CMPUT 396 – 2-Player Games

In a puzzle, one player makes all the moves so to solve a puzzle, you need to search the state space for the best possible outcome.

In 3+ player games, there could be possible coalitions. For example, in poker often players collectively gang up on the weakest player to take all their money (this player is called "the fish").

But what about 2-player games like chess, Go, hex? How should we solve them?

History of solving chess:

- 1770: Turk Automaton
  - Fake chess-playing machine, appeared to be a machine that could play chess and beat strong players
  - In fact it was an illusion where a chess master would hide inside and operate the machine.
- 1913: Zermelo (German Mathematician), "On the application of counting theory to the theory of chess"
  - Can you actually prove anything about chess given its rules?
  - Zermelo proved that with finite moves and no repetition the game will result in either a win for opponent A, a win for opponent B, or a draw
- 1947: WWII code breaking moved from being done by hand to by machine and the Enigma cryptography machine was solved
  - Americans were using computers to create thermonuclear devices
  - Claude Shannon from Bell Labs had conversations with Alan Turing where they discussed chess-playing algorithms. Through these conversations, they came up with minimax.

Adversarial Search

The simplest multi-player games have the following characteristics:

- 2 players (what position are we in currently, whose turn is it to move)
- Alternating turns
- Zero-sum (only one winner; something good for you is bad for me)
- Deterministic (no chance)

Some games like this include Go, tic-tac-toe, checkers, chess (although chess can actually end in a draw), etc.

Adversarial search: searching a 2-player game tree which captures all the possible moves in the game, where each move is represented in terms of loss and gain for one of the players.

Consider a player about to move. In order to pick the best move, you need to know how the opponent will respond. Always assume that they will take the optimal move, that they are perfect.

Why assume that your opponent is perfect?

- If you make this assumption and find a winning strategy you know you can win against all possible opponent strategies (because you already proved you could beat the best opponent strategy).
- Downside? Maybe more work? But it is worth it to know that it's the best we can do.

If you assume that your opponent always takes the optimal move and you can make a provable winning move (a move than wins against all possible opponent strategies), you have solved that position.

What does it mean to solve a game?

- You can predict the outcome (win, lose, draw) from any position, assuming that both players play perfectly.

Some definitions:

- **State:** current position of board and player to move
- **Reachable state:** any state reachable by sequence of legal moves
- **Strategy for a player:** a function that, for any reachable state, returns a legal move

In a 2-player game, to *solve a state* is to find a strategy with best worst-case performance, i.e. that guarantees the best possible score over all possible opponent strategies (so, assuming the opponent always makes a best possible move).

The initial move for any such strategy can be found by **minimax search**, where the score is the player-to-move's **minimax score**.

Play that follows a minimax strategy is called **perfect play**.

Schaeffer (solved checkers):

- His checkers program played Tinsky, but Tinsky won and later, before they could play again Tinsky died
- Schaeffer had to prove that he could beat him still, so he started his project to *solve* checkers (which he did)

Minimax

To find a minimax value, explore the game-state tree in any order that finds values of children before the value of the node.

Minimax algorithm: search space to find root state minimax value, 2 cases (min and max)

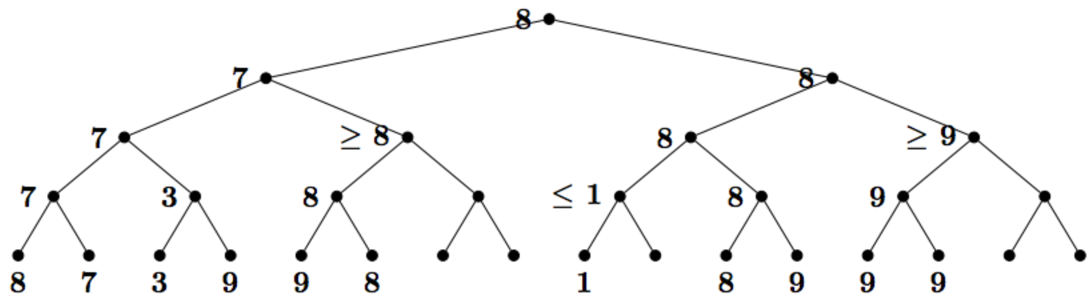
```
root node is a max node
each child of a max node is a min node
each child of a min node is a max node

def eval(s):
  for player MAX, return score of s

def minimax(s):
  if terminal(s):
    return eval(s)
  if s.player() == MAX:
    return max(for all children c of s, minimax(c))
  if s.player() == MIN:
    return min(for all children c of s, minimax(c))
```

Minimax Example:

```
node: state
edge (root,    level 1):  p1 move
edge (level 1, level 2):  p2 move
leaf label: p1 score

for each node,
  p1 minimax value ?
best play?
                    *
                /   |   \
              /     |     \
            *       *       *
          / | \    / \    / | \
         *  *  *  *   *  *  *  *
         2  3 -1  1   6 -4  2  9

                    1
                /   |   \
              /     |     \
           -1       1      -4
          / | \    / \    / | \
         *  *  *  *   *  *  *  *
         2  3 -1  1   6 -4  2  9

p1 best play from root: middle

p2 best play at level 1:
  after p1 left:   right (p1 scores -1)
  after p1 middle: left  (p1 scores  1)
  after p1 right:  left  (p1 scores -4)
```

Pruning Game Trees

When solving a state, it usually not necessary to examine the whole tree. Once a winning move is found, you can ignore the remaining moves. This is the motivation behind alpha-beta search.



```
recall min/max format:
  p1 max, p2 min, show p1 values

         a
      /     \
     b       c
    / \     /|\
   d   e   f g h
   7   5   3 ? ?

assume dfs with children explored left-to-right
         *
      /     \
     5      <=3
    / \     /|\
   7   5   3 ? ?

after examining node        we know ...
  d                         val(b) <= 7
  e                         val(b) = min(7,5) = 5
  c                         val(c) <= 3

now, no need to learn more about val(c)
... val(c) <=3    <   val(b)
... p1  prefers   b   to   c
... so prune remaining children of c from our search
```

Alpha-beta Search

Alpha-beta search is minimax search with alpha-beta pruning. It stops completely evaluating a move when at least one possibility has been found that proves the move to be worse than a previously examined move. Nodes can be pruned when beta <= alpha.

Alpha: value of the best P1 option so far on the path from the current node to the root
Beta: value of the best P2 option so far on the path from the current node to the root
Minimax can be used as a solver where the leaf scores must be true scores and you must able to reach all leaf nodes in reasonable time. It can also be used as a heuristic player so if you can find a fast heuristic you can use it on all leaves at a fixed depth (e.g. a simple chess player.

Alpha: minimum score that maximizing player is assured of

Beta: maximum score that the minimizing player is assured of

Initially, alpha is $-\infty$ and beta is $+\infty$ (both players start with the worst possible score). When the max score of the minimizing player ("beta") is less than min score of maximizing player ("alpha"), $beta \leq alpha$, so the maximizing player does not need to consider further descendants of the node (so we can "cutoff" the rest of the descendants).