Artificial Intelligence
A* + Adversarial Search

Michael McConnell

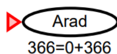
University of Vermont

March 15, 2022

A*

- We now take the cost of getting to the nodes into account as well as our estimate of the cost of getting to the goal from the node.
- We define an evaluation function $f(n) = g(n) + h(n)$
- $g(n)$: the cost so far to reach n
- $h(n)$: the cost to the goal from n
- The expansion always happens with the node of a lowest f-value in the data structure.
- A* is using what is called an admissible heuristic. This means that $h(n) \leq h^*(n)$ where $h^*(n)$ is the true cost of n.
- We also require that the straight line distance in this example of an h(n) table NEVER overestiatmes the actual road distance.

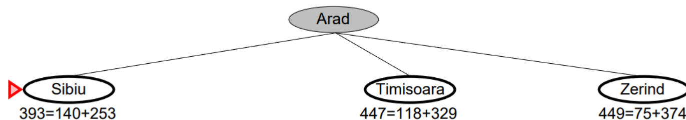
A* Example



[1]

[1] Artificial Intelligence: A Modern Approach, Norvig and Russell, 2020

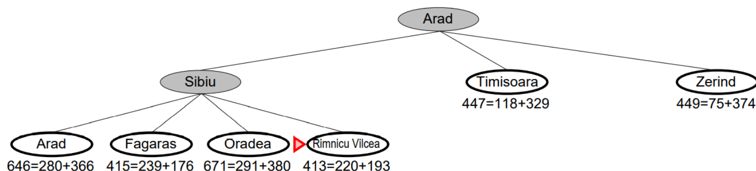
A* Example



[2]

[2] Artificial Intelligence: A Modern Approach, Norvig and Russell, 2020

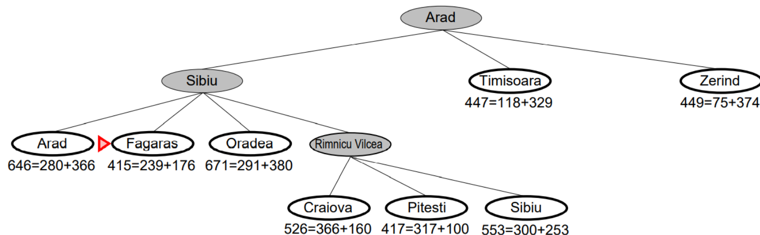
A* Example



[3]

[3] Artificial Intelligence: A Modern Approach, Norvig and Russell, 2020

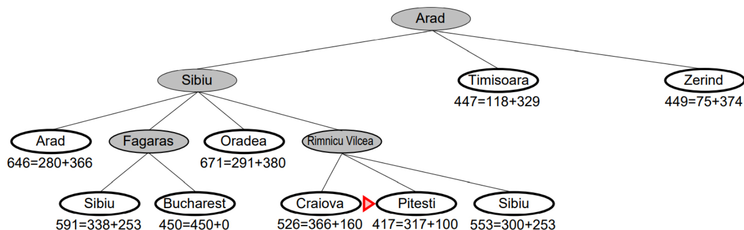
A* Example



[4]

[4] Artificial Intelligence: A Modern Approach, Norvig and Russell, 2020

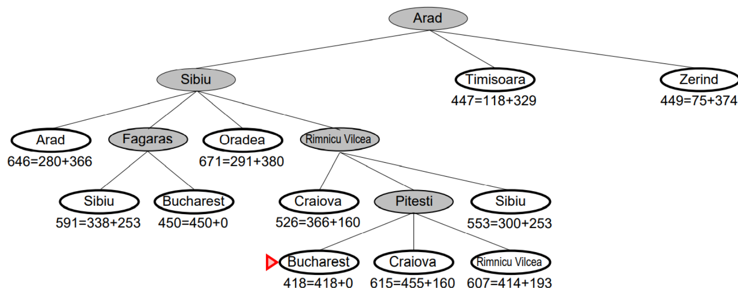
A* Example



5

[5] Artificial Intelligence: A Modern Approach, Norvig and Russell, 2020

A* Example



[6]

[6] Artificial Intelligence: A Modern Approach, Norvig and Russell, 2020

A* Properties

Completeness

- It is complete unless there are infinitely many nodes with f() $\leq$ f(G)

A* Properties

Time Complexity

- Exponential in terms of the relative error in the h() value table X the length of the solution.

A* Properties

Space Complexity

- Keeps all nodes in memory

A* Properties

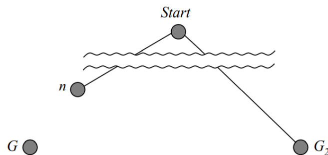
Optimality

- Yes.

A* Properties

Optimality

- Suppose there is some suboptimal goal G2 that has been generated and is in the queue. Let n be an unexpanded node on a shortest path to an optimal goal G.
- Since f(G2) $\geq$ f(n), A* cannot ever select G2 for expansion.



$$
\begin{aligned}
f(G_2) &= g(G_2) && \text{since } h(G_2) = 0 \\
&> g(G_1) && \text{since } G_2 \text{ is suboptimal} \\
&\geq f(n) && \text{since } h \text{ is admissible}
\end{aligned}
$$

[7]

[7] Artificial Intelligence: A Modern Approach, Norvig and Russell, 2020

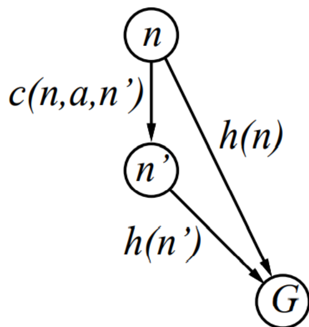
A* Optimality

- h(n) is admissible
- We are always assumping that the cost of any action between two states is greater than zero and can't be arbitrarily small.
- So if we take h*(n) (which recall is the cost of an optimal path from n to a goal node). An admissible heuristic is said to satisfy the condition that h(n) $\leq$ h*(n)
- h(n) is nonnegative and an underestimate of the cost of the shortest path from n to a given goal node. It is always optimistic.
- h(g) $=$ 0 for any goal node g
- h*(n) $= \infty$ if there is no path from n to a goal.

Monotonicity

- A monotone/consistent heuristic fully satisfies the triangle inequality. So, for nodes n and n' and for all actions we can take the following holds.
- $h(n) \leq c(n, a, n') + h(n')$
- C() is the cost of getting from the terminal state of n to the terminal state of n' via some action a.
- (There is obviously going to be more than one action between teh two nodes, in this example there is only one. In an example where there is more than one action, the inequality has to hold for all of them.)
- Monotonicity implies admissibility.
- Note: You create heuristics that are admissible but that are NOT monotonic. In these cases time and space complexity are the same, and completeness still holds. Optimality can still hold as well but there needs to be another argument added into the function.

A* Lemma



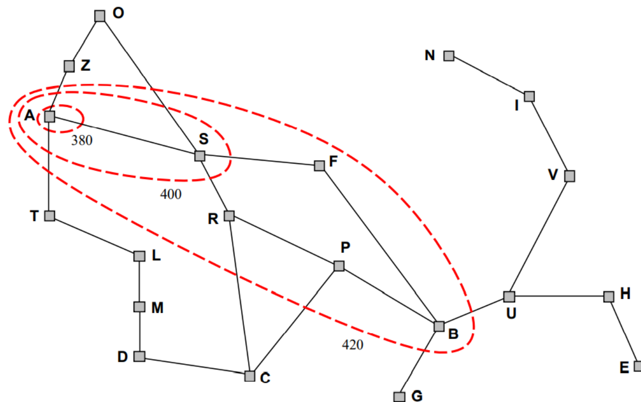
[8]

[8] Artificial Intelligence: A Modern Approach, Norvig and Russell, 2020

F-cost Contours

- f(n) along any given path is always going to be decreasing. One way to visualize this is to say that along a given path you can draw contours around all the nodes that are increasing in cost in concentric bands fanning out from a given node.

f-cost Contours



[9]

[9] Artificial Intelligence: A Modern Approach, Norvig and Russell, 2020

Beam Search

Completeness

- We can reduce the space complexity of algorithms that expand out in a 'best-first search' style like A* and Greedy search by simply adding a size limit on the frontier queue.
- If we keep the frontier smaller than some fixed size we can only search the top most promising nodes.
- Beam search is obviously not optimal, there is no guarantee it will find the best solution.
- Beam search is also not complete. There is a chance that the only route to a goal state could be pruned off the frontier and thus we would never reach it.

Heuristic Examples

- One example shown in the book that is illustrative is the 8-puzzle. We can design different heuristics for this, both of which are admissible.
- h1(n) = number of misplaced tiles.
- h2(n) = the total distance each tile is from its desired location.

8-puzzle Example

7	2	4
5		6
8	3	1

Start State

1	2	3
4	5	6
7	8	

Goal State

[10]

- h1(S) $= 6$
- h2(S) $= 14$

[10] Artificial Intelligence: A Modern Approach, Norvig and Russell, 2020

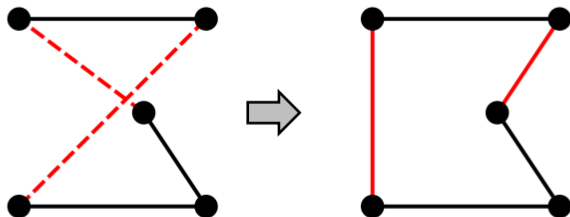
Relaxed problems and heuristics

- We can design optimal heuristics from relaxed versions of problems.
- If the rules of the 8-puzzle are changed so that a tile can move anywhere then h1(n) gives the shortest solution for example.
- If the rules of the 8-puzzle were somehow relaxed so that a tile can move to any adjacent square then h2(n) is going to give the shortest solution each time.
- Bottom Line: The optimal solution of a properly 'relaxed' problem is going to be no greater than (or at least as expensive as) the optimal solution cost of the real problem.

Iterative Improvement Algos

- The flipside to what we have been talking about are optimization problems where any solution is 'good' the goal state is the only thing we care about and any path to get there is fine.
- In these kinds of cases the state space is equivalent to a set of complete configurations, and we might want to find an optimal configuration or something that satisfies a set of constraints intelligently.
- Iterative improvement algorithms keep a single 'current' state and try to improve upon it over time.

Travelling Sales Problem

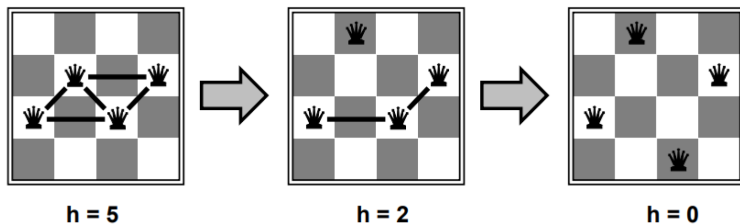


[11]

- Start off the problem with some complete route of the problem space.
- And then you might perform pairwise exchanges on the graph (doing just this simple pairwise exchange can get you within 1% of the optimal solution oftentimes).

[11] Artificial Intelligence: A Modern Approach, Norvig and Russell, 2020

N-Queens Example



[12]

- Put queens on a chessboard with no two queens in the same row, column or diagonal (make them so they aren't attacking any other pieces).
- Move a queen to reduce the number of conflicts.

[12] Artificial Intelligence: A Modern Approach, Norvig and Russell, 2020

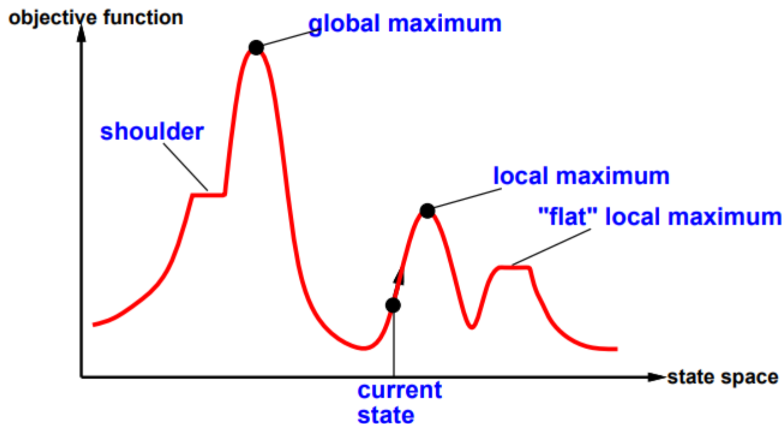
Hill Climbing Algorithm

```
function HILL-CLIMBING(problem) returns a state that is a local maximum
    inputs: problem, a problem
    local variables: current, a node
                     neighbor, a node

    current ← MAKE-NODE(INITIAL-STATE[problem])
    loop do
        neighbor ← a highest-valued successor of current
        if VALUE[neighbor] ≤ VALUE[current] then return STATE[current]
        current ← neighbor
    end
```

[13]

[13] Artificial Intelligence: A Modern Approach, Norvig and Russell, 2020

Hill Climbing Chart



[14]

[14] Artificial Intelligence: A Modern Approach, Norvig and Russell, 2020

Simulated Annealing Pseudocode

```
function SIMULATED-ANNEALING( problem, schedule) returns a solution state
    inputs: problem, a problem
            schedule, a mapping from time to "temperature"
    local variables: current, a node
                     next, a node
                     T, a "temperature" controlling prob. of downward steps

    current ← MAKE-NODE(INITIAL-STATE[problem])
    for t ← 1 to ∞ do
        T ← schedule[t]
        if T = 0 then return current
        next ← a randomly selected successor of current
        ΔE ← VALUE[next] – VALUE[current]
        if ΔE > 0 then current ← next
        else current ← next only with probability e^(Δ E/T)
```

[15]

[15] Artificial Intelligence: A Modern Approach, Norvig and Russell, 2020

Adversarial Search

- The tree and graph searches we have been looking at so far are models that allow an agent to have free roam and complete control of the given environ that they are operating in.
- More formally: A state in the environ for these previous models does not change unless the agent changes it.
- But in the real world obviously things in the environment can change outside of the control of the agent.
- If we program a robot to perform an A* search over a graph-shaped maze we have laid out on the floor, what does it do when it soemthing falls in the way of the path it wants to take?
- Search alone can merely find a path to a goal state, but the actions of another agent or event might attempt to stop you from reaching that optimal goal.

Adversarial Search Cont.

- We need to take the model of search we have previously discussed and add on features which allow us to handle changes that are out of our agents control.
- The main way of discussing this is by modelling the search tree as a game between our agent and some other agent(s).
- The simplest way to model these opponents is to say that we are trying to maximize our own scores in the game (some score vector), while the other agents are trying to maximize their own score (usually at the cost of minimizing ours).
- Each agent is trying to alter the world in some way that allows it to best benefit itself.

Brief Examples of Game Types

- **Perfect Information**: Types of games where all aspects of the game state are fully observable to all the players. There are also imperfect information games where some or all aspects of the game are hidden (think poker).
- **Deterministic Games**: Games with no 'chance' involved in their play, these can be either perfect or imperfect. Chess is a deterministic perfect information game.
- **Zero-Sum Games**: These are games that are fully competitive, if one player wins, the other players lose. Again a good example is poker, which involves both imperfect information and nondeterminsitic play.

Two-Player Zero-Sum Games

- In these types of games, we can model the game tree as essentially a search tree. The different depths of the tree reflect the alternating moves between the players as the game progresses.
- At each depth we are performing some search over the current game sub-tree in order to decide our next move.
- Neither player decides where to go in the game tree themselves. After one player moves to a given state the second player then decides which of the children of that state get chosen.
- Since the moves of the second player will change the direction of the game we have to come up with some strategy to overcome them basee on what they pick.

Minimax Algorithm

- **Simple:** Always assume that the other player is going to make the best move that they can make. After doing so we always play a move that will minimize the payoff of the other player. By minimizing the other player, we maximize ours.
- Gives us perfect play for deterministic, perfect-information games (with obvious space/time complexity caveats on this statement).
- Note: If you know ahead of time that the opponent is going to play badly there are massive optimizations that can be made which can give you better payouts.

Minimax Algorithm

- The values for each state are what the player Max (the root node player) will get for their score if both players choose their best moves in the tree.
- If the opponent plays poorly the Max player can do better, but they won't ever do worse than the play given by the minimax algorithm.
- Reiterate: For the pseduocode on the next slide to work the game tree is assumed to be finite.

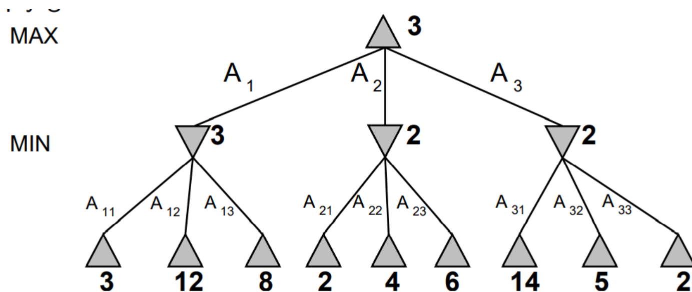
Minimax Algorithm Pseudocode

```
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 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

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

[16]

[16] Artificial Intelligence: A Modern Approach, Norvig and Russell, 2020

Minimax Example



[17]

[17] Artificial Intelligence: A Modern Approach, Norvig and Russell, 2020

- It is complete only if the tree we are looking at is finite. (Note that chess for example is technically a finite game with 10^{27586} possible games.)

Minimax Properties
Time Complexity

- $O(b^m)$

Minimax Properties

Space Complexity

- $O(bm)$

Minimax Properties

Optimality

- Yes, if the opponent is also playing optimally.

Alpha-Beta Pruning

- There are obvious optimizations we can perform that still give us the same completeness and optimality properties but that improve our complexity.
- To make a correct decision we do not have to look at the entire tree, after generating only some children at a given depth we can infer that a particular branch will not be picked and can thus stop evaluating.
- There are two types of cuts, max-cuts and min-cuts (referred to from here on as alpha cuts and beta cuts respectively).

Alpha-Beta Pruning Algo

```
function ALPHA-BETA-DECISION(state) returns an action
   return the a in ACTIONS(state) maximizing MIN-VALUE(RESULT(a, state))
```

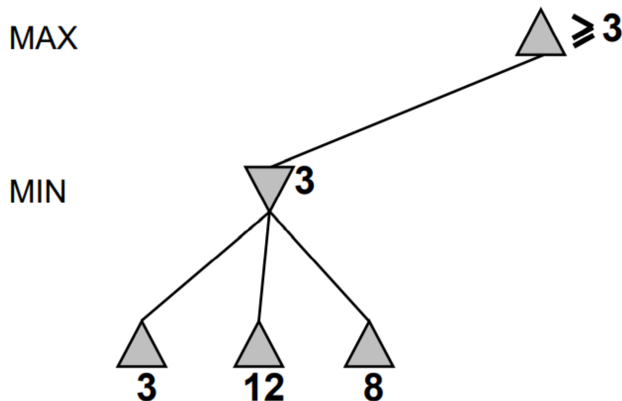
```
function MAX-VALUE(state, α, β) returns a utility value
   inputs: state, current state in game
           α, the value of the best alternative for  MAX along the path to state
           β, the value of the best alternative for  MIN along the path to state
   if TERMINAL-TEST(state) then return UTILITY(state)
   v ← −∞
   for a, s in SUCCESSORS(state) do
      v ← MAX(v, MIN-VALUE(s, α, β))
      if v ≥ β then return v
      α ← MAX(α, v)
   return v
```

```
function MIN-VALUE(state, α, β) returns a utility value
   same as MAX-VALUE but with roles of α, β reversed
```

[18]

[18] Artificial Intelligence: A Modern Approach, Norvig and Russell, 2020

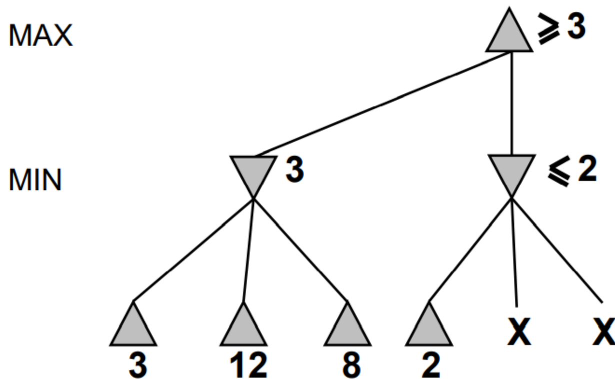
Alpha-Beta Pruning Example



[19]

[19] Artificial Intelligence: A Modern Approach, Norvig and Russell, 2020

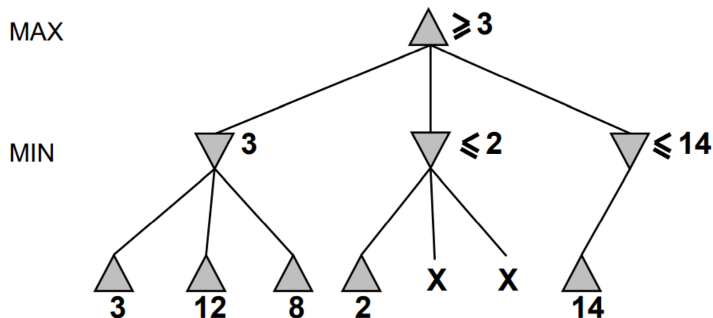
Alpha-Beta Pruning Example



[20]

[20] Artificial Intelligence: A Modern Approach, Norvig and Russell, 2020

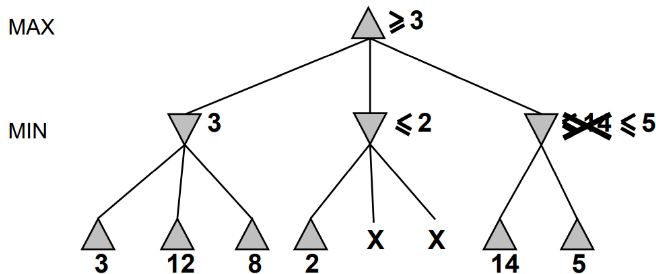
Alpha-Beta Pruning Example



[21]

[21] Artificial Intelligence: A Modern Approach, Norvig and Russell, 2020

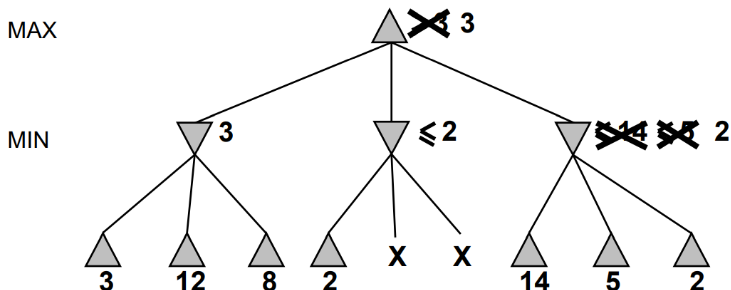
Alpha-Beta Pruning Example



22

[22] Artificial Intelligence: A Modern Approach, Norvig and Russell, 2020

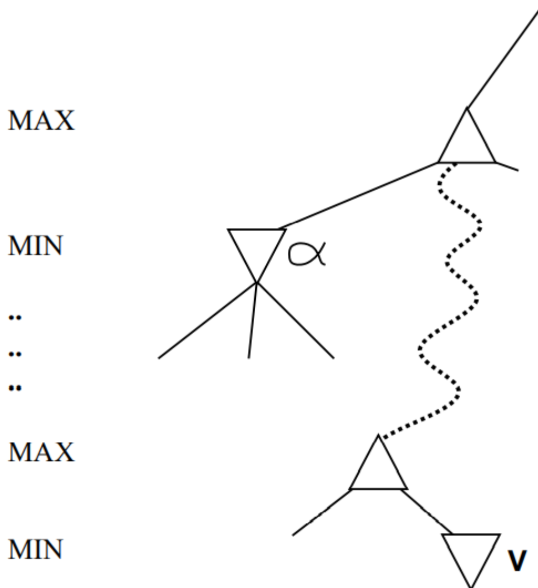
Alpha-Beta Pruning Example



[23]

[23] Artificial Intelligence: A Modern Approach, Norvig and Russell, 2020

Alpha-Beta Pruning



Alpha-Beta Properties

- Pruning (as we said) does not impact the final result.
- If we have the move branches ordered in an ideal way depending on the game we can improve the effectiveness of our pruning algorithm.
- With an ideal or 'perfect' ordering it can be shown that the time complexity reduces to $O(b^{m/2})$
- What does this mean in practice? In theory, you can search twice as deep on average.
- Aside: When IBM added Alpha-Beta pruning to their chess computer Deep Blue the average branches at each edge were reduced from 35 down to 6.

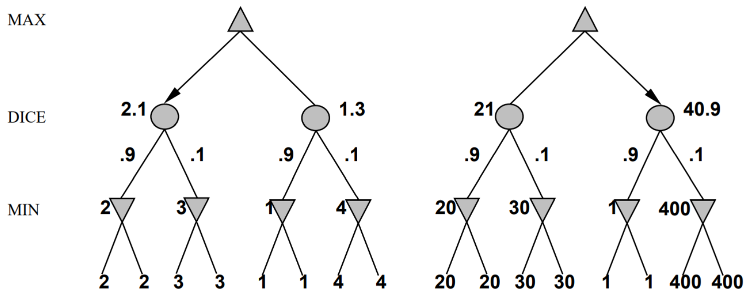
Nondeterministic Minimax

```
...
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)
...
```

[25]

[25] Artificial Intelligence: A Modern Approach, Norvig and Russell, 2020

Dice Example



[26]

[26] Artificial Intelligence: A Modern Approach, Norvig and Russell, 2020