Implementing Go Environments

CMPUT 355: Games, Puzzles, and Algorithms

Recap: Tromp-Taylor Rules

1. Go is played on a 19x19 (or other dimension) grid by Black & White
2. Each point on the grid may be coloured Black, White, or Empty
3. A point P, not coloured C, **reaches** C if there is a path of nodes of P's colour from P to a point of colour C
4. **Clearing** a colour means setting all nodes of that colour that do not reach Empty to Empty
5. Starting from an empty grid, the players alternate turns, starting with Black
6. A **turn** is either a Pass, or a move that does not repeat an earlier grid colouring
7. A **move** consists of (1) colouring an empty point one's own colour (2) Clearing the opponent's colour (3) clearing one's own colour
8. The game ends after two consecutive Passes
9. A player's **score** is the number of points of her colour, plus the empty points that reach **only** her colour
10. The player with the higher score at the end of the game is the winner. Equal scores result in a tie.

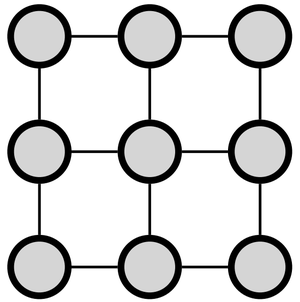
Lecture Outline

1. Recap
2. Go implementation issues
3. Example code walkthrough

Representing the Board

1. Go is played on a square grid of points, by two players called Black and White.
    - A "grid" is a graph in which each node has vertical and horizontal neighbours

- **Question:** How should we implement a board in Python?
    - i.e., what data structure to hold the current colouring?
- A 2D array makes a lot of sense
- **Questions:**
    - How do you represent a 2D array in Python?
    - Would that have any drawbacks?
    - Are there any alternatives?

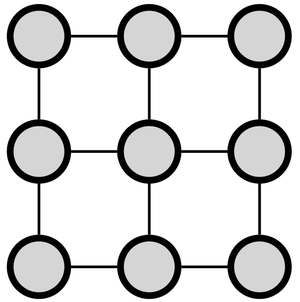


Neighbours

3. A point P, not coloured C, **reaches** C if there is a path of nodes of P's colour from P to a point of colour C

Questions:

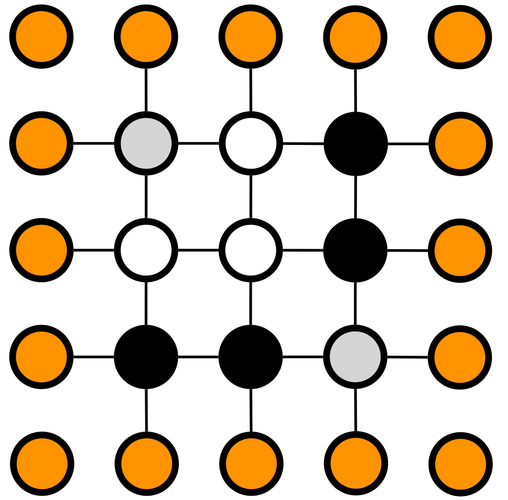
1. How many **neighbours** does each point have?
2. Why might we need to **iterate** over the neighbours of a point?
3. How can we iterate over the neighbours of a point in our board representation?
4. Is it possible to iterate over the neighbours of a point without checking for the edge of the board?
   - Why might that be desirable?



Guarded Board Format

3. A point P, not coloured C, **reaches** C if there is a path of nodes of P's colour from P to a point of colour C

- Scoring requires us to check if an Empty point reaches Black points and White points
- Capture detection requires us to check if a {Black, White} point reaches an Empty point
- Neither of these operations will behave differently if we surround the grid with points of a fourth colour **Guard** (**why?**)
- So long as we always start from a "real" point, we can then iterate over **exactly four neighbours** for all reachability operations



Captures

3. A point P, not coloured C, **reaches** C if there is a path of nodes of P's colour from P to a point of colour C
4. **Clearing** a colour means setting all nodes of that colour that do not reach Empty to Empty

Recall that a stone is captured if it is removed by clearing the opponent's colour after a move.

Questions:

- How should we detect captures after a move?
- Do we need to check every point on the grid? (**why?**)

Example Walkthrough: go_helper.py

7. A **move** consists of
    1. colouring an empty point one's own colour
    2. Clearing the opponent's colour (i.e., detecting **capture**)
    3. Clearing one's own colour (i.e., detecting **suicide**)

1. Clone the repository:
   git clone https://github.com/jrwright/games-puzzles-algorithms.git
2. Understand how **go/go_helper.py** does each of the following:
    - Represents the board
    - Checks for capture
    - Checks for superko
    - Performs a specified move
    - Computes the Tromp-Taylor score

Summary

- Guarded board representation simplifies neighbour checking
- Walked through straightforward implementation of Tromp-Taylor rules in **go/go_helper.py**