Collapse

Group Designation: Collapse_7

Group Members

- Nuno Simão Nunes Rios, up202206272@up.pt (contribution of 75%)
- Pedro Henrique Pessôa Camargo, up202102365@up.pt (contribution of 25%)

Installation and Execution

For starters, you need to have SICStus Prolog installed on your machine. Instructions on how to do install it are available @Moodle. Then download our project ZIP and extract it.

- On Linux and Mac, run the following commands on the terminal:

```
cd src/
sicstus
consult('game.pl').
play.
```

- On Windows, open the SICStus application, then head to `File > Consult > src > game.pl`. Finally, once you have access to the terminal run `play.` to start the game.

Description of the game

Collapse is a two-player board game designed by Kanare Kato. It is inspired by the *"block breaker"* genre of digital games and incorporates elements of many traditional board games. The goal is to *Collapse*, in other words, capture your opponent's pieces by colliding your pieces with theirs. A piece slides in a straight line until it collides with an enemy piece, capturing it. Players take turns and the game ends as soon as one player cannot make a valid capture on their turn or has lost all of their pieces. It's important to mention that each move must capture exactly one enemy piece, which makes the game more challenging and harder to strategize.

- Official Game Website
- Collapse Rules — rules for the game.

Considerations for game extensions

We considered many extensions for the game, namely variable-sized boards, although we did not have enough time to implement it. Implementing variable-sized boards wouldn't be hard, though, given how flexible our game architecture is. The game has 3 different modes: Human vs. Human, Human vs. Computer, and Computer vs. Computer. The user may choose a different difficulty level for each Computer in the Computer vs. Computer mode and may also choose the starting player in the Human vs. Computer mode.

Game Logic

Game Architecture

We divided the game into three main modules based on the MVC architecture- the model, the view and the controller, represented in the `game.pl`, `view.pl` and `controller.pl` files, respectively. The model is responsible for the game's logic and uses the view and controller to interact with the user. The view is responsible for all the user interface and the controller is responsible for handling user input. Each module provides its own set of predicates (like miniature APIs) for the model to take advantage of. It is also to be noted that the controller and the view are decoupled from the model and cannot directly change the game state- they simply provide an easy and more manageable way to interact with the user.

Game Configuration Representation

The game configuration dictates the nature of each player (human or computer), difficulty levels, and who controls the white or black pieces. It's essentially where we save the game settings. We represent it using an atom `config(white(Type, Level), black(Type, Level))`, where Type is the player type (`human` or `computer`) and Level is the difficulty level (1 or 2).

This configuration is saved to the game state atom, which is created using the `initial_state/2` predicate.

Internal Game State Representation

To store the current state of the game, we use an atom `state(CurrentPlayer, InitialBoard, Config)`, where the Config is the configuration atom we just mentioned. The current player is self-explanatory and can be either `white` or `black`. The board is a 2D list of atoms, which represent the game board and can be either a `w` (white cell), `b` (black cell), or `.` (empty cell).

Let's take a look at this example: `state(white, Board, config(white(human, 1), black(computer, 2)))` where InitialBoard is defined below. This state atom represents a game that has just started. It's the white player's turn to play, and he's playing against a computer, which is set to level 2, meaning it'll use the greedy algorithm to make its moves.

```
InitialBoard = [    [b,w,b,w,b,w,b,w,b],    [w,.,.,.,.,.,.,.,w],
[.,.,.,.,.,.,.,.,.],    [b,.,.,.,.,.,.,.,b],    [w,b,w,b,w,b,w,b,w] ].
```

As the game progresses, the configuration will remain the same, but the board will change as the players make their moves, and the `CurrentPlayer` will alternate. Below are visual representations of 3 examples of the game state during the *beginning*, *middle*, and *end* of a game.

- Initial state:

```
  --> It's white's turn! <--

(5)  B  W  B  W  B  W  B  W  B

(4)  W  .  .  .  .  .  .  .  W

(3)  .  .  .  .  .  .  .  .  .

(2)  B  .  .  .  .  .  .  .  B

(1)  W  B  W  B  W  B  W  B  W
```

```
   (1)(2)(3)(4)(5)(6)(7)(8)(9)

Enter coordinates as XY. (example: 12)
From -> 11
To -> 44
```

- Intermediate game state:

```
  --> It's black's turn! <--

(5)  B  W  B  W  .  W  B  W  B

(4)  W  .  .  W  .  .  .  .  W

(3)  .  .  .  .  .  .  .  .  .

(2)  B  .  .  .  .  .  .  .  B

(1)  .  B  W  B  W  B  W  B  W

   (1)(2)(3)(4)(5)(6)(7)(8)(9)

Enter coordinates as XY. (example: 12)
From -> 92
To -> 93
```

- End game state:

```
(5)  B  .  .  .  .  .  .  .  .

(4)  .  .  W  .  .  .  .  .  .

(3)  .  .  .  .  W  .  .  .  .

(2)  .  .  .  .  .  .  B  .  .

(1)  .  .  .  .  .  .  .  .  .

   (1)(2)(3)(4)(5)(6)(7)(8)(9)

=============== % ===============

   < GAME OVER, white wins! >

    Because black cannot move!

=============== % ===============
```

Move Representation

Moves are represented by the move_ atom, as follows: `move_(From, To, Captured)`, or, more explicitly, `move_((Fx, Fy), (Tx, Ty), (Cx, Cy))`. From and To are the starting and ending coordinates of the piece being moved, respectively, and Captured is the coordinates of the enemy piece

that was captured during the move- remember that in our game, a move does not replace an enemy piece's position, which is why To != Captured.

Note that we added an underscore to the `move_` atom to avoid conflicts with the `move/3` predicate. When a move is made, we only provide `From` and `To` tuples and the move predicate checks if the move is valid and return the Captured coordinates if that is the case (possible due to the flexibility of logic programming).

User Interaction

How I/O works

User interaction is achieved through the displaying of console messages and reading user input. The view handles the former and the controller the latter. The controller follows a very strict design pattern, based on two types of predictes: `get` predicates, responsible for reading user input, and `set` predicates, responsible for validating and parsing it. If the input is invalid, the `set` function will call the `get` function again (recursively) until a valid input is provided.

It's important to mention that the user can quit the game at any time by entering the **Q key** (followed by Enter) during an input prompt- the controller handles this gracefully by failing the predicate, which also fails the caller's predicate inside the model and therefore stops the game.

Game/ UI Flow

As mentioned earier, the view is responsible for the UI and all displayed messages maintain a consistent look throughout the game. The game starts by displaying the main menu, where the user can choose the game mode. Based on this choice, the user may also be prompted with additional questions, such as asking for the difficulty level of one or more players or asking for the starting player. All pre-game prompts have a default option, which can be selected by simply pressing Enter.

Once the game starts, for each move, the user is either asked to enter the coordinates of the piece to move along with the destination cell (with an `XY` format; example: `12` for X=1, Y=2) or is asked to press enter to continue the game, when the next player is a computer.

Conclusions

We are satisfied with the final result of this project and we learned immensily from it, especially when it comes to Prolog and logic programming in general. We patched all major bugs in the game and did not identify any minor issues (as of yet). We also implemented all the features that were asked of us- Collapse appears to be an easy game, but it is actually very challenging.

Below is a feature roadmap for the game, in case we decide to continue working on it in the future.

- **Move Hints**: For novice players, highlight valid capturing moves to simplify learning.
- **Variable-Sized Board**: Provide an in-menu option to choose non-standard board dimensions.
- **Animations**: Add animations to the game to make it more visually appealing.
- **More Advanced AI**: Implement a more advanced AI that uses a minimax algorithm.
- **Time Limit**: Add a time limit for each move to make the game more challenging.

Bibliography

- https://sicstus.sics.se/sicstus/docs/4.9.0/html/sicstus/Predicate-Index.html
- https://stackoverflow.com/questions/8519203/prolog-replace-an-element-in-a-list-at-a-specified-index
- https://cs.union.edu/~striegnk/learn-prolog-now/html/