Artificial Intelligence
Unit 2-2 Adversarial Search Strategies

2020-2021 Odd BE CSE VII semester

Engels. R

Unit 2 – Search Strategies

- **SEARCH STRATEGIES** (12)
  - Breadth-First Search
  - Uniform Cost Search
  - Depth-First Search
  - Depth-Limited Search
  - Iterative Deepening Search
  - Bidirectional Search
- **Heuristic Search Techniques**
  - A* Search
  - AO* Algorithm
- **Adversarial Search:**
  - Minimax Algorithm
  - Alpha beta Pruning

Adversarial Search

- Competitive Environments with multiple agents
  - Multi-agent environment
- Agents' goals are in conflict, giving rise to **adversarial search** problems
  - AKA Games
- In Mathematical **game theory**, any multiagent environment is a game
  - provided that the impact of each agent on the others is "significant,"
  - Regardless of whether the agents are cooperative or competitive

AI Games

- Deterministic, fully observable environments
- two agents act alternately
- Utility values at the end of the game are always equal and opposite
  - Such as one player wins ➔ other player loses
- Why games?
  - they are too hard to solve
  - Chess has an average branching factor of about 35 & 50 moves average per player
    - 10^{154} nodes in search space
  - Games, like the real world, require the ability to make *some* decision ***even when calculating the optimal decision is infeasible***
  - Games penalize inefficiency severely

AI game approach

- **Heuristic evaluation functions**
  - to approximate the true utility of a state without doing a complete search
- **Imperfect information**
  - Such as not all cards visible to one player
- **Pruning**
  - to ignore portions of the search tree that make no difference to the final choice
- **Environment**
  - Two players, MAX and MIN
  - MAX moves first, and then they take turns moving until the game is over
  - At the end of the game,
    - points are awarded to the winning player and
    - penalties are given to the loser

Game: formal definition

- A game is defined as a kind of search problem with the following elements

1. S_0: The **initial state**, specifies how the game is set up at the start
2. PLAYER(s): Defines which player has the move in a state
3. ACTIONS(s): Returns the set of legal moves in a state
4. RESULT(s, a): The **transition model**, which defines the result of a move
5. TERMINAL-TEST(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**.
6. UTILITY(s, p): A **utility function** (AKA **objective** or **payoff** function)
    1. defines the final numeric value for a game that ends in terminal state ***s*** for a player ***p***

More on Utility Function

- In chess, the outcome is a win, loss, or draw, with values +1, 0, or 1/2
- Some games have a wider variety of possible outcomes
  - the payoffs in backgammon range from 0 to +192
- A **zero-sum game** is (confusingly) defined as one where the total payoff to all players is the same for every instance of the game
- Chess is zero-sum because every game has payoff of either
  - 0 + 1,
  - 1 + 0 or
  - ½ + ½
- Zero-sum is traditional and makes sense if you imagine each player is charged an entry fee of ½

State Space / Game tree

- The initial state, ACTIONS function, and RESULT function define the **game tree** for the game
  - the nodes are game states and
  - the edges are moves

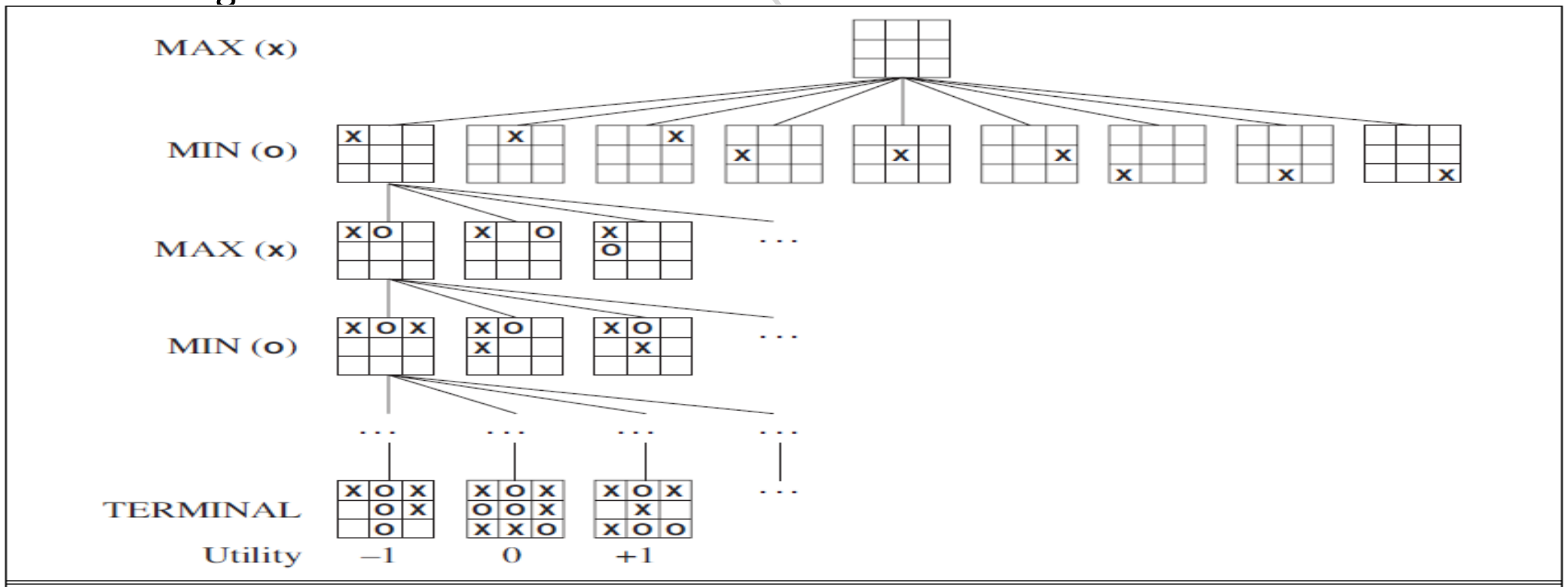


Figure 5.1 A (partial) game tree for the game of tic-tac-toe. The top node is the initial state, and MAX moves first, placing an X in an empty square. We show part of the tree, giving alternating moves by MIN (O) and MAX (X), until we eventually reach terminal states, which can be assigned utilities according to the rules of the game.

State Space / Game Tree/ Search Tree

- In Tic-Tac-Toe
  - From the initial state, MAX has nine possible moves
  - Play alternates between MAX's placing an X and MIN's placing an O
  - until we reach leaf nodes corresponding to terminal states
    - one player has three in a row or all the squares are filled
- The number on each leaf node
  - Indicates utility value of terminal state from the point of view of MAX
  - High values are assumed to be good for MAX and bad for MIN
  - (which is how the players get their names)
- Regardless of the size of the game tree, MAX's job is to search for a good move
- **Search tree**
  - A tree that is superimposed on the full game tree, and
  - examines enough nodes to allow a player to determine what move to make

State Space / Game Tree/ Search Tree

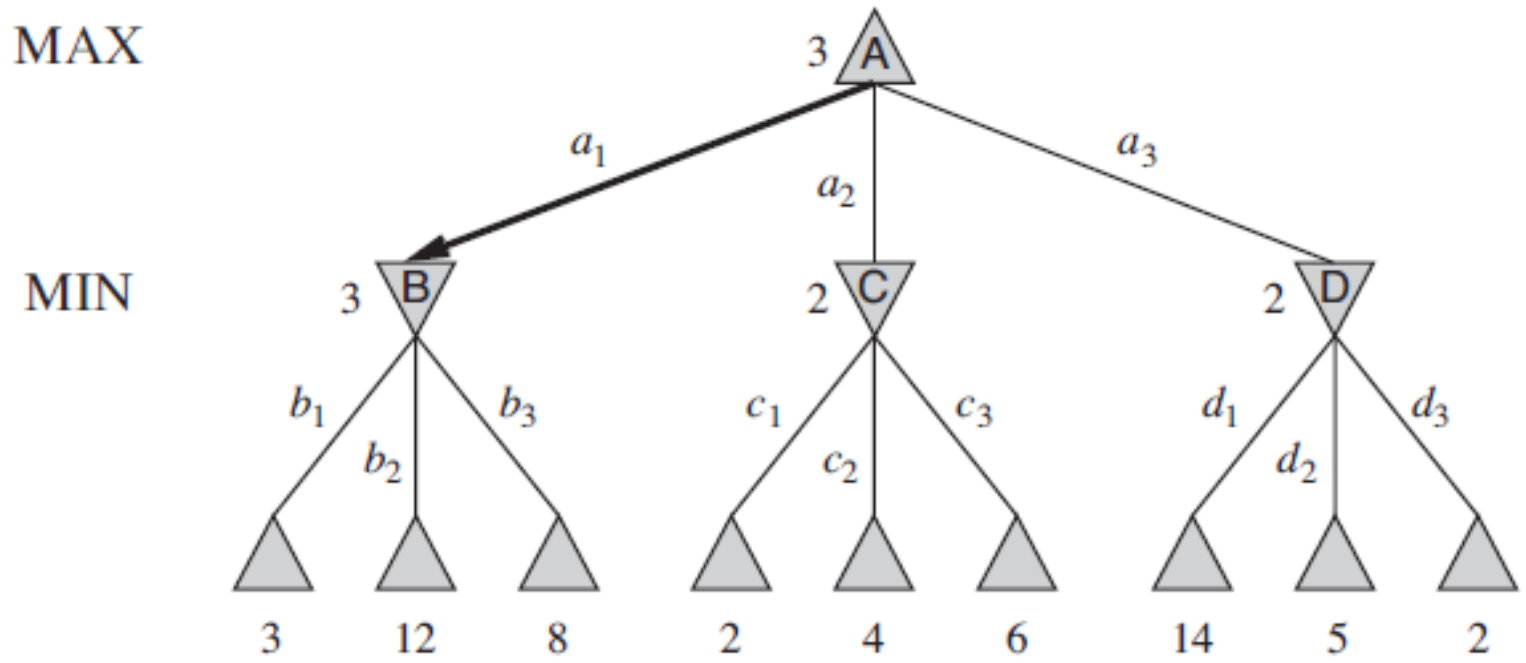


Figure 5.2 A two-ply game tree. The △ nodes are "MAX nodes," in which it is MAX's turn to move, and the ▽ nodes are "MIN nodes." The terminal nodes show the utility values for MAX; the other nodes are labeled with their minimax values. MAX's best move at the root is a_1, because it leads to the state with the highest minimax value, and MIN's best reply is b_1, because it leads to the state with the lowest minimax value.

Role of MIN and MAX in adversarial search

- In normal search problem, the optimal solution would be a sequence of actions leading to a goal state
  - In adversarial search, MIN has definitive impact
- MAX must find a contingent **strategy**, which specifies
  - MAX's move in the initial state,
  - then MAX's moves in the states resulting from every possible response by MIN,
  - then MAX's moves in the states resulting from every possible response by MIN to those moves, and so on
- This is analogous to the AND–OR search algorithm
  - MAX playing the role of OR and MIN equivalent to AND
- Roughly speaking***, an optimal strategy leads to outcomes at least as good as any other strategy***
  - when one is playing an infallible opponent

Role of MIN and MAX in adversarial search

- Optimal strategy can be determined from the **minimax value** of each node (MINIMAX(n))
  - Given a game tree
- The minimax value of a node is the utility (for MAX) of being in the corresponding state to the end of the game
  - *assuming that both players play optimally*
- Minimax value of a terminal state is just its utility
- Given a choice, MAX prefers to move to a state of maximum value,
  - whereas MIN prefers a state of minimum value

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

Minimax approach

- Definition of optimal play for MAX
  - assumes that MIN also plays optimally
  - maximizes the worst-case outcome for MAX
- Other strategies against suboptimal opponents may do better than the minimax strategy,
  - but do worse against optimal opponents
- **Utility function**: the function applied to leaf nodes
- **Backed-up value**
  - of a **max-position**: the value of its largest successor
  - of a **min-position**: the value of its smallest successor
- The **minimax algorithm** computes the minimax decision from the current state
  - Uses a simple **recursive computation of the minimax values** of each **successor** state,
  - Recursion proceeds all the way down to the leaves of the tree, and
  - then the minimax values are **backed up** through the tree as the recursion unwinds

Minimax Algorithm

```
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
```

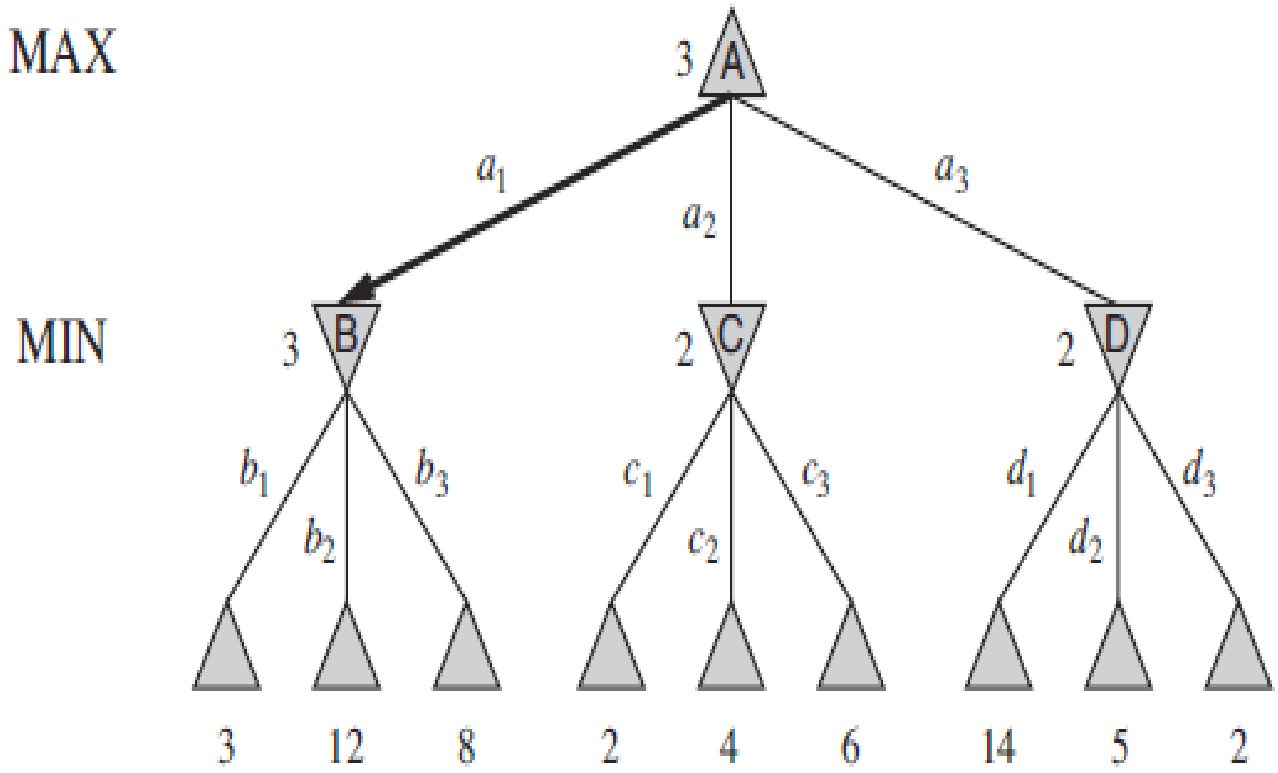
Minimax is an algorithm for calculating minimax decisions, designed to consider opponent as an optimal player

- It returns the action corresponding to the **best possible move**
  - the move that leads to the outcome with the **best utility**
  - under the assumption that the opponent plays to minimize utility
- The functions MAX-VALUE and MIN-VALUE go through the whole game tree
  - all the way to the leaves
  - to determine the backed-up value of a state

$\text{argmax}_{a \in S} f(a)$ computes the element a of set S that has the maximum value of f(a)

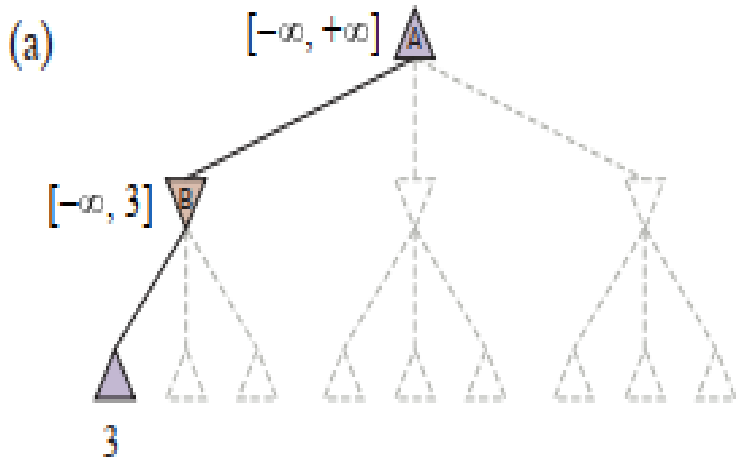
Role of MIN and MAX in adversarial search

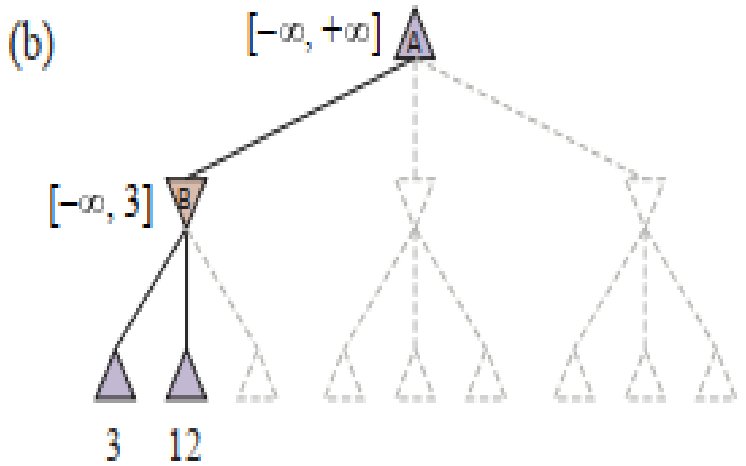
- The terminal nodes on the bottom level get their utility values from the game's UTILITY function
  - MAX nodes (MAX play turns) are labeled a1, a2…: MIN nodes b1, b2…
- The first MIN node, labelled B,
  - has three successor states with values 3, 12, and 8
  - Its minimax value is 3
  - Other two MIN nodes have minimax value 2
- The root node is a MAX node
  - Successor states have minimax values 3, 2, and 2;
  - so it has a minimax value of 3.
- We can also identify the **minimax decision** at the root
- Action **a1** is the optimal choice for MAX because it leads to the state with the highest minimax value

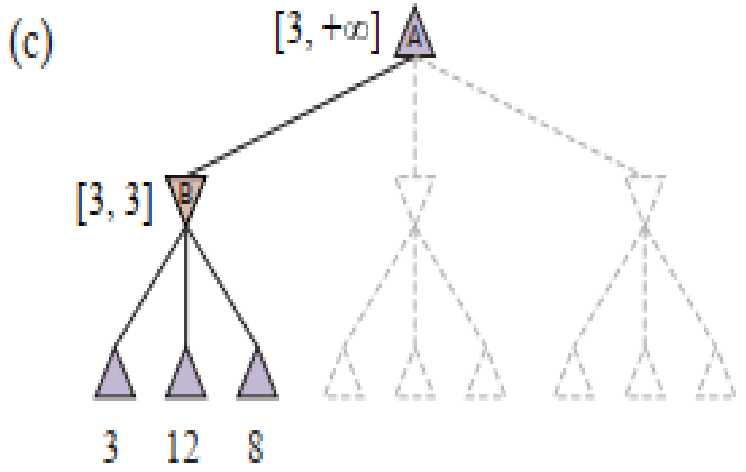


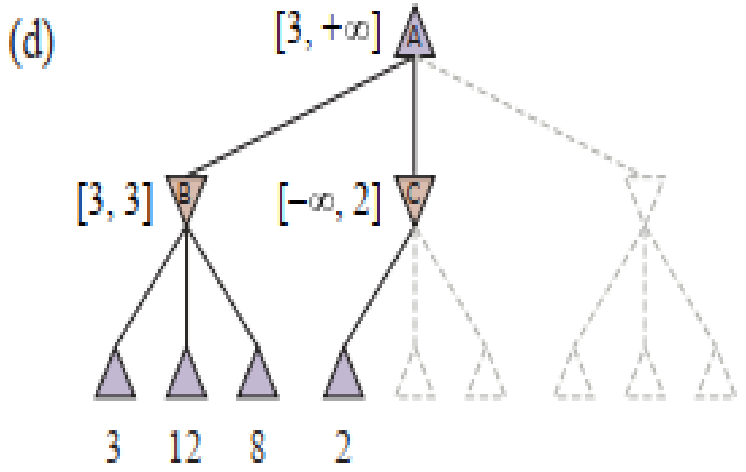
Role of MIN and MAX in adversarial search

- Stages in the calculation of the optimal decision for the game tree in Figure
  - At each point, we show the range of possible values for each node
- (a) The first leaf below *B* has the value 3. Hence, *B*, which is a MIN node, has a value of *at most* 3.
- b) The second leaf below *B* has a value of 12; MIN would avoid this move, so the value of *B* is still at most 3
- (c) The third leaf below *B* has a value of 8; we have seen all *B*'s successor states, so the value of *B* is exactly 3. Now we can infer that the value of the root is *at least* 3, because MAX has a choice worth 3 at the root



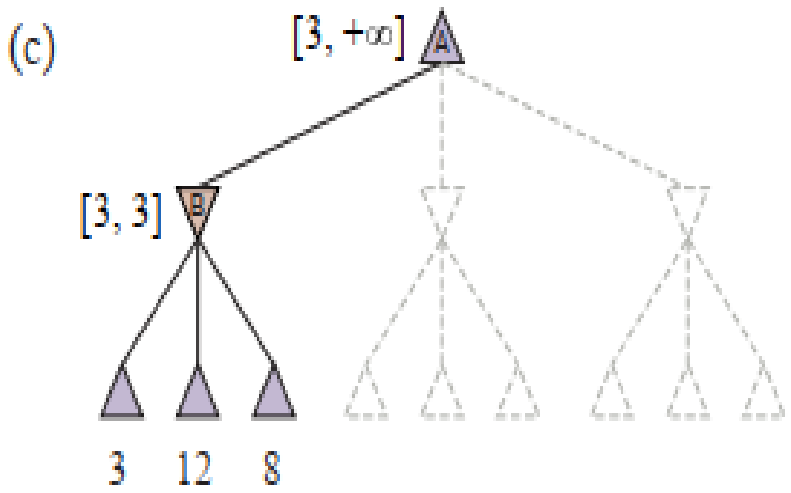


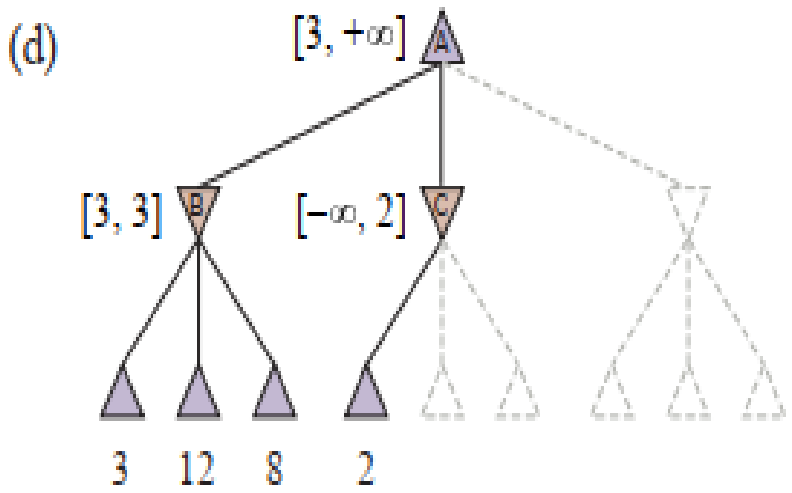


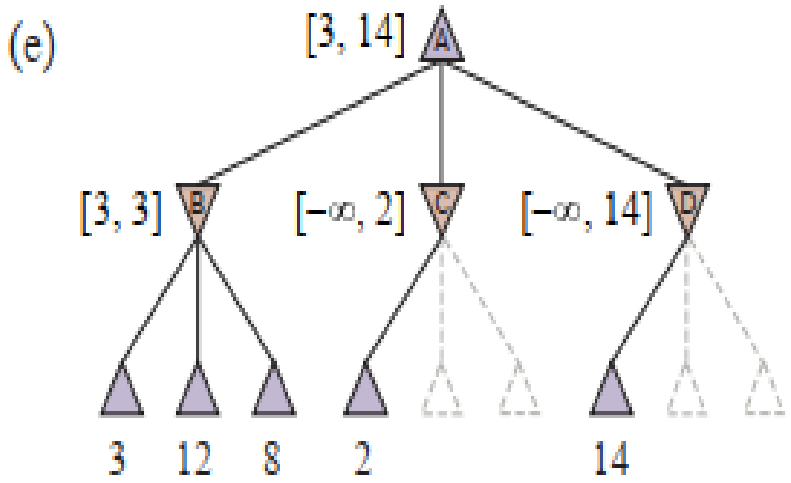


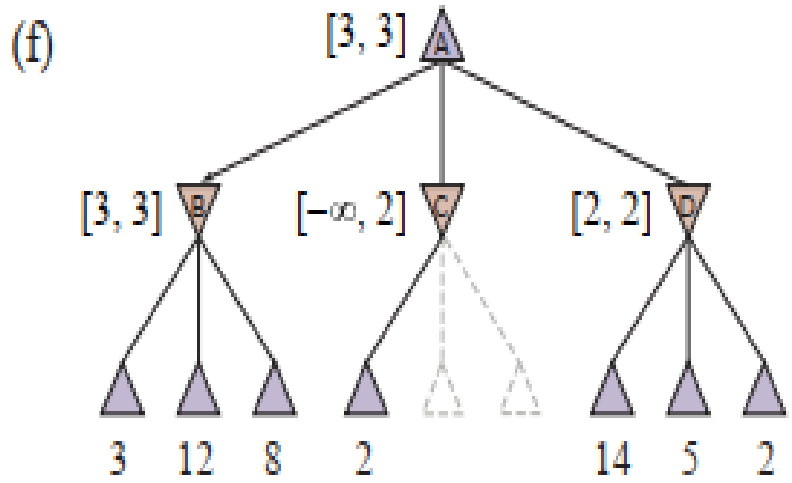
Role of MIN and MAX in adversarial search

- (d) The first leaf below *C* has the value 2.
- Hence, *C*, which is a MIN node, has a value of *at most* 2.
- But we know that *B* is worth 3, so MAX would never choose *C*.
- Therefore, there is no point in looking at the other successor states of *C*.
  - This is an example of alpha–beta pruning
-



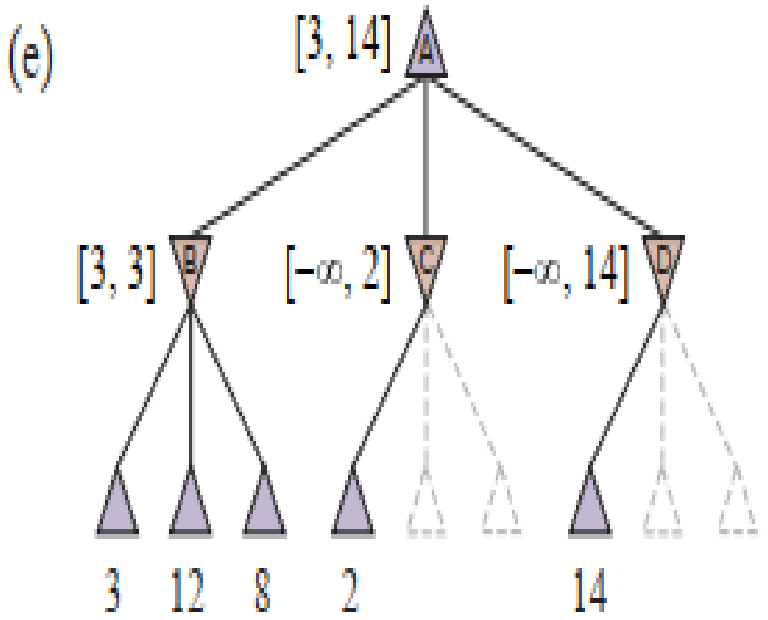


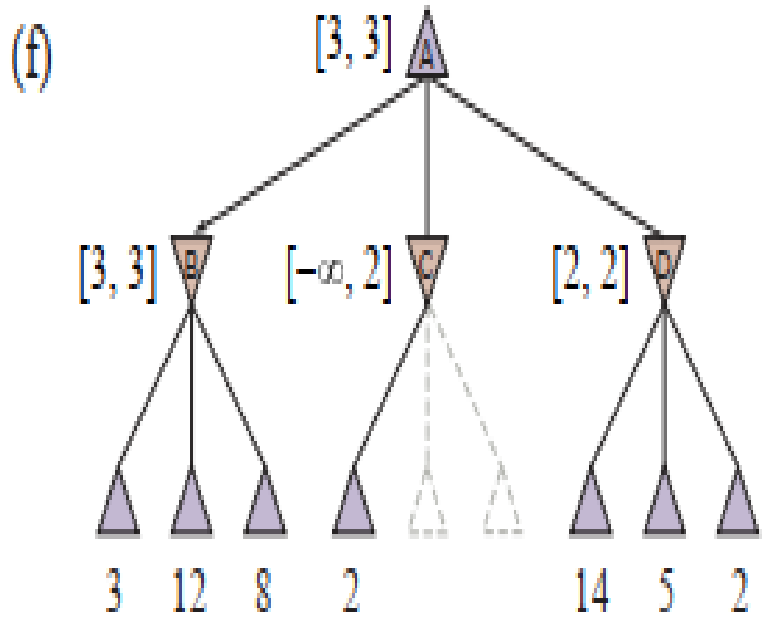




Role of MIN and MAX in adversarial search

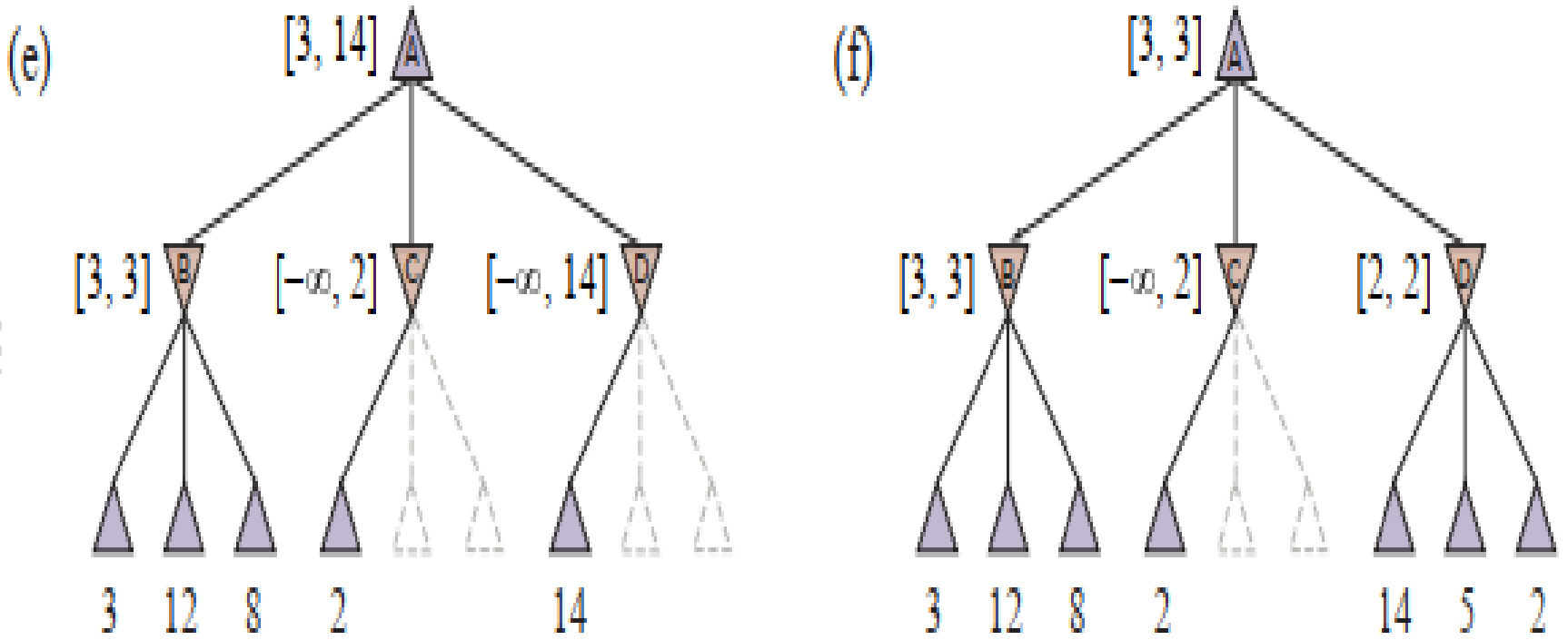
- (e) The first leaf below *D* has the value 14, so *D* is worth *at most* 14.
- This is still higher than MAX's best alternative (i.e., 3), so we need to keep exploring *D*'s successor states.
- Notice also that we now have bounds on all of the successors of the root
- So the root's value is also at most 14



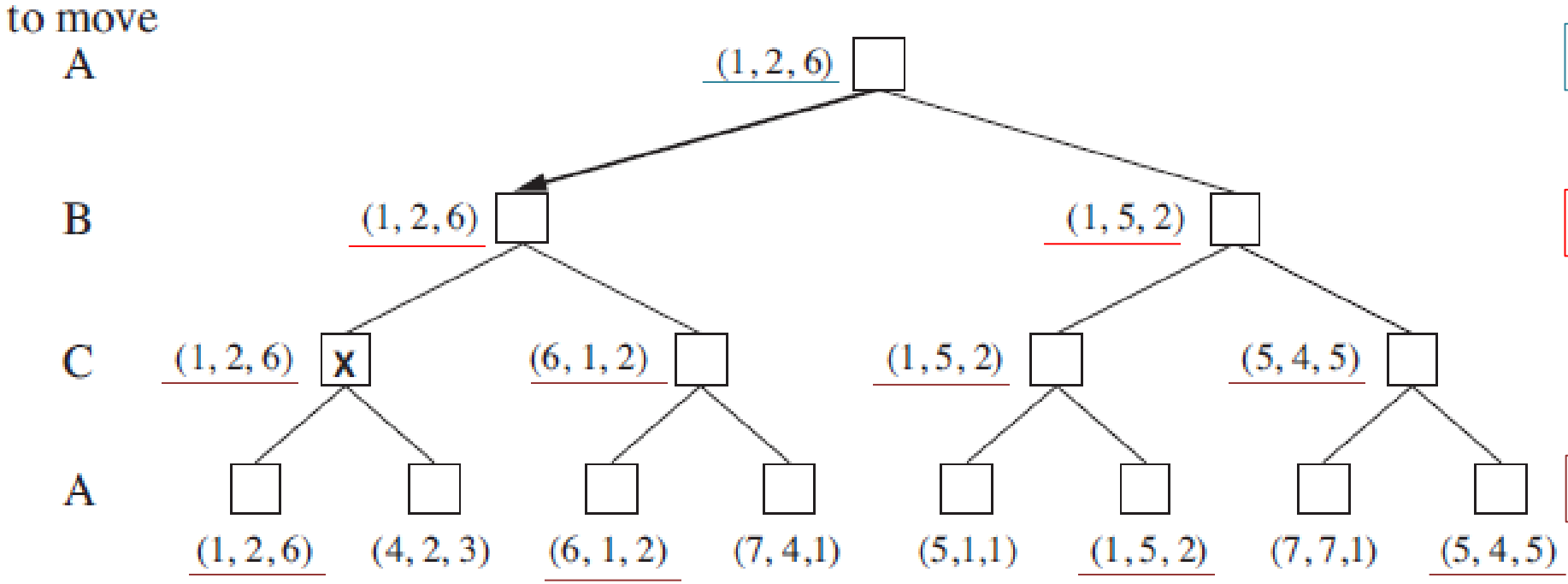


Role of MIN and MAX in adversarial search

- (f) The second successor of *D* is worth 5, so again we need to keep exploring
- The third successor is worth 2
- So now *D* is worth exactly 2
- MAX's decision at the root is to move to *B*, giving a value of 3



For three competitors/three plies in one-deep move



A chooses its best values

B chooses its best values

C chooses its best values

Figure 5.4 The first three plies of a game tree with three players (A, B, C). Each node is labeled with values from the viewpoint of each player. The best move is marked at the root.

Costs

- Perfect play for deterministic games
- **Idea**: choose move to position with highest minimax value
  = best achievable payoff against best play
- **2-ply game / One-move deep**
- Minimax algorithm performs a **complete depth-first exploration** of the game tree
  - Maximum depth of the tree is m and there are b legal moves at each point,
  - then Time Complexity of the minimax algorithm is **$O(b^m)$**
  - Space Complexity is
    - **O(bm)** for an algorithm that generates all actions at once, or
    - **O(m)** for an algorithm that generates actions one at a time
  - **Time cost is totally impractical** – but acts as a basis for the mathematical analysis of games or other practical algorithms

Minimax Strategy

- Why do we take the min value every other level of the tree?
  - These nodes represent the opponent's choice of move.
  - Computer assumes that the human will choose that move that is of least value to the computer
- Minimax properties
  - **Complete**? Yes, if tree is finite
  - **Optimal**? Yes (against an optimal opponent)
  - **Time Complexity** ($O(b^m)$)
  - **Space Complexity** (O(bm)) – Depth first exploration
- For chess, b ≈ 35, m ≈100 for "reasonable" games
  - exact solution completely infeasible

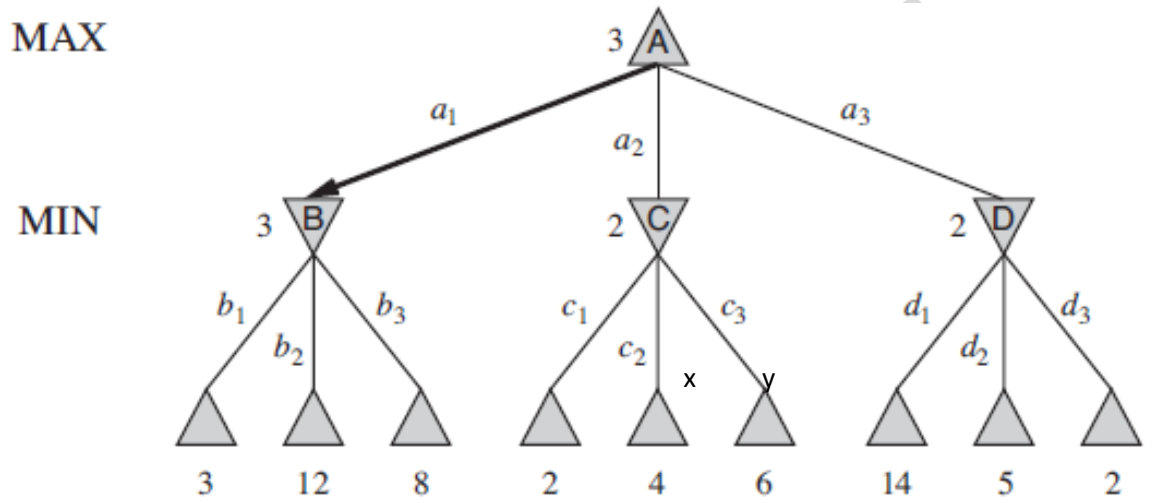
Alpha Beta Procedure

- **Alpha Beta Procedure**
  - The alpha-beta procedure can speed up a depth-first minimax search
- **Alpha**: a **lower bound** on the value that a **maximizing** node may ultimately be assigned ($v \geq \alpha$)
- **Beta**: an **upper bound** on the value that a **minimizing** node may ultimately be assigned ($v \leq \beta$)

Minimax Strategy

- Why alpha – beta pruning?
  - **number of game states** minimax search has to examine is **exponential** in the depth of the tree
- Possible to compute correct minimax decision without looking at every node in the game tree
  - Can't eliminate the exponent, but **effectively cut it in half**
- **alpha–beta pruning**
  - When applied to a standard minimax tree, **returns the same move** as minimax would
  - but **prunes away branches** that cannot possibly influence the final decision

Alpha-Beta pruning



Let the two unevaluated successors of node C in figure have values **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) (applying min for evaluated nodes)

= max(3, z, 2) (where z = min(2, x, y) ≤ 2)

= 3

The value of the root and hence the minimax decision are ***independent*** of the values of the pruned leaves **x** and **y**

Alpha-Beta Pruning

- Recognize when a position can never be chosen in minimax *no matter what its children are*
  - Max (3, Min(2,x,y) …) is always ≥ 3
  - Min (2, Max(3,x,y) …) is always ≤ 2
  - We know this without knowing x and y!
- Alpha = the value of the best choice we've found so far for MAX (highest)
- Beta = the value of the best choice we've found so far for MIN (lowest)
- **When maximizing, cut off values lower than Alpha**
- **When minimizing, cut off values greater than Beta**

Alpha-Beta pruning (General Case)

```
function ALPHA-BETA-SEARCH(state) returns an action
    v ← MAX-VALUE(state, −∞, +∞)
    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 ← −∞
    for each a in ACTIONS(state) do
        v ← MAX(v, MIN-VALUE(RESULT(s,a), α, β))
        if v ≥ β then return v
        α ← MAX(α, v)
    return v
```

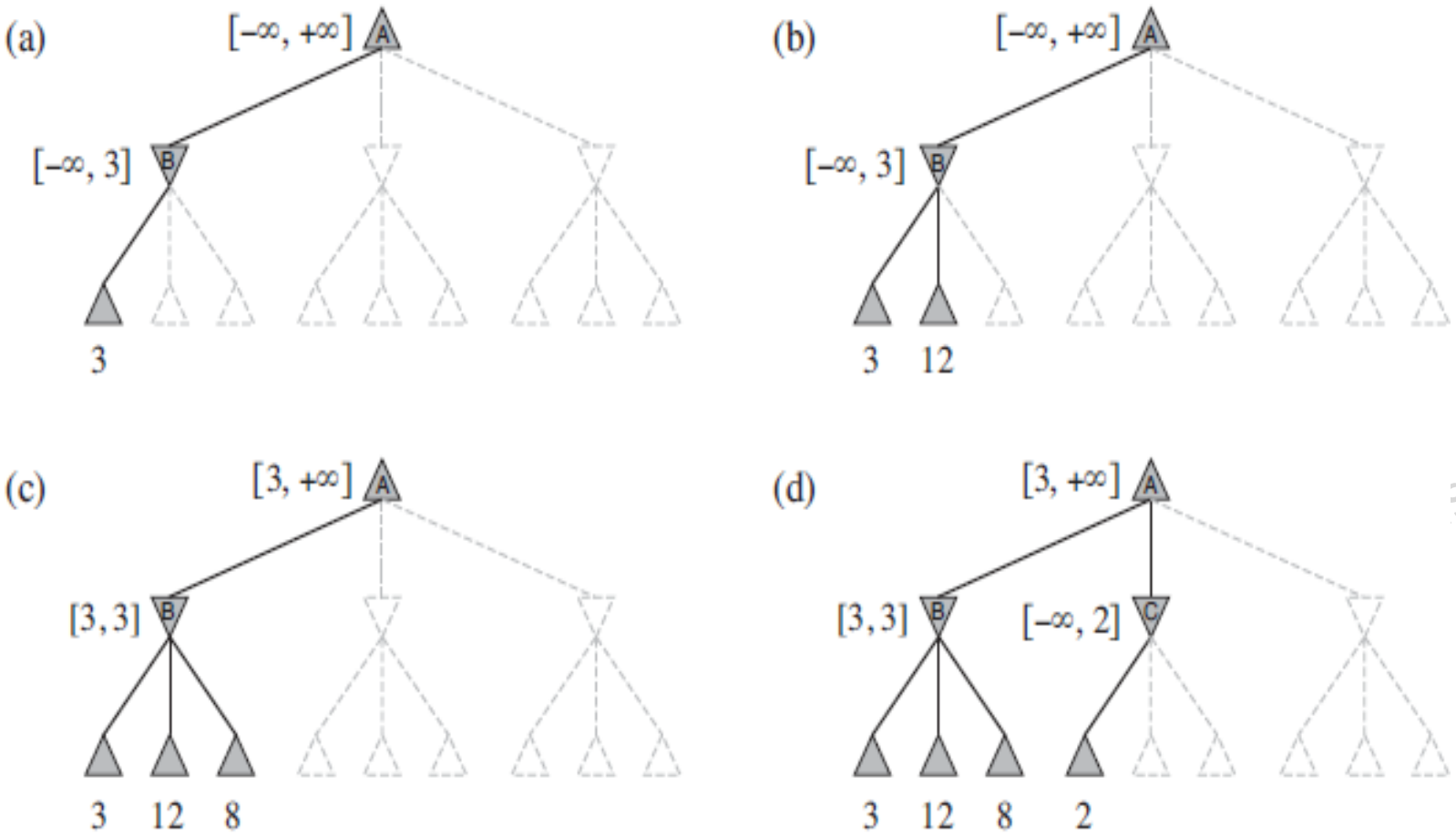
```
function MIN-VALUE(state, α, β) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← +∞
    for each a in ACTIONS(state) do
        v ← MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
        if v ≤ α then return v
        β ← MIN(β, v)
    return v
```

The alpha–beta search algorithm.

These routines are the same as the MINIMAX functions - except for the two lines, in each of MIN-VALUE and MAX-VALUE that maintain α and β

(and the bookkeeping to pass these parameters along)

Alpha-Beta pruning (step-by-step)



(a) The first leaf below B has the value 3.
Hence, B, which is a MIN node, has a value of ***at most* 3**

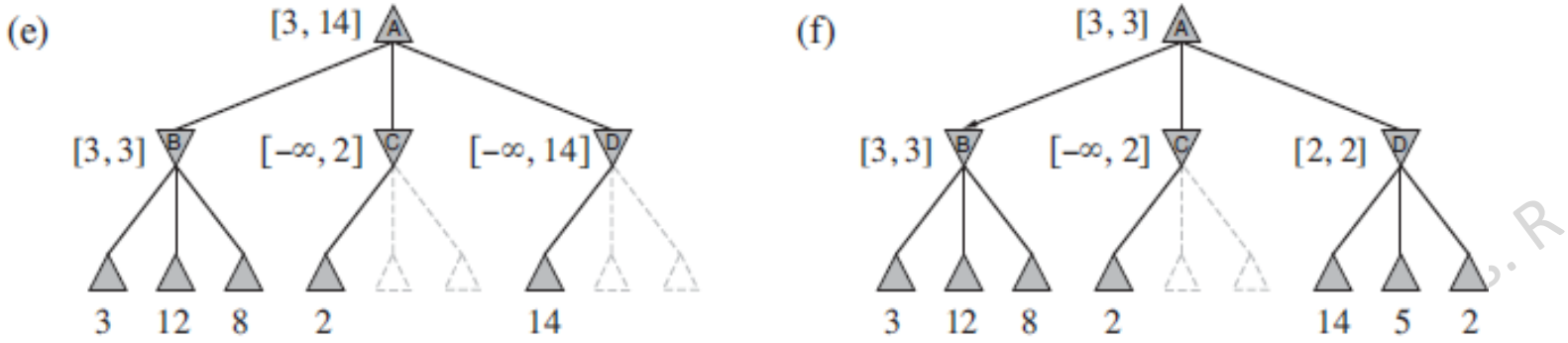
(b) The second leaf below B has a value of 12;
MIN would avoid this move, so the value of B is **still at most 3**

(c) The third leaf below B has a value of 8
All B's successor states are known,
So the value of **B is exactly 3**
So the **value of the root is *at least* 3**, as MAX has a choice ***worth 3 at the root***

(d) The first leaf below C has the value 2.
Hence, C (MIN node) has value of *at most* 2.

But we know that B is worth 3, so MAX would never choose C.
Therefore, there is no point in looking at the other successor states of C.
This is an example of alpha–beta pruning.

Alpha-Beta pruning (step-by-step)



(e) The first leaf below D has the value 14, so D is worth ***at most* 14.**
This is **still higher than MAX's best alternative** (i.e., 3), so we need to keep exploring D's successor states
Notice also that we now have **bounds on all of the successors of the root**, so the root's value is also at most 14

(f) The second successor of D is worth 5, so again we need to keep exploring.
The third successor is worth 2, so now D is worth exactly 2

MAX's decision at the root is to move to B, giving a value of 3.

Alpha-Beta pruning

Another Example:

Let the two unevaluated successors of node B in figure have values **x** and **y**.

MINIMAX(root) = max(min(4, x, y), min(18, 12, 6), min(14, 5, 2))

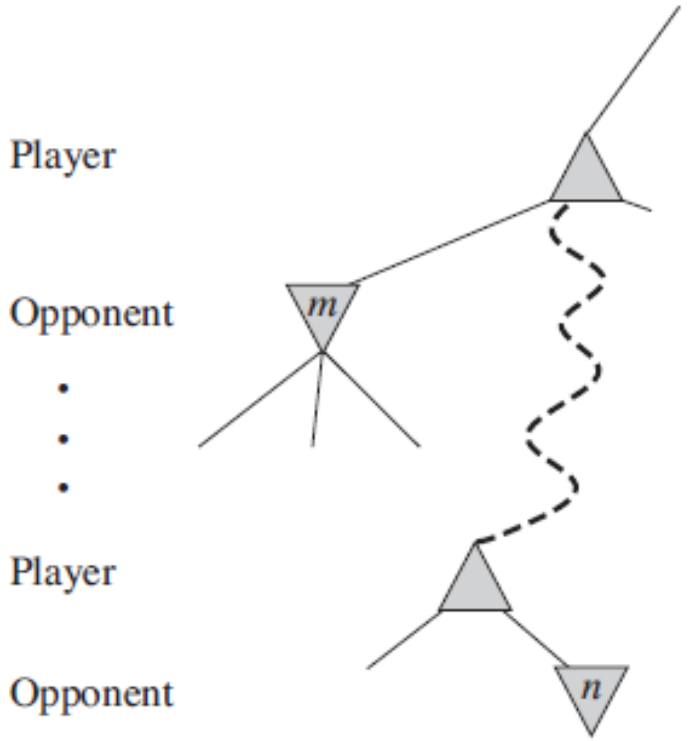
= max(min(4, x, y), 6, 2) (applying min for evaluated nodes)

= max(z, 6, 2) (where z = min(4, x, y) ≤ 4)

= 6

The value of the root and hence the minimax decision are ***independent*** of the values of the pruned leaves **x** and **y**

Alpha-Beta pruning (General Case)



General Principle:

1. Consider a node **n** somewhere in the tree such that Player has a choice of moving to that node.

2. If Player has a better choice **m** either at the parent node of **n**, or at any choice point further up, **then n** ***will never be reached in actual play***

3. So once this conclusion is reached about **n** (by examining some of its descendants), we can prune it

If **m** is better than **n** for Player, n will never be played

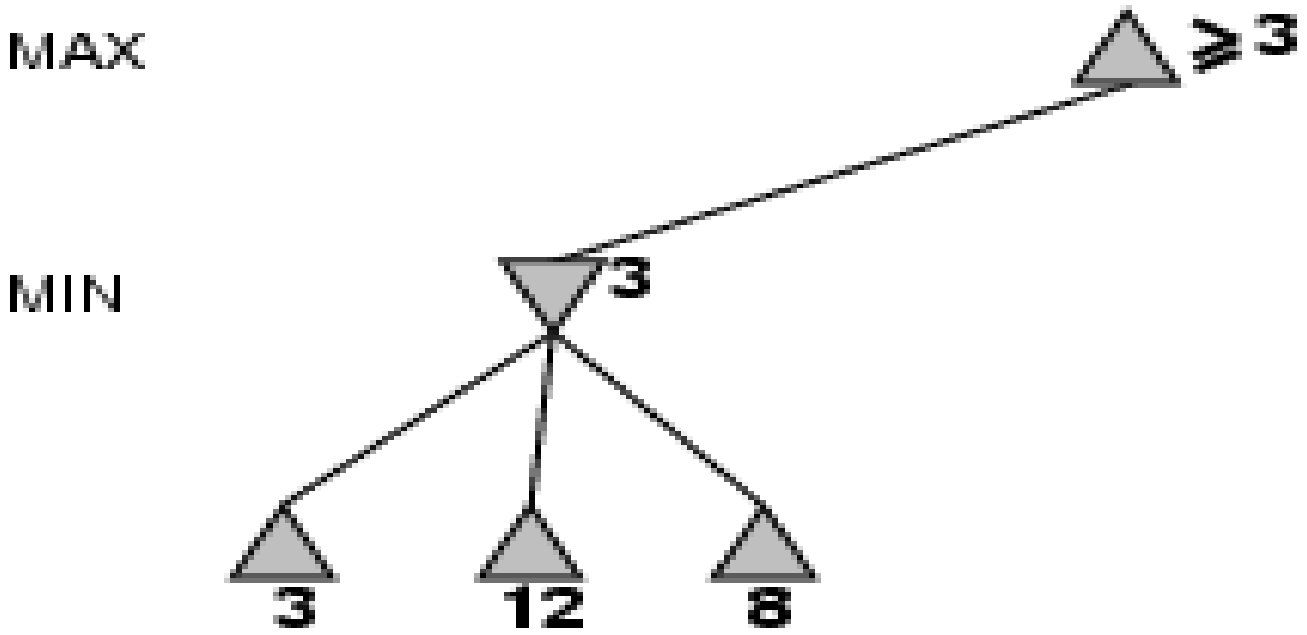
α = the value of the best (i.e., highest-value) choice we have found so far, at any choice point along the path for MAX
(best option already explored along the path to the root for maximizer)

β = the value of the best (i.e., lowest-value) choice we have found so far, at any choice point along the path for MIN
(best option already explored along the path to the root for minimizer)

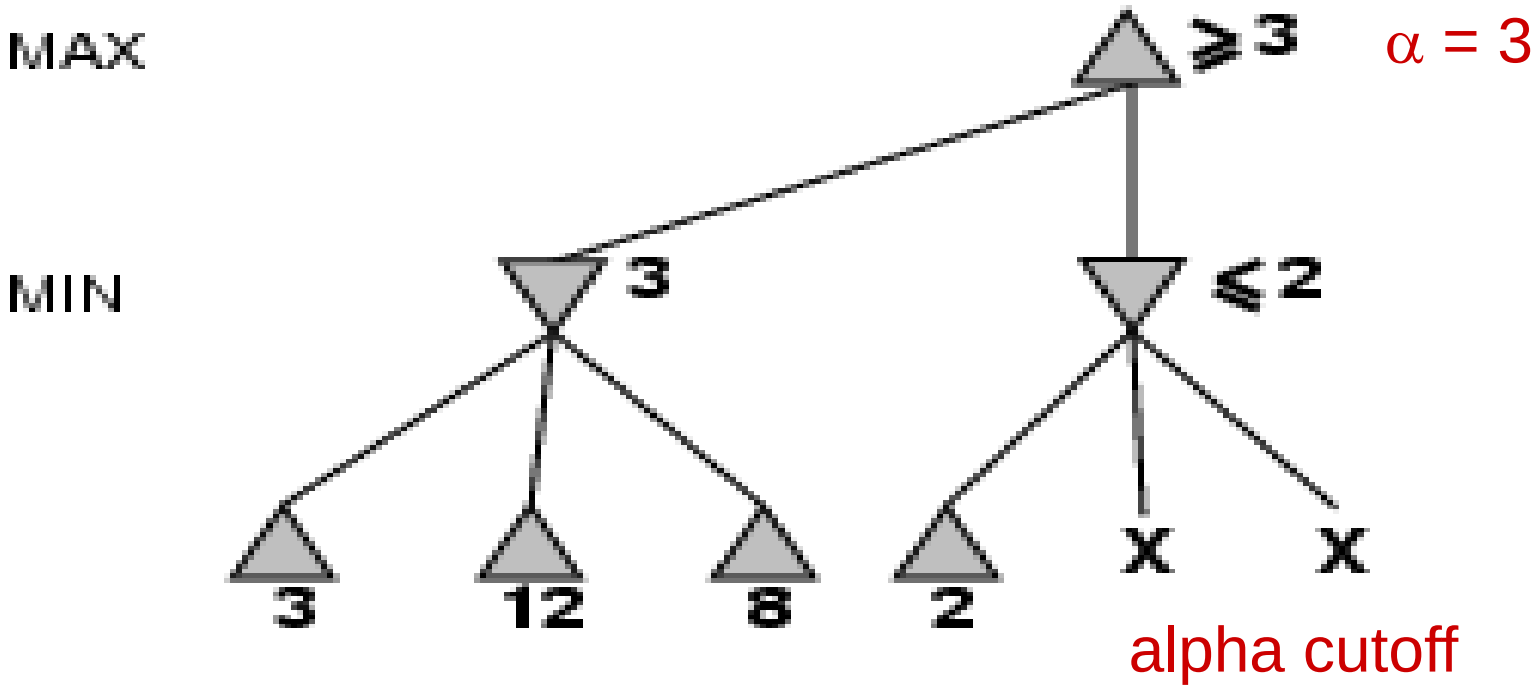
Alpha-Beta Procedure

- The alpha-beta procedure can speed up a depth-first minimax search.
- Alpha: a lower bound on the value that a max node may ultimately be assigned ($c \geq \alpha$)
- α = the value of the best (i.e., highest-value) choice we have found so far at any choice point along the path for MAX.
- Beta: an upper bound on the value that a minimizing node may ultimately be assigned ($c \leq \beta$)
- β = the value of the best (i.e., lowest-value) choice we have found so far at any choice point along the path for MIN

α-β pruning example



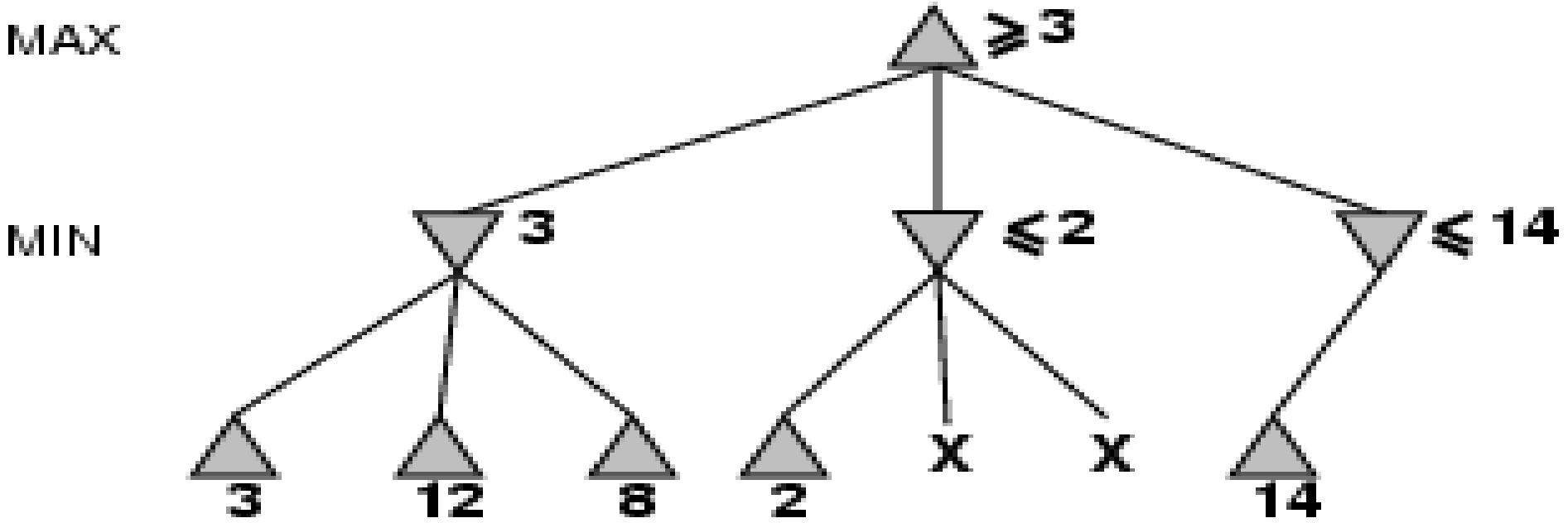
α-β pruning example



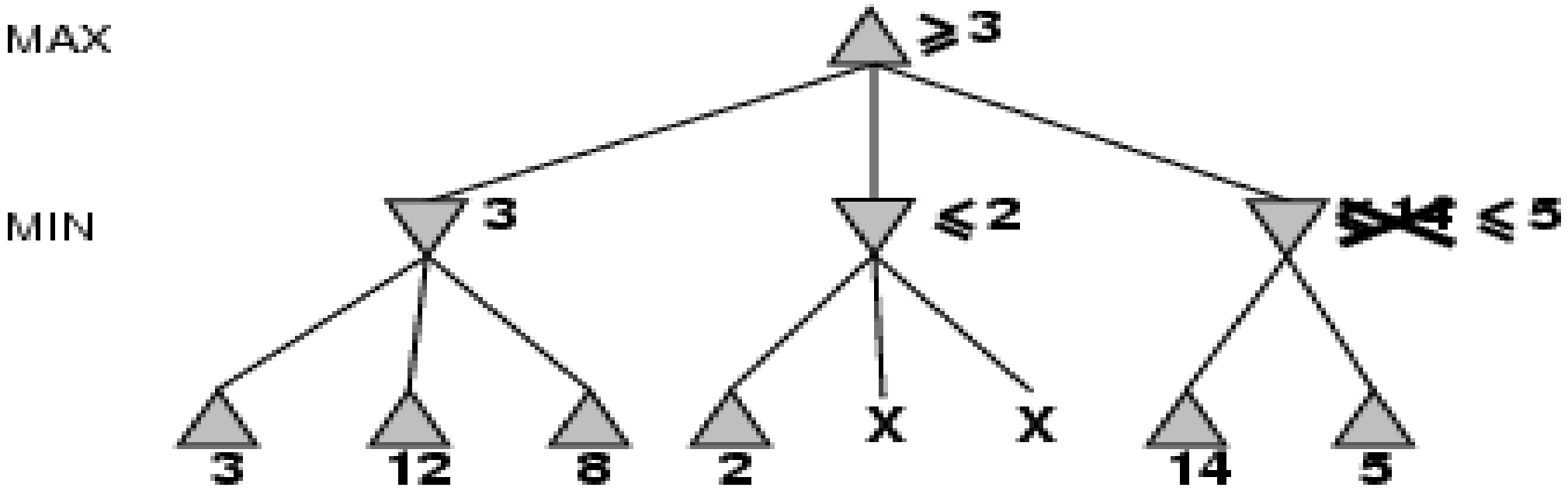
α = 3

alpha cutoff

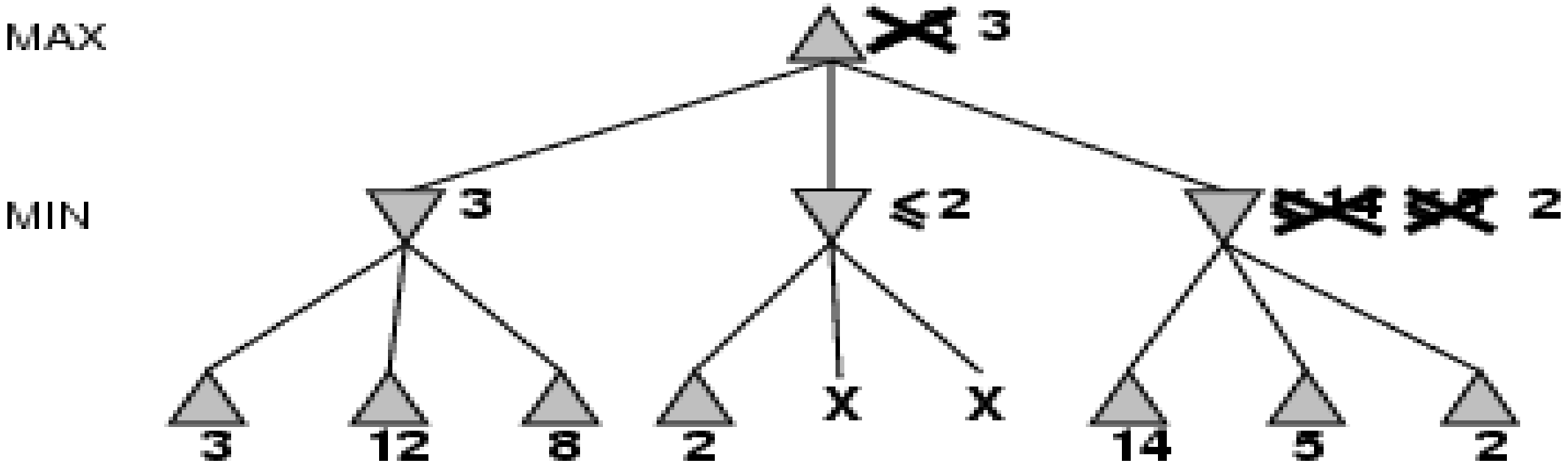
α-β pruning example



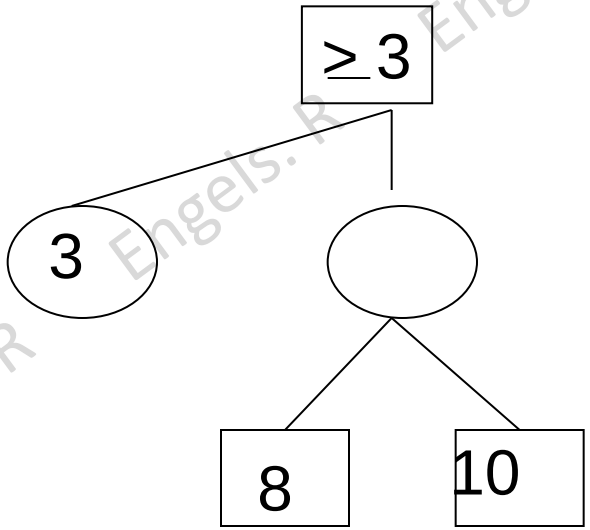
α-β pruning example



α-β pruning example



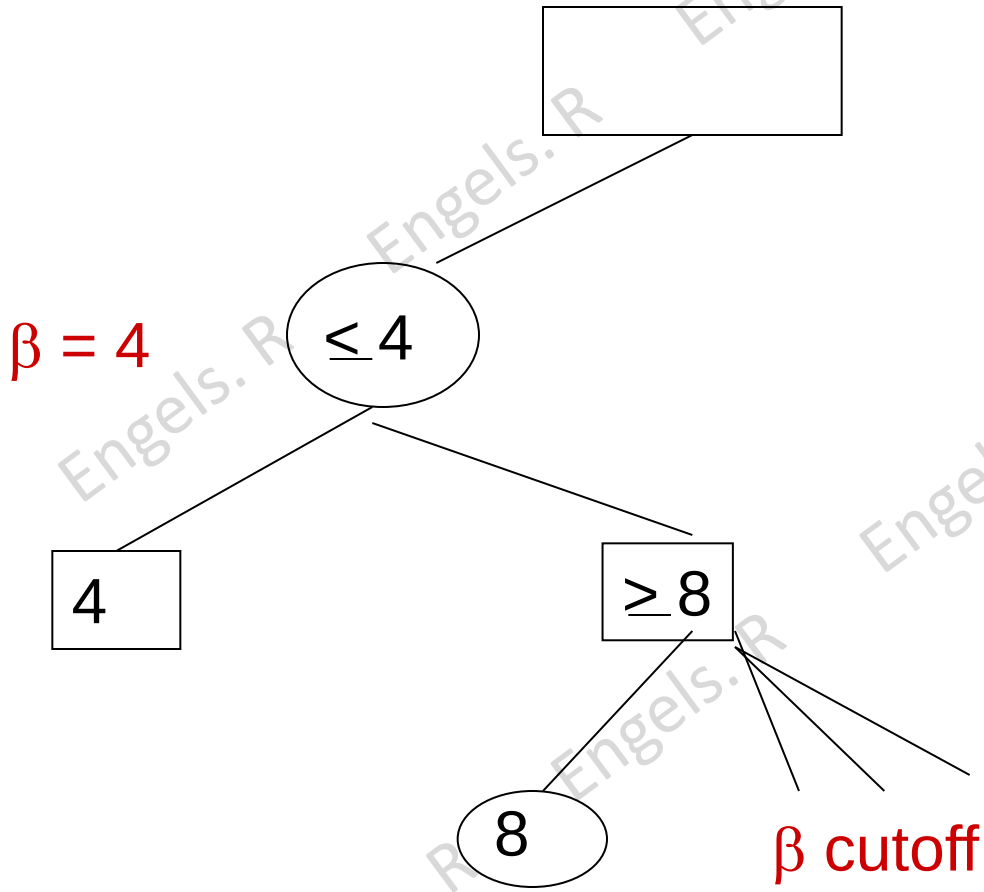
Alpha Cutoff



$\alpha = 3$

What happens here? Is there an alpha cutoff?

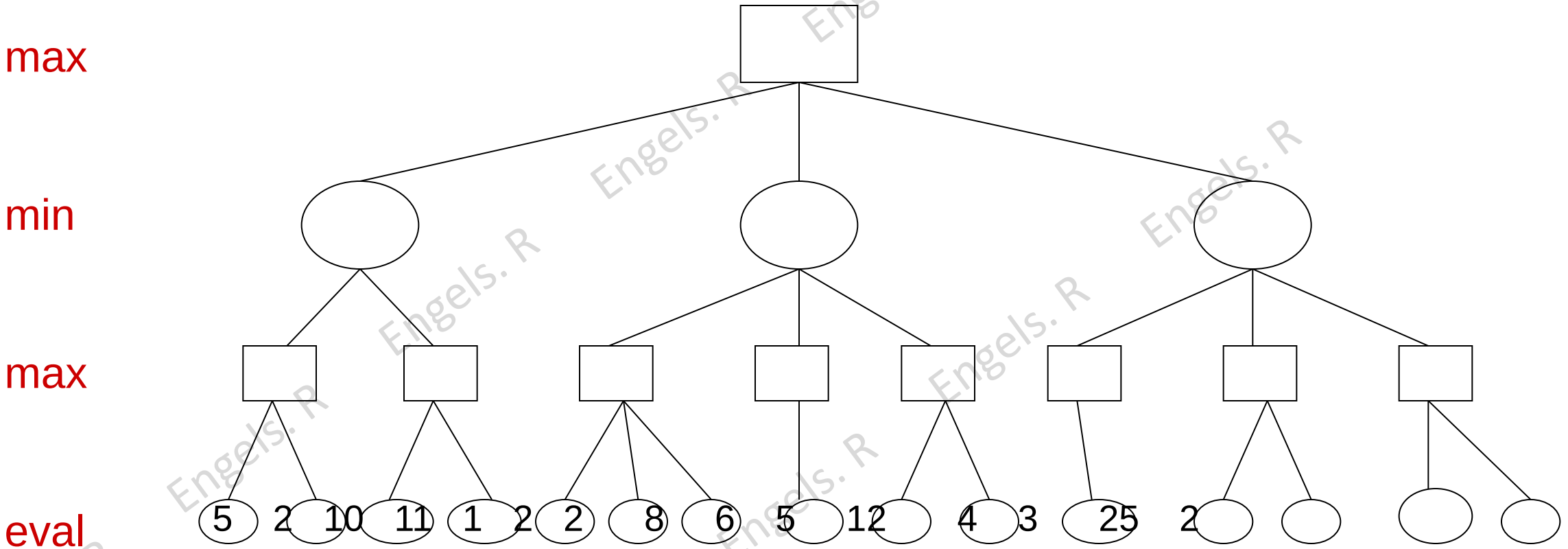
Beta Cutoff



Properties of α-β

- Pruning does not affect final result. This means that it gets the exact same result as does full minimax.
- Good move ordering improves effectiveness of pruning
- With "perfect ordering," time complexity = $O(b^{m/2})$
  → doubles depth of search
- A simple example of the value of reasoning about which computations are relevant (a form of metareasoning)

Alpha-Beta Pruning



Summary

- **SEARCH STRATEGIES** (12)
  - Breadth-First Search
  - Uniform Cost Search
  - Depth-First Search
  - Depth-Limited Search
  - Iterative Deepening Search
  - Bidirectional Search
- **Heuristic Search Techniques**
  - A* Search
  - AO* Algorithm
- **Adversarial Search:**
  - Minimax Algorithm
  - Alpha beta Pruning

References

- AIMA
  - Artificial Intelligence - A Modern Approach 3rd Edition - RUSSELL & NORVIG
- AI
  - Artificial Intelligence – Elaine Rich, Kevin Knight, B. Nair 3rd Edition