Open-World Adventure Game

MSAI 371 Knowledge Representation and Reasoning Project Report

Qingwei Lan, Liqian Ma, Wentao Yao

March 13, 2022

1 Introduction

We built a maze game with a knowledge base and inference engine called "open-world adventure", where a role-played hero can navigate, explore, and fight. The goal of this project is to make use of Prolog [1] and its inference engine to create a knowledge base with our own rules.

We built this game because our team members are fans of adventure games (e.g. Dungeons and Dragons). Our motivation and goal is to build a self-sustaining world that has multiple complex systems interacting with each other by encoding knowledge and building rules.

This world is a rule-based world with complex systems such as weather, monster fighting, exploration, day-night cycles, etc. The hero travels through the world by choosing its next move and the world's reasoning engine will determine whether the move is a valid one or not. The reasoning engine will also use existing rules and facts to determine events that may happen after the hero makes the move. For example, if the hero enters a location with thunderstorms, the hero will lose some health and stamina.

2 Game Details

In this section, we will explain the details of the game and all the systems we built. We will also explain how we encoded knowledge and how we utilized the reasoning engine to perform tasks with the encoded knowledge.

2.1 Map System

We have a map with a predefined size like 10×10. At each coordinate, we have the following objects as shown in Table 1.

Object	Visual	Explanation
empty	`0`	walkable spot
wall	`1`	not walkable
start	`2`	hero starts adventure at this location
gem	`3`	if found, the game ends and hero wins
rock	`4`	initially not walkable, but can be broken
peril	`-M`	a negative number indicates a peril (monster)

Table 1: Table of map objects, their visual representations, and explanations.

Visual Representation of Map

The map can be represented visually, shown below.

```
[
    [2, 0, -5, 0, 1, 0, 0, 0, 0, 0],
    [1, 0,  0, 3, 1, 0, 1, 0, 0, 1],
    [0, 1,  0, 0, 1, 0, 1, 1, 0, 1],
    [0, 1,  0, 0, 1, 0, 1, 0, 0, 1],
    [0, 1,  0, 0, 1, 0, 1, 0, 1, 0],
    [0, 1,  0, 0, 1, 0, 1, 0, 1, 0],
    [0, 0,  1, 0, 1, 0, 1, 0, 0, 0],
    [0, 0,  0, 0, 1, 0, 0, 0, 0, 0],
    [0, 0,  0, 0, 1, 0, 0, 0, 0, 1],
    [0, 0,  0, 0, 0, 0, 1, 0, 0, 1]
]
```

We built a system to automatically infer the map objects based on this visual representation. This system is one of the most complicated reasoning systems in our project.

We need to process each cell (row and column) to extract the object from the visual representation and insert the object as a fact into our knowledge base. Some objects (walls) cannot be changed and need to be represented statically. Other objects (Gem) can be removed from the map and need to be represented dynamically.

Knowledge Encoding & Reasoning

The map consists of facts inserted into the knowledge base. Some examples are shown below.

```
heropos(C, R).  % set the hero's position
gem(C, R).      % set the gem's position
rock(C, R).     % set the position of a rock
peril(C, R, M). % set the position of a peril
```

The objects at each location are inferred from the visual representation of the map. We built rules to infer and assert the facts into the knowledge base by interpreting the visual representations of objects as shown in Table 1.

2.2 Move System

The game is implemented as a turn-based game. The hero can choose its move during each turn. To move around the map, the hero can choose to move right, left, up, or down. Each move is validated by (1) checking whether the destination is within bounds of the map, (2) checking whether the destination cell is moveable (doesn't contain a wall or rock).

During each valid move, we retract the hero's current location (fact) and assert a new fact with the hero's new location. This is how we represent a move in our knowledge base.

Knowledge Encoding & Reasoning

The hero's position is a fact asserted into the knowledge base. A move involves retracting the hero's current position and asserting the hero's new position.

```
retract( heropos(CX, CY) ).  % retract hero's current position
assert( heropos(X, Y) ).     % assert hero's new position
```

Each move is validated through basic reasoning rules, such as whether the move is within bounds or whether the new location contains a wall or rock. For example, listed below is a basic rule to determine whether the move is within bounds.

```
in_bounds(C, R) :-
    mapsize(MC, MR),
    C #=< MC, C #> 0,
    R #=< MR, R #> 0.
```

2.3 Combat System

The combat system defines how the hero fights a monster. The damage dealt by the hero and the monster is dependent on the hero and monster's attack values and the results of rolling a dice. Again, we introduced randomness (rolling a dice) into the system to make the game more fun.

Knowledge Encoding & Reasoning

The abilities of the monster and hero are encoded as facts in the KB. For example

```
monster_health(100).  % encodes monster's health
hero_health(100).     % encodes hero's health
hero_stamina(100).    % encodes hero's stamina
hero_attack(20).      % encodes hero's attack ability
```

The combat process is defined by rules that depend on the abilities of the monster, the abilities of the hero, and the outcome of rolling a dice. A typically battle is represented below.

```
hero_attack(A),     % gets the attack ability of the hero
roll(D1),           % roll dice
roll(D2),           % roll dice
HH #= H1 + D2 * M,  % calculate impact on hero health
MH #= H2 - D1 * A,  % calculate impact on monster health

% modify hero's health
retract( hero_health(H1) ),
assert( hero_health(HH) ),

% modify monster's health
retract( monster_health(H2) ),
assert( monster_health(MH) ),
```

2.4 Day-Night Cycle System

The world holds a clock that counts time. Each move made by the hero takes 1 "tick" and a single day consists of 24 "ticks". The first 12 "ticks" are daytime and the following 12 "ticks" are nighttime. This system would have an impact on the hero's abilities and the abilities of monsters in the world. The latter part would be implemented as a follow-up because it is too complicated.

Knowledge Encoding & Reasoning

The global time is kept through a global counter. This counter is asserted into the KB as a fact. The time is updated on each clock cycle as follows.

```
time(T),             % get current time
NT #= T + 1,         % increment time by 1 tick
retract( time(T) ),  % retract current time
assert( time(NT) ).  % assert new time
```

The day/night cycle is inferred from the global time counter through simple rules.

```
is_day()   :- time(T), T mod 24 #< 12.
is_night() :- time(T), T mod 24 #>= 12, T mod 24 #< 24.
```

2.5 Weather System

There are 4 types of weather conditions.

- Clear: the weather is clear and nothing will happen; this is the default.
- Rainy: the hero will lose 1 stamina point for each clock cycle standing on the cell with rainy weather.
- Thunder: bad weather; the hero will lose 1 stamina point and 1 health point for each clock cycle standing on the cell with thunder weather.
- Foggy: the hero cannot see anything on a cell with foggy weather, so if the cell has a monster, the hero won't see it and may risk walking right into it, causing an unnecessary fight.

During each clock tick, the world will randomly modify the weather for 25% of the cells in the map. Randomness is included to ensure the game contains a sense of uncertainty.

Knowledge Encoding & Reasoning

The weather for each cell is a fact (knowledge) that is asserted into the knowledge base.

```
weather(C, R, rainy).
weather(C, R, thunder).
weather(C, R, foggy).
```

We built rules that implement the events listed above for each weather condition. For example: the hero cannot see anything on the cell with foggy weather, and if the hero tries to inspect the cell, it will print "Foggy weather, cannot see". This is accomplished through a rule that checks the weather for a cell location to see if the weather there is foggy.

```
show_cell_info(C, R) :-
    weather(C, R, foggy),        % if the weather is foggy
    write('Foggy, cannot see'),  % print information
    !.                           % stop immediately
```

3 Strengths

Programming Language

Choosing Prolog [1] as the programming language is our wisest choice. It was easy to work with and was open-sourced, so it had good community support. It also had a knowledge base and inference engine, making it easy for us to encode knowledge and derive knowledge. It also provided good performance, so we were able to encode a lot of knowledge and rules.

Encoding knowledge was easy. All we needed to do was call `assert()` and `retract()`.

```
retract( heropos(CX, CY) ).  % retract hero's current position
assert( heropos(X, Y) ).     % assert hero's new position
```

System Interactions

We chose to build our game world using dynamic / static facts and rules, which made it easy for us build the world. Furthermore, we can see how our different systems (rules) interacted with each other, which introduced conditions that we did not think of and provided interesting results.

For example, foggy weather blocked vision and does not allow the hero to discover what objects a cell contains. This is an example of the interaction between the weather and discovery systems.

Randomization

Our randomization systems, such as the weather system, introduced uncertainty into our game and provides not only fun for our players but also provides good case studies on how randomization in different systems can interact with each other. The main randomization system is our weather system. During each clock cycle we randomly change the weather of 25% of the cells.

```
change_weather(C, R) :-    % randomly change weather at cell (C, R)
    random_weather(W),     % randomly generate weather condition
    clear_weather(C, R),   % retract previous weather condition
    set_weather(C, R, W).  % set the new weather
```

4 Weaknesses

User Input Processing

Prolog is a good for encoding knowledge and performing reasoning. However, it seems a bit awkward for processing user input since it is a logic programming language.

For example, typing the following commands to move around is tedious.

```
uu().    % move up
dd().    % move down
rr().    % move right
ll().    % move left
stay().  % stay put
```

Instead, it would be easier if we could just use the arrows on our keyboard to move around.

We believe that it would be easier for the player if we used a general purpose language like Python for processing user input and feeding it into the Prolog system.

Better Way to Show Information

Right now, all our gameplay information is communicated through plain text and may be easily missed.

```
Time: 1
Hero Health:     100
Hero Stamina:    100

--------------------------
Surroundings:

    HERO:
    rr():
    ll():
    uu():
    dd():   Foggy, cannot see
--------------------------
```

We can probably improve on this in the future to make it easier for the player to read.

5 Conclusion & Future Work

In this section, we introduce some systems that could greatly enhance the complexity of our game world. These systems can interact and introduce more random scenarios that would be interesting not only for players but also for developers like us.

More Map Objects

We would like to include more map objects, such as different plants and animals. These objects can have impact on the hero or the properties of the world.

More Complicated Systems

We would like to introduce more complicated universal systems such as a physics system. We can include physical properties such as temperature, gravity, humidity, etc. that affect how the world works. This system should interact with the weather system nicely by having the weather system affect temperature and humidity properties.

Decoupling Knowledge Base and User Interaction

As mentioned in the weaknesses section, we would like to use Python to process user input and feed it into the Prolog system for knowledge processing. This decoupling would make it easier for development and gameplay.

References

[1] Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. Swi-prolog, 2010.