Chapter 5: Games and adversarial search

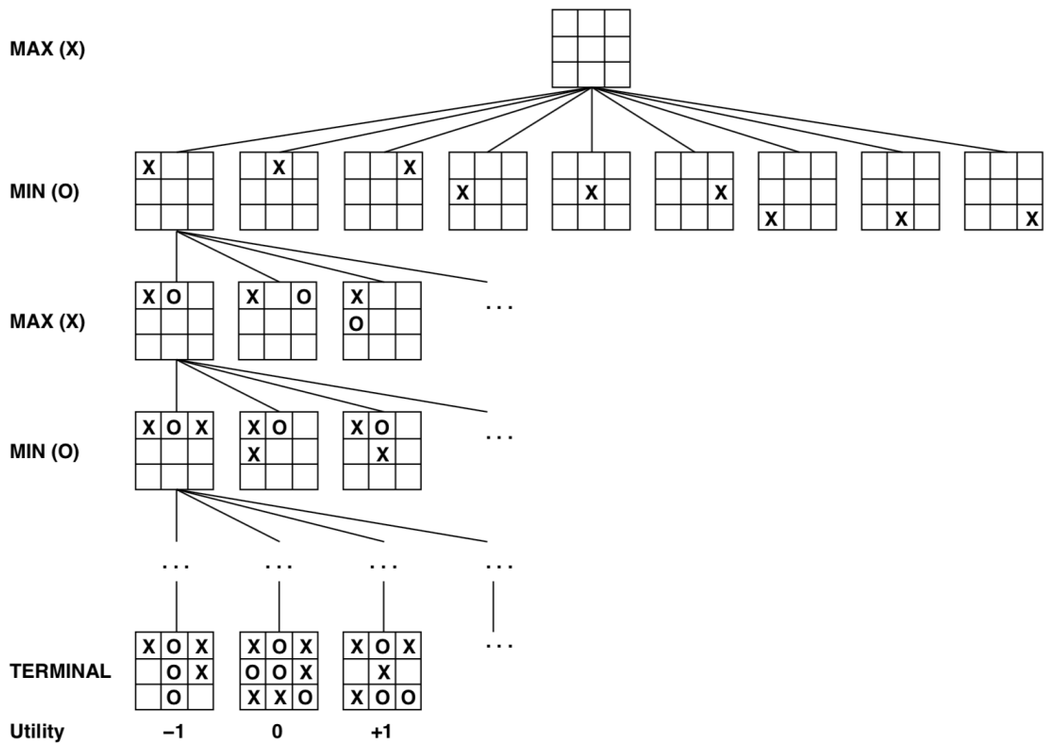
W3-Fri:
- HW0 due today.
- HW1 is out.

- In this section we will relax the assumption that the problem is single-agent; we will look at multi-agent problems.
- multi-agent, discrete, deterministic, observable
- That is, games: Chess, Go, Starcraft, etc. We will focus on the simplest type: deterministic, turn-taking, two-player, zero-sum. Similar methods work for other multi-agent problems (nondetermistic, multi-player, cooperative).
- Solution is a strategy: response for every move the opponent could play.

Components of a game

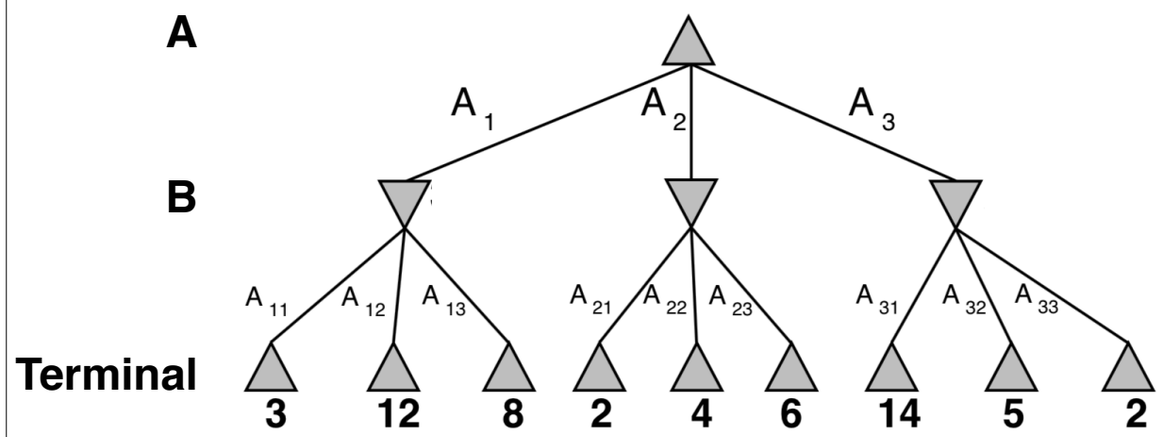
initial state s_0
first player - A or B
successor(s_1, a) -> s_2
terminal(s) -> T/F
utility(s) -> value <- score for A (+1, -1 or 0). Another name for score is "utility".

Tic-tac-toe



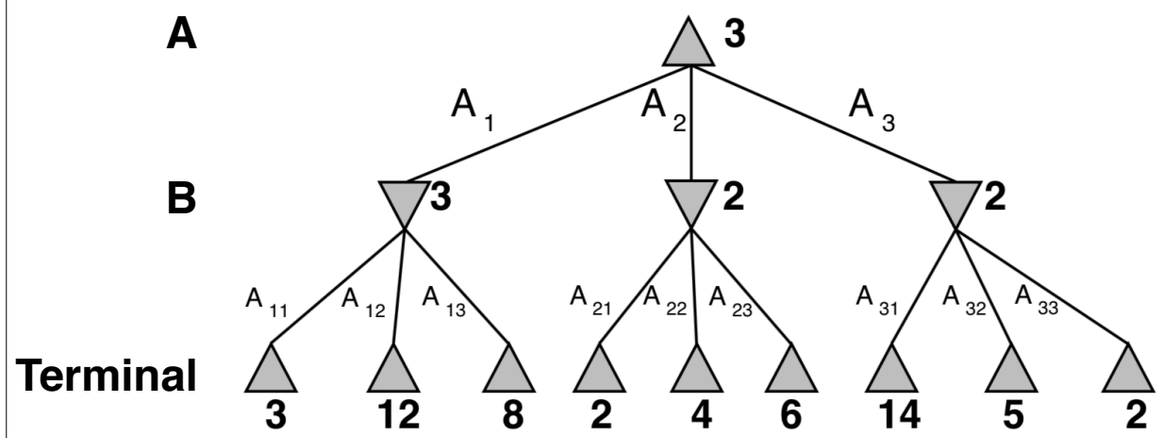
- Example: Tic tac toe.
- Game tree: like a search tree.

Minimax algorithm



- Minimax algorithm: The core of all of our game playing algorithms (like tree-search for single-agent). The core of DeepBlue, AlphaGo.
- Idea: Assume our opponent picks the best move they can (for them, worst for us). We want to get the best score given that.
- Example

Minimax algorithm



- Minimax algorithm: The core of all of our game playing algorithms (like tree-search for single-agent). The core of DeepBlue, AlphaGo.
- Idea: Assume our opponent picks the best move they can (for them, worst for us). We want to get the best score given that.
- Example. Note range of scores.

Minimax algorithm

```
def minimax(game, player=game.first_player):
    if terminal(state):
        return utility(state)
    best = -inf if player==A else inf
    for action, next_state in successors(state):
        if player == A:
            best = max(best, min(next_state))
        if player == B:
            best = min(best, max(next_state))
    return best
```

- This algorithm doesn't tell you what action to perform. Extending it to do so is a good exercise.

Minimax algorithm

```
def minimax(game, player=A):
    if terminal(state):
        return utility(state)
    best = -inf if player==A else inf
    for action, next_state in successors(state):
        if player == A:
            util = minimax(next_state, player=B)
            best = max(best, util)
        if player == B:
            util = minimax(next_state, player=A)
            best = min(best, util)
    return best
```

Properties of minimax

Complete:

Optimal:

Time:

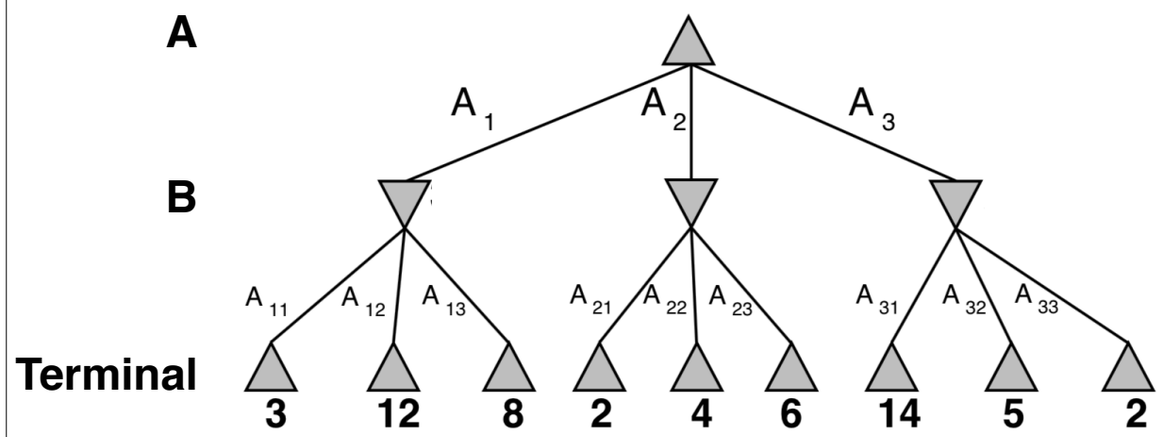
Space:

b = branching factor (number of moves)
m = maximum number of moves in game

Complete: **Yes if finite**. (Chess has specific rules for this.
Optimal: **Yes against perfect opponent**. We will do at least as well as we predicted, but we may be able to do better.
Time: b^m. Totally impractical for most games. About 10^40 for chess.
Space: b*m.

b^m is totally impractical for most games. About 10^40 for chess. Everything else we will talk about is for speeding this up.

Alpha-beta pruning

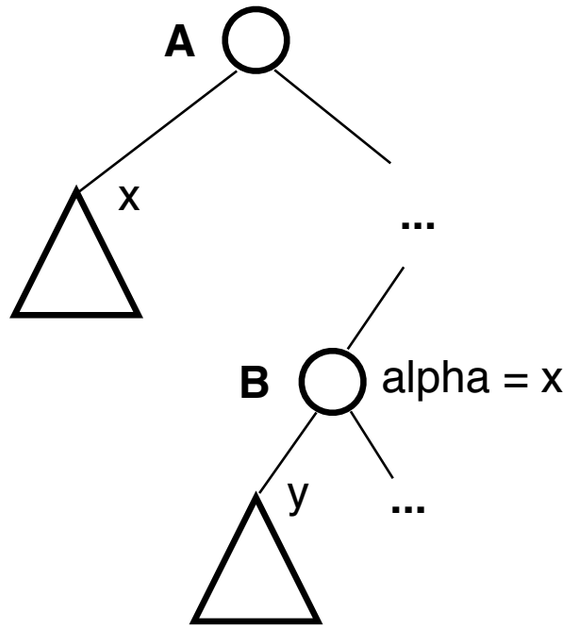


- Idea: We don't need to perfectly compute the utility of a state if we know that we already found a better move.
- Middle branch: evaluate 2; no need to continue.

Alpha-beta pruning

```
def minimax(game, player=A, alpha=-inf, beta=inf):
    if terminal(state):
        return utility(state)
    best = -inf if player==A else inf
    for action, next_state in successors(state):
        if player == A:
            if best >= beta: return best
            util = minimax(next_state, player=B,
                           alpha=best, beta=beta)
            best = max(best, util)
        if player == B:
            if best <= alpha: return best
            util = minimax(next_state, player=A,
                           alpha=alpha, beta=best)
            best = min(best, util)
    return best
```

Alpha-beta pruning



- Node 1: **>= x**
- Node 2: **<= y**.
- Do we need to search the right side of B node?
    - Suppose y <= x
    - Two options: 1) We find more positive -- don't care, B will choose y. 2) We find more negative -- A will make a different choice at node 1 anyway.

Properties?

- Pruning does not affect result! Still optimal. You always want to use it.
- Effectiveness depends on move ordering.
- With perfect ordering, time complexity = O(b^(m/2)). (Note: Not (b^m)/2 !).
- Can usually get good ordering. In chess, consider capturing pieces, putting in check etc.

Heuristic evaluation function

- Even with alpha/beta, we can't search the full tree.
- Instead, search to a fixed depth, then use a heuristic evaluation function to estimate how good the position is.
- **h(game) = estimated utility for A / probability that A wins**
- **h(chess) = difference in sum of piece values / total.**
    - **white queen, white pawn, black queen = 9 + 1 - 9 = 1 / 39.**
- Just like in search, could be more complicated: take into account piece positioning, king safety, etc. Better evaluation function -> less cost from cutting off search.
- Alternatives to fixed depth:
    - Quiescence: search deeper in volatile board states. Example: sequence of captures in chess.
    - Search only "good" moves past a certain depth.
- Example: **100 seconds/move, 10^4 nodes/sec, 35 reasonable moves/state (chess) -> 10^6 nodes/move ~= 35^(8/2) -> depth 8 search**. Makes for a pretty good chess program.

Games in practice

- **Othello - 1980**: No contest, computers are better. Potential assignment was to write a othello bot.
- **Checkers - 1994**: Computer beat 40-year world champion Marion Tinsley. Used an endgame database defining perfect play for positions with 8 or less pieces ~10^11 positions.
- **Chess - 1997**: DeepBlue beat Gary Kasparov.
- **Go - 2017**: AlphaGo beat Ke Jie. Used a neural network to define an evaluation function.