Artificial Intelligence Foundation – JC3001

Lecture 11: Search IV: Adversarial Search II

Prof. Aladdin Ayesh (aladdin.ayesh@abdn.ac.uk)

Dr. Binod Bhattarai (binod.bhattarai@abdn.ac.uk)

Dr. Gideon Ogunniye, (g.ogunniye@abdn.ac.uk)

September 2025

Material adapted from:

Russell and Norvig (AIMA Book): Chapter 5

Russell and Norvig (AIMA Book): Chapter 17/18 (17.1/18.2)

Shoham and Leyton-Brown (Game Theory)

Course Progression

- Part 1: Introduction
  1. Introduction to AI ✓
  2. Agents ✓
- Part 2: Problem-solving
  1. Search 1: Uninformed Search ✓
  2. Search 2: Heuristic Search ✓
  3. Search 3: Local Search ✓
  4. **Search 4: Adversarial Search**
- Part 3: Reasoning and Uncertainty
  1. Reasoning 1: Constraint Satisfaction
  2. Reasoning 2: Logic and Inference
  3. Probabilistic Reasoning 1: BNs
  4. Probabilistic Reasoning 2: HMMs
- Part 4: Planning
  1. Planning 1: Intro and Formalism
  2. Planning 2: Algos and Heuristics
  3. Planning 3: Hierarchical Planning
  4. Planning 4: Stochastic Planning
- Part 5: Learning
  1. Learning 1: Intro to ML
  2. Learning 2: Regression
  3. Learning 3: Neural Networks
  4. Learning 4: Reinforcement Learning
- Part 6: Conclusion
  1. Ethical Issues in AI
  2. Conclusions and Discussion

Objectives

- Games ✓
- Non-deterministic search
- Adversarial Search
  - MinMax Search
  - Alpha-Beta Pruning
  - Expectimax

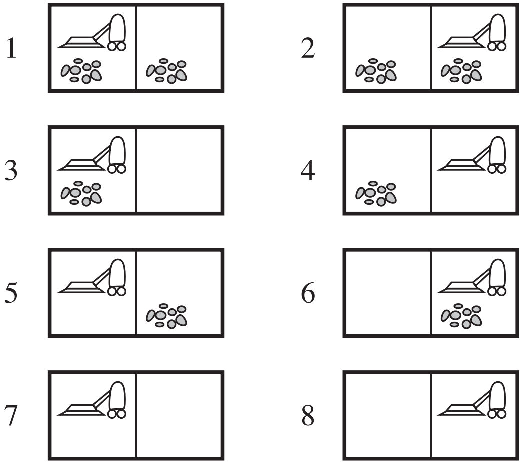
- Non-deterministic Search
- Adversarial Search

- We have seen search algorithms for environments that are **deterministic** and **fully-observable**
- When either, or both of these assumptions cannot be made, an agent needs to account for **percepts** in every move to narrow down the states where it is
- A solution can no longer be a flat **sequence** of actions, but rather a **contingency plan** (a.k.a. **strategy**)

Erratic Vacuum World

- Recall the vacuum world, but with a non-deterministic suck action:
  - When applied to a dirty square, the action cleans the current square, but sometimes also **cleans an adjacent square**
  - When applied to a clean square the action sometimes dirties the square
- This requires
  - Different **transition model**
  - Different **solution** concept

Non-deterministic Transitions



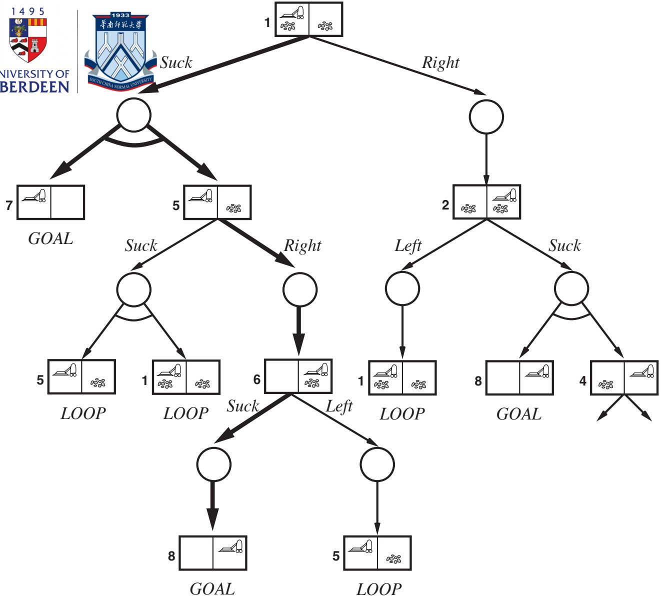
- **Result** from a state now becomes **Results**:

$$\text{Results}(1, suck) = \{5, 7\}$$

  Suck applied to state 1 results in the agent arriving either in states 5 or 7
- Plan is now a contingency plan (a tree):

  $[Suck,$ **if** $State = 5$ **then** $[Right, Suck]$ **else** $[]]$

- In deterministic environments, branching only occurs due to agent's choice (OR Nodes)
- In non-deterministic environments, the environment's choice must also be taken into account (AND Nodes)
- Solution is a subtree of the AND-OR tree that:
  - Has a goal node at every leaf
  - Specifies an action at each OR node
  - Includes every outcome branch of its AND nodes



First two levels of AND-OR Search Tree for erratic vacuum

- Or-Search: similar to regular search (agent has choice)
- And-Search: agent searching against the environment
- Notice failure condition on Or-Search

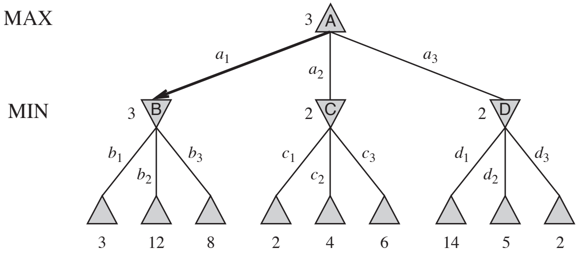
function And-Or-Graph-Search($problem$)
  return Or-Search($problem$.Initial-State, $problem$, [])

---

function Or-Search($state$, $problem$, $path$)
  if $problem$.Goal-Test($state$) **then return** [ ] ▷ *the empty plan*
  if $state$ is on $path$ **then return** $failure$
  for each $action$ in $problem$.Actions($state$) **do**
    $plan \leftarrow$ And-Search(Results($state, action$), $problem$, $[state|path]$)
    if $plan \neq failure$ **then return** $[action|plan]$
  return $failure$
function And-Search($states$, $problem$, $path$)
  for each s_i **in** $states$ **do**
    $plan \leftarrow$ Or-Search(s_i, $problem$, $path$)
    if $plan = failure$ **then return** $failure$
  return[**if** s_1 **then** $plan_1$ **else if** s_2 **then** $plan_2$ **else** $\ldots$ **if** s_{n-1} **then** $plan_{n-1}$ **else** $plan_n$]

- Non-deterministic Search
- Adversarial Search

Adversarial Optimal Decisions



- Each move we (MAX) makes has a response from MIN
- So the plan we look for is a contingent **strategy** with moves for:
  - The initial state; and
  - Every possible response from MIN
- We can calculate the value of the game recursively, assuming both players play optimally using the **Minimax** value

$$\text{Minimax}(s) = \begin{cases} \text{Utility}(s) & \textit{if } \text{Terminal-Test}(s) \\ \max_{a \in Actions(s)} \text{Minimax}(Result(s,a)) & \textit{if } \text{Player}(s) = \text{Max} \\ \min_{a \in Actions(s)} \text{Minimax}(Result(s,a)) & \textit{if } \text{Player}(s) = \text{Min} \end{cases}$$

Minimax Algorithm

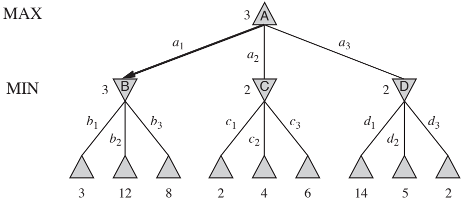
function Minimax-Decision($state$) **returns** $an\ action$
  return $\arg\max_{a \in Actions(s)}$ Min-Value($Result(state, a)$)

---

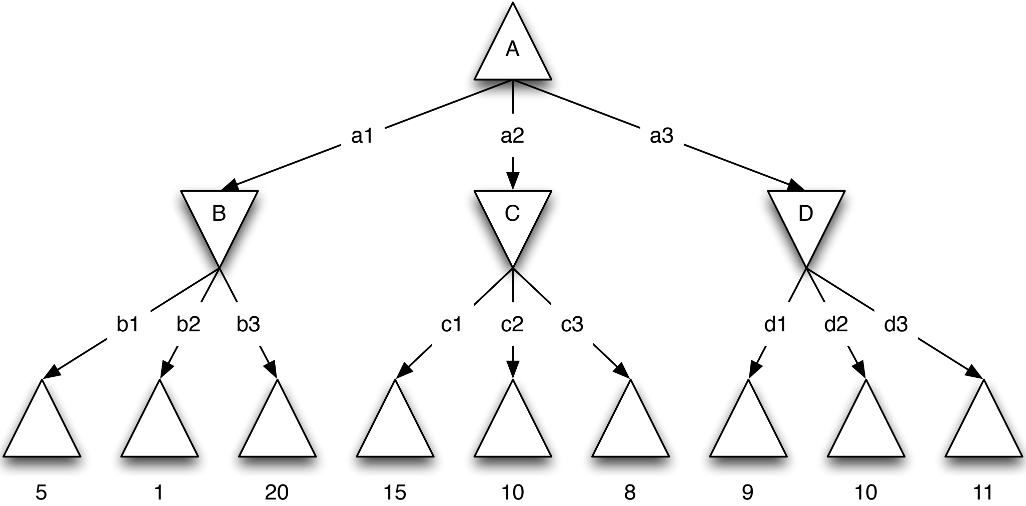
function Max-Value($state$) **returns** $a\ utility\ value$
  if Terminal-Test($state$) **then return** Utility($state$)
  $v \leftarrow -\infty$
  for each a **in** $Actions(state)$ **do**
    $v \leftarrow$ Max(v, Min-Value(Result($state, a$)))
  return v

---

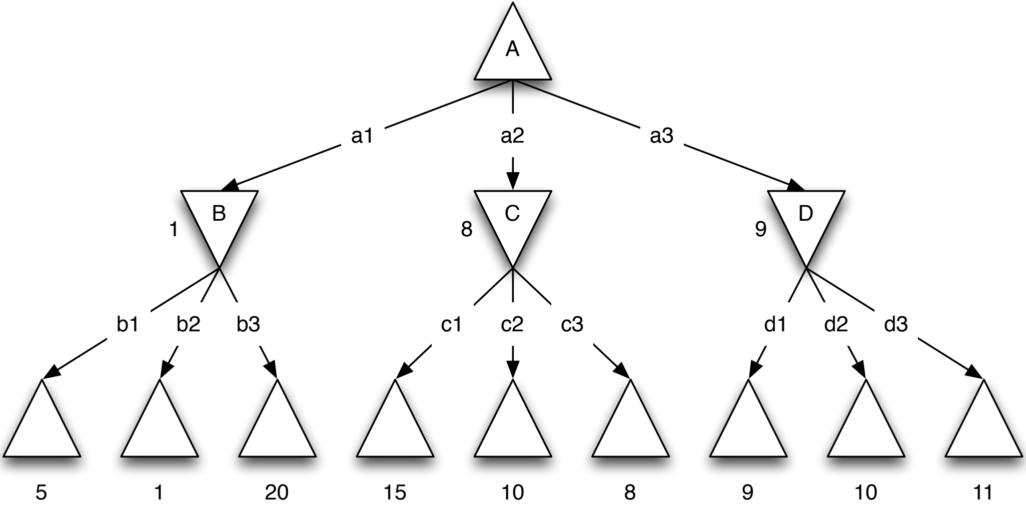
function Min-Value($state$) **returns** $a\ utility\ value$
  if Terminal-Test($state$) **then return** Utility($state$)
  $v \leftarrow \infty$
  for each a **in** $Actions(state)$ **do**
    $v \leftarrow$ Min(v, Max-Value(Result($state, a$)))
  return v



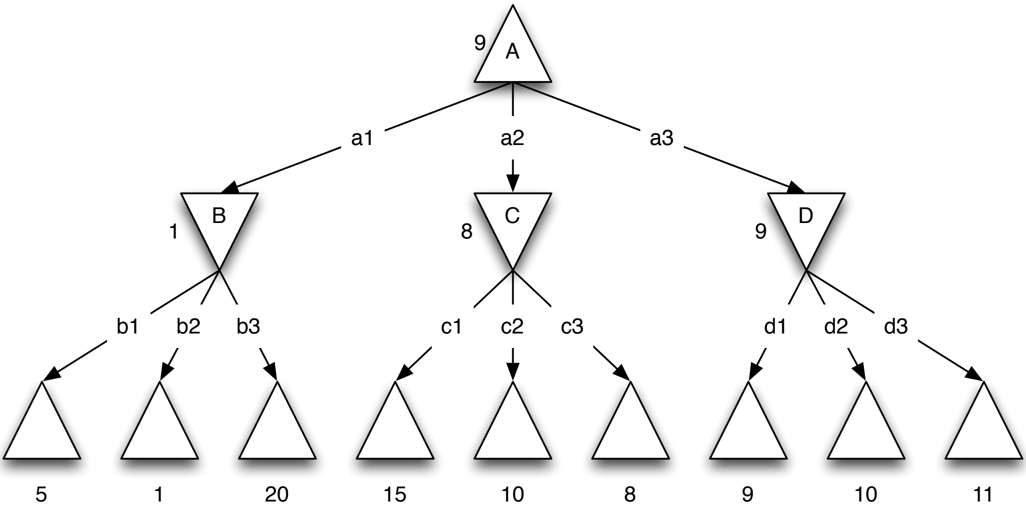
Minimax Value Question



Minimax Value Question



Minimax Value Question



Minimax Algorithm

function Minimax-Decision($state$) **returns** *an action*
  return $\arg\max_{a \in \text{Actions}(s)}$ Min-Value(Result(s, a))

---

function Max-Value($state$) **returns** *a utility value*
  if Terminal-Test($state$) **then return** Utility($state$)
  $v \leftarrow -\infty$
  for each a **in** $Actions(state)$ **do**
    $v \leftarrow$ Max(v, Min-Value(Result($state, a$)))
  return v

---

function Min-Value($state$) **returns** *a utility value*
  if Terminal-Test($state$) **then return** Utility($state$)
  $v \leftarrow \infty$
  for each a **in** $Actions(state)$ **do**
    $v \leftarrow$ Min(v, Max-Value(Result($state, a$)))
  return v

- Time Complexity
- Space Complexity

function Minimax-Decision($state$) **returns** $an\ action$
    return $\arg\max_{a \in \text{Actions}(s)}$ Min-Value(Result(s, a))

---

function Max-Value($state$) **returns** $a\ utility\ value$
    if Terminal-Test($state$) **then return** Utility($state$)
    $v \leftarrow -\infty$
    for each a **in** $Actions(state)$ **do**
        $v \leftarrow$ Max(v, Min-Value(Result($state, a$)))
    return v

---

function Min-Value($state$) **returns** $a\ utility\ value$
    if Terminal-Test($state$) **then return** Utility($state$)
    $v \leftarrow \infty$
    for each a **in** $Actions(state)$ **do**
        $v \leftarrow$ Min(v, Max-Value(Result($state, a$)))
    return v

- Time Complexity
  $O(b^m)$
- Space Complexity

Minimax Algorithm

function Minimax-Decision($state$) **returns** $an\ action$
  return $\arg\max_{a \in \text{Actions}(s)}$ Min-Value(Result(s, a))

---

function Max-Value($state$) **returns** $a\ utility\ value$
  if Terminal-Test($state$) **then return** Utility($state$)
  $v \leftarrow -\infty$
  for each a **in** $Actions(state)$ **do**
    $v \leftarrow$ Max(v, Min-Value(Result($state, a$)))
  return v

---

function Min-Value($state$) **returns** $a\ utility\ value$
  if Terminal-Test($state$) **then return** Utility($state$)
  $v \leftarrow \infty$
  for each a **in** $Actions(state)$ **do**
    $v \leftarrow$ Min(v, Max-Value(Result($state, a$)))
  return v

- Time Complexity
  $O(b^m)$
- Space Complexity
  $O(bm)$

UNIVERSITY OF ABERDEEN

Minimax Algorithm

function Minimax-Decision($state$) **returns** $an\ action$
  return $\arg\max_{a \in \text{Actions}(s)}$ Min-Value(Result(s, a))

---

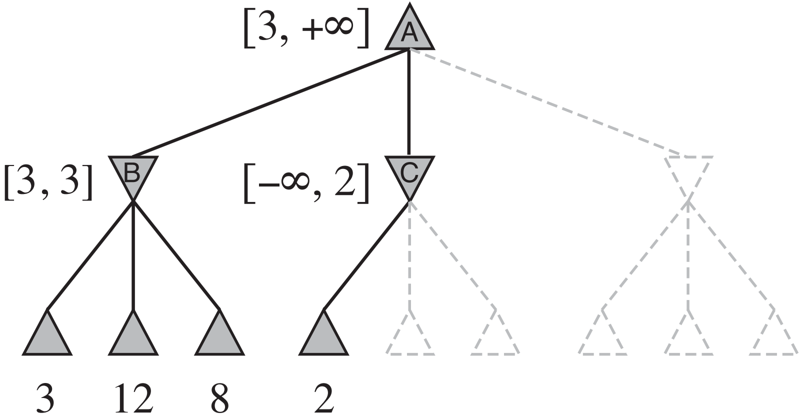
function Max-Value($state$) **returns** $a\ utility\ value$
  if Terminal-Test($state$) **then return** Utility($state$)
  $v \leftarrow -\infty$
  for each a **in** $Actions(state)$ **do**
    $v \leftarrow$ Max(v, Min-Value(Result($state, a$)))
  return v

---

function Min-Value($state$) **returns** $a\ utility\ value$
  if Terminal-Test($state$) **then return** Utility($state$)
  $v \leftarrow \infty$
  for each a **in** $Actions(state)$ **do**
    $v \leftarrow$ Min(v, Max-Value(Result($state, a$)))
  return v

- Time Complexity
  $O(b^m)$
- Space Complexity
  $O(bm)$
- Chess, on average:
  $b = 30$ $m = 40$

- Reducing complexity of b^m
  - Reduce branching factor (b)?
  - Reduce maximum search depth (m)?
  - Searching in a graph rather than a tree?


[3, +∞]
A
[3, 3]
B
[−∞, 2]
C
3
12
8
2

Reducing Branching Factor

- Alpha-Beta Pruning
  - Evaluate which nodes/branches would not affect MIN/MAX's decision
  - Based on keeping track of two parameters:
    - α - value of the best (highest) choice we have in MAX's path
    - β - value of the best (lowest) choice we have in MIN's path
- Updates these values as one goes along the tree

Alpha-Beta Pruning


```
function Minimax-Decision(state) returns an action
    v ← Min-Value(state, −∞, +∞)
    return the action in Actions(state) with value v
```

```
function Max-Value(state) returns a utility value
    if Terminal-Test(state) then return Utility(state)
    v ← −∞
    for each a in Actions(state) do
        v ← Max(v, Min-Value(Result(state, a), α, β))


    return v
```

```
function Min-Value(state) returns a utility value
    if Terminal-Test(state) then return Utility(state)
    v ← ∞
    for each a in Actions(state) do
        v ← Min(v, Max-Value(Result(state, a), α, β))


    return v
```

Alpha-Beta Pruning

```
function Minimax-Decision(state) returns an action
    v ← Min-Value(state, −∞, +∞)
    return the action in Actions(state) with value v
```

```
function Max-Value(state) returns a utility value
    if Terminal-Test(state) then return Utility(state)
    v ← −∞
    for each a in Actions(state) do
        v ← Max(v, Min-Value(Result(state, a), α, β))

        α ← Max(α, v)
    return v
```

```
function Min-Value(state) returns a utility value
    if Terminal-Test(state) then return Utility(state)
    v ← ∞
    for each a in Actions(state) do
        v ← Min(v, Max-Value(Result(state, a), α, β))

        β ← Min(β, v)
    return v
```

function Minimax-Decision($state$) **returns** $an\ action$
  $v \leftarrow$ Min-Value($state, -\infty, +\infty$)
  return the $action$ in Actions($state$) with value v

---

function Max-Value($state$) **returns** $a\ utility\ value$
  if Terminal-Test($state$) **then return** Utility($state$)
  $v \leftarrow -\infty$
  for each a **in** $Actions(state)$ **do**
    $v \leftarrow$ Max(v, Min-Value(Result($state, a$), α, β))
    if $v \geq \beta$ **then return** v
    $\alpha \leftarrow$ Max(α, v)
  return v

---

function Min-Value($state$) **returns** $a\ utility\ value$
  if Terminal-Test($state$) **then return** Utility($state$)
  $v \leftarrow \infty$
  for each a **in** $Actions(state)$ **do**
    $v \leftarrow$ Min(v, Max-Value(Result($state, a$), α, β))
    if $v \leq \alpha$ **then return** v
    $\beta \leftarrow$ Min(β, v)
  return v

Alpha-Beta Pruning

function Minimax-Decision($state$) **returns** *an action*
    $v \leftarrow$ Min-Value($state, -\infty, +\infty$)
    return the $action$ in Actions($state$) with value v

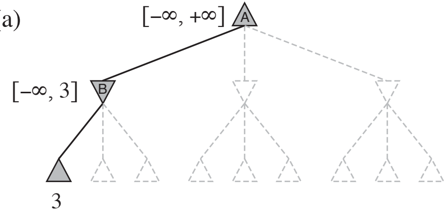
---

function Max-Value($state$) **returns** *a utility value*
    if Terminal-Test($state$) **then return** Utility($state$)
    $v \leftarrow -\infty$
    for each a **in** $Actions(state)$ **do**
        $v \leftarrow$ Max(v, Min-Value(Result($state, a$), α, β))
        if $v \geq \beta$ **then return** v
        $\alpha \leftarrow$ Max(α, v)
    return v

---

function Min-Value($state$) **returns** *a utility value*
    if Terminal-Test($state$) **then return** Utility($state$)
    $v \leftarrow \infty$
    for each a **in** $Actions(state)$ **do**
        $v \leftarrow$ Min(v, Max-Value(Result($state, a$), α, β))
        if $v \leq \alpha$ **then return** v
        $\beta \leftarrow$ Min(β, v)
    return v

(a)



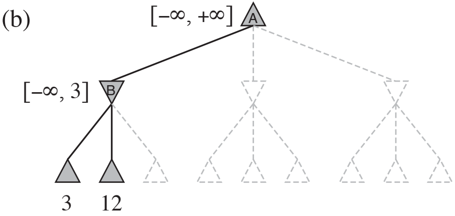
function Minimax-Decision($state$) **returns** $an\ action$
    $v \leftarrow$ Min-Value($state, -\infty, +\infty$)
    return the $action$ in Actions($state$) with value v

---

function Max-Value($state$) **returns** $a\ utility\ value$
    if Terminal-Test($state$) **then return** Utility($state$)
    $v \leftarrow -\infty$
    for each a **in** $Actions(state)$ **do**
        $v \leftarrow$ Max(v, Min-Value(Result($state, a$), α, β))
        if $v \geq \beta$ **then return** v
        $\alpha \leftarrow$ Max(α, v)
    return v

---

function Min-Value($state$) **returns** $a\ utility\ value$
    if Terminal-Test($state$) **then return** Utility($state$)
    $v \leftarrow \infty$
    for each a **in** $Actions(state)$ **do**
        $v \leftarrow$ Min(v, Max-Value(Result($state, a$), α, β))
        if $v \leq \alpha$ **then return** v
        $\beta \leftarrow$ Min(β, v)
    return v

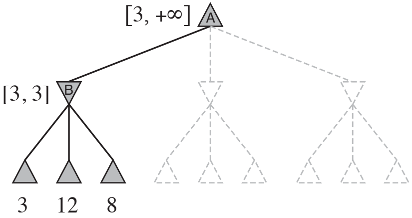


```
function Minimax-Decision(state) returns an action
    v ← Min-Value(state, −∞, +∞)
    return the action in Actions(state) with value v
```

```
function Max-Value(state) returns a utility value
    if Terminal-Test(state) then return Utility(state)
    v ← −∞
    for each a in Actions(state) do
        v ← Max(v, Min-Value(Result(state, a), α, β))
        if v ≥ β then return v
        α ← Max(α, v)
    return v
```

```
function Min-Value(state) returns a utility value
    if Terminal-Test(state) then return Utility(state)
    v ← ∞
    for each a in Actions(state) do
        v ← Min(v, Max-Value(Result(state, a), α, β))
        if v ≤ α then return v
        β ← Min(β, v)
    return v
```

(c)



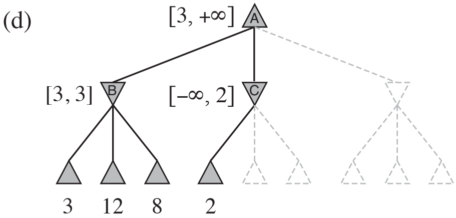
function Minimax-Decision($state$) **returns** $an\ action$
  $v \leftarrow$ Min-Value($state, -\infty, +\infty$)
  return the $action$ in Actions($state$) with value v

---

function Max-Value($state$) **returns** $a\ utility\ value$
  if Terminal-Test($state$) **then return** Utility($state$)
  $v \leftarrow -\infty$
  for each a **in** $Actions(state)$ **do**
    $v \leftarrow$ Max(v, Min-Value(Result($state, a$), α, β))
    if $v \geq \beta$ **then return** v
    $\alpha \leftarrow$ Max(α, v)
  return v

---

function Min-Value($state$) **returns** $a\ utility\ value$
  if Terminal-Test($state$) **then return** Utility($state$)
  $v \leftarrow \infty$
  for each a **in** $Actions(state)$ **do**
    $v \leftarrow$ Min(v, Max-Value(Result($state, a$), α, β))
    if $v \leq \alpha$ **then return** v
    $\beta \leftarrow$ Min(β, v)
  return v

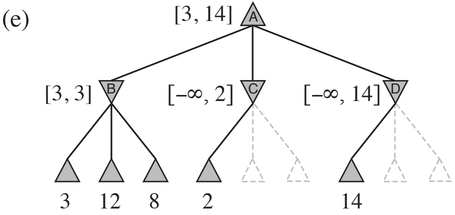


Alpha-Beta Pruning

```
function Minimax-Decision(state) returns an action
    v ← Min-Value(state, −∞, +∞)
    return the action in Actions(state) with value v
```

```
function Max-Value(state) returns a utility value
    if Terminal-Test(state) then return Utility(state)
    v ← −∞
    for each a in Actions(state) do
        v ← Max(v, Min-Value(Result(state, a), α, β))
        if v ≥ β then return v
        α ← Max(α, v)
    return v
```

```
function Min-Value(state) returns a utility value
    if Terminal-Test(state) then return Utility(state)
    v ← ∞
    for each a in Actions(state) do
        v ← Min(v, Max-Value(Result(state, a), α, β))
        if v ≤ α then return v
        β ← Min(β, v)
    return v
```

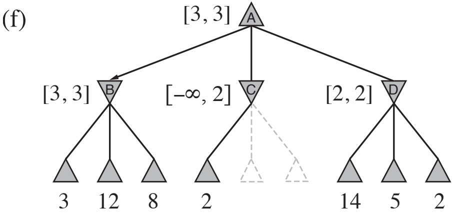


Alpha-Beta Pruning

```
function Minimax-Decision(state) returns an action
    v ← Min-Value(state, −∞, +∞)
    return the action in Actions(state) with value v
```

```
function Max-Value(state) returns a utility value
    if Terminal-Test(state) then return Utility(state)
    v ← −∞
    for each a in Actions(state) do
        v ← Max(v, Min-Value(Result(state, a), α, β))
        if v ≥ β then return v
        α ← Max(α, v)
    return v
```

```
function Min-Value(state) returns a utility value
    if Terminal-Test(state) then return Utility(state)
    v ← ∞
    for each a in Actions(state) do
        v ← Min(v, Max-Value(Result(state, a), α, β))
        if v ≤ α then return v
        β ← Min(β, v)
    return v
```



Alpha-Beta Search: Move Ordering

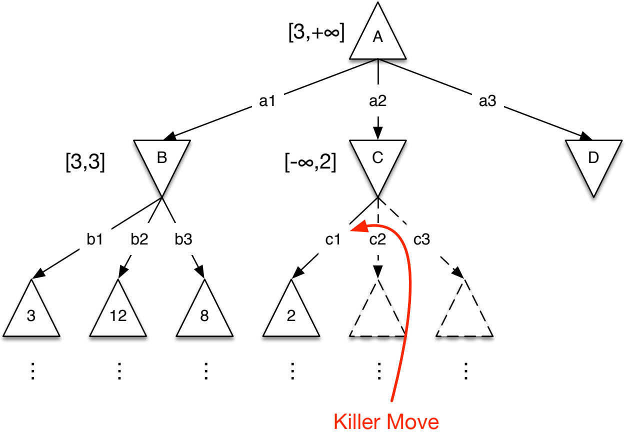
- Pruning is strongly affected by the ordering of the moves in the tree
  - A good ordering*, would enable us to prune many nodes
- Move ordering is often game-dependent knowledge (heuristic)
- Dynamic move-ordering (killer-move heuristic)

Reducing D - Killer Move

- Dynamic heuristic to determine a “good” ordering
- Search two plies ahead until Max (alt. Min) causes a beta (alt. alpha) cutoff
- The move that caused the cutoff is the killer move

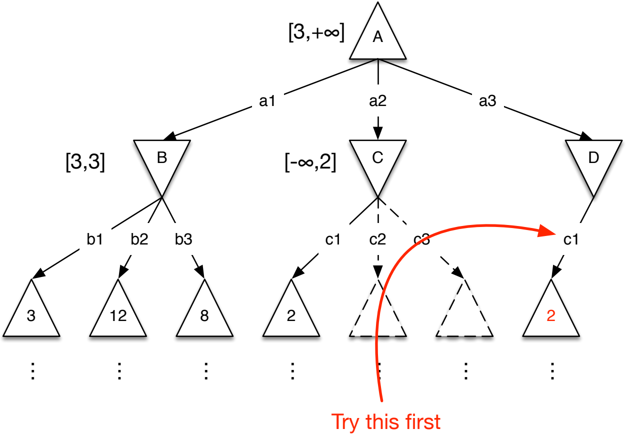
Reducing D - Killer Move

- Dynamic heuristic to determine a “good” ordering
- Search two plies ahead until Max (alt. Min) causes a beta (alt. alpha) cutoff
- The move that caused the cutoff is the killer move



Reducing D - Killer Move

- Dynamic heuristic to determine a “good” ordering
- Search two plies ahead until Max (alt. Min) causes a beta (alt. alpha) cutoff
- The move that caused the cutoff is the killer move



- Establish a cutoff depth and then estimate the value of future paths
  - Estimated value can be a heuristic (domain knowledge)
  - Values can be learned from previous games

$$\text{H-Minimax}(s,d) = \begin{cases} \text{Eval}(s) & \textit{if } \text{Cutoff-Test}(s,d) \\ \max_{a\in Actions(s)} \text{H-Minimax}(Result(s,a), d+1) & \textit{if } \text{Player}(s) = \text{Max} \\ \min_{a\in Actions(s)} \text{H-Minimax}(Result(s,a), d+1) & \textit{if } \text{Player}(s) = \text{Min} \end{cases}$$

- **Weighted linear function** over **features** of a state

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

- Example: Chess
  Current state: pieces, and positions (structure)
- Example: Magic (Card Game)
  Current state: Life Points, Cards in Play and Hand

Reducing M - Eval Function

- Evaluation function can also be learned (i.e. machine learning)
  - At cutoff we have a weighted linear function

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

  - Record the features and actual value V at each evaluation

$$w_1, w_2, \ldots, w_n, V$$

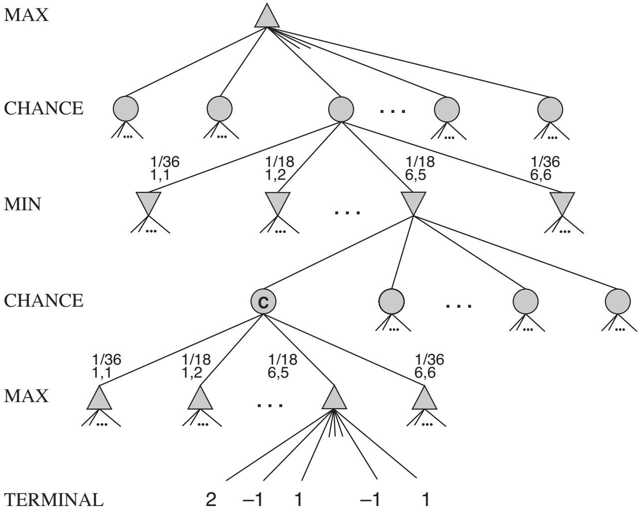
  - Use supervised learning on weights w_i to approximate actual outcome

- As in non-adversarial search, many states will be revisited
- However, only recording visited states is not enough (since MIN can deviate in the future)
- Need to store actual loop paths (memory intensive)
  - Requires “caching” strategy

- Many tabletop games, and most simulated games include an element of chance
  - e.g., Backgammon
- Outcome of agent choices is not deterministic
  - Games must take into account multiple outcomes for the player
- Solution: weight outcomes by their probability
  - Expected value

Expectiminimax

Instead of the deterministic utilities, account for stochastic utilities



$$
\text{ExpectMinimax}(s,d) = \begin{cases} \text{Utility}(s) & \textit{if } \text{Terminal-Test}(s,d) \\ \max_{a \in Actions(s)} \text{ExpectMinimax}(Result(s,a)) & \textit{if } \text{Player}(s) = \text{Max} \\ \min_{a \in Actions(s)} \text{ExpectMinimax}(Result(s,a)) & \textit{if } \text{Player}(s) = \text{Min} \\ \sum_r \mathbb{P}[r] \, \text{ExpectMinimax}(Result(s,a)) & \textit{if } \text{Player}(s) = \text{Chance} \end{cases}
$$

- Since the outcomes of all actions are stochastic, instead of computing all expected utilities, we can sample game outcomes
  - Simulate games but choose **one random outcome** per player choice
  - Monte Carlo Rollout

Adversarial Search Summary

- We've seen how to model games:
  - Building a game-tree
  - Finding the optimal moves in fully observable settings
- There are many other types of games
  - Multiplayer games
  - Stochastic Games
  - Partially-observable games

Any Questions.