CS5100: Foundations of Artificial Intelligence

Adversarial Search

Dr. Rutu Mulkar-Mehta

Lecture 4

Some slides and images used from Berkeley CS188 course notes, with permission
Some slides and images used from Cornell CS4700 course notes, with permission

Administrative

- Extra Credit Assignment:
  - Was due at 6:00PM today on Gradescope
- Assignment 1 solutions are graded and returned to you
- Project 1:
  - Due on 10/06
  - Submission instructions:
    - Please email your solutions to:
    - Professor: me@rutumulkar.com
    - TA: k.porterfield7@gmail.com
    - Subject: [PROJECT 1] <FirstName> <LastName>

Assignment 1: Solutions

(a) DFS
W, B, M, R
(b) BFS
Order of States:
W, B, M, O, R
Path:
W, M, R

Assignment 1: Solutions

(c) UCS
- Order of States:
  W, O, B, M, V, R
- Path:
  W, M, V, R

(d) Greedy
W, M , R

(e) A*
- Order of States:
  W, M, B, O, V, R
- Path:
  W, M, V, R

Assignment 1: Solutions

(f) The Heuristic is Admissible!

- h(Odesa) = 20 hrs,
- h(Budapest) = 12 hrs,
- h(Munich) = 3 hrs,
- h(Venice) = 3 hrs,
- h(Rome) = 0 hrs,
- h(Warsaw) = 30 hrs.

Heuristic shouldn't be higher than the actual cost (not lowest cost) of reaching the goal

Recap

- Search
  - Uninformed Search
  - Informed Search
- CSP
  - Ordering
    - MRV, Degree, Least Constraining Value
  - Fitering
    - Forward Checking
    - Arc Consistency

Today's Outline

- Adversarial Search (or Game playing)
  - Minimax Algorithm
  - α-β pruning
  - Expectiminimax Algorithm
  - In class Assignment

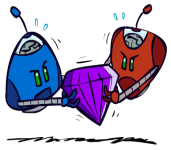
Games vs. Search problems

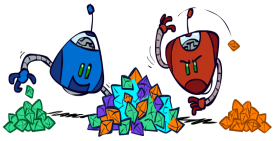
- Search Problem
  - Bad heuristics → slow solution,
  - e.g. A* will take twice as long to reach the solution
- Games
  - Bad solution → loses badly

Types of Games

- Many different kinds of games!
- Adversarial Games
  - Win of one player is a loss of another
  - Type of Game AI is most interested in
- Physical games such as ice hockey, cricket, baseball etc. are much more complicated
- "Robot soccer" is probably the only physical game that has attracted the interest of the AI community

Zero-Sum Games





- Zero-Sum Games
  - Agents have opposite utilities (values on outcomes)
  - Lets us think of a single value that one maximizes and the other minimizes
  - Adversarial, pure competition
- General Games
  - Agents have independent utilities (values on outcomes)
  - Cooperation, indifference, competition, and more are all possible
  - More later on non-zero-sum games

Game Playing State-of-the-Art

- **Checkers:**
  - 1950: First computer player.
  - 1994: First computer champion: Chinook ended 40-year-reign of human champion Marion Tinsley using complete 8-piece endgame.
  - 2007: Checkers solved!

Game Playing State-of-the-Art

Chess:

- 1997: Deep Blue defeats human champion Gary Kasparov in a six-game match.
- Deep Blue examined 200M positions per second, used very sophisticated evaluation and undisclosed methods for extending some lines of search up to 40 ply.
- Current programs are even better, if less historic.

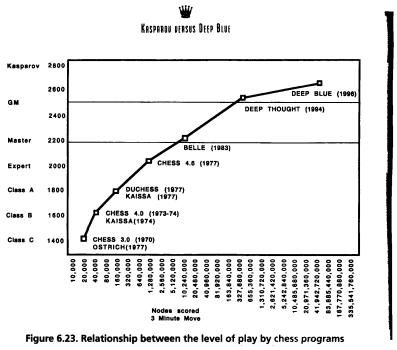


Figure 6.23. Relationship between the level of play by chess programs

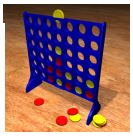
Game Playing State-of-the-Art

Go:

- In October 2015, AlphaGo became the first Computer Go program to beat a professional human Go player without handicaps on a full-sized 19×19 board.

Traditional Board Games

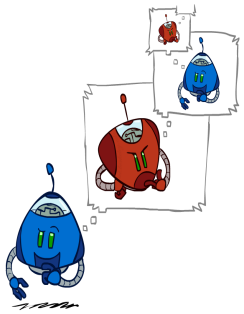
- Finite
- Two-player
- Zero-sum
- Deterministic
- Perfect Information
- Sequential



Terminology

- Perfect information:
  - All information is visible to all players at all times – e.g. Chess, Checkers, etc.
- Imperfect information:
  - Some information is not visible to all players e.g. Poker, Bridge etc.
- Zero Sum Games
  - One players' win is the other player' loss

Adversarial Search

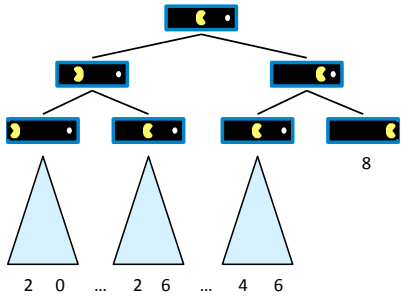


Formal definition of a game:

- Initial state
- Successor function: returns list of *(move, state)* pairs
- Terminal test: determines when game over
- Terminal states: states where game ends
- Utility function (objective function or payoff function): gives numeric value for terminal states

We will consider games with 2 players (**Max and Min**)
Max moves first.

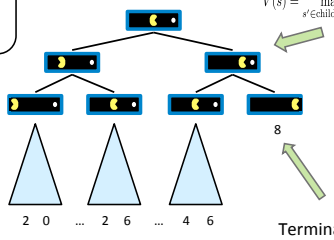
Single-Agent Search Trees



Value of a State

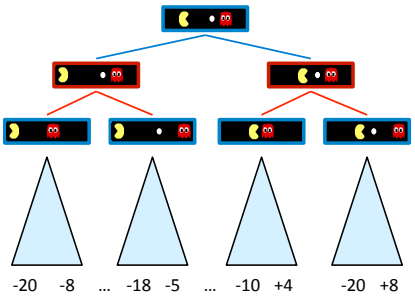
Value of a state: The best achievable outcome (utility) from that state

Non-Terminal States: $V(s) = \max_{s' \in \text{children}(s)} V(s')$



Terminal States: $V(s) = \text{known}$

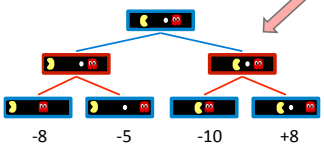
Adversarial Game Trees



Minimax Values

States Under Agent's Control: $V(s) = \max_{s' \in \text{successors}(s)} V(s')$

States Under Opponent's Control: $V(s') = \min_{s \in \text{successors}(s')} V(s)$

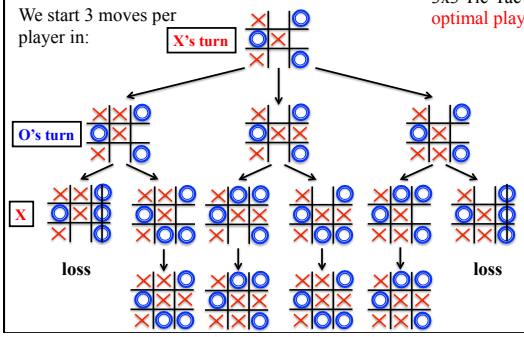


Terminal States: $V(s) = \text{known}$

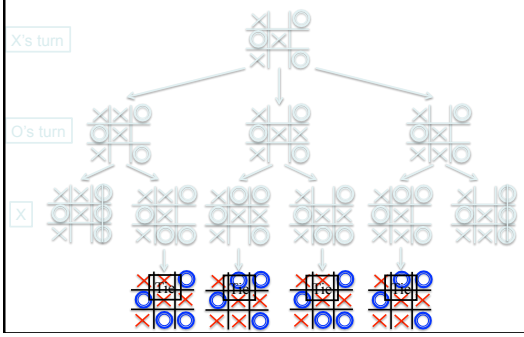
Key Idea: Look Ahead

We start 3 moves per player in:

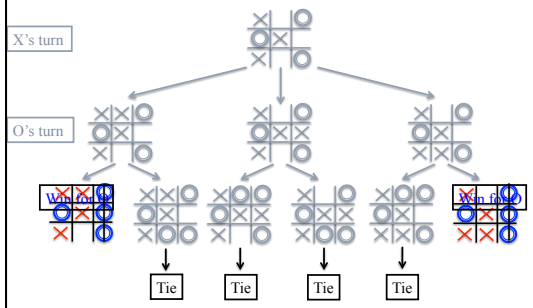
3x3 Tic-Tac-Toe optimal play



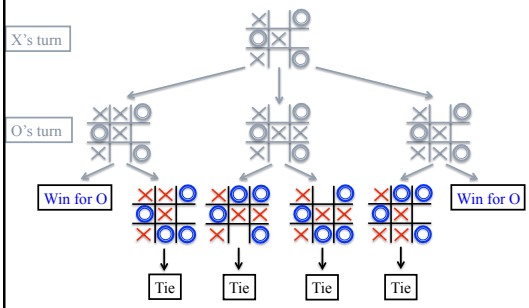
Look-ahead based Tic-Tac-Toe



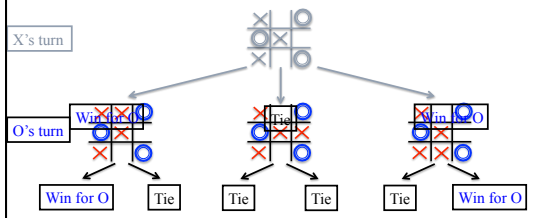
Look-ahead based Tic-Tac-Toe



Look-ahead based Tic-Tac-Toe

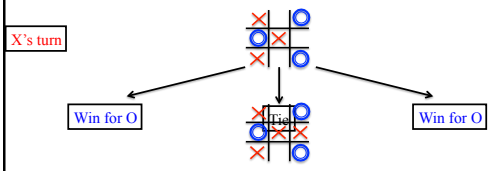


Look-ahead based Tic-Tac-Toe



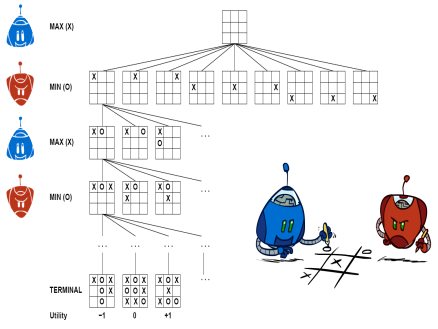
Each board in game tree gets unique game tree value (utility; -1/0/+1) under optimal rational play.

E.g. 0 for top board.



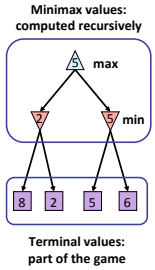
Approach:
Look first at bottom tree. Label bottom-most boards.
Then label boards one level up, according result of best possible move.
… and so on. Moving up layer by layer.x
Termed the Minimax Algorithm
Implemented as a depth-first search

Tic-Tac-Toe Game Tree



Adversarial Search (Minimax)

- Deterministic, zero-sum games:
  - Tic-tac-toe, chess, checkers
  - One player maximizes result
  - The other minimizes result
- Minimax search:
  - A state-space search tree
  - Players alternate turns
  - Compute each node's minimax value: the best achievable utility against a rational (optimal) adversary



Minimax Implementation

```
def max-value(state):
    initialize v = -∞
    for each successor of state:
        v = max(v, min-value(successor))
    return v
```

```
def min-value(state):
    initialize v = +∞
    for each successor of state:
        v = min(v, max-value(successor))
    return v
```

$$V(s) = \max_{s' \in \text{successors}(s)} V(s') \qquad V(s') = \min_{s \in \text{successors}(s')} V(s)$$

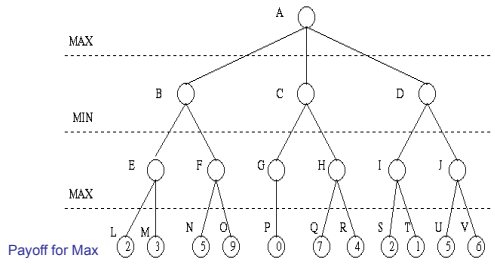
Minimax Implementation (Dispatch)

```
def value(state):
    if the state is a terminal state: return the state's utility
    if the next agent is MAX: return max-value(state)
    if the next agent is MIN: return min-value(state)
```

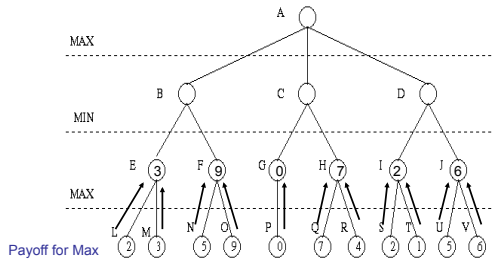
```
def max-value(state):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor))
    return v
```

```
def min-value(state):
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor))
    return v
```

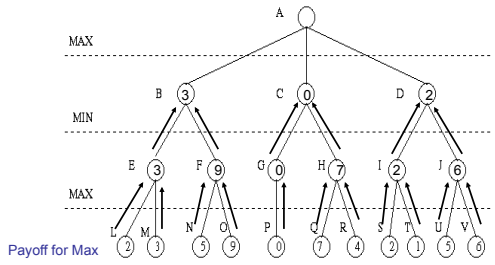
Minimax Algorithm



Minimax Algorithm (cont' d)

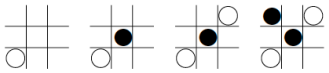


Minimax Algorithm (cont' d)



Well… Why not use a strategy / knowledge, as humans do?

1. If there is a winning move, make it.
2. If the opponent can win at a square by his next move, play that move.
3. Taking the central square is more important than taking other squares.
4. Taking corner squares is more important than taking squares on the edges.



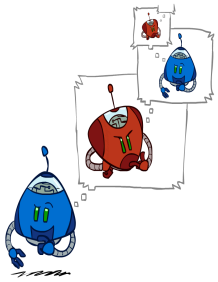
Oops!!

Consider Black uses the strategy…

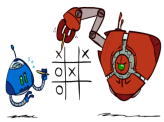
- So, although one can capture strategic knowledge of many games in high-level rules (at least to some extent), in practice any interesting game will revolve *precisely around the exceptions to those rules!*
- Issue has been studied for decades but research keeps coming back to game tree search (or most recently, game tree sampling).
- Currently only one exception: reinforcement learning for backgammon.
  - A very strong board evaluation function was learned in self-play.
    - Represented as a neural net.
    - Almost no search remained.

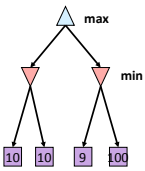
Minimax Efficiency

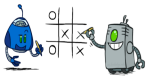
- How efficient is minimax?
  - Time: $O(b^m)$
  - Space: $O(bm)$
- Example: For chess, $b \approx 35$, $m \approx 100$
  - Exact solution is completely infeasible
  - But, do we need to explore the whole tree?



Minimax Properties







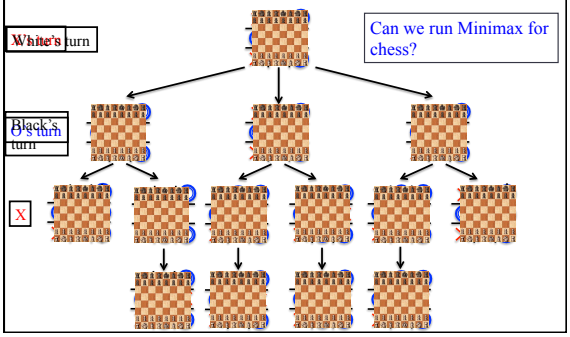
Optimal against a perfect player. Otherwise?

Demo Min vs. Optimal Max

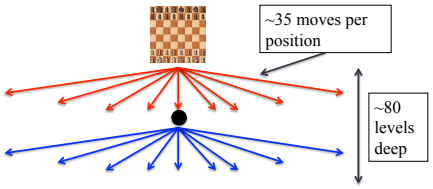
Demo: Min vs. Suboptimal Max

Resource Limits

Look-ahead based Chess



How big is this tree?



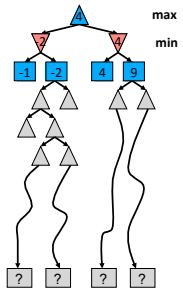
Approx. 10^120 > Number of atoms in the observable universe (10^80)

We can really only search a **tiny, miniscule faction** of this tree!

Around 60 x 10^9 nodes for 5 minute move. **Approx. 1 / 10^70 fraction.**

Resource Limits

- Solution: Depth-limited search
  - Instead, search only to a limited depth in the tree
  - Replace terminal utilities with an evaluation function for non-terminal positions
- Example:
  - Suppose we have 100 seconds
  - Can explore 10K nodes / sec
  - So can check 1M nodes per move
  - α-β reaches about depth 8 – decent chess program
- Guarantee of optimal play is gone
- More plies makes a BIG difference
- Use iterative deepening for an anytime algorithm



Depth Matters

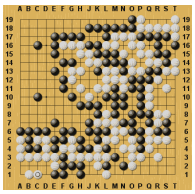
- Evaluation functions are always imperfect
- The deeper in the tree the evaluation function is buried, the less the quality of the evaluation function matters
- An important example of the tradeoff between complexity of features and complexity of computation

Video of Demo Limited Depth (2)

Video of Demo Limited Depth (10)

Minimax Limitations?

- Two important factors for success:
  - Deep look ahead
  - Good heuristic function
- Are there games where this is not feasible?



Minimax Limitations?

- Two important factors for success:
  - ~~Deep look ahead~~
  - Good heuristic function
- Are there games where this is not feasible?

Looking 14 levels ahead in Chess ≈ Looking 4 levels ahead in Go

Minimax Limitations?

- Two important factors for success:
  - ~~Deep look ahead~~
  - ~~Good heuristic function~~
- Are there games where this is not feasible?

Looking 14 levels ahead in Chess ≈ Looking 4 levels ahead in Go

Moves have extremely delayed effects

Minimax Limitations?

- Two important factors for success:
  - ~~Deep look ahead~~
  - ~~Good heuristic function~~
- Are there games where this is not feasible?

Looking 14 levels ahead in Chess ≈ Looking 4 levels ahead in Go

Moves have extremely delayed effects

Minimax players for GO were very weak until 2007…but now play at master level. Since 2015, at Expert level

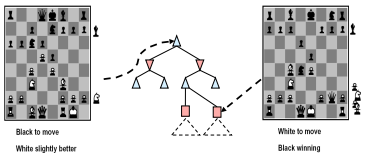
Evaluation Functions

Evaluation Function

- Performed at search cutoff point
- Must have same terminal/goal states as utility function
- Tradeoff between accuracy and time → reasonable complexity
- Accurate
  - Performance of game-playing system dependent on accuracy/goodness of evaluation
  - Evaluation of nonterminal states strongly correlated with actual chances of winning

Evaluation Functions

- Evaluation functions score non-terminals in depth-limited search

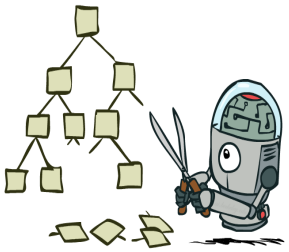


- Ideal function: returns the actual minimax value of the position
- In practice: typically weighted linear sum of features:

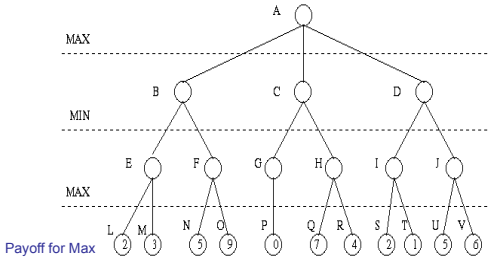
$$Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$$

- e.g. $f_1(s)$ = (num white queens – num black queens), etc.

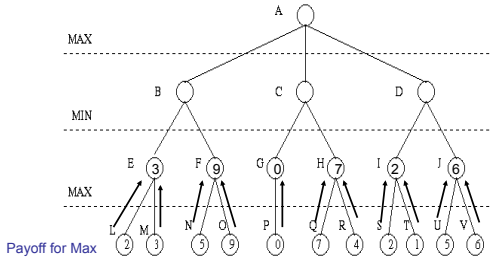
Game Tree Pruning



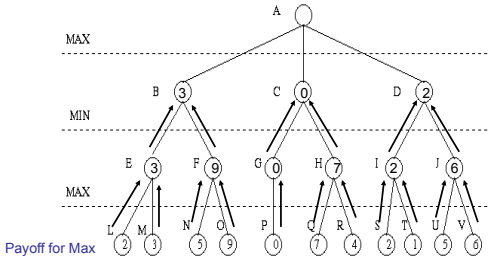
Minimax Algorithm with α-β Pruning



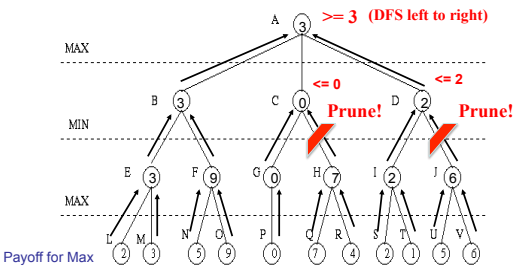
Minimax Algorithm with α-β Pruning



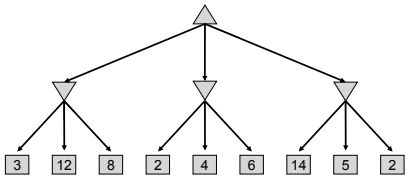
Minimax Algorithm with α-β Pruning



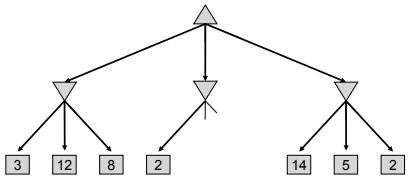
Minimax Algorithm with α-β Pruning



Minimax Example

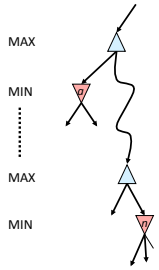


Minimax with Pruning



Alpha-Beta Pruning

- General configuration (MIN version)
  - We're computing the MIN-VALUE at some node *n*
  - We're looping over *n*'s children
  - *n*'s estimate of the childrens' min is dropping
  - Who cares about *n*'s value? MAX
  - Let *a* be the best value that MAX can get at any choice point along the current path from the root
  - If *n* becomes worse than *a*, MAX will avoid it, so we can stop considering *n*'s other children (it's already bad enough that it won't be played)
- MAX version is symmetric



Alpha-Beta Implementation

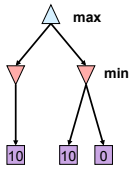
α: MAX's best option on path to root
β: MIN's best option on path to root

```
def max-value(state, α, β):
    initialize v = -∞
    for each successor of state:
        v = max(v,
          value(successor, α, β))
        if v ≥ β return v
        α = max(α, v)
    return v
```

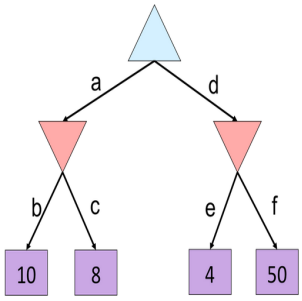
```
def min-value(state , α, β):
    initialize v = +∞
    for each successor of state:
        v = min(v,
          value(successor, α, β))
        if v ≤ α return v
        β = min(β, v)
    return v
```

Alpha-Beta Pruning Properties

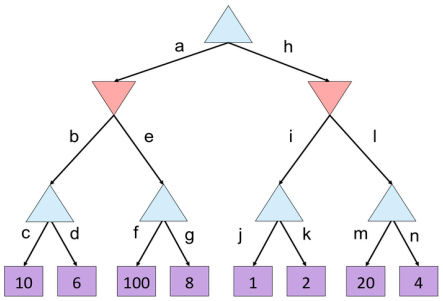
- This pruning has no effect on Minimax value computed for the root!
- Good child ordering improves effectiveness of pruning
- With "perfect ordering":
  - Time complexity drops to $O(b^{m/2})$
  - Doubles solvable depth!
  - Full search of, e.g. chess, is still hopeless…
- This is a simple example of metareasoning (computing about what to compute)



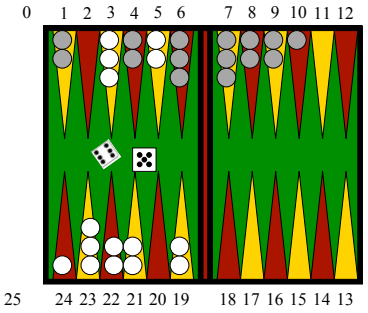
Alpha-Beta Quiz



Alpha Beta Quiz 2



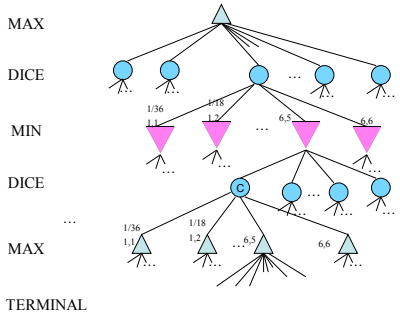
When Chance is involved: Backgammon Board



Expectiminimax

- Generalization of Minimax for games with chance nodes
- Examples: Backgammon, bridge
- Calculates expected value where probability is taken (More in Section 3 of this class)
- Over all possible dice rolls/chance events
  - Max and Min nodes determined as before
  - Chance nodes evaluated as weighted average

Game Tree for Backgammon

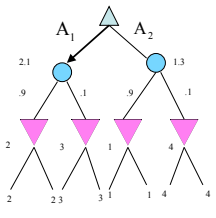


Expectiminimax

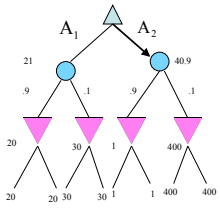
Expectiminimax(n) =

Utility(n)	for n, a terminal state
$max_{s \in Succ(n)}$ expectiminimax(s)	for n, a Max node
$min_{s \in Succ(n)}$ expectiminimax(s)	for n, a Min node
$\Sigma_{s \in Succ(n)} P(s) *$ expectiminimax(s)	for n, a chance node

Expectiminimax



.9 * 2 + .1 * 3 = 2.1



Small chance at high payoff wins. But, not necessarily the best thing to do!

Next Time: Logic and Reasoning!

- NOW – In Class Assignment 2