Pente

`Prolog` Implementation of a Board Game

PLOG 2018

Bruno Dias da Costa Carvalho up201606517
Amadeu Prazeres Pereira up201605646

December 20, 2018

Contents

Introduction

This project's objective is to implement a board game using Prolog, with 3 different modes: Player vs Player, Player vs Bot and Bot vs Bot, with the Bot having 2 different difficulties. Our game is Pente.

We implemented all of Pente's rules (see below) and created proper abstractions and compound terms to represent boards, game states, and other repetitive data elements. This will all be discussed below.

Overview and History

Pente is an *abstract strategy board game*, played usually by two players, in which the aim is to create an unbroken chain of five stones *or* to capture ten of the enemy's stones.

It was created in 1977 by Gary Gabrel at the restaurant *Hideaway Pizza*, in Stillwater, Oklahoma, USA.[1] Customers waiting for their orders to arrive would play a variation of the game on checkerboard tablecloths.[1]

Some variations allow for more than two independent players, and even teams, to play simultaneously. These require relaxing the winning conditions of the game to accommodate for the increased opposition — namely requiring only a four-in-a-row for three or four players, and allowing mixed captures for teams.[2]

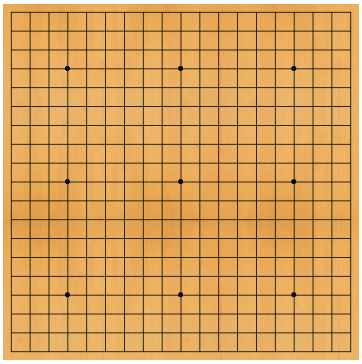

Figure 1: 19x19 *Go* board.

The game has many variations for two player games – all of which have a common ancestry in *Gomoku*, which has a significantly simpler rule-set. All variations of the game are played in the style of *Go*, on the intersections of a traditional 19x19 *Go* board with black and white pieces called *stones*. One player, *White*, uses the white stones, and the other, *Black*, uses the black stones. Once placed on the board, stones may not be moved, but may be removed from the board if *captured*.

Introductory or speed games may be played on the smaller 9x9 or 13x13 boards, but the tighter space makes it very difficult to generate common patterns of play.

Game Rules

The game's precise rules vary considerably throughout variations and sources. As such, we'll first review the common base rules and then discuss a few variations.

Base rules

Let's recall there are two winning conditions, same for White and Black:

- Form an unbroken chain of 5 or more consecutive friendly stones — vertically, horizontally or diagonally.
- Capture a total of 10 enemy stones.

Unlike traditional *Gomoku*, the chain may indeed have more than five consecutive stones.

Captures

Captures occur when two friendly adjacent stones (and only two) become bracketed by a pair of enemy stones, in a configuration depicted in Figure 2. Captures may arise in any direction.

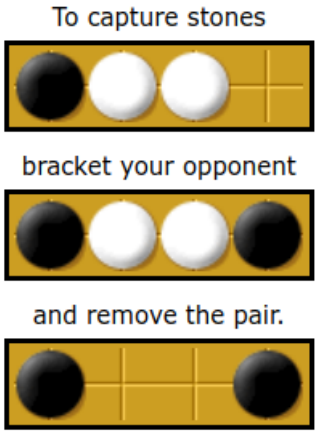

Figure 2: Capturing[3]

Variations

Some sources add the following secondary rules.

- White always plays first, in the center.[1,5] This is unlike *Renju* and *Gomoku*.
- *Suicides* are not possible – if a player places two adjacent friendly stones into a bracketed position — for example, if White forms the second panel of Figure 2 — his stones are not captured.[1,2,4,5]
- Tournament Rule: The first player's second move is restricted – it must be at least three intersections away from the center (that is, outside the board's middle 5x5).[1,2,3,4]

We'll be using these three rules hereafter.

Other sources do not specify starting player (or choose Black) and do not require the tournament rule, at least for casual play.

Among variations we find: suicides allowed, called *poofs* (*Poof-Pente*); harsher Tournament Rule (*G-Pente*, *D-Pente*); 3-in-a-row captures (*Keryo-Pente*), and others.[3,4]

Internal Representation

Representing the game's board is fairly simple. Every position in the 19x19 board is in one of three states: white piece, black piece, or empty. We'll represent the board using an SxS matrix (list of lists), whose elements are **w**, **b** or **c**, for each state respectively. This matrix will be called `Board`.

Now, each player has captured a certain number of pieces (an integer) and only plays pieces of a certain color (white or black). We'll represent the captures as a pair [`Wc`,`Bc`], with no similar general abstraction for *players* (it wasn't needed).

The overall game state will be represented by a *game/5* data compound:

game(Board, P, [Wc,Bc], Turn, Options)

where P is w or b according to who will play next, Turn is an integer starting at 0 that counts moves, and lastly, Options is a list that represents the game's options (see subsection 5.1).

Definitions, Compounds and Abstractions

There are two more data compounds: `val/4`, which holds a board evaluation (see subsection 5.5) and `node/6`, which is a node in the bot's evaluation tree (see subsection 5.6).

Several list data structures are abstracted away either by library or auxiliary predicates which act on them: `Move` is merely a row column pair [R,C]; `Range` is a list [I,J] representing an interval (which includes I and J); `Cap` is the pair of captures [Wc,Bc]; the three compounds are usually kept unexpanded in variables `Game`, `Val`, `Node`, `Tree`, etc.

Board Display

It's possible to flip the board (display only, not representation) when it is Black's turn to play, as if the two players were playing face-to-face on a physical Go board. Naturally we adjust the identification of rows and columns. By default this is turned off.

To draw the board with text (on the console) we used unicode box-drawing characters (range `u+2500`–`u+257f`). The white and black pieces become filled and empty unicode circles, respectively. This assumes a dark-themed console — white consoles will have the colors swapped. Below the board you can see the pieces captured and the turn number.

For examples see Figure 3, Figure 4 and Figure 5 below.

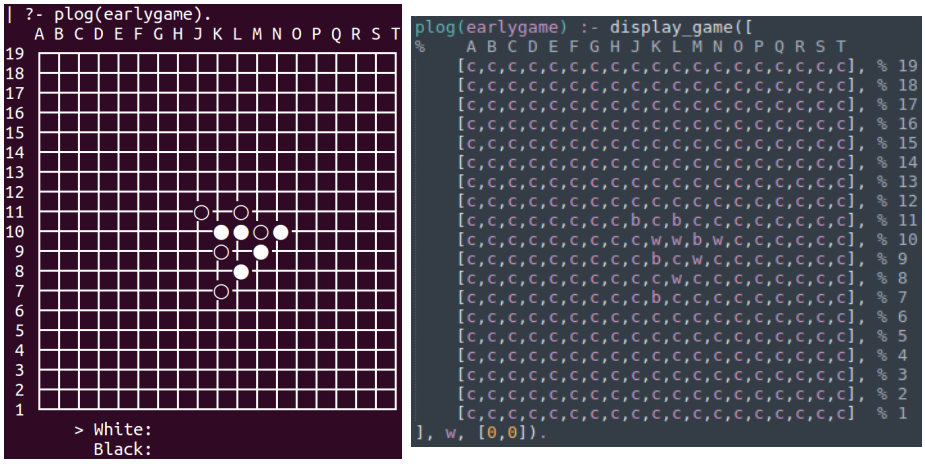

Figure 3: A game after 10 moves. It is White's turn to play.

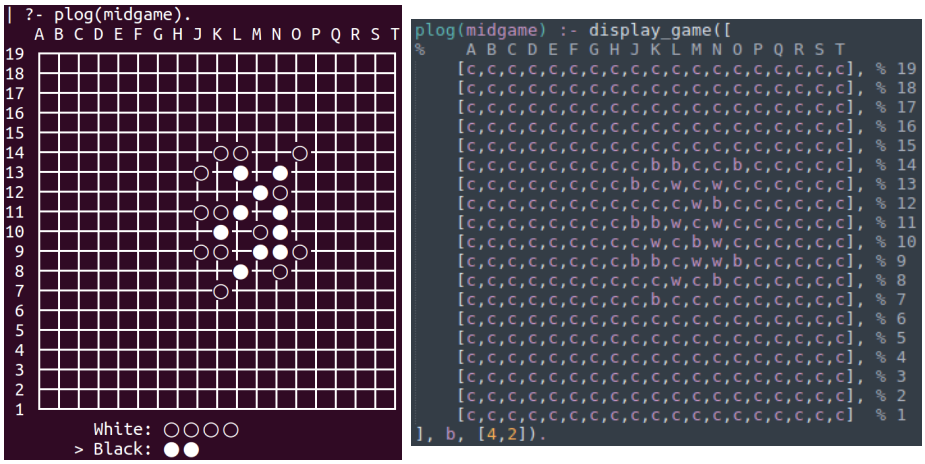

Figure 4: A game after 29 moves. It is Black's turn to play.

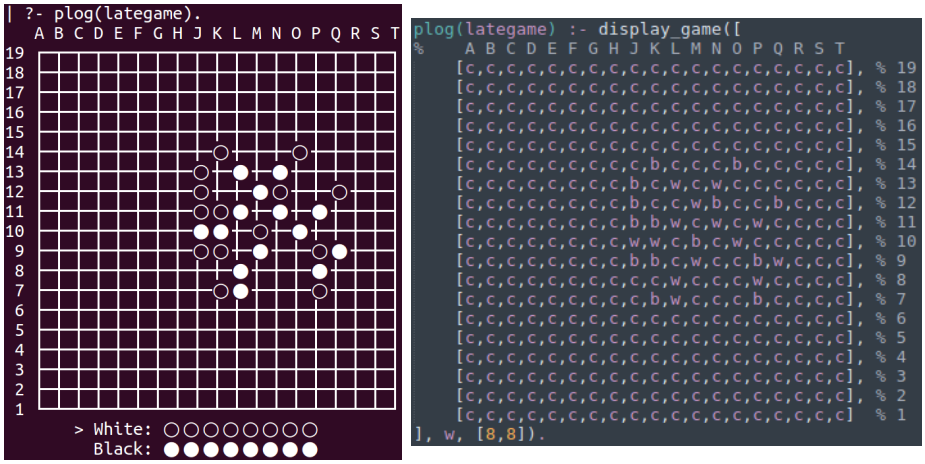

Figure 5: A game after 44 moves. It is White's turn to play, and win at H11.

Game Logic

The entry points are `pente/[0,1]` and `play/[0,1]`. The functors accept an `OptionsList` argument specifying implementation details or game rules only; the game type — Player vs Player, Player vs Bot and Bot vs Bot — are chosen through a simple menu afterwards.

See also predicate `help/[0,1]`.

Options

The options accepted in the `OptionsList` argument of `play/1` are the following (use `help` or `help(Option)` to get a description on the console):

- **`board_size(S)`** or **`size(S)`**, `S` in $\{7, 9, 11, ...\}$
  Defines the board size (SxS), must be odd. Default: 19.
- **`difficulty(D)`**, `D` in $\{1, 2, 3, 4, 5\}$
  Sets the bot difficulty (with 5 being the most difficult) by selecting a set of predefined values for depth, padding and width. If given values for depth, padding and width, these override those selected by the difficulty.
- **`depth(D)`**, `D` in $\{0, 1, 2, 3, ...\}$
  Depth of the bot's search tree.
- **`padding(P)`**, `P` in $\{1, 2, 3, ...\}$
  Padding of the active subboard used by the search tree.
- **`width([W1,W2,...])`**, `Wi` in $\{1, 2, 3, ...\}$
  For each search node on depth i (top is depth 0) recurse the search tree only for the W_i best moves. If the width is not defined for all the depths, the last width in the list will be used repeatedly. Please be mindful that bot runtime is $o(W1 \times \cdots \times W_D)$ on average per move.
- **`width(W)`**, `W` in $\{1, 2, 3, ...\}$ Use the width W for all depths.
- **`flip_board(B)`** or **`flip(B)`**, `B` is `true` or `false`
  Flip the board print on the console for Black's turn. Default: false.
- **`tournament_rule(B)`** or **`rule(B)`**, `B` is `true` or `false`
  Use the tournament rule. Default: true.

Valid Moves

In Pente, the player's possible moves do not depend on which player is going to play. The available moves are the same for each one. With this in mind, we implemented several predicates that give us a list with all the possible moves a player can make: `empty_positions/[2,3]`, `valid_stones/[2-5]` and their `within_boundary` counterparts.

- **valid_moves/2**: `valid_moves(+Board, -ListOfMoves)`
  This gives the player all the empty positions on the board, does not take in consideration the game's turn or if the user wants to play using the tournament rule.
- **valid_moves/3**: `valid_moves(+Board, +Turn, -ListOfMoves)`
  More specific than the previous one, it takes a Turn and makes sure that at the turn number 0 the only available move for the player is the center intersection, and at turn number 2 implements the Tournament Rule.
- **valid_moves/4**: `valid_moves(+Board, +Turn, +Rule, -ListOfMoves)`
  Takes into account the game Turn and if the Tournament Rule is active or not.

Validation and Execution of Moves

Input is expected quietly whenever it is the player's turn to make a move. Moves are typed into the console like

```
K10.
```

and column K, row 10 is deduced. If invalid, new input will be silently expected.

When a player chooses a move it needs to be checked before being accepted. This is done by asserting the move is in the appropriate list of valid moves (see `valid_move/[2-4]`) returned from one of the predicates described above.

Once the player move is valid we have to do several things:

(i) Place the player's stone at the given position.
(ii) Check if that stone bracketed the opponent, in every one of the eight possible directions. In that case, remove those 2 stones from the board and add 2 captures to the player's total captures.
(iii) Update the next player playing.
(iv) Increment the game's turn by 1.

Items (i) and (ii) above are carried out by `place_stone/5` through the use of `remove_dead_stones/5`. The whole process is `move/3` as seen from the main game loop.

Game Over

In the end of every game iteration a test is performed by **game_over/2** to check for either winning condition, for either player.

In case of victory, a message is displayed for that player and the game cycle is stopped, ending the game successfully. Otherwise the game must go on.

Board Scoring

Our board evaluation idea is simple in theory, but hard to implement *effectively*. Consider an SxS Board. It has S rows, S columns, $2S-1$ diagonals parallel to the main diagonal – we call them *left* diagonals, as the longest starts on the top left – and $2S-1$ *right diagonals*, perpendicular to the main diagonal. All of these are ways of splitting the two-dimensional Board into multiple one-dimensional *lists*.

Good Pente patterns might show up in any such list, namely four and five-in-a-rows, or simpler patterns at the beginning of the game. Hence we decided the value of a Board is determined entirely by the individual values of these lists. No two-dimensional patterns are recognized or searched for when evaluating a Board. The value of a game position is the board value plus a function of the player captures.

The Board's value is stored in compound `val/4`:

```
val(RowValues, ColValues, LeftValues, RightValues)
```

Board value is based entirely on *list patterns*. Through `pattern/2`, a *score* is given to each short pattern comprising one to six pieces all of the same color. From patterns as simple as[1]

```
- W -
```

to patterns such as

```
- W - W W - W W W - -
```

Scores range from 1 for a single piece to 2^{100} for a five-in-a-row.

To score a given List, for every recorded pattern one counts its number of occurrences in List, multiples it by the pattern's score, and adds all of them up to get the value of List.

Patterns made with White pieces are positive (`pattern/2`), those made with Black pieces are their negative. Scoring each list is done with `evaluate/2`, which accumulates `score/2`; the Board's total value is computed with `evaluate_board/2`, and finally the value of the game position is computed with `totalval/3`, which needs the player captures.

There are about 150 patterns scored by hand, for a total of 300 `score/2` searches in every list[2]. Some of these are not very meaningful and can be removed to increase efficiency substantially, but we did two other things to speed up evaluation for the Bot:

- Instead of evaluating an entire board every time with `evaluate_board/2`, we use mostly `reevaluate_board/4` which takes two Boards that differ by just one stone placement (one move), the initial Board's value, and computes the new Value, only having to reevaluate, most of the time, four new lists.
- Predicate `evaluate/2` is dynamic and every new list evaluation result is stored using `asserta` (the lemma idiom). This results in tens of thousands of clauses for `evaluate/2` once the bot kicks in, making evaluation mostly a matter of lookup.

[1] The dashes represent empty intersections

[2] Predicates `p/2`, `e/1` and `a/1` present different ways of investigating the score of a list

Motivation

If we create a decent, fast Bot but our hand-picked scores are very bad, they can be improved at a future time through machine-learning and self-play, or on a lower scale through simple experimentation.

By downscaling a two-dimensional board into multiple one-dimensional lists we lose tremendous amounts of information, but to look for various different two-dimensional patterns in a 19x19 board efficiently is just computationally infeasible. Look at Go for example.

Another idea we had was to look mostly/exclusively for forcing patterns on the board (possibly two-dimensional) like

```
- W - W W W - -
```

which require Black to respond locally, possibly twice; or even patterns like the *tessera*

```
- W W W W -
```

which are win-in-one bar any captures. But these don't help the Bot play the first few moves in the game, are very hard-coded, are require much more knowledge of the game than the one we have.

Computer Bot Moves

We made an elaborate, although static and largely imperfect Bot to play Pente which uses board evaluations for the lookahead tree.

The lookahead tree is *not* (necessarily) binary. It's rooted in the current game position, its children are the possible moves being analyzed, and in general every subtree head is one move away from any of its children. The tree is static: it has fixed depth and fixed width set at game startup, which do not change throughout (see subsection 5.1).

A node in the tree (and the tree itself) is represented by `node/6`:

```
node(Board, P, Val, Cap, Children, Worth)
```

where P is the next player (the one making the move M that transforms the node into one of its children), Children is a key-valued ordered list of children nodes (empty list for leaves), and Worth is a score-like integer which is the current *value* of the node.

The *worth* of a node is initially just the value of the game position it is assigned, but it will change after the node has generated its children: once a subtree is generated, the worth of the leaf nodes bubbles up the tree naturally, according to the following rules:

- If a parent node generates a child node which is surely a losing position (for the child node's player), then the parent node is surely a winning position (for the parent node's player). Once this child node is found, no more child nodes need be generated any longer for this parent node. See `build_children/3` and `recurse_children`

- The parent node is assigned the worth of the best of his children.[3] Before this, the children are ordered by worth. To ensure repeated bot games don't play out the same way, the children are clumped by worth, randomized within each clump, then put back together. See `reorder_clumps/2`
- If a parent node generates a child node which is surely a winning position (for the child node's player), then the parent node's player should not play the move leading to said child node position. This means the child node can be effectively discarded. See `recurse_children_loop/5`. [4]

The Bot generates a tree every time the other player makes a move (see `analyze_tree/3`, called by the game loop). The search is *mostly* depth-first: the Bot starts by creating children for every move available within a certain subboard of the Board which surrounds the active area of the Board (the one which has pieces), done with `build_children/3`. Then these children get filtered (according to option depth) and depth-first recursion follows for each good child.

Motivation

Apart from the obvious, with depth 2 or higher – and with reasonably high width values – this helps the bot answer simple forcing moves like

```
W W W W -
```

and with depth 4 or higher, might help the bot answer more advanced threats such as

```
- - W W W - -
```

or create multiple ones like these itself.

Conclusion

This project required a lot of work, but was very rewarding. We achieved everything we stipulated from the beginning, and some more. In our opinion we took what was asked for and elevated it to another level giving the player a lot more options to customize both the bot and the game.

The biggest difficulties when developing this game was, definitely, the making of the Bot's lookahead evaluation tree. It has some faults (for one it doesn't resign), but we think we achieved the goals with this and are pretty happy with how it turned out.

One thing that we would like to improve are the scores of each pattern, in our opinion these aren't perfectly tuned and need some serious work.

[3] Remember White wants positive score/worth, Black wants negative score/worth

[4] Some care must be taken to ensure the root will have at least one child.

References

[1] Wikipedia. *Pente*. URL. (Visited on 10/17/2018).

[2] winning-moves.com. *Pente – The Classic GAME of Capture and 5-in-a-Row*. URL. (Visited on 10/18/2018).

[3] www.pente.net. *How to play the game of Pente*. URL. (Visited on 10/17/2018).

[4] www.pente.org. *Pente Game Rules*. URL. (Visited on 10/17/2018).

[5] www.renju.nu. *Rules of Pente, Keryu-Pente and Ninuki-Renju*. URL. (Visited on 10/17/2018).