Team 74

Radwa Sherif

Alaa Megahed

18 October 2018

Project 1 Report

Problem Description:

In this project, the aim is to implement a general search procedure with multiple search (queueing) strategies, and use them to solve the SaveWestros problem where Jon Snow is placed in a 2D grid with whitewalkers, and obstacles. The goal is to kill all whitewalkers with the minimum number of dragonglasses, that can be obtained from the dragonstone (limited by the maximum number of pieces allowed for Jon to carry at once).

Search-Tree Node:

We implemented a class for general search-tree nodes.

As explained in the lecture, a node should have knowledge of:

1. the state to which the node corresponds,
2. the parent node,
3. the operator applied to generate this node,
4. the depth of the node in the tree, and
5. the path cost from the root.

In addition to the variable "evaluate", that is used to compare different nodes and is set differently depending on the strategy used for search. For Greedy strategies, it is set to the values of the heuristic functions, for all other search strategies, it is set to the sum of the heuristic value + the path cost, given that the heuristic function evaluates to 0 in case of using uninformed search strategies. Moreover, the compareTo(Node node) method resolves the conflicts of two nodes having the same "evaluate" values by comparing the depths.

Search Problem:

Problem ADT:

We implemented an abstract class for general search problems.

For each problem, the following need to be defined:

1. the initial state,
2. a list (array) of all possible operators,
3. a path cost function, which is defined by stating the cost of each operator,
4. a heuristic evaluation function to be used in informed search algorithms (in an uninformed search, it always returns 0)
5. a goal test, and
6. a function that expands a node and returns a list of the resulting nodes.

All of the information has to be defined in the subclasses of the SearchProblem, since they have different implementations depending on the specifications of each problem.

Generic Search Procedure:

In the SearchProblem class, we implemented an instance generic search method similar to the one mentioned in Lecture 2 slides.

The method takes the strategy as input. If the strategy is "ID", for *Iterative Deepening*, it loops starting from 0, in each iteration increments the maximum depth by 1, and calls the *generalSearch(String strategy, int max_depth)* method, till it reaches a solution. For all other search strategies, it just calls the generalSearch method with a max_depth = 10^7.

The *generalSearch(String strategy, int max_depth)* method proceeds as follow:

1. Initialize a SearchQueue q where it can keep track of the search-tree nodes.
2. Create a root node with the initial state and put it in the q.
3. While the q not empty loop:
4. Current ← remove first node in q
5. If current passes the goal test
6. Then Return a solution
7. Else children ← expand(current, strategy)
8. Foreach node in the children:
9. If it is not repeated in its branch and does not exceed the max_depth
10. Then add the node to q.
11. Return null if no solution is found.

Search Algorithms:

All search algorithms share the same general search procedure mentioned earlier, the difference is the queueing function. Therefore, in our implementation of the SearchQueue data structure, it uses one of the three built-in data structures: Queue, Stack, or PriorityQueue depending on the search strategy used in each problem. Three operations can be performed: add node, remove first node, and check whether the SearchQueue is empty.

BFS

We use a queue, where new nodes are enqueued at the end (FIFO).

DFS and ID

We use a stack, where new nodes are enqueued at the front (LIFO).

For all other search strategies, i.e., UC, Greedy, and A*, we use PriorityQueue datastructure, where the nodes are being sorted (according to a CompareTo(Node node) method in the Node class).

UC

The nodes are sorted by their path cost values, the nodes with less path cost at the front.

Greedy

The nodes are sorted by their heuristic values.

A*

The nodes are sorted by the sum of their path cost value + heuristic value.

SaveWesteros Problem:

A subclass from the SearchProblem class, that tackles SaveWestros search problem. It takes the initial state as input.

Westros State

A subclass from the abstract State class, which contains all the information that describes a state in the SaveWestros search problem.

A state is represented by a grid, where each cell has a value between 0 and 4. Where 0 represents an empty cell, 1 represents the cell where Jon stands, 2 denotes the dragonstone, 3 corresponds to an alive white walker, and 4 represents an obstacle.

The state also has the number of remaining dragonglass pieces, the number of alive white walkers, and the maximum number of dragonglasses allowed for Jon to hold.

Operators and Costs

It also initializes the list of operators, along with their costs in the constructor. We define 6 operators: move up, move down, move right, move left, kill all adjacent white walkers, and pickup dragonglasses. The costs of movement operators are set to 1, the cost of killing is set to 8, and the cost of picking up dragonglasses is set to 0.

The costs are designed this way taking into account that the optimality criteria is minimizing the number of dragonglasses used to kill all whitewalkers. In a case where the agent has to choose between using one dragonglass to kill a fewer number of whitewalkers, and making a few moves to use the same dragonglass to kill more whitewalker, it should choose to move to kill as many whitewalkers using one dragonglass as possible. Thus, we want to place a higher cost on the killing action.

An upper bound on the appropriate cost of killing should be the movement cost multiplied by the area of the grid O(N * M). That is in the case where the agent (theoretically) has to traverse the entire grid to reach a position in which it can kill more whitewalkers with the same dragonglass. This upper bound is never reached in a realistic scenario and using a cost function in the order of O(N*M) causes some instances of the problem to run for a very long time. Therefore, we make a reasonable assumption about the appropriate cost of the killing action and fix it to 8.

Expand a Node

The expand(Node node) method returns the list of new nodes resulting from expanding a node. It defines the preconditions and consequences of applying each operator, as we only add nodes if they represent valid states.

Starting from the state of the parent node, it loops over all of the 6 operators and creates a new node if the operator is applicable.

Cases 1 through 4: represent movements. Check the validity of the cell to which Jon is moving, it has to be inside the grid borders, and it is either empty or Dragonston, then update the grid accordingly if the movement is valid.

Case 5: Kill all adjacent white walkers if there is any and Jon holds at least one dragonglass) If applied, update the number of remaining white walkers and dragon glasses.

Case 6: Pickup dragon glass. Applicable only if Jon is standing at the cell of dragon stone, and not holding the maximum number of dragonglass pieces allowed for him. If applied, update the number of dragon glasses to be the maximum allowed for Jon.

If the node is to be added, i.e., the operator was valid, then add new node to the list of children. The parent of the new node is the node that we are currently expanding, the depth is equal to the parent's depth + 1, and the path cost is the parent's path cost + the cost of the operator used.

Note that we eliminate the repeated states in one branch. Before adding a new node, we check whether any of its ancestors have the same state, if this is the case, the node will be discarded.

Goal Test

The goal test checks whether all white walkers in the grid are killed or not.

The path cost of each node is explained previously, by adding the parent's path cost + the cost of the operator used to reach the child.

The heuristic functions will be discussed later.

Main Functions Description:

1. GenGrid(.):

   This function randomly generates a grid, which represents the initial configuration of the world. The grid is of size n x m, where n and m are either 4 or 5. The grid contains a random number of whitewalkers, ranging from 1 to 7 and a random number of obstacles from 1 to 3. The whitewalkers, obstacles, dragon stone are all placed at randomly generated locations. Jon Snow always starts at the bottom right corner.

2. Search(.):

   This function take as input the randomly generated grid from GenGrid(), a string representing the search strategy, and a boolean indicating whether or not we want to generate a visualization for the solution steps. It creates an initial WestrosState instance, and then instantiates an instance of the SaveWestros class, which is a subclass of SearchProblem. This instance then runs the search algorithm, and stores the result in a SearchProblem.Result instance. If visualize is true, it prints a visualization of the steps that the agent takes in order to reach a solution. Note that in this visualization, when Jon, denoted by the letter J, steps in a cell containing a dragonglass, denoted by the letter S, the cell remains visualized by a letter S.

Heuristic Functions:

Two admissible heuristic functions are defined to calculate the estimated cost to reach the goal from a node.

First admissible heuristic function is defined as follows:

h1(n) = ⌈W / 3⌉* 8, W = the number of remaining white walkers.

Second admissible heuristic function is defined as follows:

h2(n) = ⌈W/ 3⌉* 8 + (D - 1),
W = the number of remaining white walkers,
D = the Manhattan distance between Jon and the farthest alive white walker.

Argument of Admissibility

It is obvious that the second heuristic function dominates the first one, that is, *h2(n)* is always greater than or equal to *h1(n)*, hence it is enough to show the admissibility of *h2(n)*.

For each node, the goal can be reached by killing all the alive white walkers. The maximum number of white walkers that Jon can kill at once using one dragonglass is 3, since he can only kill white walkers in the 4 adjacent cells, however he can not be surrounded by 4 white walkers at the same time, therefore, in the best scenario case, he would be able to kill each 3 white walkers using one dragonglass. Hence, the cost of killing the remaining white walkers would be $\geq \lceil W/3 \rceil * C$, where W is the number of remaining white walkers and C is the cost of the operator "kill adjacent white walkers" which is fixed to be 8.

Additionally, in order to kill a white walker, the agent has to be in one of the adjacent cells, so we should consider the cost of moving to the adjacent cell of the farthest alive white walker, in the best scenario case this would be ≥ the Manhattan distance between Jon and the farthest alive white walker - 1. By adding the cost of killing white walkers and the cost of minimum required movement, we get that the heuristic function $h2(n) = \lceil W/3 \rceil * 8 + (D - 1) \leq$ the actual cost function.

Running Example 1:

Initial state:

grid = new byte [][] { {0, 0, 0, 4}, {0, 3, 0, 3}, {0, 0, 3, 0}, {0, 0, 2, 1}};

Which represents this grid:

- - - O
- W - W
- - W -
- - S J

max_dragonglasses = 3

BFS

STEPS:

Move left - Pick up dragon glasses - Kill adjacent Whitewalkers - Move up - Move up - Kill adjacent Whitewalkers

EVALUATION:

Cost = 19
Expanded nodes: 50
Depth = 6

DFS:

STEPS:
Move left - Pick up dragon glasses - Kill adjacent Whitewalkers - Pick up dragon glasses - Move right - Move up - Kill adjacent Whitewalkers - Move left - Move left - Kill adjacent Whitewalkers

EVALUATION:
Cost = 29
Expanded nodes: 10
Depth = 10

ID:

STEPS:
Move left - Pick up dragon glasses - Kill adjacent Whitewalkers - Move up - Move up - Kill adjacent Whitewalkers

EVALUATION:

Cost = 19
Expanded nodes: 108
Depth = 6

UC:

STEPS:
Move left - Pick up dragon glasses - Move left - Move up - Move left - Move up - Move up - Move right - Move right - Move down - Kill adjacent Whitewalkers

EVALUATION:
Cost = 17
Expanded nodes: 555
Depth = 11

GR1:

STEPS:
Move left - Pick up dragon glasses - Kill adjacent Whitewalkers - Move up - Move up - Kill adjacent Whitewalkers

EVALUATION:
Cost = 19
Expanded nodes: 31
Depth = 6

GR2:

STEPS:
Move left - Pick up dragon glasses - Kill adjacent Whitewalkers - Move up - Move up - Kill adjacent Whitewalkers

EVALUATION:
Cost = 19
Expanded nodes: 7
Depth = 6

A*1:

STEPS:

Move left - Pick up dragon glasses - Move left - Move left - Move up - Move up - Move up - Move right - Move right - Move down - Kill adjacent Whitewalkers

EVALUATION:
Cost = 17
Expanded nodes: 499
Depth = 11

A*2:

STEPS:
Move left - Pick up dragon glasses - Move left - Move left - Move up - Move up - Move up - Move right - Move right - Move down - Kill adjacent Whitewalkers

EVALUATION:
Cost = 17
Expanded nodes: 448
Depth = 11

Running Example 2:

Initial state:

grid = new byte [][] { {{0, 2, 0, 4, 3}, {0, 3, 0, 3, 0}, {3, 0, 0, 0, 0}, {0, 3, 4, 3, 1}};

Which represents this grid:

- S - O W

- W - W -

W - - - -

- W O W J

Max_dragonglasses = 2

BFS

STEPS:
Move up - Move left - Move left - Move up - Move up - Move left - Pick up dragon glasses - Kill adjacent Whitewalkers - Move down - Move down - Kill adjacent Whitewalkers - Move up - Move up - Pick up dragon glasses - Move down - Move down

- Move right - Move right - Kill adjacent Whitewalkers - Move up - Move right - Kill adjacent Whitewalkers

EVALUATION:
Cost = 48
Expanded nodes: 56807
Depth = 22

DFS:

STEPS:
Move up - Move left - Move left - Move up - Move up - Move left - Pick up dragon glasses - Kill adjacent Whitewalkers - Pick up dragon glasses - Move right - Move down - Kill adjacent Whitewalkers - Move right - Move right - Kill adjacent Whitewalkers - Move left - Move left - Move left - Move left - Move up - Move right - Pick up dragon glasses - Move right - Move down - Move right - Move right - Move down - Move left - Kill adjacent Whitewalkers - Move right - Move up - Move left - Move left - Move left - Move left - Kill adjacent Whitewalkers - Move right - Move right - Move up - Move left - Pick up dragon glasses - Move right - Move down - Move right - Move right - Move down - Move left - Move left - Move left - Kill adjacent Whitewalkers

EVALUATION:
Cost = 88
Expanded nodes: 109
Depth = 50

ID:

STEPS:
Move up - Move left - Move left - Move up - Move up - Move left - Pick up dragon glasses - Kill adjacent Whitewalkers - Move down - Move down - Kill adjacent Whitewalkers - Move up - Move up - Pick up dragon glasses - Move right - Move down - Move down - Move right - Kill adjacent Whitewalkers - Move right - Move up - Kill adjacent Whitewalkers

EVALUATION:
Cost = 48
Expanded nodes: 142319
Depth = 22

UC:

STEPS:
Move up - Move left - Move left - Move up - Move up - Move left - Pick up dragon glasses - Move right - Move down - Move down - Move left - Kill adjacent Whitewalkers - Move up - Move up - Pick up dragon glasses - Move down - Move right - Move down - Move right - Kill adjacent Whitewalkers - Move up - Move right - Kill adjacent Whitewalkers

EVALUATION:
Cost = 42
Expanded nodes: 821870
Depth = 23

GR1:

STEPS:
Move up - Move left - Move left - Move up - Move up - Move left - Pick up dragon glasses - Kill adjacent Whitewalkers - Pick up dragon glasses - Move down - Move down - Kill adjacent Whitewalkers - Move up - Move right - Kill adjacent Whitewalkers - Move up - Move left - Pick up dragon glasses - Move right - Move down - Move right - Move down - Kill adjacent Whitewalkers - Move right - Move up - Kill adjacent Whitewalkers

EVALUATION:
Cost = 58
Expanded nodes: 1380
Depth = 26

GR2:
STEPS:
Move up - Move left - Move left - Move up - Move up - Move left - Pick up dragon glasses - Kill adjacent Whitewalkers - Pick up dragon glasses - Move down - Move down - Kill adjacent Whitewalkers - Move up - Move right - Kill adjacent Whitewalkers - Move up - Move left - Pick up dragon glasses - Move down - Move right - Move right - Move down - Kill adjacent Whitewalkers - Move up - Move right - Kill adjacent Whitewalkers

EVALUATION:
Cost = 58
Expanded nodes: 1845
Depth = 26

A*1:

STEPS:
Move up - Move left - Move left - Move up - Move up - Move left - Pick up dragon glasses - Move right - Move down - Move down - Move left - Kill adjacent Whitewalkers - Move up - Move up - Pick up dragon glasses - Move down - Move right - Move down - Move right - Kill adjacent Whitewalkers - Move right - Move up - Kill adjacent Whitewalkers

EVALUATION:
Cost = 42
Expanded nodes: 344342
Depth = 23

A*2:

STEPS:
Move up - Move left - Move left - Move up - Move up - Move left - Pick up dragon glasses - Move right - Move down - Move down - Move left - Kill adjacent Whitewalkers - Move up - Move up - Pick up dragon glasses - Move down - Move right - Move down - Move right - Kill adjacent Whitewalkers - Move up - Move right - Kill adjacent Whitewalkers

EVALUATION:
Cost = 42
Expanded nodes: 79845
Depth = 23

Comparison of Search Strategies:

Example 1:

	BFS	DFS	ID	UC	Greedy1	Greedy2	A*1	A*2

Completeness	yes	no	yes	yes	no	no	yes	yes
Optimality	no	no	no	yes	no	no	yes	yes
No. expanded nodes	50	10	108	555	31	7	499	448
Cost	19	29	19	17	19	19	17	17
Depth	6	10	6	11	6	6	11	11

Example 2:

	BFS	DFS	ID	UC	Greedy1	Greedy2	A*1	A*2
Completeness	yes	no	yes	yes	no	no	yes	yes
Optimality	no	no	no	yes	no	no	yes	yes
No. expanded nodes	56807	109	142319	821870	1380	1845	344342	79845
Cost	48	88	48	42	58	58	42	42
Depth	22	50	22	23	26	26	23	23

Comments:

(1) Completeness

In general, only BFS, ID, UC, and A* are complete, given that the cost function evaluates to greater than or equal to 0, that is, the path cost of a node is less than or equal to the cost of its successors, which is the case in the SaveWestros problem. Additionally, the DFS strategy is

complete in our implementation, by eliminating the repeated states in the same branch, therefore the infinite branches are avoided and the algorithm would return a solution if there exists any. Furthermore, the greedy strategy in our case, with both heuristic functions, would always return a solution if one exists, for the same reason as DFS, that is, it will never get stuck in a loop.

(2) Optimality:

Only UC and A* search strategies always return optimal solutions.

(3) Number of expanded nodes:

The behaviour of the search strategies is similar in both examples in terms of the number of expanded nodes.

By comparing BFS, DFS, and ID, we find that DFS is expanding a relatively much fewer number of nodes, taking into consideration the elimination of repeated states in each branch. On the other hand, ID is expanding the most number of nodes among these three strategies, since it performs DFS repeatedly with incrementing the maximum depth. BFS falls in between as it expands all the nodes with depth less than (or probably equal to) the depth of the shallowest solution.

As for the search strategies that produce optimal solutions, namely UC and A*, it is always the case that A* expands less (or equal) number of nodes than the UC, regardless of the used heuristic function. Besides, it is remarkable that the A* strategy expands less number of nodes when using the second heuristic function instead of the first one, which is reasonable since *h2(n)* dominates *h1(n),* thus, every node expanded by *A*2* is also expanded by *A*1,* and in most cases, *A*1* expands more nodes.

How to run the program:

The code is implemented in Java. It is divided into two packages: main, and project1. The main package contains the classes used for general search problems, that is, Node, Operator, State, and SearchProblem. While the project1 package contains the classes related to SaveWestros problem.

To run the project, simply go to the Main.java class (in project1 package) and set the parameters of the Search method. The first parameter is the randomly generated grid, so it should probably

be left as is. The second parameter is a string denoting the search strategy and should be set to one of the following: [BFS, DFS, ID, UC, GR1, GR2, A*1, A*2]. The third parameter should be set to true if you want to see a visualization of the solution steps, one by one.