Dag En Nacht_1

Bruno André Moreira Rosendo - up201906334 (50%) Pedro Miguel Jesus da Silva - up201907523 (50%)

Instalattion and Execution

To play the game you first need to have SICStus Prolog 4.7 or a newer version currently installed on your machine plus the folder with the source code.

Next, on the SICStus interpreter, consult the file *play.pl* located in the source root directory:

```
?- consult('./play.pl').
```

If you're using Windows, you can also do this by selecting `File` -> `Consult...` and selecting the file `play.pl`.

Finally, run the the predicate play/0 to enter the game main menu:

```
?- play.
```

The game is best enjoyed by selecting a bold and impactful font into SicStus' terminal.

Game Description

Board

The board is an NxN squared board, where N is an odd number between 11 and 19. The board has a checkered pattern of black and white spaces, with the four corners all being white. The board starts off empty.

Gameplay

The players are Black and White, with Black going first. Each has a supply of stones in their color.

On each turn, a player can take one of the following actions:

- Drop a stone of their color onto a black square - a stone may never be placed in a white square.
- Shift a stone of their color already on a board, in a black square, to an adjacent white square.

The winner is the first player to get five of their stones in a row horizontally or vertically, or four stones in row diagonally on white spaces. Stones on black spaces cannot win with a diagonal line.

Optional rules:

- To mitigate Black's advantage first move advantage, the players may agree to prohibit Black from winning with the easier B-W-B-W-B orthogonal line and force him to win only with the harder W-B-W-

B-W orthogonal line

Game Logic

Game state internal representation

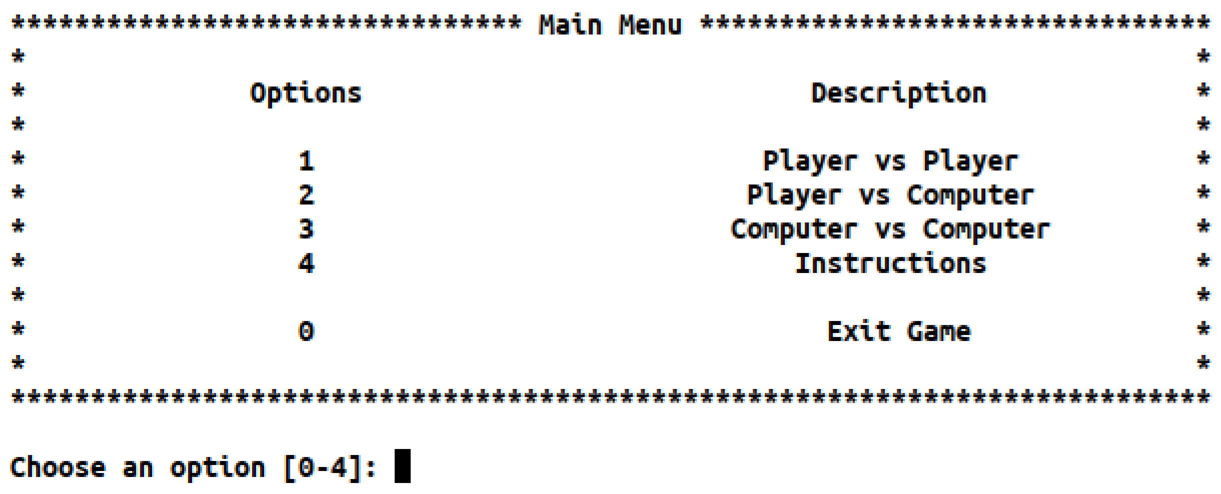
The **game state** is composed of the **current state for the board** and the **colour of the current player**.

- The **board** is represented by a **list of lists**: each list is a line on the board and each element in the list is a board cell. A **cell** is represented by a compound athom of the **colour** of the cell (**w** if white and **b** if black) and the **state** of the cell (**w** if there is a white piece there, **b** if there is a black piece there or **e** if the cell is empty).
- The **colour** of the **current player** is an athom which takes the value of **w** if White is playing or **b** is Black is playing.

Game state visualization

The predicates for the game visualization are separated into two different modules: `menu` and `game`, representing the menu and game state's interaction and display:

- The `menu` module has a few helper predicates meant to reutilize the code and ease the creation of new menu sections (*menu_io.pl*), e.g., **menuTitle/1** and **menuOptionsHeader/2**. The main menu predicates are created using those, like **mainMenu/0** and **chooseDifficulty/2** (*menu.pl*).



```
********************************* Main Menu *********************************
*                                                                           *
*              Options                            Description              *
*                                                                           *
*                 1                            Player vs Player            *
*                 2                           Player vs Computer           *
*                 3                          Computer vs Computer          *
*                 4                              Instructions              *
*                                                                           *
*                 0                               Exit Game                *
*                                                                           *
*****************************************************************************

Choose an option [0-4]: 
```

- Likewise, the `game` module has helper predicates used for interaction (*game_io.pl*), for example, **readUntilValidRow/2** and **askTypeOfMove/2**. These are used by the main predicates in *game.pl*, like **gameLoop/3** and **chooseMove/3**. All the user interactions are validated (either in the I/O or move execution predicates) and inform the players of possible errors before asking for another input.

- The game has a flexible NxN board, which must be an odd number between 11 and 19, dictated by the rules. The initial game state can be obtained with the predicate **initialState/2**:

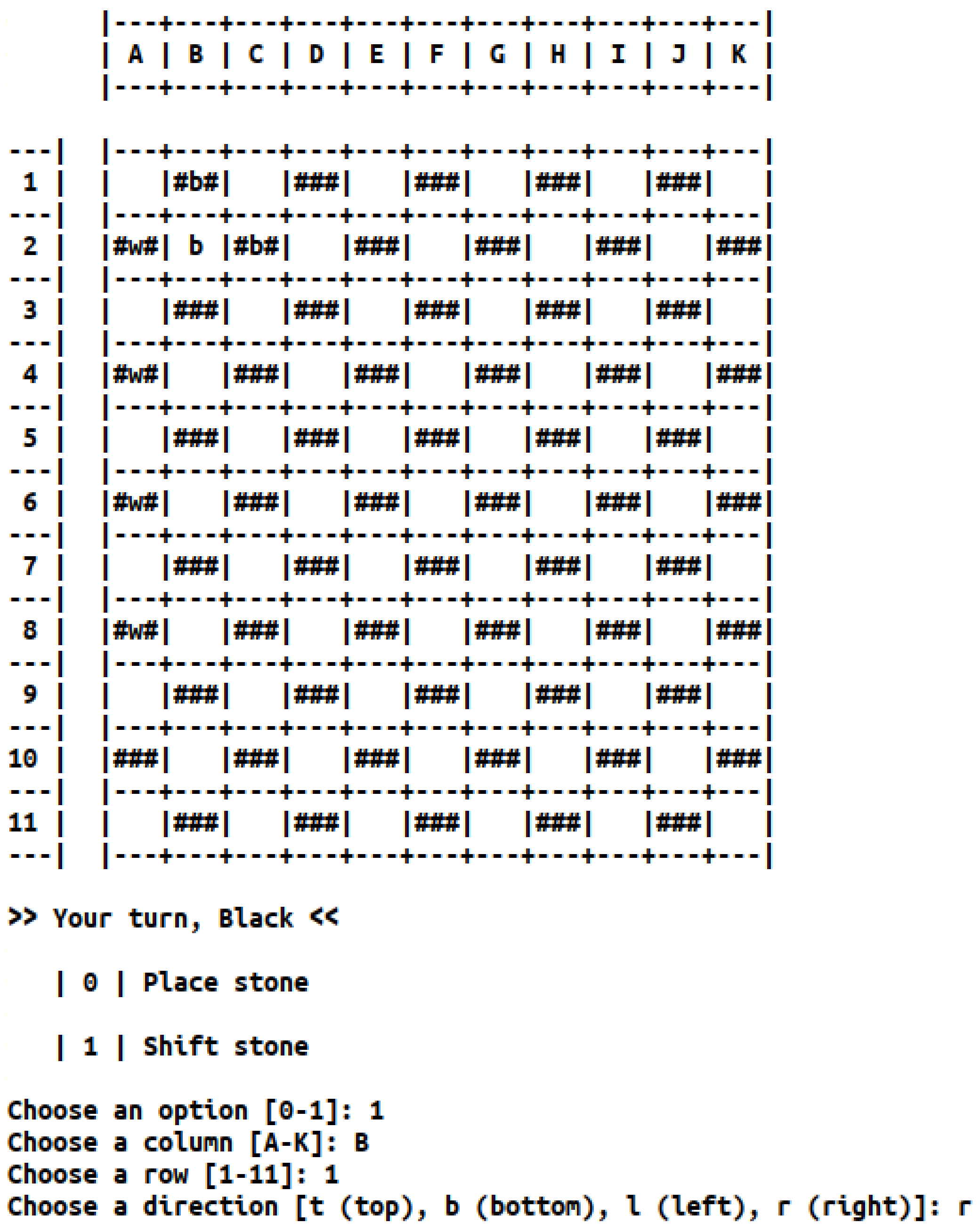
  ```
  initialState(+Size, -GameState)
  ```

  Note: The first player is chosen randomly

- The game board is displayed to the user by the predicate **displayGame/1**:

  ```
  displayGame(+GameState)
  ```

  It uses other support predicates like **displayColumns/1**, which displays the columns header, and **boardLine/3**, whichs prints a line of the board. Other important predicates for displaying the game include **displayBotMove/2**, which informs the user of the computer's move, and **printPlayerTurn/1**, which notifies the player of his turn.



```
       |---+---+---+---+---+---+---+---+---+---+---|
       | A | B | C | D | E | F | G | H | I | J | K |
       |---+---+---+---+---+---+---+---+---+---+---|

---|   |---+---+---+---+---+---+---+---+---+---+---|
 1 |   |   |#b#|   |###|   |###|   |###|   |###|   |
---|   |---+---+---+---+---+---+---+---+---+---+---|
 2 |   |#w#| b |#b#|   |###|   |###|   |###|   |###|
---|   |---+---+---+---+---+---+---+---+---+---+---|
 3 |   |   |###|   |###|   |###|   |###|   |###|   |
---|   |---+---+---+---+---+---+---+---+---+---+---|
 4 |   |#w#|   |###|   |###|   |###|   |###|   |###|
---|   |---+---+---+---+---+---+---+---+---+---+---|
 5 |   |   |###|   |###|   |###|   |###|   |###|   |
---|   |---+---+---+---+---+---+---+---+---+---+---|
 6 |   |#w#|   |###|   |###|   |###|   |###|   |###|
---|   |---+---+---+---+---+---+---+---+---+---+---|
 7 |   |   |###|   |###|   |###|   |###|   |###|   |
---|   |---+---+---+---+---+---+---+---+---+---+---|
 8 |   |#w#|   |###|   |###|   |###|   |###|   |###|
---|   |---+---+---+---+---+---+---+---+---+---+---|
 9 |   |   |###|   |###|   |###|   |###|   |###|   |
---|   |---+---+---+---+---+---+---+---+---+---+---|
10 |   |###|   |###|   |###|   |###|   |###|   |###|
---|   |---+---+---+---+---+---+---+---+---+---+---|
11 |   |   |###|   |###|   |###|   |###|   |###|   |
---|   |---+---+---+---+---+---+---+---+---+---+---|

>> Your turn, Black <<

   | 0 | Place stone

   | 1 | Shift stone

Choose an option [0-1]: 1
Choose a column [A-K]: B
Choose a row [1-11]: 1
Choose a direction [t (top), b (bottom), l (left), r (right)]: r
```

Move execution

The strategy for validating and applying a move was to create the predicate **move/3**:

```
move(+GameState, +Move, -NewGameState)
```

The predicate will fail if the given move is not valid.

Since the game has **two types of moves** (placing a stone or shifting one sideways a single time), the predicate move needed to have a **rule for each type**.

The first rule validates the move with **validatePlaceStone/2** and applies the move with **placeStone/3**. The second rule validates the move with **validateShiftStone/3** and applies with **shiftStone/4**.

The main helper predicates for this are **color/2** and **state/2**, which give information on the cell to use in validation, and **replaceCell/5**, which applies a change to the board.

Game Over

The strategy for checking if the game ended was to create the predicate **gameOver/2**:

```
gameOver(+GameState, -Winner)
```

The predicate will succeed if the game has ended.

Since the game has **three ways to win** (have five stones of one color in a row horizontally/vertically or four stones in row diagonally on white spaces), the predicate gameOver/2 needed to have a **rule for each win condition**.

The first rule checks all columns, the second one checks all rows and the third one all white diagonals.

The main helper predicates for this are **whiteDiagonals/2**, which gives all white diagonals of a board as a list of lists and **checkWin/3**, which based on the current player, a list of lines and the number of pieces in a row he needs to win, if that list contains a win condition.

List of valid moves:

The first step for the creation of the bot was to create the predicate **validMoves/2**:

```
validMoves(+GameState, -Moves)
```

Using **findall/3** and **move/3** we are able to find all valid moves a player can make on a given turn.

Game state evaluation

The strategy for evaluating the game board was to create the predicate **evaluateBoard/2**:

```
evaluateBoard(+State, -Value)
```

The approach of this predicate tries to model the question: *How many pieces am I missing to win in this position, and how many is my opponent missing?* If the opponent is ahead, the best course of action is to try and block his play. Otherwise, the best plan is to keep developing our "own" board. The code fflows like so:

- Join all vectors (lines, columns and diagonals) of the board where a win is possible.
- Calculate, for each player, an immediate value based solely on their pieces. This value represents the number of pieces the player needs to place or shift in order to win (if the opponent is blocking, the value is increased too). A vector is evaluated by the predicate **evaluateVector/4** and the value is better the smaller it is.
- Then, the **overallValue/3** predicate takes those two intermediate results and calculates the final value. If the player's value is 0, it means he already won (best outcome). Otherwise, we have two distinct scenarios:
  - The opponent is ahead: this is a bad position, since the opponent will win if the player doesn't block him. Hence, a low value for the player (in this case, it'll be 20 minus the opponent's value, since a large value means a bad board state).
  - The player is ahead: this is a good position, with great value for the player. In this case, the overall value is calculated by subtracting the opponent's intermediate value to the player's one (a constant of 5 is also added to guarantee that 0 is the minimum and always represents a win).
  - It's worth noting that the opponent is ahead even if the intermediate values are the same, since he'll be the next to play.

Computer move

The strategy for deciding the computer moves was to create the predicate **chooseMove/3**:

```
chooseMove(+GameState, +Level, -Move)
```

This predicate uses **validMoves/2** to get the list of possible moves and then chooses one based on the bot's level:

- **Easy**: Chooses a random move, using *random_select/3*.
- **Hard**: Chooses the best current move (greedy algorithm), based on the board's evaluation (recall **evaluateBoard/2**). For that purpose, uses *setof/3* to get the list of possible moves ordered by value and chooses the smallest one.

Conclusion

The board game *Dan En Nacht* was successfully implemented in the SicStus Prolog 4.7 language. The game can be played Player vs Player, Player vs Computer or Computer vs Computer (with the same or different levels).

One of the difficulties on the project was displaying an intuitive board in the SicStus terminal, which has a very limited set of characters and customization. This limits the game design, since it's hard to display black/white cells and black/white pieces at the same time. This issue was mitigated by using the characters 'b' and 'w', which isn't ideal.

Another limitation of the game is the bot's algorithm, which only looks at an immediate play, greatly reducing its cleverness. A possible improvement would be to implement another level with a better algorithm, for example, minimax.

Sources

- https://boardgamegeek.com/boardgame/347536/dag-en-nacht
- https://boardgamegeek.com/thread/2615621/new-game-dag-en-nacht