1. Problem Description

We have a 5x5 grid world. The Main character (Snow) starts at the right bottom cell in this grid (grid[4,4]). Around this world, There are generated enemies (white walkers) in our example we took four in four random cells that Snow needs to kill. To kill these white walkers, Snow needs to find the source of the weapons (dragon stones) necessary to kill them which is also randomly generated in a cell in the world.

If Snow chooses to use a dragon stone in a certain cell, He will kill the white walkers in the surrounding cells (in the four directions). Additionally, There are two obstacles generated in random cells in the world to prevent Snow from going to that cell. Snow wins if there are no white walkers left in the world.

---

2. Implementation

2.1 Node

The node object have five attributes :

1) Object state — /* object that have all data about the current state of the game (X&Y position of character in the grid, number of dragon stones that the character have, and the number of white walkers still alive). */
2) Node parent — // The parent node of the current node.
3) Object operator — // The action that led to the current node (left, up, etc..).
4) int depth — // The current depth in the tree.
5) int path_cost — // The cost of the path from the parent node to the current.

Additionally, there are two constructors for the node. The first is for initialization, and the second is for creating node with given parameters.

2.2 Abstract Tree Search

This class is responsible for solving the problem through the solve(problem problem) method. This method expands the nodes and checks for the goal state. In this funtion we use the ChooseLeafNode() that is initialized in the subclass that extends this class for each search technique.

There are two abstract methods that we override in these subclasses:

1) Collection<Node> nodeCollection(); /* responsible for assigning the data structure that is suitable for each search technique. */

2) Node chooseLeafNode(Collection<Node> nodeCollection, Problem problem);
/* responsible for choosing which node do we expand according to the used search technique. */

The class has two attributes:

1) int expansionNumber // nubmer of expanded nodes in the solving process.
2) Collection<Node> expansionSequnce
/* The variable that we add the expnded nodes to (the data structure varies from one search technique to the other as mentioned above). */

2.3 Save Westeros Problem

It extends the Problem class and overrides it's methods:

1) Object getInitialState(); // Gets the initial state of the problem;
2) boolean isGoal(Object state); // Checks whether a given state is a goal state. Using the number of WW in the node state;
3) Collection<Object> getActions(Object state); // Returns all possible actions from a given state.
4) Object getNextState(Object state,Object action);
/* Return the state that is resulted from a given state if the given action in Made. */
5) int getStepCost(Object start, Object action, Object dest);
/* return the total path cost if the given given action is made on the given state resulting in the given destination state */
6) Object[] Search(Cell[][] grid, String strategy, boolean visualize);
/* Function asked by the instructor to be used in testing different grids and strategies and has a boolean called visualized if true prints a map after each action taken to reach goal and returns an array that calculates number of expanded nodes, actions taken and path cost */
7) String printSolution(Node solution);
/* Function used to return a String contains data needed to be used in Search function */

The class has two attributes:

1) state initialState /* Initializes the initial state of the problem to locate Snow in

the right-most bottom cell having NO dragon stones and facing FOUR white walkers in our grid. */

2) world main // Initializes the world to be a 5x5 grid in our example.

Kindly note that we created a class called World and MapGen we initialize our world, generate the grid, obstacles, WW and dragoneStore. Also in this class we have functions to check cells in the map (Check Up, Check Down, Check Right, Check Left and Check current) we create an instance of this class in SaveWesteros and used is as our map. For more details on this check the Main Functions section.

2.4 Main Functions

Note that solve and chooseLeafNode functions are sometimes implemented in the search techniques' classes to satisfy the technique implementation.

Class	Method	Return	Description
Abstract Tree Search	Node solve(Problem problem)	Node	The main search function that keeps on expanding the nodes and checking whether it's goal or not.
	Collection <Node> expand(Node node, Problem problem)	Collection <Node>	Expanded a specific Node
	Node chooseLeafNode(Collection<Node> nodeCollection, Problem problem);	Node	This function takes as an input of collection and problem and based on what type of search it chooses the leaf to return to be expanded
	Collection<Node> nodeCollection();	Collection <Node>	Create an instance of the data type used in saving(preserving the nodes in)
Save Wasteros	Boolean isGoal(Object state)	boolean	Check whether this state is goal state or not.

	Object[] Search(Cell[][] grid, String strategy, boolean visualize);	Array [3]	Function asked by the instructor to be used in testing different grids and strategies and has a boolean called visualized if true prints a map after each action taken to reach goal and returns an array that calculates number of expanded nodes, actions taken and path cos
	Object getInitialState();	Object (State)	Returns the initial state of the problem.
	Collection <Object> getActions(Object state)	Collection <Object>	Returns a list of actions that could be done from this Node or State
	Object getNextState(Object state, Object action)	Object	Returns a State, using a current state and specific action on this state.
	Int getStepCost(Object start, Object action)	int	Given a current state (Node) and an action it returns the path cost.
MapGen	Void GenGrid()	void	Generate the grid containing WW, Dragonstore and obstacles in random positions.
	Cell[][] vHelper(Cell[][] mhelp, Actions a, int sX, int sY)	Cell [][]	Helper method for Visualize method which takes a map and an action and prints the map after performing the action
	void pMap(Cell[][] map, int x, int y)	Void	pMap prints the input map and x and y is the dimensions of the input map
	void Visualize(Cell[][] map, Node s)	void	Prints the maps as the agent perform actions to reach the goal, takes as input a map and Node and prints the sequnce of actions from root on

world	Void printMap()	void	Prints map in a readable way.
	Cell checkRight(int x, int y)	cell	Check what's in the cell on the right (obstacle, stones' source, or white walker)
	Cell checkLeft(int x, int y)	cell	Check what's in the cell on the left
	Cell checkUp(int x, int y)	cell	Check what's in the upper cell
	Cell checkDown(int x, int y)	cell	Check what's in the cell downwards
	Int checkSurroundingWW(int x, int y)	int	Gives the number of surrounding white walkers
State	CompareState(State si,State sj)	Boolean	Compares two states and return whether they are equal or not
	repeatCheck(State s, ArrayList<State> List)	Boolean	Take an list of states and a state and return true if the state is in the List , this is used to stop repeated states

2.5 Search Functions

2.5.1 Breadth-first search algorithm

We used linked lists to put the expanded nodes in because we need a FIFO design data structure to pop the first node entering the list. When a node is expanded it is popped and the expanded nodes are added to the end of the list. This way we ensure that we don't go to the next depth level until we expand all the nodes in the current level.

2.5.2 Depth-first search algorithm

We used stacks to put the expanded nodes in because we need a FILO design data structure to pop the last entered node.We expand the last entered node to the list. When a node is expanded it is popped from the stack of expanded nodes and we push the expanded nodes to the stack.

2.5.3 Iterative-deepening search algorithm

The same as Depth-first algorithm. The difference is that a cut-off value for the depth level is used iteratively starting with cut-off value 0. If the the cut-off depth is reached without finding the goal state, the cut-off is incremented and problem is resolved until we reach a goal state.

2.5.4 Uniform-cost search algorithm

We used arraylists to put the expanded nodes in. When we expand a node, we sort all the nodes in the arraylist ascendingly according to the path_cost attribute. This makes the node with the least overall path cost at the beginning of the arraylist. That's why we always expand the first node in the list. So, the cycle goes between expand and sort until a goal state is found. The path cost is calculated according to the actions. An action cost is assigned to each action and the path cost is the addition of all actions costs.

2.5.5 Greedy search algorithm

The same as Uniform-cost algorithm. The difference is that the path cost is calculated using Heuristic functions.

2.5.6 A* search algorithm

The same as Greedy search algorithm. The difference is that the path cost is calculated using Heuristic functions + path cost. In other words, It combines the Uniform-cost and the Greedy search algorithms.

2.6 Heuristic Functions

2.6.1 Manhattan distance

If (Snow has no dragon stones) ===> his goal is the dragon stones' store
The function is : abs(Snow.x - dragonstore.x) + abs (snow.y - dragonstone.y)
If (Snow has dragon stones) ===> his goal is the nearest White Walker
The function is : abs(Snow.x - WW.x) + abs (Snow.y - WW.y).

2.6.2 Euclidean distance

If (Snow has no dragon stones) ===> his goal is the dragon stones' store
The function is : SQRT (SQUARED (abs(Snow.x - dragonstore.x)) + SQUARED(abs (snow.y - dragonstone.y))).

If (Snow has dragon stones) ===> his goal is the nearest White Walker
The function is : SQRT (SQUARED (abs(Snow.x - dragonstore.x)) + SQUARED(abs (snow.y - WW.y))).

2.7 Running Examples

2.7.1 Example one

empty	dragonstone	empty	empty	empty
empty	obstacle	empty	empty	obstacle
ww	empty	empty	ww	empty
empty	ww	empty	empty	ww
empty	empty	empty	empty	snow

Expansion Technique : A* search using Manhattan heuristic function
Goal node reached => SUCCESS
Expansion number is 1490
level number is 13
Cost to reach goal => 109
Expansion Technique : A* search using Euclidean heuristic function
Goal node reached => SUCCESS
Expansion number is 1490
level number is 13
Cost to reach goal => 109
Expansion Technique : Greedy search using Manhattan heuristic function
Goal node reached => SUCCESS
Expansion number is 1490
level number is 13
Cost to reach goal => 14
Expansion Technique : Greedy search using Euclidean heuristic function

Goal node reached => SUCCESS
Expansion number is 1490
level number is 13
Cost to reach goal => 7
Expansion Technique : Breadth First
Goal node reached => SUCCESS
Expansion number is 1490
level number is 13
Cost to reach goal => 95
Expansion Technique : Depth First
Goal node reached => SUCCESS
Expansion number is 66
level number is 47
Cost to reach goal => 330
Expansion Technique : Iterative
Goal node reached => SUCCESS
Expansion number is 1114
level number is 13
Cost to reach goal => 95
Expansion Technique : Uniform Cost
Goal node reached => SUCCESS
Expansion number is 1490
level number is 13
Cost to reach goal => 95

2.7.2 Example two

empty empty ww empty empty
empty empty empty dragonstone empty
ww ww empty ww empty
empty empty empty empty empty
obstacle empty empty obstacle snow

Expansion Technique : A* search using Manhattan heuristic function
Goal node reached => SUCCESS
Expansion number is 533
level number is 11
Cost to reach goal => 83
Expansion Technique : A* search using Euclidean heuristic function
Goal node reached => SUCCESS
Expansion number is 533
level number is 11
Cost to reach goal => 83

Expansion Technique : Greedy search using Manhattan heuristic function
Goal node reached => SUCCESS
Expansion number is 533
level number is 11
Cost to reach goal => 8
Expansion Technique : Greedy search using Euclidean heuristic function
Goal node reached => SUCCESS
Expansion number is 533
level number is 11
Cost to reach goal => 3
Expansion Technique : Breadth First
Goal node reached => SUCCESS
Expansion number is 533
level number is 11
Cost to reach goal => 75
Expansion Technique : Depth First
Goal node reached => SUCCESS
Expansion number is 124
level number is 67
Cost to reach goal => 494
Expansion Technique : Iterative
Goal node reached => SUCCESS
Expansion number is 436
level number is 13
Cost to reach goal => 95
Expansion Technique : Uniform Cost
Goal node reached => SUCCESS
Expansion number is 533
level number is 11
Cost to reach goal => 75

2.8 Performance Comparison

It was noticed that Depth first Algorithm is not complete it keeps running in infinite loop but we prevented this by making the agent visit every state only once so the Depth first Algorithm terminates

It was noticed that Algorithms that uses any kind of sort (takes in consideration that path cost or heuristic function) takes more time to run as it needs to sort the data structure each time a node is expanded. But it guarantees

a solution. In our result and all the search strategies would get us the same result but the Iterative-deepening search algorithm gets less expanded nodes.

Complete Search Algorithms:

All search strategies we tested on random 5*5 4 white walker grid gets solutions .

Optimality:

We have observed that strategies Iterative deepening search and depth first search gets less nodes expanded but they do not get optimal solution other strategies gets optimal solution and gets the same number of expanded node and same solution. Aslo We guess the A* can perform better on Larger grids in terms of expanded nodes and optimality but we did not manage to get results for large grids .

2.9 Running Program Instructions

1) By running the main function in the SaveWestero.java file
2) You give Search(Cell[][] grid, String strategy, boolean visualize) parameters to run and it prints everything.
3) If you want to generate random grid you can use (Cell[][] mapR = MapGen.GenGrid(5, 5, 4).clone();)

2.10 AS for the citation we used couple of websites to make sure that our abstract implementation is going the right way using JAVA.