Computing Science (CMPUT) 455

Search, Knowledge, and Simulations

Ting-Han Wei

Department of Computing Science
University of Alberta
tinghan@ualberta.ca

Fall 2020

455 Today - Lecture 2

Topics:

- Assignment 1 preview: Gomoku player
- About Python 3 Go code
- Basic data structures and algorithms for Go Programs
- Algorithms for legal moves, capture, ko, eyes
- Some details on implementation of `Go0` and `Go1` programs

Coursework

- Ongoing coursework:
  - Continue Lecture 1 Activities
  - Do Quiz 0 and Quiz 1
  - Read Krakovsky, Reinforcement Renaissance
- New coursework:
  - Read assignment 1
  - Form teams - see under assignments
  - Do Lecture 2 Activities

Assignment 1 Preview

- Task: implement a random player for the Gomoku (Five in a Row) game based on our Go0 code
- Goals:
  - Understand the code base of the Go0 and Go1 players
  - Modify it to implement a different game
  - Become familiar with Python coding

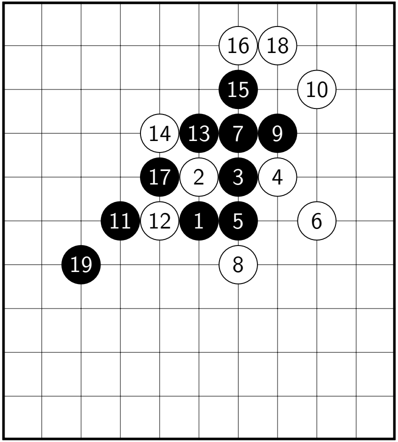
Go0 and Go1 Program Review

- Download program code - part of Activities
- Written in Python 3
- Used to demonstrate basic data structures and algorithms in Go
- Also used as starting point for Assignment 1
- `Go0` plays completely random legal moves
- `Go1` does not fill simple eyes (see last class)

Assignment 1 Starter Code

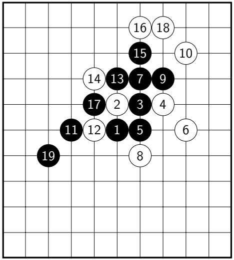
- Download `assignment1.tgz` from assignment page
- Contains copy of `go` directory, for you to modify
- Contains public tests for the assignment

Gomoku or Five in a Row



- Place a stone of your color, as in Go
- First to make 5 or more in a row wins
- Example: Black just won
- Board full, no 5 in a row: draw
- Differences to Go
  - Completely different win condition
  - No capturing, suicide, ko

Assignment 1: Random Gomoku Player



Your computer player should:

- Place a stone of your color on a random empty point
- Recognize the end of the game:
  - One side made 5 or more in a row
  - The board is full, nobody won
- Start from `Go0` sample code
- Implement some GTP commands related to Gomoku rules
- Details in the Assignment 1 specs

Organization of Go Code

- All Python code on course web page
- All Go programs in `go` directory
- Implementation of `Go0` and `Go1` in Python code files in `go`
  - Utility functions shared by all Go programs
  - Simple Go board
  - `Go0` and `Go1` players

Board and GTP

- `board_util.py`
  constants representing colors, conversion of moves, colors from and to text, list of legal moves
- `board.py`
  simple (and slow) implementation of a Go board, initialize board, checking if move is legal, play move, liberties, simple eye
- `gtp_connection.py`
  GTP connection for a given Go playing engine and Go board - receive and parse commands, call functions of the engine or board to compute replies, format replies, handle errors

Go0 and Go1 Players

- `Go0` - file `Go0.py`
  - `Go0`
    player class, defines its name, version and `get_move` function to generate a move
  - `run`
    Main function creates a board, a Go0 player and a GTP connection
- `Go1`
  - `gtp_connection_go1.py`
    example for how to extend the GTP connection with an extra player-specific command
  - `Go1.py`
    similar to `Go0.py`, but note use of `GtpConnectionGo1` instead of `GtpConnection`

Implementing a Go Board and Go Rules

- Representing the board
- Updating the board after a move
  - Recognize capture
- Checking for legal moves
  - Recognize suicide and repetition (simple ko)

Why Bother with an Efficient Board Representation?

- Most game programs are based on search and simulation
- Billions of moves played and taken back during a game
- Playing strength strongly depends on amount of search
- So, make it as fast as possible
  - Our first Python codes are maybe 100.000 times slower than state of the art
  - Mostly, that is due to algorithms and data structures, not Python...
  - We start simple
  - Later (Lecture 6) we will study more efficient ways

Representing State of a Point

- Three possible states: empty, black or white
- We could use the new-ish Python 3 enumeration type
  `https://docs.python.org/3/library/enum.html`

```
class BoardColor(Enum):
    EMPTY = 0
    BLACK = 1
    WHITE = 2
```

- In current program we just use integer codes for colors

```
EMPTY = 0
BLACK = 1
WHITE = 2
```

Representing the Go Board - 2d Array

- Most direct representation: 2-dimensional array (or Python list)
- Store a point on the board at coordinates `[x][y]` in array
- Sample code fragment in: `go2d.py`

```
MAXSIZE = 7
board = [[EMPTY for x in range(MAXSIZE)]
                for y in range(MAXSIZE)]
print(board)
board[3][4] = BLACK
print(board)
```

Drawbacks of Two-dimensional Array

- Overhead from 2-d address calculation
- Need two variables `(x, y)` to represent a single point
- Often need two computations, for `x` and `y` separately
- Complex checking for boundary cases

```
if x > 0 and y > 0
and x <= MAXSIZE and y <= MAXSIZE
```

- `if` statements introduce conditional branches and slow down execution

Go Board as One-dimensional Array

- Solution: use a simple 1-dimensional array
- From `(x,y)` to single index `p = x + y * MAXSIZE`
- Back from `p` to `x` and `y` by integer division and modulo operators
  - `x = p % MAXSIZE`
  - `y = p // MAXSIZE`

Indices of board points for 7×7:

```
 0  1  2  3  4  5  6    % points on first line
 7  8  9 10 11 12 13    % second line
14 15 16 17 18 19 20    % third line
21 22 23 24 25 26 27    % ...
28 29 30 31 32 33 34
35 36 37 38 39 40 41
42 43 44 45 46 47 48
```

1-d Array Pre-computations

- Can precompute many frequent calculations
  - Lookup tables, e.g. `x = xCoord[p]`
- Frequent operations use simple offset, constant time
  - Go to neighbors and diagonals
  - Check if on border, or has neighbor
  - Many more..

Drawbacks of Simple One-dimensional Array

- Edges of board still needs special case treatment (lots of `if` statements)

```
0  1  2  3  4  5  6
7  8  9 10 11 12 13
```

- Index 6 and 7 are not neighbors...
- There is no neighbor upwards from 4...
- Similar for going down from bottom edge

Solution: Add Padding


```
#  #  #  #  #  #  #  #
#  .  .  .  .  .  .  .
#  .  .  .  .  .  .  .
#  .  .  .  .  .  .  .
#  .  .  .  .  .  .  .
#  .  .  .  .  .  .  .
#  .  .  .  .  .  .  .
#  .  .  .  .  .  .  .
#  #  #  #  #  #  #  #  #
```

Image source:

`https://www.gnu.org/`

`software/gnugo/gnugo_15.html`

- Solution: add extra “padding”
  - Above board
  - Below board
  - Between rows
- Use new "off the board" code for these points: `BORDER = 3`

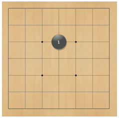
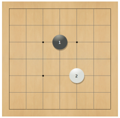
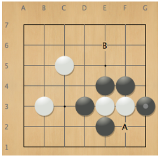
Advantages:

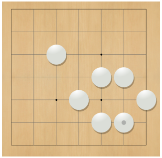
- Neighbors in all 8 directions are valid array indices
- No wraparound to next line
- Off-board recognized by checking `board[p] == BORDER`

Comments for Board Representation

- Standard in Go: 1-d board with extra padding
- Other special purpose representations are possible:
  - Bitsets, one set per color
  - List of stones
  - Cover board with small patterns, e.g. 3×3 squares
    - Will use this as "simple features" later
- Optional resource to learn more:
  `https://chessprogramming.wikispaces.com/Board+Representation`
  detailed discussions for chess
- Next: Playing and Undoing Go moves

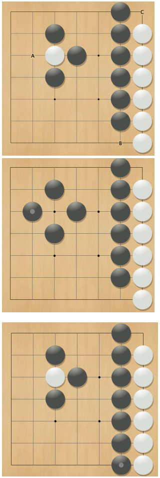
Playing and Undoing Moves



- `play_move(p, color)`
  Put stone of given `color` on point `p`
- Simplest case: just need
  `board[p] = color`
- Major complication:
  recognize captures and remove captured stones
- Closely related to `play_move`:
  check if move on `p` is legal, before playing it...

Capturing Stones



- Which opponent stones are captured?
- Black move A captures one stone
- Black move B does not capture anything...
- To check if B is a capture:
  Must check neighbors of the whole block for liberties
- Must find the liberty at C to decide that B is not a capture

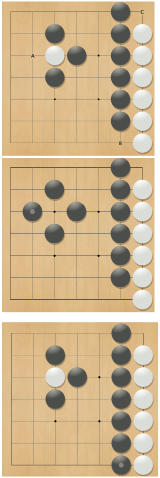
Update Board After a Capture

- For this simple data structure it is easy
- Just change the color of the points

```
for stone in capturedBy(p, color):
    board[stone] = EMPTY
```

- More efficient data structures keep more information, need more updates

Capturing Stones Algorithm



- Which opponent stones are captured?
- Look at all neighbors `nb` of `p` which are stones of opponent
- Check if block of `nb` loses its *last liberty*
- Similar to *floodfill* in graphics, or depth-first search in graph
- Look at all stones connected to `nb`
- If any stone has a liberty (other than `p`), stop: no capture
- If no stone in the block has another liberty, then all are captured

Floodfill Algorithms

- Go board can be viewed as a graph
- Node = intersection of lines on board
- Edge = line segment connecting two neighboring intersections
- How to find connected components in a graph?
- Floodfill algorithms, based on graph search

Example:
`https://en.wikipedia.org/wiki/Flood_fill`

Floodfill Algorithms

Basic ideas

- Keep track of points already visited (e.g. mark them)
- Visit all neighbors
- If they are the right color, then recursively visit their neighbors
- Depth-first search (DFS)
- Different ways to implement
    - Explicit recursion, e.g.
    - Store points to be processed in a stack
- Resources page has some references for your review

Floodfill Application in Go - Blocks of Stones

- Find blocks = connected set of stones
- See code in `simple_board.py`
- Find a block, then check if it has any liberties or should be removed (captured)
- Function `_block_of` implements basic stack-based dfs
- Function `_has_liberty` checks neighbors of block to find liberty
- Question (Activity 2e): is this efficient? Can you think of a faster way?

Implementing Go Rules

- I explained Go rules informally in Lecture 1
- For programming we need a more formal version
- Popular example of minimalistic ruleset:
  Tromp-Taylor rules (next slide)
- Main question in practice:
  check if move is legal

Tromp-Taylor Rules

From `http://tromp.github.io/go.html`

1. Go is played on a 19x19 square grid of points, by two players called Black and White.
2. Each point on the grid may be colored black, white or empty.
3. A point P, not colored C, is said to reach C, if there is a path of (vertically or horizontally) adjacent points of P's color from P to a point of color C.
4. Clearing a color is the process of emptying all points of that color that don't reach empty.
5. Starting with an empty grid, the players alternate turns, starting with Black.
6. A turn is either a pass; or a move that doesn't repeat an earlier grid coloring.

Tromp-Taylor Rules Continued

8. A move consists of coloring an empty point one's own color; then clearing the opponent color, and then clearing one's own color.
9. The game ends after two consecutive passes.
10. A player's score is the number of points of her color, plus the number of empty points that reach only her color.
11. The player with the higher score at the end of the game is the winner. Equal scores result in a tie.

Comments:

- Compare the "reach" definition in point 3 with floodfill.
- These rules allow suicide (why?). It is a bit more complex to write formal rules that forbid it.

Checking If Move is Legal

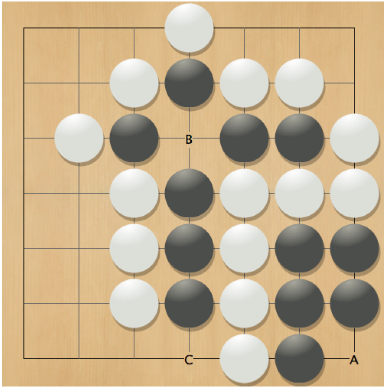
Check three conditions:

`isLegal(p, color):`

1. `board[p] == EMPTY`
2. `not isSuicide(p, color)`
3. `not repetition(p, color)`

Remark: in our program, we call `play_move` on a copy of the board. It makes the same checks and returns a boolean.

Checking Suicide



- Very similar to checking capture for the other color
- Main difference: the move can connect several blocks, and none of them may have another liberty
- See examples: Black A is suicide, Black B is not because liberty at C

Checking Suicide in Go0

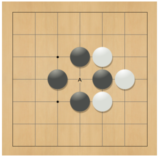
In function `play_move`:

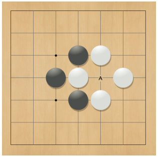
```
block = self._block_of(point)
if not self._has_liberty(block): # undo suicide
    self.board[point] = EMPTY
    return False
```

Checking Repetition

- Repeating same board position is illegal
- Naive check is very expensive:
  - Keep record of all previous positions
  - Compare with current position point for point
- Can be done much faster (Lecture 6)
- Think about how you would optimize it
- `go` code checks only the most frequent case: simple ko (next slide)

Checking Simple Ko Repetition





- After capture of a single stone `s`:
- set `ko_recapture = s`
- After any other move:
  set `ko_recapture = None`
- If `p == ko_recapture`
  and
  "`p` would capture a single stone":
- Then `p` is illegal
- Details in function `play_move` near the end

Undo, Taking Back Moves

- For search, need to consider many alternative moves
- Need undo: take back move before trying another
- Main problem: deal with captured stones
- How to implement undo?
- Two basic approaches
  - Copy-and-modify
  - Incremental with change stack
- Note: `Go0` and `Go1` do NOT implement undo

Undo With Copy-and-modify

- For each move:
  - copy the board
  - modify the copy
  - make the copy the new board
- Keep a stack of all boards, one per position
- To undo a move, simply pop the top board from stack, use the previous one
- Pro: simple to implement, simple data copies are fast on modern hardware
- Con: uses much memory, lots of copying state

Change Stack

- Single Go board, plus a stack
- At start of each move, `push` a special marker onto stack
- Record each change: store old value on stack
- Example:
  - `board[43]` was `BLACK` before capture
  - push `(43, BLACK)` onto stack
  - Then change the board, e.g. `board[43] = EMPTY`

Incremental Undo with Change Stack

- To undo a move:
- Restore old values recorded on stack
- Stop when reaching the special marker
- Example:
  - `pop()` returns `(43, BLACK)`
  - Restore old board state, `board[43] = BLACK`
- Pro: no copying, minimal number of operations
- Con: more work to implement correctly

Summary and Outlook

- Discussed most of the basics of implementing Go
- Go board data structure, padded 1d array
- Checking legal moves, playing and undo
- Next time: start discussing human decision-making