Media Engineering and Technology Faculty
German University in Cairo

Artificial Intelligence

Assignment I

Hady Mohamed 34-3051

Islama Elgohary 34-5639

Bavly Shenouda 34-4463

Contents

Chapter 1

Introduction

1.1 Understanding The Problem

The problem starts out with an n*m grid, with the cells of the grid containing one of the following objects:

- **Empty cell**: Can be accessed by the agent, but has no special properties.
- **White walkers**: Can not be accessed by the agent and act as enemies that the agent is meant to destroy.
- **Dragon glass**: Can be accessed by the agent to pick a given number of dragon glass to be used as weapons.
- **Obstacle**: Can not be accessed by the agent and has no special properties.
- **Jon Snow**: The search agent.

Jon Snow, the search agent, can do one of the following set of operations (as long as it obeys the current cell properties): Move in any of the main 4 direction, collect dragon glass from dragon glass cells, attack all white walkers in cells adjacent to him in the 4 main directions (attacking multiple adjacent white walkers still only costs one dragon glass).

The goal is for the search agent to find and kill all white walker on map, if that is possible given the set of operations and cell properties. The agent tries to achieve its goal by exploring all the different possibilities that are the outcome of trying different viable operations at different states. This clearly results in a search tree which we can easily visualize the agent trying to expand at every step, searching for its goal so it can execute it on the given grid. It helps to think of the agent "thinking" of all possible outcomes then executing a solution if one is found. The way the agent traverses/relaxes the aforementioned search tree, is dictated by the queuing strategy that we give our agent.

Chapter 2

Abstract Data Structures

2.1 Node

The node abstract data structure represents a generic node in the search tree that the agent has to explore while looking for solutions. Given the requirements of the task at hand, it became easy to think of how nodes would be represented. The data structure had the following attributes:

- **Node parent**: The parent Node of the current node. This is useful for printing the solution if on exists.
- **Operator operator**: the action/operation carried out by the agent.
- **int depth**: The depth of the node in the search tree. This is important for some search algorithms.
- **int pathCost**: The total path cost of getting to that node from the root.
- **State state**: This was introduced later to optimize the search performed and avoid repeating states that have been previously explored.

The data structure also has a method responsible for printing the step by step solution that the agent to chose based on the given search strategy to solve an instance of a problem. The function takes as parameters the final, goal, node and traces its parents until it reaches the root and then prints this sequence of nodes.

2.2 State

The generic state class can not contain any attributes, but is there for robustness, which is why the state class of the problem will be discussed instead. The save westros state class has the following attributes:

- **Cell cell**: The coordinates of the agent at the current state.
- **int dragonGlass**: The amount of dragon glass held by the agent at the current state.
- **int whiteWalkersLeft**: The count of remaining white walkers on the grid at the current state.
- **grid**: A version of the actual grid that is representative of the current state.

The data structure has a method that allows it to be compared to other instances of the same type. This was important for one of the data structures used (the data structure used was a tree map. Similarly, a hashing function would have been required in case of a hash map). The data structure also has a function for printing instances of this type. This is useful for various reasons but mainly for printing the final solution.

2.3 Problem

The search problem data structure represents a generic instance of the discussed problem and has the following attributes:

- **ArrayList¡Operator¿ operators**: A list of operators
- **State state**: An instance of the state object encapsulating the information of the initial state of the problem.

The data structure also had a method responsible for checking whether or not a given state is a goal state or not.

2.4 Operator

A generic class representing the actions that the agent should be able to perform. It has the following attributes and functions:

- **String name**: a string defining the name of the operation
- **int cost**: A number representing the cost of applying the current instance of an operator (cost of applying a certain operation).
- **Node run(Node node)**: A function that applies an instance of the operator on an instance of a node and returns the new node that results from applying this operator.

Chapter 3

Save Westros

3.1 Save Westros Problem

A sub class of the generic problem data structure, discussed earlier in this report, that acts is specifically tailored for the problem discussed in this project and it has the following attributes and methods:

- **int n**: The length of the grid.
- **int m**: The width of the grid.
- **char[][] grid**: A 2-dimensional grid representing the world in which the agent is in. The possible cells were discussed in 1.1
- **char WHITE_WALKER**: represents a white walker with character(w).
- **char EMPTY_CELL**: represents an empty cell on the grid and is represented by char (.).
- **char OBSTACLE**: represents an obstacle cell on the grid and is represented by char (o).
- **char DRAGONGLASS**: represents a dragon glass cell on the grid and is represented by char (d).
- **char JON_ON_DRAGONGLASS**: represents a dragon glass cell on the grid and is represented by char (d).
- **char[] grid_elements**: A 1-dimensional array holding different blocks of the grid. The frequency of each element represents how likely it is to be generated.
- **void initializeOperators()**: A method that initializes the problem's set of operators.

- **void genGrid()**: A method that generates either an n*m or a 4*4 grid (if non-positive numbers were entered by the user) with random cells, but with higher weight given to empty cells.
- **boolean goal(State state)**: A method that returns true if the given state is a goal state or false otherwise.

Chapter 4

Main Functions

4.1 genGrid()

The Function is implemented in SaveWestros. It creates a grid with dimensons n x m where n is the number of rows and m is the number of columns. If the user enters a non-positive number, the function generates a 4 x 4 grid. The gris i then randomly filled with objects discussed in 1.1 with more weight given to empty cells, to reduce the possibility of producing unsolvable grids and not overwhelm the agent with inaccessible cells. The weight is controlled through an array where the frequency of grid elements dictate its weight to be randomly assigned to a cell in the grid. Finally, the agent (JON_SNOW) is placed on the right most bottom most cell of the grid and initial state is created. The number of dragon glass the agent starts with, is randomly chosen but has an upper bound equal to the number of white walkers + 1. The initial state contains the initial location of Jon Snow, the number of dragon glass it starts with and the number of white walker currently on the grid.

4.2 search(problem, Search Queue, visualize)

The function is implemented in Search Class. It applies the generic search algorithm. It takes as input the problem instance, an instance of search queue and a boolean to visualize the grid. The function first initializes the queue by creating a node for the initial state (root). While the queue is not empty, the function takes the first node in the queue. If the node is a goal, the function terminates or it prints the steps from the initial node to the goal if visualize is true. If the state is not a goal state, then the function calls expand on the node and adds the resulting nodes to the queue.

4.3 expand()

The function is implemented in the Search class. it takes a node and an array list of operators. The function runs the operators on the nodes and returns an array list of the resulting nodes.

Chapter 5

Search Algorithms

5.1 Generic Search Queue

An abstract class that defines a generic queue data structure. It contains only two variables, the initial node and a TreeMap to hold the already visited nodes. Different extensions of this abstract class will define the ordering of the queue, which will in turn control how the exploration works in different search algorithms.

5.2 Breadth-First Search

Breadth First search implements Generic Search queue, applying a traditional queue functionality to it. The first in first out behaviour guarantees that a breadth first search is achieved. Starting from the initial state, all the possible operators' are added to the queue, thus even when the head node is dequeued and its children is added to the queue, the nodes from the parent level are still ahead of the children in the queue.

5.3 Depth-First Search

Depth First Search implements Generic Search queue, but rather with the behaviour of a stack. The first in last out behaviour guarantee that whenever a node is explored, all its children are put at the head of the search queue, hence its children shall be explored first, ahead of all higher level nodes. This procedure insures a depth first search.

5.4 Iterative Deepening Search

Iterative Deepening Search implements Generic Search queue again in the form of a stack to support first in last out behaviour. In addition, it keep also introduces 2 new variables, which indicated the current depth in the search tree as well as the maximum allowed depth in the search tree. A node is only added to the stack if its depth is less than or equal the maximum depth allowed. This queue imitates a similar behaviour to the one introduced in Depth-First search, yet the depth check allows the search algorithm to check for a solution in the shallower parts of the search tree before it goes any further.

5.5 Uniform Cost Search

Uniform Cost Search implements Generic Search queue as priority queue that prioritize the nodes in the queue according to the least possible cost first. To insure that the tree is always growing in the direction of the most cost efficient direction. To add a node to the queue, it is first checked it already exist in the visited nodes list that is present in the parent class Generic Search queue and is added if the state is un-visited or visited yet the cost presented by the new node introduces a cheaper option to reach the same state.

5.6 Greedy Search

Greedy Search extends Uniform Cost Search. Which means it also has a priority queue, but the major difference here is the nodes comparison. A new specific node comparator is implemented that uses one of the heuristic functions discussed earlier instead of the cost used in Uniform Cost search. The priority queue uses this new comparator in order to arrange the nodes with the lowest cost according to the heuristic function to be at the head of the queue.

5.7 A* Search

A* search also implements Uniform Cost Search, but again with a new competitor. The new comparator joins a node's path cost and add it to the cost introduced by the chosen heuristic function and return a total joint cost. This new comparator is used to order the search queue with the lowest cost node at the head. This means that both the cost to reach the current node as well as the heuristic cost of each following move is taken into consideration when deciding which node to explore next. Which defines the A* search behaviour.

Chapter 6

Heuristics

6.1 remainingWalkers()

The function takes a node and returns the cost of attacking the white walkers. It is calculated by multiplying the number of remaining walkers by the cost. The number of walkers is divided by 3 to account for the number of walkers that can be attacked with one attack action (if Jon is in a cell between 3 walkers).

6.1.1 Admissibility Argument

If the remaining walkers is less than 3, then the cost will be 0. If the number of walkers is 3, the cost will be that of killing 1 walker. Otherwise, since the number of walkers is divided by 3, the function will never overestimate the cost because there will be 2 cases. The first being that the walkers can all be attacked at once, then the cost will be accurate. Else, The cost will be less than the actual cost. The function also does not account for the cost of approaching the walkers.

6.2 distanceToFarthestWalker()

The function takes a node and returns the cost of walking to the farthest white walker. It calculates the manhattan distance to the walker.

6.2.1 Admissibility Argument

The function does not add the cost of attacking the white walker which is much greater than the cost of reaching the white walker, hence, it will never overestimate the cost.

6.3 distanceAndRemainingWalkers()

The function takes a node and returns the cost of walking to the farthest white walker plus the cost of killing the remaining white walkers. The function adds the results of the aforementioned functions to calculate the cost.

6.3.1 Admissibility Argument

The actual cost will be the total distance covered + the cost of picking dragonglass + the cost of killing the white walkers. Since we never overestimate the cost of reaching the white walkers and the cost of killing the white walkers and we do not add the cost of picking the dragon glass, the function will never overestimate.

Chapter 7

Running The Code

The algorithms were tested against 2 grids.

.	.	w	.
.	.	.	.
.	.	w	.
w	.	d	j

Table 7.1: Grid 1

.	w	.	w
d	.	.	.
.	.	.	.
.	.	.	j

Table 7.2: Grid 2

7.1 How it works

The main method is in "saveWestros.WestrosSearch".

- choose the queuing function by initializing the desired queue as the 2nd parameter to the search function in the main method. You can choose from:
  - DepthFirstSearch()
  - BreadthFirstSearch()
  - IterativeDeepeningSearch()
  - UniformCostSearch()

  - GreedySearch(h)
  - AstarSearch(h)

  where h is a string representing the heuristic function and can be "h1"(remainingWalkers), "h2"(distanceToFarthestWalker) or "h3"(distanceAndRemainingWalkers)

- To change the maximum number of dragon glass pieces Jon can carry, modify "maxDragonGlass" in "saveWestros.JonOperator"
- Run the code
- Do you want to randomize the grid? enter y for yes and n for no
- If user chooses not to randomize the grid, they need to choose between grids 1 and 2 by typing either 1 or 2.
- If they choose to randomize the grid, The grid is randomly generated.
- If a solution exist The steps to the solution are printed in the console. If it does not, the user is informed that the grid has no solution and the grid is printed.

7.2 Comparing Search Strategies

7.2.1 Completeness

Since we memoize the states we never revisit a state. Hence, If a solution exists, it will always be found. If a solution does not exist, the program will terminate. In this specific case, our implementation of DFS is complete. BFS is always complete and UCS is also complete since the cost is increasing with the depth. However, for the iterative deepening, the memoized states are always reset on each iteration therefore, we give it a limit of $1 * 10^9$ to prevent it from running forever.

7.2.2 Optimality

Since an optimal solution uses the minimum number of dragon glass pieces, we assigned the attack a cost of $n * m + 1$ where n and m are the dimensions of the grid. We give all other actions a cost of 1 which will make it cheaper to walk the whole grid than to attack a white walker. This way, we ensure that the agent will prefer to attack multiple white walkers in one attack even if it means making more moves.

Given that DFS does not account for the number of dragon glass pieces used, therefore it is not optimal. BFS and ID are both optimal since the cost increases with the depth. The Greedy algorithm is not completely optimal since it does not take into account the actual path cost. The UCS and A* algorihtms both give the optimal solution. However UCS expands more nodes , since it uses only the actual path cost, making it less effecient compared to the A* which uses both the path cost and a heuristic function.

7.2.3 Expanded Nodes and Total Cost

- DFS: On the first grid, DFS expanded 13 nodes and had a total cost of 60. On the second grid, it expanded 27 nodes with a total cost of 74.
- BFS: On the first grid, BFS expanded 100 nodes and had a total cost of 40. On the second grid, it expanded 117 nodes with a total cost of 45.
- ID: On the first grid, ID expanded 357 nodes and had a total cost of 57. On the second grid, it expanded 426 nodes with a total cost of 45.
- UCS: On the first grid, UCS expanded 174 nodes and had a total cost of 40. On the second grid, it expanded 163 nodes with a total cost of 45.
- Greedy: Using the cost of killing the remaining walkers heuristic function. On the first grid, It expanded 29 nodes and had a total cost of 60. On the second grid, it expanded 35 nodes with a total cost of 64.
- A*: Using the Manhattan distance + the remaining walkers as the heuristic function. On the first grid, It expanded 153 nodes and had a total cost of 40. On the second grid, it expanded 153 nodes with a total cost of 45.

From the previous results. we can see that for this particular problem, DFS is the worst algorithm cost-wise while UCS and A* are optimal. BFS did get the optimal solution as well, however, that is only because the path cost increases with the depth. We also see that Greedy did not give the optimal solution. In terms of the number of Expanded nodes, ID used the maximum number of nodes followed by UCS then A* then BFS followed by Greedy and DFS.

ID expanded the maximum number of nodes since it expands the same node multiple times in each iteration and adds to it new nodes. UCS gets the solution with the optimal cost by exploring all the options which explains why it expanded a relatively high number of nodes. As of A*, A8 is similar to UCS in that it uses the path cost to get the cheapest route. However, it uses the heuristic function to remove the need to check all possible routes and that is why it has a high number of expanded nodes but less than the nodes expanded by UCS. Greedy chooses the immediate cheapest step according to the heuristic function thus, trying to minimize the heuristic function of the number of white walkers, the algorithm also reduced the number of expanded nodes to achieve the goal. As for BFS and DFS, The number of nodes depends only on the problem since each of them is non-informed about the goal state's relative location or the cost.