CA318
Advanced Algorithms and AI Search

Games, Minimax, Alpha-Beta Pruning; Anytime Algorithms

Games

Game Playing Strategies

[non-]Adversarial Games

- In the previous lectures we focused on optimal search strategies when there was a single, definable goal (**G**)
- Even though these search strategies had different implementations, they were characterized by all having a **non-adversarial** nature
- In other words, when searching from the optimal path from **S-G**, we did not need to worry about an opponent blocking or impeding our progress

Adversarial Games

Now we move on to a new problem:

- How to search for the optimal solution in a space in which an opponent (or set of opponents) seeks to prevent this.
- These are **competitive environments**,
  - two or more agents have conflicting goals, giving rise to adversarial search problems.
- These classes of problem are called **Adversarial Search and Games**

Game Theory - 3 Views

There are at least three ways to analyze multi-agent environments:

#1 A very large number of agents:

- consider them in the aggregate as an **economy**,
- allowing us to do things like predict that increasing demand will cause prices to rise, without having to predict the action of any individual agent
- These are sometimes called **zero-sum** games

Game Theory - 3 Views

#2 A stochastic hostile environment:

- adversarial agents are a part of the environment
  - a part that makes the environment nondeterministic.
- But if we model the adversaries in the same way that, say, rain sometimes falls and sometimes doesn't
- we miss the idea that our adversaries are actively trying to defeat us,
  - whereas the rain supposedly has no such intention

Game Theory - 3 Views

#3 A stochastic hostile environment:

We explicitly model the adversarial agents with the techniques of adversarial game-tree search.

This is the approach we will follow here.

2 Player Zero Sum Games

perfect information; no win-win

No win-win

- The games most commonly studied within AI (such as chess and Go) are what game theorists call **deterministic**, two-player, turn-taking
- "Perfect information" is a synonym for "fully observable,"
- "zero-sum" means that what is good for one player is just as bad for the other:
  - there is no "win-win" outcome.

Definition of a Game

A game can be formally defined with the following elements:

- S_0: The **initial state**, which specifies how the game is set up at the start.
- TO-MOVE(s): The player whose turn it is to move in state s.
- ACTIONS(s): The set of legal moves in state s.
- RESULT(s, a): The **transition model**, which defines the state resulting from taking action a in state s.
- IS-TERMINAL(s): A **terminal test**, which is true when the game is over and false otherwise. States where the game has ended are called **terminal states**.
- UTILITY(s, p): A **utility function** (also called an objective function or payoff function), which defines the final numeric value to player p when the game ends in terminal state s. In chess, the outcome is a win, loss, or draw, with values 1, 0, or $1/2$.[2] Some games have a wider range of possible outcomes—for example, the payoffs in backgammon range from 0 to 192.

Approaches to Playing Games

Using a computer to play a game

Ways to play a board game

There are four methods for a computer to play a game.

- Analysis + Strategy + Tactics = Move
- IF-THEN rules
- Look Ahead and Evaluate
- Brute Force - Evaluate the entire tree of Possibilities
- Look ahead as far as possible

#1 Analysis + Strategy + Tactic = Move

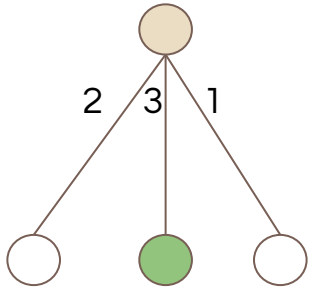
Analysis + Strategy + Tactics = Move

- This method is not computationally successful (or even possible)
- This is how you would play a game
- It has been tried many times and nearly always results in failure
- We don’t have an adequate way of understanding how a human does this

#2 IF-THEN Rules

IF-THEN rules

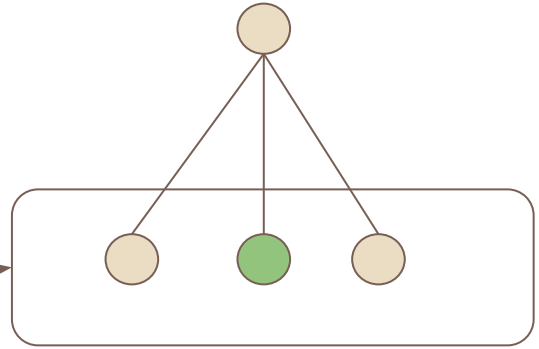
- Look at the board
- Rank the possible **moves**
- Pick the highest rank **move**
  - Doesn't work for chess
  - Does work for Draughts



The choice is obvious here

#3 Lookahead and Evaluate

Here we look ahead at the possible board situations reachable from the root and we ask **which of these situations is best?**



Board Static States

In order to make a decision here, we need a way to evaluate the board state.

Every board configuration is defined by a tuple of features

$$S = g(f_1, f_2, ..., f_N)$$

Each f maps to a chess characteristic such as King Safety, Pawn Structure, Number of Pieces etc.

Linear Scoring Polynomial

$$S = Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \cdots + w_n f_n(s) = \sum_{i=1}^{n} w_i f_i(s)$$

We are probably going we weight each feature f with some constant a, add these together, and compute a static value **S** for the board in each lookahead, given a board state **s**.

S is more formally called a **weighted linear function**

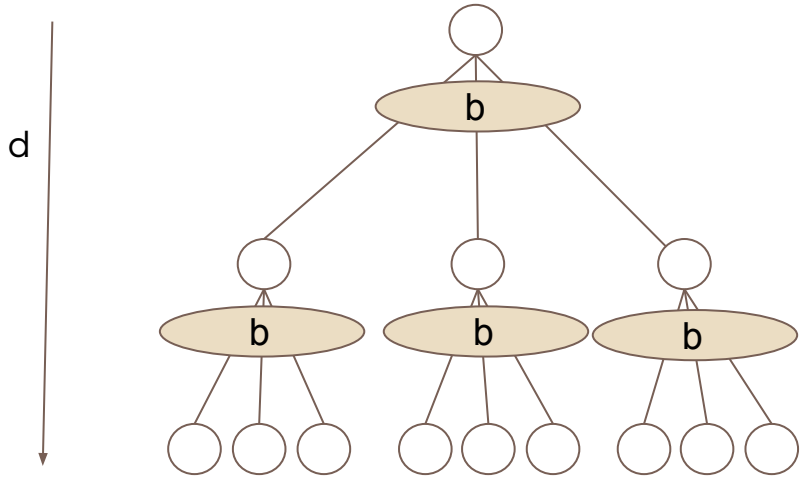
#4 Brute Force

- The idea here is to enumerate all possible paths, and pick the best path for the given situation.
- Before we use this approach however, we need to make sure that enumerating the entire search space is feasible
- Feasibility comes in two parts:
  - Storage feasibility
  - Compute feasibility
- To decide whether it is safe to brute force a problem, lets recap on some important information

Branching Factor and Levels

Branching Factor: The average number of children per node (**b**)

Depth: the total number of levels in the tree (**d**)



Terminal / Leaf Nodes

So the number of Terminal / Leaf nodes is a function of the depth and the branching factor

$$L_d = b^d, d = \{0, 1, 2..., N-1\}$$

Q: How many leaf nodes do we have where **d**=2 and **b**=3

Back to brute force chess

- Lets assume we have two players playing chess, each player makes 60 moves,
  - giving us a depth of 120
- Now, what is the average branching factor in chess?
  - Most people say this is around 10
- This gives us the number of leaf nodes for Chess Brute Force:

$$L = 10^{120}$$

Is this Feasible?

Lets see how feasible 10^{120} really is:

Size	Task
1×10^{80}	All the atoms in the universe
3×10^{7}	Seconds in a year
1×10^{9}	Nano seconds in a second
1×10^{10}	Number of years since universe began
10^{106}	= Every atom completes one computation every nanosecond since the beginning of time

Brute force - conclusion...

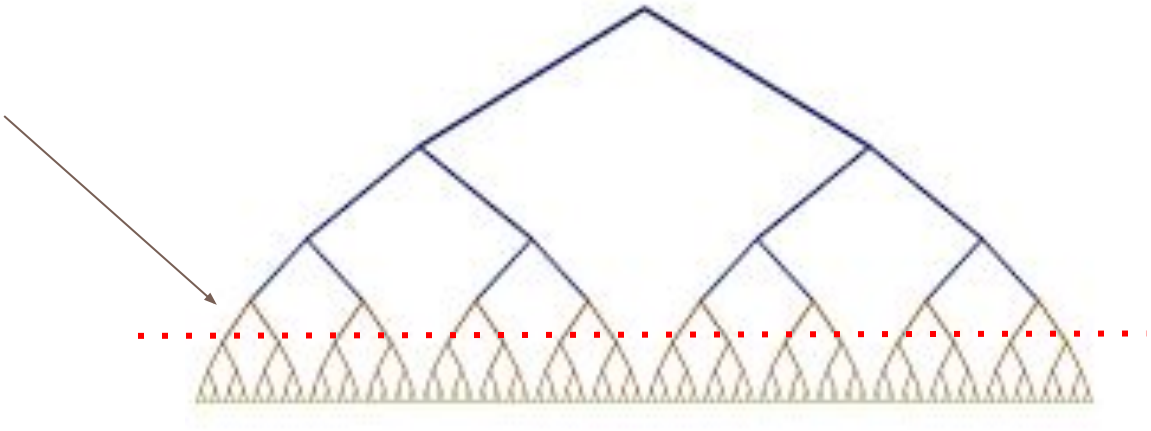
So, if all of the atoms in the universe were doing static evaluations at nanosecond speeds since the beginning of the Big Bang, we'd still be 14 orders of magnitudes short.

#5 Look ahead as far as possible

- As brute force is not feasible, we will look ahead
  - **as far as possible**
- And statically evaluate each of the leaf nodes found
- But how do we account for the adversarial nature of the game?
  - By alternately **minimizing** and **maximizing** the static values **S**

Limit the lookahead

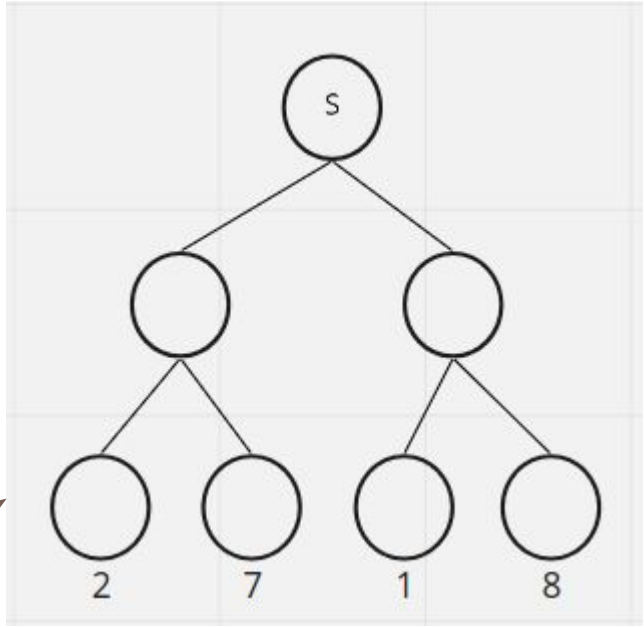
We will lookahead only so far as is computationally feasible (red dashed line), leaving the deeper leaf nodes unevaluated



Adversarial Games

Now we can model adversarial games by using a tree **d**=2 and **b**=2.

The principles shown here apply to all trees of arbitrary **b** and **d**.



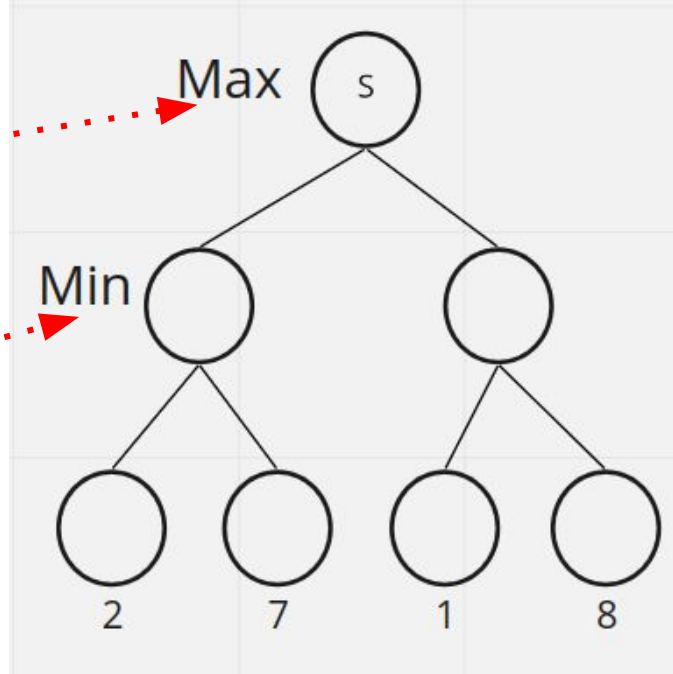
The value of the board from the perspective of the player at the top (ie, the computer)

Min and Max

The player at the top wants to drive the play - as much as possible - towards the big numbers.

We call this player the **maximizing** playrer

But the adversary, or opponent, does not have that objective. They wish to drive towards the small numbers. We call this the **minimizing** player

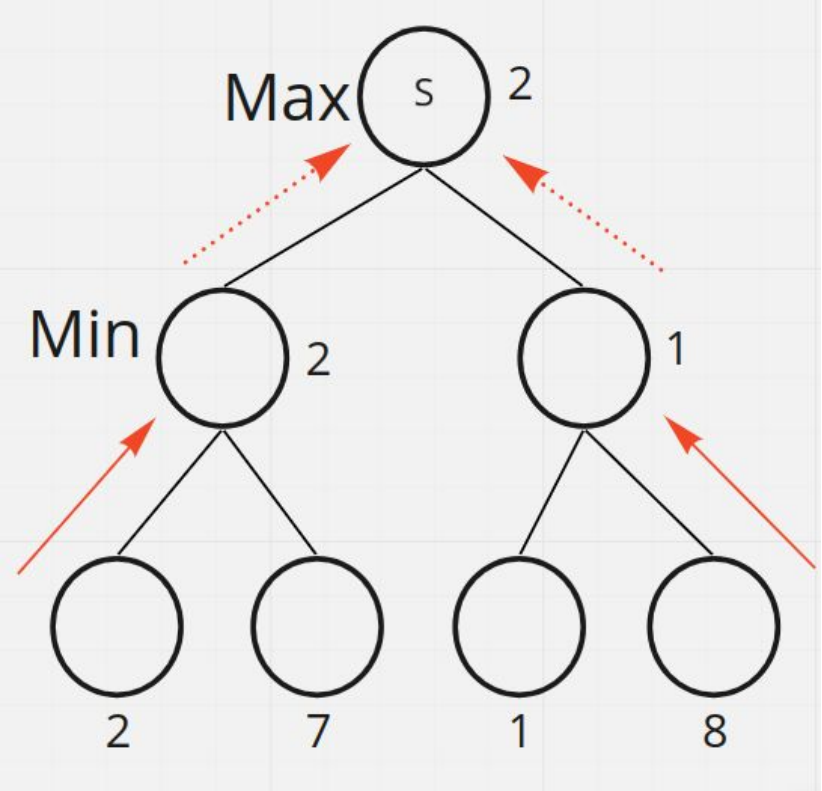


Lets step through this

This the minimax algorithm.

It's very simple.

You go down to the bottom of the tree, you compute static values, you back them up level by level, and then you decide where to go.



Compromise

- And the minimizer goes to the left, too, so the play ends up here:
  - far short of the 8 that the maximizer wanted
  - and less than the 1 that the minimizer wanted
- This is an adversarial game.
- You're competing with another player.
- So, you don't expect to get what you want

In essence, in this case, the game ends in a compromise or averaging effect

Complexity

Once again, lets reflect on the complexity here

Minimax (with ***n*** levels) has a complexity of:

$$\Theta(b^n)$$

Clearly this is **highly undesirable**. So we will try to improve this complexity by using ***αβ*** pruning

Aside

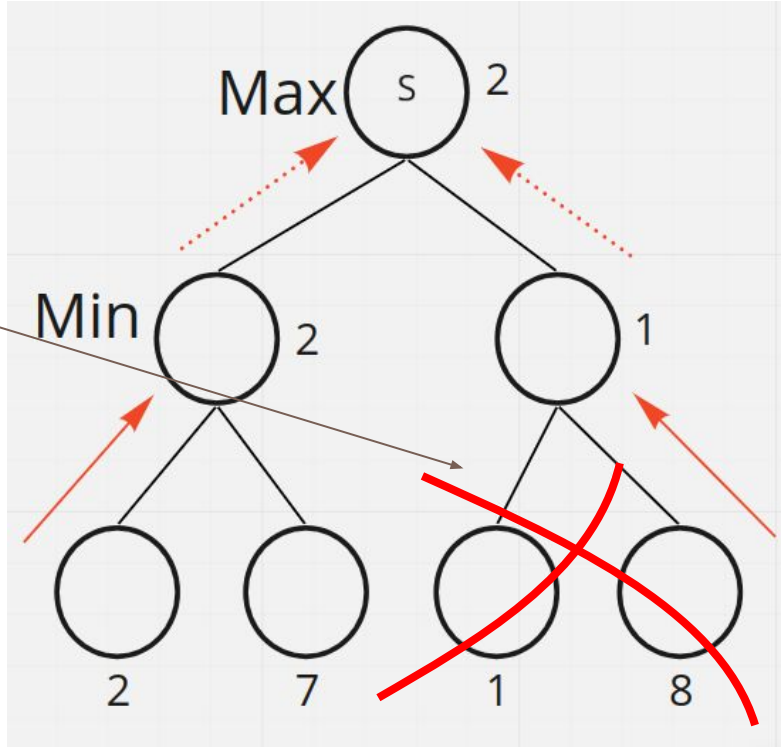
- A chess player that can get down 6 levels deep is probably a reasonable player
- But a player that can get down 15 or 16 levels deep, you could beat a grand master!
- Research has shown that if you get far enough down, the only static metric that really matters is piece count

$\alpha\beta$ Pruning Search

Similar to Branch and Bound

αβ pruning

- The minimizer pushes **LHS**=2 and **RHS**=1 up the tree.
- When the Maximizer sees the **RHS**=1 it treats this right branch as if it doesnt exist.
- Thereby eliminating possibly huge computational effort
- The 8 doesn't matter and indeed could have been very large.
- **The RHS is nonetheless pruned**



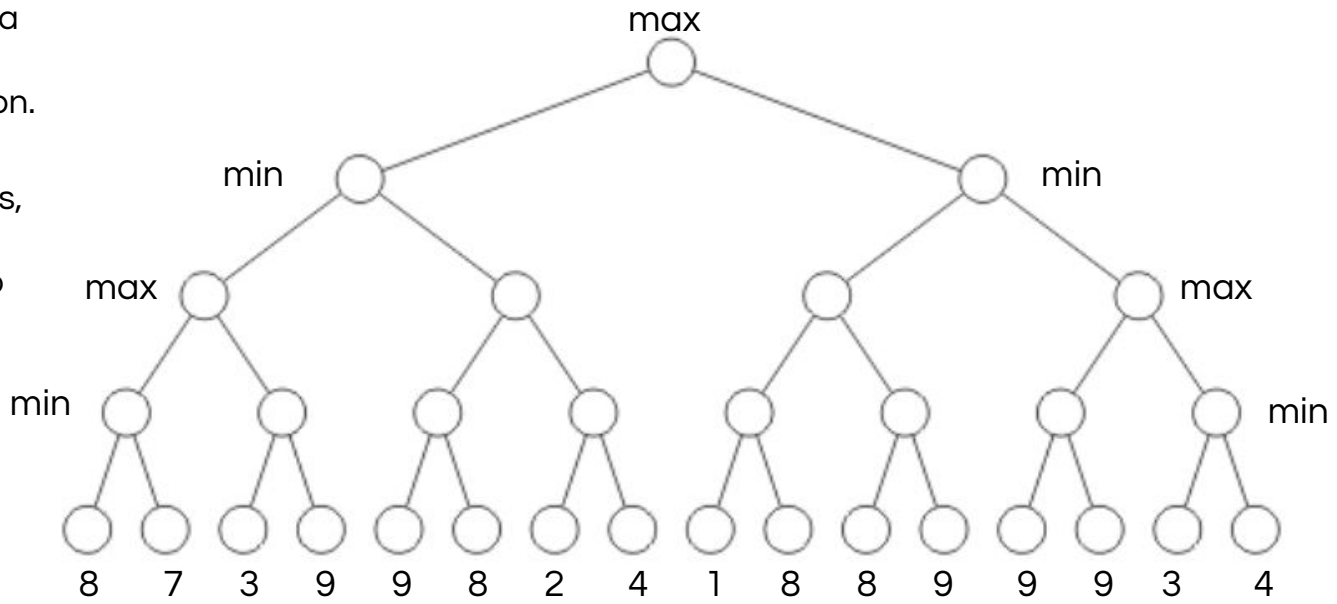
$\alpha\beta$ Comments

- The $\alpha\beta$ pruning approach is a layer on top minimax
- It is not a replacement for minimax
- Just like we added features to branch and bound to make it more efficient

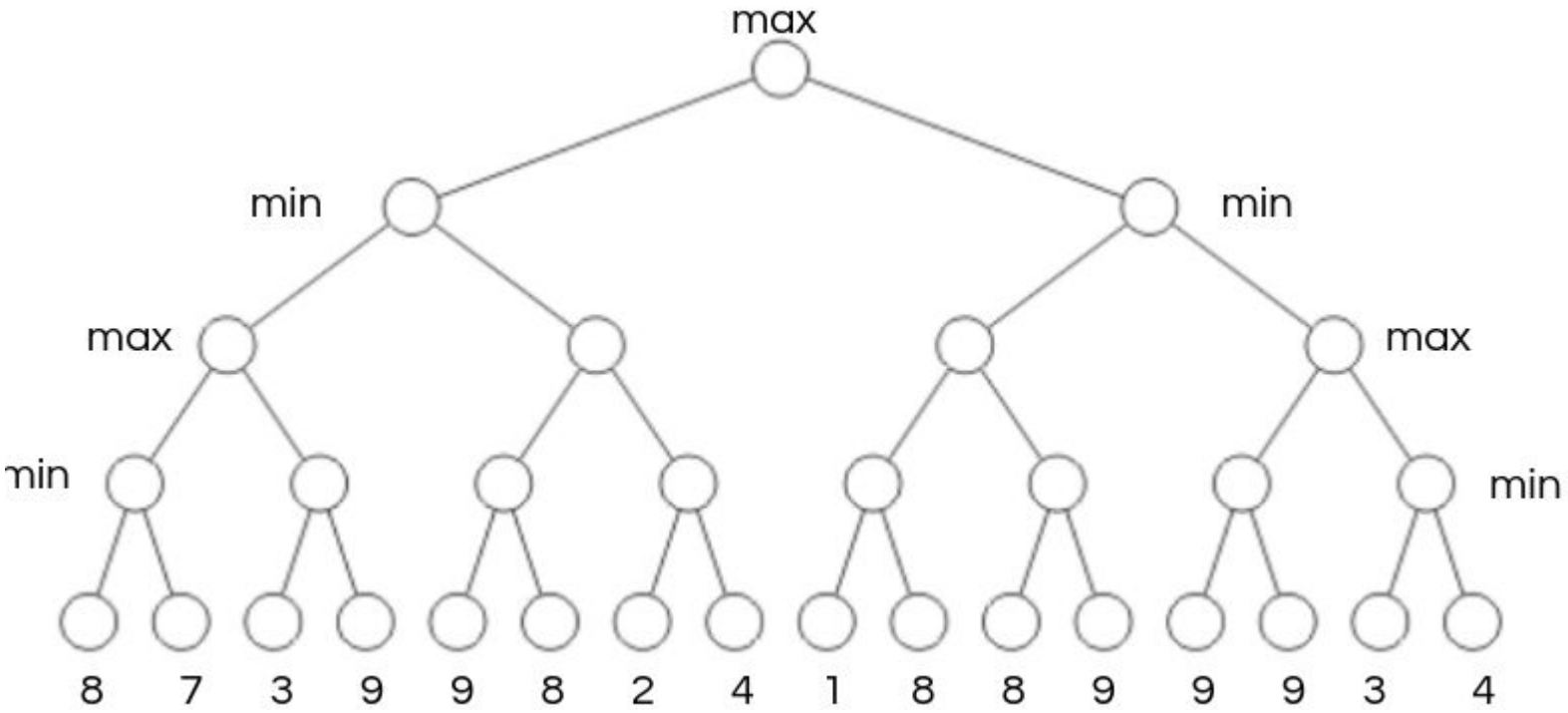
Tree of d=4

Now that we are clear on minimax with ***αβ*** pruning lets consider a case where d=4, a more complex situation.

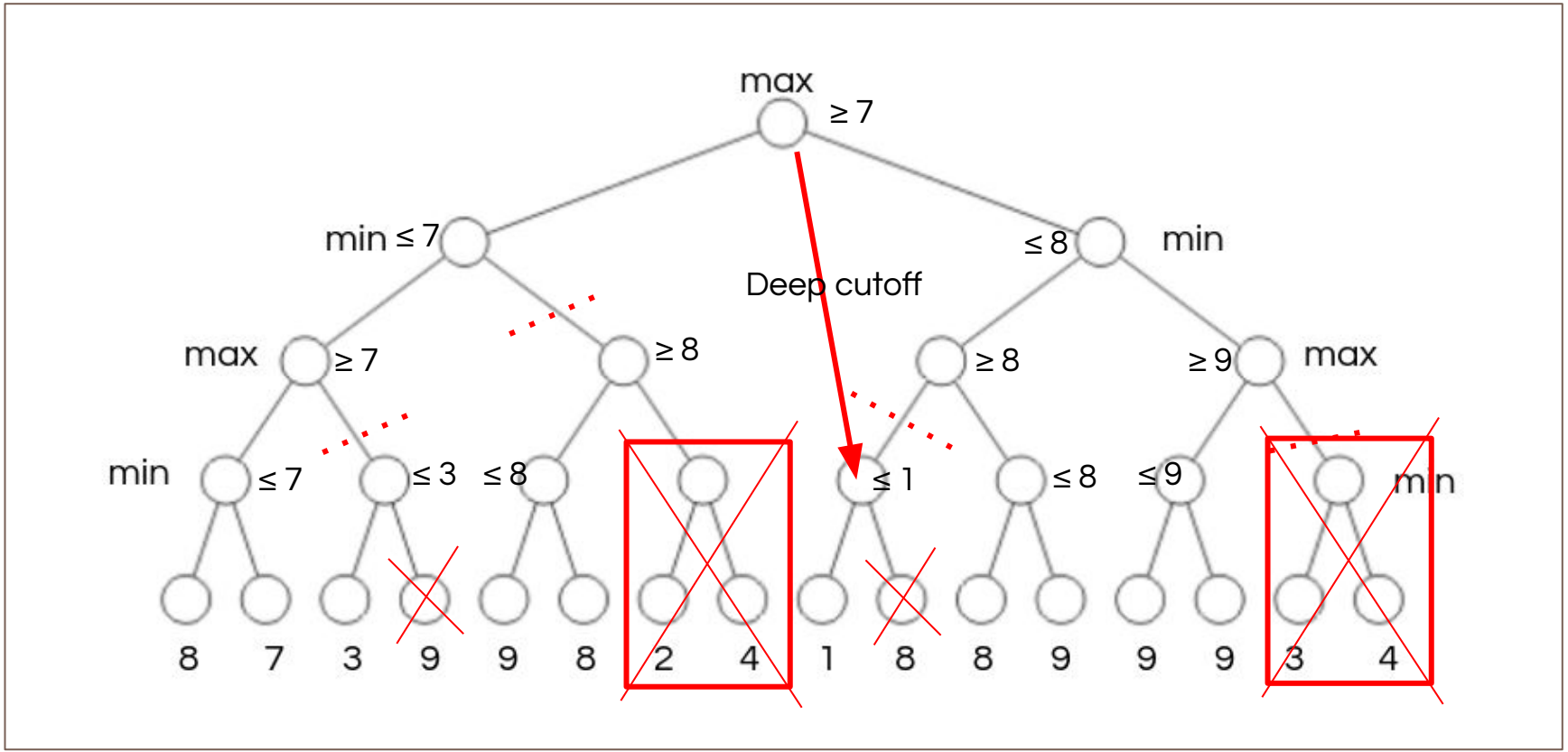
We will also **assume lazy** evaluation. That is, we won't evaluate a leaf node if there is no sense in doing so



Lets work through this



Solution


max
≥ 7
min ≤ 7
≤ 8
min
Deep cutoff
max
≥ 7
≥ 8
≥ 8
≥ 9
max
min
≤ 7
≤ 3
≤ 8
≤ 1
≤ 8
≤ 9
min
8
7
3
9
9
8
2
4
1
8
8
9
9
9
3
4

Final Solution

- The value of 8 the we finally arrive at at the top of the tree is not the minimum and its not the maximum
- It's the compromise number that's arrived
  - by virtue of the fact that this is an adversarial situation.

How good is this solution

In an optimal arrangement, and using ***αβ*** pruning, the actual number of static evaluations is given to us by:

$$S \approx 2b^{d/2}$$

The **d/2** exponent is what makes the difference from having to stop at **d=7** and being able to continue to **d=14**

Question

If we go down the same number of levels, how much less work do we do for **b=4, d=6**

$$S \approx 2b^{d} \quad \text{versus} \quad S \approx 2b^{d/2}$$

?

Answer

```
> d <- 6
> b <- 4
> b^d
[1] 4096
> b^(d/2)
[1] 64
> 4096/64
[1] 64
```

Pure minimax algorithm

An algorithm for calculating the optimal move using minimax—the move that leads to a terminal state with maximum utility, under the assumption that the opponent playa to minimize utility.

The functions M AX -VALUE and M IN -VALUE go through the whole game tree, all the way to the leaves, to determine the backed-up value of a state and the move to get there

```
function MINIMAX-SEARCH(game, state) returns an action
  player ← game.TO-MOVE(state)
  value, move ← MAX-VALUE(game, state)
  return move

function MAX-VALUE(game, state) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v, move ← −∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MIN-VALUE(game, game.RESULT(state, a))
    if v2 > v then
      v, move ← v2, a
  return v, move

function MIN-VALUE(game, state) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v, move ← +∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MAX-VALUE(game, game.RESULT(state, a))
    if v2 < v then
      v, move ← v2, a
  return v, move
```

Minimax with *αβ* pruning Algorithm

The alpha–beta search algorithm. Notice that these functions are the same as the MINIMAX -S EARCH functions, except that we maintain bounds in the variables α and β, and use them to cut off search when a value is outside the bounds

```
function ALPHA-BETA-SEARCH(game, state) returns an action
  player ← game.TO-MOVE(state)
  value, move ← MAX-VALUE(game, state, −∞, +∞)
  return move

function MAX-VALUE(game, state, α, β) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← −∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MIN-VALUE(game, game.RESULT(state, a), α, β)
    if v2 > v then
      v, move ← v2, a
      α ← MAX(α, v)
    if v ≥ β then return v, move
  return v, move

function MIN-VALUE(game, state, α, β) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← +∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MAX-VALUE(game, game.RESULT(state, a), α, β)
    if v2 < v then
      v, move ← v2, a
      β ← MIN(β, v)
    if v ≤ α then return v, move
  return v, move
```

Progressive Deepening

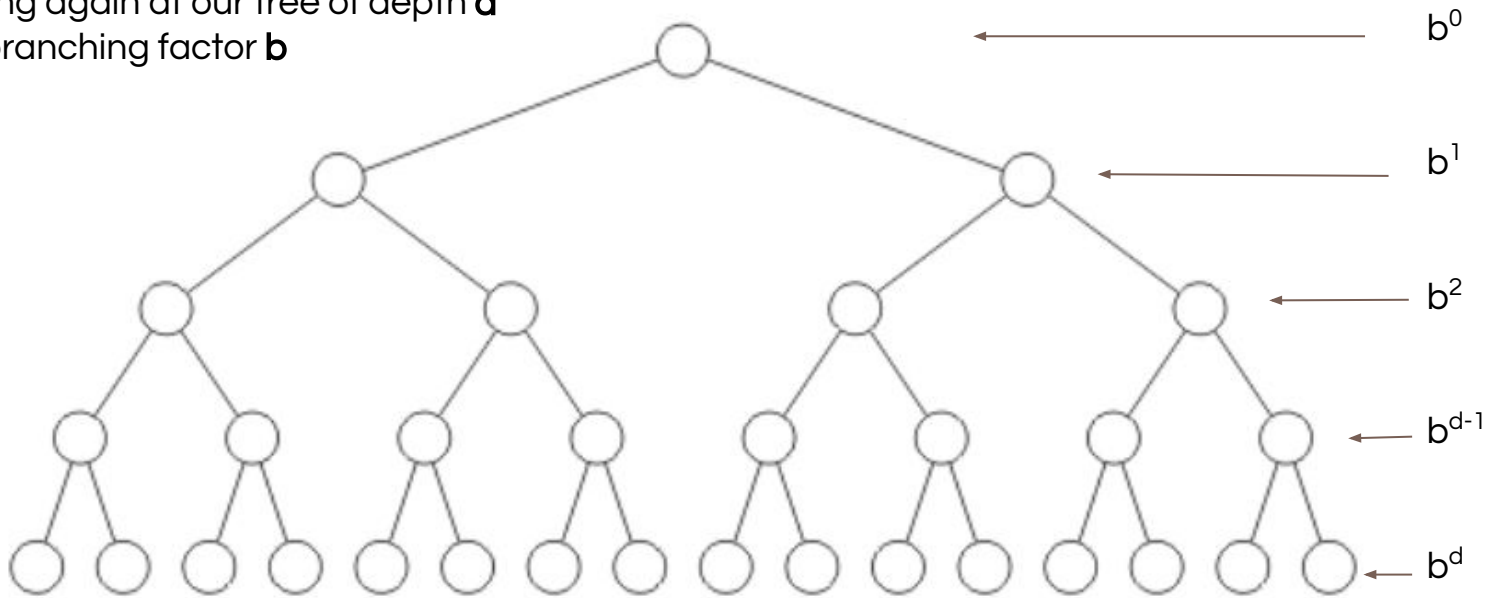
Anytime algorithms

Branching Factor

- Branching factor changes: ie, not constant
- It changes with each move
- It also changes with each game
- We will now consider a final improvement to minimax, known as progressive deepening
- Progressive Deepening is a type of **Anytime Algorithm**
  - No matter when it is called, it always has an answer

Progressive Deepening

Looking again at our tree of depth **d** and branching factor **b**



Bounding the computation time

- Clearly, descending the tree incurs more computations
- But we need to descend as deeply as possible to ensure the best results
- How do we handle the situation where, at level **d**, we spend too much time computing, and the clock expires?
  - We calculate $\mathbf{S = b^d}$ but run out of time
- Clearly we need to compute a solution at
  - $\mathbf{S = b^{d-1}}$ … but what if we run out of time here?
  - Calculate a $\mathbf{S = b^{d-2}}$

How much calculation do we save?

- Suppose **b**=10, we calculate **d**=4:
  - $S = 10^4 = 10000$
- But if we run out of time, or if the branching factor at **d** is too large, lets calculate S for the previous level **d-1**:
  - $S = 10^3 = 1000$
- So by calculating at **d-1** then we compute only 10% of the workload at **d**
- More generally, we compute **1/b** for each previous level
  - $S = 10^2 = 100$

We always have a move...

This is how much we spend getting our **anytime** insurance policy

$$S = 1 + b + b^2 + ... + b^{d-1}$$

This means

$$bS = b + b^2 + ... + b^d$$

$$bS - S = b^d - 1$$

$$S = \frac{b^d - 1}{b - 1} \approx b^{d-1}$$

What this means

So, with an approximation factored in

- the amount of computation needed to do insurance policies at every level
- is not much different from the amount of computation needed to get an insurance policy at just one level,
  - the penultimate level.

This is progressive deepening

Uneven Tree Development

- So far, we've pretended that the tree always grows in an even way to a fixed level.
- But there's no particular reason why that has to be so.
- Some situation down at the bottom of the tree may be particularly dynamic.
- In the very next move, you might be able to capture the opponent's Queen.

Uneven Tree Development

- So, in circumstances like that, you want to expand a little extra search.
- There's no particular reason to have the search go down to a fixed level.
- Instead, you can develop the tree in a way that gives you the most confidence that your backed-up numbers are correct.

Deep Blue Chess Player

Uses:

- Minimax
- $\alpha\beta$ Pruning
- Progressive Deepening
- Parallel Evaluation
- Uneven Tree Development

Finally … 1/2

- Is this a model of anything that goes on in our own heads?
- Is this a model of any kind of human intelligence?
- Is going down 14 levels what human chess players do when they win the world championship?

Finally … 2/2

- This is substituting raw power for sophistication.
- When a human chess master plays the game,
  - they have a great deal of chess knowledge in their head
  - and they recognize patterns.
- It's very clear that they've Chess masters develop a repertoire of chess knowledge
  - that makes it possible for them to recognize situations and play the game
- The play much more like our very first option #1.

Thank you

Any Questions?