Computing Science (CMPUT) 455 Search, Knowledge, and Simulations

James Wright

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

Fall 2021

1

Today's Topics:

- Go rules revisited more details
- Profiling Python 3 code
- Improving the performance of our Go code
- Short quiz 3 review

Coursework and Uploads

Assignment 1

- Assignment 1 was due yesterday
- Feedback from TA via email by end of today
- Late/second submission deadline Wednesday 11:55pm
- 20% deduction; best of both submissions
- Second submission allowed for any reason
- Activities Lecture 6
 - profiling/optimization exercises

Go Rules Revisited

- Goal: tidy up some loose ends regarding rules
- Popular variations in rule sets
- Scoring at end of game
- Full repetition rules

- Introduced basic rules
- Showed examples of how to score at the end
- Implemented legal moves and reasonable policy for when to pass at the end in ${\tt Gol}$
- Position repetition: implemented only simple ko

- There are many different versions of Go rules
- All agree on how to handle the vast majority of situations
- Differences in details related to:
 - What is a legal move?
 - When does the game end?
 - How to score the game at the end?
 - How to resolve different opinions about scoring?

- Chinese, Japanese and Korean rules
- Ing, AGA (American Go Association), New Zealand, Tromp-Taylor rules
- · Most of these again have different versions and revisions

Main Differences (1): End of Game and Scoring

- When exactly is the game over?
 - Two or three passes
 - We use two

Main Differences (1): End of Game and Scoring

- When exactly is the game over?
 - Two or three passes
 - We use two
- How to score at the end?
 - Area scoring: count own stones plus surrounded empty points
 - Territory scoring: count surrounded points plus captured stones (and prisoners)
 - All rules: add komi to score
 - We use area scoring
 - · Easier to implement and play correctly at the end
 - In territory scoring, playing inside your surrounded areas costs points

Main Differences (2): Which Moves are Legal?

- Differences regarding suicide and repetition
- Most rules forbid suicide (we do too)
 - Exceptions: e.g. Tromp-Taylor rules
- Repetition: basic ko vs full board repetition
 - Our programs only recognize basic ko

Review: Repetition Rules - Basic Ko



- From top to middle picture: White can capture one black stone by playing A
- From middle to bottom picture: Now if Black captures back one white stone...
- The position would repeat, infinite loop
- This is called a (basic) ko.
- Go rules forbid such repetition

Repetition - Longer Loop



A four move loop in Go. Black passes on move 3.

- Example of a longer repetition loop
- This really happens in games between weaker Go programs
- If White tries to play move 4 in the corner, it repeats the position from four moves ago
- If both continue like this, infinite loop
- Gol does not recognize or prevent such repetition

Repetition - Triple Ko















A triple ko leading to a six move long loop in Go.

- Many rule versions forbid that the same board position is repeated in a game
- In the examples, the last, loop-closing move is illegal
- Such rules are often called superko rules
- They handle complex loops and situations with multiple active ko

Positional vs Situational Superko

- Superko idea: do not repeat the same board position
- What exactly is "the same"?
- Two main answers:
- Positional superko (PSK)
 - Ignore whose turn it is, only compare board
- Situational superko (SSK)
 - Compare whose turn it is as well as board
- Even more details: how do pass moves affect the repetition ban?

- Simplest but slowest:
 - Compare against all previous positions
 - Much too slow in practice
- One solution: use hashing to detect potential repetition
- Simple, effective trick (not complete solution): check if a move has ever been played before
- No details now, some later in search chapter

Profiling and Code Optimization

- Our Go0 and Go1 Python sample codes are very slow
- They were written for simplicity, not speed
- This is usually a good first approach see quotes next slide
- Optimization is very important in search, but it can wait a bit
- We can optimize *if* and when we need it
- First, look where the time is spent
- Profiling is an easy way to check this

Fred Brooks, The Mythical Man-Month (1975)

The management question, therefore, is not whether to build a pilot system and throw it away. You will do that. [...]

Hence plan to throw one away; you will, anyhow.

Don Knuth, Structured Programming with go to Statements (1974)

We should forget about small efficiencies, say about 97% of the time:

premature optimization is the root of all evil.

Limits of Optimization

- There is often an (approximate) 80-20 rule: 80% of the improvement can come from 20% of the code
- With search, it can be even higher
- However, Amdahl's law limits the amount of speedup

Limits of Optimization

- There is often an (approximate) 80-20 rule: 80% of the improvement can come from 20% of the code
- With search, it can be even higher
- However, Amdahl's law limits the amount of speedup

Example

- Assume a program spends 80% of its time in one function
- We manage to speed this function up 100x
- Question: How much is the overall speedup?

Limits of Optimization

- There is often an (approximate) 80-20 rule: 80% of the improvement can come from 20% of the code
- With search, it can be even higher
- However, Amdahl's law limits the amount of speedup

Example

- Assume a program spends 80% of its time in one function
- We manage to speed this function up 100x
- Question: How much is the overall speedup?
 - Less than 5x

Amdahl's Law



Image source:

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

Amdahl's_law

- Amdahl's Law (1967)
- How does speeding up one part of program speed up the whole?
- Often used for parallel programming
- Main idea: the parts of the program that are not optimized limit the overall speedup

- p = percentage of program that is speeded up
- *s* = speedup for that part
- Runtime before optimization: 1
- Runtime after optimization: (1 p) + p/s
- Speedup limit for the whole program:

• limit =
$$\frac{1}{(1-p)+p/s}$$

• Simplified version: assume *s* very large, then *p*/*s* is very small, ignore . . .

• limit
$$\approx \frac{1}{1-p}$$

- 80% of program speeded up, so p = 0.8
- s = 100 speedup for the optimized function
- Speedup limit for the whole program:

• limit =
$$\frac{1}{(1-p)+p/s} = \frac{1}{(1-0.8)+0.8/100} \approx 4.81$$

• Simplified version:

• limit
$$\approx \frac{1}{1-p} = 5$$

- Define a test that runs your program with a typical workload
- Run it with a special program called profiler
- Profiler tells you details of the program execution
- Profilers can be on the function level or instruction level
- How often was piece of code executed?
- How long did it take?
- Possibly, lower level details such as cache misses

Simple Profiling in Python with ${\tt cProfile}$ - Code

```
See code profile_Go1.py
import cProfile
from Gol import Gol
. . .
def play_moves():
    .....
    play 100 random games of 100 moves each
    for profiling.
    .....
    . . .
cProfile.run("play_moves()")
```

Simple Profiling in Python

- See code profile_Go1.py
- Try it out with
- ./profile_Go1.py > profile.txt
- sort -k 2 -r profile.txt
- This sorts by total time per function
- Try other options for -k to sort by other criteria
- Example: sort -k 1 -r profile.txt

- cProfile is a built-in module, no need to install anything
- Downside: overhead of profiling is also measured
- More advanced profilers are available for download:
 - Profilehooks
 - pycallgraph

See profiling on our Python language page

- Gol is slow
- · For search and simulation, speed is very important
- How to improve the code?
- Both low-level optimizations and better algorithms help
- Case study: a series of improvements to Go1
 - Result: Go2 same algorithm as Go1 but faster

- First, pick a test to measure the speed
- Here: play 100 games on 7×7 board
- Repeat:
 - Run test games with profiler
 - Identify the most expensive functions
 - Try to improve them by optimization or better algorithms

Profiling Go1

- Profile with cProfile
- Total time: 6.2 seconds
- Worst 5 individual functions listed below (all in board.py)

Calls	Time	Name
561025	1.960	neighbors_of_color
2287541	0.680	get_color
610480	0.679	_neighbors
43441	0.662	_block_of
18268	0.405	play_move

Profiling Go1

- Also look at cumulative time
- Function itself plus other functions it calls
- Sort by column 4:

sort -k 4 -r profile.txt

• Some interesting functions listed below (all in board.py)

Calls	Cumulative Time	Name
10974	4.429	is_legal
25584	3.566	_detect_and_process_capture
43441	3.368	_block_of
561025	3.351	neighbors_of_color
43441	1.359	_has_liberty

- Best: avoid calling a function
- Second best: speed up a function, avoid unneeded computation
- Here: detecting captures is most expensive

- Start by reading the expensive code carefully
- Can we avoid unneeded computation?
- Here: read _has_liberty, neighbors_of_color

```
def neighbors_of_color(self, point, color):
    nbc = []
    for nb in self._neighbors(point):
        if self.get_color(nb) == color:
            nbc.append(nb)
    return nbc
```

- We do not need to compute the whole list
- Stop if we find one liberty
- neighbors_of_color is still used in other places
- Add a function that is optimized for our task

```
def find_neighbor_of_color(self, point, color):
    for nb in self._neighbors(point):
        if self.get_color(nb) == color:
            return nb
    return None
```

```
def _has_liberty(self, block):
    for stone in whereld(block):
        if self.find_neighbor_of_color(stone, EMPTY
            return True
    return False
```

Profiling Again

- Total time reduced from 6.2 to 6 seconds
- Reduction in _has_liberty by calling cheaper find_neighbor_of_color instead of neighbors_of_color
- Nice improvement for a little work, but not a huge win
- Can we avoid the many floodfills altogether?
- We do the floodfill for each neighbor of a stone
- We only need to know "does block have at least one liberty"?
- Can we check that more effectively?

- We can store such a liberty for each stone $\ensuremath{\mathtt{s}}$
- In the code: liberty_of[s]
- Check capture: just check if board at location liberty_of[s] is still empty
- If yes, no floodfill is needed (why?)
- If no, we just played there
 - Do floodfill to try to find a *different* liberty for s
 - If success: update liberty_of[s]
 - If fail: yes it is a capture

Result, and More Floodfill Optimization

- Total time reduced from 6 to 4.4 seconds
- Success!
- Next: try to reduce calls to expensive floodfill functions
- Idea: instead of always computing a block:
- First check the 4 neighbors of the stone if there is a liberty there
- Result: Total time reduced from 4.4 to 3.7 seconds
- · Cost: more complex code, adds special case

Calls	Time	Name
66323	0.669	find_neighbor_of_color
18645	0.396	play_move
32369	0.367	_is_surrounded
264389	0.321	_neighbors
147455	0.294	neighbors_of_color
828018	0.257	get_color

Calls	Time	Name
66323	0.669	find_neighbor_of_color
18645	0.396	play_move
32369	0.367	_is_surrounded
264389	0.321	_neighbors
147455	0.294	neighbors_of_color
828018	0.257	get_color

Optimizing Neighbors, First Try

- · Called often: compute list of neighbors of a point
- Each call creates a new list
- Some neighbors are off the board (state BORDER), causing more tests in code
- Precompute a neighbors array for each point
- Include only on-board neighbors
- Result: EPIC FAIL, runtime over 11 seconds
- Why? board is copied and neighbors array recomputed over 11000 times

```
def is_legal(self, point, color):
    board_copy = self.copy()
    legal = board_copy.play_move(point, color)
    return legal
```

- This function is the reason for FAIL with previous optimization
- Slow: copy the board, then try to play the candidate move to see if it is legal
- Solution: Implement is_legal without play_move
- Success! Total time reduced from 4.4 to 2.5 seconds
- Cost: increased code complexity, some redundancy in is_legal and play_move

Details

Calls	Time	Name
51038	0.528	find_neighbor_of_color
75984	0.288	neighbors_of_color
21163	0.227	_is_surrounded
166427	0.207	_neighbors
495786	0.181	get_color
7418	0.145	play_move
{prev: 18645	0.396	play_move}

- play_move calls: less than half as many
- Many other function calls also significantly reduced

- Now we are no longer copying the board at each legal move check
- Now the neighbors optimization works beautifully
- · Result: Total time reduced from 2.5 to 2 seconds
- Success!
- There are more opportunities to optimize but Martin stopped here

Summary

- Discussed profiling and optimization
- Some concrete case studies
- Overall about 3x faster now, from 6 to 2 seconds on test
- Strategies:
 - Save computation
 - Precompute
 - Compute data incrementally when there are only small changes
 - Catch and handle frequent simple cases early

Summary

- Discussed profiling and optimization
- Some concrete case studies
- Overall about 3x faster now, from 6 to 2 seconds on test
- Strategies:
 - Save computation
 - Precompute
 - Compute data incrementally when there are only small changes
 - Catch and handle frequent simple cases early
- Very few optimizations are win-win. The speed often comes at the cost of code complexity
- Remember Knuth:

premature optimization is the root of all evil

- Quiz 3, Problem Solving and Decision Making & Maybe We Should Leave That Up to the Computer.
- 79 attempts. Average grade: 91.2%
- Lowest scores: Q7: 73.7%

- Quiz 3, Problem Solving and Decision Making & Maybe We Should Leave That Up to the Computer.
- 79 attempts. Average grade: 91.2%
- Lowest scores: Q7: 73.7%
- Q7: George Polya developed the concept of satisficing to express "good-enough" solutions.
 - No. Satisficing was introduced by Herbert Simon.

Quiz 3: Q21

Q21: After reading the article, do you have any follow-up questions or comments? For example, what do you want to learn more about? How does the article relate to the topics in the outline of this course? How does it relate to other things you have learned? Which topics or articles do you want to read next? Etc.

Quiz 3: Q21

Q21: After reading the article, do you have any follow-up questions or comments? For example, what do you want to learn more about? How does the article relate to the topics in the outline of this course? How does it relate to other things you have learned? Which topics or articles do you want to read next? Etc.

Popular answers:

- Practical applications/examples of computational decision-making
 - Especially models used in finance
- Reinforcement learning
- Mixed human-computer decision-making
- Ethical aspects of computer decision-making
- Automated detection of decision-making going wrong?