The German University in Cairo

Media Engineering and Technology

Endgame AI Agent

AI Project #2 Report

Team 4

Ahmed Mohamed Fawzy Ahmed Darwish	37-9080
Omar Mohamed Youssef Elkilany	37-0732
Nesrin Abdulaziz Alsayed Attia	37-16218

Table of Contents

Description of the Problem

For this project, the Endgame problem was simplified and represented using the situation calculus. Unlike Project #1, the problem definition does not have warriors, and the number of infinity stones to be collected is only 4. In addition, the maximum grid size is 5x5. The state during the changing scenarios is described using first-order logic formulae. There are predicates (fluents), actions, a knowledge base, and successor state axioms. The goal situation is for Iron Man to reach the cell where Thanos exists after collecting all stones.

A situation consists of an $M \times N$ grid with Iron Man in a position (x, y), Thanos in a static position $(ThanosX, ThanosY)$, and four or less stones, which were not picked up by Iron Man yet, in different locations. In all subsequent discussions, the x variable will refer to the row number while the y variable will refer to the column number. Together, they form a certain position in the grid. The s variable will refer to a situation.

The available actions for the agent include: *up, down, left,* and *right*, which change the location of the agent inside the boundaries of the grid with the expected semantics. *Up* moves the agent one row upward in the grid unless the agent is in the uppermost row. *Down* moves the agent one row downward in the grid unless it is in the lowermost row. *Left* moves the agent one column to the left in the grid unless it is in the leftmost column. *Right* moves the agent one column to the right in the grid unless it is in the rightmost column.

Movement to non-Thanos locations is always possible. Movement to the Thanos location is allowed only after collecting all the stones. The *collect* action causes the agent to collect the stone present in its own cell if such stone exists. Finally, the *snap* action is used to kill Thanos when all the stones are collected and Iron Man is in the Thanos cell. The *snap* operator has no effect on the search problem; therefore, it was not implemented as part of it. It is appended to the final solution, and it is assumed that Iron Man can only snap once.

The implemented predicates are:

1. $ironMan(x, y, s)$, which is true if ironman is in position x, y in situation s.
2. $stone(x, y, s)$, which is true if a stone exists in position x, y in situation s.
3. $thanos(x, y)$, which is true if the pair (x, y) equals the pair $(ThanosX, ThanosY)$, which describes the position of Thanos and is in the knowledge base.
4. $grid(x, y)$, which is true if the pair (x, y) equals the pair (M, N), which are the grid dimensions in the agent knowledge base.
5. $snapped(s)$, which is true if s is a goal state.

The Knowledge Base

Description of the Knowledge Base

The knowledge base is a set of predicates that the agent initially knows to be true. Some of them are situation-independant while others are only true in the starting situation (s_0). There are no variables in the initial knowledge base.

The knowledge base predicates are as follows:

- $grid(M, N)$ means the initial size of the grid is $M \times N$.
- $thanos(ThanosX, ThanosY)$ means that Thanos is in cell $(ThanosX, ThanosY)$.
- $ironMan(A, B, s_0)$ means in situation s_0 Iron Man is in cell (A,B).
- $stone(L, Z, s_0)$ means that in situation s_0, a stone exists in cell (L, Z). Initially, Iron Man knows that there are 4 stones in the grid in 4 positions.

Generation of the Knowledge Base

To generate the knowledge base for our agent, we use a Java method *GenGrid()*, which takes as input a string that follows the format mentioned in the project description. The method first delimits the string by semicolons and applies a *StringTokenizer* on the output, which divides the string at the locations of commas. A *StringBuilder* is used to build the final string that will be saved to the file. The method examines the tokens in order and adds sentences to the *StringBuilder* object. An end-of-line character ('\n') is added after each sentence for readability.

For the infinity stones, we keep looping through the remaining tokens until all stones are added to the knowledge base. Finally, a *PrintWriter* is used to create the file and output the string in the *StringBuilder* object.

Successor State Axioms

The successor state axioms describe how the world changes (or persists) in response to the actions of the agent. result(A, S) is the resulting state when the agent performs the action A in the situation S. In our problem, we define two successor state axioms:

1. `stone(X, Y, result(A, S))`
   It enumerates the possible ways a stone can exist in position (X, Y) as a result of an action A in a situation S. For a stone to exist in location (X, Y) in the situation result(A, S), it must also exist in position (X,Y) in situation S, and the action A must have no effect on the stone. Examining all the possible actions of the agent, we can arrive at the following conclusions:

   - If `A = right or A = up or A = left or A = down`, then stone(X, Y, S) must also be true. A movement does not affect the position of a stone; it only changes Iron Man's position.
   - If `A = collect`, then stone(X, Y, S) is true under a certain condition: Iron Man must be in a cell other than (X, Y) in situation S (collecting another stone). Therefore, ironMan(X, Y, S) must be **false**.

   It should be noted that this successor state axiom consists only of the the frame axiom because there is no action that reintroduces a stone into the grid. For example, Iron Man cannot drop a stone at a certain location.

Formally:

$$
\begin{aligned}
&\forall x \forall y \forall a \forall s[\\
&\text{stone}(x, y, \text{result}(a, s)) \rightarrow \\
&\quad [\\
&\quad\quad a = \text{Right} \vee a = \text{Up} \vee a = \text{Left} \vee a = \text{Down} \vee \\
&\quad\quad (a = \text{Collect} \wedge \neg \text{ironMan}(x, y, s)) \\
&\quad] \wedge \text{stone}(x, y, s)]
\end{aligned}
$$

2. `ironMan(X, Y, result(A, S))`
   It enumerates the possible ways Iron Man can be in position (X, Y) as a result of taking an action A in situation S. Examining all the possible actions of the agent, we can arrive at the following conclusions:

   - If `A = right`:
     Then ironMan(X, Y - 1, S) must have been true. Iron Man was in position (X, Y - 1) in situation S, and as a result of the right action, he moved one cell in the positive y direction. Furthermore, Y < N must be true because the current right movement should not get Iron Man outside the grid.
   - If `A = up`
     Then ironMan(X - 1, Y, S) must have been true. Iron man was in position (X-1, Y) in situation S, and as a result of the down action, he moved one cell in the negative x direction. Furthermore, X >= 0 must be true because the current up movement should not move Iron Man outside the grid.
   - If `A = left`
     Then ironMan(X, Y + 1, S) must have been true. Iron man was in position (X, Y + 1) in situation S, and as a result of the left action, he moved one cell in the negative y direction. Furthermore, Y >= 0 must be true because the current left movement should not move Iron Man outside the grid.
   - If `A = down`
     Then ironMan(X + 1, Y, S) must have been true. Iron man was in position (X+1, Y) in situation S, and as a result of the down action, he moved one cell in the positive x direction. Furthermore, X < M must be true because the current down movement should not move ironman out of the grid.

   ❖ **Note that:** If IronMan were to move to a cell (X, Y), one of the following conditions must hold:

   a. thanos(X, Y) must be false because IronMan must not move to a cell where Thanos is without all the stones.
   b. stone(K, L, S) must be unsatisfiable: There must be no assignment to (K, L) that can make Stone(K, L, S) true. If Iron Man collected all the stones,

Stone(K, L, S) will be unsatisfiable, and thus Iron Man will be able to move to any cell, including the one where thanos is.

- If `A = collect`

  Then ironMan(X, Y, S) must have been true. Iron man was in position (X, Y) in situation S, and stone(X, Y, S) must also be true because of our definition of the action *collect*. There must be a stone in situation S in the same position as Iron Man for successful collection.

Formally: (with the liberty of including addition and subtraction)

$$
\begin{aligned}
&\forall x \forall y \forall a \forall s[\\
&\quad \text{ironMan}(x, y, \text{result}(a, s)) \rightarrow \exists x1 \exists y1 \exists m \exists n(\\
&\qquad \text{ironMan}(x1, y1, s) \wedge \\
&\qquad \text{grid}(m,n) \wedge \\
&\qquad (\\
&\qquad\quad (a = \text{Collect} \wedge x1 = x \wedge y1 = y \wedge \text{stone}(x1, y1, s)) \vee \\
&\qquad\quad (\\
&\qquad\qquad [\\
&\qquad\qquad (a = \text{Right} \wedge x1 = x \wedge y = y1 + 1 \ \wedge y1 + 1 < n) \vee \\
&\qquad\qquad (a = \text{Down} \wedge x = x1 + 1 \wedge y1 = y \ \wedge x1 + 1 < m) \vee \\
&\qquad\qquad (a = \text{Left} \wedge x1 = x \wedge y = y1 - 1 \ \wedge y1 - 1 >= 0) \vee \\
&\qquad\qquad (a = \text{Up} \wedge x = x1 - 1 \wedge y1 = y \wedge x1 - 1 >=0) \\
&\qquad\qquad] \\
&\qquad\qquad \wedge (\neg \text{thanos}(x,y) \vee \nexists x2,y2(\text{stone}(x2, y2, s))) \\
&\qquad\quad) \\
&\qquad) \\
&)]
\end{aligned}
$$

Note that: The axiom was written in this way for prolog optimizations. ironMan(x1, y1, s) and grid(m,n) are common between the effect and the frame axioms. We defined the actions to be applicable only if they are valid. Therefore, if Iron Man was in position (X, Y) in state S and did an invalid action, such as *collect* in the absence of a stone, there will be no valid resulting state.

Initiating Queries using Snapped(S)

I. Description

Snapped(S) is a prolog predicate that is true iff Iron Man performs a *snap* at the Thanos location. Situation S is represented as a result of an action A in a previous situation. Through a series of actions starting from the initial situation s0, the agent can reach situation S.

II. Implementation

We describe the goal situation S. First, S = result(snap, Sp) means that situation S is the result of the action *snap* in the previous situation Sp. For Iron Man to snap Thanos, they must be in the same cell in situation Sp. Therefore, the following predicates must be true:

1. thanos(X, Y).
2. ironMan(X, Y, Sp).

Because of our definitions of the successor state axioms, Iron Man will not be able to enter the Thanos cell unless he has all four stones. Therefore, we do not need to explicitly say that there should not be any stones in Sp. Prolog will attempt to find a situation Sp that will make our predicate Snapped(S) true. It will explore all the possible options and yield a situation if there is any.

The predicate also works backward. Given a situation S, it can verify whether it is a goal situation. For example, if we call the predicate Snapped(S), S = result(left, result(right, s0)), if will return true only if S is a goal.

Example Runs of the Agent

Grid #1

For the first example, the following grid was used, which is the same as the grid in the project description:

```
+---+---+---+---+---+
|   |   |   |   |   |
+---+---+---+---+---+
|   | S | I |   |   |
+---+---+---+---+---+
|   | S | S |   |   |
+---+---+---+---+---+
|   |   | S | T |   |
+---+---+---+---+---+
|   |   |   |   |   |
+---+---+---+---+---+
```

Input: 5,5;1,2;3,4;1,1,2,1,2,2,3,3

Running *GenGrid* on the input, gives us the following knowledge base:

grid(5, 5).

ironMan(1, 2, s0).

thanos(3, 4).

stone(1, 1, s0).

stone(2, 1, s0).

stone(2, 2, s0).

stone(3, 3, s0).

And finally, we query the problem as follows:

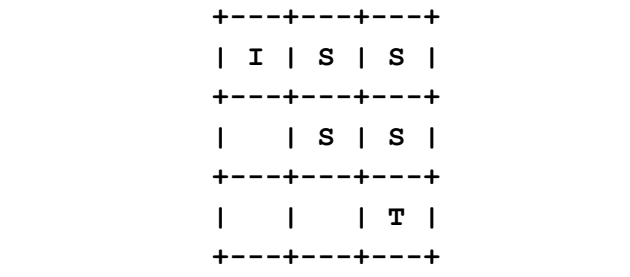
Query: snapped(S).
Result: S = result(snap, result(right, result(collect, result(down, result(right, result(collect, result(right, result(collect, result(down, result(collect, result(left, s0)))))))))))

Naturally, our implementation runs both ways, so if we were to pass a situation to the query, it will return true or false based on whether it is a goal.

Query: snapped(result(left, result(right, s0))).
Result: false.

Grid #2

The grid for the remaining of the queries is:




```
Input: 3,3;0,0;2,2;0,1,0,2,1,1,1,2
```

Running *GenGrid* on the input, gives us the following knowledge base:

grid(3, 3).

ironMan(0, 0, s0).

thanos(2, 2).

stone(0, 1, s0).

stone(0, 2, s0).

stone(1, 1, s0).

stone(1, 2, s0).

For the queries run on this grid, we will use the semicolon operator to ask Prolog for other possible situations.

Query: snapped(S).

Result: S = result(snap, result(down, result(collect, result(right, result(collect, result(down, result(left, result(collect, result(right, result(collect, result(right, s0)))))))))) ;

S = result(snap, result(down, result(right, result(collect, result(left, result(collect, result(down, result(collect, result(right, result(collect, result(right, s0)))))))))) ;

S = result(snap, result(down, result(right, result(collect, result(left, result(collect, result(down, result(collect, result(right, result(collect, result(right, s0)))))))))) ;

S = result(snap, result(right, result(down, result(collect, result(left, result(collect, result(down, result(collect, result(right, result(collect, result(right, s0)))))))))).

Query: snapped(result(snap, result(down, result(collect, result(right, result(collect, result(down, result(left, result(right, result(left, result(collect, result(right, result(collect, result(right, s0)))))))))))))).

Result: true.

General Observations about Runtime

Our implementation is capable of solving grids; however, it is slow because we had limited control over how the search in Prolog proceeds. We did not use IDS because it slowed our implementation down. The or statements in the ironMan successor state axiom are a good replacement for IDS because of how prolog's backtracking works.

We have experimented with the code and arrived at a new approach that can produce solutions in only a few seconds for grids up to 8x8. The main assumption behind this approach is that the problem can be divided into several subproblems, which can then be solved individually. By assuming that the stone pickup should be ordered, Iron Man can proceed to find situation S1, where the first stone is collected; S2, where the second stone is collected; S3, where the third is collected; and S4, where the fourth is collected. Finally, S5, where Thanos and Iron Man are in the same cell, is found.

Building the solution this way allows us to instantiate S2 with the S1 situation (and similarly for subsequent situations), which saves runtime. Furthermore, because Iron Man seeks the stones, the solution is much faster.

On a 5x5 grid with stones in the corner cells and both Thanos and Iron Man in the middle row, the implementation presented in this report did not terminate after 3 hours of computations. On the other hand, the solution with the assumptions produced an answer in less than 5 seconds.

This approach, however, is not guaranteed to find all possible solutions. It finds only a subset. Furthermore, it does not work backward and cannot be used to verify whether a solution is valid.

References and Citations

1. Prolog Documentation: https://www.swi-prolog.org/pldoc/index.html.
2. Java 7 Documentation: https://docs.oracle.com/javase/7/docs/api/.
3. Introduction to Artificial Intelligence, Winter 2019 edition, German University in Cairo. http://met.guc.edu.eg/Courses/CourseEdition.aspx?crsEdId=951.