**Members:**

- **Michael Khalil 37-3063 /T15**
- **Basem Rizk 37-14415/T14**
- **Moustafa Nawar 37-10858/T17**

**(TEAM 74)**

# >Logic Based Agent AI Report<

## Implementation Discussion

- ### GenGrid java method

The GenGrid takes a string as input a string of the grid description and splits it to gain data about the location of Thanos, the location of Ironman, and the location of the four stones. It then writes the following data into a file named “KB{i}.pl” (where i is a static number that increments with each calling of GenGrid ):

```
tAt(X,Y).
sAt(ID,X,Y,s0).    // Written 4 times through a for-loop, one for each stone.
iAt(X,Y,s0).
```

X and Y are the height and width position respectively. The “s0” symbol denotes the initial state. ID is an ID for each stone that starts with 1 and increments with each loop cycle till it reaches 4. That way each stone has a unique ID from 1 to 4.

- ### Syntax and semantics of action terms and predicates

  - **gridSize(H, W)**

Defines the grid of the allowed space where ironman can traverse with: **H** refers to maximum point in rows, **W** refers to maximum point in columns, and **0 is the minimum** in either case.

  - **tAt(X, Y)**

Defines the position of thanos, **X** refers to which row, and **Y** refers to which column.

  - **sAt(ID, X, Y, s0)**

Defines an existence of an infinity stone at the very beginning: **X and Y** refers to the position of the stone, ,**ID** refers to just an identification number, and **s0** refers to the initial situation, when no action has been performed.

  - **iAt(X,Y, s0)**

Defines where ironman exists at the initial situation **s0** before any attempts to solve the problem, **X, Y** defines his position.

- ## Successor-state axioms

  - **iAt(X,Y,S)**

Describes the restrictions and the logic behind every action including **{collect, left, right, down, up}** in the case of any traversing in the grid space at any situation **S**, ahead of the planned action.

In our implementation, this SSA is represented as follows:

Ironman is at position (**X**,**Y**) in state **S "iAt(X,Y,S)"**, if and only if:

He was already there, and he performed the collect action;
Or, he was in a neighbor cell in the previous and he performed a move action that got him to the current cell.

In our implementation, we omit any restrictions on the collect action since these restrictions are already taken care of by the SSA of **stoneExists(ID, X, Y, COLLECTED, S)**. Also, a persistence rule is not needed since ironman cannot be at the same position in the next state except when he performs a collect action.
And because no other action can be performed while ironman is in a position. So the collect rule is the only rule needed for persistence. The restriction of the movement actions are implemented while performing an action.

  - **stoneExists(ID, X, Y, COLLECTED, S)**

Here **ID, X, Y** define a stone the same way the predicate **sAt(ID,X,Y,s0)** define them, **COLLECTED** is 0 at situation **s0**, or some other **S**, indicating that stone has not been collected yet, hence it is collectible at the point;
<u>Or,</u> it is **1**, if and only if, at situation **S**, an infinity stone with **ID, X, Y** have been collected, hence does not exist anymore, and can not be collected.

In our implementation, **stoneExists(ID, X, Y, COLLECTED, S)** predicate is implemented in the following way:
A collected stone **"stoneExists(ID, X, Y, 1, S)"** exists in situation **S**, if and only if:
The same stone was uncollected in the previous situation **S0:** "**stoneExists(ID, X, Y, 0, S0)",** and the last action performed was the collect action; OR,
The same stone was already collected in the previous situation.

But, in our implementation, an uncollected stone SSA is omitted since we do not care if the uncollected stone exists or not in each situation. We only care if it is still uncollected in general. So instead of

**“stoneExists(ID, X, Y, 0, S0)”**, we replace the previous situation **S0** with the initial situation **s0**. So the SSA for A collected stone above **“stoneExists(ID, X, Y, COLLECTED, S)”** becomes the following with difference underlined:

A collected stone **“stoneExists(ID, X, Y, 1, S)”** exists in situation **S**, if and only if:
The same stone was uncollected <u>in the initial situation **s0: “stoneExists(ID, X, Y, 0, s0)”**</u> and the last action performed was the collect action; OR,
The same stone was already collected in the previous situation.

## ● snapped(S) predicate

**S** is basically the plan, or in other words, the solution of the problem, and the predicate is satisfied only if the solution meets the description of a successful plan of the problem (defined in **snapped1(S)** predicate) on branching depth limit of **50** (defined in **generateLimit(L)** predicate), with favor for the short branches; meaning **an iterative deepening mechanism.**

**snapped1(S)** predicate sets the condition of a plan to be successful;

- The restriction that the plan ends with a situation **result(snap, S0)**, where the **last action performed is snap**. following some other situation to be unified with **S0**.

- The restriction that ironman, and thanos are at the same position at the **S0.**Thanos position is fixed (defined by **tAt(X,Y)**, where X, Y here are already defined values in the knowledge base of the agent); however, ironman have is restricted and defined by **iAt(X,Y,S0)**, where **S0** is the situation before snapping, and have lots of possibilities.

- The restriction that **the 4 stones have been collected before snapping** defined by **{ sAt(1,_,_ ,1, S0)**, **sAt(2,_,_1,S0)**, **sAt(3,_,_1,S0)**, **sAt(4,_,_,1,S0) }**, referring to the collection of the 4 stones.

## Running Examples (2 Ex.)

```
GRID 1 : "5,5;1,2;3,4;1,1,2,1,2,2,3,3"
```

```
    0   1   2   3   4
0 |   |   |   |   |   |
1 |   | s | i |   |   |
2 |   | s | s |   |   |
3 |   |   |   | s | t |
4 |   |   |   |   |   |
```

**PLAN** **Around 15 seconds.**

result(**snap,**result(**right,**result(**collect,**result(**right,**result(**down,**resu lt(**collect,**result(**right,**result(**collect,**result(**down,**result(**collect,**resu lt(**left,**s0)))))))))))

```
GRID 2 : "5,5;2,2;4,2;4,0,1,2,3,0,2,1"
```

```
    0   1   2   3   4
0 |   |   |   |   |   |
1 |   |   | s |   |   |
2 |   | s | i |   |   |
3 | s |   |   |   |   |
4 | s |   | t |   |   |
```

**PLAN** **Around 8 minutes.**

result(**snap,**result(**right,**result(**right,**result(**collect,**result(**down,**resu lt(**collect,**result(**left,**result(**down,**result(**collect,**result(**left,**result( **down,**result(**collect,**result(**up,**s0)))))))))))))

## References

1. SWI-PROLOG Online documentation. (call_with_depth_limit)