Computing Science (CMPUT) 657

Algorithms for Combinatorial Games

Martin Müller

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

Winter 2022

Cmput 657 Today, Lecture 1

- Introduction, course outline, Canvas
- Sample game - Clobber
- Two player games and their algorithms
- Combinatorial games, theory and algorithms

What is Cmput 657 about?

Broad Goals of this Course:

- Understand and use the algorithms of combinatorial games
- Basics of two player board games
- Combinatorial games as *sum games*
- Split a game into a sum of *subgames*
- Write working code for real games, use packages such as `CGSuite` and `MCGS`
- Use the techniques in your own *projects*

Goal: work with Sum Games

Some algorithm topics for sum games:

- Play a sum of games - choose one move in one subgame each turn
- Solve games when each subgame is small
- Take advantage of “special” types of games, i.e. *impartial* or *all-small* games
- Play well when subgames are large

Organization - Main Points

- This course has only lectures. No labs
- Main course site `https://webdocs.cs.ualberta.ca/~mmueller/courses/657-Fall2025/`
  - All content - slides, assignments, course information
  - Course outline
  - Assessments - readings, assignments, quizzes, research project
- Canvas - see link from main site
  - Quizzes, submitting assignments, course forum, announcements, access to non-public material (if any)

Contact/Communication

- Instructor: Martin Müller (email mmueller)
- Talk to me after class to ask questions, or email
- Use the **Canvas Discussions** to ask questions, discuss course material
- Read the **Canvas Announcements**
- `MCGS` technical questions: Taylor Folkersen (email folkerse)

Main Topics of Cmput 657

Three main topics, plus research projects

1. Two player board games - minimax, alphabeta etc.
   - “Full board search”, does not use subgame structure
2. Combinatorial games
   - Take advantage of subgame structure
   - Local search and analysis in each subgame
   - Slow the exponential growth of the solving cost
3. Efficient algorithms and case studies for combinatorial games
   - More advanced algorithms, temperature and thermographs, playing well in a large sum game
4. Work on course project (overlapping with part 3)

- Also see the **overview** on the website

Programming, Software, and Languages

- Software: `CGSuite, MCGS` (more later)
- Existing sample code is in a variety of languages
  - C++, C, Python, CGScript (CGSuite scripting language)
- Basic Python code provided for many algorithms

Programming - Expectations

- This is a hands-on course
- I expect you to do a good amount of programming yourself, and read/use existing codes
- For performance you may need to go to a more efficient language (e.g. C++ not Python)
- I expect you can read all sample code given
- I expect you can modify code and test it
- For your projects you can choose any language, as long as I can run it on a standard dept. linux machine

Part I

Two Player Games

Basic Concepts and Types of Games

Basic Concepts and Types of Games

- Sample game - Clobber
- Games and decision-making
- State spaces
- Types of games
- Combinatorial games

First Sample Game - Clobber

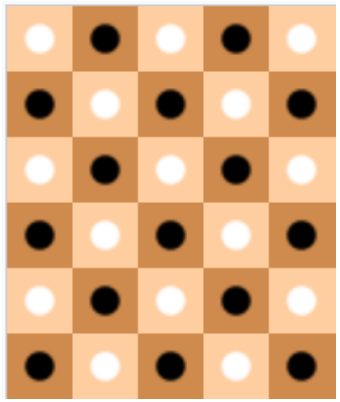


Image from

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

- Two player game (Black and White), White goes first
- Move: “clobber” one of opponent’s adjacent pieces
- End of game: making the last move wins
- No draws
- Now: quick demo game

How to Solve a Game?

- Mathematical analysis
- **Computer-based search**
  - Look ahead into the future
  - Optimize decisions systematically at each step
  - Adversarial, minimax search:
    - Optimize for *opponent* when it is their turn
  - Efficiency:
    - Which (sequences of) moves do we need to look at?
    - What is a good *move order* in which to look at moves?
  - Divide and conquer:
    - Can we solve a game by analysing its parts (subgames)?

Some Examples of State Space Search Methods

- Blind search: Breadth-first search, depth-first search
- Single agent heuristic search: A*, Greedy Best-first search
- **Game tree search: minimax, alphabeta, proof number search**
- Monte Carlo Tree Search
- Local search for single agent, e.g. hill-climbing
- **Local search in combinatorial games: one subgame at a time**

Problem-Solving Idea: Divide and Conquer

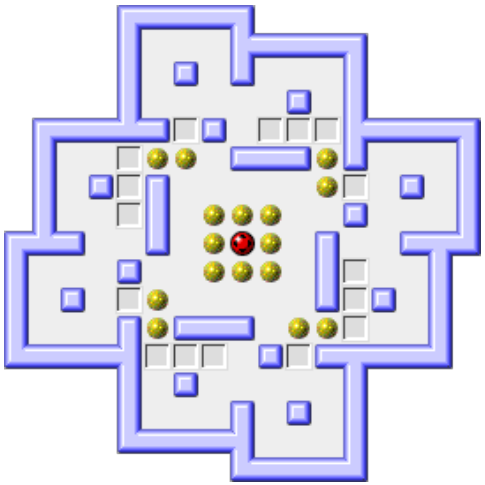


Image source:

`http://sneezingtiger.com/sokoban/levels.html`

- Break problem into smaller sub-problems
- Solve them and combine solutions
- Examples: dynamic programming, branch and bound
- Example: Sokoban puzzle: solve each “room” separately
- **Example: Go endgame puzzles, can find value of move locally**

Types of Games

- Many ways to classify games
- The algorithms can be very different depending on the type
- Some important criteria:
  - Number of players
  - Chance element (luck, dice, ...) vs no chance element
  - Full vs incomplete information
  - Cooperative vs adversarial games
  - Many more, see e.g. `https://en.wikipedia.org/wiki/List_of_types_of_games`

Number of Players

- One player: puzzle
  - Can often be solved as optimization problem
  - Shortest path, or maximize reward
  - Classical heuristic search methods, A* etc.
- Two players: here in this course
- Three or more players:
  - More difficult theory
  - Coalitions between players
  - Not clear what is rational play
  - Technically: can have multiple Nash equilibria with different payoffs for our player

Chance Element

- Examples: roll dice, draw a card
- Chance determines which actions are possible (or what effect they have)
- More difficult to look ahead
- Need to look at different chance results at each step
- Also compare: MDP with probabilistic transitions vs deterministic MDP

Classical Two Player Games

- We focus on “classical” two player games:
- No chance
- Players move alternately: I play, you play, I play,...
- A move instantly changes the state (no duration, no slow transitions)
- Simplest, most frequent case is *zero-sum*:
  my win is opponent’s loss
- Examples: chess, checkers, Go, Clobber, Tic-Tac-Toe, ...
- We focus on this type of game in this class

Combinatorial Games

- Classical two player games with additional structure
- Game (often, eventually) consists of several *independent subgames*
- A move in one subgame leaves all the others unchanged
- Demo: 1×10 Clobber
- **Key question** in this course: can we use subgame structure to solve these games (much) more efficiently?
- In the first few weeks, we will discuss solving classical two player games in general. Then, we will specialize on combinatorial games.

Decision-making, States and State Spaces

Making Complex Decisions

- How to make good decisions?
- Consider many alternatives
- Consider short-term and long-term consequences
- Evaluate different options and choose the best-looking one
- Understanding and comparing sequences of actions is the main step in making such decisions

Why Study Decision-Making using "Classical" Games?

- Simple, controlled environment
- Still hard to solve or play well
- Interesting for many people
- Games and results are easy to understand
- Playing games well requires good decision-making skills
- We can study some *core problems* of decision-making without being distracted by too many complications

States and Actions

- State = “possible configuration of a system”
  - Two player game: current position - where are the pieces, whose turn is it, possibly (some of) the history of previous moves
- Action - leads from one state to another
  - Two player game: a move by the current player
- State determines which actions are possible
- Here: no uncertainty, no “acts of nature”

State Space

- State space - directed graph of states connected by actions
- Start state = initial configuration of the game
- Terminal state: game over
- Result: who wins when game is over
- Goal: find a winning strategy
  - **win no matter what the opponent plays**

Example: State Space of Clobber on 2×2 Board

- Demo: work out the state space, introduce notation

Formal Framework

Formal Framework for Two Player Games

- Set S of game states and A of actions
- Two players, often called Black and White
- Game: sequence of states and actions (moves)
- Start state s_0
- Action a_i instantly leads to next state, s_{i+1}
- Keep going until we reach a terminal state s_n
- Sequence $(s_0, a_0, s_1, a_1, ...s_n)$

Formal Framework for Two Player Games (2)

- Result is a partial function defined only on terminal states
  - Example: black wins, white wins, draw
- Sometimes we only write the actions $(a_0, a_1, ...a_n)$
  - Example: games where states are determined from game rules and actions

State Space

- A *state space* is:
  - A graph with all the possible states of a problem
  - Edges in graph show how states are connected by actions
- State space represented as directed graph $G = (S, E)$:
- Nodes in S: game states
- Directed edges in E: moves
  - Edge $e = (s_1, s_2)$ contains:
  - State s_1 before move
  - State s_2 after move

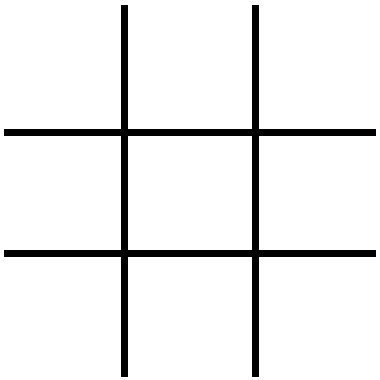
Terminal State

- A *terminal state* has no possible moves (actions)
- No outgoing edges in graph
- The rules of a game decide:
  - When the game is over (When do we reach a terminal state?)
  - What is the result in a terminal state?

Results in Terminal States

- Simplest case: only two results, *win* and *loss*
- Can use a boolean: true = win, false = loss
- Classical combinatorial games have only these two results
- Other results: draw, no-result (e.g. chess, Tic Tac Toe)
- Other results: numeric value (win by X points)
- Can map simple results to numbers: e.g. +1 for win, -1 for loss, 0 for draw
- Compare with rewards in reinforcement learning

Game: Tic Tac Toe



- 3×3 board
- Two players, X and O
- To win, make 3 in a row horizontally, vertically or diagonally
- Draw is possible (board full, no three in a row)

Game: Go

See separate slide set for Go

Types of State Spaces

Types of State Spaces

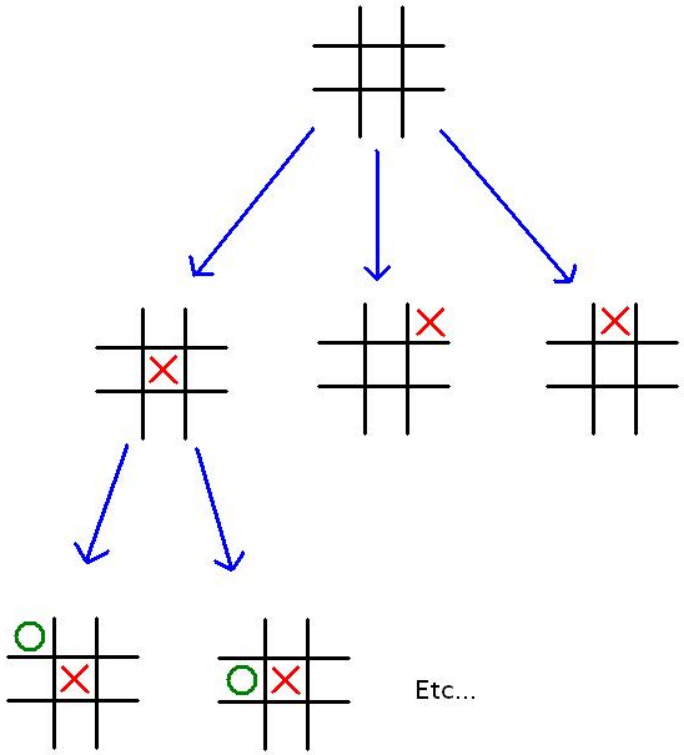


Image source:

`sciencefair.math.iit.edu`

- Assume *root* at the top is current state
  1. Tree
  2. DAG (directed acyclic graph)
  3. DCG (directed cyclic graph)
  4. Tree is easiest for search, DCG hardest
- *Game graph, game tree* are other terms for state space of games

Complexity of State Space

- Some measures of game complexity:
- Size of state space
- Branching factor (number of actions in state)
- Difficulty of game can depend on many other things
  - Is there a simple strategy?
  - A mathematical theory?
  - Many master games to learn from?
  - Good heuristics?

Simple Tree Model of State Space

- *Constant branching factor* b
- Each *interior* node in tree has b children
- *Uniform depth* d
- Each path from root is d actions long

Simple Model - Count Nodes

- How many nodes?
- 1 root node, 1 = b^0 total nodes at depth 0
- b children of root, b^1 total nodes at depth 1
- Each child has b new children, total b^2 at depth 2
- ...
- Last level b^n nodes at depth n
- Total nodes $1 + b + ...b^n = (b^{n+1} - 1)/(b - 1)$
- For large b, this is close to b^n - last level dominates

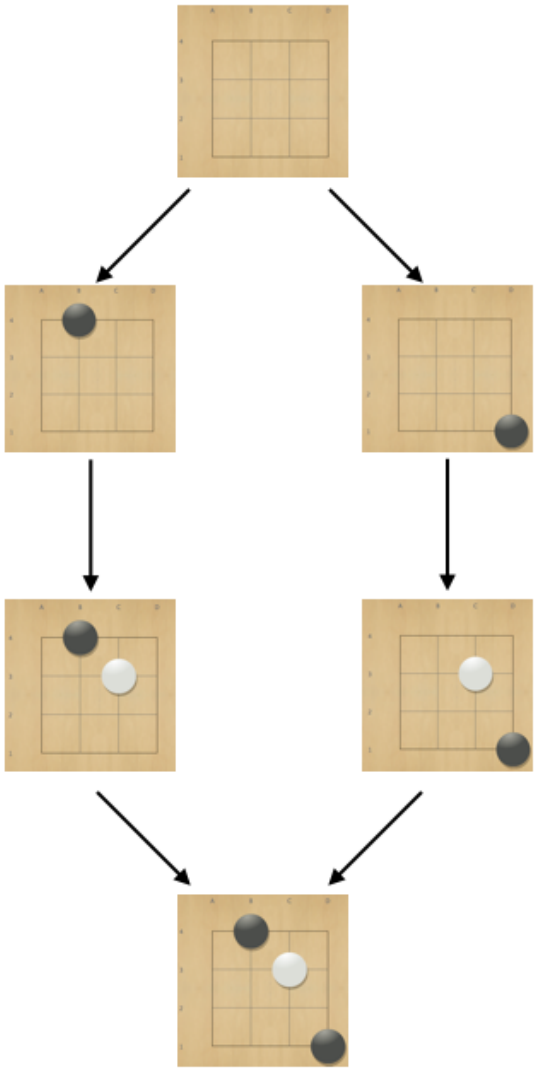
Example - 7×7 Go

- Show slides - game of Go
- 7×7 Go, start on empty board
- Process 1000 states/second
- Simple tree model, $b = 49$

Table: Estimated *additional effort* to search one level deeper

Depth	New states	Added search time
0	1	1ms
1	49	50ms
2	49^2	2.4s
3	49^3	2 min
4	49^4	1.6 hrs
5	49^5	3.2 days
6	49^6	160 days
7	49^7	21.5 years

DAG (Directed Acyclic Graph)



- Idea: single node for all *equivalent* states
- Different paths to same node
- Can lead to huge reduction in state space
  - *Subtree (really Sub-DAG)* below is no longer duplicated

Tree vs DAG

- Tree model
  - Each action leads to a new node
- DAG model
  - Only one node for *equivalent* states
- Massive reduction in size of state space
- Advantages of DAG model:
- Avoid redundant computations
  - No copied subtrees or sub-DAGs
- Share results of analysis - compute once, re-use often

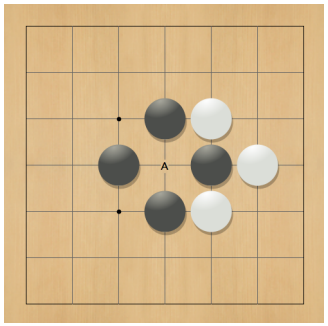
Limitations of DAG Model

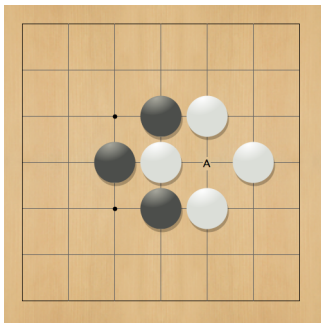
- Main problems:
  - Need memory to store and recognize equivalent states
  - Some algorithms designed only for trees, not for DAGs
- Example: propagating information up towards root
  - Only one path up in tree - efficient
  - Many paths in DAG - many ancestors

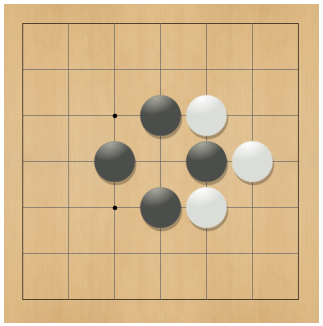
Problems with DAG Model in Go

- Go rules depend on history, not just on current board
- Ko rule: position repetition is forbidden (next slide)
- A move in the same board position may or may not be legal depending on which previous states we've seen
- Kishimoto (former PhD student here): efficient solution to so-called GHI (graph history interaction) problem
  - Idea: similar states often have same proof of win/loss
  - Idea: can determine when history is irrelevant and use that

Repetition Rules - Basic Ko

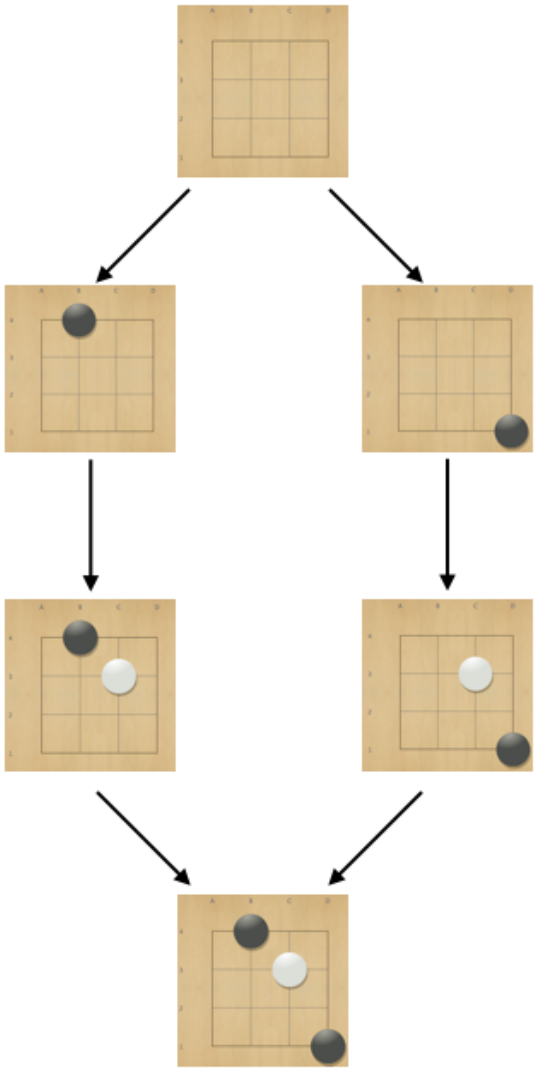






- From top to middle picture: White can capture one black stone by playing A
- From middle to bottom picture: Now if Black captures back one white stone...
- The position would repeat, infinite loop
- This is called a (basic) ko.
- Go rules forbid such repetition

Counting States in a DAG



- Simplified Go example, ignore symmetry and captures
- Depth 0: 0 black, 0 white stones
- Depth 1: 1 black, 0 white stones
- Depth 2: 1 black, 1 white stones
- Depth 3: 2 black, 1 white stones
- Depth d: $\lceil d/2 \rceil$ black stones and $\lfloor d/2 \rfloor$ white stones
- How many ways to put that many stones on a board with 49 points?

Counting States in a DAG (continued)

- Example:
- Board with 49 squares
- How many different ways to place 5 black stones?
- Answer: $\binom{49}{5} = 1906884$
- How many different ways to place 5 black stones and 3 white stones?
- Answer: $\binom{49}{5} \times \binom{44}{3} = 1906884 \times 13244 \approx 25.2$ billion

Complexity of Popular Games

- Most real games do not follow a simple (b,d) model
- Big table in
  `https://en.wikipedia.org/wiki/Game_complexity`
- Different measures of complexity
- Complexity depends strongly on size of board, type of moves
- In Go, the theoretical complexity is much higher
  - Main reason: capture, play again on same point

Re-Interpreting the Tree and DAG Models

- Our model so far:
- State space as a graph
- Nodes are states, edges are actions
- Tree and DAG are special cases of graphs
- New view of the same trees and DAGs
  - A way to organize all action sequences

Organizing Sequences in Trees

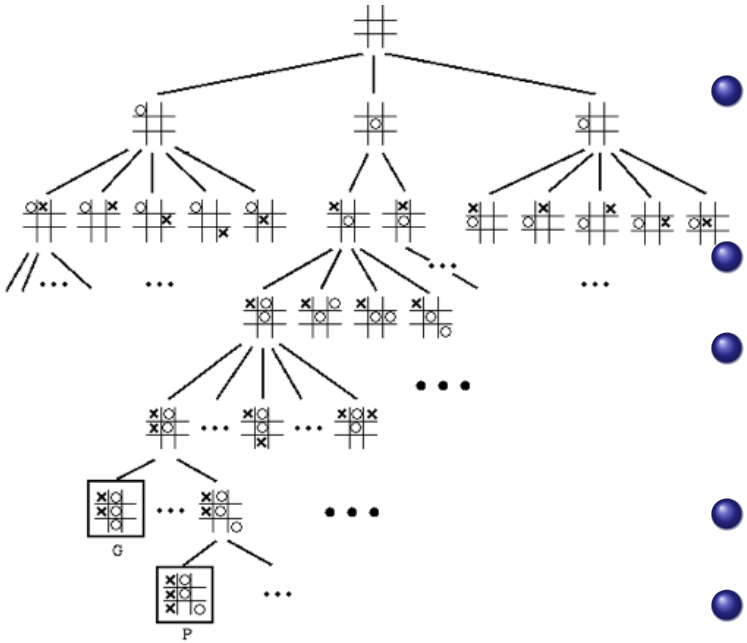
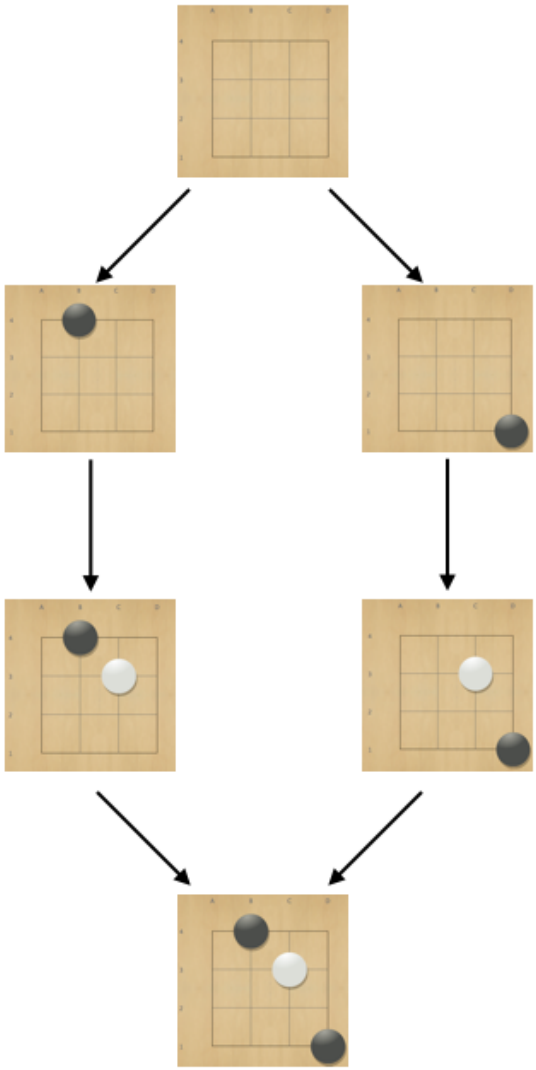


Image source: `http://web.emn.fr`

- Set of all possible state-action sequences
- Organize them such that:
- Any two sequences share their *longest common prefix*
- Branch as soon as they differ
- Result: we get exactly the tree representation of the state space

Organizing Sequences in a DAG



- Similarly, we can relate sequences to the DAG model
- Start with sequences-as-tree model
- Merge two different sequences when they both reach equivalent states
- Result: exactly the DAG representation

Summary

- Basic concepts of games, especially two player games
- States, moves, state spaces, evaluation
- Next topics: solving two player games, minimax and alphabeta, proof number search