Computing Science (CMPUT) 455

Search, Knowledge, and Simulations

James Wright

Department of Computing Science
University of Alberta
james.wright@ualberta.ca

Fall 2021

455 Today - Lecture 13

- Today's topic: Understanding and improving simulations in Go
- Almost-random simulations in the Go3 program
- Strengths and weaknesses
- Bias and variance
- Improvements: filtering and selective move policies
- Rules and patterns, MoGo patterns

Coursework

- Assignment 2:
  - Resubmission (with 20% penalty) was due last night
  - Marks by next week
- Reading: Rémi Coulom, *Computing Elo Ratings of Move Patterns in the Game of Go.*
- Quiz 7: Simulations
  - due Monday (Oct 25)

Scaling of Go3 vs Go2

- Board size 5 $\times$ 5, komi 4.5
- Simulations/move: 10, 20, 50, 100
- Opponent: Go2 random player
- Go3 clearly better than random

Player	Wins %
Sim(10)	98% ($\pm$1.4)
Sim(20)	100% ($\pm$0)
Sim(50)	100% ($\pm$0)
Sim(100)	100% ($\pm$0)

Simulation Speed in Go Revisited

- Example: 7×7 Go
- Playing one move in simulation is more complex in Go than TicTacToe
  - Simulation can be longer than 50 moves
- Many more legal moves than TicTacToe (almost 50 at start of game)
- Example: 100 simulations/move: total
  $100 \times 50 \times 50 = 250,000$ simulated moves *for making a single move decision in opening*
- To play whole 7×7 game:
  many millions of simulated moves

Speed comparison: Go3 vs Fuego

- Speed comparison: 5x5 Go, empty board
- Fuego: `http://fuego.sourceforge.net`
  open source C++ code
- Go3, almost-random simulations
  - Speed: $\approx$ 120 sim. / second
- Go3, rule-based simulations
  - Speed: $\approx$ 30-40 sim. / second
- Fuego MCTS, rule-based simulations plus tree search
  - Speed: $\approx$ 10,000 sim. / second
  - About half of that time spent on simulations
- Go3 simulations are about 200 - 500 times slower

Uniformly Random Simulations in Go - Strengths and Weaknesses

- Last class: simulation-based player `Go3`
- How well does `Go3` play Go?

Uniformly Random Simulations in Go - Strengths and Weaknesses

- Last class: simulation-based player `Go3`
- How well does `Go3` play Go?

Strengths:

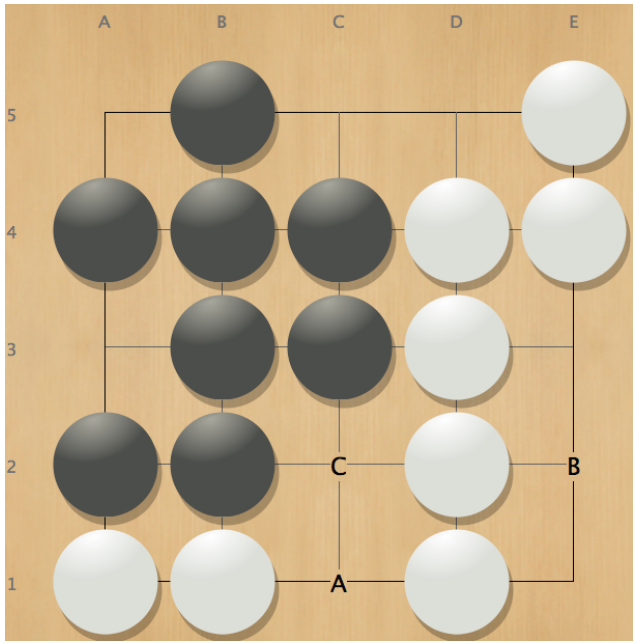
- Can find many simple tactics
- Can find sure wins near the end
- Can avoid many simple blunders that the random player may play
- Improves with number of simulations, up to a limit

Uniformly Random Simulations in Go - Strengths and Weaknesses

Weaknesses:

- Main weakness:
  assumes that the opponent plays randomly...
- It will play the moves that work best against random
- Those moves may fail against strong opponents
- Example: make a “silly threat”
  - A strong opponent will answer the threat, no gain
  - A random opponent will only answer with small probability
    - Effect: the threat looks good in simulation

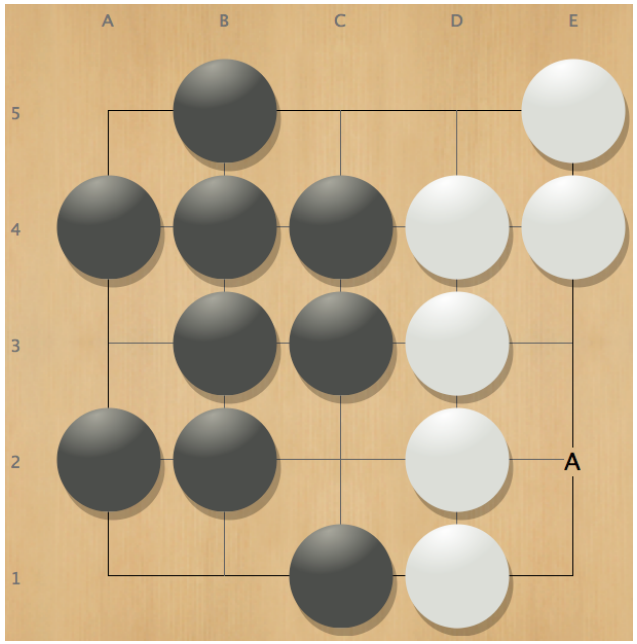
Uniformly Random Simulations in Go - Example for Strength



Win rates:
(e2, 1.0),
(c2, 0.75),
(c1, 0.72), ...

- Black to play, 5.5 komi
- e2 is only winning move - kills whole white group
- c1 captures two stones, but is not enough to win
- With 100 simulations, Go3 clearly finds e2 is best
- Weakness: program also thinks that bad moves c1 and c2 win
  - It does not matter in practice since e2 is ranked higher

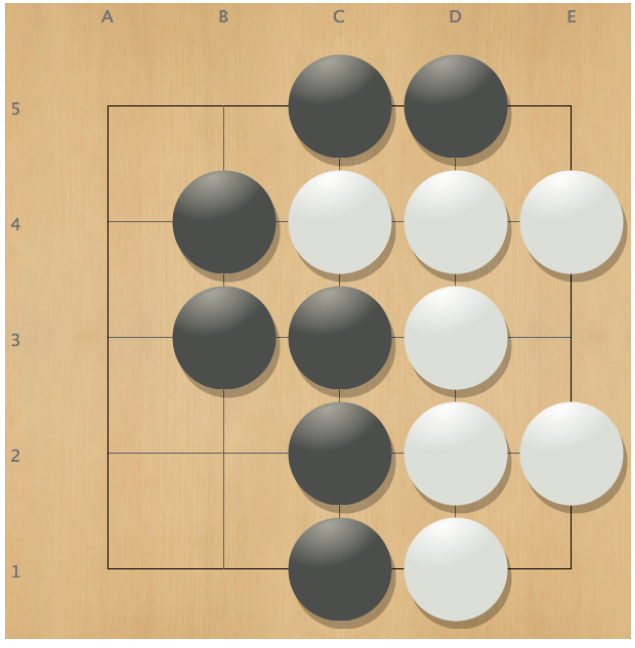
Uniformly Random Simulations in Go - Example Continued



Win rates:
(e2, 0.98),
(a1, 0.36),
(c5, 0.36),...

- Position after black mistake playing c1
- FlatMC now finds a win for White very clearly
- Simulations find that making two eyes leads to win for White
- Emergent good behavior, achieved without any special knowledge of how to make eyes
- The simulations show the value of this move

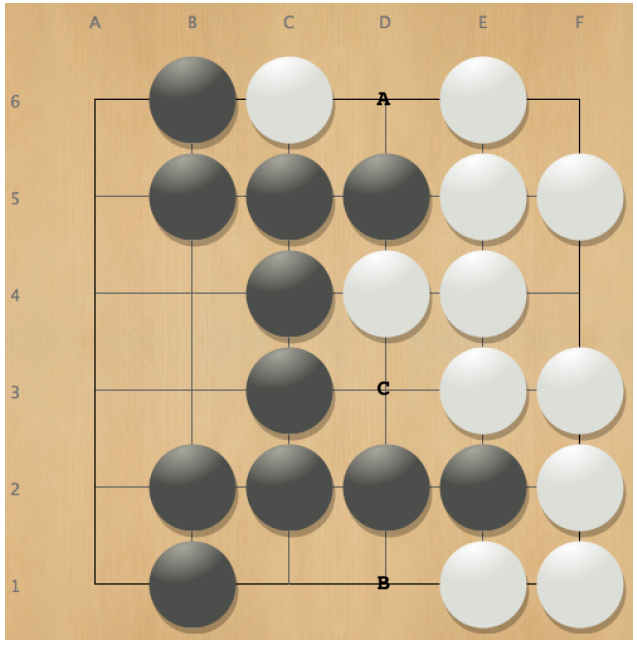
Uniformly Random Simulations in Go - Example 3 - strength



Win rates:
(b5, 0.78),
(a3, 0.55),...
(e5, 0.29).

- Black to play, 3.5 komi
- e5 is blunder, “self atari” endangers three stones
- FlatMC can clearly see that e5 is worse than other moves
- Winrate goes down since these stones are often captured in simulations
- Almost any other move wins (b5 is best)

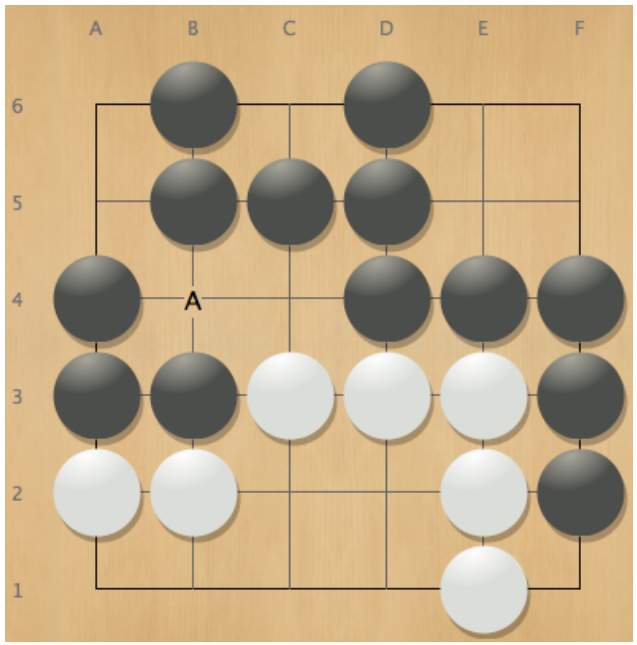
Uniformly Random Simulations in Go - Example 4 - strength



Win rates:
(d6, 0.86),
(d1, 0.79),
(d3, 0.72),...

- d6 is worth 2 points territory (plus the stone played)
- d1 is worth 1 extra point
- d3 is neutral, no extra points
- Strength: program gets correct ordering of move values
  $v(d6) > v(d1) > v(d3)$
- Weakness: difference in winrates is not large...
  - Lots of random noise from plays after the first move

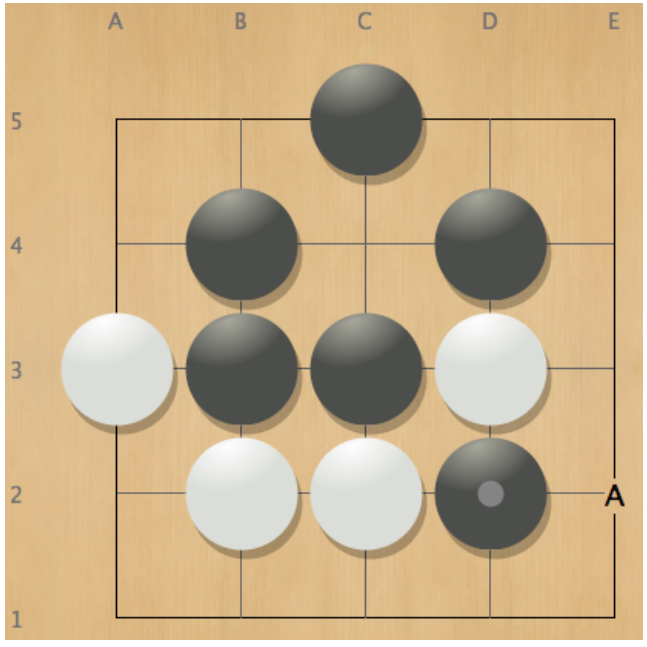
Uniformly Random Simulations in Go - Example 5 - Weakness



Win rates:
(b4, 0.35),
(a5, 0.24),...

- White to play, 1.5 komi
- b4 and a5 are silly threats, bad moves, lose points
- Simulation player likes them best
- Against the random opponent in the simulations,
  these moves often work
- The quiet move at c4 would win
- However, its simulation winrate is low
  - Why? My guess is that the white corner group often dies in simulation

Uniformly Random Simulations in Go - Example 6 - Weakness



Win rates:
(e2, 0.34),
(d1, 0.32),
(e3, 0.3),...

- Simulations are over-optimistic here
- White totally destroyed
- White still has $>$ 30 % winrate in random simulation
- White sometimes captures the black stone d2 in the simulations and wins
- e2, d1, e3 are all threats to capture d2 that:
  - Fail against a competent opponent
  - Can work against random

Ideal Simulation vs Reality

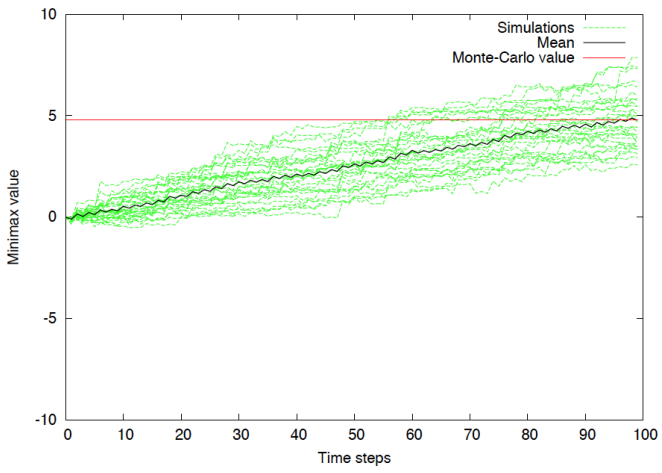
- An *ideal simulation* would preserve wins and losses exactly
  - If starting position is a win with best play by both:
    $\rightarrow$ simulation is also a win
  - If starting position is a loss
    $\rightarrow$ simulation is also a loss
- This would imply:
  - Each simulation is a perfect game
  - Simulation gives a perfect evaluation function
- That is VERY unrealistic!
  - If we know how to play a game perfectly,
    we do not need any search or simulation...
- What are properties of *good* simulations in practice?

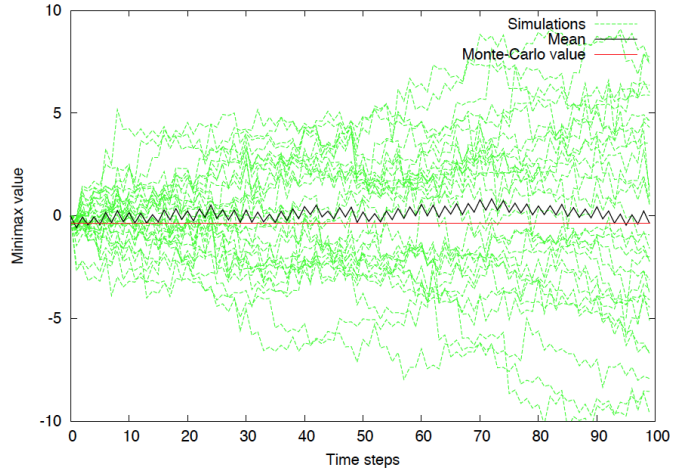
Real Simulations - Mean, Bias and Variance

- Evaluate a state by doing n simulations
- Simulation i outcome is win ($s_i = 1$) or loss ($s_i = 0$)
- Evaluation = winrate (mean outcome) of simulations
- Mean outcome μ of the simulations
  - $\mu = \frac{\sum s_i}{n}$
- Variance
  - $var = \frac{\sum (s_i - \mu)^2}{n}$
- Ideal playout: mean = minimax score, variance = 0
- Real playout: mean $\neq$ minimax score, variance > 0
- Bias = Difference between mean and minimax score m
  - bias = $|m - \mu|$

Bias vs Variance - What is More Important?

- Silver and Tesauro 2009: low bias is much more important for good Monte Carlo simulations
- High bias: drift, result gets worse with more time steps in simulation (left picture)
- High variance can be addressed by doing more simulations (right picture)





Improving Simulations

- Random simulations make many mistakes
- Random simulations have some systematic weaknesses, causing bias and variance
- How to improve them?
- Answer from early Go program MoGo:
  add game-specific filters and rules
- One way is to hardcode “obvious local replies”
- Goal: make the simulation more stable
  - Safe stones and territories should remain safe for the rest of the simulation

The MoGo Program

- MoGo was one of the first successful Monte Carlo Go programs
  - Developed around 2007/2008
  - Won many competitions
  - Won (high handicap) games vs human professionals
- Two main reasons for success:
  - First Go program to use the UCT algorithm for Monte Carlo Tree Search (later in this course)
  - Improved playout policy by simple tactical rules and 3×3 patterns (this class)

Improving Go3 Almost-Random Policy

- Go3 contains several ways to change the simulations
- Improve on almost-random default simulations in Go3
- Filter:
  - Avoid some huge blunders in simulated games
  - Examples:
    - Do not fill eyes (already in Go3)
    - New: avoid “selfatari”
- Selective Move Policies:
  - Find *promising* moves based on rules
  - If promising moves exist:
    ignore all other moves

Details on Filtering

- A filter decides whether a move should be used or not
- We have seen a simple filter already
- Random players `Go1`, `Go2` filter one point eyes
- The `Go3` default simulation policy also filters these eye-filling moves
- We can filter other bad moves, too
- How does filtering work?

Simplest Filtering Algorithm

- Here, `filter` can be any move-checking function
- A checking function returns a boolean result - should a move be filtered?
- Examples: eye-filling, selfatari, ...

```
def naiveGenerateMoveWithFilter():
    moves = generate_moves()
    for move in moves:
        if filter(move):
            remove move from moves
    if len(moves) > 0:
        return random.choice(moves)
    else:
        return None
```

More Efficient: Lazy Filtering

- Simple algorithm is inefficient
- Calls `filter` for all legal moves on the whole board
- Much faster way: pick random move first, then check

```
def filterMovesAndGenerate(moves):
    while len(moves) > 0:
        candidate = random.choice(moves)
        if filter(candidate):
            remove candidate from moves
        else:
            return candidate
    return None

def generateMoveWithFilter():
    moves = generate_moves()
    return filterMovesAndGenerate(moves)
```

Filtering - How to Efficiently Remove a Move

- `moves` stored in Python list or array
- How to remove move at some index `i`?
- Slow way: move all elements `i+1, i+2,...` down by one
- Much faster way: replace `moves[i]` by last element

  ```
  moves[i] = moves[-1]
  moves.pop()
  ```

- Note: much faster but changes order of list items

```
>>> moves = [1,2,3,4,5,6,7,8,9]
>>> i=2
>>> moves[i] = moves[-1]
>>> moves.pop()
>>> moves
[1, 2, 9, 4, 5, 6, 7, 8]
```

Filtering - How to Efficiently Remove a Move

- `moves` stored in Python list or array
- How to remove move at some index `i`?
- Slow way: move all elements `i+1, i+2,...` down by one
- Much faster way: replace `moves[i]` by last element

  ```
  moves[i] = moves[-1]
  moves.pop()
  ```

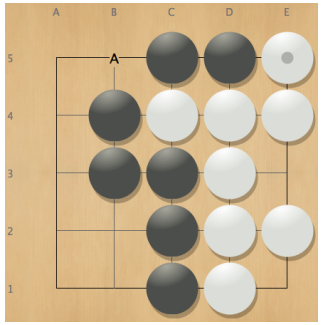
- Note: much faster but changes order of list items
  - Question: does that matter?

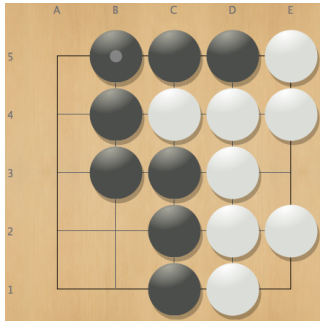
```
>>> moves = [1,2,3,4,5,6,7,8,9]
>>> i=2
>>> moves[i] = moves[-1]
>>> moves.pop()
>>> moves
[1, 2, 9, 4, 5, 6, 7, 8]
```

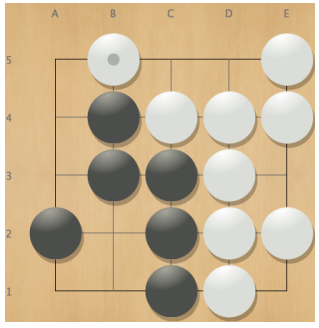
Filters in Go3

- Always: filter eye-filling moves
- Optional: also filter selfatari moves
- Next slides:
  - What is atari and selfatari?
  - Implement the selfatari filter

Atari

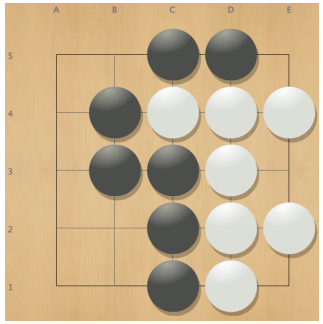


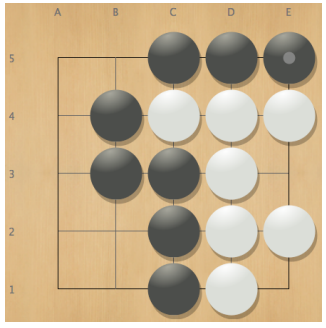


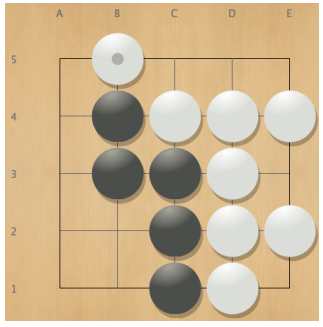


- "Be in atari" means stones have only 1 liberty
- "Give atari" means to reduce the opponent's stones to 1 liberty
- Most direct form of threat in Go
  - First picture: White e5
    "gives atari on Black d5 and c5"
  - d5 and c5 now have only one liberty at b5
  - Second picture:
    To defend, Black connects at b5
  - Third picture: Black played elsewhere, allowing White to capture on b5

Prevent Selfatari

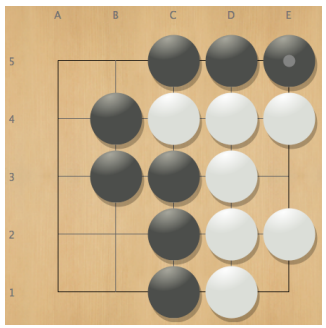


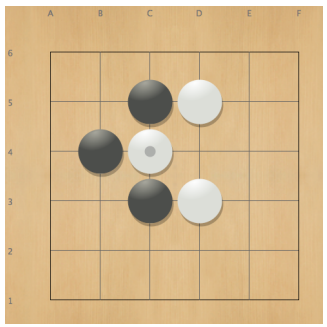




- Selfatari means to reduce your own stones to one liberty
- Most of the time, selfatari is a basic type of blunder
- Stones c5, d5 have two liberties
- Bad move e5 Black takes away one liberty
- Three stones now have only one liberty
- Now White can capture three stones

Implementing Selfatari Filter





Two cases

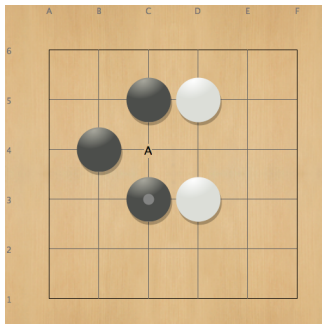
- Case 1, as in example before
  - Existing block with two liberties
  - Own move fills one liberty
- Case 2, single stone selfatari
  - New block, single stone
  - Placed so that it has only one liberty

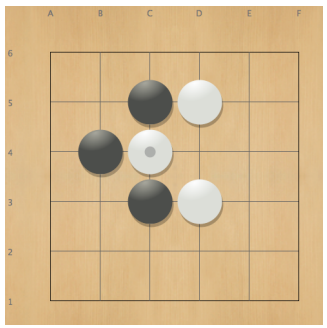
Selfatari Filter for Existing Block

```
def selfatari(block, move):
# move is on a liberty of block
oldNumLiberties = libertyCount(block)
if oldNumLiberties == 2:
    play(move)
    newNumLiberties = libertyCount(block)
    undoMove()
    if newNumLiberties == 1:
        return True
return False
```

- Existing block, play on own liberty
- Would it have only 1 liberty afterwards?

Single Stone Selfatari Filter





- Move creates new block,
  has no neighbors of own color
- Move is not a capture
- Stone has only 1 liberty
- Usually, such moves are very bad,
  only strengthen the opponent
- Sometimes, such a move is a good
  sacrifice

From Filters to Rules and Patterns

- Filters are one way to improve simulations
  - Avoid moves that are typically bad
- We can also go the opposite way:
  - Generate only moves that are typically good, urgent
- Rules and patterns follow this approach

Combining Rules and Filters

- We can use both rules and filters
- Example: rule selects move m, but it is eye-filling
- Still filter such moves
- Call `filterMovesAndGenerate` for list of rule-based moves only
- If one move survives the filter, play it
- If all rule-based moves are filtered:
  - Choose (with filtering) among remaining moves
  - Implementation: see next slide

Rule-Based Randomized Playout

- Heuristic rule selects subset of all legal moves
- Choose randomly among those moves

```
def generateMoveWithRulesAndFilter():
    moves = ruleBasedGenerateMoves()
    move = filterMovesAndGenerate(moves)
    if move != None:
        return move
    moves = generateOtherLegalMoves()
    return filterMovesAndGenerate(moves)
```

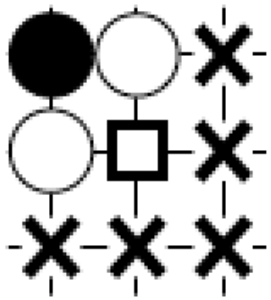
Examples of Rules

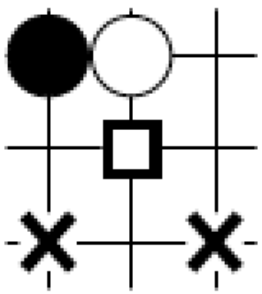
- Atari Capture: Capture *last* opponent stone
- Atari Defense: Save nearby own stones from capture
- Capture *any* opponent stone
- Extend number of liberties
- Play “good shape” pattern
- Many more...

From Rules to Patterns

- So far we looked mostly at tactical filters and rules
- Those were based on liberties of blocks
- Another choice: look at a small local area around move
- Decide which moves are good or bad,
  based on *pattern* of stones nearby
- Example from original MoGo program:
  3×3 and 2×3 patterns

MoGo Patterns - Idea





- Urgent response patterns
- Opponent just played move *m*
- Check empty points *p* near that move
- Up to 8 neighbors and diagonal neighbors
- Check 3×3 area around *p*
- Pattern decides: should the program play *p* with high priority?
- Point labeled with square in middle of pattern: urgent MoGo pattern move

MoGo Patterns, Part 1

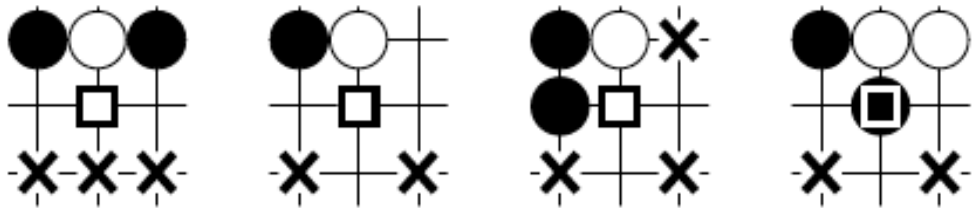


Fig. 5. Patterns for Hane. True is returned if any pattern is matched. In the right one, a square on a black stone means true is returned if and only if the eight positions around are matched and it is black to play.

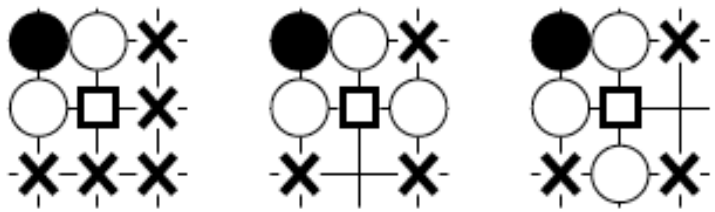


Fig. 6. Patterns for Cut1. The Cut1 Move Pattern consists of three patterns. True is returned when the first pattern is matched and the next two are not matched.

MoGo Patterns, Part 2

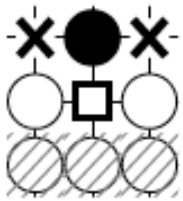


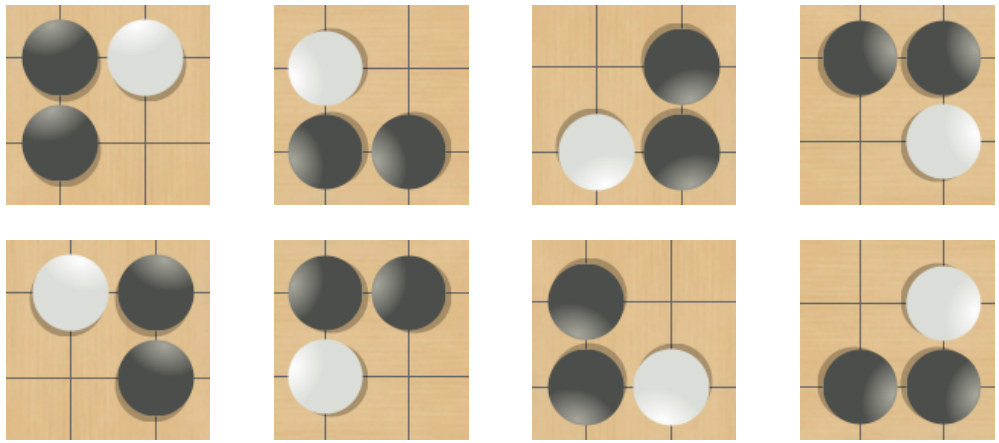
Fig. 7. Pattern for Cut2. True is returned when the 6 upper positions are matched and the 3 bottom positions are not white.

Fig. 8. Patterns for moves on the Go board side. True is returned if any pattern is matched. In the three right ones, a square on a black (resp. white) stone means true is returned if and only if the positions around are matched and it is black (resp. white) to play.

How were MoGo Patterns Found?

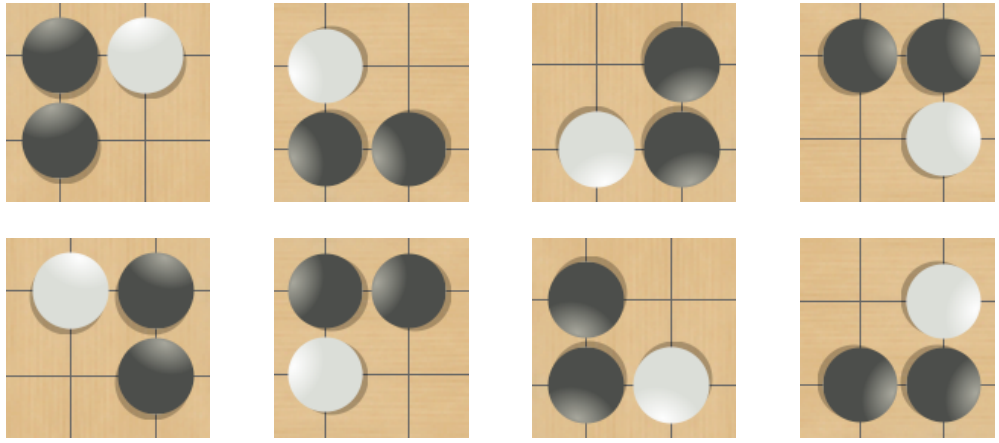
- Why these moves and not others?
- I don't know for sure
- Manual process, probably by trial and error
- Most of these moves are urgent or "stabilizing" local responses
- Soon we will study machine learning to replace the manual process

Dealing with Symmetries



- Four operations can generate symmetrical patterns
- 2 rotations: 0 degrees, 90 degrees
- 2 vertical flips: don’t flip, flip
- 2 horizontal flips: don’t flip, flip
- 2 colors: don’t swap colors, swap colors (not shown)

Dealing with Symmetries



- Total $2 \times 2 \times 2 \times 2 = 16$ possible combinations of operations
- Each choice gives a different symmetry of the same pattern
- Self-symmetric patterns have fewer cases
- Example: pattern can be equal to its flipped version

Programming MoGo-Style Patterns

- Implementation from open source program Michi
- Code in `Go3, util/pattern.py`
- Also see `https://github.com/pasky/michi`
- Table of patterns with different “wildcards”
- Wildcards match more than one state on board - among empty, black, white, off board
- Expand wildcards to all matching 3×3 patterns

Sample Pattern Definitions

```
["XOX",  # hane pattern - enclosing hane
 "...",
 "???"],

["XO.",  # hane pattern - non-cutting hane
 "...",
 "?.?"],
```

Codes:

- `X` = black stone
- `O` = white stone
- `.` = empty
- `?` = any color

Expanding Wildcards Example

- Pattern with wildcard `?` matching any color

```
["?OX",  # side pattern - cut
 "X.O",
 "   "]
```

- Expanded - wildcard replaced by each possible color:

```
"XOX"          "OOX"          ".OX"
"X.O"          "X.O"          "X.O"
"   "          "   "          "   "
```

Expanding Wildcards - Details

- State of point:
  `X` = black stone, `O` = white stone,
  `.` = empty, " " = off board (for edge patterns)
- Wildcards can stand for more than one color
- `?` = "any color": `X`, `O`, `.`
- `x` = "not `X`": `O`, `.`
- `o` = "not `O`": `X`, `.`
- `pat3_expand`:
  - Expand all wildcards in a pattern
  - Create list of full patterns without wildcards
- Example on previous slide

MoGo Style Simulation Policy

- Use `generateMoveWithRulesAndFilter()` from a few slides ago, with pattern-based filter
- `ruleBasedGenerateMoves()` checks 3x3 patterns nearby
- `filter()` checks selfatari *for pattern moves only*

```
def ruleBasedGenerateMoves():
  patternMoves = []
  for p in empty 8-neighbors of last move:
      if 3x3-area around p matches a pattern:
          patternMoves.append(p)
  ...
```

`Go3` Player Revisited

- Default `Go3`: Run with `python3 Go3.py`
- Simple Monte Carlo Player
- Almost-random simulations
  - Filter eye-filling moves only
  - No selective rules
  - All legal moves (after filter) equally likely
- Optional arguments: next slide

Options in `Go3`

- Number of simulations/move (default 10)
  - Example: run with 100 simulations/move
  - `python3 Go3.py -sim 100`
- Move selection method (default simple)
  - Example: use UCB for move selection
  - `python3 Go3.py -moveselect ucb`
- Type of simulations (default random)
  - Example: use rule-based simulations
  - `python3 Go3.py -simrule rulebased`

Summary

- Pure random playouts have strengths,
  but also systematic weaknesses
- Simulations as evaluation: have bias and variance
- Try to reduce errors by using filters and rule-based move generation
- Examples of moves removed by filters:
  - eye-filling moves
  - selfatari
- Example of rule-based move generation
  - 3×3 pattern responses