Media Engineering and Technology Faculty
German University in Cairo

Endgame - Artificial Stones

Introduction to Artificial Intelligence
Project I

Author: George Maged Maurice Gendi
Ahmed Ashraf AbdelKareem
Youssef Tarek Salah Eldin

Supervisors: Assoc. Prof. Haythem Ismail
Eng. Nourhan Ehab

Submission Date: 03 November, 2019

Contents

Chapter 1

Introduction

1.1 Problem

Given a grid of size MxN the goal is to navigate the map, collect stones and kill Thanos. This is a simplified version of the idea behind the movie Avengers: End Game, simulated as a searching AI agent system in the representation of the character iron-man. The map is divided into a cells of equal width and height, each cell can have one of the following (iron-man, Thanos, warriors and stones). The searching AI agent is to find a path through the map, collecting all six infinity stones, killing warriors if necessary and moving finally to the location where Thanos resides and performing a snap action. There are a number of constraints imposed on the agent to test the capability of finishing the task optimally. Constraints are expressed as the agent's health after performing an action, actions performed by the agent may result in it's health being reduced. Following the measures in table 1.1. Also, as certain actions that agent cannot perform table 1.2.

Situation	Cost
being in an adjacent cell to a warrior	-1/warrior
choosing to kill a warrior	-2/warrior
collecting a stone	-3
being in an adjacent, or same cell as Thanos	-5

Table 1.1: Health constraints imposed on iron-man

Situation
cannot move to same cell as a warrior
can be present at the same cell as Thanos or the stones
cannot move outside the boundaries of the map

Table 1.2: Action constraints imposed on iron-man

1.2 Example

This is an example grid with the corresponding formulated plan that solves that grid.

Input

5,5;1,2;3,1;0,2,1,1,2,1,2,2,4,0,4,1;0,3,3,0,3,2,3,4,4,3

		S	W	
	S	I		
	S	S		
W	T	W		W
S	S		W	

Output

up, collect, left, down, collect, down, collect, right, collect, kill, down, down, left, collect, left, collect, right, up, snap

Chapter 2

Implementation

The project is implemented using java 1.8, based on Russell and Norvig's Artificial Intelligence - A Modern Approach 3rd Edition [1]. Each class is responsible for a part of solving the problem, This section is dedicated to document each class.

2.1 `class` SearchTreeNode

It represents the nodes in the search tree, each node consists of attributes describing the current state, cost and depth of the node.

`State` state
The current state of the problem stored in `class` State.

`SearchTreeNode` parent
The parent node of the current node

`Operators` leadingOperator
The `enum` Operator that lead to the current node.

`int` cost
The path cost from the root to the current node instance.

`int` heuristicCost
The path heuristic function cost assigned to the node. (*default*= 0)

`int` depth
The current depth of the node.

`String` backTrack()
Backtrack to the root of the tree and formulate a plan using the leadingOperator variable from each node discovered during the backtracking process.

`ArrayList<SearchTreeNode>` pathFormRoot()
Same concept as backTrack() but returns the actual nodes along the path not just the plan.

`int` compareTo()
Our implementation of the search tree node extends the `class` Comparable. With a method responsible for comparing an instance of SearchTreeNode to another SearchTreeNode instance. based on the searching algorithm being used.

Algorithm	compareTo
Greedy	$heuristicCost_1 - heuristicCost_2$
A*	$(cost_1 + heuristicCost_1) - (cost_2 + heuristicCost_1)$
default	$cost_1 - cost_2$

Table 2.1: Comparable compareTo for SearchTreeNode

2.2 `interface` SearchProblem

This interface is a generic form of representing any search problem. It includes the following method definitions.

`State` initialState()
The method that will be responsible for returning the initial state of the search problem.

`SearchTreeNode` transitionFunction(`SearchTreeNode` node, `Operators` operator)
The method that will be responsible for taking in a node and an operator. Applying the operator on the node and returning a new node with the configured state.

`boolean` goalTest(State state)
The method that will be responsible for checking if a state passes the goal test of the search problem.

`int` pathCost(`SearchTreeNode` node, `Operators` operator)
The method that will be responsible for returning the new cost of the node based on the operator performed.

2.3 `class` EndGame

The EndGame class is the programmatic representation of the problem described in 1. It extends the `class` SearchProblem and overrides it's methods with the logic required to solve the problem.

`Pair<Integer, Integer>` gridSize
A pair representing the grid size, MxN as defined in 1.

`Pair<Integer, Integer>` thanosPosition
A pair representing the position of Thanos on the grid.

`State` initialState()
Parsing the input string and constructing several `class` attributes to describe the problem.

`SearchTreeNode` transitionFunction(`SearchTreeNode` node, `Operators` operator)
Transitioning through the map while enforcing that actions that cannot be performed are return as a `null` rather than a new SearchTreeNode instance.

`boolean` goalTest(State state)
The goal test being, collecting all stones and killing Thanos. Without having a comulated cost of more than 100.

`int` pathCost(`SearchTreeNode` node, `Operators` operator)
Path cost represented at the damage imposed on iron-man at a the given node.

2.4 `class` GeneralSearch

This is the class responsible for handling the different kinds of searching strategies, it has the following attributes and function and a list of implemented algorithms is provided.

List of Algorithms

Algorithm	Enum
Breath-First-Search	BF
Depth-First-Search	DF
Uniform-Cost-Search	UC
Iterative-Deepening-Search	ID
Greedy Search	GR1 & GR2
A*	AS1 & AS2

Table 2.2: List of algorithms implemented

`Queue<SearchTreeNode>` nodes
The queue instance where nodes are stored based on the expansion sequence or their cost. For algorithms that do not consider cost, the `Queue` is initialized to a `LinkedList` instance. And as a `PriorityQueue` for searching strategies that consider cost.

`HashMap<String, Boolean>` uniqueStates
This is used to store the states that are already explored, to avoid any repeated states and make the searching process terminate faster. Each `State` is serialized and used a key for the `HashMap` instance.

Expansion Note

A given `State` is checked during expansion if it is repeated using the `HashMap`. If the `State` is repeated, it is discarded.

Termination

When a given `SearchTreeNode` is found to be a **goal node**. The **backTrack()** method is invoked on the `SearchTreeNode` instance and a plan is returned. However, if there is no solution discovered a `null` value is returned.

2.5 Important Methods

Some methods that don't belong to one class but are important to note.

`String` solve(`String` grid, `SearchingAlgorithms` strategy, `boolean` visualize)
The method takes in a `String` grid representing the grid of the problem. A `SearchingAlgorithms` that chooses an algorithm to solve the problem. and a `Boolean` visualize that if set to true, results to a graphical representation of the solution.

2.6 Heuristic Functions

For the *Greedy* and $A*$ searching algorithms we implemented two different heuristic functions.

2.6.1 Heuristic 1

$$h_1(n) = count(s) * 3$$

where s is the **remaining stones**.
This is an **admissible** heuristic function, it will never overestimate the cost. This is due the fact that the cost of killing or passing by warriors and being in an adjacent cell as Thanos will dominate this function.

2.6.2 Heuristic 2

$$h_2(n) = \begin{cases} min(\{3 * \frac{distance(\mathbf{I}, s_i)}{diagonal}\}) & \text{if } count(s) > 0 \\ 0 & \text{if } count(s) = 0 \end{cases}$$

where I, s_i and s are **Iron Man's position**, **a stone in** s and **remaining stones** respectively.
This is an **admissible** heuristic function with two cases. The first case covers the situation were there are stones left to collect, In which case, we compute the distance between all stones and iron-man divided by the greatest diagonal that can fit in the grid multiplied by 3. This is admissible because the first case always bound the cost of closest stone between 0 and 3.

Chapter 3

Results

3.1 Example Inputs and Outputs

3.1.1 Example 1

Input

8,8;0,3;4,4;1,1,2,2,3,3,5,0,5,5,0,7;1,0,0,1,0,5,2,4,3,1,4,1,4,3

Comparison

Strategy	Cost	Nodes Expanded
BF	46	467795
DF	66	335976
UC	31	482913
ID	58	791579
GR1	47	150
GR2	72	101438
AS1	31	436316
AS2	31	482913

Output Using AS1

down, down, kill, right, right, right, right, up, up, collect, down, down, down, down, down, left, left, collect, down, left, left, up, kill, down, right, right, right, up, up, up, up, left, left, left, up, up, left, kill, down, right, down, right, right, down, right, down, down, right, down, down, left, left, left, left, left, left, left, up, up, collect, down, right, right, right, right, right, right, right, up, up, up, up, left, left, left, left, down, collect, up, left, collect, up, left, collect, up, right, right, down, down, right, down, down, snap;31;436316

3.1.2 Example 2

3.1.3 Input

15,15;12,13;5,7;7,0,9,14,14,8,5,8,8,9,8,4;6,6,4,3,10,2,7,4,3,11,10,0

Comparison

Strategy	Cost	Nodes Expanded
BF	38	736617
DF	82	27060
UC	30	776545
ID	38	760854
GR1	41	559
GR2	72	137351
AS1	30	453853
AS2	30	776545

3.1.4 Output Using AS1

up, right, up, up, up, up, up, up, left, left, left, left, left, up, up, left, left, left, left, left, up, left, up, left, left, left, down, down, down, down, down, down, collect, up, up, up, up, right, up, right, right, right, right, right, right, right, right, right, up, up, right, right, right, right, down, down, down, down, down, down, down, down, left, down, down, down, left, left, down, down, left, left, left, down, collect, up, up, up, right, right, right, right, right, right, up, up, left, left, left, left, left, left, left, up, left, left, left, collect, right, right, right, right, right, collect, up, up, right, right, right, right, right, down, down, down, collect, up, up, up, up, up, up, up, up, left, left, left, left, up, left, left, left, down, down, right, down, down, down, collect, left, snap;30;453853

3.2 Comments

Considering the examples mentioned above. **UC**, **AS1** and **AS2** produce the same cost, which is the optimal cost. However, **AS1** explores less nodes than **AS2** which explores less nodes than **UC**. Meaning, $h_1(n)$ dominates $h_2(n)$. **GR1** explores the least amount of nodes, but not with an optimal cost. The most expensive strategy is **DF** considering it outputs the first goal node it hits without considering any costs. **ID** costs falls between **BF** and **DF**.

Appendix

List of Tables

Bibliography

[1] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2009.