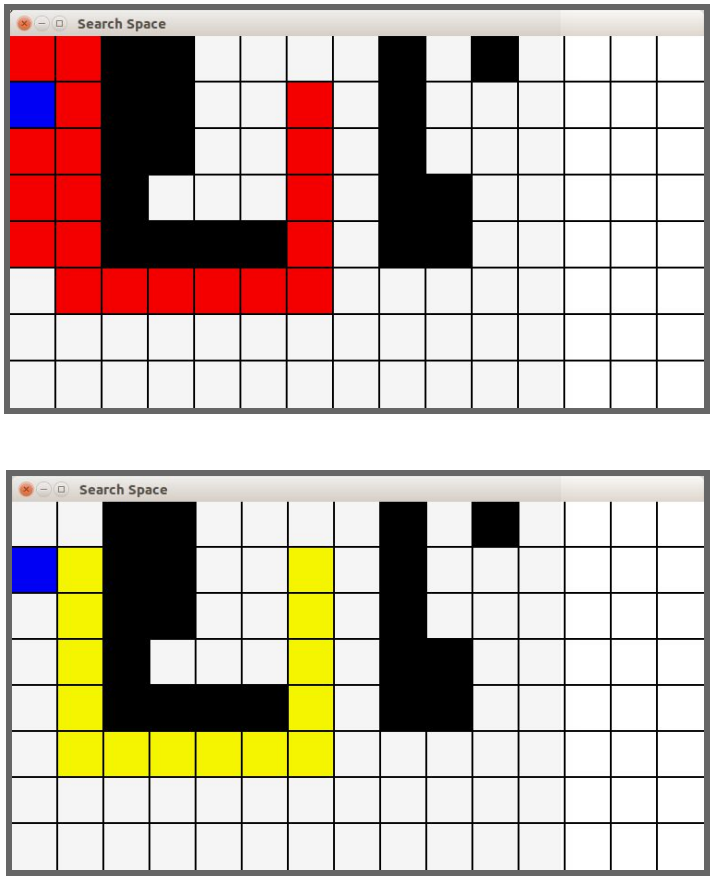


Figure 10: GBFS - **NxM = 15*8**
Nodes Explored = 21

In figure 10 we can see the effectiveness in being greedy for GBFS, in this particular problem the goal was found very quickly at little Node Exploration cost, and particularly at little storage cost. However in figure 11 we can see how GBFS can quickly become inefficient finding the final goal, by not having a ‘remembered’ list of paths GBFS now doesn’t have the calculated cost it has already taken when it takes the wrong turn, when A* would simply observe both paths as it continues to test the best h(n).

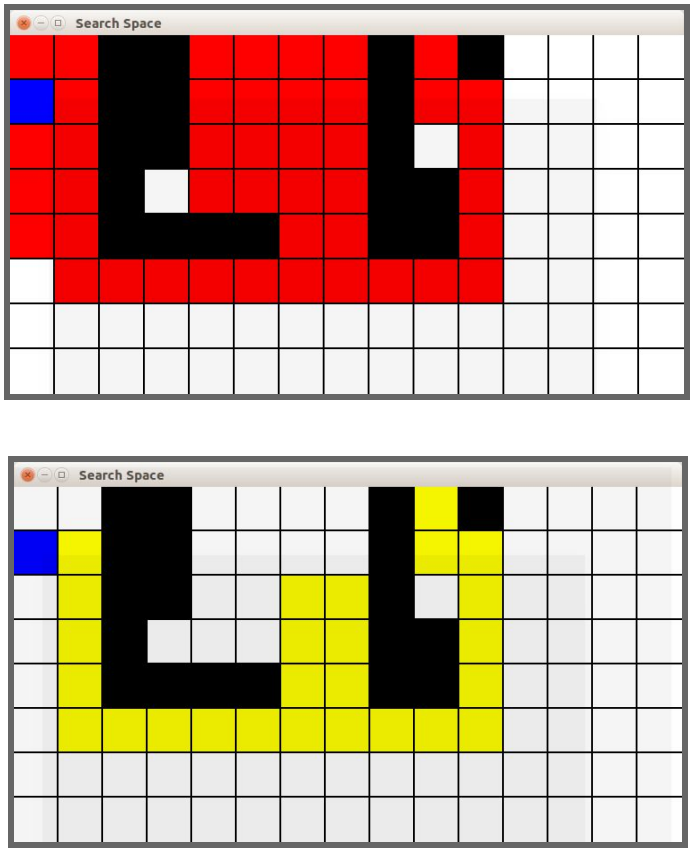


Figure 11: GBFS - **NxM = 15*8**
Nodes Explored = 78

2.3 Custom search

**Due to bugs within both custom searches they will not be covered as rigorously, these bugs will be outlines.*

MBA*

Basic properties

MBA* operates the same as A* with 1 minor change, it operates at a finite ‘explored’ set. That is given a specified maximum for storage, it will discard the worst h(n) node in its current explored set. Allowing a very low storage requirement. Thus MBA* becomes a merge of GBFS and A* removing some of the excess storage of A* while keeping inadmissibility from GBFS - because it does not store everything it may take the wrong path and ‘forget’ the h(n) of previous nodes that now may be better options.

As we can see below MBA* still searches the same amount of nodes to reach its goal, however it's 'explored' set is remaining truncated. This appears to be the early stages of progression towards Hill Climbing Algorithms, however they are designed to handle the fact that a path may become redundant.

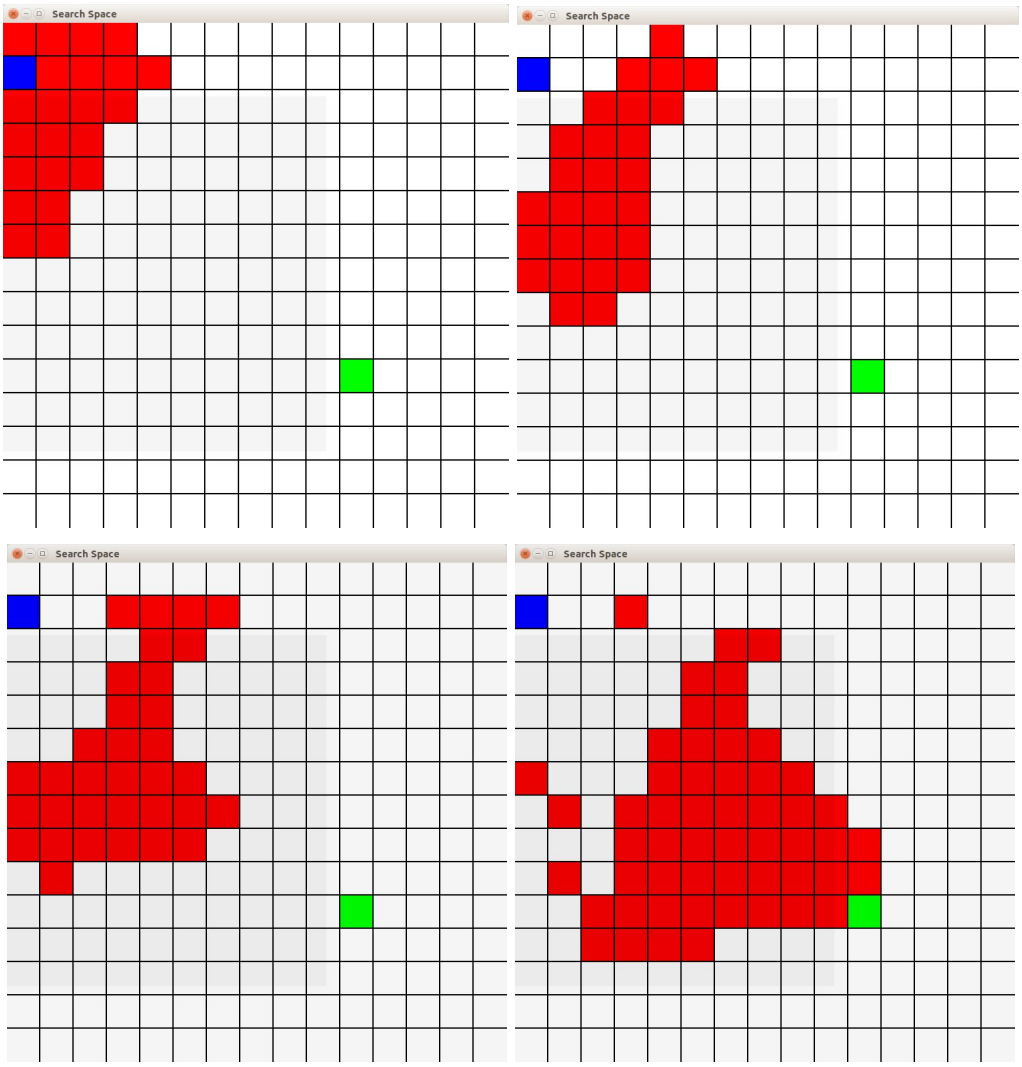


Figure 12: MBA*- **NxM = 15*15**
Nodes Explored = 123

Bi-Directional search (BDS):

Basic properties/runtime

BDS is essentially 2 algorithms working at once from each end of the state space, i.e. one from the start and one from the goal. The primary improvement on this search is runtime; by having 2

searches at once we can potentially change $O(b^d)$ to $O(b^{d//2})$. Of Course this relies on them following the same search heuristic, which in our case using A* will guarantee this.

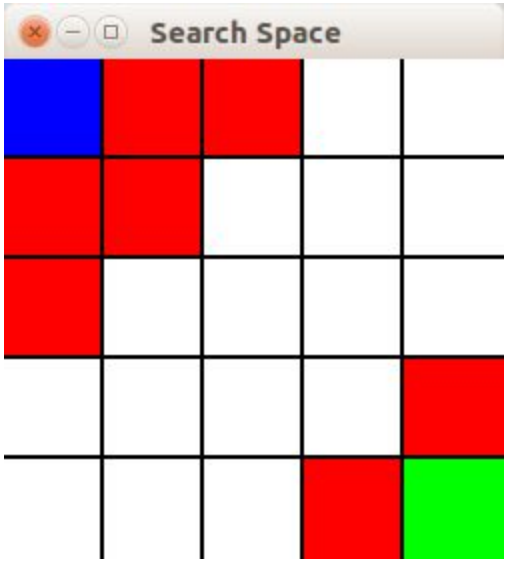


Figure 13: Bi-directional in action- **NxM = 5*5**
Nodes Explored = 123

Above in figure 13 we can see BDS in action, the primary issues had resolving this search was finding the path from a successful joining in the centre.

Features/Bugs/Missing:

Primary Bugs:

- MBA* does not return a path i.e. LEFT; RIGHT etc. due to difficulty trimming the explored path for the best search tree. Still appears to find goal correctly each time though.
- Bidirectional does not return correct path traversed and is now in maintenance.

Research:

Research tool A:

Each of the previously discussed visualizations were constructed through a visualizer built for the pathfinding application - see program instructions on how to activate. The initial terminal print out used in early algorithm implementations was then changed to store the string representations of each change in state, so when the algorithm has finished, the GUI can render each of those individual states (See the appendix for the RenderFunction). The SFML graphics library for c++ was used.

This can be selected through a command line argument of GUI (see readme.txt). In addition it is used in the testing mode to show 1 in every 20 of the randomly generated test maps for visual inspection.

Research tool B:

To allow mass simulation of the algorithms a ‘RandomMap’ class was made. This works by taking a max width and height and randomly populating it with a small amount of ‘WALL_TYPE’. Because knowing if a map will have a solution after providing walls is an NP-complete problem, this was both programmed to only provide a small amount of walls given a map size and in addition was run through the A* function to proof it prior to being tested by all the algorithms.

The secondary use of these simulations was that 100 maps were created and run through each of the algorithms, with the Search details for each map written to a csv file which could then be used to graph the outcomes of the visualization below.

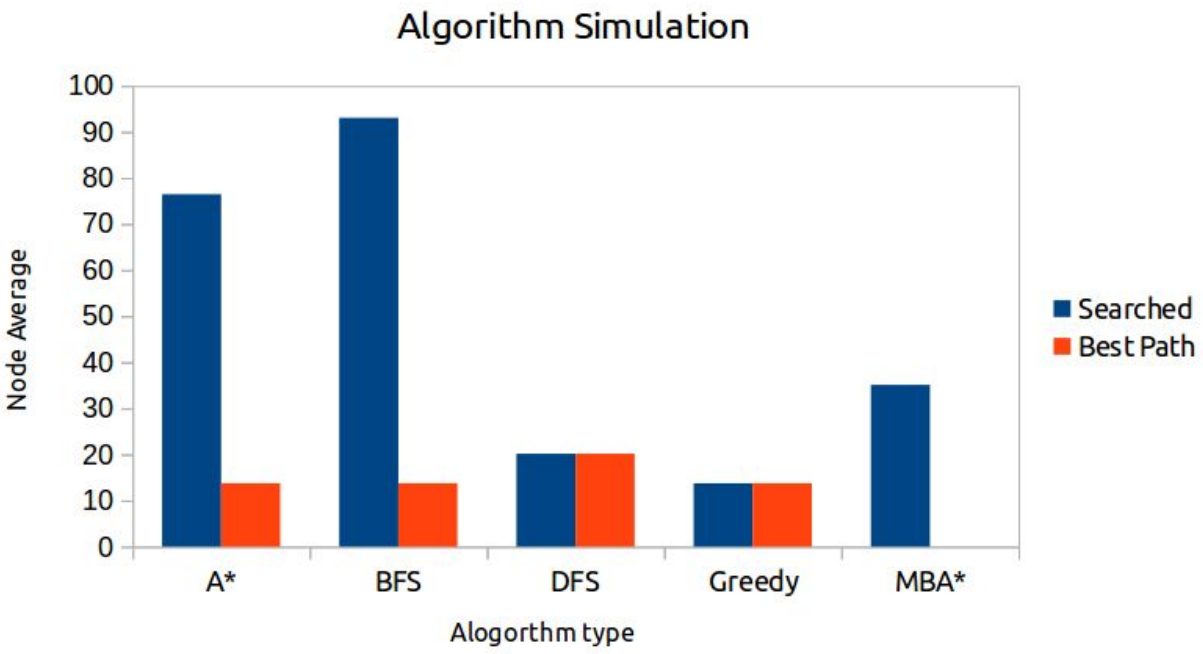


Figure 14: Algorithm average performance over 100 x randomized map generation.

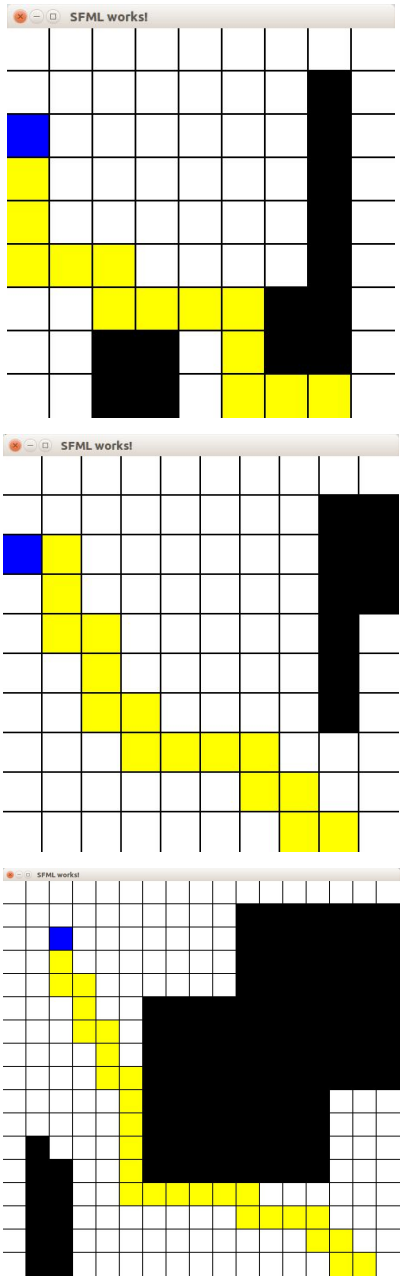
As we can observe in figure 14, on average each algorithm is performing as expected and outlined in the earlier discussion. A* and BFS have identical BestPath Node averages because they are expected to find the best path every time. In addition, due to A*'s better heuristic it was able to on average search less nodes. As you can see in Figure 15, the maps were quite simple and always had the start and goal nodes at relatively constant locations, this probably prevented A* from having a far lower 'Searched' average than BFS.

The maps did not have any 'end leaf nodes' i.e a search tree would never end because an algorithm could continue along 1 path until it had searched all of them. Because of this, DFS in these small sized maps did quite well in total nodes search, which as expected is the same as the 'Best Path' because of this '1-search' for all nodes approach. If this was a Tree-state map, then DFS may eventually reach a leaf node where it finds no goal and have to take a new branch to search and thus have a 'total searched' greater than its 'best path'.

It also appears that Greedy was gifted with easier maps, unlike the more complex one used in the earlier discussion, thus greedy appeared most effective in these maps due to easy paths allowing little regard for 'dead-ends'.

Finally, MBA* did not have 'best path' data due to its remaining bugs. It is also important to note that of all the algorithms in this simulation it failed 11 times. During this simulation we operated at a MAX_PATHS range of 25. Which for A* on big maps could have causes infinite loops and thus a timeout failure. This displays exactly the drawbacks and benefits of MBA*. We can see that it did in fact find the goal with a good average nodes expanded however these failures

outline the risk of not storing previous states. Additionally, these failings may just be attributed to a bug, as we can consider the fact that GBFS did not fail at all.



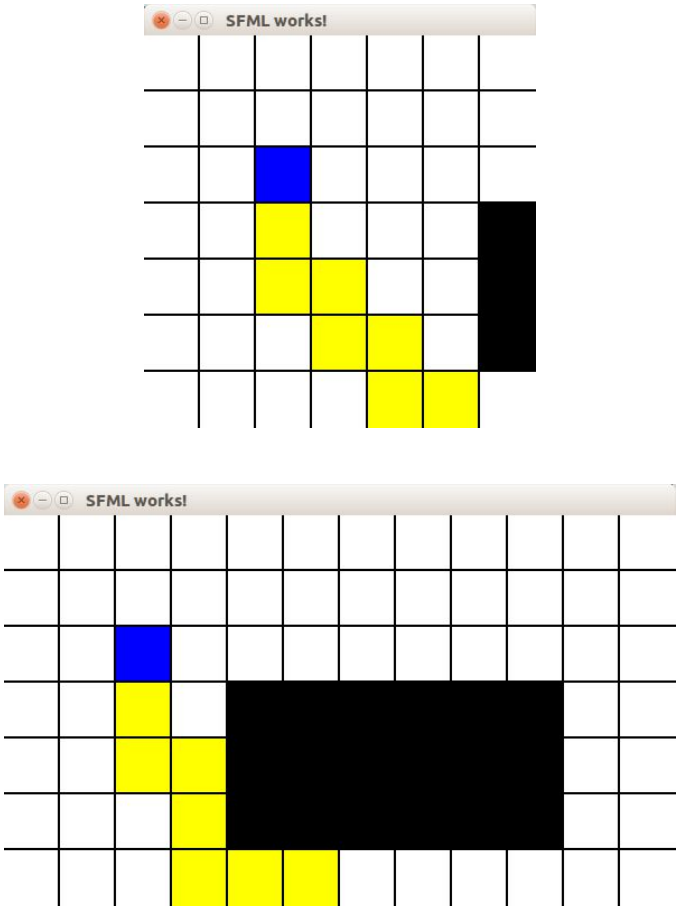


Figure 15: Randomized map generations (1 from each alternate width / height constraints).

Conclusion:

In conclusion we can say that, ‘simple’ problems such as those shown above, which don’t have dead-ends, are on average most efficiently completed by GBFS. Given that A* is the most reliable for more difficult maps; we could consider implementing one of its predecessors such as IDA* to minimize its primary weakness which is exploration cost. Essentially we are taking the best aspects of MBA* and GBFS to create a reliable but efficient algorithm for this style of problem. Of Course if the path cost were to change and become non-uniform we may re-assess the best algorithm.

References:

Russell, S & Norvig, P 2010, Artificial Intelligence A Modern Approach, Pearson Education, Upper Saddle River, New Jersey.

Acknowledgements/Resources:
readme.txt

Glossary:

Admissible:

A heuristic function for an informed search algorithm is ‘admissible’ if it never overestimates the path to the goal, that is, it should always calculate the distance to the goal as equal to the real distance to the goal.
h(n) <= f(n)

Expand Children:

When given a single node we can ‘Expand its children’. This is done by taking all the ‘Actions’ available for that Node and checking what Node is returned form our ‘State List’ given that action. Thus we are returning a list of all the Nodes resulting from all a nodes possible actions.

Explored Path:

A stored list of states that the search algorithm has already looked at, enabling it to remember where it has already been.

Frontier:

A queue structure of data containing nodes in a search tree that a search algorithm has obtained from their parent node, but has yet looked at.

State List:

A list of ‘states’ which have been constructed by the programmer/problem solver for the given problem. For example, in a tree search diagram each node/state will have a set of actions available, by choosing one of those action we can look up the state list to see what node will be returned from that list.

Appendix

```
std::vector<Path*> SearchType::ExpandNode(Node* currentNode, Problem& problem) {
    int actionCount = 4;
    //Need to track path (nodes and actions)
    std::vector<Path*> successors;

    //first in action count is LEFT
    for( int i = 0; i < actionCount; i++){
        //Try action and return node from it
        Node* n = GetActionChild(currentNode, ActionType(i));

        //If returned null then we did not have a node for this action
        // OR if its in out current path we do not want to follow it - to avoid loops
        if(n == NULL || IsInCurrentPath(n)) {
            continue;
        }

        n->Depth = currentNode->Depth + problem.PathCost;
        n->PathCost = problem.PathCost;
        n->Parent = currentNode;

        //Add to vector of Path types
        successors.push_back(new Path(n, ActionType(i)));
    }

    return successors;
}
```

1.) **ExpandNode - From SearchType base class.**

```
Node* SearchType::GetActionChild(Node *node, ActionType action) {

    ActionType a = action;
    //Add write to JSON file implementation?

    std::tuple<int,int> instruction;
    //Get lookup instructions
    switch(a){
        case ActionType::LEFT:
            instruction = std::make_tuple(0,-1);
            break;
        case ActionType::RIGHT:
            instruction = std::make_tuple(0,1);
            break;
        case ActionType::UP:
            instruction = std::make_tuple(-1,0);
            break;
        case ActionType::DOWN:
            instruction = std::make_tuple(1,0);
            break;
        default:
            std::cerr << "Error with action look up " << a << std::endl;
            instruction = std::make_tuple(0,0);
            break;
    }

    //Get Node* for action stated - dereference for assigning
    return StateLookUp(node, instruction);
}
```

2.) GetActionChild - From SearchType base class.

```
///Using Pointer Address comparisons to look up list of Node*
Node* SearchType::StateLookUp(Node* node, std::tuple<int,int> instruction) {
    for(int i = 0; i < stateList.size(); i++){
        for(int j = 0; j < stateList[i].size(); j++){
            if(node == stateList[i][j]){

                //Check if this index exists in our 2d array
                if(i+std::get<0>(instruction) < 0 || i+std::get<0>(instruction) > stateList.size()-1 ||
                   j+std::get<1>(instruction) < 0 || j+std::get<1>(instruction) > stateList[i].size()-1) {
                    Node* n = NULL;
                    return n;
                }
                else if(stateList[i+std::get<0>(instruction)][j+std::get<1>(instruction)] == NULL){
                    Node* n = NULL;
                    return n;
                }
                else if(stateList[i+std::get<0>(instruction)][j+std::get<1>(instruction)] == node->Parent){
                    Node* n = NULL;
                    return n;
                }
                else{
                    return stateList[i+std::get<0>(instruction)]
                    [j+std::get<1>(instruction)];
                }
            }
        }
    }
}
```

3.) StateLoopUp - From SearchType base class.

```
SolutionResponse Astar::Search(Problem& problem, Node* nodeSearch) {

    //To be returned solution
    SolutionResponse* solution;
    RenderCurrentMap(nodeSearch, problem);

    //First node so is the lowest
    Node* searchNode = nodeSearch;
    //Current distance from node
    currentLowestCostFunction = problem.InitialState->goalCost + problem.PathCost;

    do{
        vector<Path*> children = ExpandNode(searchNode, problem);

        //Add nodes from paths to frontier
        for (auto& p : children) {
            PushFrontier(p);
            PushPath(p);
            RenderCurrentMap(p->pathNode, problem);
        }

        //Gets best heuristic value from frontier
        Path* bestPath = HeuristicFunction(problem);

        if (GoalTest(problem, bestPath->pathNode)) {
            //Remove this to see the full Path search of A*

            trimmedPath = TrimPath(problem, exploredPath);
            solution = new SolutionResponse(trimmedPath, "success");

            //trimmedPath = TrimPath();
            return *solution;
        }

        searchNode = bestPath->pathNode;

    } while(!FrontierIsEmpty());

    //Frontier was empty so we did not find the goal
    std::cerr << "Could not find solution - empty child";
    solution = new SolutionResponse("failure");
    return *solution;

}
```

4.) A* Search function

```
void SearchType::RenderCurrentMap(Node* currentPosition, Problem& problem) {

    std::stringstream ss;
    std::string vis = "";

    std::vector<std::string> tempVec = std::vector<std::string>();
    for (int j = 0; j < stateList.size(); j++) {
        for (int i = 0; i < stateList[j].size(); i++) {

            if (stateList[j][i] == problem.InitialState) {
                ss << "S" << " | ";
                vis = "S";
            }
            else if (stateList[j][i] == currentPosition) {
                ss << "H" << " | ";
                vis = "H";
            } else if (stateList[j][i] == problem.GoalState) {
                ss << "X" << " | ";
                vis = "X";
            } else if (stateList[j][i] == NULL) {
                ss << "W" << " | ";
                vis = "W";
            } else {
                ss << "1" << " | ";
                vis = "1";
            }
            tempVec.push_back(vis);
        }
    }

    //combine string groups to have all paths search
    if (stringPathVec.size() > 0) {
        for (int i = 0; i < tempVec.size()-1; i++) {
            if (stringPathVec[stringPathVec.size() - 1][i] == "H") {
                tempVec[i] = "H";
            }

        }
    }
    stringPathVec.push_back(tempVec);

   // std::cout << std::endl << std::endl;
```

5.) Render Function - From SearchType base class.

```
switch (WALL_TYPE(wallType)) {
    case
        WALL_TYPE::LONG_SMALL:
        //If values are under 0.5 we must make them 1
        tempHeight = (int)(locationLimitY * 0.6 > 1 ? locationLimitY * 0.6 : 1);
        tempWidth = (int)(locationLimitX * 0.05 > 1 ? locationLimitX * 0.05 : 1);
        //Start at random position ('2' is that we don't want to block the starting position)
        tempX = rand() % (locationLimitX);
        tempX = tempX < 1 ? 1 : tempX;
        tempY = rand() % (locationLimitY);
        tempY = tempY < 1 ? 1 : tempY;
        break;
    case
        WALL_TYPE::SHORT_SMALL:
        //If values are under 0.5 we must make them 1
        tempHeight = (int)(locationLimitY * 0.3 > 1 ? locationLimitY * 0.3 : 1);
        tempWidth = (int)(locationLimitX * 0.05 > 1 ? locationLimitX * 0.05 : 1);
        //Start at random position ('0.2' is that the width / height ('thickness') must be 'small' relative to maps
        tempX = rand() % (locationLimitX);
        tempX = tempX < 1 ? 1 : tempX;
        tempY = rand() % (locationLimitY);
        tempY = tempY < 1 ? 1 : tempY;
        break;
    case
        WALL_TYPE::THICK_LARGE:
        //If values are under 0.5 we must make them 1
        tempHeight = (int)(locationLimitY * 0.5 >= 1 ? locationLimitY * 0.5 : 1);
        tempWidth = (int)(locationLimitX * 0.5 >= 1 ? locationLimitX * 0.5 : 1);
        //Start at random position ('2' is that we don't want to block the starting position)
        tempX = rand() % (locationLimitX);
        tempX = tempX < 1 ? 1 : tempX;
        tempY = rand() % (locationLimitY);
        tempY = tempY < 1 ? 1 : tempY;
        break;
    case WALL_TYPE::THICK_SMALL:
        //If values are under 0.5 we must make them 1
        tempHeight = (int)(locationLimitY * 0.25 >= 1 ? locationLimitY * 0.25 : 1);
        tempWidth = (int)(locationLimitX * 0.25 >= 1 ? locationLimitX * 0.25 : 1);
        //Start at random position ('2' is that we don't want to block the starting position)
        tempX = rand() % (locationLimitX);
        tempX = tempX < 1 ? 1 : tempX;
        tempY = rand() % (locationLimitY);
        tempY = tempY < 1 ? 1 : tempY;
        break;
```

6.) Wall randomization code - from RandomPaths.cpp