Artificial Intelligence Report

Alaa Kurdi 34-878 T-11
Ali Waleed 34-673 T-07
Ahmed Kamal 34-3110 T-11

October 18, 2018

1 Problem Description

The problem lies within a MxN Grid where n is greater than or equal to 4 environment where there's an agent starts at the initial state at specific node , The grid at the initial state should contain cells of the following 4 types

1. John's state where our agent lies at the initial state.
2. Randomly Placed Obstacles , those nodes shall not be expanded or part of the path as they are not accessible.
3. Randomly Placed White Walkers , White Walkers are Opponents that should be killed and vanished to reach our goal.
4. Randomly Placed DragonStone Cell, this cell is accessed to gain Dragon Glasses .
5. Randomly Placed Free Cells , those cells are accessible to the agent to move through.

The problems consists of some operators that are used to make transitions between grid through the map .The operators allowed are the following:

***Note: Any of the following moves would only happen if the operator would lead to node that are within boundaries of our map.*.**

1.1 Transition operators:

1. Move North (where the user can move one cell up
2. Move South (where the user can move one cell down
3. Move East (where the user can move one cell to the right
4. Move West (where the user can move one cell to the left

The user can use operators that would result only in new state ; in details, user can move to any of the surrounding four cells using the operators only if the resulted node position is either a position of a free cell or a White Walkers cell , if it's a free cell that the user can move freely to the next node position , else if it's a white walker than the agent should kill the white walker before moving to the next position . The agent can kill any number of white walkers within the surrounding with one dragon glass and this dragon class is then not usable. The agent should prioritize to kill white walkers if he has dragon glass at any state (the user should prioritize to be in a state that allows him to kill many white walkers with the least number of dragon glass, so if the agent is surrounded by three white walkers then this is the best case as the user can kill them all with one dragon glass) . if the users' dragon glasses are finished and the users' next node to access is of type white walker then he must find a path to the Dragon stone to collect more glasses to complete his journey towards the goal (which is killing all the white walkers). After killing any white walker , the cell becomes free and the user can then move through it .

1.2 The Goal

Killing all white walkers with the least number of dragon glasses.

2 Search Tree Node ADT

Search tree node consists of the following in our implementation

1. A state which is instance of the class state , Class State contain the following:
   - A state string that encapsulates the information at the current state including the position , number of dragon glasses left , number of white walkers left , and the updated grid which will be discussed in the save Westeros discussion.
2. Another node , which is the parent of the current node to keep track of the path
3. An Operator consists of the name of the operator and the cost calculated after using the current operator
4. The depth , the depth is recursively calculated as referring to the parent node so (1+ depth(parent node))
5. the path cost , which is the cost from the initial state to the current node which is calculated recursively by referring the previous costs taken until reaching this node . so (1+ pathcost (parent node))

In our problem

1. the position of the node in the MxN grid in terms of x axis and y axis coordinates
2. An updated grid that defines current grid after series of actions (so if a cell initially was a white walker after it's killed it will be free then at the next states' grids this cell's type should be free
3. Number of white walkers that are left at the current state
4. Number of dragon glasses left at the current state
5. The task which helps to define the next action so if the task is a "Refill" then this means we have no Dragon glasses and we should go to the dragon stone , else if it's a "Kill" then this means a user have dragon glasses and can proceed.

   In our problem the Westeros Operator Class does contain also the following

   - The cost of killing a white walker .
   - The change in both x and y which would happen after executing this operator .

3 Search Problem ADT

problem in the generic search tree defines the following functions

1. A list of operators , which is the actions that can be taken in the problem.
2. A function **getOperators()** that returns the operators of the current problem.
3. A function **getInitialState()** that gets the initial state which where john is starting from.
4. A boolean function **testgoal()** that checks if the input state is a state where the goal is achieved.
5. A **toString()** function of the grid to visualize the grid as a string.

Note: This was the abstract problem , Our problem is discussed in the save Westeros discussion.

4 SaveWesteros Problem

We have the saveWesteros problem class that defines the following:

1. A 2D Grid which will contain the cells .
2. N and M which are the max coordinates of our grid.
3. The types of the cells that will be generated by the grid :- White Walkers , Obstacles , and Dragon Stone .

Note: Type john is initialized at a specific position when generating the grid.

The class consists of three Main Functions , one that creates and generates the grid and one that generates the Operators of our problem that we need to define when we initialize the problem.

- **InitializeGrid()** simply generates a grid of NXM of randomly assigned types of cells , but we make sure that John lies with a specific position by overriding the grid with John's position. We also make sure that there's one Dragonstone that we also overrides the grid cells at a random position , and we make sure that this position is not John's position . Now we have a grid that contain some white walkers, obstacles , free cells , John's cell , and one Dragon glass cell
- **InitializeOperators()** initializes instances of class operator that defines the series of actions or operators that can be executed and the consequence of executing them as every operator as discussed takes the Operator name , the change in the x and y that will occur after executing the operator , and the cost of executing this operator.

- **TestGoal()** (Overrides the TestGoal() function in the General Problem abstract class) that checks if the goal of the problem is achieved or not , we check in our problem of the count of the white walkers is 0 then it will return true then the goal is achieved; False otherwise.
- **GridToString()** (Overrides the GridtoString() function in the General Problem abstract class) that visualizes the grid cells in a string.

5 Main functions

Here we will talk about the main functions that work to solve the problem with the search strategy as an input

First we will start with the search function which will lead us to the other main functions sequentially

The main Search function takes the following 3 parameters as an input

- A **problem** , which is our saveWesteros problem in our case .
- A **Search** Strategy which defines what Search algorithm will be implemented .
- **Visualize** Boolean , which is true when the goal is found so the program prints the path to goal and the expansion sequence of the solution

The functionality is as follows regardless of the search strategy

- we get the initial state of our problem
- we make a queue and add this node to the queue
- we initialize a variable called expanded nodes which is a counter of the nodes that we have expanded which is removed from the queue
- we remove the first element of the queue which is the initial node we have just added
- The removed node should be expanded next and added to the arraylist of expanded nodes , and here we come to define the functionality of expanding a node .

The node expansion function is called expand() and it takes the following two parameters:

- A node
- A list of operators of our problem .

Note: We expand the node using all the operators in our array list which means we need to apply the operators on the current node to define what to expand next so we need to handle this apply functionality with all its logic and here comes the apply().

The apply() takes the following as an input :

1. An Instance of WesterosState , Westeros state(x coordinate ,y coordinate , Number of dragon glasses at the current state , numbers of white walkers left) defines the state in our problem , the class contain the following important functions
    - The WithinBoundaries() checks if the cell is within boundaries of our MxN grid
    - The WhiteWalkersNeighbours() checks if there's any white walker around at the current state .
    - The WhiteWalkersNeighboursPositions() return the position of the white walkers around if there is any .
2. A grid that is the grid of the current state of the node
3. Boolean KilledWW that we use to check if we killed a white walker or not to consider the cost.
4. The ChangeInX and ChangeInY attributes which defines the change in x and the change in y
5. The number of Dragon Glasses at the current state of the node.
6. ArrayList of nodes which is updated with the positions of the white walkers around the current node at the current state

Note: First we check if the node is within boundaries , if it's a white walker or an obstacle then we cannot apply any operators on it so we return nothing ;Otherwise , it's a white walker . we check also if the current task is to refill or to kill , if it's a refill then the dragon glasses count is 0 then we check if the new cell is of type dragon stone then we add the dragon glasses and return this new node with the new state . Otherwise it means we have dragon glasses , then we check the positions of the white walkers around us if there's any and keep track of their positions and we decrement the dragon glasses count and change the KilledWW to true which means we have killed a white walker.

The next step as we collected the information needed is to change the grid and kill the whitewalker and this is implemented as the following:

1. Generate a new grid with the current new state
2. As we have the positions of the white walkers as previously mentioned , now we can turn the white walkers cell in the positions arraylist to free which means they were around the agent and is already killed.
3. we update the newly generated grid with the new types of the white walkers which are now free cells.
4. we use the KilledWW Boolean to check if it's true then we have killed a white walker and we need to add this cost of killing a white walker to the current state cost.
5. finally we return the node to the **expand()** function if there's any.

Note: if the node is not null then we add it to the list of expanded nodes and return this list to the search function we mentioned at the beginning. we apply the add function of the current search strategy and add them expanded list which defines the path .

6 Search strategies

We override the function of the General Search Strategy Class.

1. **Breadth First Search**

   - We create a queue using a Linked List data Structure. We have a visited ArrayList in the general Search strategy class of the visited nodes with the same states so we don't visit them again.
   - We override the **Add()** function of the Generic class with adding nodes to the queue but before that we check if they are visited before then we don't add them ; otherwise , we update the visited array and add them to the list
   - We override the **RemoveFirst ()** of the generic class by removing the first element of the queue and return it so we can apply our logic on it later.
   - **isEmpty()** check if the queue is empty.

2. **Depth First Search**

   - **Add()** same as Breadth first search but we push to the stack.
   - **RemoveFirst()** return the element at the top of the stack .
   - **isEmpty()** check if the stack is empty

3. **Iterative Deepening**
   Same as **DFS** but

   - **isEmpty()** we check if the the stack is empty then we return true ; otherwise, we check if it's not empty and the current depth is equal to the maximum depth then we have to increment the depth to start the search all over but with new incremented depth.
   - **Add()** we make sure the depth of the current node is smaller than or equal to the limit depth then we can add it and expand it later and we also make sure if it's already visited before or not so we don't repeat the same states again.

4. **Uniform Cost**

   - Same functions as BFS but when we add to the queue we make sure it's sorted in ascending order according to the path cost we can expand the not yet expanded nodes with the least cost . The path cost is defined based on the transition operators cost and an action operator "kill".

5. **Greedy Search**

- Same the BFS but when we add to the queue we make sure the queue is sorted in ascending order according to the Heuristic function so we can expand the least first .
- We implement the heuristic function which will be discussed later in the heuristic functions discussion.

6. **A* Search**

   - Same as the Greedy but now we sort the queue ascendingly according to the Heuristic function added to the Path Cost

7 Heuristic Functions

1. **First Heuristic Function :**

   (a) **Explanation** Our heuristic function is function of two important variables , the number of white walker left at the currently estimated state and the max number of white walkers that can be killed at this state , the function is as follows :- int heuristic = (int) Math.ceil(The Number of currently alive white walkers/ maximum number of white walkers that can be killed at the best case));

   (b) **Admissibility** The heuristic is admissible because the heuristic function worst case scenario is to kill each white walker left at a different state not at once which is equal to the real path cost but will never exceed it.

2. **Second Heuristic Function :**

   (a) **Explanation** Our heuristic function is function of two important variables, it depends on the position of Jon on the grid and the cell types on the grid and the task of the agent whether the agent's task is to kill the white walkers or reach the DragonStone and refill, the function is as follows :- At the start of the function we check if Jon's position is in DragonStone, since we refill if Jon does not have any DragonGlass and is entering the DragonStone cell, therefore if the number of DragonGlass has increased in comparison to the number of DragonGlass of the parent node, then Jon wanted to refill, therefore this should be given the highest priority and thus we return 0 in this case. In another case if Jon does not have any DragonGlass, the higher priority should be reaching DragonStone to refill his DragonGlass, therefore we return the distance between Jon and the DragonStone by iterating until we reach the cell of DragonStone on the grid to a get to know the coordinates of the DragonStone. The function returns (Math.abs(DragonStoneX - JonX) + Math.abs(DragonStoneY - JonY))

   In the final case, this case would be that Jon is attempting to kill WhiteWalkers, therefore the higher priority should be given to getting closer to WhiteWalkers, thus killing them. We iterate to find the minimum distance between a WhiteWalker and Jon which is calculated like the previous case (Math.abs(WhiteWalkerX - JonX) +

Math.abs(WhiteWalkerY - JonY)). We iterate through the whole grid trying to find the minimum distance between Jon and a WhiteWalker, therefore in other words the function returns the Minimum value of (Math.abs(WhiteWalkerX - JonX) + Math.abs(WhiteWalkerY - JonY)) calculated.

(b) **Admissibility** The heuristic is admissible because in a 4x4 grid the worst case would be that w White Walker is found on the top left cell represented by (0,0) and Jon in the bottom right represented by (3,3) the heuristic function would return a value of 6 and the real path cost would be 6 only to reach the WhiteWalker without adding the killing cost, therefore the heuristic function is underestimating the cost and would never exceed the real path cost.

8 Testing and Comparing

1. **Round 1**

   - Start of round

     Free WhiteWalker WhiteWalker Free

     Free DragonStone WhiteWalker WhiteWalker

     WhiteWalker Obstacle Free Free

     Free WhiteWalker Obstacle Jon

   - **BF**
     List of actions: North, Kill, West, Kill, North, Kill, West, Kill, West, Kill, South, South, Kill,
     Number of Expanded nodes = 16
     The Path length = 13
     The Path Cost = 127

Kill North 1 3 DragonGlass:2, Kill North 1 2 DragonGlass:1, Kill North 0 2 DragonGlass:0, Collected DragonGlass 1 1, Kill North 0 1 DragonGlass:2, Kill South 2 0 DragonGlass:1, Kill East 3 1 DragonGlass:0

   - **DF**
     List of actions: North, Kill, West, Kill, North, Kill, West, Kill, West, Kill, South, South, Kill,
     Number of Expanded nodes = 16
     The Path length = 13
     The Path Cost = 127

Kill North 1 3 DragonGlass:2, Kill North 1 2 DragonGlass:1, Kill North 0 2 DragonGlass:0, Collected DragonGlass 1 1, Kill North 0 1 DragonGlass:2, Kill South 2 0 DragonGlass:1, Kill East 3 1 DragonGlass:0

- **ID**
  List of actions: North, Kill, West, Kill, North, Kill, West, Kill, West, Kill, South, South, Kill,
  Number of Expanded nodes = 600
  The Path length = 13
  The Path Cost = 127

Kill North 1 3 DragonGlass:2, Kill North 1 2 DragonGlass:1, Kill North 0 2 DragonGlass:0, Collected DragonGlass 1 1, Kill North 0 1 DragonGlass:2, Kill South 2 0 DragonGlass:1, Kill East 3 1 DragonGlass:0

- **UC**
  List of actions: North, West, Kill, North, Kill, North, Kill, West, South, West, Kill, South, South, Kill,
  Number of Expanded nodes = 125
  The Path length = 14
  The Path Cost = 109

Kill North 1 2 DragonGlass:2, Kill North 0 2 DragonGlass:1, Kill East 1 3 DragonGlass:1, Kill West 0 1 DragonGlass:0, Collected DragonGlass 1 1, Kill South 2 0 DragonGlass:2, Kill East 3 1 DragonGlass:1

- **Greedy1**
  List of actions: North, Kill, North, North, Kill, West, Kill, West, South, West, Kill, South, South, Kill,
  Number of Expanded nodes = 44
  The Path length = 14
  The Path Cost = 109

Kill North 1 3 DragonGlass:2, Kill West 0 2 DragonGlass:1, Kill South 1 2 DragonGlass:0, Kill West 0 1 DragonGlass:0, Collected DragonGlass 1 1, Kill South 2 0 DragonGlass:2, Kill East 3 1 DragonGlass:1

- **Greedy2**
  List of actions: North, Kill, North, North, Kill, West, Kill, West, South, West, Kill, South, South, Kill,
  Number of Expanded nodes = 127
  The Path length = 14
  The Path Cost = 109

Kill North 1 3 DragonGlass:2, Kill West 0 2 DragonGlass:1, Kill South 1 2 DragonGlass:0, Kill West 0 1 DragonGlass:0, Collected DragonGlass 1 1, Kill South 2 0 DragonGlass:2, Kill East 3 1 DragonGlass:1

- **A*1**
  List of actions: North, Kill, North, North, Kill, West, Kill, West, South, West, Kill, South, South, Kill,
  Number of Expanded nodes = 127

The Path length = 14
The Path Cost = 109

Kill North 1 3 DragonGlass:2, Kill West 0 2 DragonGlass:1, Kill South 1 2 DragonGlass:0, Kill West 0 1 DragonGlass:0, Collected DragonGlass 1 1, Kill South 2 0 DragonGlass:2, Kill East 3 1 DragonGlass:1

- **A*2**
  List of actions: North, Kill, North, North, Kill, West, Kill, West, South, West, Kill, South, South, Kill, Number of Expanded nodes = 126
  The Path length = 14
  The Path Cost = 109

Kill North 1 3 DragonGlass:2, Kill West 0 2 DragonGlass:1, Kill South 1 2 DragonGlass:0, Kill West 0 1 DragonGlass:0, Collected DragonGlass 1 1, Kill South 2 0 DragonGlass:2, Kill East 3 1 DragonGlass:1

Comparison

(a) **General**

- **Completeness** All strategies are proved to be complete .
- **Optimality** The Uniform cost , Greedy and A* search were the most optimal provided that the cost never decrease at any state , The BFS were not Optimal as we know for BFS to be optimal all costs should be the same .As it does not take prioritize the lower costs operations , higher costs operators are executed with respect to the order of the queue which is not right. The DFS is not optimal as it doesn't has the least cost out of all the implemented strategies . To be optimal all white walkers should be descendants of the first expanded node which is not the case here .The iterative deepening has the a short path length as it keeps in the memory only the upper nodes which is not the most nodes when iterating, but it does has a very high cost so it's not optimal .
- **Expanded Nodes** The DF is efficient in terms of expanded nodes as it expands only 16 nodes , this is a consequence of having 5/6 of the total white walkers children of each other which proves the efficiency in terms of expanded nodes in this case of the DFS; meanwhile, the ID expands a lot of nodes as it Iterates many times over the same nodes. The Greedy*1 also expanded only 44 nodes which shows the influence of considering the number of left white walkers as part of the heuristic function . Other Strategies were close in terms of high number of expanded nodes .

(b) **BFS**

- **Completeness** Complete as it should be.

- **Optimality** Not Optimal , we know for BF to be optimal all path costs should be the same as we ignore the costs , taking into consideration states with higher cost than others with lower cost which comes first at the queue including free cells that are visited multiple times .
- **Expanded Nodes** = 122 , as we discussed the algorithm keep expanding nodes as level by level with no consideration to the cost , so cells with lower costs at some states are visited multiple times.

(c) **DFS**

- **Completeness** In this case it's complete as we keep expanding and we will never be stuck , also the grid is finite and we keep track of the visited states so the case where we fall in a loop of transition between same states will never happen. Optimality:Not optimal , as it doesn't has the least cost out of all the implemented strategies . To be optimal all white walkers should be descendants of the first expanded node which is not the case here
- **Expanded Nodes** = 127

(d) **UC**

- **Completeness** Complete , provided that the cost is not decreasing at any state in the search
- **Optimality** In this case , it's optimal with respect to the strategies implemented in the scope

2. **Round 2**

- Start of round
  Free Free WhiteWalker Obstacle
  Free Obstacle Free Obstacle
  DragonStone Obstacle WhiteWalker Free
  WhiteWalker WhiteWalker WhiteWalker Jon

- **BF**
  List of actions: Kill, West, Kill, North, North, Kill, North, West, West, South, South, Kill,
  Number of Expanded nodes = 65
  The Path length = 12
  The Path Cost = 88

Kill West 3 2 DragonGlass:2, Kill North 2 2 DragonGlass:1, Kill West 3 1 DragonGlass:1, Kill North 0 2 DragonGlass:0, Collected DragonGlass 2 0, Kill South 3 0 DragonGlass:2

- **DF**
  List of actions:Kill, West, Kill, North, North, Kill, North, West, West, South, South, Kill,

Number of Expanded nodes = 27
The Path length = 12
The Path Cost = 88

Kill West 3 2 DragonGlass:2, Kill North 2 2 DragonGlass:1, Kill West 3 1 DragonGlass:1, Kill North 0 2 DragonGlass:0, Collected DragonGlass 2 0, Kill South 3 0 DragonGlass:2

- **ID**
  List of actions: Kill, West, Kill, North, North, Kill, North, West, West, South, South, Kill,
  Number of Expanded nodes = 344
  The Path length = 12
  The Path Cost = 2

Kill West 3 2 DragonGlass:2, Kill North 2 2 DragonGlass:1, Kill West 3 1 DragonGlass:1, Kill North 0 2 DragonGlass:0, Collected DragonGlass 2 0, Kill South 3 0 DragonGlass:2

- **UC**
  List of actions: Kill, West, Kill, North, North, Kill, North, West, West, South, South, Kill,
  Number of Expanded nodes = 70
  The Path length = 12
  The Path Cost = 88

Kill West 3 2 DragonGlass:2, Kill North 2 2 DragonGlass:1, Kill West 3 1 DragonGlass:1, Kill North 0 2 DragonGlass:0, Collected DragonGlass 2 0, Kill South 3 0 DragonGlass:2

- **Greedy1**
  List of actions: Kill, West, Kill, North, North, Kill, North, West, West, South, South, Kill,
  Number of Expanded nodes = 31
  The Path length = 12
  The Path Cost = 88

Kill West 3 2 DragonGlass:2, Kill North 2 2 DragonGlass:1, Kill West 3 1 DragonGlass:1, Kill North 0 2 DragonGlass:0, Collected DragonGlass 2 0, Kill South 3 0 DragonGlass:2

- **Greedy2**
  List of actions: Kill, West, Kill, North, North, Kill, North, West, West, South, South, Kill,
  Number of Expanded nodes = 69
  The Path length = 12
  The Path Cost = 88

Kill West 3 2 DragonGlass:2, Kill North 2 2 DragonGlass:1, Kill West 3 1 DragonGlass:1, Kill North 0 2 DragonGlass:0, Collected DragonGlass 2 0, Kill South 3 0 DragonGlass:2

List of actions: Kill, West, Kill, North, North, Kill, North, West, West, South, South, Kill,
Number of Expanded nodes = 69
The Path length = 12
The Path Cost = 88

Kill West 3 2 DragonGlass:2, Kill North 2 2 DragonGlass:1, Kill West 3 1 DragonGlass:1, Kill North 0 2 DragonGlass:0, Collected DragonGlass 2 0, Kill South 3 0 DragonGlass:2

- **A*2**
  List of actions: Kill, West, Kill, North, North, Kill, North, West, West, South, South, Kill,
  Number of Expanded nodes = 70
  The Path length = 12
  The Path Cost = 88

Kill West 3 2 DragonGlass:2, Kill North 2 2 DragonGlass:1, Kill West 3 1 DragonGlass:1, Kill North 0 2 DragonGlass:0, Collected DragonGlass 2 0, Kill South 3 0 DragonGlass:2

Comparison

- **Completeness** All are complete and achieve the goal.
- **Optimality** In terms of optimality , The DF seems the most optimal with respect to other Implemented strategies as it expands significantly less nodes than other search strategies with the same cost . The first heuristic showed more efficiency than the second heuristic function when applied to greedy search strategy as it has expanded only 31 nodes .
- **Expanded Nodes** 27 for DF is the best as most of the white walkers were descendants of each other so there was no exceeded backtracking which means expanding less nodes , 31 for First greedy and 344 for iterative deepening which is a huge number of ID which means that the strategy kept expanding and incrementing the depth to reach the goal of killing the last white walker which falls at a deep level.

9 How it works

By running the main method in the westerosSearch class. The westerosProblem is initialized and thus creating a random 4x4 grid. The 4x4 grid will be printed out for the user and then the user would be asked to enter the search strategy they want to try, these codes are the ones used to choose a search strategy.

- BF: Breadth-First Search.
- DF: Depth-First Search.
- ID: Iterative-Deepening Search.
- UC: Uniform-Cost Search.
- GS1: Greedy Search with the first heuristic function
- GS2: Greedy Search with the second heuristic function
- AS1: A* Search with the first heuristic function
- AS2: A* Search with the second heuristic function
- The following are the heuristic functions available:
  - Number of white walkers alive / 3
  - Manhattan distance between Jon's position and the nearest white walker
- The Output of the Strategy
  - In case there is a possible way for Jon to win:
    1. The path Jon moved to reach the goal state
    2. The length of the path
    3. The cost of the path
    4. The number of expanded nodes
    5. The starting grid for the user to trace the agent's actions
    6. A list of kills and refills made throughout the game
  - In case Jon failed to win:
    1. A print statement of failure appears