Artificial Intelligence applied on the Risk! game

Luca Fuligni (luca.fuligni@studio.unibo.it), Andrea Franzon (andrea.franzon@studio.unibo.it), Orfeo Ciano (orfeo.ciano@studio.unibo.it)

Alma Mater Studiorum - Università di Bologna

Abstract. Our goal is to create an application that uses artificial players for the game of Risk!. Some of the existing implementations of Risk! present customizable artificial players but their behavior seems to be neither too smart nor naive and this can tire out the human player. We intent to achieve the goal in a different way: we chose to use Prolog, a declarative language, in contrast with the existing open-source implementations. We learnt that the usage of Prolog reduces the development time and efforts by offering a good abstraction.

Keywords:artificial intelligence, Prolog, Risk, game, dices.

1 Introduction

Our goal is to create an application that uses artificial players for the game of Risk! The context is the usage of the Artificial Intelligences on this game that is a strategy, turn-based, multiplayer game and contains some randomness due to the throwing of dices. More exactly we use A.I. for the game strategy. We choose risk! because unlike games in the theory's games it presents many problematic for example the choice of the attacking territories or the prediction of which territory the enemy will attack.

Usually the existing open-source implementations of this game uses imperative languages to describe the A.I. strategy. We, instead, propose to separate the A.I. aspect from the program by using a declarative logic language. We choose Prolog because it includes the first-order logic and backward chaining mechanism natively. Moreover we don't choose an imperative language, like Java, for the A.I. aspect because all the tree search and inference functionality included in some libraries are less efficient than the Prolog engine.

As a declarative language we choose Prolog instead of other languages like Lisp because we have to infer more on records than on lists. Our intent is to use the Prolog language for describing the artificial player's decisions.

The ability of each artificial player (and so the difficulty for a human player) is characterized by the set of rules that define the inference engine of each artificial player. The particularity of our solution is the connection between prolog that represents the A.I. and Java with which we develop the rest of our application.

2 The game of Risk!

Risk! is a turn-based strategic board game for two to six players in which the goal is indicated by an objective card. The game is played on a board depicting a political map of the Earth that contains 42 territories distributed in 6 continents.

There are several goals but in our application, for simplicity and for a general purpose, we consider only the aim to own 24 territories.

At the start-up each player distributes a fixed number of armies on the board. In each turn there are three phases: placing armies, attacking, fortifying. In the first phase the player puts new armies into the owned territories. In the second the player attack a nearby enemy territory, the success or fail of an attack is decided by the throw of dices. In the third phase the player can move some armies between two owned nearby territories only once. The phases will repeat themselves until a player reaches his goal.

3 Related works

This approach can be used in games which present randomness and multiplayer issues or in games where the artificial player's behavior is strictly connected with logical description like Risk!.

Nowadays there are several implementations of Risk!. The most famous is Risk Digital of Hasbro but there are a lot of Risk! clones like Lux for Linux, Dominion for mobile phones or other flash games can be found over the Internet. Some of those implementations present customizable artificial players but their behavior seems to be neither too smart nor naive and this can tire out the human player.

Our approach, besides focusing on efficiency, challenges the logical aspect in a different and more suitable way in order to improve the game-play.

4 Methodology

Our first implementation is a Java model apt to represent the game board and all the informations that it contains (territory borders, neighborhood relations, continents, territory ownership, ...). We also define the Prolog knowledge-base structure paying attention on data consistency with the Java model.

We propose three different A.I. difficulty levels: easy, medium and hard. The easy level performs random actions. Instead the hard level will be an extension of the medium level that is an "intelligent" implementation of the same Prolog predicates. Each skill that distinguish medium and hard difficulty is mapped into Prolog predicates. We use "GNU Prolog for Java" to embed the Prolog interpreter into our Java application because it respects the ISO standards.

We assume that the aim is to conquer 24 territories to simplify the study of the game. We choose to guide the A.I. through some well-known cases in each phase of the player's turn.

4.1 Knowledge Base

The Risk! Map can be considered as a set of countries (territories). For each territory we define:

- a relation of neighborhood between territories;
- the territory's ownership;
- the number of armies a territory holds;

We represented this knowledge in Prolog by defining the following set of facts:

player/1 represents a player, identified by his color (blue, red, green, pink, black, yellow).
territory/1 represents a territory, identified by his name.
owner/2 represents the ownership of a territory identified by the denoted player.
neighbor/2 represents the relation of the neighborhood between two territories.
army/2 represents the army number for each territory.

So this is the simplest knowledge base we can have:

```
player(red).
player(green).

territory(siberia).
territory(jacuzia).
territory(cita).

owner(siberia,red).
owner(jacuzia, red).
owner(cita,green).

neighbor(siberia,jacuzia).
neighbor(siberia,cita).
neighbor(jacuzia,cita).
neighbor(jacuzia,siberia).
neighbor(cita,jacuzia).
neighbor(cita,siberia).

army(siberia,3).
army(jacuzia,4).
army(cita,2).
```

4.2 Moves

Since a turn is made up of three phases, we define three separated predicates:

place_army/2 Given the player's color, it gives back the territory on which an army should be placed.

attack/3 Given a player's color,it gives back two territories: an attack source (territory owned by the player), and an attack destination (territory owned by an enemy player).

move/3 Given a player's color,it gives back two territories: the territory-source from which the armies should be moved, and the territory-destination to which the armies should be moved.

Here are some sample queries:

```
?- place_army(red,Destination).
Yes, Destination=siberia.
```

```
?- attack(red,Source,Destination).
Yes, Source=yacuzia, Destination=cita.
```

```
?- move(red,Source,Destination).
Yes, Source=siberia,Destination=jacuzia.
```

The implementation of those predicates will vary according to A.I. difficulty. In this way the responsibility of the Java engine is to update the knowledge base and perform the move suggested by the prolog engine.

4.3 Randomness representation in Prolog

Because of randomness component of the game, exploring a decisional tree to choose which territory to attack, could lead to a general low-responsivity. This is because we'd have to insert two randomness level (attack and defense dices) into the decision tree increasing exponentially the number of the nodes in that level.

Therefore we decide to represent the randomness components using a table that contains, given the numer of attacker and defender army, the probability to win an attack. Inside the Knowledge Base in Prolog we represent this information with *victory/3*:

```
victory(Attack#Army, Defense#Army, Probability).
```

So that the predicate "attack",given a fixed threshold value, knew the configuration attacker army/defender army that two territories must have to perform an attack.

4.4 From Java to Prolog

Java has the responsibility of generating the knowledge base. In order to accomplish this task we use the Visitor pattern on the game table (map) object.

```
1  PrologVisitor visitor = new PrologVisitor();
2  for(Continent continent:continents)
3  {
4      for(Territory territory:continent.getTerritories())
5          visitor.visit(territory);
6  }
```

Once the knowledge base is updated, it is loaded by the Prolog engine:

```
1  Environment env = new Environment();
2  //Loading the knowledge base file and A.I. implementation
3  env.ensureLoaded(AtomTerm.get("knowledge.pl"));
4  env.ensureLoaded(AtomTerm.get("easy.pl"));
5  interpreter = env.createInterpreter();
6  env.runInitialization(interpreter);
```

The execution of a query:

```
1  //Calling the query: ?- place_army(red,Territory).
2  VariableTerm answerTerm = new VariableTerm("Territory");
3  Term[] args = {AtomTerm.get(playerColor),answerTerm};
4  CompoundTerm goalTerm = new CompoundTerm(AtomTerm.get("
       place_army"), args);
5
6  //Executing the query and testing if succeded
7  int rc = interpreter.runOnce(goalTerm);
8  if(rc==PrologCode.SUCCESS || rc==PrologCode.SUCCESS_LAST)
9  {
10
11     //Getting the Prolog indicated territory from the
           game table
12     Term value = answerTerm.dereference();
13     String name = TermWriter.toString(value);
14     Territory destination = table.getTerritory(name);
15 }
```

The result of this query will be used by the Risk game engine that performs the corresponding move.

5 Case-study

During an attack, to handle the randomness of the dices, we generate a table (based on Markov chains) which contains the probability of winning an attack by the number of attack and defense armies.

Each player's difficulty is characterized by a threshold value, only if the probability of winning is above this value, the player will attack. Due to this decision

sometimes the A.I. reaches a deadlock situation where no one either attacks or moves armies between the territories. To break the deadlock we need to hardcode some specific Prolog conditions.

After thit we again tested the system by letting two players, with the same A.I. difficulty level, fight each-other. Otherwise, if there is at least one player with a different difficulty level, then the specific conditions are not so evident, so the Turing test is passed.

To overcome these problems a finer solution could be to represent the model as a Constraint Linear Programming problem in which every decision is driven by an objective function that will either be minimized or maximized, depending on the action that the player has performed. The objective function varies according to the evaluation of different variables. This evaluation is either rough or refined depending on the player's difficulty level.

6 Conclusions

Our implementation grants a good level of playing for every kind of player. Otherwise the CPL solution would require a fine tuning of the objective function for each difficulty in order to meet the optimal value for each A.I. level. This will produce an unnatural playing style. The Prolog interpreter is an optimal solution for these kinds of problems. By using Prolog it is easier to switch between different A.I. logics (and approaches) leaving the core application unchanged.

References

1. Osborne, Jason A. April 2003: "Markov Chains for the RISK Board Game Revisited", Mathematics Magazine 76
2. S. Russell e P. Norvig, 2005: "Intelligenza artificiale. Un approccio moderno" volume 1 Seconda Edizione.
3. L.Console, E.Lamma, P.Mello, M.Milano, 1997: "Programmazione Logica e Pro-log" Seconda Edizione UTET.