Imperial College London

Adversarial Search

(Game Search)

Murray Shanahan

Notes based on Ch.6 of Russell & Norvig

Overview

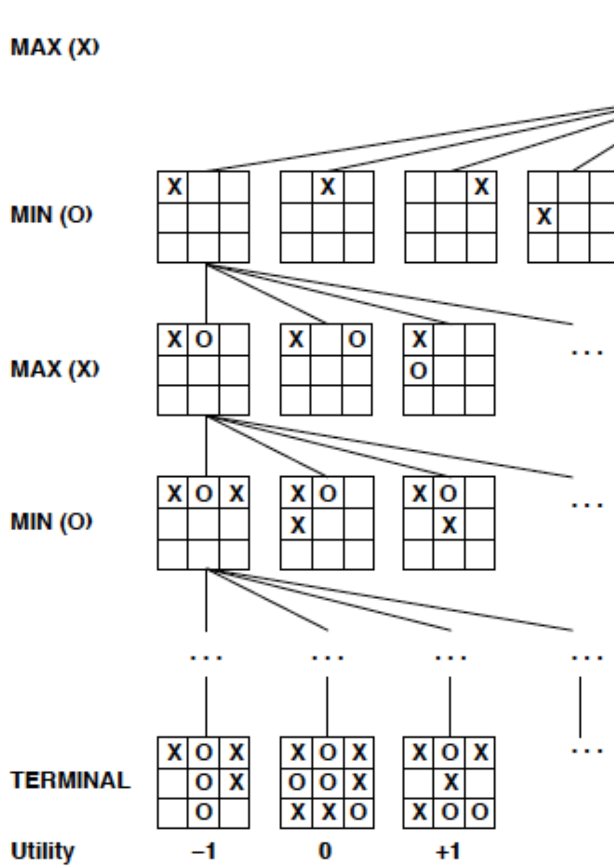
- Types of games
- Mimimax
- α-β pruning

Types of Games

- Games search involves an unpredictable opponent
- Games can be classified along several dimensions
- We'll be looking at two-player deterministic games where there is perfect information, such as chess

	Deterministic	Chance
Perfect information	chess, drafts, go, othello	backgammon, monopoly
Imperfect information	battleships	bridge, poker, scrabble

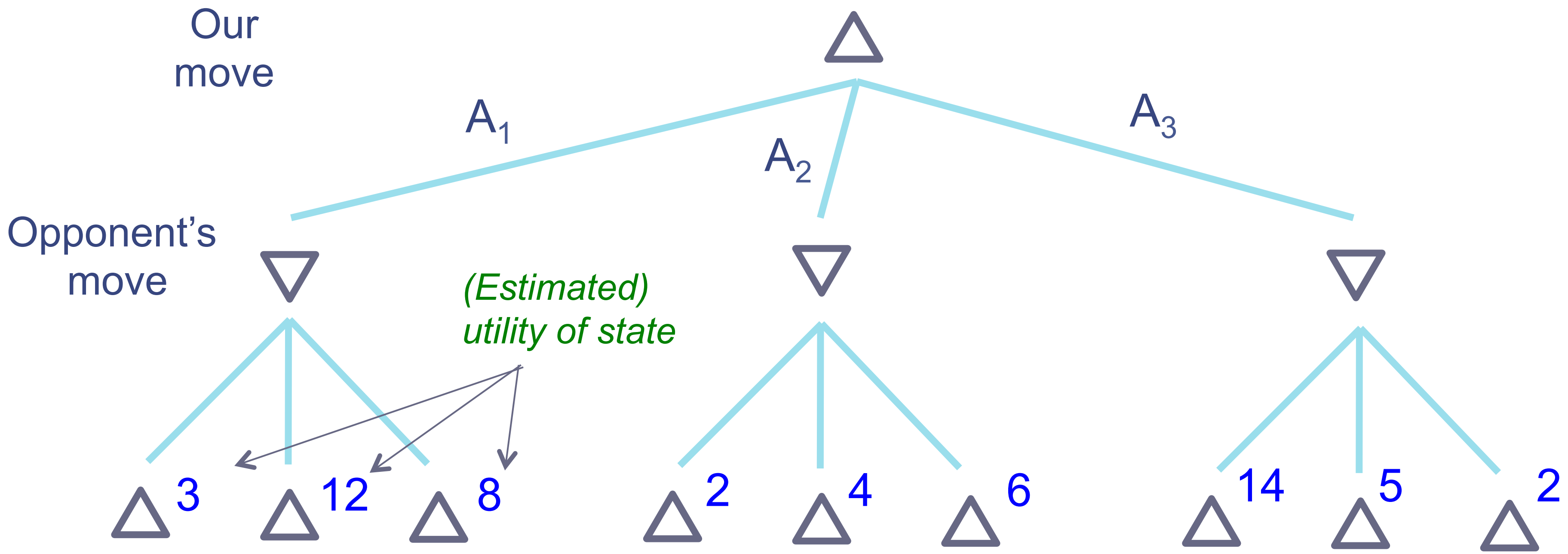
Game Trees



- The task is to search a game tree like this for the best move
- The search can go all the way to the game's terminal states
- Or it can finish after a given depth (an *n*-ply search is a search to depth *n*)
- An evaluation function is needed to estimate the utility of any given state

Choosing the Best Move

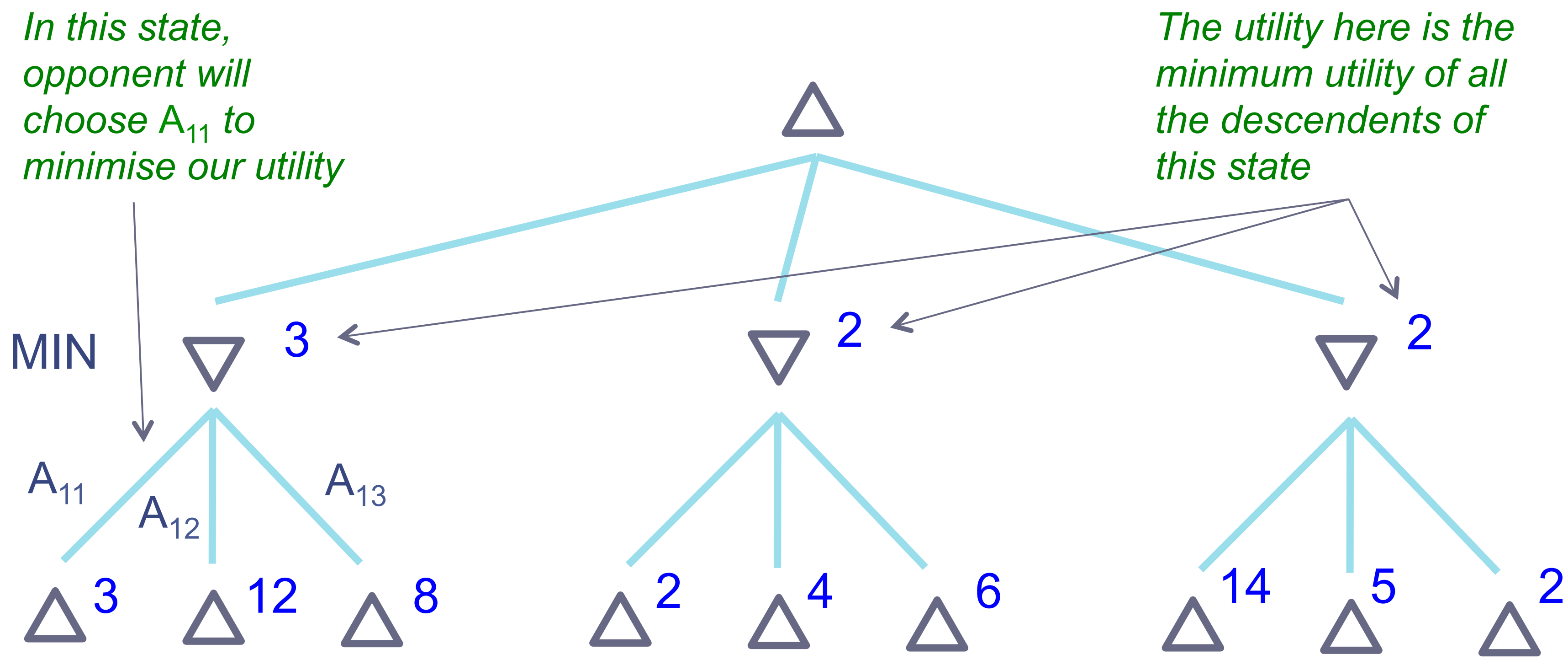
- Suppose we have the following search tree
- Should we select action A_1, A_2, or A_3?



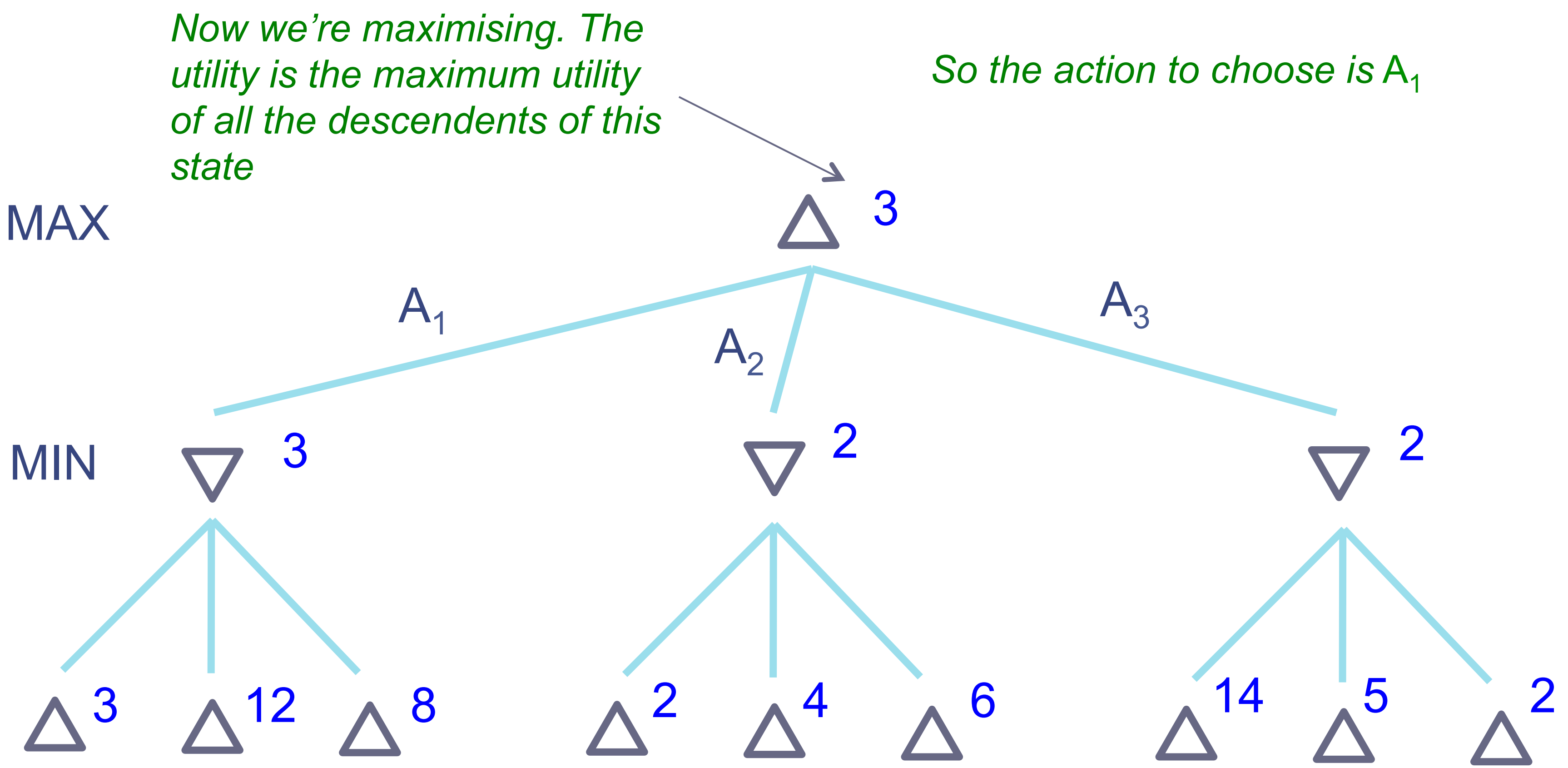
The Minimax Principle

- Assume that the opponent will always make the worst move for us
  - This is the action with the lowest estimated utility
- But we will always make the best move
- So utilities can be propagated up the tree, alternating between minimizing utility (opponent’s move) and maximizing utility (our move)
- The move we make is the one with the maximum utility at the root

The Minimising Phase



The Maximising Phase



The Minimax Algorithm 1

- The algorithm is expressed as a pair of mutually recursive functions `MinValue` and `MaxValue`
- `Eval(s)` is the evaluation function. It yields an estimate of the utility of state `s`

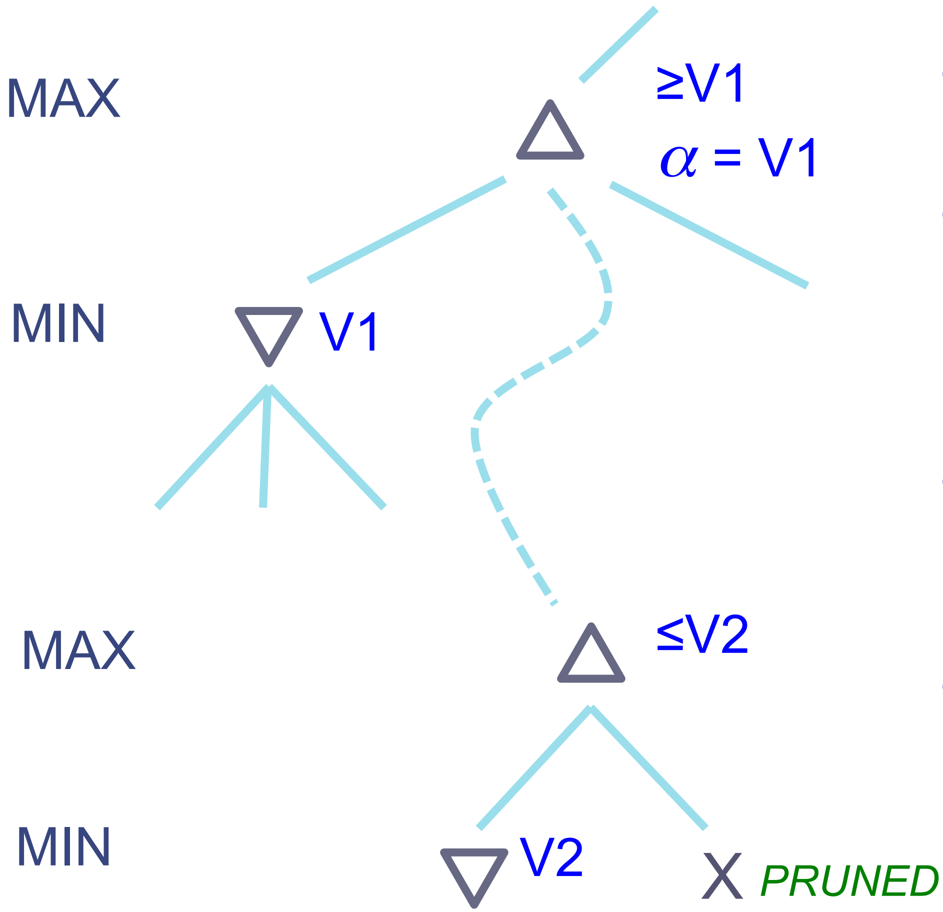
```
function MinValue(s,d)
      if s is a terminal state or d = MaxDepth
            return Eval(s)
      else
            v := ∞
            for each action a possible in s
                  v := Min(v,MaxValue(Result(a,s),d+1)
            return v
```

The Minimax Algorithm 2

- The best move is the action `a` that maximises `MinValue(Result(a,S0),1)` where `S0` is the current state of the game

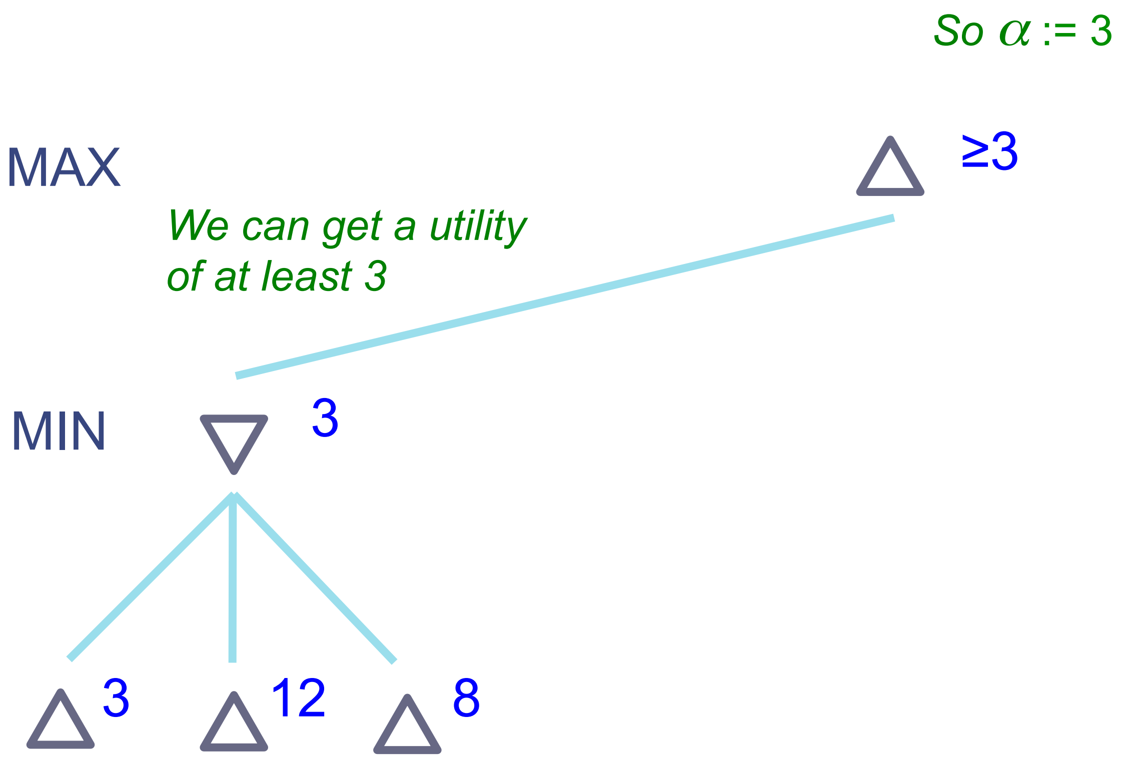
```
function MaxValue(s,d)
      if s is a terminal state or d = MaxDepth
            return Eval(s)
      else
            v := –∞
            for each action a possible in s
                  v := Max(v,MinValue(Result(a,s),d+1)
            return v
```

α-β Pruning

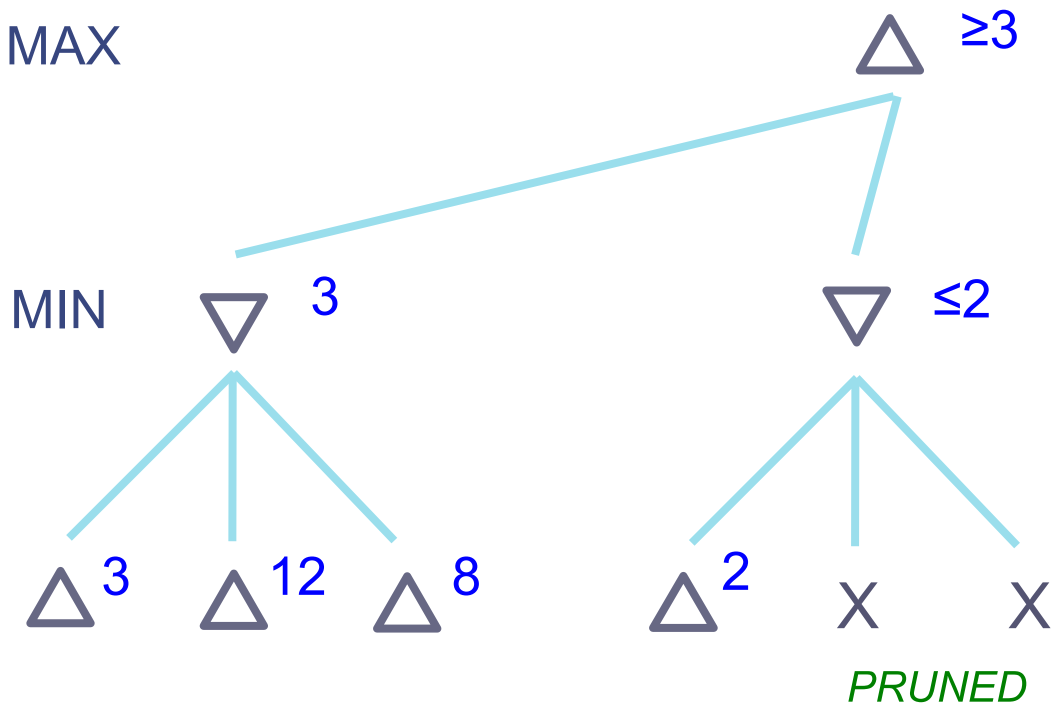


- Minimax performs a lot of redundant search
- It can be improved by keeping track of the best MAX value found so far (α) and the worst MIN value (β)
- There is no point in exploring MAX branches worse than α or MIN branches better than β
- Here, if we have V2 < α there is no need for MAX to explore more branches for that node

α-β Pruning Example

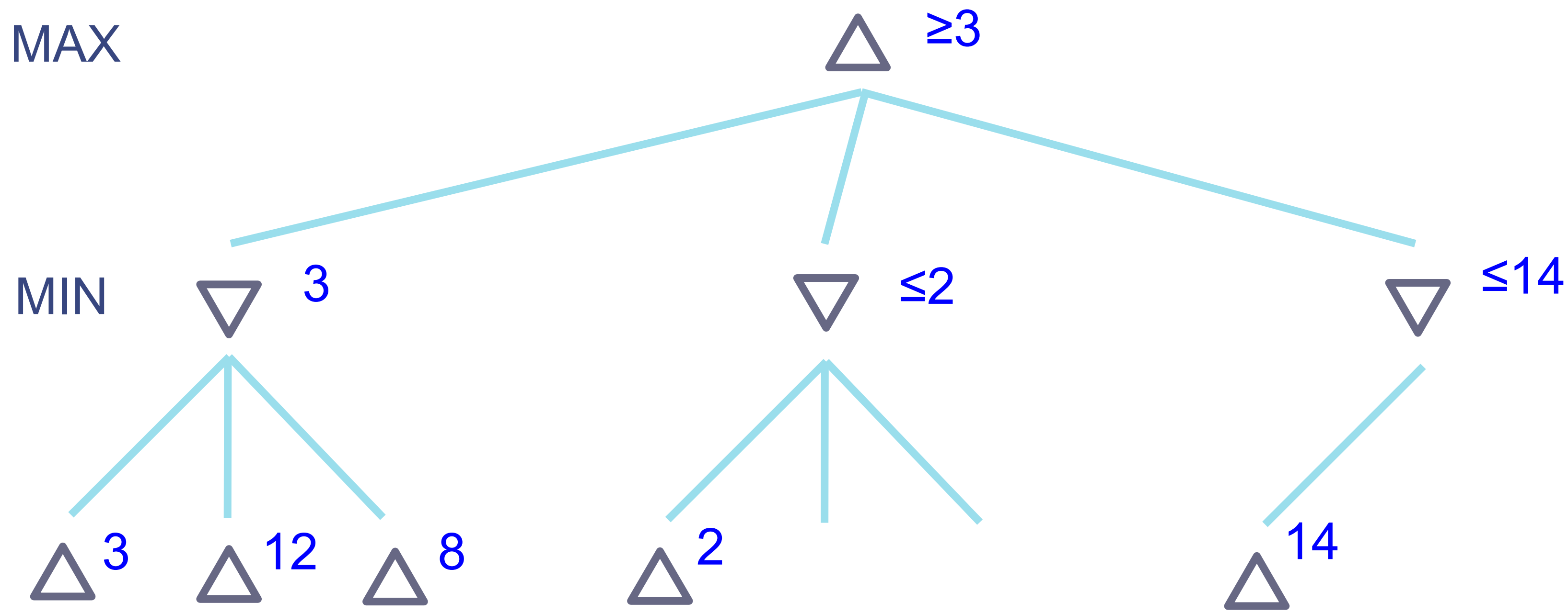


α-β Pruning Example

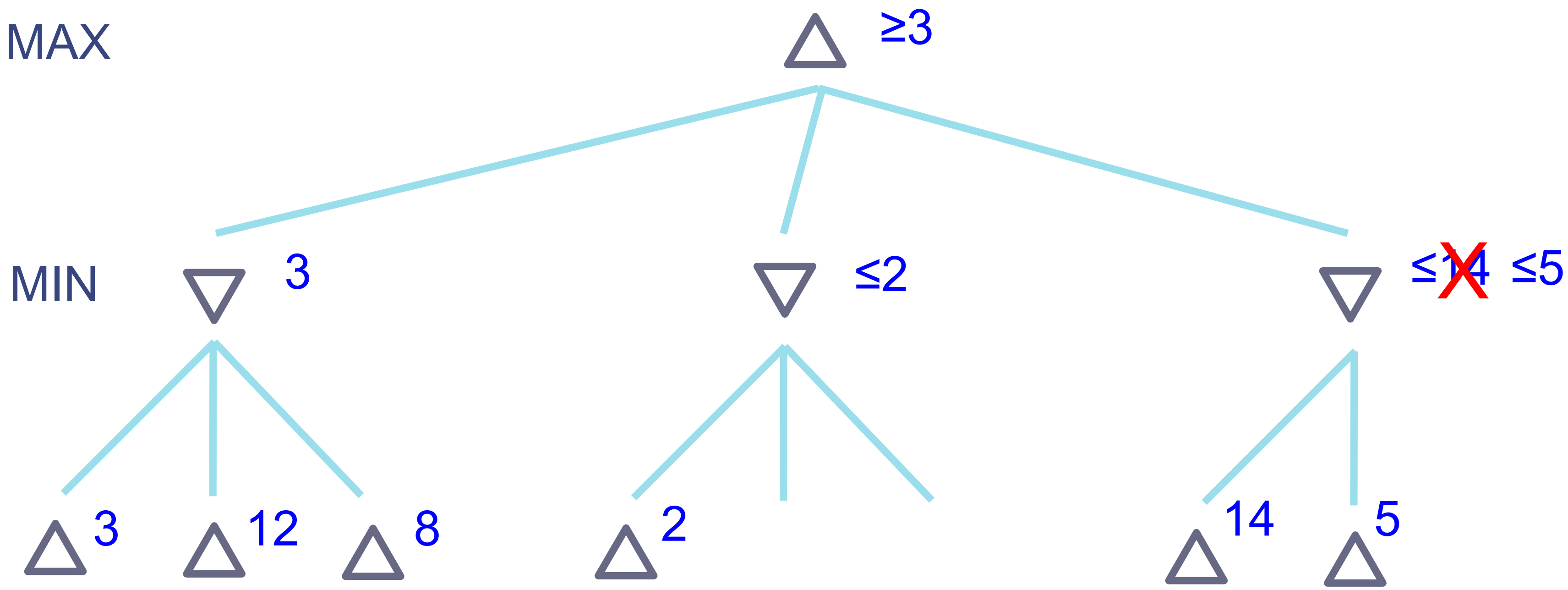


The opponent can force a utility of 2 or worse down this branch. But we already have $\alpha = 3$ So there's no need to look further here

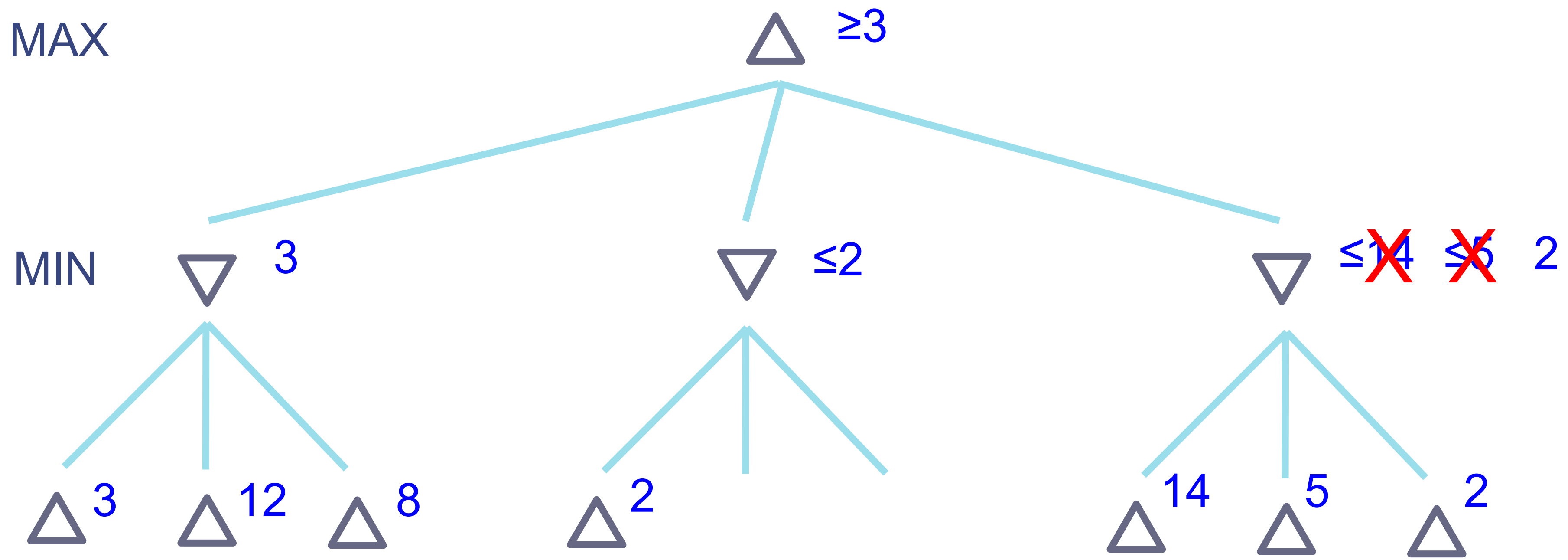
α-β Pruning Example



α-β Pruning Example



α-β Pruning Example



The Alpha-Beta Algorithm

- The Minimax algorithm is extended. Here's the new `MinValue`. `MaxValue` is analogous with roles of α and β reversed
- Need to maximise `MinValue(Result(a,S0),-∞,∞,1)`

```
function MinValue(s,α,β,d)
      if s is a terminal state or d = MaxDepth
            return Eval(s)
      else
            v := ∞
            for each action a possible in s
                  v := Min(v,MaxValue(Result(a,s),α,β,d+1)
                  if v ≤ α return v
                  else β := Min(β,v)
            return v
```

Optimality

- If there is no depth limit, minimax is guaranteed to find the optimal move against an optimal opponent
- Alpha-beta will find the same move as minimax (but faster)
- If the opponent is not optimal …
  - Consider an opponent that picks random moves. Then minimax might not be the best strategy for maximising expected reward
- If there is a depth limit, then minimax finds the optimal move for the given limit and evaluation function

Expected Utility

- If the opponent picks random moves, we should pick the move with maximum *expected* utility
- Here, mimimax would choose A_1, but the best move is A_3