ADVERSARIAL SEARCH

Introduction

- One of the reasons we use search in AI is to help agents figure out what to do.
- As we mentioned before, considering how sequences of actions can be put together allows the agent to plan.
- (We will come back to this topic again in a few lectures' time).
- Using the techniques we have covered so far, we can have a single agent figure out what to do when:
  - It knows exactly how the world is;
  - Each action only has one outcome; and
  - The world only changes when the agent makes it change.

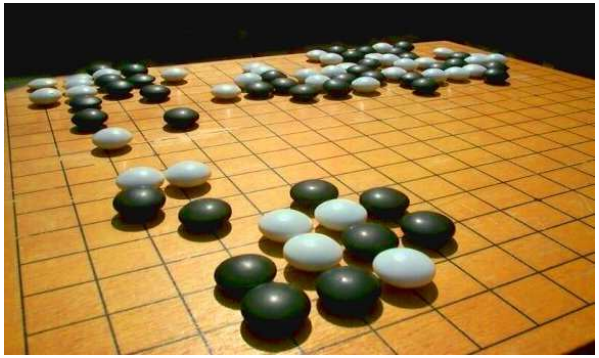
- In other words we can plan when the world is:
  - Accessible;
  - Deterministic; and
  - Static
- Obviously these are unrealistic simplifications.
- Here we will consider how to handle one kind of dynamism:
  - Other agents messing with the world.
- (In later lectures we will work towards dealing with other kinds of complication.)

Computers and Games

- The kind of search we will look at is applicable in a *two person, perfect information, zero sum* games.
- Perfect information:
  - Both players know exactly what the state of the game is.
- Zero sum:
  - What is good for one player is bad for the other.
- This is true of many games.

- Chess, Checkers, Go, . . .





- Othello, Connect 4, . . .

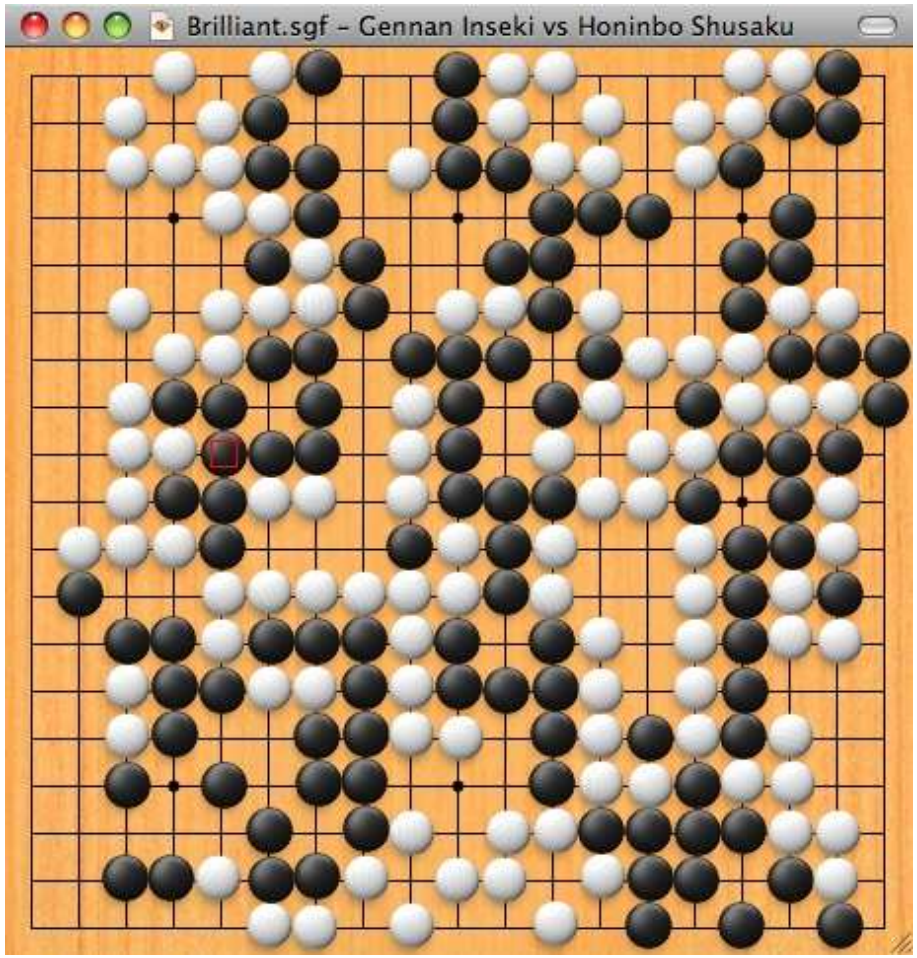

- And less traditional games

- These games are relatively easy to study at an introductory level.

- They have been studied just about as long as AI has been a subject.

- Some games are easily “solved”:
  - Tic-Tac-Toe
  - Connect 4
- Others have held out until recently.
  - Checkers
  - Chess
- Yet others are far from being mastered by computers.
  - Go
- Chance provides another complicating element.

- Why is Go so hard?



- For many of these games
  - State space
  - Iconic

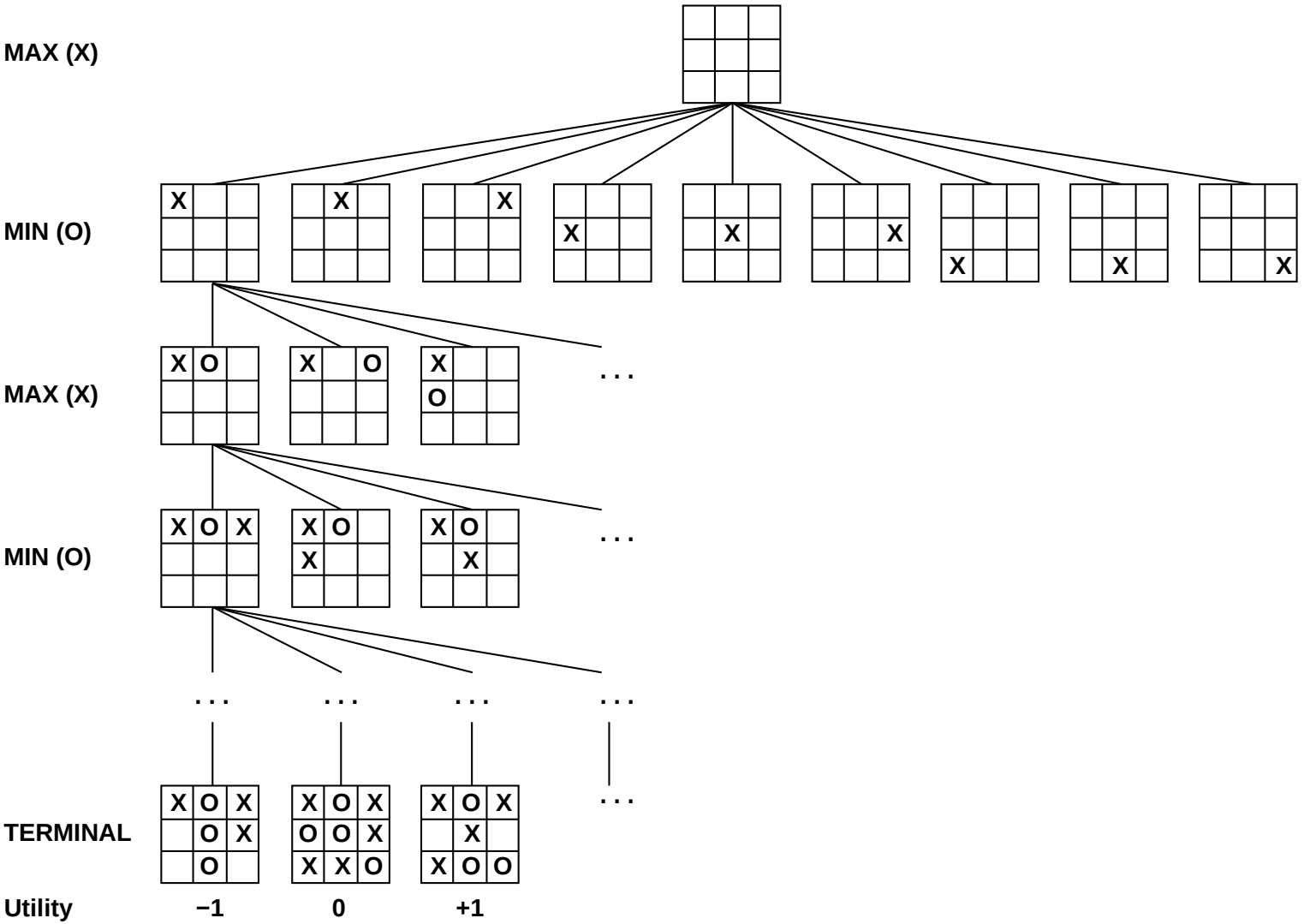
  representations seem natural.
- Moves are represented by state space operators.
- Search trees can be built much as before.
- However, we use different techniques to choose the optimal moves.

Game formulation

- We can think of a game as a search problem with the following elements:
  - The initial state, S_0.
  - PLAYER(s), which player can move in state s.
  - ACTION(s), the legal set of actions a in state s.
  - The transition model RESULT(s, a) which gives the outcome of each action in each state.
  - TERMINAL-TEST(s), which says if the game is over.
  - UTILITY(s, p), a *utility* function which says what the benefit is to player p of the game ending in state s.
- The utility will vary from game to game. In chess, the utility to a player is 1, 0, or $\frac{1}{2}$.

- A *zero sum* game is one in which the total gains of both players sum to zero.
- The usual way that chess is scored makes it a *constant sum* game — the scores of the two players add up to 1.
- It is easy to transform a constant sum game into a zero-sum game by a simple (affine) transformation of the utility function.
- We therefore often use the terms interchangeably.

- As for a search problem, the initial state, action set and transition model define a *game tree* for the game.
- Nodes are states of the game and edges are moves.
- We draw the tree assuming two players, MAX and MIN (for reasons that will become obvious) where MAX moves first.
- The next slide gives the tree for tic-tac-toe (or what I would call "noughts and crosses").


MAX (X)
MIN (O)
MAX (X)
MIN (O)
TERMINAL
Utility
−1
0
+1

- Since MAX moves first, even numbered layers are the ones where MAX gets to choose what move to make.
- The first node is on the zeroth layer.
- We call these "MAX nodes".
- "MIN nodes" are defined similarly.
- A *ply* of *ply-depth* k are the nodes at depth $2k$ and $2k + 1$.
- We usually estimate, in ply, the depth of the "lookahead" performed by both agents.

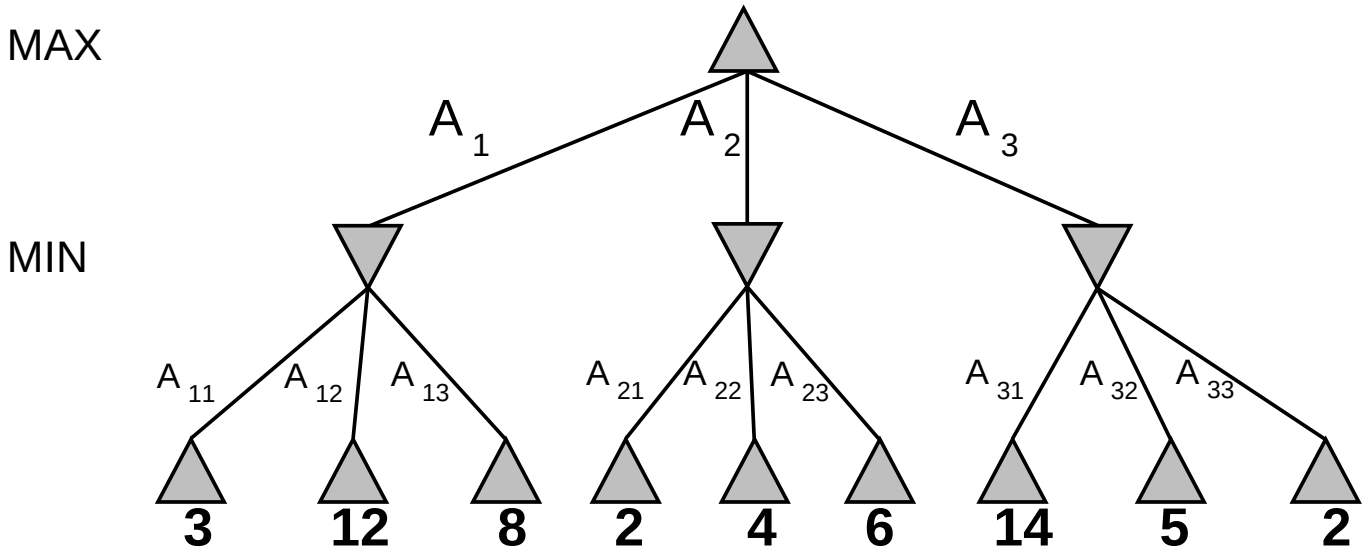
- In general we can't search the whole tree.
- Tic-tac-toe might have less than $9!$ ($362,880$) terminal nodes but many games are bigger.
  - Chess: 10^{40} nodes
  - $\approx 10^{20}$ centuries to build game tree.
- Distinguish between the game tree and the *search tree* which is what we actually construct.
- Typically we just search to a limited horizon (like depth-bounded).
- Then evaluate (using some heuristic) the leaf nodes of the seach tree.
- Then extract the best move at the top level.

- Key thing is that we have to take into account what the other player is doing.
- Rather than the simple path that is a solution in a search problem, we need a *contingent strategy*.
- We need to look at MAX's first move, then every move MAX might take in response to what MIN does, and so on.
- This gives us an *optimal* strategy in the sense that we do as well as we possibly can (as well as any other strategy) against an infallible opponent.

  (There may be better strategies against weaker opponents).

Minimax search

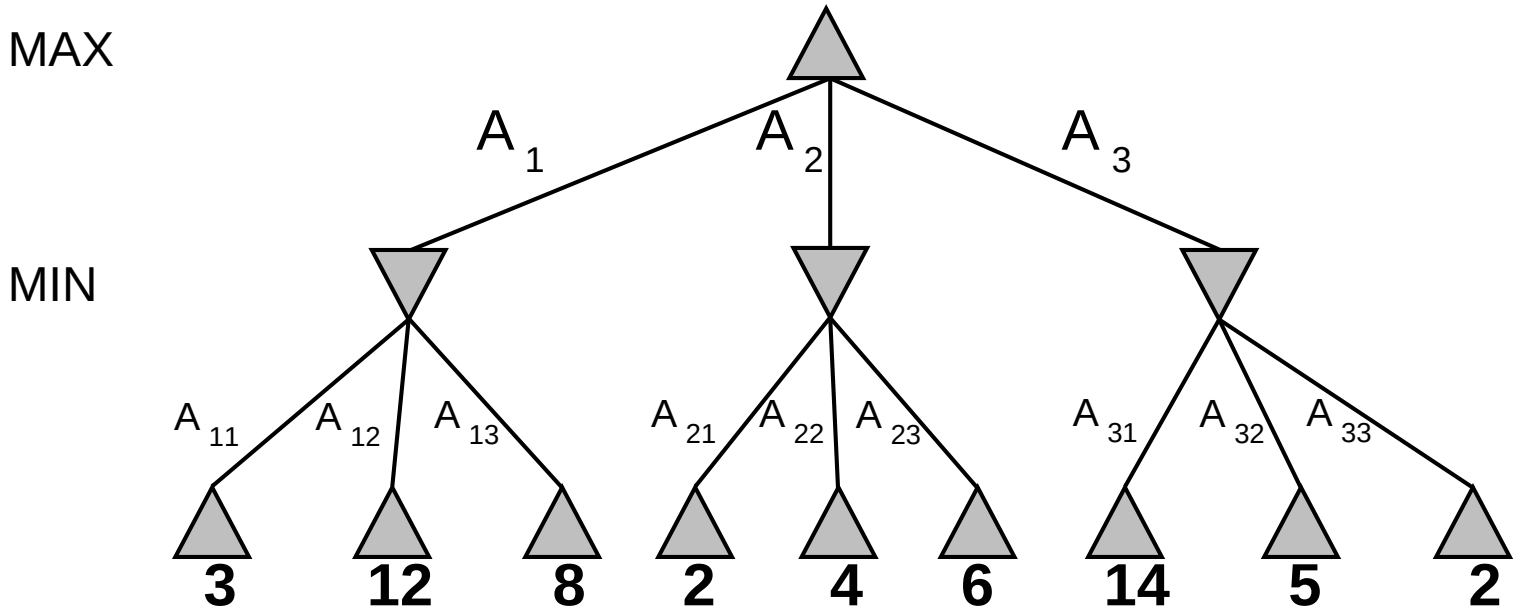
- One-ply (two move) search tree:

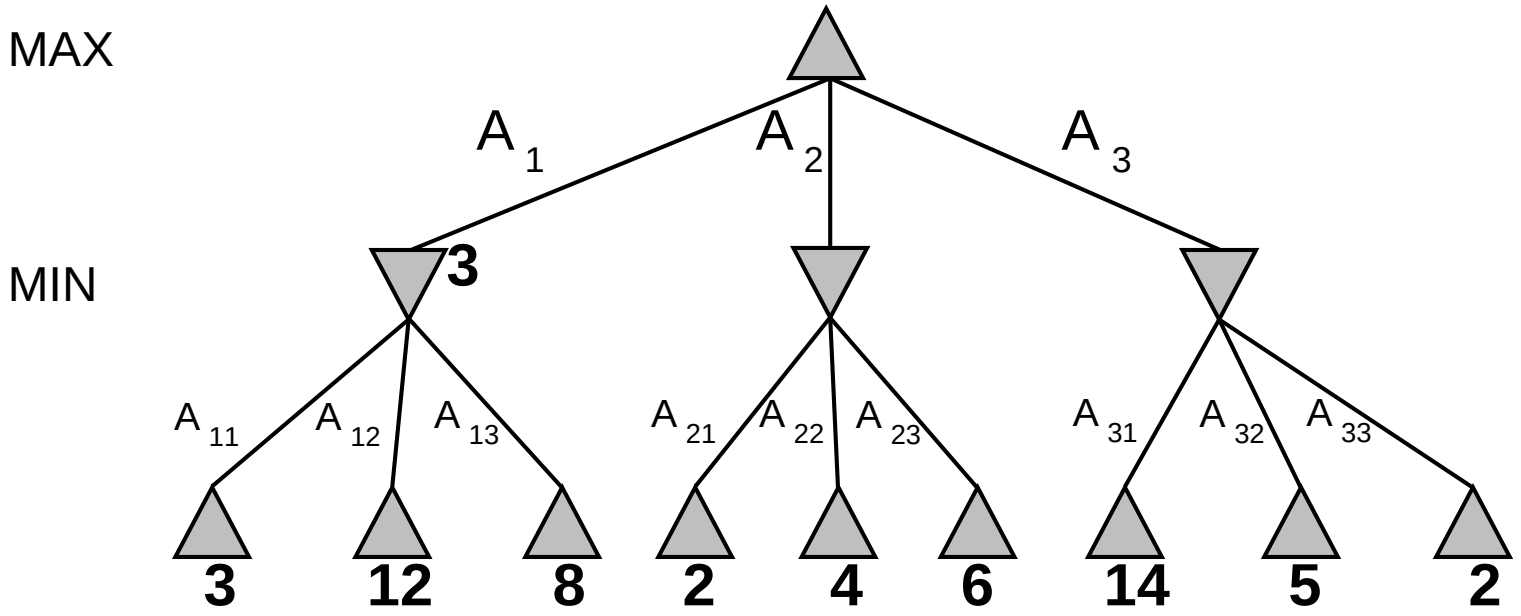


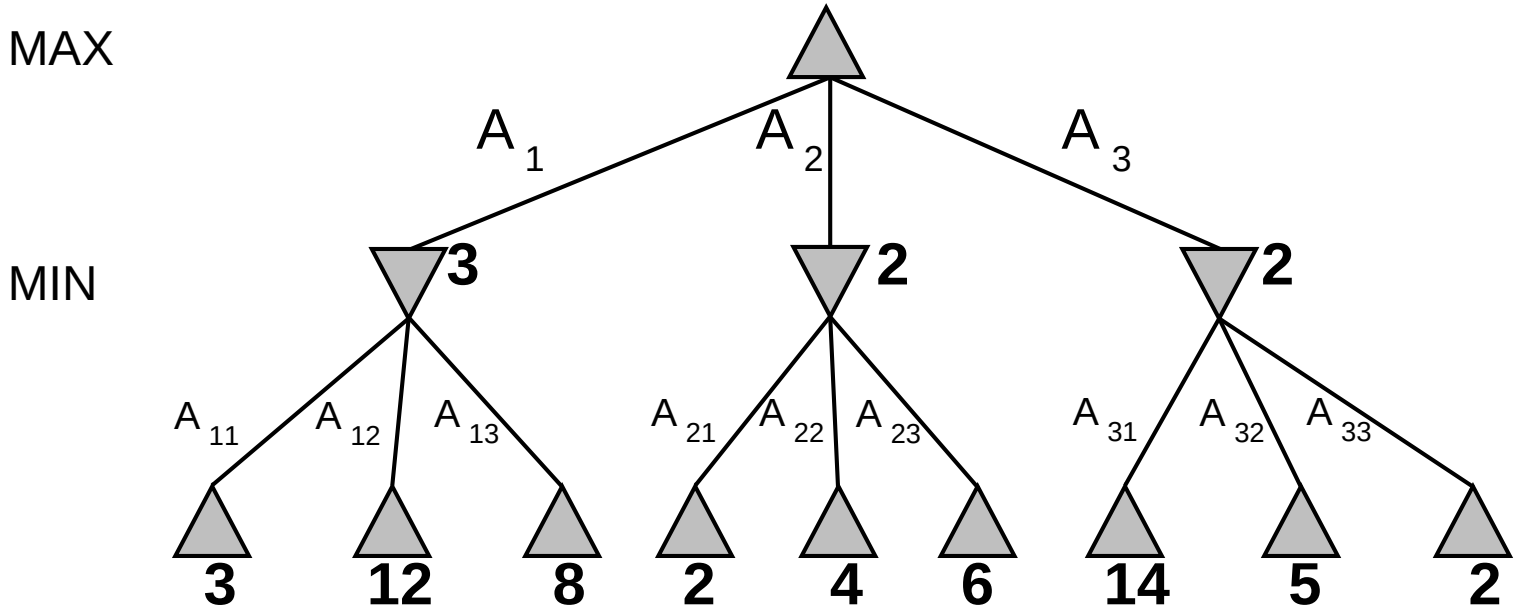
- We determine the optimal strategy by establishing the *minimax* value of each node, which is the value to MAX of being in the state corresponding to s.
- Well, the value assuming that both players finish the game out perfectly.
- How do we do this?

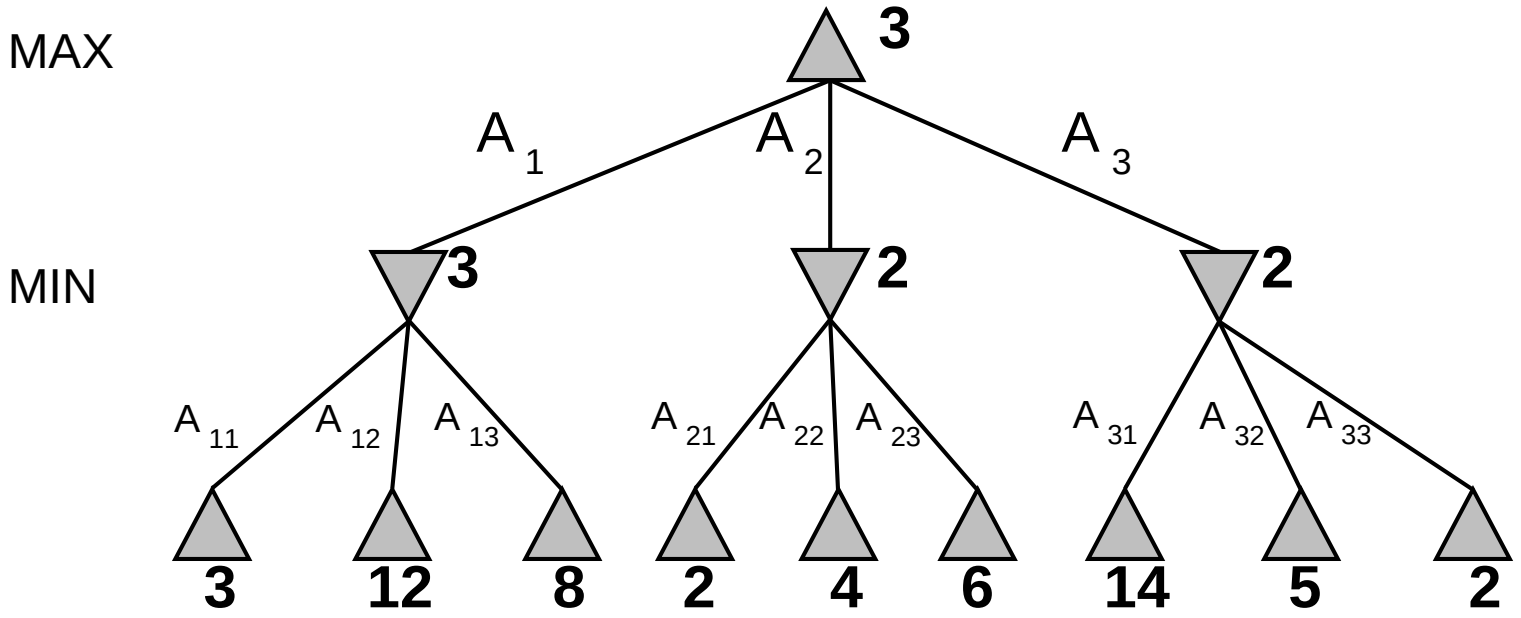
- Assume our utility function gives terminal nodes high positive values if they are good for MAX
- And low values if they are good for MIN
- Now, look at the leaf nodes and consider which ones MAX wants:
  - Ones with high values.
- MAX could choose these nodes *if* it was his turn to play.
- So, the value of the MAX-node parent of a set of nodes is the max of all the child values.

- Similarly, when MIN plays she wants the node with the lowest value.
- So the MIN-node parent of a set of nodes gets the min of all their values.
- We back up values until we get to the children of the start node, and MAX can use this to decide which node to choose.


MAX
MIN
A 1
A 2
A 3
A 11
A 12
A 13
A 21
A 22
A 23
A 31
A 32
A 33
3
12
8
2
4
6
14
5
2


MAX
MIN
A 1
A 2
A 3
3
A 11
A 12
A 13
A 21
A 22
A 23
A 31
A 32
A 33
3
12
8
2
4
6
14
5
2


MAX
MIN
A 1
A 2
A 3
3
2
2
A 11
A 12
A 13
A 21
A 22
A 23
A 31
A 32
A 33
3
12
8
2
4
6
14
5
2


MAX
MIN
3
A 1
A 2
A 3
3
2
2
A 11
A 12
A 13
A 21
A 22
A 23
A 31
A 32
A 33
3
12
8
2
4
6
14
5
2

- There's an algorithm for this.

function MINIMAX-DECISION(*state*) **returns** *an action*
  inputs: *state*, current state in game

  return the *a* in ACTIONS(*state*) maximizing MIN-VALUE(RESULT(*a*, *state*))

---

function MIN-VALUE(*state*) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow \infty$
  for *a*, *s* in SUCCESSORS(*state*) **do** $v \leftarrow$ MIN(v, MAX-VALUE(*s*))
  return *v*

```
function MAX-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← −∞
    for a, s in SUCCESSORS(state) do v ← MAX(v, MIN-VALUE(s))
    return v
```

- How about problems when we can't build the search tree down to the leaf nodes?
- We choose a number of ply to search and have a utility function for incomplete states.
- Then we use the minimax algorithm as before.
- There is an assumption which is that the utility function works as a better guide on nodes down the tree than on the direct successors of the start node.
- This should be the case (modulo horizon effects).
- Let's look at a concrete example—Tic-tac-toe.

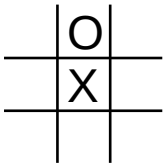
- Let MAX play crosses and go first.
- Breadth-first search to depth 2.
- Utility function $e(p)$:

$$e(p) = \begin{cases} \infty & \text{if } p \text{ is a win for MAX} \\ -\infty & \text{if } p \text{ is a win for MIN} \\ val & \text{otherwise} \end{cases}$$

where

$$\begin{aligned} val \;=\; & (\text{possible winning rows columns diagonals for MAX}) \\ & -(\text{possible winning rows columns diagonals for MIN}) \end{aligned}$$

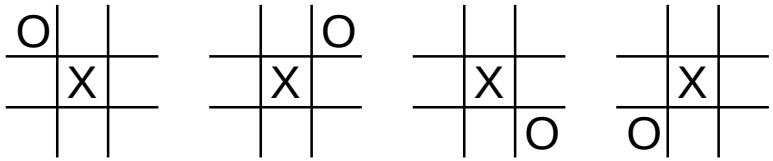
- So,



- Scores:

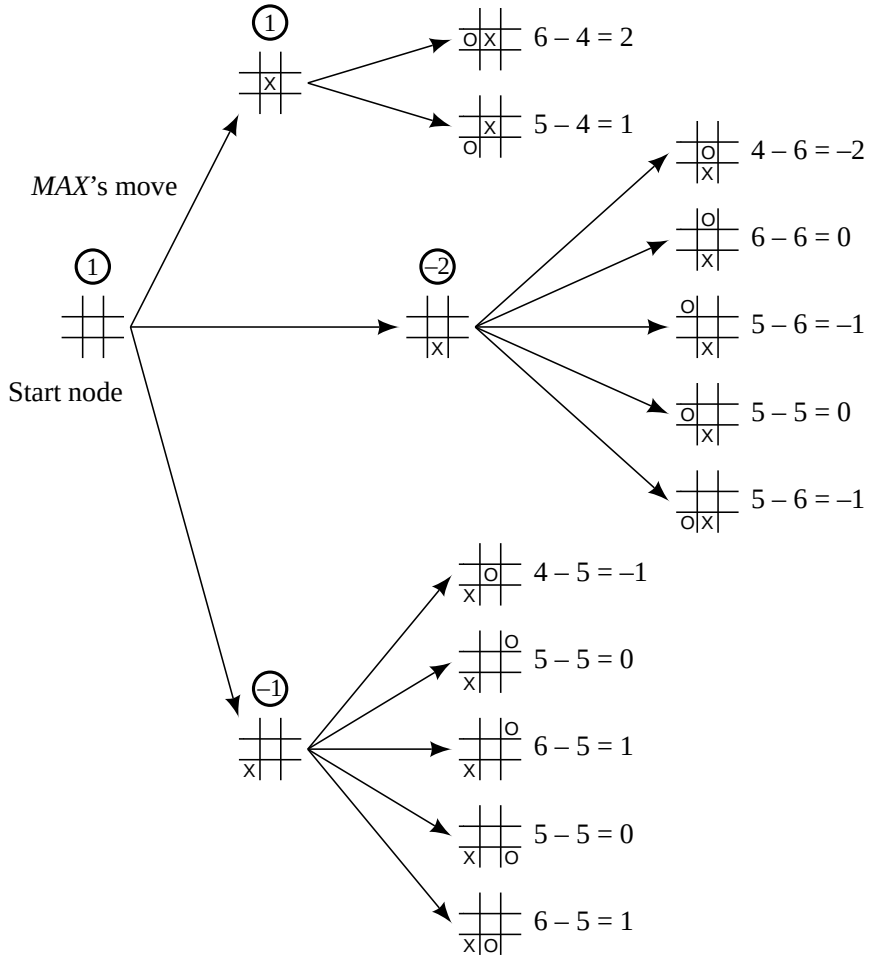
$$6 - 4 = 2$$

- We also use symmetry to avoid having to generate loads of successor states, so

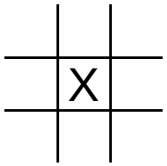


- are all equivalent.

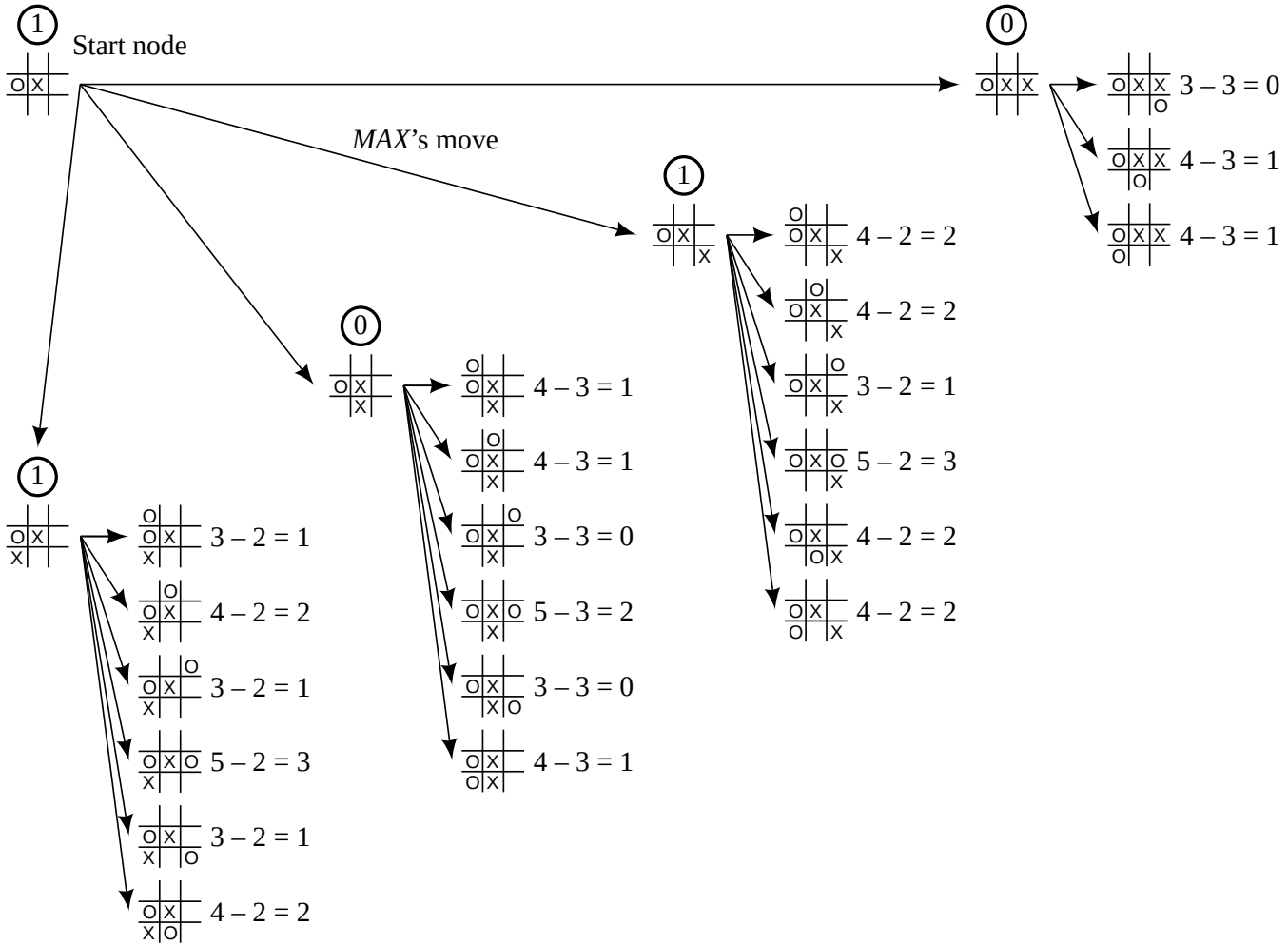
- We run the depth 2 search, evaluate, and back up values



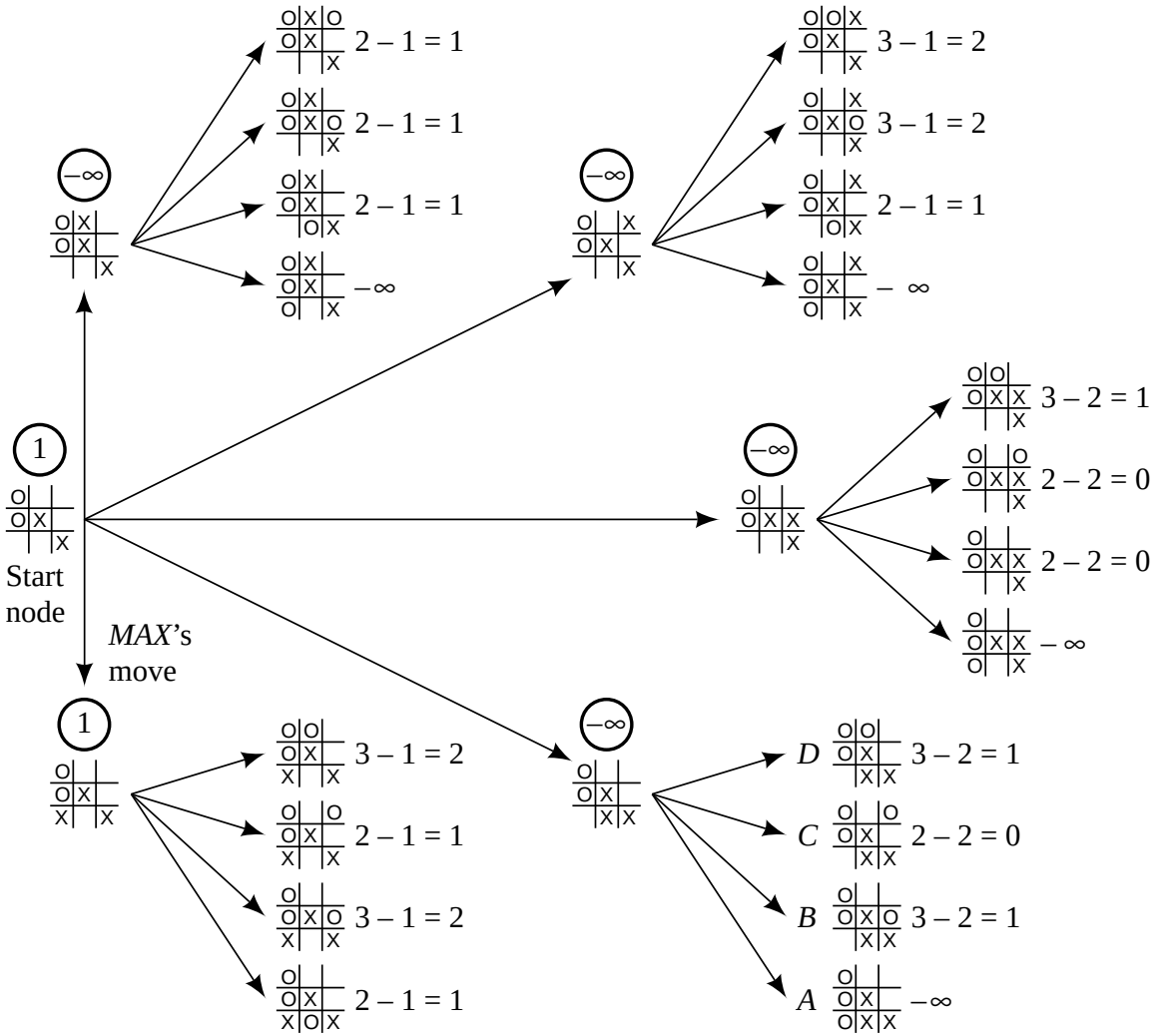
- Unsurprisingly (for anyone who ever played Tic-tac-toe):



- Is the best move.
- So MAX moves and then MIN replies, and then MAX searches again:



- Here there are two equally good best moves.
- So we can break the tie randomly.
- Then we let MIN move and do the search again.

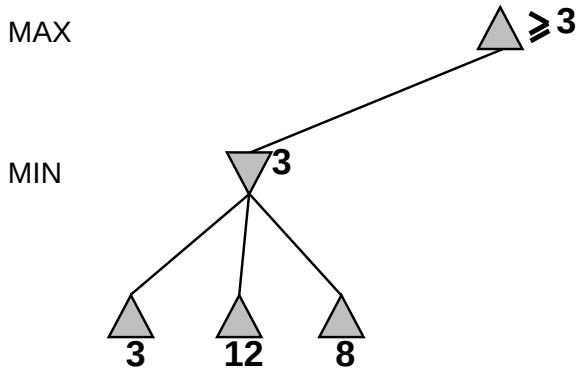


- And so on.

Alpha-Beta search

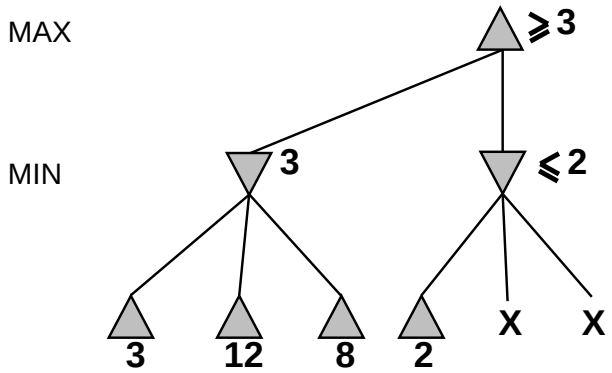
- Minimax works very neatly, but it is inefficient.
  - Exponential in the number of moves.
- The inefficiency comes from the fact that we:
  - Build the tree,
  - THEN back up the values
- If we combine the two we get massive savings in computation.
- How do we manage this?
- Let's think through the construction of the search tree.

- After we have built the first branch, we know that the least valuable move will be worth 3 to MAX.



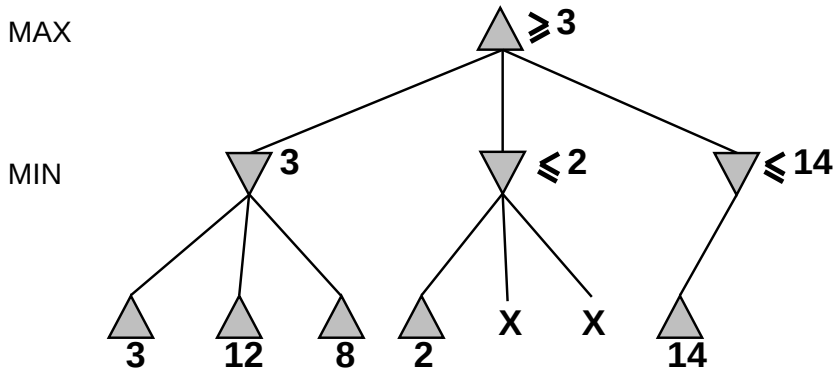
- Put another way, since we have a MIN node worth 3, the worst that MAX can do is to have a move worth 3.

- Now as soon as we have the first leg of the second branch, we know that we don't need to expand the branch any more.



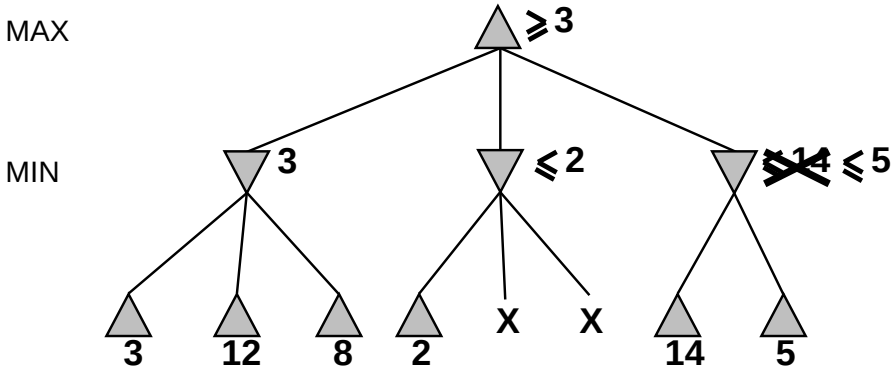
- It will have at most a value of 2 and MAX won't pick it.

- Things aren't so easy with the third branch.

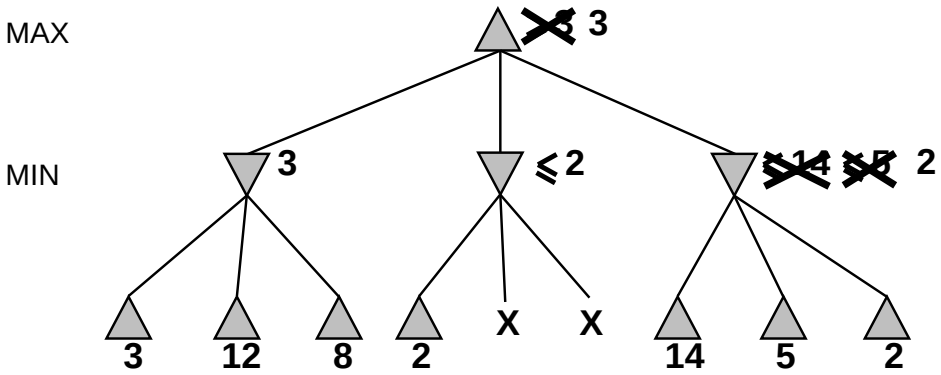


- After the first leg, the branch could have a very high value.

- But this value soon falls.

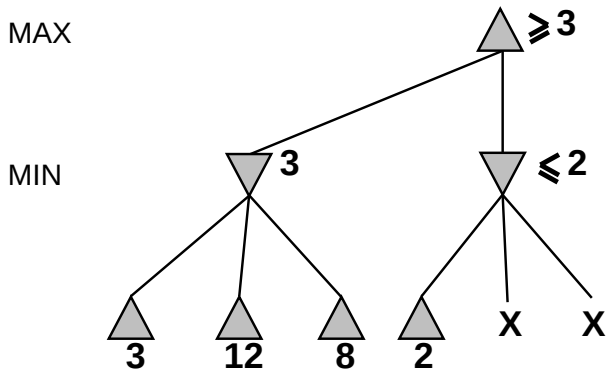


- And then falls again when we explore the final leaf node.



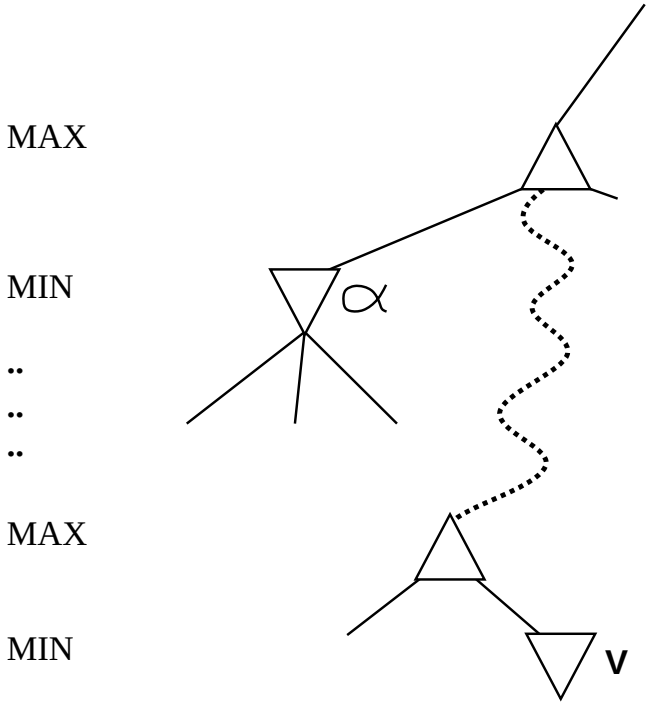
Now, how can we use this?

- Well, the reason we could avoid an search here is that we got to a MIN node that was going to have a smaller value than a MIN node we had already got a value for.



- Anytime that happens we can stop searching down that branch.

- We give the name α to that high value at the MIN node.



- If $\alpha > v$ we will never get to the node labelled with v.

- We can also do the dual of this.
- That is we can prune below a MIN node, when we find child MAX node that has a higher value than an explored MIN node.
- We give the name β to the low value at the MAX node.

function ALPHA-BETA-DECISION(*state*) **returns** an action

$v \leftarrow$ MAX-VALUE($state, \infty, -\infty$)
return the a in ACTIONS(*state*) with value v

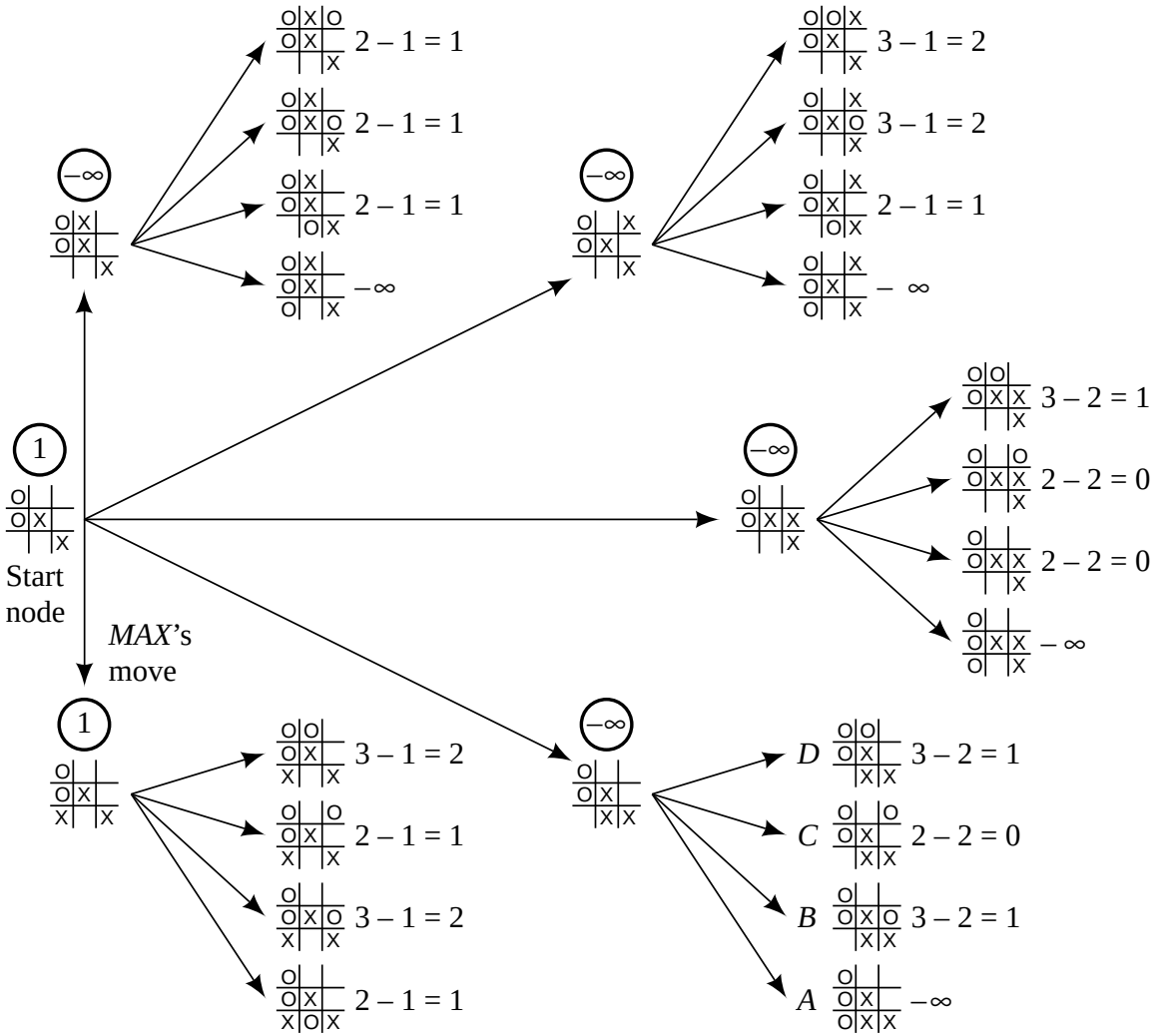
---

function MAX-VALUE(*state*, α, β) **returns** *a utility value*

if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
$v \leftarrow -\infty$
for a in ACTIONS(*state*) **do**
$v \leftarrow$ MAX(v, MIN-VALUE(RESULT(s, a), α, β))
if $v \geq \beta$ **then return** v
$\alpha \leftarrow$ MAX(α, v)
return v

```
function MIN-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← ∞
  for a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s, a), α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v
```

- Let's look at this in the context of Tic-tac-toe.



- Well, when we get to node A, we don't have to expand any further.
- So we save the evaluation of B, C and D.
- We also don't have to search any of the nodes below these nodes.
- This does nothing to stop MAX finding the best move.
- It also works when we don't have a winning move for MIN.

- Using α-β pruning always gives the same best move as full minimax.
- However, often the approach involves less searching.
- For example, for chess, if we can evaluate 1 million moves a second, then we can search around 5 ply.

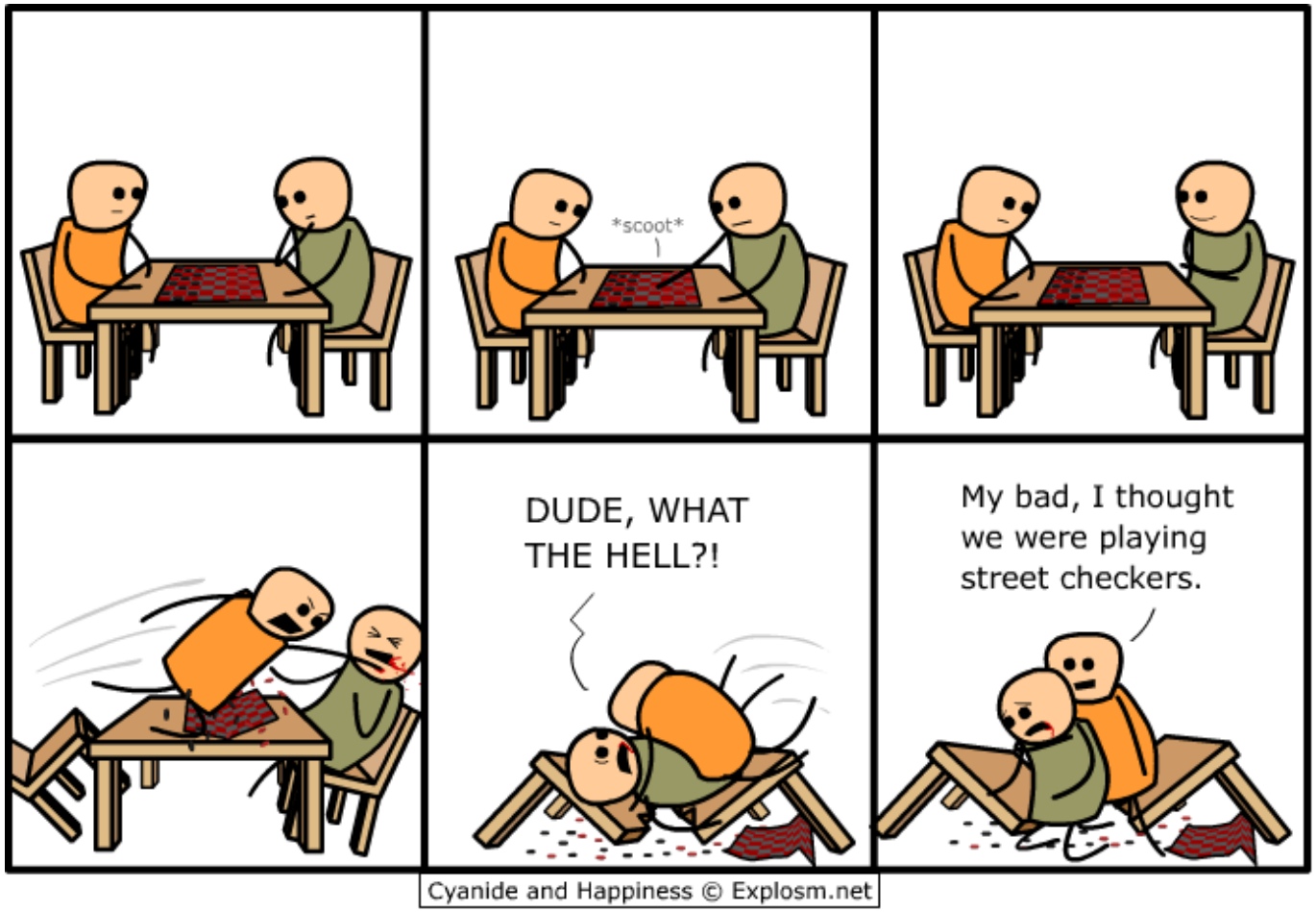
  (Assumes using aorund 3 minutes a move, which is about right for tournament play)

  – Easily beaten by an average human player.
- With α-β pruning we can search about 10 ply.

  – Expert level play.

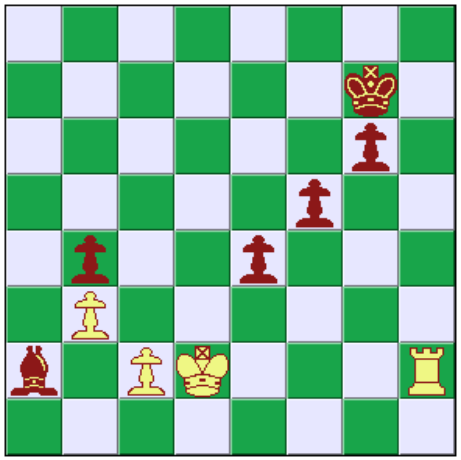
Forward pruning

- The neat thing about α-β pruning is this fact that it always gives the same move as a full search.
- However, game trees can still get too big.
- When this happens we can employ *forward pruning*.
  - Just don't build the whole tree.
- One approach is *beam search*, we just expand the *n* best moves at each level.
  - A mistake in evaluation can be fatal.
- Better is PROBCUT which keeps statistics from previous games and prunes nodes which are *likely* to be outside the current α-β window.


scoot
DUDE, WHAT THE HELL?!
My bad, I thought we were playing street checkers.
Cyanide and Happiness © Explosm.net

Horizon effects

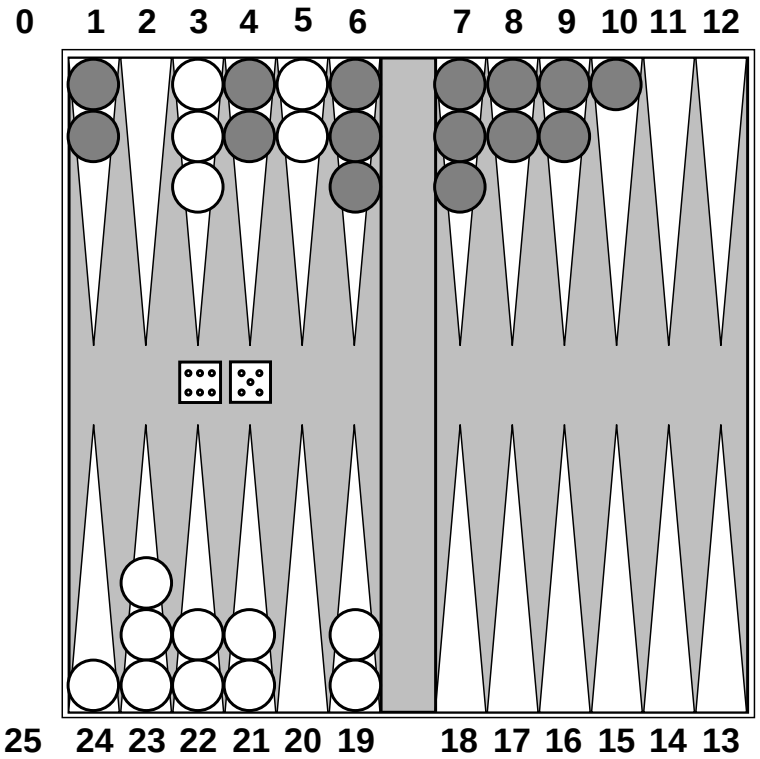
- When we search a fixed number of ply, we can easily get misled by the evaluation function.
- There might be a very bad position just over the *horizon*.



- Here, Red is going to lose its bishop, but putting white into check appears to avoid the problem.

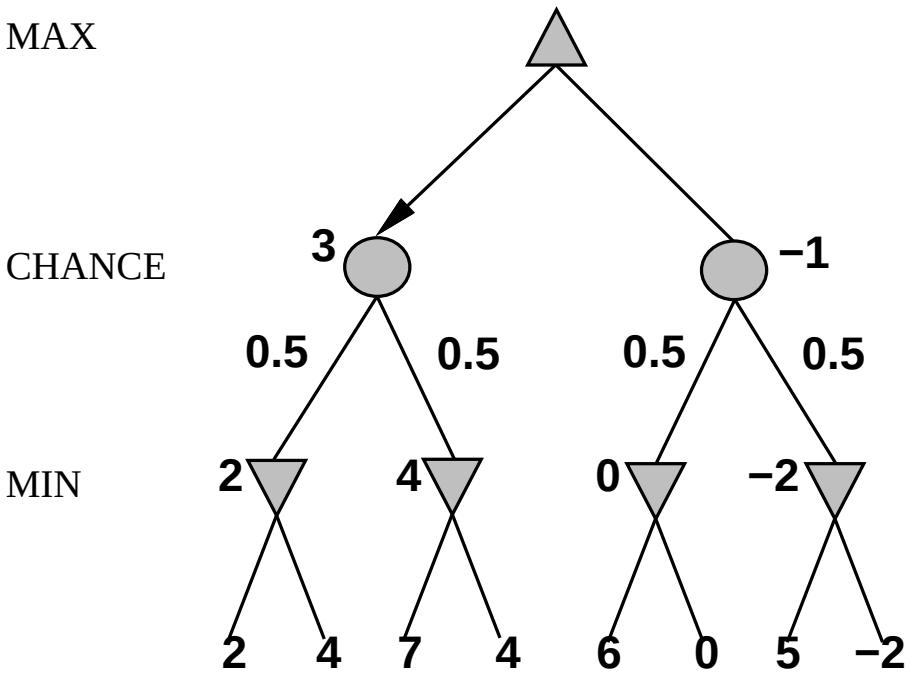
- Equally, if we don't search far enough, we might not see the bad consequences of an action.
- Stop at *quiescent* nodes (value is the same as it would be of you looked ahead a couple of moves).
- Can be exploited by opponents; pushing moves back behind the horizon.
  - Kasparov tried to work this trick against Deep Blue in their first match.
- A similar problem occurs because we assume that players always make their best move:
  - "Bad" moves can mislead a minimax-style player.

Stochastic games



- White is choosing to move without knowing what Black's potential next moves will be.

- We handle this by expanding the game tree.
- Add an additional player, CHANCE who controls how we get from MAX nodes to MIN nodes
- CHANCE tells us how probable it is to make a given transition.
- We use this probability to average over the children if each chance node when we back values up.
  - We compute the *expected value* of the moves.
- We then do the usual minimax thing.



- The value of the leftmost CHANCE node is:

$$(0.5 \times 2) + (0.5 \times 4)$$

- Here's the heart of an algorithm to do this kind of evaluation:

  if *state* is a MAX node **then**
  return the highest EXPECTIMINIMAX-VALUE of SUCCESSORS(*state*)
  if *state* is a MIN node **then**
  return the lowest EXPECTIMINIMAX-VALUE of SUCCESSORS(*state*)
  if *state* is a chance node **then**
  return average of EXPECTIMINIMAX-VALUE of SUCCESSORS(*state*)
  . . .

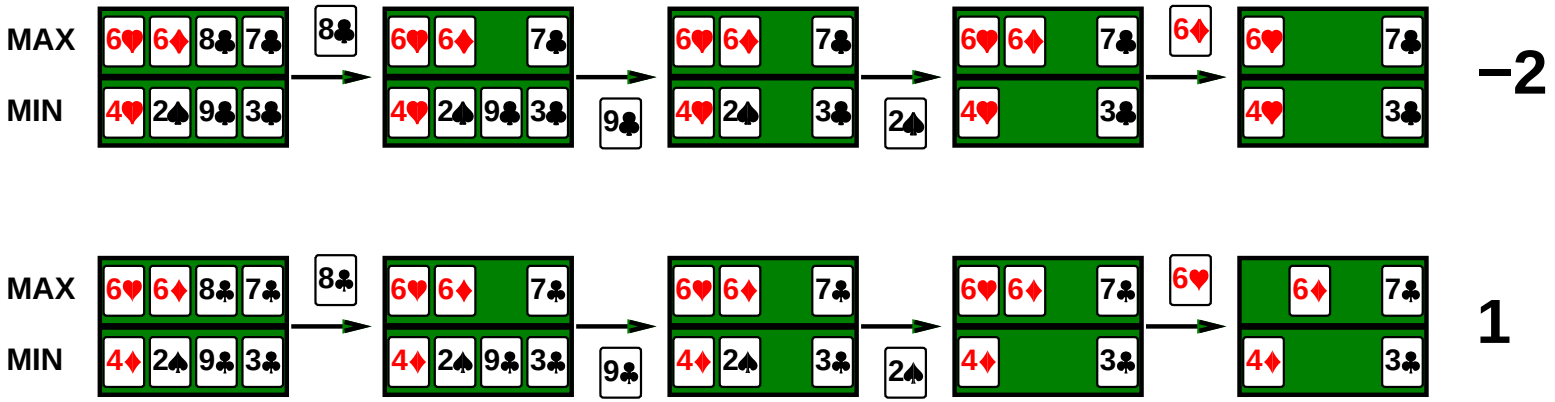
Partial observability

- In many games you don't have full information about the state of the game.

- In card games the uncertainty is in what the allocation of cards is.
- A bit analagous to a big dice throw at the start of the game.
- Intuition:
  - Consider all possible allocations.
  - Play out each one and evaluate it.
  - Average over the possible values of the moves
- If s is a deal, we pick the a:

$$\text{argmax}_a \sum_s \Pr(s)\textsc{Minimax}(\textsc{Result}(s, a))$$

- Four-card hearts hand, MAX to play first

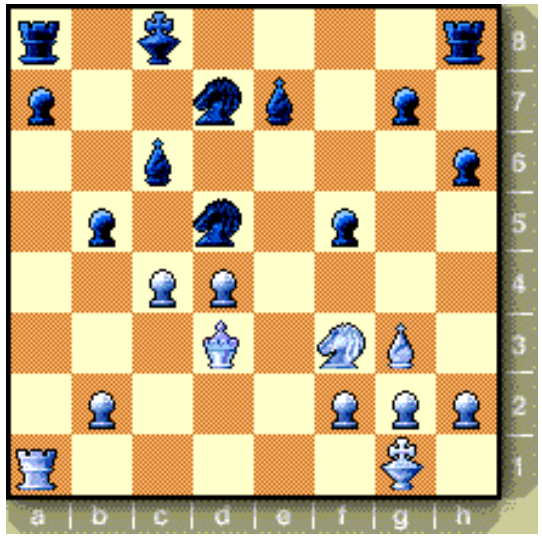


- Repeat for all possible combinations and average over the outcomes to assign values to states.
- Then do minimax.

State of the art

- Checkers:
  - Chinook ended 40-year-reign of human world champion Marion Tinsley in 1994.
  - Used an endgame database defining perfect play for all positions involving 8 or fewer pieces on the board, a total of 443,748,401,247 positions.
- Othello:
  - A computer program defeated the human world champion in 1997.
  - Generally acknowledged that humans can beat computers at Othello.

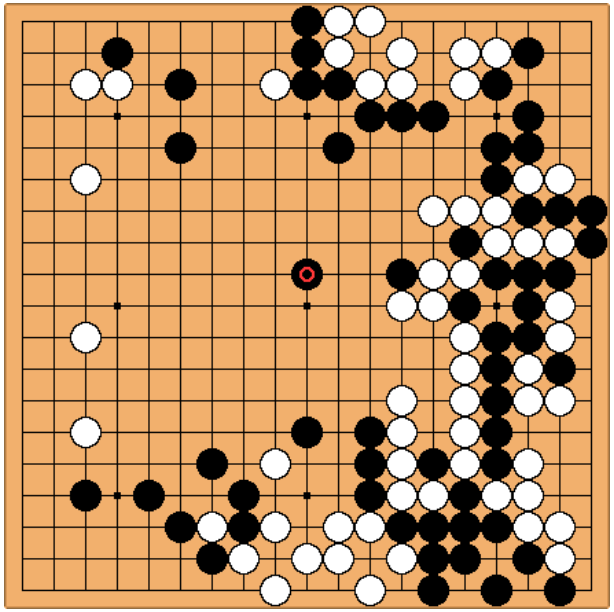
- Chess:
  - Deep Blue defeated human world champion Gary Kasparov in a six-game match in 1997.



  - Kasparov had defeated the program the previous year.

- Chess:
  - Deep Blue searched 200 million positions per second, and used very sophisticated evaluation, allowing search to regularly reach 14 ply.
  - Also followed some interesting lines of play up to 40 ply.
  - Had an extensive database of opening and closing moves.
  - Deep Blue used specialised hardware, but increases in processor speed mean equivalent performance can now be obtained using off-the-shelf hardware.
  - Recent games suggest that computers have pulled ahead of all human players.

- Go:
  - $b > 300$ and evaluation functions are hard to write.



  - However, simulation-based techniques have started to have some success.
  - On a reduced board (9×9) programs play at the master level.
  - But still play like amateurs on a full board.

Summary

- We have looked at game playing as adversarial state-space search.
- Minimax search is the basic technique for finding the best move.
- α/β search gives greater efficiency, especially when coupled with forward pruning.
- Games of chance can be handled by adding in the random player CHANCE.
- Partial observability adds another level of complexity.