

CUHK CSCI3230

Prolog Programming Assignment_{v2} - Go Program

Due date: 23:59:59 (GMT +08:00), **11th** November, 2013

Introduction

Go is a two-player board game played on a grid. One player owns the white playing pieces (i.e. the white stones) and another one owns the black playing pieces (i.e. the black stones). They alternatively place black and white stones on the vacant intersections. The stone will be lifted when being captured by the opponent. The winner of the game is the one who can conquer a larger total area of the board (i.e. territory) than the opponent.

You are going to implement a simplified *Go player program*. This program can deduce a good move from the current state of the board. Computationally, Go is considered a difficult AI problem. Therefore, we modify the rules of the original Go in the assignment. And we apply heuristics: instead of finding the best move in the current state of the game board, your program finds the best move within a bounded number of steps.

Requirement

In this assignment, you are required to implement Prolog predicates:

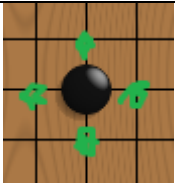
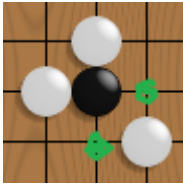
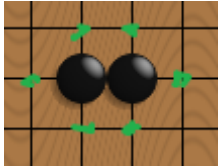
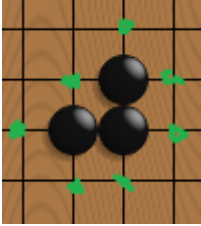
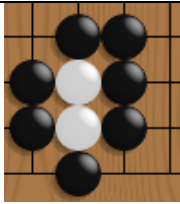

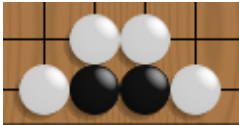
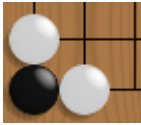
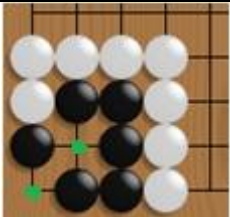
`life_and_death(Board,LastMove,LifeAndDeathBoard).`

`find_good_move(WhoMoveFirst,Depth,Board,BestMove,ScoreOfBestMove).`

Basic concepts

A stone remains on the board when it has at least one “liberty” (Chinese: 氣). This stone is said to be “alive” (Chinese: 生). A liberty is the number of vacant intersection points next to a stone. Two stones are said to be connected if they are adjacent to each other and they form a group. Stones in the same group can share their liberties. Some basic examples of stones pattern are illustrated in Table 1. At the beginning of each turn, all the stones are alive. A stone is said to be captured if it does not have any liberty.

Table 1. A list of examples to illustrate how to count liberties.

 <p>Each stone without any stones next to it has four liberties (marked in green).</p>	 <p>The black stone has only two liberties left.</p>	 <p>The two black stones are adjacent to each other so they form a group. They share the six liberties.</p>
 <p>Three black stones form a group and share the seven liberties.</p>	 <p>The six liberties of white stones are occupied by the black stones, i.e. the white stones have no liberty left. The white stones will be removed.</p>	 <p>The group of white stones has only one liberty left.</p>
 <p>The four liberties of the group of black stones are occupied by the white stones, i.e. the black stones have no liberty left so the black stones will be removed.</p>	 <p>The black stone at the corner originally has two liberties. After placing the white stones, no more liberties for it so it will be removed.</p>	 <p>There are two groups of black stones and they share two liberties. Note that white stones can never be placed in the two liberties.</p>

Board and move representation

Given a rectangular grid, the stones are placed in the intersections. Each intersection will be in one of the four states.

- b: a black stone on the intersection.
- w: a white stone on the intersection.
- m: a vacant space that the players can place a stone.
- s: a vacant space that both players cannot place a stone.

Whether a vacant space is in state m or state s will be set in the program input. Besides, we use a coordinate system (Top,Left) as shown in fig. 1.

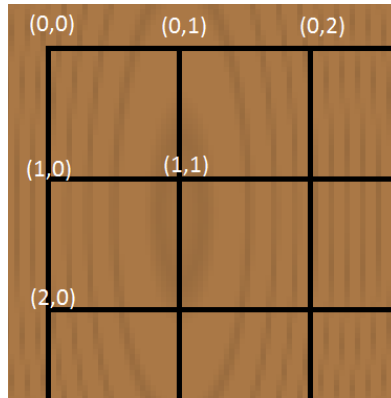


Figure 1. Coordinate system of the board.

With the coordinate system, we can define a move precisely. We will use a compound term $move(X,Y)$ to represent a move in the board. For example, if a player place a stone on position (1,1), we represent it by $move(1,1)$. Note that the actual type of the stone depends on who is the first player.

From the state of board, we can derive the life and death board. Each intersection will be in one of the five states.

- bL: a black stone on the intersection which has liberty.
- wL: a white stone on the intersection which has liberty.
- bD: a black stone on the intersection which does not have liberty.
- wD: a white stone on the intersection which does not have liberty.
- aL: a vacant space.

To obtain the life and death board, we use the following predicate:

`life_and_death(Board,LastMove,LifeAndDeathBoard).`

, where the *Board* is a list of lists representing the actual board, *LastMove* is a compound term of *move* storing the last stone placed on the board, and *LifeAndDeathBoard* stores the life and death board. The predicate requires the *LastMove* because of rule 3.

Rules

We use a simplified set of rules for the Go game.

1. Black and White must make a move in every turn.
2. A move consists of placing one stone of one's own color on a vacant space (in state m) on the board.
3. A stone or solidly *via grid lines* connected group of stones of one color is captured and removed from the board when all the intersections directly adjacent to it are occupied by the enemy. *Capture of the enemy takes precedence over self-capture.*
4. The game ends when recreating a former board state.

5. A player's territory consists of all the *board* points he has occupied.
6. The player with more territory wins.

Those who know the actual rules of Go should note that players are not allowed to pass a move, have extra constraints on where to place the stones (rule 2), and the difference in counting the territory of a player (rule 5).

Heuristics to find a good move

Because of limited computation power, the program will determine the best move by examining the next D moves. Each move is evaluated recursively using the *ScoreMove* and *Score* functions in Algorithm 1. If there are multiple good moves with the same score, the program will only return **ONE** of the best moves.

Algorithm 1

ScoreMove(P,BoardOrigin,Board,Move)

Output: The score of the move
 BoardAfter = The Board after executing Move by player P
 C0=Count all the stones of player P in BoardOrigin
 C1=Count all the stones of player P in BoardAfter
 C2=Count all the stones of opponent of player P in BoardOrigin
 C3=Count all the stones of opponent of player P in BoardAfter
 Return (C1-C0)+(C2-C3)

Score(BoardOrigin,Board,P,Depth)

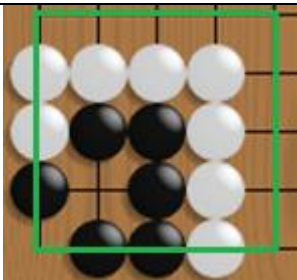
Output: One of the best moves and its score
 If Depth < 0
 Return Error
 If Depth == 0
 Return (nil,+inf)
 E = a set of all valid moves for player P based on the Board
 If Depth==1
 S = {}
 For each move e in E
 Insert ScoreMove(P, BoardOrigin,Board,e) in S
 Find the maximum score S_{\max} in S and the corresponding move M_{best}
 Return (M_{best} , S_{\max})
 Else
 NextPlayer = Opponent of player P
 S = {}
 For each move e in E
 TempBoard = Result after putting the stone according the move e on
 Board
 ($M_{\text{opp}}, S_{\text{opp}}$) = Score(BoardOrigin,TempBoard,NextPlayer,Depth-1)
 Insert (e, S_{opp}) in S
 Find the **minimum** score S_{\min} in S and the corresponding move M_{best}
 Return (M_{best} , S_{\min})

We will find the best move using the following predicate

`find_good_move(WhoMoveFirst,Depth,Board,BestMove,ScoreOfBestMove).`

,where *WhoMoveFirst* will be the player making the first move, *Depth* is the search level, *Board* represents the actual Board, *BestMove* is the output storing the a *move(Top,Left)* structure, and *ScoreOfBestMove* is also the output storing the score of this move.

Illustration 1

Actual Board	Board in our representation	Life and death board																																																		
	<table><tr><td>s</td><td>s</td><td>s</td><td>s</td><td>s</td></tr><tr><td>w</td><td>w</td><td>w</td><td>w</td><td>s</td></tr><tr><td>w</td><td>b</td><td>b</td><td>w</td><td>s</td></tr><tr><td>b</td><td>m</td><td>b</td><td>w</td><td>s</td></tr><tr><td>m</td><td>b</td><td>b</td><td>w</td><td>s</td></tr></table>	s	s	s	s	s	w	w	w	w	s	w	b	b	w	s	b	m	b	w	s	m	b	b	w	s	<table><tr><td>aL</td><td>aL</td><td>aL</td><td>aL</td><td>aL</td></tr><tr><td>wL</td><td>wL</td><td>wL</td><td>wL</td><td>aL</td></tr><tr><td>wL</td><td>bL</td><td>bL</td><td>wL</td><td>aL</td></tr><tr><td>bL</td><td>aL</td><td>bL</td><td>wL</td><td>aL</td></tr><tr><td>aL</td><td>bL</td><td>bL</td><td>wL</td><td>aL</td></tr></table>	aL	aL	aL	aL	aL	wL	wL	wL	wL	aL	wL	bL	bL	wL	aL	bL	aL	bL	wL	aL	aL	bL	bL	wL	aL
s	s	s	s	s																																																
w	w	w	w	s																																																
w	b	b	w	s																																																
b	m	b	w	s																																																
m	b	b	w	s																																																
aL	aL	aL	aL	aL																																																
wL	wL	wL	wL	aL																																																
wL	bL	bL	wL	aL																																																
bL	aL	bL	wL	aL																																																
aL	bL	bL	wL	aL																																																

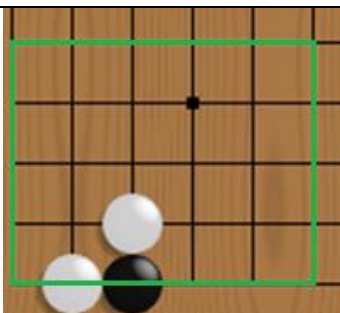
?-

```
life_and_death([[s,s,s,s,s],[w,w,w,w,s],[w,b,b,w,s],[b,m,b,w,s],
[m,b,b,w,s]],move(2,0),LDBoard).
```

LDBoard =

```
[[aL,aL,aL,aL,aL],[wL,wL,wL,wL,aL],[wL,bL,bL,wL,aL],[bL,aL,bL,
wL,aL],[aL,bL,bL,wL,aL]].
```

Illustration 2

Actual Board	Board in our representation																														
	<table><tr><td>s</td><td>s</td><td>s</td><td>s</td><td>s</td><td>s</td></tr><tr><td>m</td><td>m</td><td>m</td><td>m</td><td>m</td><td>s</td></tr><tr><td>m</td><td>m</td><td>m</td><td>m</td><td>m</td><td>s</td></tr><tr><td>m</td><td>m</td><td>w</td><td>m</td><td>m</td><td>s</td></tr><tr><td>m</td><td>w</td><td>b</td><td>m</td><td>m</td><td>s</td></tr></table>	s	s	s	s	s	s	m	m	m	m	m	s	m	m	m	m	m	s	m	m	w	m	m	s	m	w	b	m	m	s
s	s	s	s	s	s																										
m	m	m	m	m	s																										
m	m	m	m	m	s																										
m	m	w	m	m	s																										
m	w	b	m	m	s																										

For the first query, there can be more than one possible answer for a good move. While in the second query, the search depth is increased by 1 and the program deterministically chooses a move at (4,3). The purpose of this move is to avoid the black stone being captured.

?- `find_good_move`

```
(b,1,[[s,s,s,s,s,s],[m,m,m,m,m,s],[m,m,m,m,m,s],[m,m,w,m,m,s],
[m,w,b,m,m,s]],BestMove,Score).
```

```
BestMove = move(1,1) .
```

```
Score = 1 .
```

```
?- find_good_move
```

```
(b,2,[[s,s,s,s,s,s],[m,m,m,m,m,s],[m,m,m,m,m,s],[m,m,w,m,m,s],  
[m,w,b,m,m,s]],BestMove,Score) .
```

```
BestMove = move(4,3) ,
```

```
Score = 0 .
```

Marking Scheme

In the marking, we will use SWI-Prolog. There will be ten test cases and each case contributes 10% of the total marks of the assignment.

Submission

You should submit **a single file *student-id.pl*** through our online system, e.g. if your student ID is 1301234567, the name should be 1301234567.pl. Please limit the file size to be less than 1MB. You can submit multiple times (before the deadline), and the final grade will be the highest mark of all the submissions for this assignment. Make sure that you have **removed all the *unrelated predicates or statements*** in your .pl file before submission. Also, make ensure that your program can return the results for all the test cases in 1 minute in our machine. If your program fails to do so, **zero** mark will be given. If you can complete a 13x13 board in 30 seconds using a machine in the CSE Lab, the running time will be acceptable.

Late submission will lead to marks deduction which is subject to the following penalty scheme.

Number of Days Late	Marks Deduction
1	10%
2	30%
3	60%
4 or more	100%

Plagiarism will be seriously punished.

Extension of the assignment

This part is for students who are interested in extending the current program **after** submitting a workable version of your assignment. You may modify the program so as to make it into real Go player program. You may also use pruning technique in score calculation. It is also good to test your program on the life and death problem in Go game.

References

Rules of Go: http://en.wikipedia.org/wiki/Rules_of_Go

Minimax algorithm: <http://www.stanford.edu/~msirota/soco/minimax.html>

Reference manual of SWI-Prolog: <http://www.swi-prolog.org/pldoc/refman/>

Have fun!