AI Project 2

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

November 21, 2019

1 How to Run

For some weird ****, the program does not terminate unless the KB that is generated by genGrid is copied to the very same file of the program called Endgame.pl. I tried to load it from the Endgame.pl file writing :-['KB1'] but it does not terminate!!!. I also tried to load the KB using ['KB1'] on the terminal but still I face the same problem so before running the program to test it please make sure that the knowledge base you want (1 or 2) are in the same file with the program Endgame.pl at the beginning. I already did that for the 2 KBs I generated.

This code uses iterative deepening so to run the program instead of writing snapped(S), use ids(snapped(S), Initial_Depth_Limit, Depth_Of_Final_Result). You can ignore the last argument and just write _ but for the depth limit start from a low number like 1 or 2 in order not to pass the depth of the first result.

Of course because this is iterative deepening, the program will not terminate if the input situation is wrong (unless of course the depth hits the limit for integer numbers in prolog which is gonna take forever unleeesss again if we set the new depth limit to be 2 * the previous one. In that case it will take a couple seconds to figure out that the situation is wrong).

The second KB takes toooooo long (approx. 20 30 mins) to finish much longer than the first one since it is 2 levels deeper but it takes a couple of seconds if given the right solution. The solution for the 2 KBs are written as comments in the code ;-).

2 Description of genGrid

The genGrid method is pretty simple. It takes the input with the same format that was presented in the project description. It then split it using ; then split creates the main predicates that are gonna be explained in the next section. genGrid also prints a visualisation of the grid.

3 Predicates

3.1 dimensions(I, J)

dimensions just specifies the length and width of the grid where I and J are the length and width respectively. Nothing complicated :-)

3.2 thanos_location(I, J)

Location of thanos on the grid.

3.3 stone(I, J)

this predicate is written 4 times; once for each stone. The I and J are the indices of the stone.

3.4 iron_man_location(I, J, Stones, S)

The I and J are the indices of Iron Man and the S is the current situation Iron Man is in. Stones is the list of stones that are yet to be collected. Every condition that makes the predicate true is well documented in the code itself.

3.5 snapped(S)

S is the situation which is either s0 or result(Action, Another_situation).

3.6 ids(snapped(S), Initial_Depth_Limit, Result

This is the main predicate to run to test the program. Initial_limit should be a low value just not to pass the shallowest goal. Result is the depth of the result and of course you can ignore it and replce it with _.

4 Actions

There are 6 actions; up, down, left, right, collect and snap.

5 Implementation of successor-state axioms

There is only 2 fluents which are iron_man_location() and snapped().

snapped is very straight forward. the conditions of it to be true is that Thanos and Iron Man must be on the same location and the Stones list in iron_man_location() must be empty.

As for iron_man_location(I, J, Stones, Situation). it handles movement and collecting stones. it recursively calls itself and changes the I and J if the action is a movement and changes the Stones list if the action is collect. Of course this is done in reverse order. That means if the action is move up then iron man was

in the previous situation in the above cell so the New_I of the previous situation is Old_I + 1. The same happens when the action is collect so we add to the list of stones instead of removing.