CmpE 260 - Principles of Programming Languages
Spring 2023 - Project 1
`4D Arena`

Deadline: 25 April 2023 17:00

Assistants: Alper Ahmetoğlu, Ali Nasra
`ahmetoglu.alper@gmail.com`, `ali.nasra@boun.edu.tr`

1 Introduction

In this project, you will implement a high-level logic for agents to battle with their enemies in an arena. One peculiar thing about this arena is that it is 4-dimensional: along with two spatial dimensions, there is a time dimension (agents can time-travel), and a parallel universe dimension (agents can jump to parallel universes). When an agent tries to make a time-travel to a previous time (possibly to another universe), the history is forked and a new parallel universe is created. Agents can also jump to the current timestep. There are some preconditions for time-travel stuff which will be explained in Section 2.

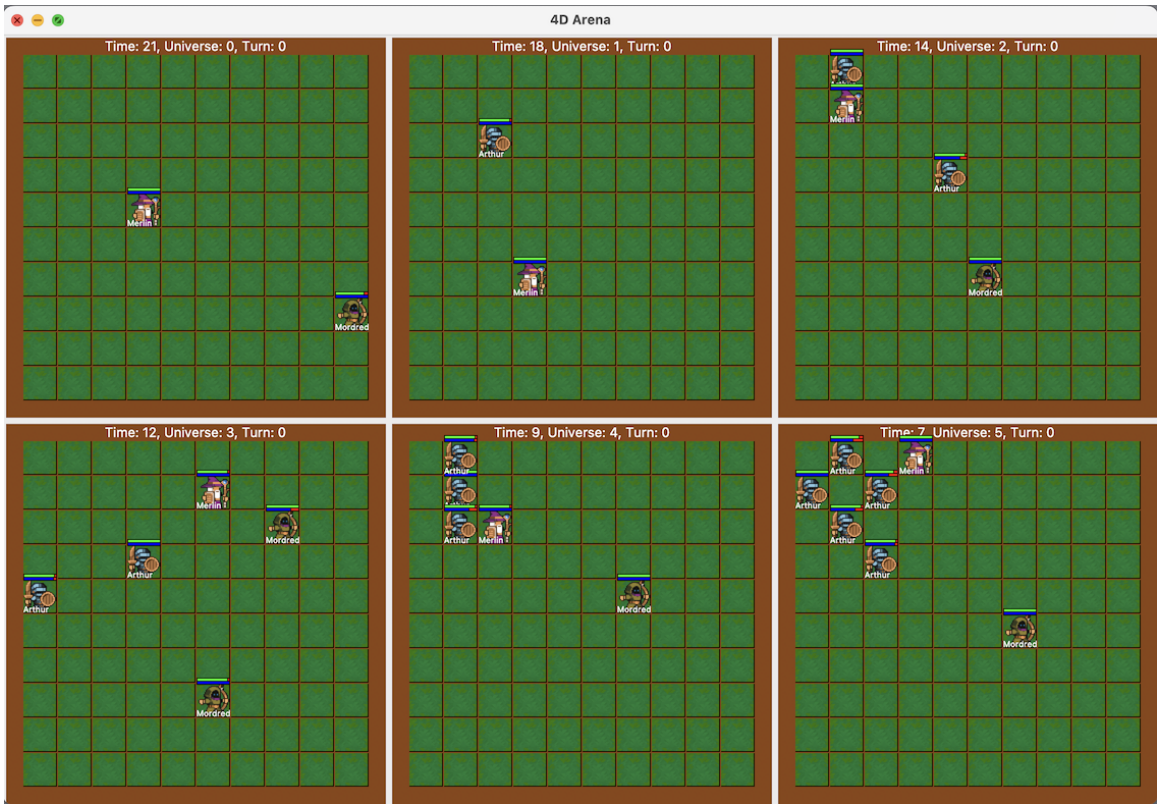

Figure 1: A screenshot from 4D Arena.

2 4D Arena Environment

A screenshot from the game is shown in Figure 1. A very basic renderer written in Python (with tkinter) is provided for you. This renderer first steps the simulation by querying `main_loop(Steps)` for a given number of steps, then collects `state(StateId, Agents, TurnIndex, TurnOrder)` and sequentially visualizes states. You can run it with

```
python3 renderer.py <scene_file> <number_of_steps>
```

To run the environment definitions in your prolog interpreter, you can use the following command:

```
swipl -s main.pro scene1.pro
```

After you open your SWI-Prolog session, you can simulate a single turn with a specific action by running:

```
?- step_universe_turn_with_action(0, [move_left]).
```

This will be useful for debugging your agent. You can run the simulation with all universes with:

```
?- main_loop(100).
```

where 100 is the terminal time.

2.1 Classes

There are three different agent classes: warrior, wizard, and rogue. Warrior is a heavy-armored class that can deal high damages in close range. Wizards can send magic missiles from a far, useful for a hit-and-run strategy. As they are adept with magic, time/universe jumps cost less mana (energy) points for them. Rogues with their bows can make a ranged attack (though shorter than magic missile). They have high agility which allows them to take less damage from wizards. The below is the full definition of these agents:

```
Warrior = agent{x:1, y:1, class:warrior, health:100, mana:100, agility:2, armor:8,
name:'Arthur'}
Wizard = agent{x:1, y:1, class:wizard, health:100, mana:100, agility:5, armor:2,
name:'Merlin'}
Rogue = agent{x:1, y:1, class:rogue, health:100, mana:100, agility:8, armor:5,
name:'Mordred'}
```

- `health` is decreased when another agent attacks her.
- `mana` is decreased when the agent jumps to another time/universe.
- `agility` protects the agent from magic missiles.
- `armor` protects the agent from melee and ranged attacks.
- `name` is an arbitrary name of the agent. You will give unique names to your agents if you are planning to do the optional part.

2.2 Actions

Actions that are available to agents are as follows:

- `move_right`: increases the `x` property of the agent if possible.
- `move_up`: increases the `y` property of the agent if possible.
- `move_left`: decreases the `x` property of the agent if possible.

- `move_down`: decreases the `y` property of the agent if possible.

If the agent is at edges of the map, move action towards the edge do not work. Also, the agent cannot move to a cell that is occupied by another agent.

- `portal (UniverseId, Time)`: agent jumps to the state at `Time` on `UniverseId`. If time-travel is successful, a new state (on a new universe) is created which is a copy of the target state with the travelling agent in it. The agent vanishes from its previous universe. The agent spends

$$\texttt{TravelCost} * (|\texttt{CurrentTime} - \texttt{Time}| + |\texttt{CurrentUniverse} - \texttt{UniverseId}|) \quad (1)$$

  mana points, where `TravelCost` is 2 for wizards, and 5 for warriors and rogues.

- `portal_to_now (UniverseId)`: agent jumps to the state with the current time on `UniverseId`. If time-travel is successful, and a new state is created which is a copy of the target state with the travelling agent in it. The agent vanishes from its previous universe. The agent spends

$$\texttt{TravelCost} * (|\texttt{CurrentTime} - \texttt{TimeOnTargetUni}| + |\texttt{CurrentUniverse} - \texttt{UniverseId}|) \quad (2)$$

  mana points where `TravelCost` is 2 for wizards, and 5 for warriors and rogues. This action does not create a new universe.

- `melee_attack (TargetAgentId)` (only warriors): decreases the target agent's `health` property by 20 - `TargetAgent.armor`. This action can only be performed if the distance between the agent and the target agent is 1.

- `magic_missile (TargetAgentId)` (only wizards): decreases the target agent's `health` property by 10 - `TargetAgent.agility`. This action can only be performed if the distance between the agent and the target agent is 10.

- `ranged_attack (TargetAgentId)` (only rogues): decreases the target agent's `health` property by 15 - `Distance` - `TargetAgent.armor`. This action can only be performed if the distance between the agent and the target agent is 5.

- `rest`: increases the `mana` property of the agent by 1.

If an agent's health is dropped to 0, the agent is removed from the game. After each action, the turn index is changed to the next agent in the turn order. If the turn index is equal to the length of the turn order, the turn index is set to 0 and time is incremented. The turn order is a list of agent ids.

2.3 Time/Universe Jump Mechanism

This project is heavily influenced by 5D Chess[1], and therefore the time/universe travel mechanics are similar to it. An example of time/universe jumps are shown in Figure 2. Mainly, if other preconditions are met for time/universe travel, the target state for the time/universe jump is copied, the agent is added to the state, and a new universe is created. If the target state is not in the past for the target universe (in other words, if the agent tries to jump to the current time of that universe), a new universe is not created. These two cases are labeled in actions as `portal` and `portal_to_now`, respectively. Some example cases are shown in Figure 2. At `Time=3`, the red agent makes a time-travel to the `Time=1` on the same universe. For this action, the agent would have spent 10 mana points. Since the agent has travelled to a time point that is not a current time on the travelled universe (the same universe in this case), a new universe (`Universe=1`) is created with a clone of the state at `Time=1` with the new agent in it. The same case happens at `Time=1`, `Turn=2` when the blue agent travels to `Time=0`, `Universe=0`; a new universe (`Universe=2`) is created with the clone of the target state with the travelling agent in it. On the other hand, when the green agent travels to the current time on `Universe=1`, there is no new universe creation.

Below are the other preconditions for `portal` travel:

[1] `https://store.steampowered.com/app/1349230/5D_Chess_With_Multiverse_Time_Travel/`

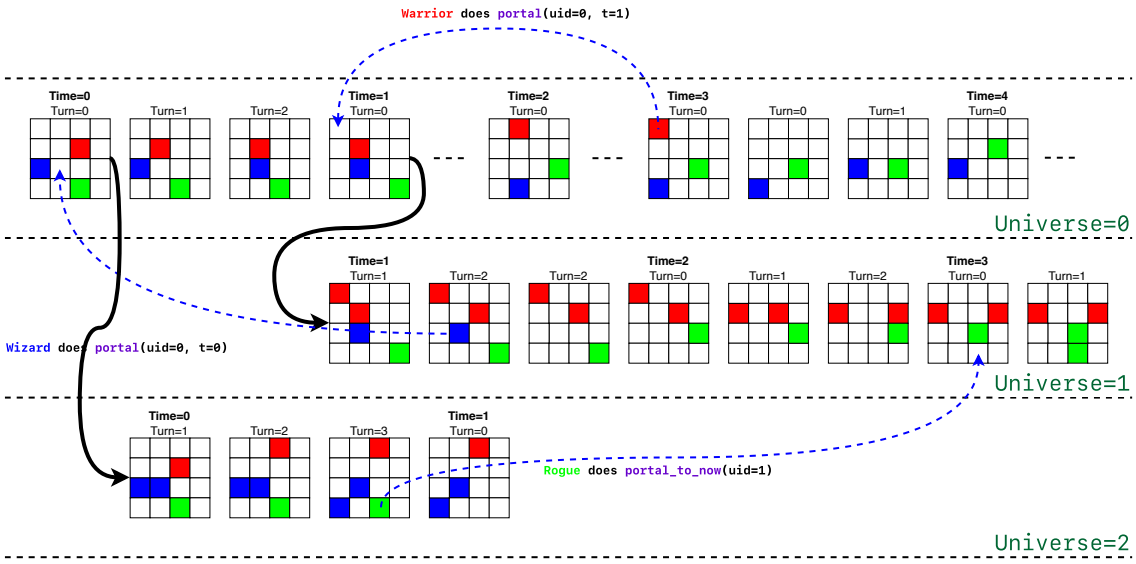

Figure 2: Example of time/universe jumps.

- The agent cannot travel to a time that would require a new universe creation if `global_universe_id` has reached to `universe_limit`.
- The agent cannot travel to any other time/universe if she is alone in her current universe.
- The agent should have enough mana points to travel to the target time/universe. The cost of travelling is given in Equation 1.
- The earliest turn at the target universe is treated as the target state that is going to be cloned (in Figure 2, there are two `Time=3`, `Turn=0` states as the red agent has travelled. The first state with `Time=3`, `Turn=0` is a valid target state).
- The tile that the agent is currently on should not be occupied at the target state.

Below are the other preconditions for `portal_to_now` travel:

- The agent cannot travel to any other time/universe if she is alone in her current universe.
- The target universe should not be the same as the current universe.
- The agent can only travel to universes that are at the start of the new time (i.e., `Turn=0`). In case there are two states with the same time and turn, the last state is considered as the target state (the reverse logic of the `portal` travel).
- The agent should have enough mana points to travel to the target time/universe. The cost of travelling is given in Equation 2.
- The tile that the agent is currently on should not be occupied at the target state.

Although it looks complicated at first, for the most part of the project, you do not need to worry much about the time/universe travel mechanics. The mechanics are implemented for you in the `simulator.pro` file. You just need to understand which states are valid for predicates 7 and 8.

3 Knowledge Base

The mechanics of the game is implemented for you in several files:

- `main.pro`: This is the main file that you should load. It loads all the other files, and initializes the game. You should not modify this file (but you can change the initial state of the game for debugging purposes).
- `simulator.pro`: This file contains the mechanics of the game. You should not modify this file.
- `agent.pro`: Currently, this file only contains a random policy. It will be used for battling agents with different policies.
- `solution.pro`: This is the file that you should modify. It contains the predicates that you are going to implement.

4 Predicates

In this section, we will go over the predicates that you are going to implement.

4.1 `distance(+Agent, +TargetAgent, -Distance)` 10 points

This predicate will compute the Manhattan distance between two agents. The Manhattan distance is also known as ℓ_1 norm:

$$D([x_1, y_1], [x_2, y_2]) = |x_1 - x_2| + |y_1 - y_2| \tag{3}$$

Examples:

```
$ swipl -s main.pro scene1.pro
?- state(_, Agents, _, _), A=Agents.0, B=Agents.1, distance(A, B, Distance).
Distance = 16.
?- state(_, Agents, _, _), A=Agents.2, B=Agents.0, distance(A, B, Distance).
Distance = 8.
```

4.2 `multiverse_distance(+StateId, +AgentId, +TargetStateId, +TargetAgentId, -Distance)` 10 points

This predicate will compute the multiverse distance between an agent and a target location in a different state. A different state might be in a different universe, and/or in a different time. The distance is computed as follows:

$$D([x_1, y_1, t_1, u_1], [x_2, y_2, t_2, u_2]) = |x_1 - x_2| + |y_1 - y_2| + \text{TravelCost} * (|t_1 - t_2| + |u_1 - u_2|) \tag{4}$$

The travel cost is 2 for Wizards and 5 for other classes.

Examples:

```
$ swipl -s main.pro scene1.pro
?- multiverse_distance(3, 0, 0, 1, Distance).
Distance = 9.
?- multiverse_distance(3, 1, 0, 2, Distance).
Distance = 3.
```

4.3 nearest_agent(+StateId, +AgentId, -NearestAgentId, -Distance) 10 points

This predicate will return the nearest agent for a given agent and a state. In case of a tie, you are free to return any of the nearest agents. The name of the nearest agent should be different than the name of the given agent. You will use `distance` predicate to find the nearest agent.

Examples:

```
$ swipl -s main.pro scene1.pro
?- nearest_agent(3, 0, NearestAgentId, Distance).
NearestAgentId = 2,
Distance = 7.
?- nearest_agent(3, 1, NearestAgentId, Distance).
NearestAgentId = 2,
Distance = 7.
?- nearest_agent(3, 2, NearestAgentId, Distance).
NearestAgentId = 0,
Distance = 7.
```

4.4 nearest_agent_in_multiverse(+StateId, +AgentId, -TargetStateId, -TargetAgentId, -Distance) 10 points

Similar to 4.3, this predicate will return the nearest agent in the multiverse (i.e., possibly in a different time/universe). The name of the nearest agent should be different than the name of the given agent. You will use `multiverse_distance` predicate to find the nearest agent.

Examples:

```
$ swipl -s main.pro scene2.pro
?- nearest_agent_in_multiverse(3, 0, TargetStateId, TargetAgentId, Distance).
TargetStateId = 3,
TargetAgentId = 1,
Distance = 5.
?- nearest_agent_in_multiverse(3, 1, TargetStateId, TargetAgentId, Distance).
TargetStateId = 3,
TargetAgentId = 2,
Distance = 2.
?- nearest_agent_in_multiverse(3, 2, TargetStateId, TargetAgentId, Distance).
TargetStateId = 3,
TargetAgentId = 1,
Distance = 2.
```

4.5 num_agents_in_state(+StateId, +Name, -NumWarriors, -NumWizards, -NumRogues) 10 points

This predicate will return the number of agents in a given state for each class. Do not count agents with the same name.

Examples:

```
$ swipl -s main.pro scene1.pro
?- num_agents_in_state(3, 'Arthur', NumWarriors, NumWizards, NumRogues).
```

```
NumWarriors = 0,
NumWizards = 1,
NumRogues = 1.
?- num_agents_in_state(3, 'Merlin', NumWarriors, NumWizards, NumRogues).
NumWarriors = 1,
NumWizards = 0,
NumRogues = 1.
```

4.6 difficulty_of_state(+StateId, +Name, +AgentClass, -Difficulty) 10 points

This predicate will return the difficulty of a given state for a given agent. The difficulty is computed as follows:

$$\text{Difficulty} = 5 * \text{NumWarriors} + 8 * \text{NumWizards} + 2 * \text{NumRogues} \quad \text{(for warriors)} \tag{5}$$

$$\text{Difficulty} = 2 * \text{NumWarriors} + 5 * \text{NumWizards} + 8 * \text{NumRogues} \quad \text{(for wizards)} \tag{6}$$

$$\text{Difficulty} = 8 * \text{NumWarriors} + 2 * \text{NumWizards} + 5 * \text{NumRogues} \quad \text{(for rogues)} \tag{7}$$

Examples:

```
$ swipl -s main.pro scene1.pro
?- difficulty_of_state(3, 'Arthur', warrior, Difficulty).
Difficulty = 13.
?- difficulty_of_state(3, 'Merlin', wizard, Difficulty).
Difficulty = 10.
?- difficulty_of_state(3, 'Mordred', rogue, Difficulty).
Difficulty = 10.
?- difficulty_of_state(3, 'Morgana', wizard, Difficulty).
Difficulty = 15.
```

4.7 easiest_traversable_state(+StateId, +AgentId, -TargetStateId) 10 points

This predicate will return the easiest traversable state for a given agent. A state is traversable if the agent can reach it by using `portal` or `portal_to_now` actions. The given state is assumed to be traversable as well. If there are multiple states with the same difficulty, you are free to return any of them. You will use `difficulty_of_state` for sorting states by difficulty.

Examples:

```
$ swipl -s main.pro scene1.pro
?- set_random(seed(1)), main_loop(20), easiest_traversable_state(166, 1, TgtStId)
TgtStId = 224.
?- set_random(seed(1)), main_loop(20), easiest_traversable_state(425, 0, TgtStId)
TgtStId = 166.
?- set_random(seed(1)), main_loop(20), easiest_traversable_state(425, 1, TgtStId)
TgtStId = 224.
```

4.8 basic_action_policy(+StateId, +AgentId, -Action) 10 points

This predicate will return the action that an agent should take in a given state. This is a very basic (not an optimal) action policy. For a given state, the agent should decide on the action as follows:

1. The agent should try to `portal` to the `easiest_traversable_state` if possible (i.e., if the game rules allow: the agent's mana, whether the tile is occupied or not, etc).
2. If it cannot travel to the easiest traversable state, it should approach to the nearest different agent (i.e., agent with a different name) until it is in the attack range of the nearest agent.
3. Once the agent is in the attack range of the nearest agent, it should attack the nearest agent.
4. If the agent cannot execute the above actions, it should rest.

You are free to choose any decision in case of a tie (e.g., if there are multiple nearest agents, you can choose any of them).

Examples:

```
$ swipl -s main.pro scene1.pro
?- set_random(seed(1)), main_loop(20), basic_action_policy(166, 1, Action)
Action = [portal_to_now, 1].
?- set_random(seed(1)), main_loop(20), basic_action_policy(166, 2, Action)
Action = [ranged_attack, 1].
?- set_random(seed(1)), main_loop(20), basic_action_policy(224, 0, Action)
Action = [portal_to_now, 0].
?- set_random(seed(1)), main_loop(20), basic_action_policy(224, 3, Action)
Action = [portal_to_now, 0].
?- set_random(seed(1)), main_loop(20), basic_action_policy(327, 0, Action)
Action = [move_up].
?- set_random(seed(1)), main_loop(20), basic_action_policy(327, 2, Action)
Action = [portal_to_now, 0].
?- set_random(seed(1)), main_loop(20), basic_action_policy(327, 4, Action)
Action = [portal_to_now, 1].
?- set_random(seed(1)), main_loop(40), basic_action_policy(682, 0, Action)
Action = [rest].
?- set_random(seed(1)), main_loop(40), basic_action_policy(618, 2, Action)
Action = [move_right].
?- set_random(seed(1)), main_loop(40), basic_action_policy(859, 4, Action)
Action = [move_left].
```

4.9 A Better Policy (optional, no bonus)

The basic action policy is a very primitive policy as you might have seen. Feel free to implement a better policy in `get_agent_action`. We will collect your policies (if they are different than random and basic) run each policy with a different agent, and battle all agents in a big arena. You are free to choose your agent class.

5 Documentation

Please explain what each predicate is for with comments in the code. Codes with no comments might lose points if a predicate is too hard to understand.

6 Submission

You will submit two files: `solution.pro`, `feedback.txt`. Your code should be in one file named `solution.pro`. First four lines of your `solution.pro` file must have exactly the lines below since it will be used for compiling and testing your code automatically:

```
% name surname
% student id
% compiling: yes
% complete: yes
```

The third line denotes whether your code compiles correctly, and the fourth line denotes whether you completed all of the project, which must be `no` if you're doing a partial submission. This whole part must be lowercase and include only the English alphabet (also note that there is a single space after %). Example:

```
% alper ahmetoglu
% 2012400147
% compiling: yes
% complete: yes
```

We are interested in your feedback about the project. In the `feedback.txt` file, please write your feedback. This is optional, you may leave it empty and omit its submission.

7 Tips and Tricks

Do not panic. Although the project may seem long at the first glance, it can be done in a reasonable amount of time. It might be better to first examine `simulator.pro`.

- Do not rush. First think about the requirements, what you need, how you can solve it in a modular way. If you can verbally describe your needs, you can probably implement it easily.
- Formalize the problem, then try to convert the logic formulate to Prolog.
- Try to imagine computation steps (each individual unification) that Prolog does for each of your predicate. The speed of your program might drastically change even if you change the order of your clauses.
- You can use findall/3, bagof/3 or setof/3.
- You are not allowed to use `assert/1` and `retract/1`.
- You can (and actually should) use extra predicates. The ones given above are compulsory.
- If a predicate becomes too complex, either divide it into some predicates or take another approach. Use debugging (through trace/1), approach your program systematically.