Computing Science (CMPUT) 455

Search, Knowledge, and Simulations

Martin Müller

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

Fall 2022

455 Today - Lecture 2

Topics:

- How to get from `Go0` to `Go1`?
  - Recognizing and not filling *eyes*
  - See Lecture 1 slides 59-68
- Assignment 1 preview: random NoGo 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

Old coursework:

- Read the course syllabus
- Register for exams with LAC - see links on eClass
  - Note - there **will still be classes** in midterm week. Try to avoid scheduling your exam during class time

New coursework:

- Read assignment 1
- Form teams - see under assignments
- Do Lecture 2 Activities
- **Especially - install GoGui** and try out `Go0` and `Go1`
- Try it by yourself first. Need help? See install GoGui on eClass forum

Assignment 1 Preview

- Task: implement a random player for the game NoGo 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
- **New code base this year. Your solution must be based on our current Go0 code base**

Assignment 1 -
Lessons Learned in Previous Years

- Lesson 1: Testing is Key
- In my humble opinion, 99% of programmers do not test enough
- That includes grad students and professors...
- Test everything
- Test after each change
- Use the tools we provide
- Re-check the spec against what your program does

Test your Submission Before you Submit

- Do the same steps we will do
- Check if everything is according to the specification: names, directory structure, contents
- Unpack in a new directory
- Run the public test cases with this code

More Lessons Learned

- Lesson 2: manage your time
- I think that 90% do not manage their time properly
- That includes professors and grad students...
- No matter what the deadline, and how much time there is, the number of questions grows exponentially in the last two days
- The last two days are the

  worst possible time

  to do your work. Stress for all.
- Pace yourself. Manage your time. Do things early.
  - It will be **seriously good for you**.

Lessons Learned - Polya's Principles

- We will soon discuss Polya's book *How to Solve It*
- Polya's first principle: **Understand the Problem**
- Read the spec. Implement it. Completely and precisely.
- At the end, read specs again and **review your solution** (Polya's principle 4)
- If you do not completely understand all of the problem:
  - Ask for clarification
  - The sooner the better
- To really understand details and possible obstacles:
  - You need to engage and code and work with the material
  - Just reading the spec will not be enough
- Time management -
  engage early to leave enough time for these activities

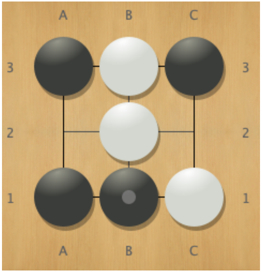
Go0 and Go1 Program Review

- 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

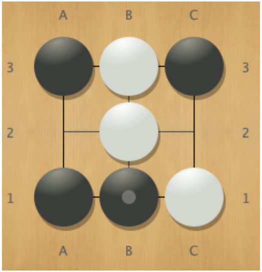
- Download `assignment1.tgz` from assignment page
- Contains an `assignment1` directory, for you to modify
- Contains public tests for the assignment

The Game of NoGo



- Same as in Go:
  - Place a stone of your color on an empty point
  - Suicide is illegal
- Differences to Go:
  - Capturing is also illegal
  - Passing is also illegal
  - Completely different win condition:
  - The last player who can make a move wins.
  - If it is your turn and you cannot move, you lose.
  - Example on left: White lost (why?)

Assignment 1: Write a Random NoGo Player



Your computer player should:

- Place a stone of your color as a random legal move **according to the rules of NoGo**
- Recognize the end of the game:
  - Current player has **no** legal move
- Start from `Go0` sample code
  - Only a few changes needed
- Implement some GTP commands related to NoGo rules
- Details in the Assignment 1 specs

Lecture 2: Data Structures and Implementations for Simple Go Programs

Organization of Go and Other Code

- Page with descriptions and links to all Go programs: `https://webdocs.cs.ualberta.ca/~mmueller/courses/cmput455/html/Go-programs.html`
- Most programs just add code to the previous one
- Some exceptions, for example Go2 changes the board
- All other Python code, organized by lecture: `https://webdocs.cs.ualberta.ca/~mmueller/courses/cmput455/python/`
- This week: look at common implementation of `Go0` and `Go1` programs

Go0 and Go1 Programs

- `Go0` and `Go1` players
- Simple Go board
- Utility functions shared by all Go programs
- Unit tests and other tests
- Main components on next slides
- Full overview of all files on `https://webdocs.cs.ualberta.ca/~mmueller/courses/cmput455/html/Go-programs.html#Go0and1`

Go Board

- `board_base.py`
  Constants representing colors, other basic definitions, conversion of moves, colors from and to text, list of legal moves. Used by `board.py` and throughout all our Go codes
- `board.py`
  simple (and slow) implementation of a `GoBoard` class. Initialize empty board, check if move is legal, play move, compute blocks, liberties, captures, suicide, check if point is simple eye
- `board_util.py`
  board-related utility functions

Go Text Protocol (GTP)

`gtp_connection.py`

- Implements GTP connection for a Go playing engine or Go board
- Receive and parse commands from user or script or UI
- Call functions of the engine or board: compute replies, format replies
- Also handles errors

Go0 and Go1 Players

- `Go0` player - 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` player - two files
  - `gtp_connection_go1.py`
    example for how to extend the GTP connection with an extra player-specific command
  - `Go1.py`
    similar to `Go0.py`
    - Filters out eye-filling moves
    - Uses `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...
  - Start with simple code
  - Later (Lecture 6) we will study more efficient ways

Representing the State of a Point

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

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

- In our programs we just use integer codes for colors
- Defined in board_base.py

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

Representing the Go Board - 2d Array

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

```
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 2D 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
```

1D 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`

Branch Prediction

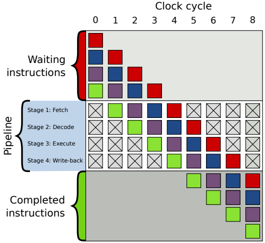


Image source:

`https://en.wikipedia.org/wiki/Branch_predictor`

- Modern processors use a pipelining architecture
- Earlier phases of later instructions are executed simultaneously with later phases of earlier instructions
- When a conditional branch is encountered, processor guesses whether it will be taken
- When it guesses wrong, all of the progress on later instructions has to be thrown away

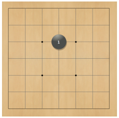
Branch Prediction Examples - "if" in board.py

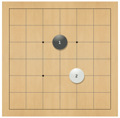
```
def _has_liberty(self, block: np.ndarray) -> bool:
    """
    Check if the given block has any liberty.
    block is a numpy boolean array
    """
    for stone in where1d(block):
        empty_nbs = self.neighbors_of_color(stone, EMPTY)
        if empty_nbs:
            return True
    return False

def neighbors_of_color(self, point: GO_POINT, color: GO_
    """ List of neighbors of point of given color """
    nbc: List[GO_POINT] = []
    for nb in self._neighbors(point):
        if self.get_color(nb) == color:
            nbc.append(nb)
    return nbc
```

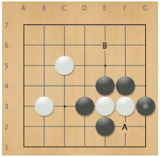
Comments for Board Representation

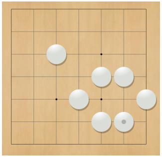
- Standard in Go: 1D board with extra padding
- Other special purpose representations are possible:
  - Bitsets, one set per color
  - List of stones
  - Cover board with small patterns, e.g. 3×3 squares
    - Will use this to implement “simple features” later
- Optional resource to learn more: `https://www.chessprogramming.org/Board_Representation` detailed discussions for chess
- Next: Playing and Undoing Go moves

Playing a Move





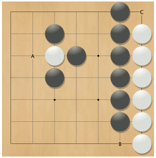


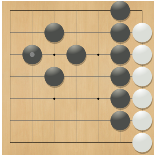


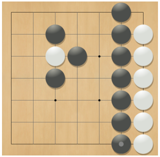
In `board.py`:

- `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
- `is_legal(p, color)`:
  - Check if a move on `p` is legal, before playing it..
  - Closely related to `play_move`

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

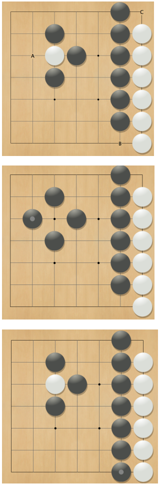
Update Board After a Capture

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

```
board[captures] = 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 = point = intersection of lines on board
- Edge = line between two neighboring points
- 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. `dfs(p)` calls `dfs(neighbor)`
  - 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 `board.py`
- Find a block, then check if it has any liberties or should be removed (captured)
- Function `connected_component` implements basic stack-based dfs
- Function `_has_liberty` checks empty neighbors of block to find liberty
- Question - Activity 2e: is this code efficient? No? 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 a 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 a Move is Legal

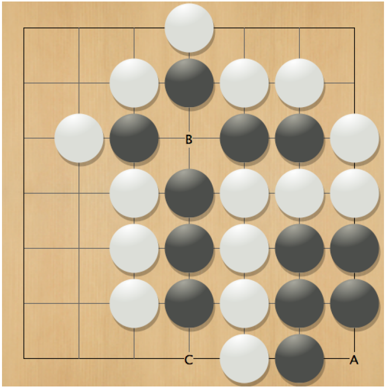
Check three conditions:

`isLegal(p, color):`

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

- Remark: in our `Go1` program, we call `play_move` on a copy of the board. It makes the same checks and returns a boolean.
- Remark 2: this is slow!

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 of liberty at C

Checking Suicide in Go0

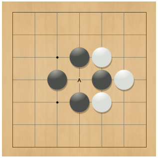
- Part of function `play_move`
- First, we put the stone on the board
- Next, we process captures
- Next, we compute the block that the stone is part of
- We check if the block has any liberties
- If not, we remove it, and the move is suicide (illegal)
- If has liberties, we continue other steps in `play_move`

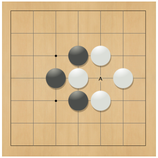
```
block = self._block_of(point)
if not self._has_liberty(block):
    # undo suicide move
    self.board[point] = EMPTY
    return False
```

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
- Our 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

- **NOT implemented** in `Go0` and `Go1`
- Search considers many alternative moves
- Need undo: take back one move before trying another
- Main problem: deal with captured stones
- How to implement undo?
- Two basic approaches
  - Copy-and-modify board
  - Incremental with change stack

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
  - Remove 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

Change Stack

- Single Go board, plus a stack
- No copying of whole boards
- 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 `MARKER`
- Example:
  - `pop()` returns `(43, BLACK)`
    - Restore old board state, `board[43] = BLACK`
  - Next `pop()` returns `MARKER`
    - Done with undo, all changes have been reversed
- Pro: no copying, minimal number of operations
- Cons: more work to implement correctly,
  more branching in code

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