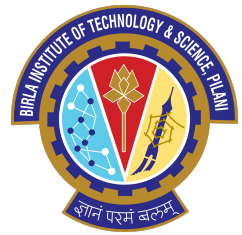
IS-ZC444: ARTIFICIAL INTELLIGENCE

Lecture-08: Adversarial Search, Constraint Satisfaction



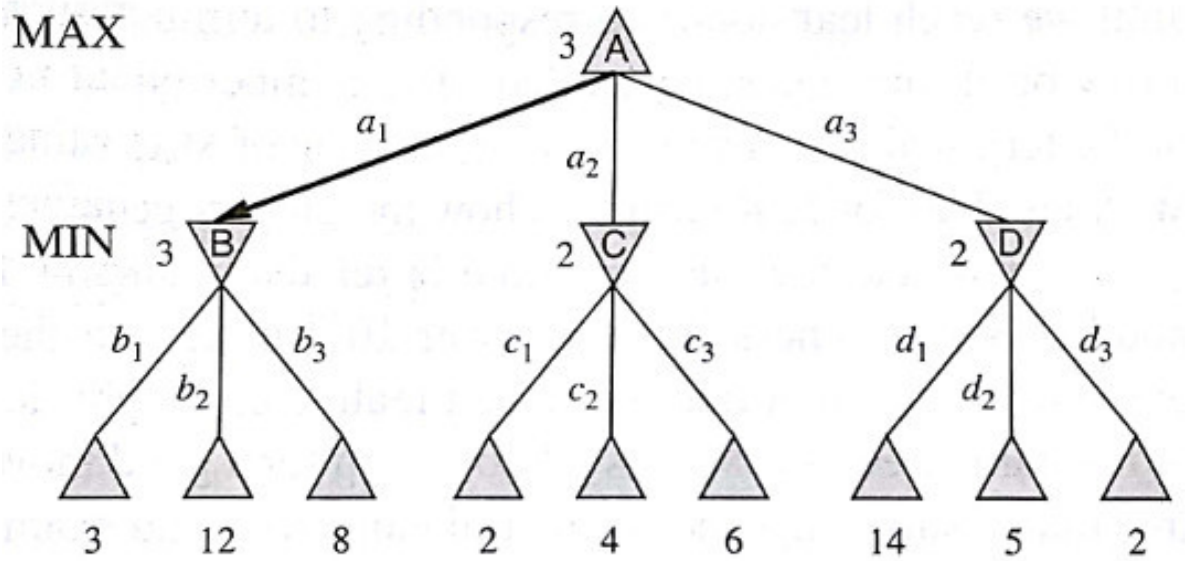
Dr. Kamlesh Tiwari
Assistant Professor
Department of Computer Science and Information Systems,
BITS Pilani, Pilani, Jhunjhunu-333031, Rajasthan, INDIA

September 26, 2018 (WILP @ BITS-Pilani Jul-Nov 2018)

Recap:

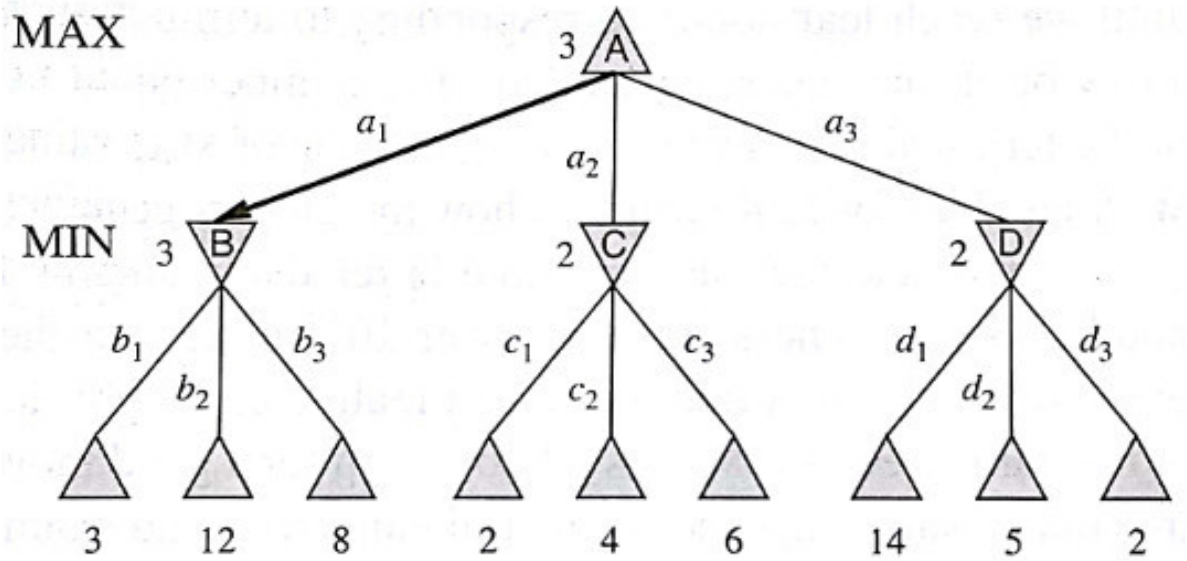
- In continuous state space: branching factor (number of next states) becomes infinite use gradients to move
- With nondeterministic actions: when it is not sure what would be the next state, AND-OR Search is useful
- Partial observations: when percepts do not suffice to pin down the exact state, belief states could be used to navigate
- Online search and unknown environment: agent interleaves computation and action
- Adversarial search (game): when two agents MIN and MAX have conflicting goals in competitive multiagent environment
  - Deterministic, fully-observable, turn-taking, two-player, zero-sum
  - Initial state S_0, it is PLAYER(s)'s tern, he can take any action from ACTIONS(s), result of the move be RESULT(s, a) sate, if the test TERMINAL_TEST(s) gives true when game is over, and reward is UTILITY(s, p)
  - MAX must find a contingent **strategy**

Recap: Two half moves is one **ply**



[1] utility value for MAX of being in corresponding state (assuming then onwards both player play optimally)

Recap: Two half moves is one **ply**

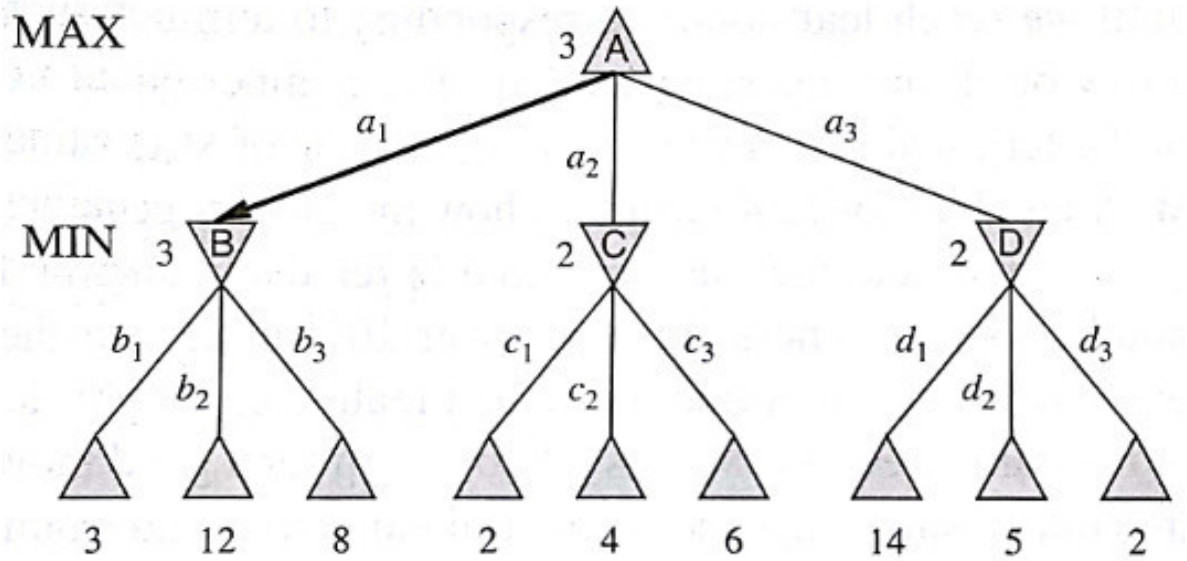


Given the game tree, optimal strategy can be determined from **minimax value** of each node.

$$MINIMAX(s) = \begin{cases} UTILITY(s) & \text{if } TERMINAL\_TEST(s) \\ \max_{a \in Actions(s)} MINIMAX(RESULT(s,a)) & \text{if } PLAYER(s) = MAX \\ \min_{a \in Actions(s)} MINIMAX(RESULT(s,a)) & \text{if } PLAYER(s) = MIN \end{cases}$$

[1] utility value for MAX of being in corresponding state (assuming then onwards both player play optimally)

Recap: Two half moves is one **ply**



Given the game tree, optimal strategy can be determined from **minimax value** of each node.

$$MINIMAX(s) = \begin{cases} UTILITY(s) & \text{if } TERMINAL\_TEST(s) \\ \underset{a \in Actions(s)}{argmax}\, MINIMAX(RESULT(s,a)) & \text{if } PLAYER(s) = MAX \\ \underset{a \in Actions(s)}{argmin}\, MINIMAX(RESULT(s,a)) & \text{if } PLAYER(s) = MIN \end{cases}$$

Action a_1 is the optimal choice [1]

(essentially optimizing worst-case outcome for MAX)

[1] utility value for MAX of being in corresponding state (assuming then onwards both player play optimally)

Recap: MINIMAX Algorithm

Returns the action corresponding to best move

```
function MINIMAX-DECISION(state) returns an action
  return arg max_{a ∈ ACTIONS(s)} MIN-VALUE(RESULT(state, a))

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(s, a)))
  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(s, a)))
  return v
```

Recursion proceeds all the way down to the leaves.

Recap: MINIMAX Algorithm

Returns the action corresponding to best move

```
function MINIMAX-DECISION(state) returns an action
  return arg max_{a ∈ ACTIONS(s)} MIN-VALUE(RESULT(state, a))

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(s, a)))
  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(s, a)))
  return v
```

Recursion proceeds all the way down to the leaves.Time complexity $O(b^m)$ that is impractical but provides a basis of solution.

ALPHA-BETA Pruning

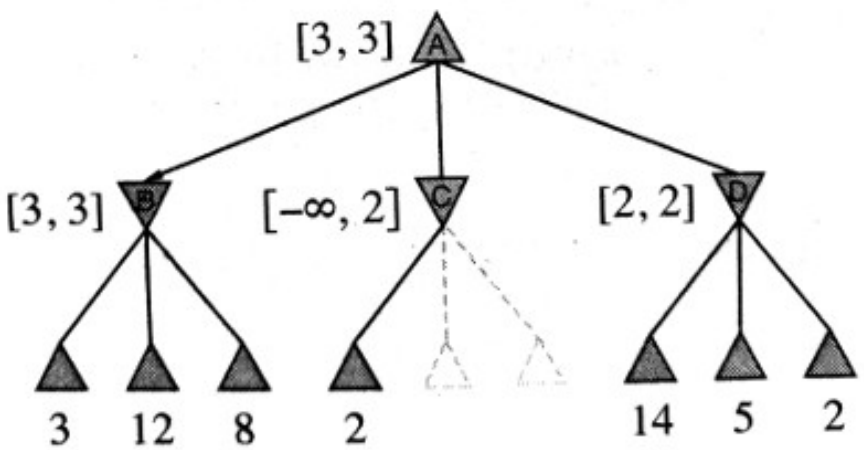
- Number of nodes to examine in minimax search is exponential in the depth of tree $O(b^m)$.
- Sometime we can make it $O(b^{m/2})$ using **alpha-beta pruning**

ALPHA-BETA Pruning

- Number of nodes to examine in minimax search is exponential in the depth of tree $O(b^m)$.
- Sometime we can make it $O(b^{m/2})$ using **alpha-beta pruning**
- When applied to standard minimax tree, it returns the same move as minimax but, prunes away branches that cannot possibly influence the decision.

ALPHA-BETA Pruning

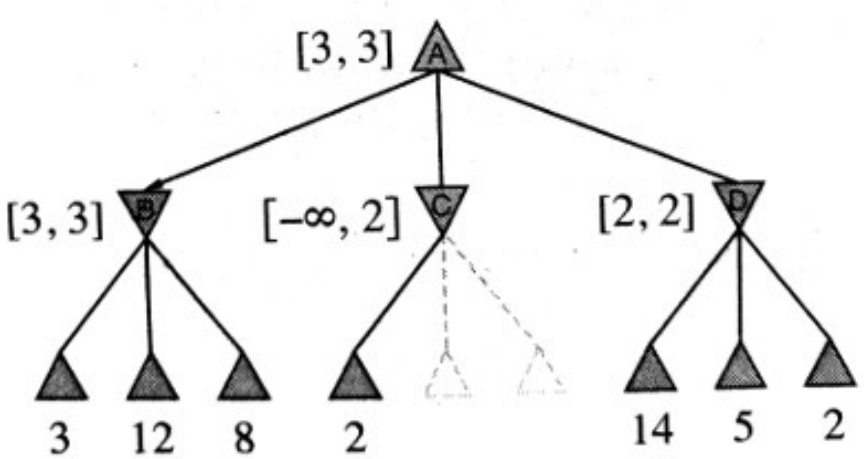
- Number of nodes to examine in minimax search is exponential in the depth of tree $O(b^m)$.
- Sometime we can make it $O(b^{m/2})$ using **alpha-beta pruning**
- When applied to standard minimax tree, it returns the same move as minimax but, prunes away branches that cannot possibly influence the decision.



Consider two unevaluated successor of node C have value x and y

ALPHA-BETA Pruning

- Number of nodes to examine in minimax search is exponential in the depth of tree $O(b^m)$.
- Sometime we can make it $O(b^{m/2})$ using **alpha-beta pruning**
- When applied to standard minimax tree, it returns the same move as minimax but, prunes away branches that cannot possibly influence the decision.

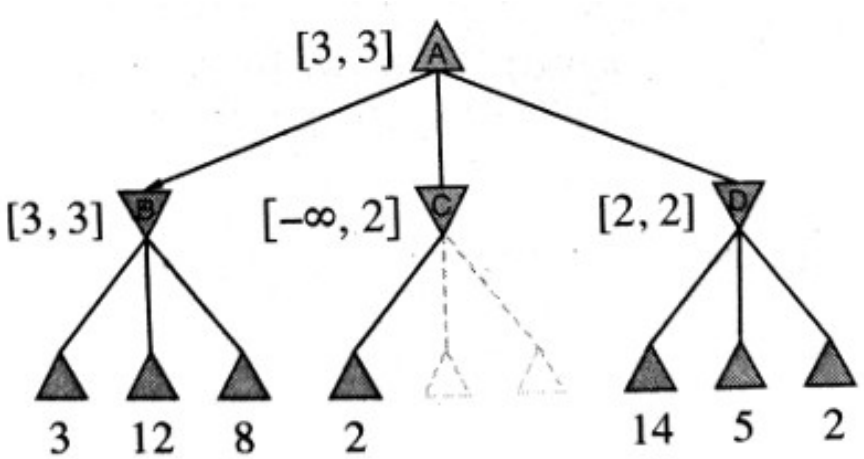


Consider two unevaluated successor of node C have value x and y

MINIMAX(root)
= max(min(3,12,8), min(2,x,y), min(14,5,2))

ALPHA-BETA Pruning

- Number of nodes to examine in minimax search is exponential in the depth of tree $O(b^m)$.
- Sometime we can make it $O(b^{m/2})$ using **alpha-beta pruning**
- When applied to standard minimax tree, it returns the same move as minimax but, prunes away branches that cannot possibly influence the decision.

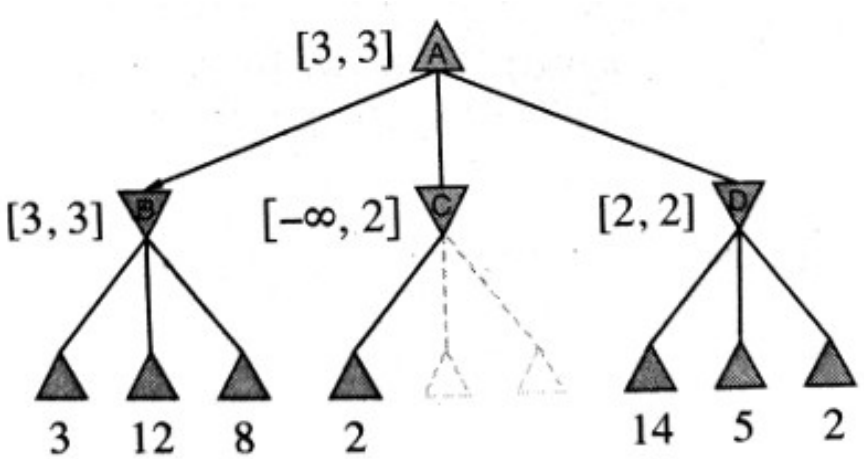


Consider two unevaluated successor of node C have value x and y

MINIMAX(root)
= max(min(3,12,8), min(2,x,y), min(14,5,2))
= max(3, min(2,x,y), 2)

ALPHA-BETA Pruning

- Number of nodes to examine in minimax search is exponential in the depth of tree $O(b^m)$.
- Sometime we can make it $O(b^{m/2})$ using **alpha-beta pruning**
- When applied to standard minimax tree, it returns the same move as minimax but, prunes away branches that cannot possibly influence the decision.

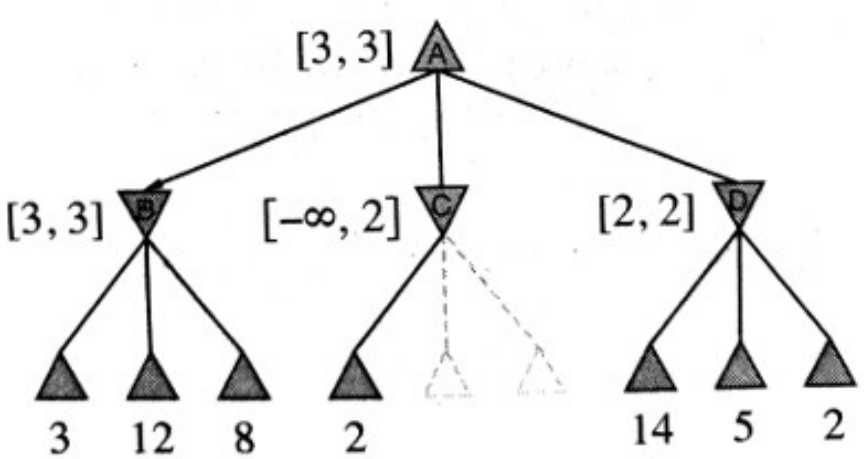


Consider two unevaluated successor of node C have value x and y

MINIMAX(root)
= max(min(3,12,8), min(2,x,y), min(14,5,2))
= max(3, min(2,x,y), 2)
= max(3, z, 2) where z=min(2,x,y)$\leq$ 2

ALPHA-BETA Pruning

- Number of nodes to examine in minimax search is exponential in the depth of tree $O(b^m)$.
- Sometime we can make it $O(b^{m/2})$ using **alpha-beta pruning**
- When applied to standard minimax tree, it returns the same move as minimax but, prunes away branches that cannot possibly influence the decision.

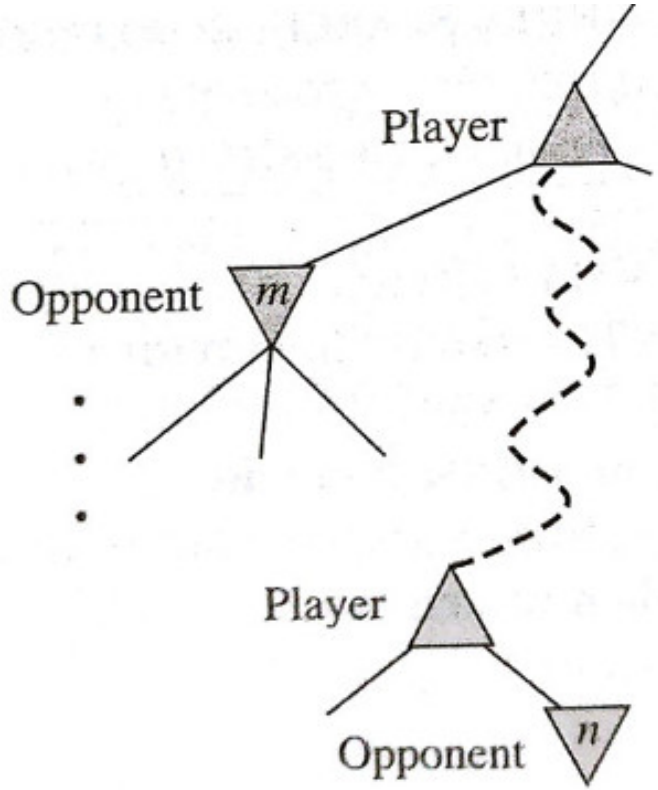


Consider two unevaluated successor of node C have value x and y

MINIMAX(root)
= max(min(3,12,8), min(2,x,y), min(14,5,2))
= max(3, min(2,x,y), 2)
= max(3, z, 2) where z=min(2,x,y)$\leq$ 2
= 3

ALPHA-BETA Pruning

- Alpha-beta pruning can be applied to trees of any depth, and it is often possible to prune entire subtree rather than just leaves.



If *m* is better than *n* for player then we would never go to *n* in play

α	$=$	value of best choice (highest) found so far for MAX
β	$=$	value of best choice (lowest) found so far for MIN

ALPHA-BETA Pruning

function ALPHA-BETA-SEARCH(*state*) **returns** an action
  $v \leftarrow$ MAX-VALUE($state, -\infty, +\infty$)
  return the *action* in ACTIONS(*state*) with value v

---

function MAX-VALUE($state, \alpha, \beta$) **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(s,a), α, β))
    if $v \geq \beta$ **then return** v
    $\alpha \leftarrow$ MAX(α, v)
  return v

---

function MIN-VALUE($state, \alpha, \beta$) **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(s,a) , α, β))
    if $v \leq \alpha$ **then return** v
    $\beta \leftarrow$ MIN(β, v)
  return v

ALPHA-BETA Pruning

function ALPHA-BETA-SEARCH(*state*) **returns** an action
  $v \leftarrow$ MAX-VALUE(*state*, $-\infty, +\infty$)
  return the *action* 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 each 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*)
  $v \leftarrow +\infty$
  for each a **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if $v \leq \alpha$ **then return** v
    $\beta \leftarrow$ MIN(β, v)
  return v

Order matters. So, examine likely to be best successor first.

ALPHA-BETA Pruning

function ALPHA-BETA-SEARCH(*state*) **returns** an action
  $v \leftarrow$ MAX-VALUE(*state*, $-\infty, +\infty$)
  return the *action* 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 each 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*)
  $v \leftarrow +\infty$
  for each a **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if $v \leq \alpha$ **then return** v
    $\beta \leftarrow$ MIN(β, v)
  return v

Order matters. So, examine likely to be best successor first.

Is it possible?

ALPHA-BETA Pruning

function ALPHA-BETA-SEARCH(*state*) **returns** an action
  $v \leftarrow$ MAX-VALUE($state, -\infty, +\infty$)
  return the *action* in ACTIONS(*state*) with value v

---

function MAX-VALUE($state, \alpha, \beta$) **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(s,a), α, β))
    if $v \geq \beta$ **then return** v
    $\alpha \leftarrow$ MAX(α, v)
  return v

---

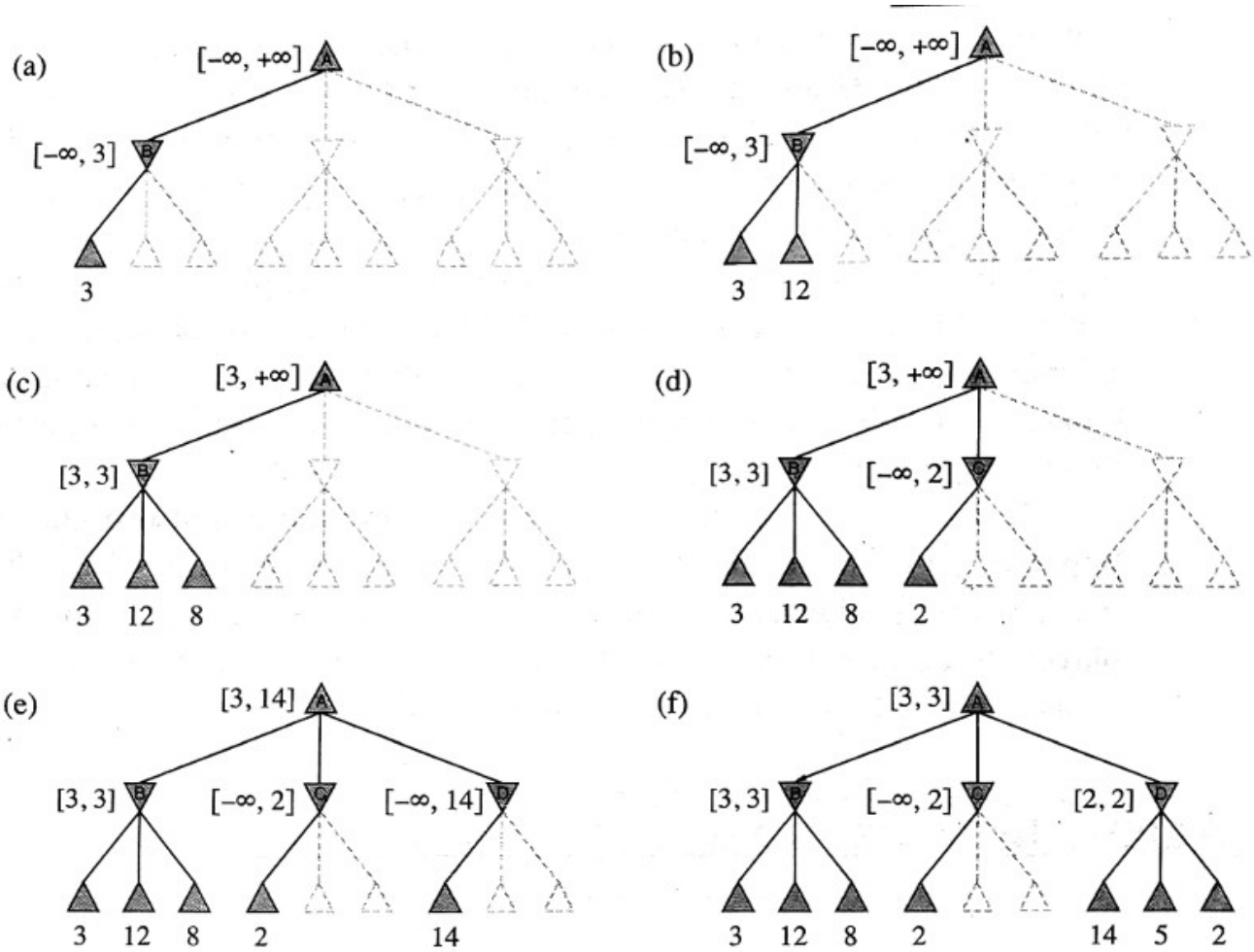
function MIN-VALUE($state, \alpha, \beta$) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow +\infty$
  for each a **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if $v \leq \alpha$ **then return** v
    $\beta \leftarrow$ MIN(β, v)
  return v

Order matters. So, examine likely to be best successor first.

Is it possible?

No (*perfectly*)

In-action: ALPHA-BETA Pruning



Move Ordering

- With perfect ordering we need to examine only $O(b^{m/2})$ nodes
- When successors are examined in random order it needs to examine $O(b^{3m/4})$ nodes
- In chess, a strategy that capture $\rightarrow$ threat $\rightarrow$ forward $\rightarrow$ backward, gets you to within about a factor of 2 of the best case $O(b^{m/2})$
- Try first the move that were found useful in past (killer move)
- Iterative deepening could help (adds constant fraction time)
- Hash table of previously seen positions (**transposition table**) can restrict re-computation of states

Imperfect Real-time Decision

- Generating entire search space is overkill
- Covering till large depth is not possible
- Idea is to cutoff the search earlier

$$H\text{-}MINIMAX(s,d) = \begin{cases} Eval(s) & \text{if } CUTOFF\_TEST(s,d) \\ argmax_{a \in Actions(s)} H\text{-}MINIMAX(RESULT(s,a), d+1) & \text{if } PLAYER(s) = MAX \\ argmin_{a \in Actions(s)} H\text{-}MINIMAX(RESULT(s,a), d+1) & \text{if } PLAYER(s) = MIN \end{cases}$$

Evaluation Function

- Returns the estimate of expected utility
- Performance would deeply depend on it
- Eval(win) $\geq$ Eval(draw) $\geq$ Eval(lose)
- Computation should be quick
- Value should relate to actual chance of winning
- Can use features (Number of pawns, Number of queens, ...)
- Can have various categories (all pawn vs one pawn, etc) Suppose experience suggests 72% of states encountered in the two-pawn vs. one pawn category lead to win, 20% lose, and 8% in draw. then eval could be

$$0.72 \times 1 + 0.20 \times 0 + 0.08 \times 0.5 = 0.76$$

- **Weighted linear function** are also possible

$$Eval(s) = w_1 \times f_1(s) + w_2 \times f_2(s) + w_3 \times f_3(s) + ...$$

Important Considerations

- **Cutting off search**: having fixed depth may not be a good idea. **Quiescent** states are unlikely to have wild swing so expand till it. **Horizon effect** is difficult to eliminate so **singular extension**[2] could be used
- **Forward Pruning**: cutting off the some search without further consideration. (we do not evaluate all possibilities **beam search**, we only use some in our mind) This may be fatal.
- **Search Vs Lookup** We have some policy on how to start and finish well. Use them as lookup

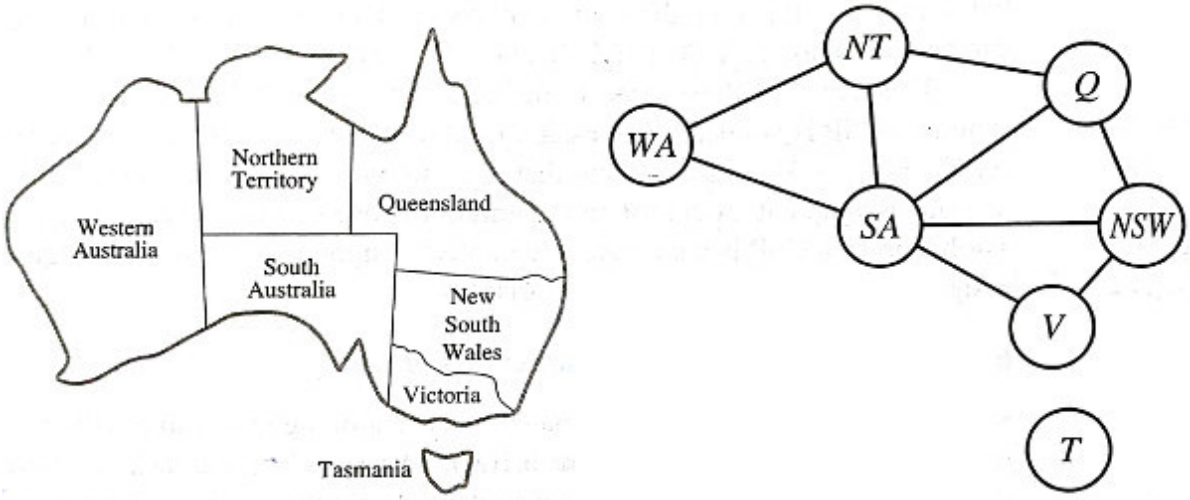
[2] can I take a previous good move

Constraint Satisfaction

- A Constraint Satisfaction Problem (CSP) has three components
  1. X as a set of variables $\{X_1, X_2, X_3, ..., X_n\}$
  2. D as a set of domains $\{D_1, D_2, D_3, ..., D_n\}$
  3. C set of constraints that specify allowable combinations of values
- Each D_i contains allowable set of values $\{v_1, v_2, v_3, ..., v_k\}$ for X_i
- Each C_i contains a pair <scope, relation>
  such as $<(X_1, X_3), X_1 \neq X_3>$
- To solve CSP it needs state space and a notion of solution
- An assignment of variables such as $(X_1 = v_1, X_2 = v_2, ..., X_n = v_n)$ that does not violates any constraints is called **consistent** or legal solution.
- Our target to have complete assignment that is consistent.

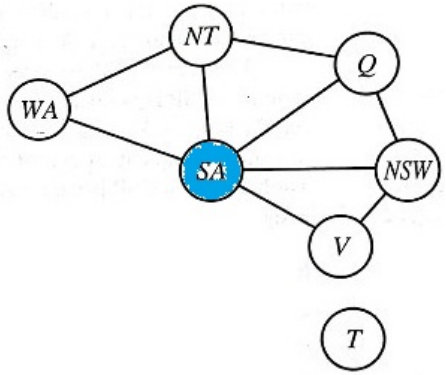
Example: Map Coloring

Given three colors {red,green,blue}, can you color following map such that no two neighboring region have same color.



- X = {WA, NT, Q, NSW, V, SA, T}
- C = {SA≠WA, SA≠NT, SA≠Q, SA≠NSW, SA≠V, WA≠NT, NT≠Q, Q≠NSW, NSW≠V}
- There are many solution to the problem
- It is helpful to visualize as a **constraint graph** (node: variable, edge: participate in constraint)

Constraint Propagation

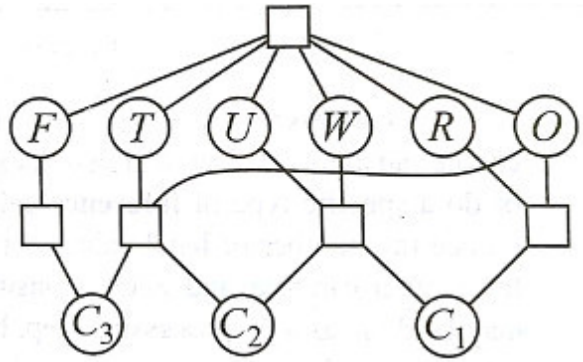


- Once you choose (SA=blue)
- you can conclude that none of its five members could take the color blue
- Without taking advantage of constraint propagation, search needs to consider $3^5 = 243$ assignments for the five neighbors.
- With constraint propagation, it needs $2^5 = 32$ only
- 87% reduction

Example: Cryptarithmethc Puzzle

Consider following addition (we have to find digits)

$$\begin{array}{r} T\ W\ O \\ +\ T\ W\ O \\ \hline F\ O\ U\ R \end{array}$$



- Each letter represents a different digit
- We need Alldiff{F,T,U,W,R,O}
- Additional constraints are

$$\begin{aligned} O + O &= R + 10 \times C_{10} \\ C_{10} + W + W &= U + 10 \times C_{100} \\ C_{100} + T + T &= O + 10 \times C_{1000} \\ C_{1000} &= F \end{aligned} \tag{1}$$

Thank You!

Thank you very much for your attention!

Queries ?

(Reference[3])

[3] 1) Book - *AIMA*, ch-05+06, Russell and Norvig.