Search in Games

CompSci 370
Ron Parr
Duke University
Department of Computer Science

Why Study Games?

- Many human activities can be modeled as games
  - Negotiations
  - Bidding
  - TCP/IP
  - Military confrontations
  - Pursuit/Evasion
- Games are used to train the mind
  - Human game-playing, animal play-fighting

Why Are Games Good for AI?

- Games typically have concise rules
- Well-defined starting and end points
- Sensing and effecting are simplified
  - Not true for sports games
  - See robocup
- Games are fun!
- Downside: Getting taken seriously (not)
  - See robo search and rescue

Some History of Games in AI

- Computer games have been around almost as long as computers (perhaps longer)
  - Chess: Turing (and others) in the 1950s
  - Checkers: Samuel, 1950s learning program
- Usually start with naïve optimism
- Follow with naïve pessimism
- Simon: Predicted computer chess champ by 1967
- Many, e.g., Kasparov, predicted that a computer would *never* be champion

Checkers: Tinsley vs. Chinook

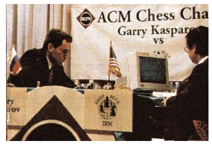
Name: Marion Tinsley
Profession: Taught mathematics
Hobby: Checkers
Record: Over 42 years loses only 3 games of checkers
World champion for over 40 years

Mr. Tinsley suffered his 4th and 5th losses against Chinook (1994)

Chinook

- First computer to become official world champion of Checkers!

Chess: Kasparov vs. Deep Blue



Kasparov		Deep Blue
5’10”	**Height**	6’ 5”
176 lbs	**Weight**	2,400 lbs
34 years	**Age**	4 years
50 billion neurons	**Computers**	32 RISC processors + 256 VLSI chess engines
2 pos/sec	**Speed**	200,000,000 pos/sec
Extensive	**Knowledge**	Primitive
Electrical/chemical	**Power Source**	Electrical
Enormous	**Ego**	None

1997: Deep Blue wins by 3 wins, 1 loss, and 2 draws

Jonathan Schaeffer

Chess: Kasparov vs. Deep Junior

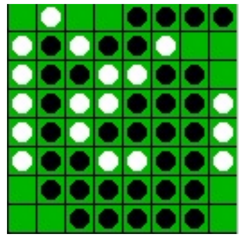
Deep Junior

8 CPU, 8 GB RAM, Win 2000
2,000,000 pos/sec
Available at ~$1000

(Note: Lesser hardware, but more clever software)

August 2, 2003: Match ends in a 3/3 tie!

Othello: Murakami vs. Logistello



Takeshi Murakami
World Othello Champion

1997: The Logistello software crushed Murakami by 6 games to 0

AlphaGo

- Go believed to be qualitatively different from other games due to enormous branching factor – brute force can't search very deep
- Played best living human go player, Lee Sedol, (possibly one of the strongest go players ever) in March 2016, winning 4/5 games
- Used between 1K and 2K CPUs, 150-300 GPUs

- Play was described as surprising and original; Made moves that were unexpected at first, but made sense in hindsight
- Played the 2017 best go player (Ke Jie) in May 2017, winning 3/3
- AlphaGo Zero learns entirely from self-play, stronger, less computation

“Solved” Games

- A game is solved if an optimal strategy is known
- Strongly solved = solved for all positions (tic tac toe)
- Weakly solved = solved for some (e.g. starting) positions (checkers – spoiler alert: it’s a tie)

- Why bother playing solved games?

Source: Wikipedia

Simple Game Setup

- Most commonly, we study games that are:
  - 2 player
  - Alternating
  - Zero-sum
  - Perfect information
- Examples: Checkers, chess, backgammon
- Assumptions can be relaxed at some expense
- Economics studies case where #of agents is very large
  - Individual actions don’t change the dynamics

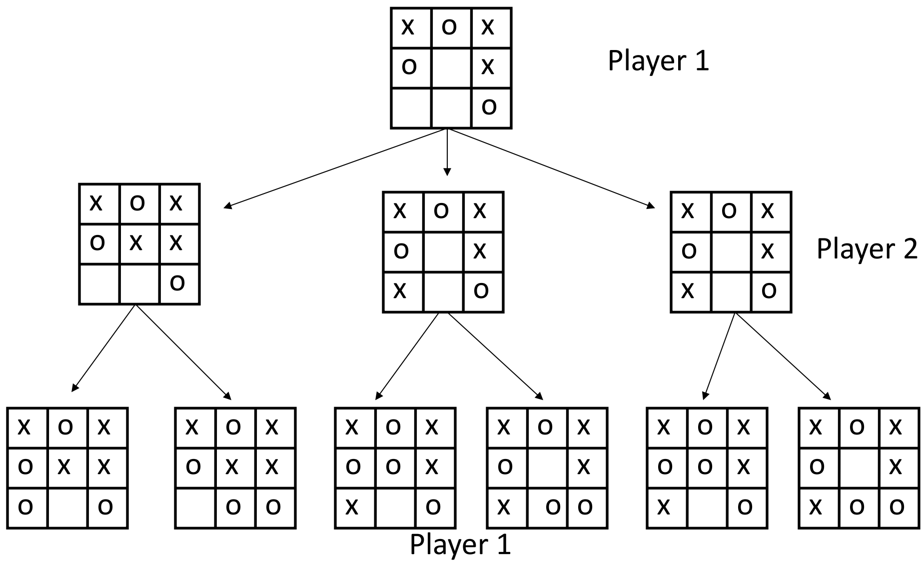
Zero Sum Games

- Assign values to different outcomes
- Win = 1, Loss = -1
- With zero sum games every gain comes at the other player's expense
- Sum of both player's scores must be 0
- Are any games truly zero sum?

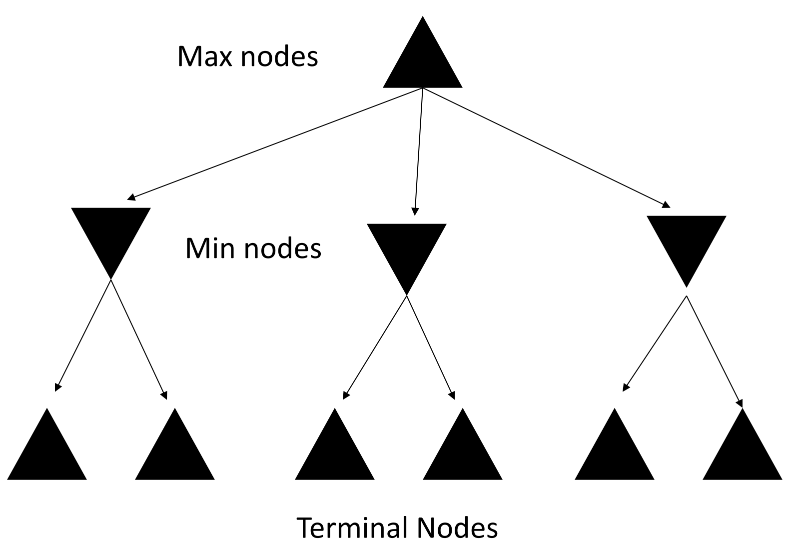
Characterizing Games

- Two-player alternating move games are very much like search
  - Initial state
  - Successor function
  - Terminal test
  - Objective function (heuristic function)
- Unlike search
  - Terminal states are often a large set
  - Full search to terminal states usually impossible

Game Trees



Game Trees (abstracted)



Minimax

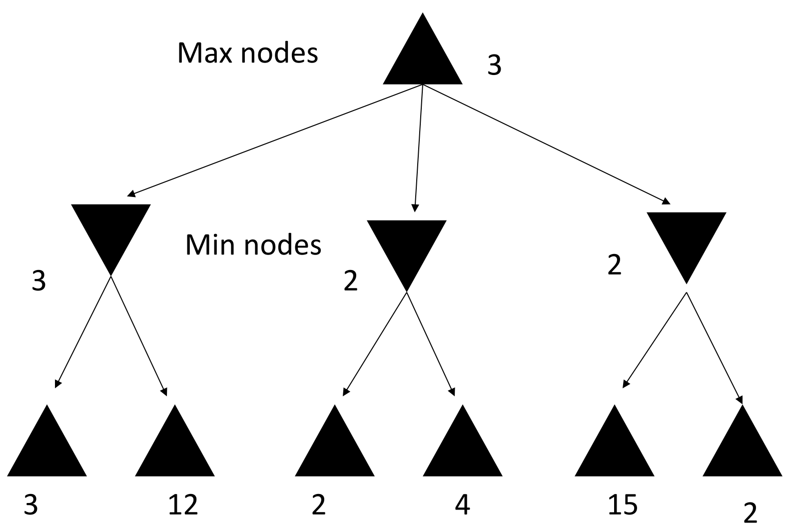
- Max player tries to maximize his return
- Min player tries to minimize max's return
- This is optimal for both (assuming zero sum)

$$\text{minimax}(n_{\max}) = \max_{s \in \text{succesors}(n)} \text{minimax}(s)$$

$$\text{minimax}(n_{\min}) = \min_{s \in \text{succesors}(n)} \text{minimax}(s)$$

Note: Trivial to implement as two mutually recursive functions

Minimax Values



Minimax Properties

- Minimax can be run depth first
  - Time $O(b^m)$
  - Space $O(bm)$
- Assumes that opponent plays optimally
- Based on a worst-case analysis
- What if this is incorrect?

Minimax in the Real World

- Search trees are too big
- Alternating turns double depth of the search
  - 2 ply = 1 full turn
- Branching factors are too high
  - Chess: ~35 (per turn)
  - Go: ~361 (per turn)
- Full search from start to end never terminates in non-trivial games

Real-Time decisions

- The state space is enormous: only a tiny fraction of this space can be explored within the time limit (e.g., 3min for chess)

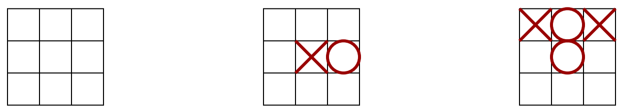
1. Using the current state as the initial state, build the game tree uniformly to **the maximal depth h (called horizon) feasible within the time limit**
2. Evaluate the states of the leaf nodes
3. Back up the results from the leaves to the root and pick the best action assuming the worst from MIN
4. Repeat

Evaluation Function

- Function e: state s $\rightarrow$ number e(s)
- e(s) is a heuristic that estimates how favorable s is for MAX
- e(s) > 0 means that s is favorable to MAX (the larger the better)
- e(s) < 0 means that s is favorable to MIN
- e(s) = 0 means that s is neutral

Example: Tic-tac-Toe

e(s) = number of rows, columns,
and diagonals open for MAX
- number of rows, columns,
and diagonals open for MIN



8-8 = 0   6-4 = 2   3-3 = 0

Construction of an Evaluation Function

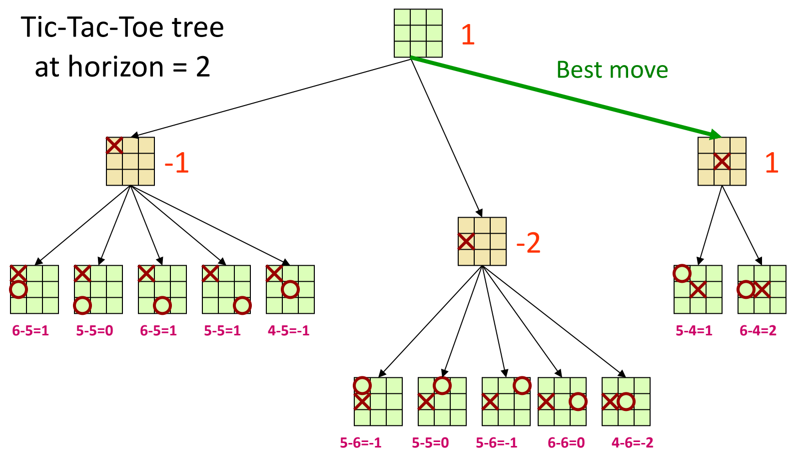
- Usually a weighted sum of “features”:

$$e(s) = \sum_{i=1}^{n} w_i f_i(s)$$

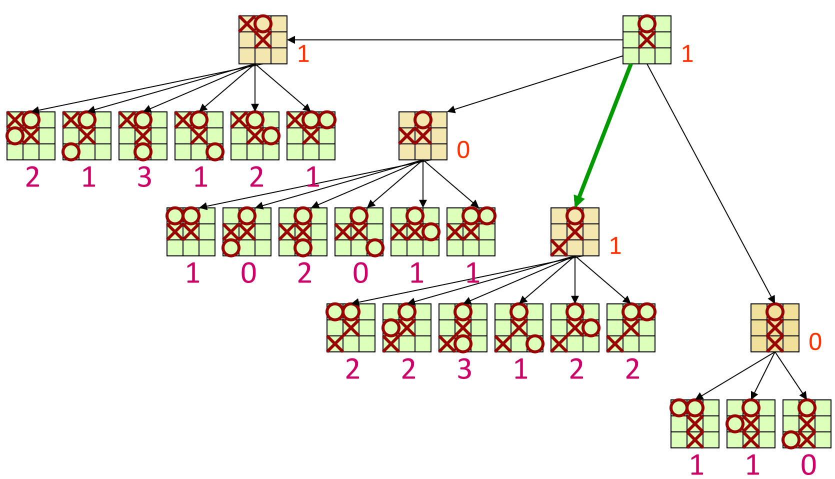
- Features may include
  - Number of pieces of each type
  - Number of possible moves
  - Number of squares controlled

Backing up Values

Tic-Tac-Toe tree
at horizon = 2



Continuation



Why use backed-up values?

- If e is to be trusted in the first place, then the backed-up value is a better estimate of how favorable STATE(N) is than e(STATE(N))

- Why? Presumption that e() is more accurate closer to the end of the game

Search Control Issues

- Horizon effects
  - Something interesting is just beyond the horizon?
  - How do you know?
- When to generate more nodes?
- If you selectively extend your frontier, where?
- How do you allocate a fixed amount of game time?

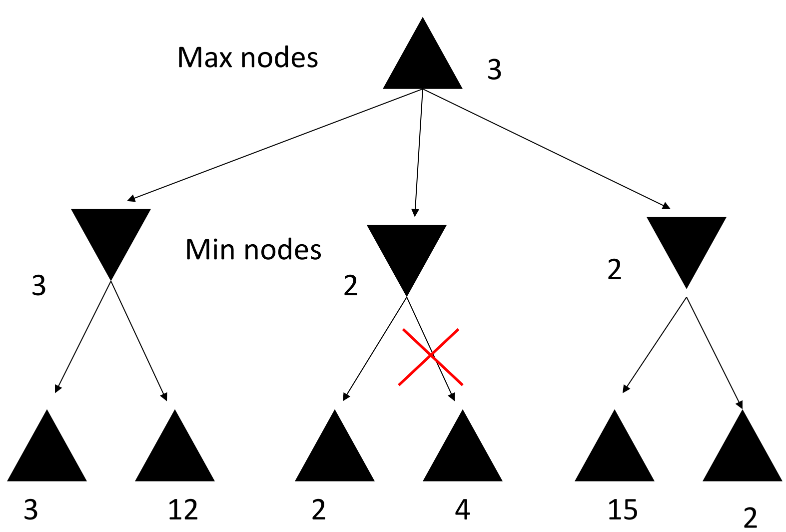
- Many extensions to minimax to address this, but most important improvement is alpha-beta pruning

Pruning

- *The most important search control method is figuring out which nodes you don't need to expand*

- Use the fact that we are doing a worst-case analysis to our advantage
  - Cut off search at min nodes when max can already force a better outcome (for max)
  - Cut off search at max nodes when max can already force a better outcome (for min)

Alpha-beta pruning



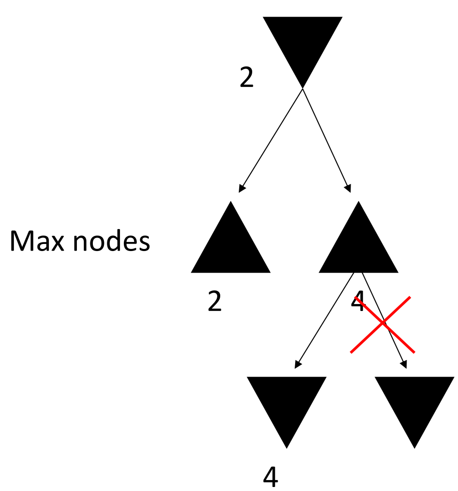
How to prune

- We still do (bounded) DFS
- Expand at least one path to the “bottom”

- If current node is **max** node, and **min** can force a *lower* value, then prune siblings

- If current node is min node, and max can force a *higher* value, then prune siblings

Max node pruning



Implementing alpha-beta

```
max_value(state, alpha, beta)
if cutoff(state) then return eval(state)
v = -∞
for each s in successors(state) do
  v = max(v, min_value(s, alpha, beta))
  if v ≥ beta then return v
  alpha = max(alpha,v)
end
return v
```

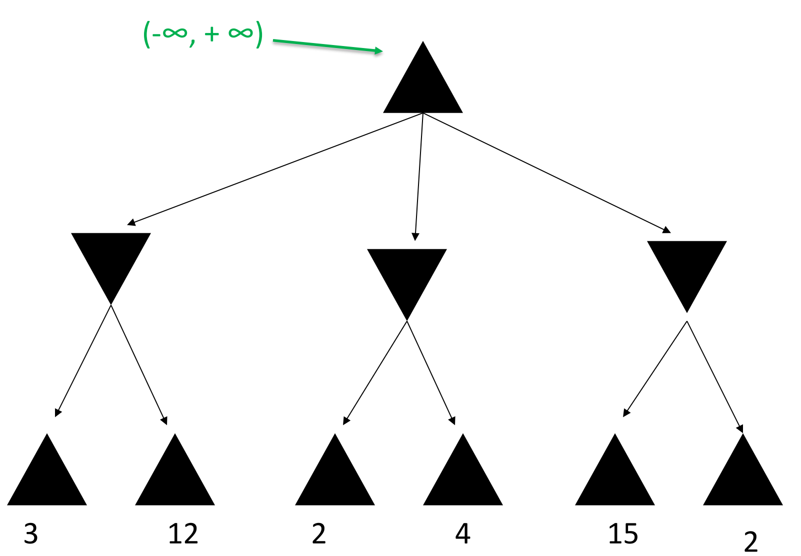
beta=value of best guaranteed option available to min

alpha=value of best guaranteed option available to max

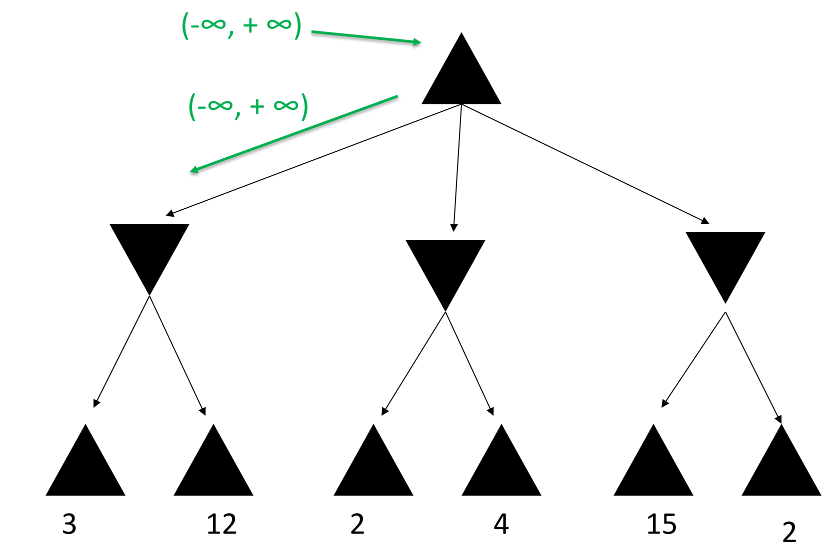
```
min_value(state, alpha, beta)
if cutoff(state) then return eval(state)
v = ∞
for each s in successors(state) do
  v = min(v, max_value(s, alpha, beta))
  if v ≤ alpha then return v
  beta = min(beta,v)
end
return v
```

Call:
max_value(root, -∞, + ∞)

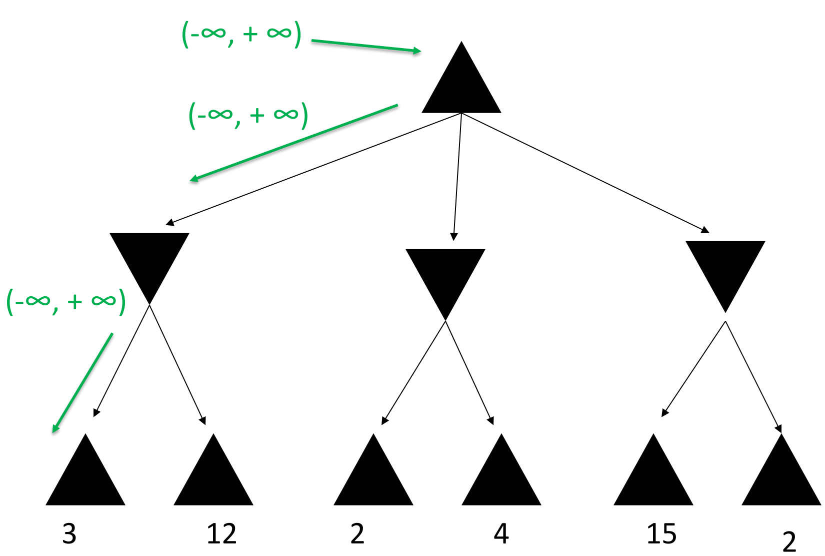
Alpha-beta in action



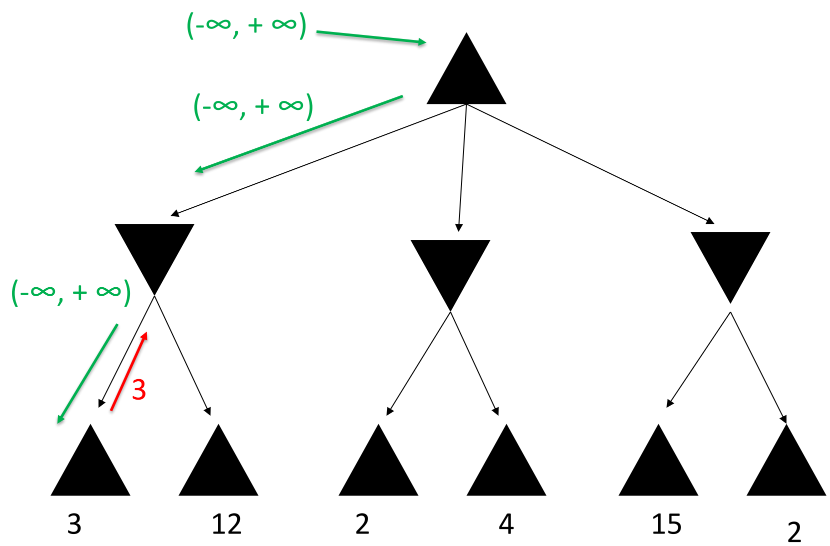
Alpha-beta in action



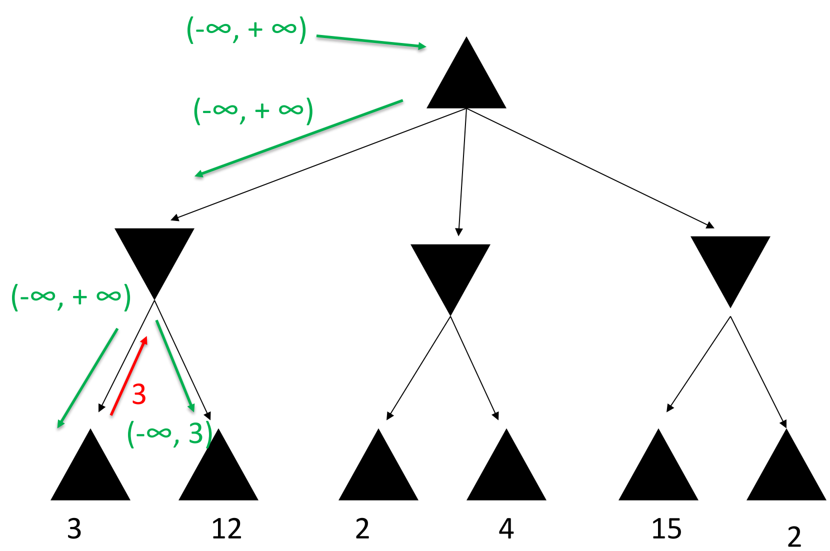
Alpha-beta in action



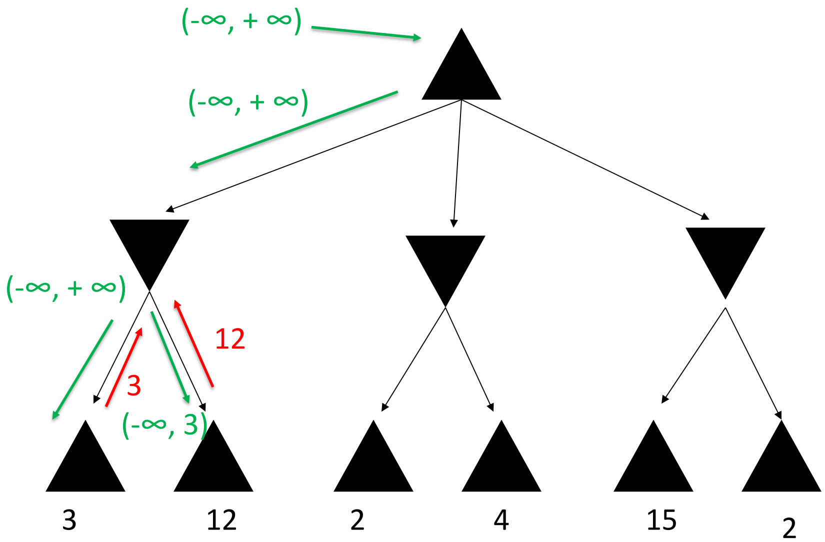
Alpha-beta in action



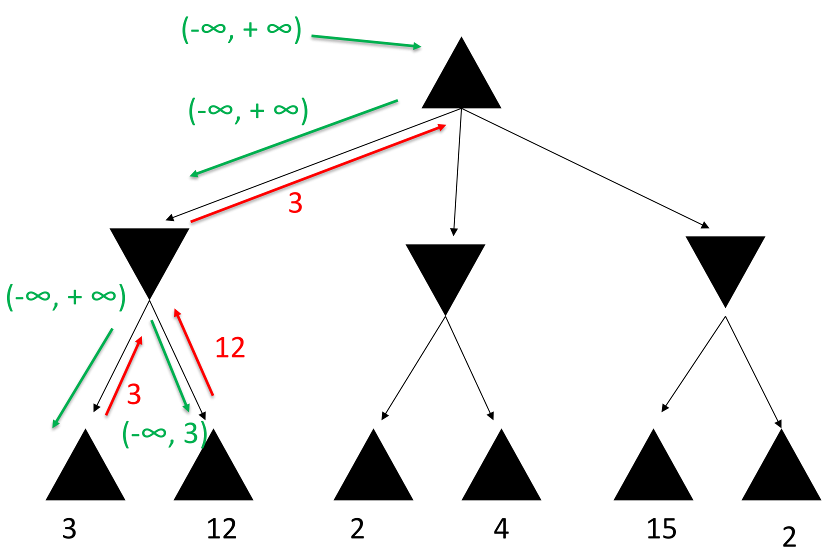
Alpha-beta in action



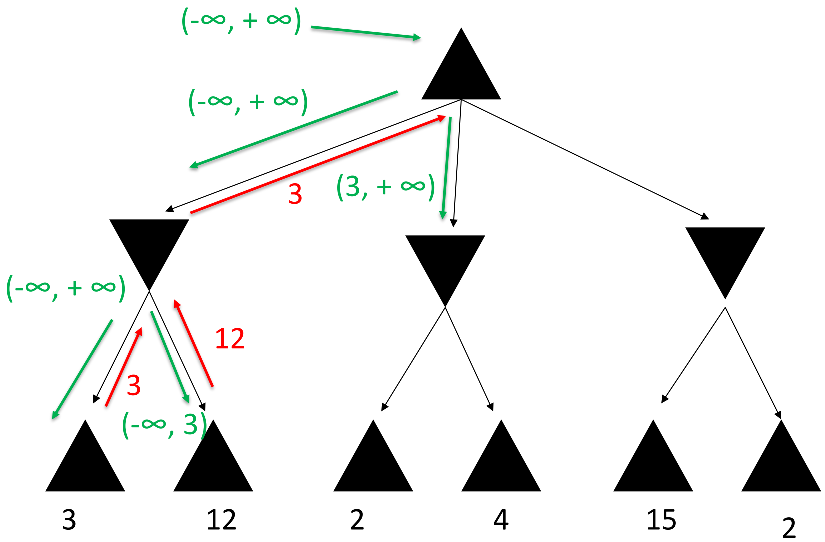
Alpha-beta in action



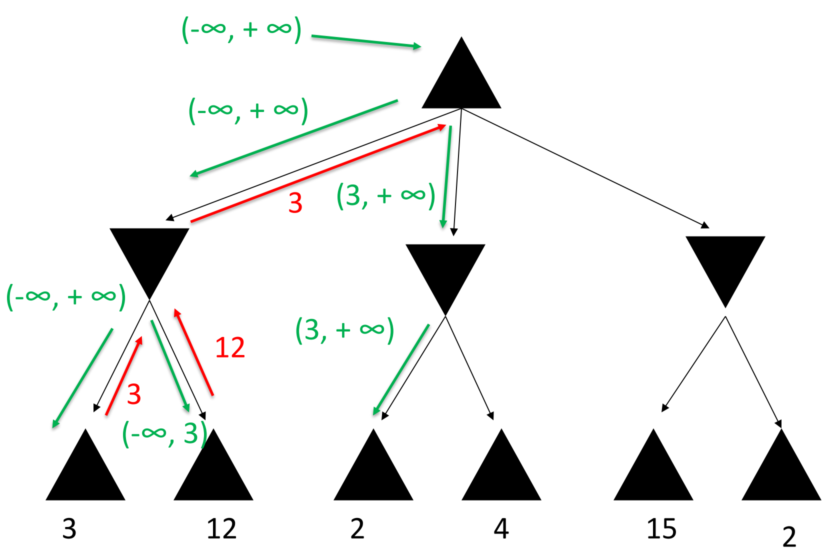
Alpha-beta in action



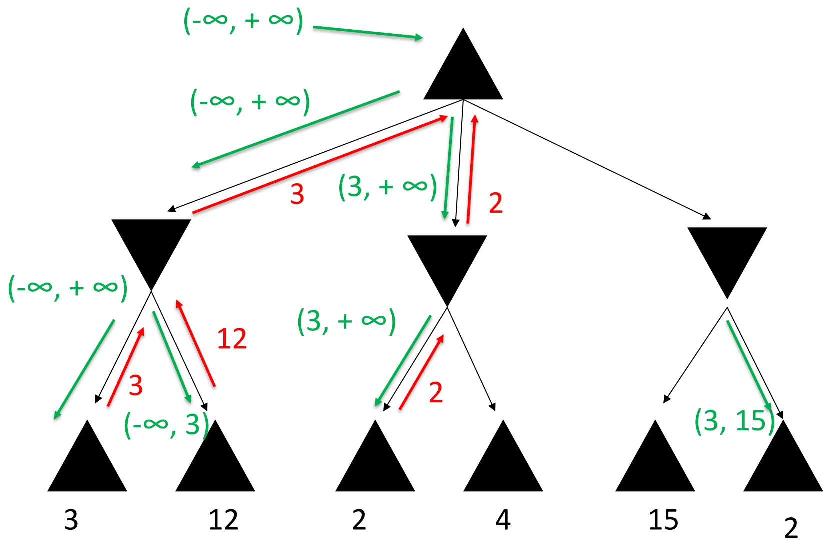
Alpha-beta in action



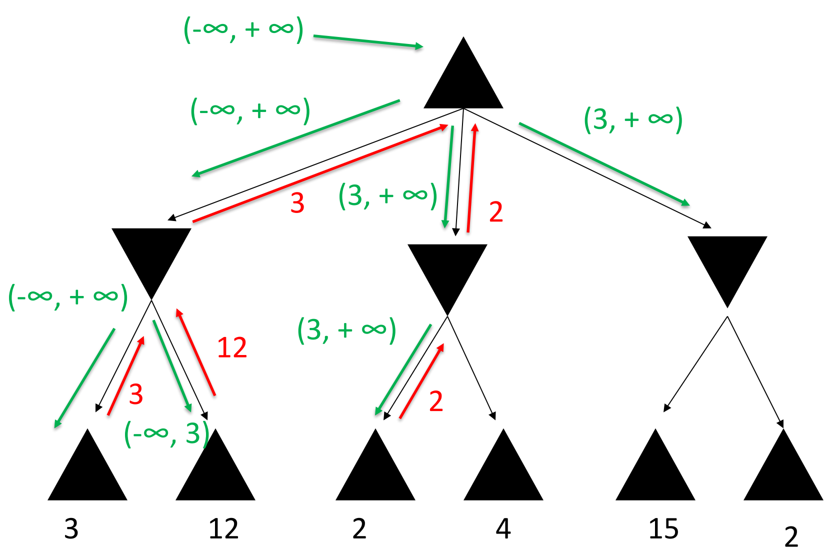
Alpha-beta in action



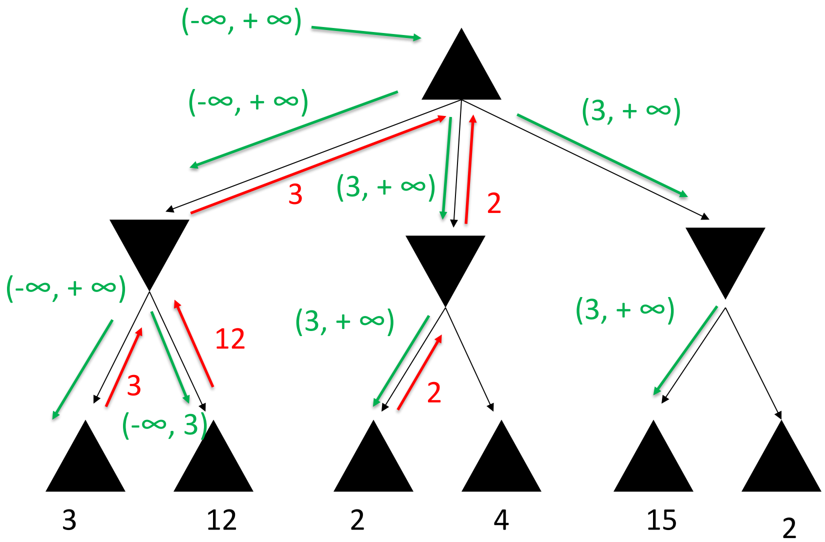
Alpha-beta in action



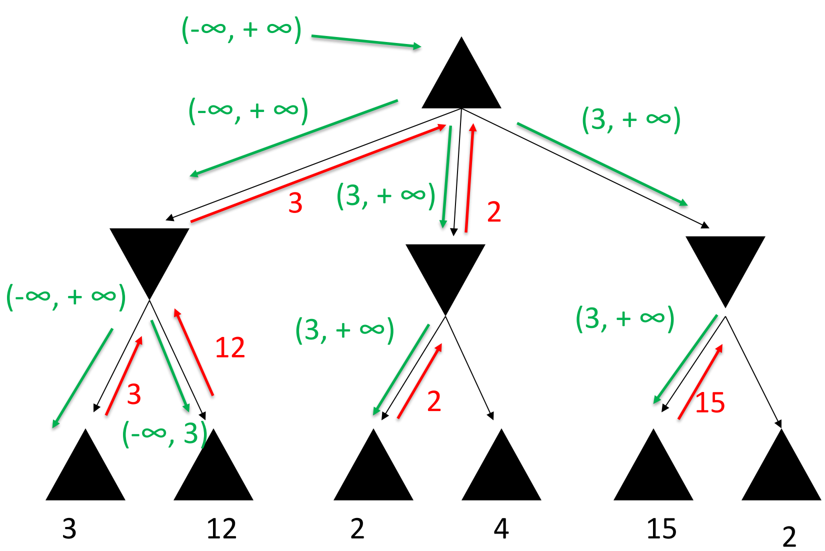
Alpha-beta in action



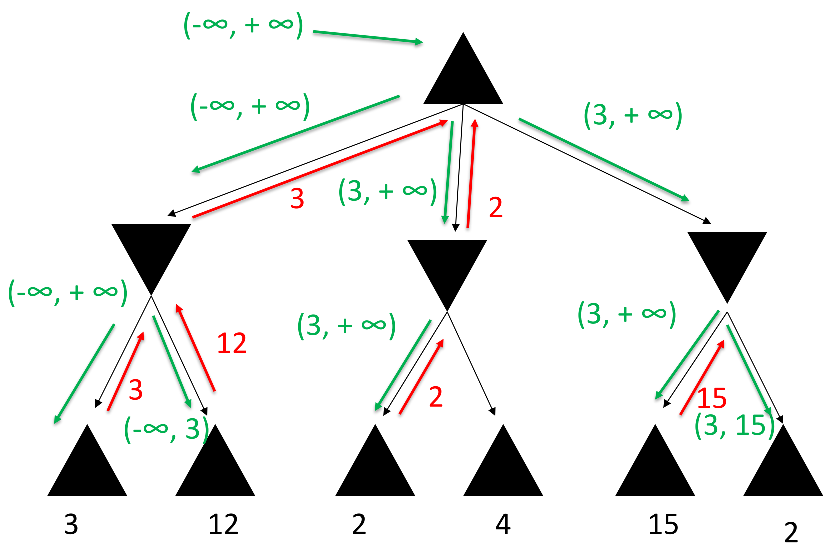
Alpha-beta in action



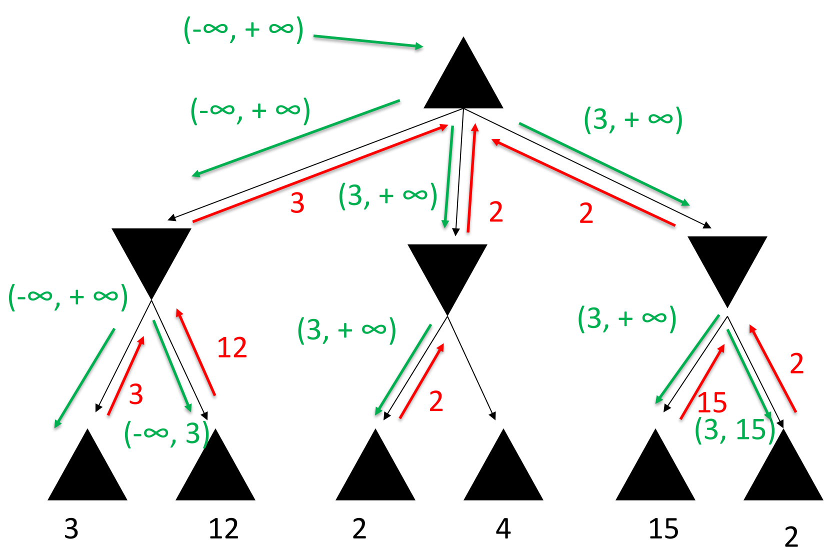
Alpha-beta in action



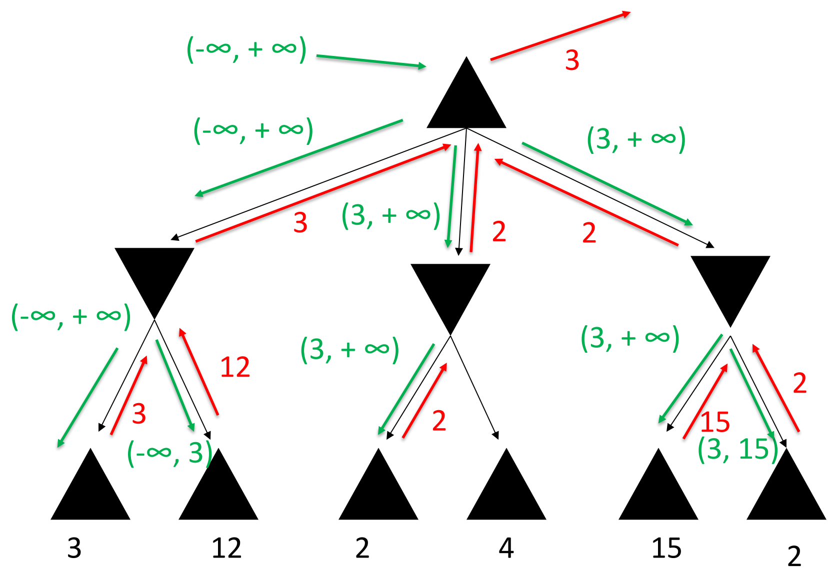
Alpha-beta in action



Alpha-beta in action

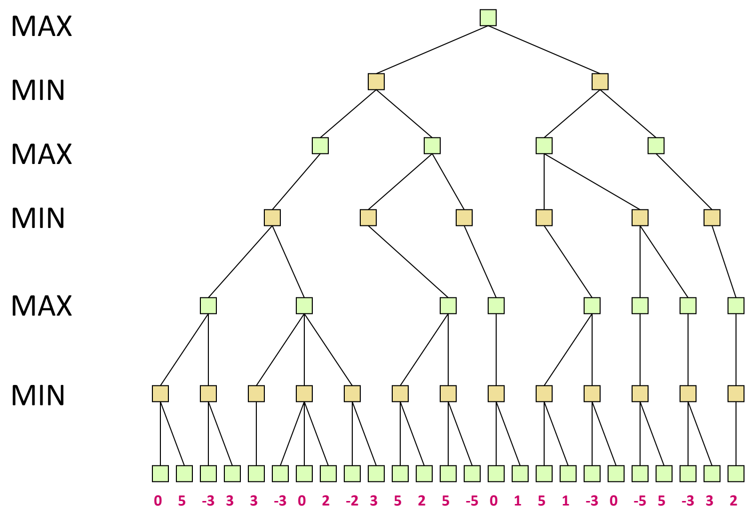


Alpha-beta in action

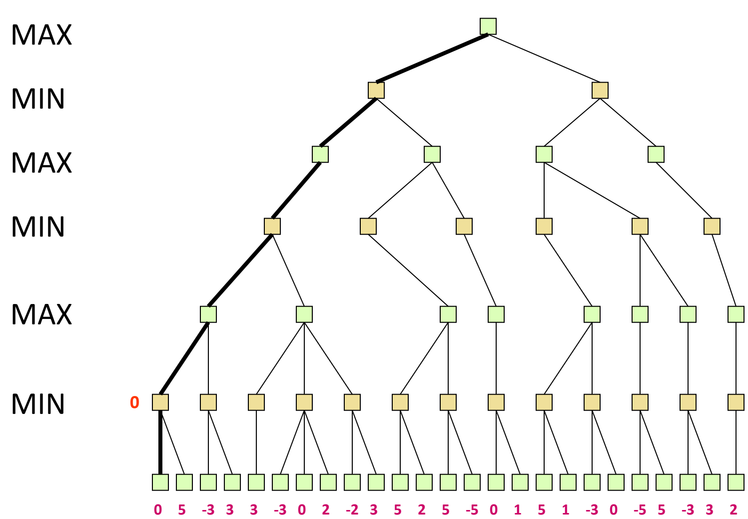


A Larger Example
(w/o Alpha-Beta values)

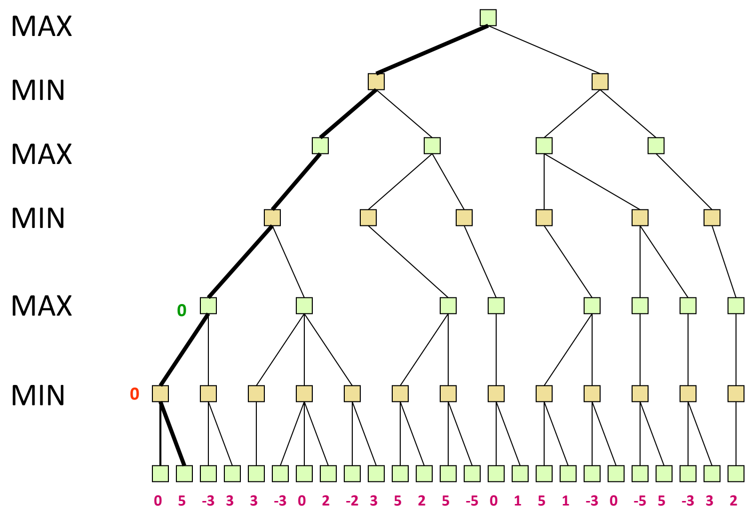
Example



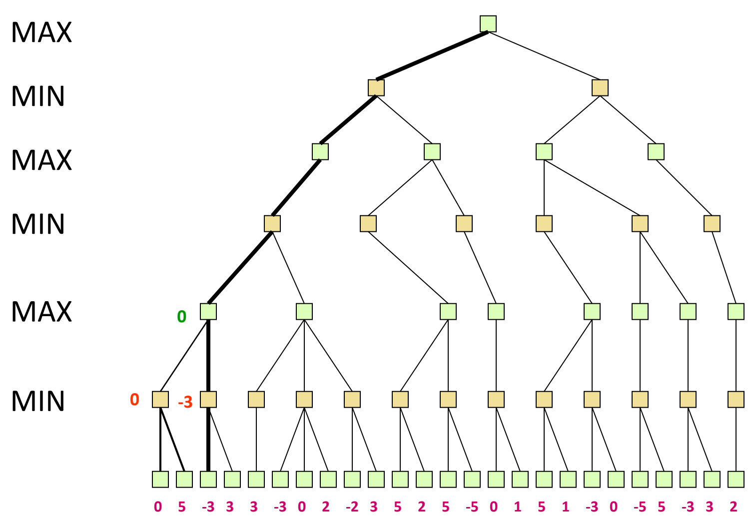
Example



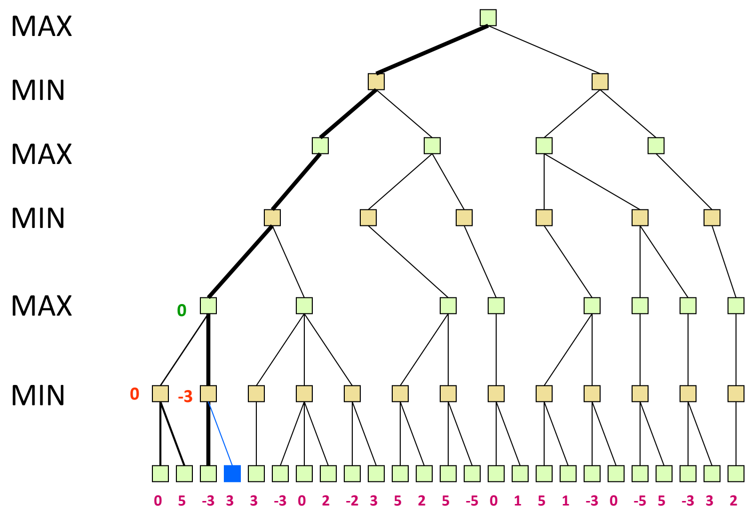
Example



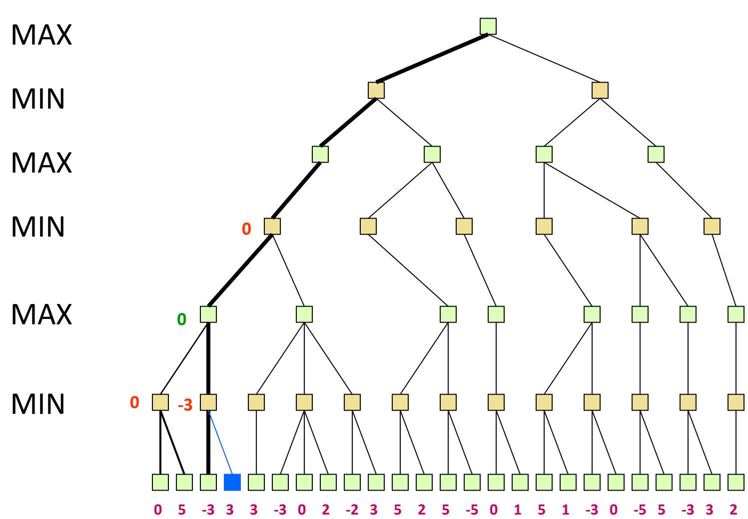
Example



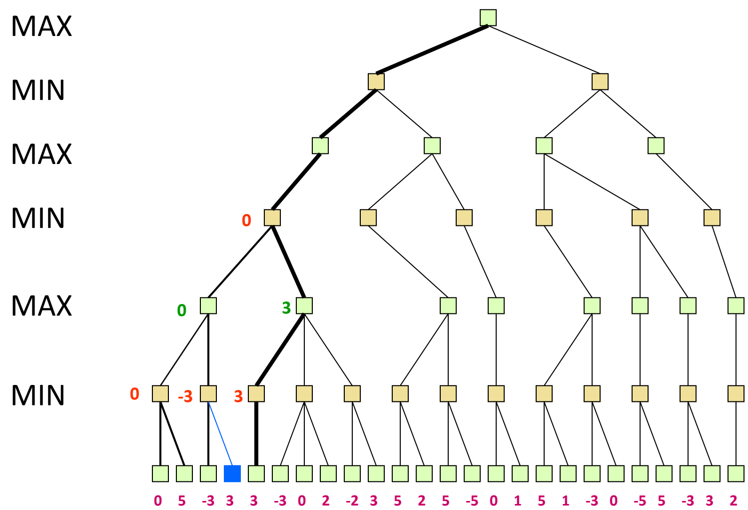
Example



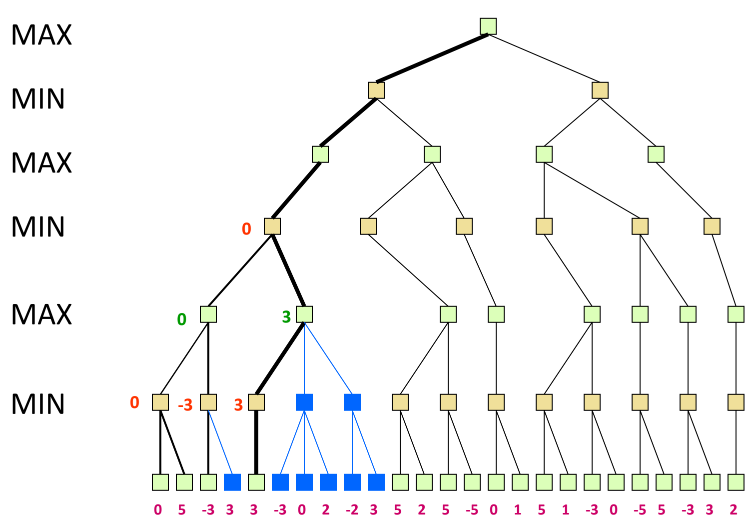
Example



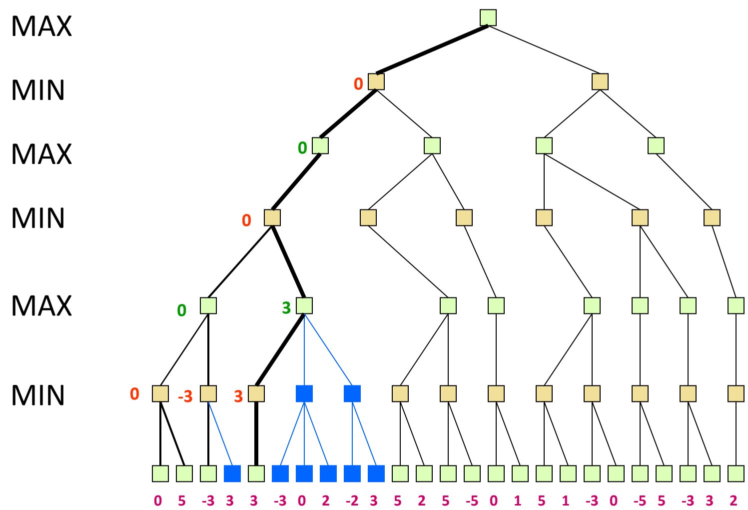
Example



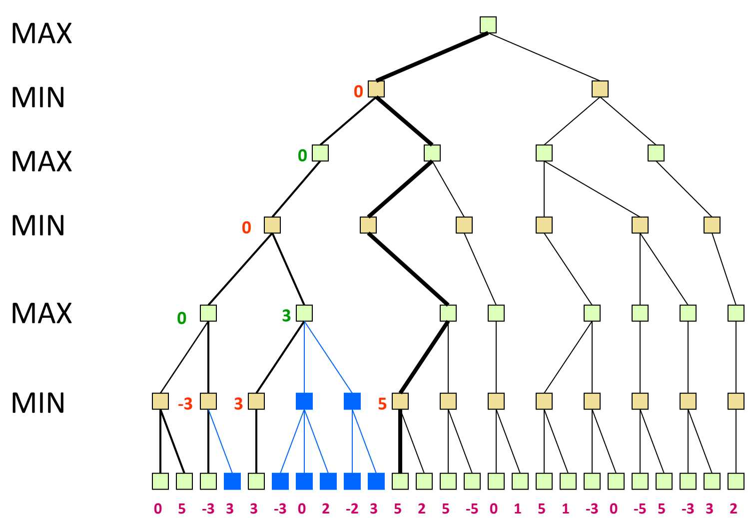
Example

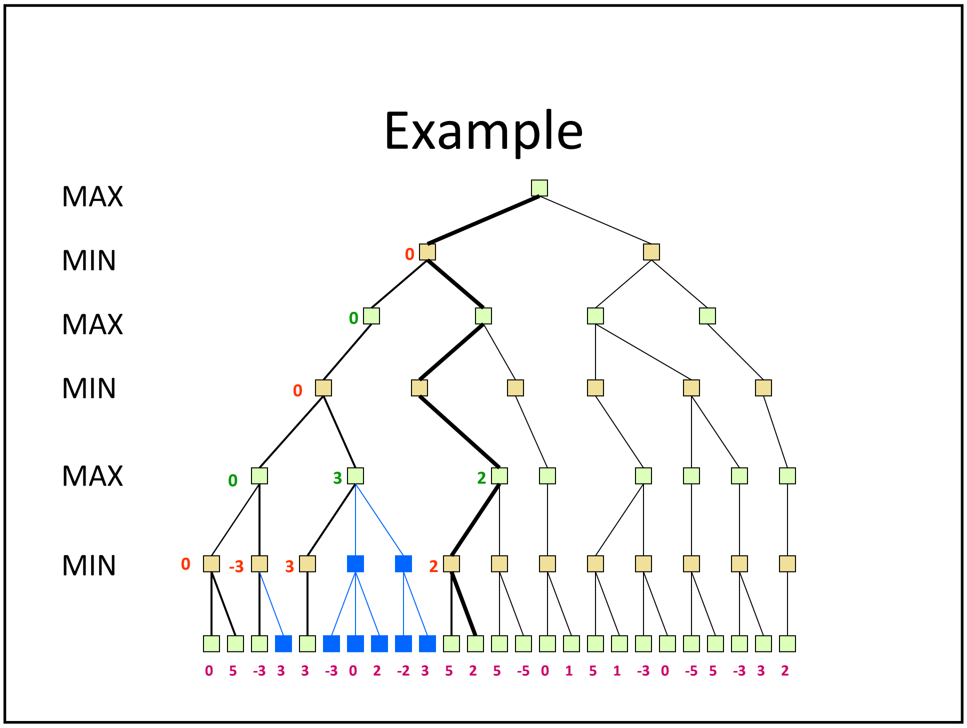


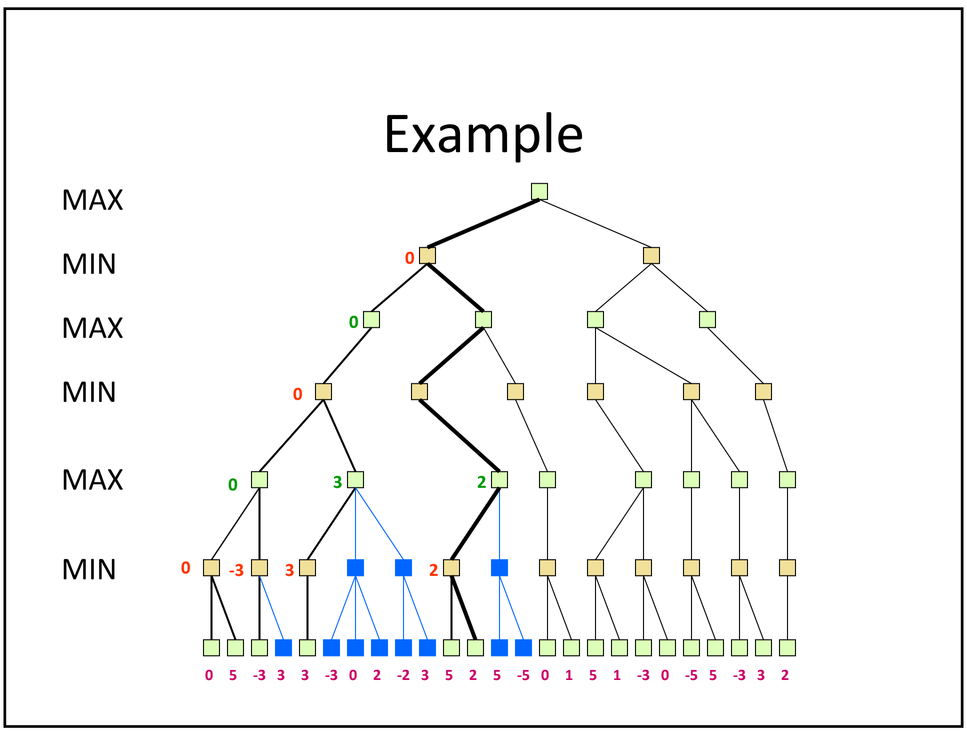
Example



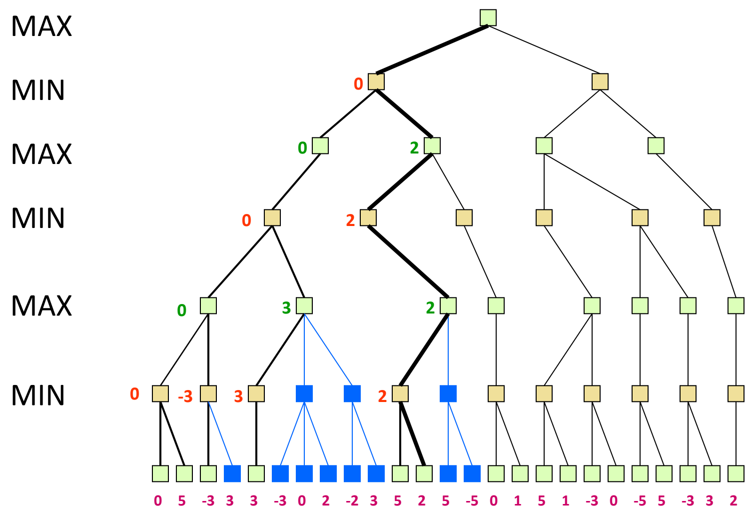
Example



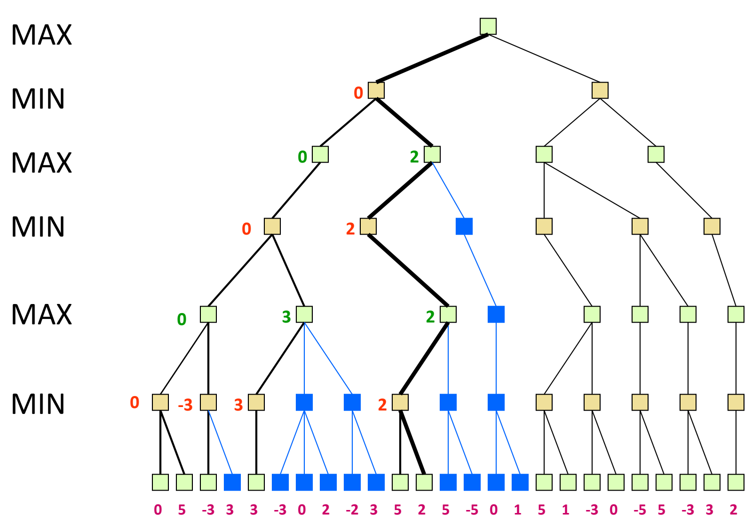

Example
MAX
MIN
MAX
MIN
MAX
MIN
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2


Example
MAX
MIN
MAX
MIN
MAX
MIN
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

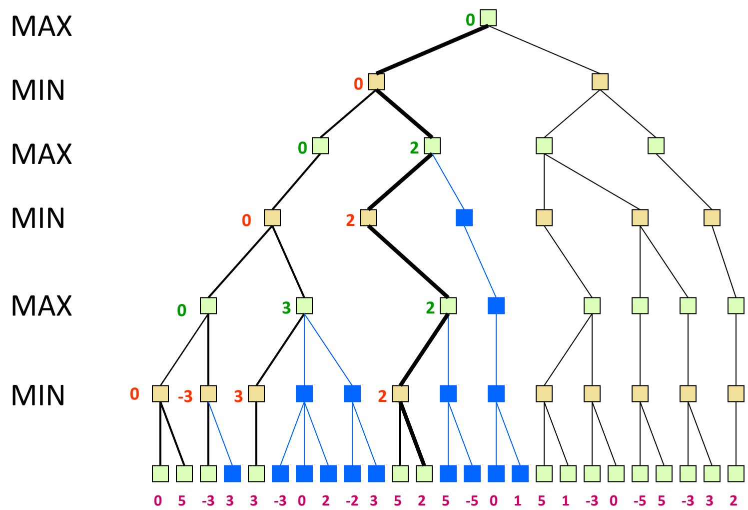
Example



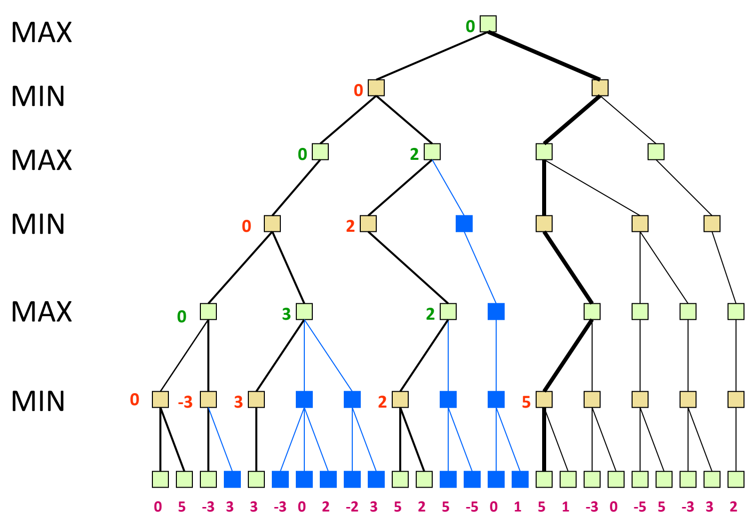
Example



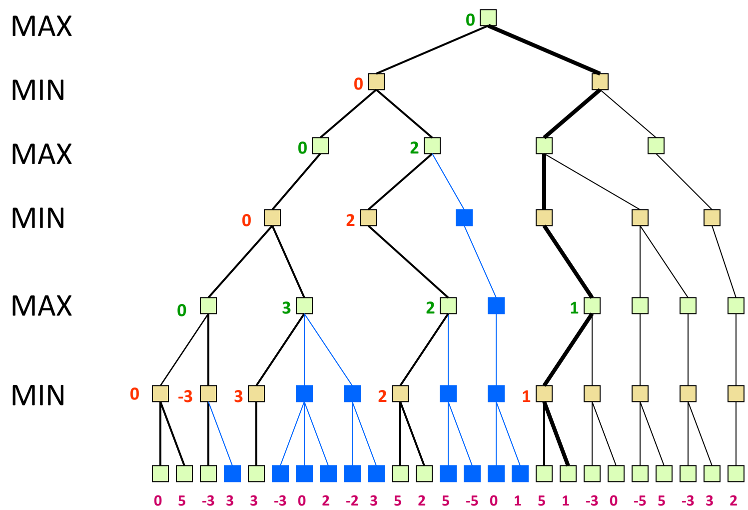
Example



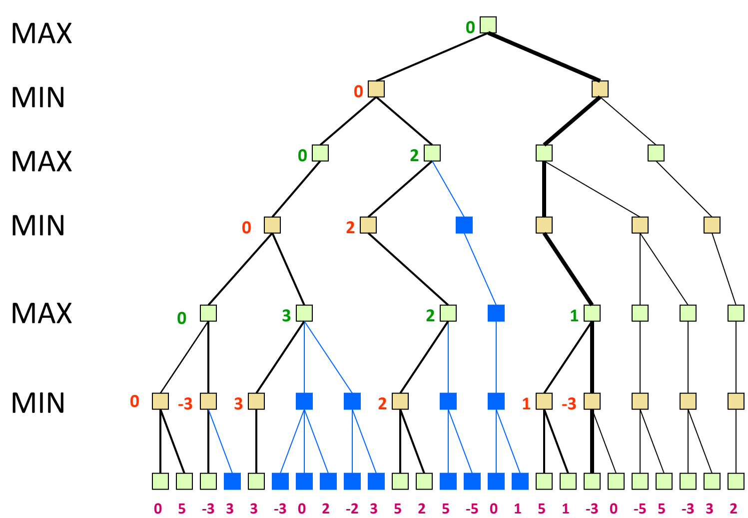
Example



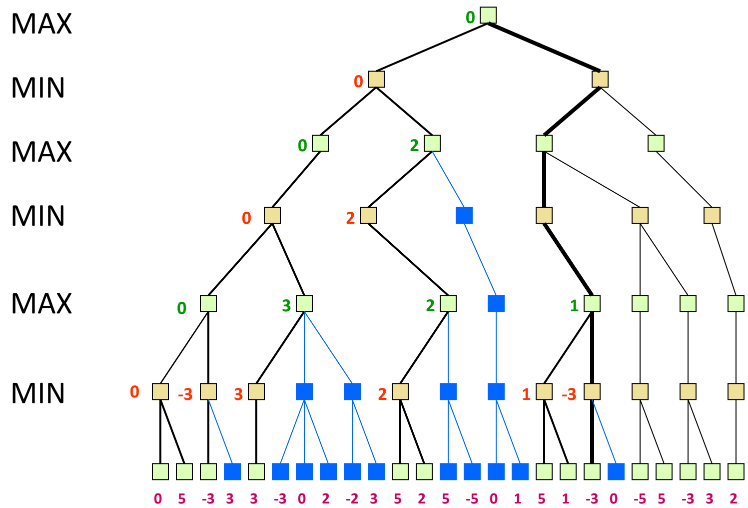
Example



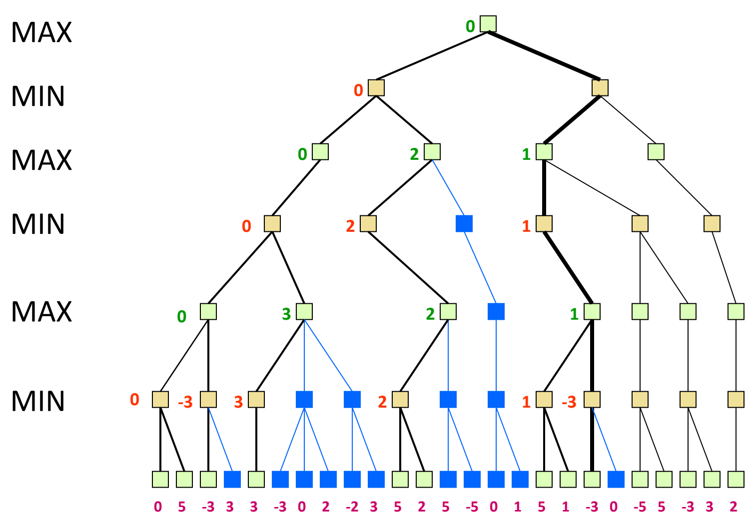
Example



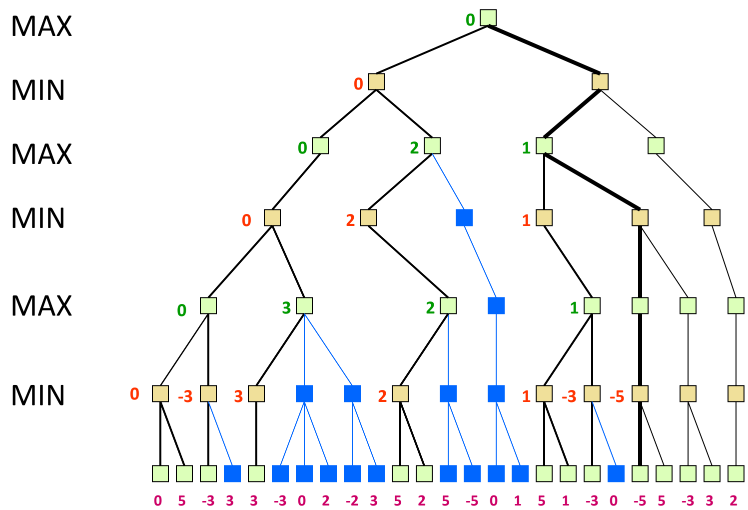
Example



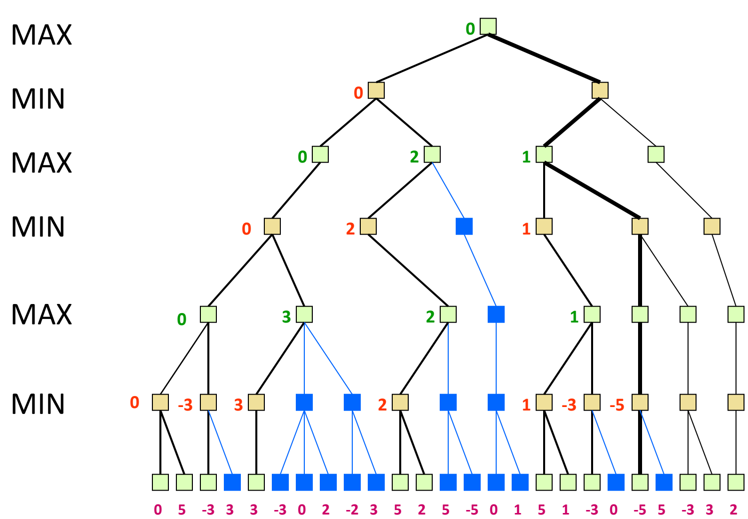
Example



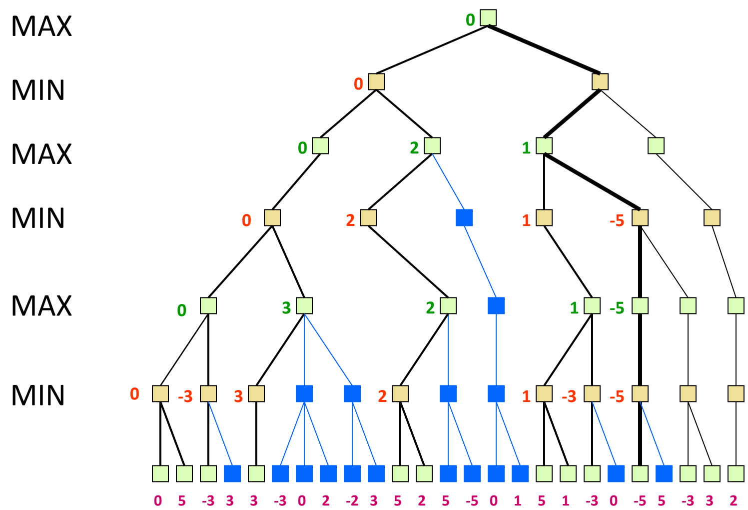
Example



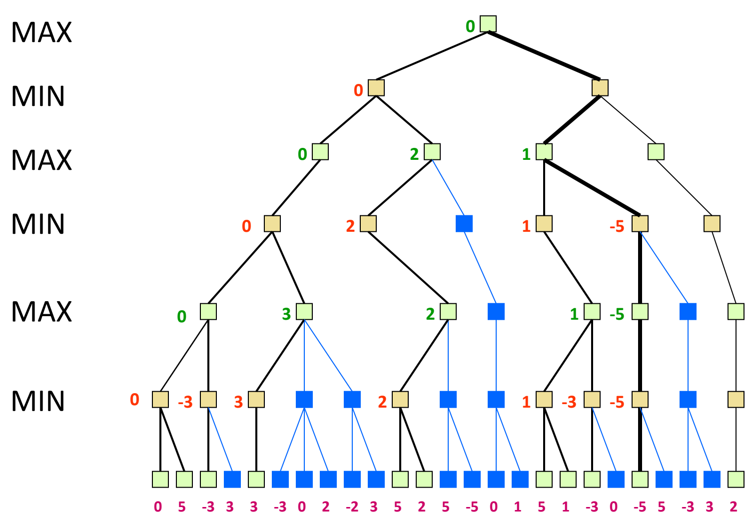
Example



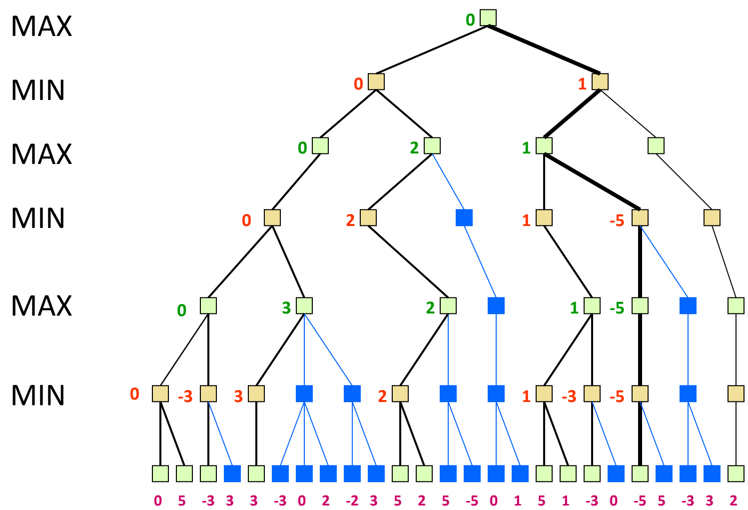
Example



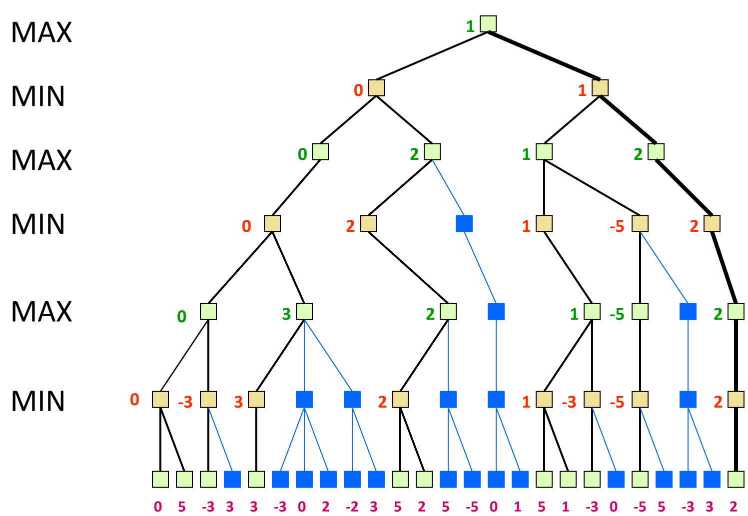
Example



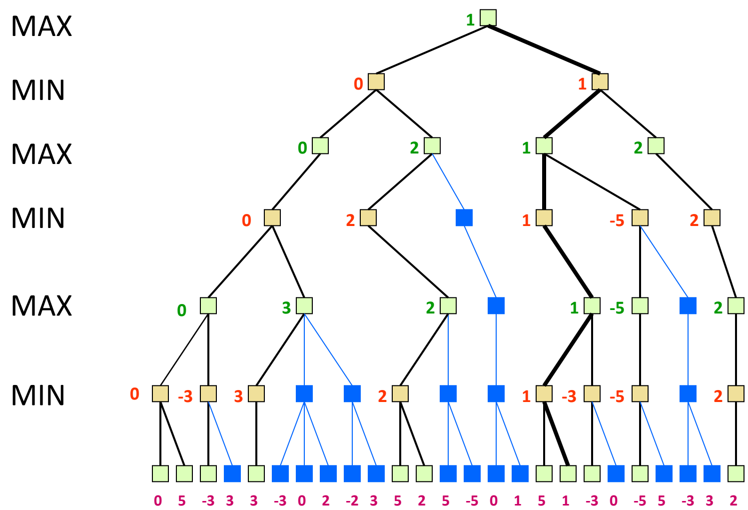
Example



Example

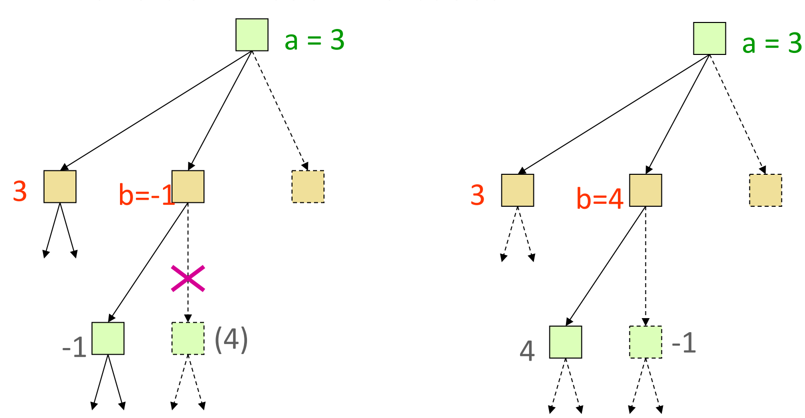


Example



How much do we gain?

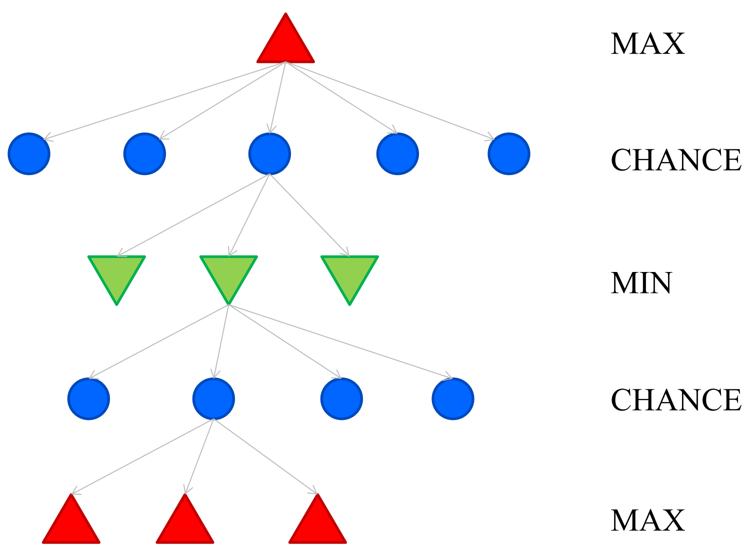
- Consider these two cases:



How much do we gain?

- Assume a game tree of uniform branching factor b
- Minimax examines $O(b^h)$ nodes = worst case for alpha-beta
- The gain for alpha-beta is maximum when:
  - The children of a MAX node are ordered in decreasing backed up values
  - The children of a MIN node are ordered in increasing backed up values
- Then alpha-beta examines $O(b^{h/2})$ nodes [Knuth and Moore, 1975]
- But this requires an oracle (if we knew how to order nodes perfectly, we would not need to search the game tree)
- If nodes are ordered at random, then the average number of nodes examined by alpha-beta is $\sim O(b^{3h/4})$

What About Probabilities?



Expectiminimax

- n random outcomes per chance node
- $O(b^m n^m)$ time

$$\text{eminimax}(n_{\max}) = \max_{s \in \text{succesors}(n)} \text{eminimax}(s)$$
$$\text{eminimax}(n_{\min}) = \min_{s \in \text{succesors}(n)} \text{eminimax}(s)$$
$$\text{eminimax}(n_{\text{chance}}) = \sum_{s \in \text{succesors}(n)} \text{eminimax}(s) p(s)$$

Expectiminimax is nasty

- High branching factor
- Randomness makes evaluation fns difficult
  - Hard to predict many steps into future
  - Values tend to smear together
  - Preserving order is not sufficient
- Pruning chance nodes is problematic
  - Need to prune based upon bound on an expectation
  - Need a priori bounds on the evaluation function

Dealing with High Branching Factors

- High branching = shallower search cutoff
- Shallower search cutoff = greater dependence on e()
- But… Coming up with good e() is hard!

- Solution: Use reinforcement learning
- Backgammon (TD-gammon)
  - Go (AlphaGo)

Source: Wikipedia

Learning e()

- Recall:
  - If e is to be trusted in the first place, then the backed-up value is a better estimate of how favorable STATE(N) is than e(STATE(N))
  - Why? Presumption that e() is more accurate closer to the end of the game
- Idea:
  - Why not propagate our experience to tune e()?
  - Reinforcement learning can do this!

Multiplayer (>2 player) Games

- Things *sort-of* generalize, but can get complicated
- Vector of possible values for each player at each node
- What's wrong with assuming all players act selfishly? (Note: In pacman, we assume all ghosts are conspiring against you, which really makes them one big agent.)
- Correct treatment requires machinery of game theory (later in the course)

Major Take Home Points

- Game tree search is a special kind of search
- Alpha-beta is a big win
- Most successful players use alpha-beta
- Good evaluation functions critical to performance in depth limited search