**Introduction**

In this project our aim is to make a General search function that takes two parameters the first one is the problem we want to solve the second parameter is the strategy we are going to follow to solve the problem. Our search function will follow this strategy on this problem till it reaches the solution or sometimes it gets stuck and stop if there is no solution for the problem. In our case the problem which we are aiming to solve (Save Westeros) is helping john snow to find the best way to kill white walkers with least possible number of dragon glasses so, we are going to have a 2d grid consists of cells. This 2d grid is a representation of the environment that john snow will be in. In this environment there is two things beside John snow and the white walkers first Dragon stone where john can get dragon glasses from and second, some obstacles that john snow should avoid so, each cell in our grid may be empty or contains an obstacle or a white walker or john snow and only one cell contains The Dragon stone. At the Beginning john snow will be standing in the bottom right cell in the grid and his first goal is to go to the cell that contains Dragon stone so he can get maximum number of dragon glasses he can take to kill white walkers. There are some rules that john should follow first he can not go to any cell contains an obstacle or white walker so in order to kill a white walker he should be in a cell adjacent to the white walker. One dragon glass can be used to kill any white walker adjacent to the cell that john is standing in. If john ran out of dragon glasses he should go back to The Dragon stones factory.

**Description of our implementation**

Our code is divided into two parts the first part is related to the general search for any problem and the second part is for the problem we are aiming to solve which is Save Westeros. Let’s describe the first part which we call Search

**The search part** consists of three classes and one Interface and one Enum

The Interface which is called **State: -** Consists of only one method called isSame we use it to make sure that we are not going to expand node with same state again we are going to discuss it more later in our problem

The Enum which is called **Strategy: -** Consists of the names of strategies we are going to follow to solve any problem which are BF (Breadth-first search), DF (Depth-first-search), ID (iterative deepening search), UC (uniform cost search), GR1 (Greedy search with 1st heuristic function), GR2 (Greedy search with 2nd heuristic function), AS1 (A\* search with 1st heuristic function) and AS2 (A\* search with 2nd heuristic function) we will discuss how we implement each later.

1st class which is called **Node: -** it is the search-tree node each node has state , parent (it is generated from in the base case the parent is null (root)), operator (you used to reach current node)and depth (represents it`s level and it is equal to zero at root node).

There are getter methods for all of the attributes and method that is used to check if this node is root or not. Root node is the first node, it has no parent and no operators and it`s depth is equal to zero we have a method called isRoot(node) which checks if this node is root or not.

2nd class which is called **Operator: -** It is a class for an operator you can use in a problem each operator has a name and cost if there is no cost for an operator we automatically assign it to zero in the constructor else we have another constructor for given name and cost we have getter method for both attributes and set method only for the cost.

3rd class which is called**Problem: -** it is an abstract class for any problem consists of 3 attributes first one is a list of all operators in a problem, second is the initial state of a problem and finally the depth limit which is the biggest depth we can reach in our search we make it equal to the biggest possible value for integer and we assign it in the constructor. We made getter methods to all the attributes and setter method only for the list of operators. We have an abstract method called isGoal which is made to check if the current node is goal node or not in a problem. We will discuss it in our problem Save Westeros. We have also another abstract method called expand that returns list of all of the successor nodes by looping over a list of all available operators. We have the general search function that returns the goal node we reached at the end of solving the problem. In this method we take the problem and strategy as parameters, we initialize a linked list called nodes where we put the nodes that are going to be visited and we initialize list for repeated states then we add the root to nodes and we loop till nodes became empty we remove the first node in nodes list and check if its goal or not. If it is a goal, we break and return this node else we will get all the successors of this node and loop on them, we will make sure first that this successor node state is not visited before using isRepeated method that takes the successor node and the repeatedState list and apply isSame method on them to make sure that this successor node is not expanded before. If it returns false it will add it to nodes linked list else it will skip it. where this successor node will be in the linked list that depends on the strategy.

**BF:** We add the successor node at the end of the nodes linked list.

**DF:** We add the successor node at the beginning of nodes linked list.

**UC:** We add the successor node to the nodes list at the right position so that nodes list will be sorted according to the cost value

**ID**: We add the successor node at the beginning of the nodes list. There is a method in problem class for ID strategy (IterativeDeepeningSearch) that takes only the problem and start with limit equal zero then we loop and call general search method and expand nodes with depth level equal to the limit and it returns the list of successors empty or till we reach the goal we keep increasing the limit each iteration.

**GR1/GR2:** We will apply the heuristic function using abstract method called evaluateHeuristicOne, which will apply heuristic function on the node and on the sucessorNode if the output of applying it on node is less than the output of applying it on successorNode we will add the sucessorNode at the end of the nodes list otherwise we add it at the beginning of nodes list. Same if it is GR2 but using evaluateHeuristicTwo function instead of evaluateHeuristicOne function.

**AS1/AS2:** We will apply both evaluateHeuristicOne function and calculatePathCost function and add the results to each other, same but on successor node if the output of applying that on node is less than the output of applying that on successorNode we will add the sucessorNode at the end of the nodes list else we add it at the beginning of nodes list. We will do the same if it is AS2 strategy but we use evaluateHeuristicTwo function instead of evaluateHeuristicOne function. There is also two abstract functions. We have two abstract methods evaluateHeuristicOne and evaluateHeuristicTwo both used to apply the heuristic on node and return the admissible cost of this node will discuss the function in our problem later

**The second part** which is the problem we are aiming to solve Save Westeros in our code consists of 4 classes: Cell, Grid, SaveWeteros and SaveWesterosState

The 1st class is **Cell: -** consists of two attributes row and column we have one constructor that takes number of rows and columns and assign them to the attributes we have getter method for each attribute

The 2nd class is **Grid: -** consists of 11 attributes the two integers m and n for the dimension of the grid, 2d array list of cells, a dragon stone cell, an agent cell, list of obstacles cells, list of white walkers cells and 4 static integers the first two represent maximum and minimum number of rows and columns and the other two represent the minimum number of obstacles and white walkers. We have two constructors one for completely random size n number of rows and m number of columns (minimum 4 rows and 4 columns maximum 60 rows and maximum 60 columns) and another constructor for generating square grid (minimum 4 rows and 4 columns maximum 60 rows and maximum 60 columns). We have getter methods for all attributes except for the static ones. The grid is randomly generated so the places of obstacles, white walkers and dragon stone are randomly located except for john snow who stands in the initial state in the bottom right cell. We have two methods initCells that initialize all the cells of the grid and generateMap that makes random number of white walkers (minimum number of them will be 3 maximum will be ((n/4) + 3)) and locate them randomly, makes random number of obstacles (minimum number of them will be 3 maximum will be ((n/4) +3) and locate the dragon stone in random cell. Also and we save all white walker positions in white walkers list and all obstacle positions in obstacles list and dragon stone position in dragon stone attribute. There is a method called gridInfo that print out the information of the grid number of white walkers and obstacles and the grid size

The 3rd class **SaveWesterosState: -** This class implements state, it has three attributes list of the positions of white walkers, the agent position and the number of dragon glasses the agent have. We have getter methods for each attribute and isSame method which takes state as parameter and checks if this state is same as the current state.

The 4th class is **SaveWesteros: -** This class has the main method where we test our implementation on Save Westeros problem. It extends problem class. It has 8 attributes first one is the grid, The next two list of white walkers and list of obstacles, The dragon stone cell position, The agent capacity, and finally static costs for the operators in our problem: move cost(The cost is 6 and the possible movements are up, down ,right and left), pickup cost(The cost is 5) and kill cost(The cost is 4). We have getter methods for all of the attributes except for the static variables. We have a method called cloneCells we copy the list so each state will have a copy of itself not it`s reference. We have method called prepareSearch that set the attributes of save Westeros after generating the grid, we set the agent capacity of dragon glasses to random value (minimum number of whiter walkers maximum number of white walkers + 2) and finally we set the initial state of our problem.

We have a method called getSequenceOfMoves that returns a string of the sequence of operators we used to reach the node goal by getting the expanded nodes operators. We have visualizeGrid method That prints the grid to make you see the sequence of steps and changes after each step in the grid. We have canVisitCell that checks if we can move to this cell or not as john snow cannot go to a cell contains white walker or obstacle.

We have methods for generating grid to have the options if you wanted to be completely random in size and if you wanted to make it square with n size or you want to make it with m (number of rows) and n (number of columns).

We have isGoal method, remember it was an abstract class in problem our Goal here is to check if there are still white walkers (return false) or there are no more white walkers (returns true). We have calculatePathCost that takes a node and calculate what was the cost of the operator which we used to reach this node plus the path cost of the parent and it returns the total cost.

We have search method takes the grid, strategy and visualize. First, we set the grid and call prepareSearch method and make array list for the result of the sequence of moves to reach the goal and the cost to reach it and the nodes we went to. If the strategy is ID we call iterativeDeepeningSearch giving it our problem and will return the goal node else we are going to call generalSearch method giving it the problem and the strategy and will return also the goal node. Then we check if this goal node is not equal null and get the nodes we visited to reach the goal node and get also the sequence of moves we took to reach the goal node and finally we get the cost to reach this goal node and we get the sequence of moves we made to reach the goal node. Then we check if visualize is true we call visualizeGrid method and finally we return the result.

We have killWhiteWalkers method that takes the list of white walkers cells and the current cell of the agent we first clone the cells with white walkers and then we check if there is any adjacent cell to the agent cell with white walker we put this cell in a list called predicate cell and we check if there is cells in the predicate list. Then we remove it from the clonedWhiteWlker list and we decrease the number of dragon glasses with by one. At the end we return SaveWesterosState with the agent cell, the new number of dragon glasses and clonedWhiteWalkers list.

We also have the expand method which was an abstract method in problem class. In this method our aim is to get the list of nodes generated from the current node by iterating on a list of operators. We make an empty list of nodes called expansion which the method returns at the end. First we need to make sure that the depth of the agent cell node does not exceed the depth limit, then we set the agent cell to stateAgentCell variable and we set cells variable to be equal to the grid and we get also the row and column of the agent cell and we loop ­over all the operators we can make from the current agent cell (up, down , right, left, kill or pickup ) and add it to the expansion list. Before moving we check if we can visit this cell first then if yes we create new node for this cell and we set its path cost and add this new node in the expansion list. if the operator is pickup then the new agent cell will be at the same position. We check if the agent cell is the dragon stone cell and we create a new node of this cell and set everything in this node and then we set the path cost of this node and finally we add it to expansion list. if the operator is kill then the new agent cell will be the same. We call killWhiteWalkers if the number of dragon glasses greater than zero and there is still white walkers. We make new node and initialize it with new operator. And its cost will be equal cost of kill – number of white walkers we killed and then we set the new path cost by calling calculatePathCost method to this new node and then we add it to expansion list. After trying all possible operators, we return the expansion list.

There is also evaluateHeuristicOne that takes a node and apply the heuristic function on it. Our first heuristic function is basically getting the remainder number of white walkers and divide them by 4. Then multiply the result by the number of dragon glasses. If the number of dragon glasses greater than zero we will return the result else we will add the cost of going to dragon stone cell plus picking up dragon glasses. Now the question is why this is admissible because we are taking in to consideration the best-case scenario that john will always be between 4 white walkers.

The last method we are going to discuss in Save Westeros is the **Main method**

1st We initialize saveWesteros variable

2nd We initialize grid for it and generate it.

3rd We make list of result

4th We test on BF strategy by making the result to be equal the return of search method in Save Westeros class giving it the grid and the strategy to be BF If the result after applying the strategy is Empty we print No solution else we print result which contains sequence of moves from initial state till goal node and number of nodes we expanded.

5th We test on DF strategy by making the result to be equal the return of search method in Save Westeros class giving it the grid and the strategy to be DF If the result after applying the strategy is Empty we print No solution else we print the resut

6th We test it on UC strategy by making the result to be equal the return of search method in Save Westeros class giving it the grid and the strategy to be UC If the result after applying the strategy is Empty we print No solution else we print the result

7th we test it on ID strategy by making the result to be equal the return of search method in Save Westeros class giving it the grid and the strategy to be ID If the result after applying the strategy is Empty we print No solution else we print the result

8th We test it on GR1 strategy by making the result to be equal the return of search method in Save Westeros class giving it the grid and the strategy to be GR1 If the result after applying the method is Empty we print No solution else we print the result

9th We test it on AS1 strategy by making the result to be equal the return of search method in Save Westeros class giving it the grid and the strategy to be AS1 If the result after applying the method is Empty we print No solution else we print the result