18 de novembro 2018

Faculdade de Engenharia da Universidade do Porto

Pente

Board Game

Programação em Lógica – Turma 3 - #Pente_4

Relatório Intercalar

Rui Jorge Leão Guedes – up201603854

João Fernando da Costa Meireles Barbosa – up201604156

Introduction

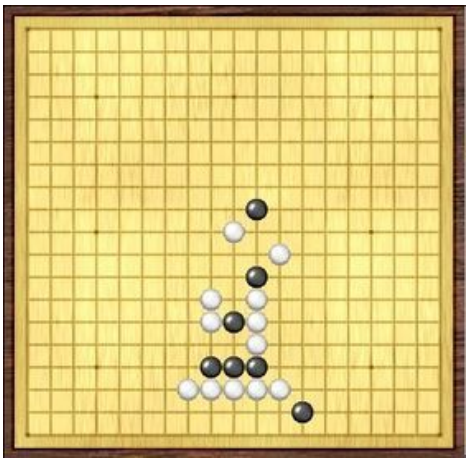
In order to get introduced to declarative programming, we were tasked with the development of a board game in Prolog. The chosen game, which in our case is Pente, a board game with Japanese roots, has to be deterministic and the final application must allow for three types of game modes (Player vs Player; Player vs AI; AI vs AI) where the AI can be set to at least two different difficulty levels. The user interface is meant to be established through a command line interface, although it can be extended with a user interface in another language through the use of sockets, for example.

With the development of this application, it is expected that we obtain some initial experience with declarative programming, Prolog in specific, applied in various domains; Prolog execution flow (flow of the game), constraint programming (rules of the game) and artificial intelligence (AI Player with various difficulties).

Pente Board Game

Origin

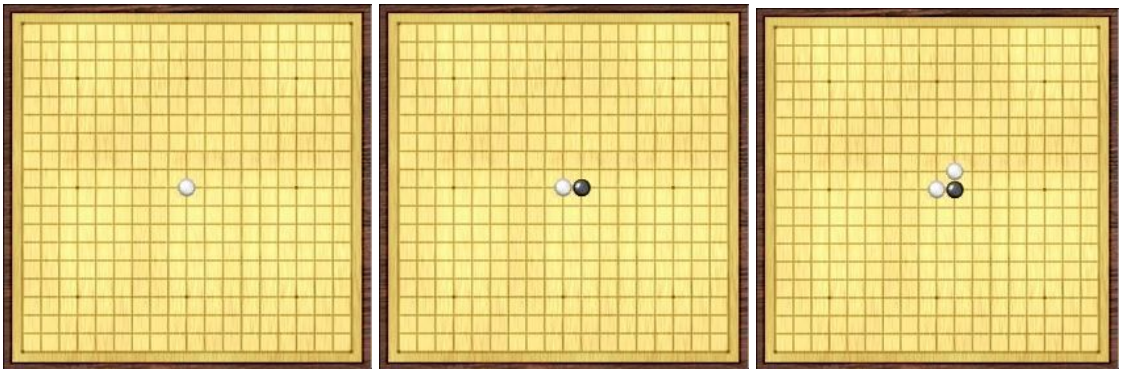
Pente is a strategy board game created by Gary Gabrel in 1977. Gary was a dishwasher at Hideaway Pizza, a restaurant in Stillwater, Oklahoma. Customers played Pente at this restaurant while waiting for their order. The game is based off a Japanese game called *ninuki-renju*, a professional variant of Gomoku. Both are played in a 19x19 Go board with white and black stones. Pente (πέντε), for reasons better understood knowing the rules, corresponds to the number five in Greek. Pente is played with two opposing players, however it can also be played by, at most, four people. They can either play in pairs of two, acting as partners, or play as independent players where each player has their own different colored stones. For the project, however, only the two player version is planned to be implemented.



A Sample Game of Pente

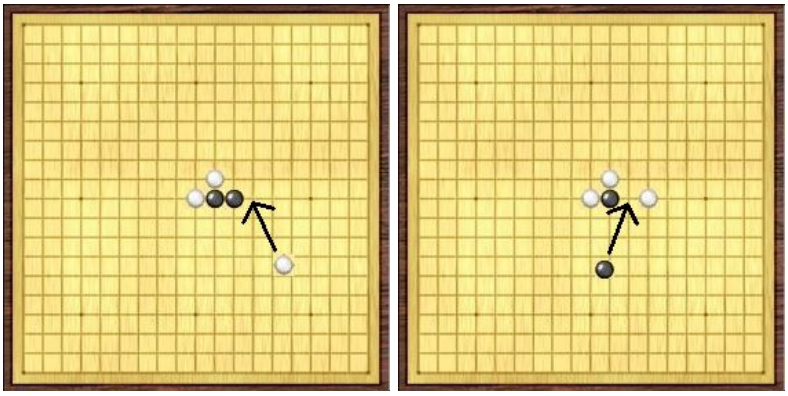
Ruleset

Pente is played with two players and, like with most board games, players alternate their moves, with white being the first player. Each move corresponds to the placement of a single stone of the player's color on an intersection of the board. The placement is arbitrary, as long as the intersection is free (The board is initially empty) except for the first move which must be in the center of the board. In a Tournament variation, in order to even the game, the second move of the first player must be at least three intersections away from the center. This rule, however, is not planned to be implemented.



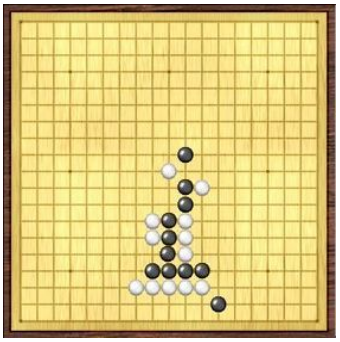
Representation of the evolution of a Pente Game, each board describing a turn.

During the game there's a single event that changes the state of the board, other than the regular placement of stones, called capture. Captures occur when, along any direction (Horizontal, Vertical or Diagonal), two contiguous stones of a color are surrounded by two stones of another color (For Example: X-O-O-X). When a capture occurs, the surrounded stones are removed from the board. It is only considered a capture if the capturing player takes the initiative. Multiple captures can occur in a single move.



In the leftmost image, a capture occurs and the black stones are removed, ***unlike*** *the rightmost image where initiative is being taken by the "victim".*

There are two winning conditions in Pente. A player wins when they either perform five captures (capture 10 opponent stones) or when they place at least five stones of their color in a row along any direction (Horizontal, Vertical, Diagonal). The picture on the right represents a possible final state of the game, where the White player wins the game due to having five white stones in a row. If all the board's cells are occupied without any of the winning conditions applying to a player, the game is considered a draw.



Sample of a victory state in Pente for the white player

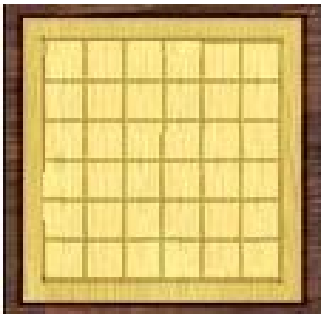
Game Logic

Game State Representation

The game state can be represented by the location of stones on the board. Throughout the game, its state changes with the mutation of the cells (intersections) on the board as specified in the ruleset. Each cell can be either free or occupied with a white or black stone.

Since the Go board is layed out in respect to two axis (Columns and Rows), a 2D Matrix was chosen as the data structure to keep the state of each cell of a board, thus representing, as a whole, one state of the game. This Matrix corresponds to a List where each element corresponds to a row of the board. Each row is then also represented as a List of cells. Each cell is an atom which represents its state ('0' -> free, '1'-> white stone, '2'-> black stone). The initial state of the board (where all its cells are free) can be retrieved from `initialBoard(BoardSize, P)`, in which *BoardSize* determines the size of board to be retrieved. Due to performance reasons for the AI, we choose not to implement the 19x19 board and opted for the 7x7, 9x9 and 13x13 board sizes. The number of pieces in a row and number of captures required to win vary with the board size. For this report, we will use the 7x7 board which has as winning conditions, 4 pieces in a row or 7 captures made.

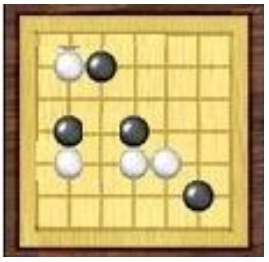
```
initialBoard('7x7', [ ['0', '0', '0', '0', '0', '0', '0'],
                      ['0', '0', '0', '0', '0', '0', '0'],
                      ['0', '0', '0', '0', '0', '0', '0'],
                      ['0', '0', '0', '0', '0', '0', '0'],
                      ['0', '0', '0', '0', '0', '0', '0'],
                      ['0', '0', '0', '0', '0', '0', '0'],
                      ['0', '0', '0', '0', '0', '0', '0']
                    ]).
```



List representation of the board(7x7) in its initial state compared to its graphical appearance

As previously stated, in order for the game state to change, each cell must be mutable. This functionality has been encapsulated into two predicates: `getPiece(LinNo, ColNo, Board, Piece)` and `setPiece(LinNo, ColNo, OldBoard, NewBoard, Piece)`. These predicates allow to easily get or set the state of a cell, knowing their position on a board, by simply navigating through the data structure (When setting a cell, the updated board must be unified to a new variable, since a new List is created). Using this method, it's possible to evolve the state of the game until we reach an end state, whether when a player wins or when the game draws due to a lack of free cells.

```
intermediateBoard('7x7', [   ['0', '0', '0', '0', '0', '0', '0'],
                             ['0', '1', '2', '0', '0', '0', '0'],
                             ['0', '0', '0', '0', '0', '0', '0'],
                             ['0', '2', '0', '2', '0', '0', '0'],
                             ['0', '1', '0', '1', '1', '0', '0'],
                             ['0', '0', '0', '0', '0', '2', '0'],
                             ['0', '0', '0', '0', '0', '0', '0']
                             ]).
```



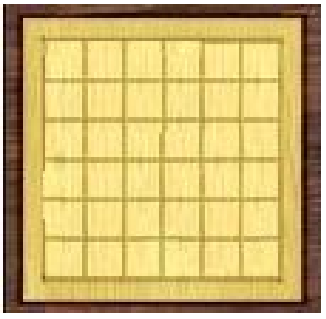
```
finalBoard('7x7', [   ['0', '0', '0', '0', '0', '0', '0'],
                      ['0', '1', '2', '1', '0', '0', '0'],
                      ['0', '0', '0', '0', '0', '0', '0'],
                      ['0', '2', '2', '2', '0', '0', '0'],
                      ['0', '1', '1', '1', '1', '0', '0'],
                      ['0', '0', '0', '0', '0', '2', '0'],
                      ['0', '0', '0', '0', '0', '0', '0']
                    ]).
```

List representation of the board in a possible intermediate and final state compared to its graphical appearance

Board Visual Representation (CLI)

The board is displayed to the user with a more user friendly design using a predicate called `displayGame(Board)`. This predicate is responsible for reading the board and converting its elements to their respective view representation by calling the predicate `modelToView(BoardElement, ElementView)` (0 -> '+', 1 -> 'O' 2 -> 'X'). Aside from a cleaner looking layout, similar to the Go board, row and column identifiers are added to aid the player in the input of their move.

```
  A  B  C  D  E  F  G
7 +--+--+--+--+--+--+ 7
  |  |  |  |  |  |  |
6 +--+--+--+--+--+--+ 6
  |  |  |  |  |  |  |
5 +--+--+--+--+--+--+ 5
  |  |  |  |  |  |  |
4 +--+--+--+--+--+--+ 4
  |  |  |  |  |  |  |
3 +--+--+--+--+--+--+ 3
  |  |  |  |  |  |  |
2 +--+--+--+--+--+--+ 2
  |  |  |  |  |  |  |
1 +--+--+--+--+--+--+ 1
  A  B  C  D  E  F  G
Player One -> [0] Captures      Player Two -> [0] Captures
```



Output in the console of displaying the initial state of the board, compared to its graphical counterpart

Valid Moves List

In order to obtain all the possibles moves from a certain state, we developed the predicate `valid_moves(Board, MoveList)`. This predicate receives a game state, represented by the board, and recursively visits every cell to determine whether is it empty ('0') or not (any other symbol). When a cell is empty, the move correspondent with placing a piece in that cell is valid, therefore it is added to the list of valid moves using the following structure: `move(LineNo, ColNo)`.

```
valid_moves(Board, MoveList) :- boardSize(Board, LineSize, ColSize),
                                valid_moves(Board, MoveList, LineSize, ColSize,
ColSize).

valid_moves(_, [], 0, _, _).
valid_moves(Board, MoveList, LineSize, ColSize, 0):- DecLine is LineSize - 1,
                                                     valid_moves(Board,
MoveList, DecLine, ColSize, ColSize), !.

valid_moves(Board, [move(LineSize, ColNum) | MoveList], LineSize, ColSize,
ColNum):-  getPiece(LineSize, ColNum, Board, '0'),
           !,
           DecColNum is ColNum - 1,
           valid_moves(Board, MoveList, LineSize, ColSize, DecColNum).

valid_moves(Board, MoveList,LineSize,ColSize, ColNum):- DecColNum is ColNum - 1,
                                                        valid_moves(Board,
MoveList, LineSize, ColSize, DecColNum).
```

Move Execution

The game transits from a state to another state when a move is made. Every move must be validated, executed and evaluated, in this specific order. These three phases occur at main predicate responsible for the game loop **gameStep/3**.

The validation of a move consists in checking whether the position where the move is going to be made is valid or not. A position is considered valid if its coordinates are within the board limits and if its correspondent cell is not occupied. This validation phase is only present for the human player, since the AI retrieves its valid moves in the manner specified above. The validation of a user's input position is done in the same predicate that retrieves it, handleInput(Board, CurrPlayer, Line, Column).

```
validateUserInput(Board, Line, Column) :- emptyBoard(Board), !,
                                          boardSize(Board, LineSize, ColSize),
                                          Line * 2 =:= LineSize + 1,
                                          Column * 2 =:= ColSize + 1.
validateUserInput(Board, Line, Column) :- getPiece(Line, Column, Board, '0').
```

After being validated, a move needs to be executed and posteriorly evaluated. For this purpose we developed the predicate **move/7**. which executes a validated move, using the already mentioned **setPiece/5** predicate and then evaluates the new game state with the use of two predicates: `updateSequence(CurrPiece, +SetBoard, +SetLine, +SetColumn, +CurrSequenceNo, -NewSequenceNo)` and `updateCaptures(+CurrPiece, +NextPiece, +SetBoard, -NewBoard, +SetLine, +SetColumn, +CurrCaptureNo, -NewCaptureNo)`, which, respectively, updates the number of pieces in a row and number of captures for a player. Only the sequences and captures containing the last piece are checked, since these are the only fornations that potentially alter the value for the new game state. These updated parameters are used in the last phase of evaluation which computes the score of a move using the `value` predicate analysed in the board evaluation chapter. Calculating the score with the transition of game state allows for easier game end detection and to, in a simple manner, update the board and players while the AI chooses its move.

Game End

As previously stated, Pente is a game with three possible endings: win by number of pieces in a row, or by number of captures and draw, due to no more available moves (all the board's cells are occupied). The predicate `game_over(NewBoard, NewCurrPlayer, NewNextPlayer, Score)`, determines if the game has ended, using three ordered clauses.

The first clause attempts to unify the argument `Score`, which corresponds to the current player score computed by **move/7**, with the value 100 which, as will become exposed later, correspond to the current player winning, whatever the reason. This unification, if successful, means that the current player, who is `NewNextPlayer` since the roles are switching, won the game.

```
game_over(NewBoard, NewCurrPlayer, NewNextPlayer, 100):-  !,
        winGame(NewBoard, NewNextPlayer, NewCurrPlayer).
```

The second clause checks if the board is full resorting on **fullBoard/1**. Its success means there are no more possible moves to be made, therefore the game ends in a draw.

```
game_over(NewBoard, NewCurrPlayer, NewNextPlayer, _):- fullBoard(NewBoard), !,
                                                        drawGame(NewBoard,
NewNextPlayer, NewCurrPlayer).
```

If neither of the previous two clauses succeed, the game must progress with the other player becoming the active player, therefore the last default clause simply invokes **gameStep/3**.

```
game_over(NewBoard, NewCurrPlayer, NewNextPlayer, _):- !, gameStep(NewBoard,
NewCurrPlayer, NewNextPlayer).
```

Board Evaluation (Heuristics)

A certain game state can be statically evaluated (evaluation of the state itself only) using the predicate **value/6**. It outputs the score of a game state based off each player's maximum number of pieces in a row ($x_{sequence}$) and number of captures ($x_{capture}$), x being a player $x \in \{C, N\}$, where C is the player who last made a move and N its opponent.

The **score** computed by this predicate can be described by the following evaluation function (For simplicity sake, the evaluation function will be shortened as $value(S, C, N)$, where S is the game state being evaluated, and C / N are the players as described above):

$$value(S, C, N) = \begin{cases} -100 & \Leftarrow N\ wins \\ 100 & \Leftarrow C\ wins \\ C_{capture} - N_{capture} + C_{sequence} - N_{sequence} & \Leftarrow otherwise \end{cases}$$

Two important properties of this function, related with how the AI works, are the following:

- $P_1 : \forall S,\ value(S, C, N) = -value(S, N, C) \Rightarrow \sum_i [value(S_i, C, N) + value(S_i, N, C)] = 0$
- P_2 : For every state S_1 and S_2, where C has a bigger advantage* in S_1 than in S_2, $value(S_1, C, N) > value(S_2, C, N)$

*Advantage in terms of the heuristics used for the evaluation function (Number of captures and pieces in a row).

```
winning_conditions(7 , 5 , 4).
winning_conditions(9 , 7 , 4).
winning_conditions(13, 10, 5).

value(Size, _, _, NextCaptureNo, NextSequenceNo, -100) :-
winning_conditions(Size, WinCaptureNo, WinSequenceNo),
                                                (NextCaptureNo >=
WinCaptureNo ; NextSequenceNo >= WinSequenceNo).

value(Size, CurrCaptureNo, CurrSequenceNo, _, _, 100):-
winning_conditions(Size, WinCaptureNo, WinSequenceNo),
                                                (CurrCaptureNo >=
WinCaptureNo ; CurrSequenceNo >= WinSequenceNo).

value(_, CurrCaptureNo, CurrSequenceNo, NextCaptureNo, NextSequenceNo, Score):-
      Score = CurrCaptureNo - NextCaptureNo + CurrSequenceNo - NextSequenceNo .

value(Size, player(_, _, CurrCaptureNo, CurrSequenceNo), player(_, _,
NextCaptureNo, NextSequenceNo), Score):- value(Size, CurrCaptureNo,
CurrSequenceNo, NextCaptureNo, NextSequenceNo, Score).
```

AI Move Selection

The **choose_move/7** predicate implements an alpha-beta pruning algorithm, an optimization of minimax used in various domains such as decision theory, game theory and statistics. In minimax, there usually is a maximizing player and a minimizing player (Higher score is better for the maximizing player and worse for the minimizing player). Due to the complexity of Pente, it is not viable to apply the algorithm since the end states of the game. Instead it generates a tree from the current state with a predefined depth, depending on the difficulty.

```
aiDepth(player(easyAI, _, _, _),   1).
aiDepth(player(mediumAI, _, _, _), 2).
aiDepth(player(hardAI, _, _, _),   3).
```

Pente is a zero-sum game, meaning that when a player gets an advantage, the opponent gets an equivalent disadvantage (The heuristics function has this property - P_1). This means that, when the AI player uses the algorithm to minimize its opponent's maximum advantage, it also maximizes its own minimum advantage, so the chosen move is guaranteed to have the calculated score, regardless of the opponent's strategy.

In alpha-beta pruning, the minimum score the maximizing player is assured of (α) and the maximum score the minimizing player is assured of (β) are kept and updated while iterating the descendants of a node. When alpha becomes larger or equal than beta ($\alpha \geq \beta$), it becomes pointless to continue iterating through these descendants, since the node being analysed will never be chosen by the algorithm, so we can cut this sub-tree.

```
pruning(_, MaxBoard, MaxBoard, _, _, _, MaxCurrPlayer, MaxCurrPlayer, _,
MaxScore, MaxScore, Alpha, Beta) :- Alpha >= Beta, !. % No need to keep
processing the tree, we can stop here.

pruning(Board, BestBoard, MaxBoard, MS, CurrPlayer, NextPlayer, BestCurrPlayer,
MaxCurrPlayer, Depth, BestScore, MaxScore, Alpha, Beta) :- !, minimax(Board,
BestBoard, MaxBoard, MS, CurrPlayer, NextPlayer, BestCurrPlayer, MaxCurrPlayer,
Depth, BestScore, MaxScore, Alpha, Beta).
```

Usually, in this algorithm, when iterating the descendants, you swap between the maximum and the minimum score (Due to changing from one player to another). This requires two separate clauses for handling each case, specially since the update of the α and β values is different in each case as well. By considering both the mentioned properties of the evaluation function (P_1 and P_2), this necessity can be avoided, when recursively invoking

the **choose_move/7** predicate as a goal, by swapping both players and the inverted $\alpha - \beta$ values before the call, and by inverting the score computed by this predicate afterwards.

```
NegAlpha is -Beta,
NegBeta is -Alpha,
choose_move(NewBoard, _, NextPlayer, NewCurrPlayer, _, DecDepth, NegScore,
NegAlpha, NegBeta),
Score is -NegScore.
```

Conclusions

The final application fulfills all the initially described objectives, enforcing an accurate ruleset of Pente and providing an interface with the user, allowing it to play the game against other users or against an AI. Its development was a great learning experience, serving as a good introduction to Prolog and declarative programming (constraint logic programming in particular).

While overall satisfied with the end result, some aspects regarding the structuring of data are a little lacking due to bad early design choices. Despite efforts in correcting these mistakes as they became more apparent, a few artifacts still remain such as the Player structure and the lack of structuring of cell positions (Line and Column). Another aspect we'd like to improve is the heuristics of the AI, which is its major downfall. Counting certain formations that give an edge to a Player (double-threat formations, formations that lead to double-threat) in the evaluation function allow the AI to be aware of these factors without affecting performance (Detection of double-threats at depth 1, compared to the depth 3 necessary with the current heuristics).

References

Wikipedia - Pente (24 April 2018)
https://en.wikipedia.org/wiki/Pente

pente.org - Game Rules
https://pente.org/help/helpWindow.jsp?file=playGameRules

swi-prolog.org
http://www.swi-prolog.org/