AI Project 1

Ahmed Hathout 34-9785 Omar Muhamed 34-15154
Zeyad Ezzat 37-11600

November 3, 2019

1 Description of the problem

There is an M * N grid. Iron Man (the agent) is in some cell (i, j) in that grid. There are 6 infinity stones that iron man should collect. To collect a stone, Iron man must be in the same cell as the infinity stone. collecting an infinity stone inflicts damage of 3 to Iron Man. There are K warriors placed in some known cells. Every warrior gives Iron Man a damage of 1 once in an adjacent cell where an adjacent cell is a cell that is either to the north, south, west or east to Iron Man. Iron Man can kill a warrior if they are in an adjacent cell and this causes 2 points of damage to Iron Man. Iron Man can not be in the same cell that a warrior is in. Thanos is also in some known location on the grid and once adjacent to Iron Man, he does 5 points of damage to Iron Man. Iron Man can not kill Thanos. Iron Man can be in the same cell with Thanos only if he collected all the infinity stones. Iron Man can then snap and if so the problem is solved.

2 Classes

2.1 SearchTreeNode

This class is implemented the same way it was stated in the lecture slides. A search tree node has 5 attributes; currentState, parentNode, lastAction, currentDepth and costFromRoot.

This class has getters, setters, equals and hascode methods. It also has a method called heuristics. This calculates a rough estimate of the cost from this node to the goal. To be admissible, it returns the minimum remaining damage that is to be taken. That is, the number of the remaining stones * the damage taken to collect the stone. plus the damage taken from Thanos before the snap action. There is also the method AS. It is for the A* algorithm and it returns the value from the method heuristics plus the cost of this node from the root.

2.2 GenericSearchProblem

This is also implemented as the lecture says. it has 2 attributes; actionSet, initialState.

It has 2 abstract methods (and the getters); expand which takes a node and applies some permissible actions on it to get new children nodes which contain different states and goalTest which takes a node and returns true if the state of this node is a goal state and false otherwise.

2.3 EndGame

It extends GenericSearchProblem and has the remaining attributes describing the problem that are neither in the GenericSearchProblem class nor State class. These attributes are the location of thanos and the length and the width of the grid.

The class implements the abstract methods expand and goalTest. expand() returns all the nodes that can be generated from the given node. It applies movement while handling what is a possible move and what is not and also handles collect, kill and snap. Of course it also handles the cost whether it is damage taken from enemies or from the stones. To calculate the damage taken, it calls another method which is computeTotalCost.

goalTest is very simple. It just returns the value of snapped attribute inside the State class.

ComputeTotalCost takes a node and assigns a value to the costFromRoot attribute in it. it does so by calculating the damage taken from near enemies and from the last action plus the damage taken previously to reach that node.

3 Main Functions

Most of the main functions were already discussed above. The 2 remaining ones are genericSearchProcedure and visualzeSolution. genericSearchProcedure is implemented using the pseudo code given in class. It handles the different search strategies and returns the goal node. It is given a search problem and a search strategy. The string output can then be generated from the goal node that is returned at the end and this done in the solve function.

visualizeSolution takes a goal node and prints all the actions followed by the grid resulted from that action.

4 Search Algorithms

The search alogrithms all use the same generic search function. They just differ in how they enqueue nodes. So to handle these different search alogrithms there is just a switch statement that checks the input strategy string and creates a corresponding priority queue. in breadth first it enqueues the nodes with least depth first and the other way arround in depth first and the remaining are the same as they were represented in class.

5 Heuristic function

The heuristic functions are fairly simple. they calculate a rough estimate of the cost from this node to the goal. To be admissible, they return the minimum remaining damage that is to be taken. That is, -for the first heuristic function- the number of the remaining stones * the damage taken to collect the stone. plus the damage taken from Thanos before the snap action. As for the second heuristic function, the damage taken from Thanos is not added. The 2 functions return 0 if the node is a goal node. For the A* algorithm it must be the case that it is admissible because AStarEvaluation() method returns the cost from the root to reach the current node plus the returned value from heuristic() that was just descriped above. It is admissible because the rough estimate is actually the damage that must be taken from the infinity stones and thanos before snapping.

6 Two running examples

There are 3 examples in the source code

7 Comparison

Regarding completeness in this specific problem, since we do not allow repeated states, all of the search strategies are complete since we do not have infinite number of states (no cycles).

As for optimality, UC and A* are optimal. The others are not. We can see that clearly when we run the algorithms. UC and A* give the same cost which is less than all the other strategies.

The number of expanded nodes is much higher in UC and A* (especially UC) more than the other algorithms since they do not take into consideration the number of nodes to expand but the cost of expanding the nodes. GR expands the lowest number of nodes because it only cares about how to reach the goal in the least amount of steps without caring about the cost of the path taken to reach that goal.