1 INTRODUCTION

This covers solving the search problem of coastguard rescue simulation with different search algorithms and approaches.

Coastguard rescue simulation simulates the robot which has to navigate through the Giant's Causeway and find the shortest path to take people in danger to the safe position. The map consists of hexagonal elements (figure 1) and contains two types of obstacles. The robot can not move through the obstacle type one in any direction and can move only horizontally through the obstacle type two (figure 2).

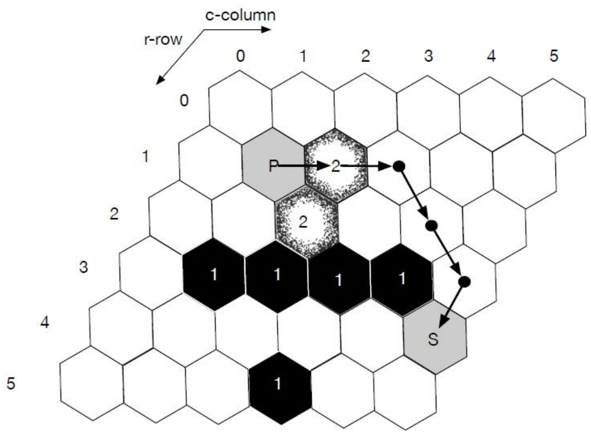


Figure 1: Example map

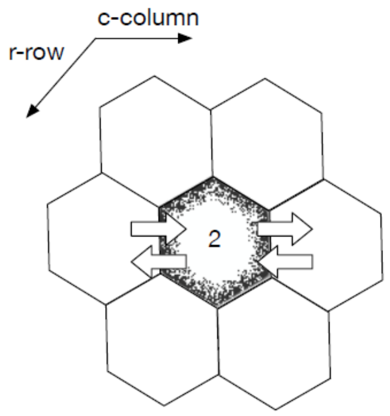


Figure 2: Obstacles of type 2

There are provided different maps with various configurations. The configuration determines the location of the start position and goal position. The aim of the search algorithm is to find the fastest way from the start position to the goal position.

In this simulation, the environment is: deterministic, fully observable, static, known.

PEAS model

Agent Type	**Performance Measure**	**Environment**	**Actuators**	**Sensors**
Autonomous coastguard rescue robot	Find the shortest path (path cost), efficiency (number of visited states)	Giant's Causeway	Rubber tracks	GPS, coast map

- **States**: A state specifies the location of the robot on the map with coordinates (figure 1).
- **Initial state**: Any state can be designated as the initial state. It is the starting position of the robot.
- **Goal state**: The state that the robot tries to achieve.
- **Actions**: Movement from one state to another in 6 directions in figure 3.
- **Transition model**: Given a state and action, this returns the resulting state. The robot can only move along the grid, from free cell to free cell using the coordinate system avoiding obstacles 1 and 2 accordingly.
- **Goal test**: This checks whether the current state matches the goal state.
- **Path cost**: Each step costs 1, so the path cost is the number of steps in the path.

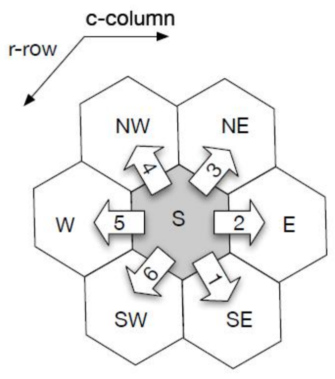


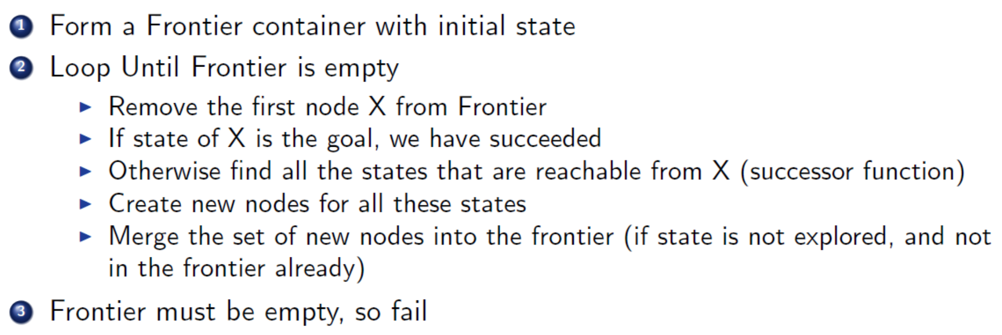
Figure 3: Navigation system

2 DESIGN AND IMPLEMENTATION

The attached file consists of 9 classes. Main is used to run the required algorithms. Map and Conf are used to specify the coast map and its configuration, where Coord defines coordinates of each state. The rest is explained in more details below.

General search

This is an uninformed type of search and is briefly described in figure 4.



1. Form a Frontier container with initial state
2. Loop Until Frontier is empty
   - Remove the first node X from Frontier
   - If state of X is the goal, we have succeeded
   - Otherwise find all the states that are reachable from X (successor function)
   - Create new nodes for all these states
   - Merge the set of new nodes into the frontier (if state is not explored, and not in the frontier already)
3. Frontier must be empty, so fail

Figure 4: A simplified description of General search

In the file, it is called GeneralSearch class. The frontier is represented by ArrayList because it keeps the order and can simply add and remove elements.

The node represents each state that is expanded to the frontier. A node is an object (Node class) that contains information such as state (coordinates), parent node, path cost, f-cost (in informed search).

Every time it checks whether the frontier is empty, it prints frontier and decides how it removes a node from the frontier according to the required algorithm. To keep track of explored nodes, it adds the removed node to the ArrayList of explored nodes. The implementation of BFS and DFS is different according to the way it is manipulated with the frontier.

BFS is first-in-first-out, which means it always removes the node from the beginning (index: 0) of the frontier.

DFS is last-in-first-out, which is implemented by removing the last node from the frontier (index: length of the frontier minus one).

Removed node is then checked whether its state is equal to the goal state. If it is true, it then takes the node and look at its parent node and parent node of that parent node and so on, until it constructs the whole path from the initial state to the goal. It also prints the path cost and the number of explored nodes (nodes removed from frontier).

If it does not yet find the goal, it expands the removed node (current node to be explored), which means that it looks around that node what other states are available. This is done through expand() method that GneralSearch class inherited from the abstract class ExpandNode. This class is used to look around the current state for available states where the robot could move next.

The expand method first calls other method expandedStates() from which it gains available states around the state of the explored node. It expands the states in a certain order (shown in figure 3), which also serves as a tie-breaking strategy (the order of nodes in the frontier). This is done by doing steps in a specific direction, for example, to go south-east it must do one step in the row and one in the column, directionSteps is an integer array that contains these steps. The algorithm then loops through and do the steps; then it checks what kind of obstacle a new state represents. If it is obstacle 1, it does not add it into expanded states. For obstacle 2, it allows only if the explored node's state is in the same row. It also considers whether the state of the explored node has obstacle 2, which would again mean it can only move to the states in the same row. All of this is surrounded by try-catch block which rules out the states that are out of the map. The array list of expanded states is then returned to the expand method, where the states that are already in the frontier or in explored nodes are omitted.

The expand() method then returns ArrayList of nodes, that are added to the frontier one by one, and the whole cycle is repeated. For DFS to keep the tie-breaking strategy because the algorithm takes nodes from the end of the frontier, we need to add new nodes to the frontier in order to preserve the tie-breaking strategy. This means we add to the frontier first node with the state in SW direction then W and so on so that if we remove the last node from the frontier, we have SE direction first to explore. This is done by reversing nodes obtained from the expand().

Informed Search

The informed search differs from uninformed because it contains information about how far is the goal. This information is represented by the f-cost function, which is represented by fCost property of each node. Node class also have the function fCost() that calculates the value for each node according to the search type. Best first search (BestF) f-cost is equal to manhattan distance from the current state to the goal, whereas for A* it equals to manhattan distance plus path cost (cost from an initial state to the current state).

- Point A (r_A, c_A)
- Point B (r_B, c_B)

1. Reverse the sign of rows:
   - $nr_A = -r_A$
   - $nr_B = -r_B$
2. Difference Col
   - $\Delta_c = c_A - c_B$
3. Difference Row
   - $\Delta_r = nr_A - nr_B$
4. If Δ_c and Δ_r have the same sign
   - H-Distance$= |\Delta_r| + |\Delta_c|$
5. else
   - H-Distance$= max(|\Delta_r|, |\Delta_c|)$

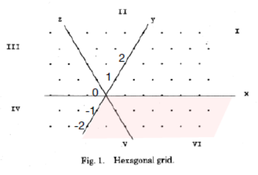


Figure 5: Manhattan distance calculation

Nodes are not sorted according to the tie-breaking strategy; therefore, in informed search, PriorityQueue for the frontier is used and programmed to sort nodes in the queue according to their fCost. Thus, the algorithm always polls the node with the lowest fCost. Manhattan distance is calculated according to figure 5, and path cost is taken from each node. Additionally, Euclidian heuristic is added that is calculated by parallelograms in figure 6 and 7.

Informed search, expand() method also considers whether the node's state that was already explored or is in frontier has higher fCost than the new state. That means if the new state has lower fCost, it is added to the frontier. The other parts of the algorithm still work on the same principles.

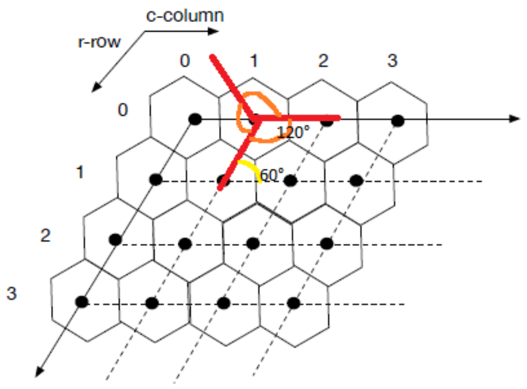
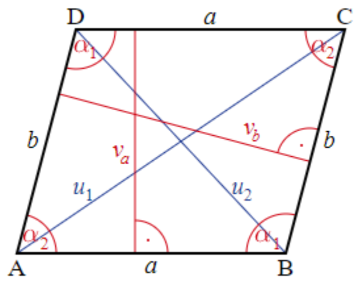


Figure 6: Deduction of angles



$$S = av_a = bv_b \qquad o = 2 \cdot (a + b)$$

$$v_a = b \cdot \sin\alpha_1 \qquad v_b = a \cdot \sin\alpha_1$$

$$\sin\alpha_1 = \sin\alpha_2 \qquad \alpha_1 + \alpha_2 = 180^\circ$$

$$u_{1,2} = \sqrt{a^2 + b^2 - 2 \cdot ab \cdot \cos\alpha_{1,2}}$$

Figure 7: Formulas for parallelogram

Furthermore, bidirectional search (TwoAgentSearch class) was implemented. It uses two agents where one starts in the initial state and the other one in the goal state. They both use BFS with their own frontiers. This algorithm finds the path when the agents have the node with the same state in their frontier or in the explored list, which is solved by nodeIntersect() function.

3 EVALUATION

BFS and DFS are uninformed searches; thus, they prefer nodes based on tie-breaking strategies, which is blindly looking for the solution. Breadth-first takes out nodes from frontier FIFO, whereas depth-first in a LIFO manner. BFS usually ends up with the significant amount of explored nodes, but it guarantees to find a best possible solution because it goes one depth in search tree at the time and path cost for one step always equals one.

If the tie-breaking strategy is lucky, DFS finds the solution quickly with less explored nodes. Since algorithm ruled out already explored nodes and those in a frontier, it is not getting stuck in any loop, thus always finds the solution if there is one. Although, this solution is not always optimal and compared to BFS can have higher path cost but tends to have less explored nodes.

Best first and A* are informed searches that prefer nodes according to the estimated cost (f-cost) to the goal. BestF always explores first the node with the lowest distance to the goal. It does not guarantee the best solution, but the algorithm is highly effective in terms of explored nodes. On the other hand, Astar always guarantees to find an optimal solution. As a trade-off, it explores more nodes then BestF.

Euclid is basically BestF, but instead of manhattan, euclidian distance is used. Euclid shows the actual distance to the goal, but since in this case, path cost is one in all directions BestF usually outperforms Euclid.

In the vast majority of cases, the bidirectional search has beaten BFS, because it uses two agents' power. Unfortunately, it does not always produce the best solution, because their nodes can sometimes intersect in non-optimal solution. Nevertheless, it usually performs with a low rate of explored nodes and easily beats DFS. It has lower path cost solutions than DFS with similar effectiveness.

Conclusion

Bidirectional search is suitable where the solution is needed to find fast, and it does not matter on the path cost that much. The same applies to BestF and Euclid, which are even more efficient.

BFS has a straightforward implementation and always finds the best solution, but slower, with many explored nodes. For this problem where it is needed to find the most optimal solution as fast as possible, AStar is the best solution.