Chapter 6 - Adversarial Search and Games

Russell, S., & Norvig, P. (2022). *Artificial Intelligence - A Modern Approach* (4th global ed.). Pearson.

Contents

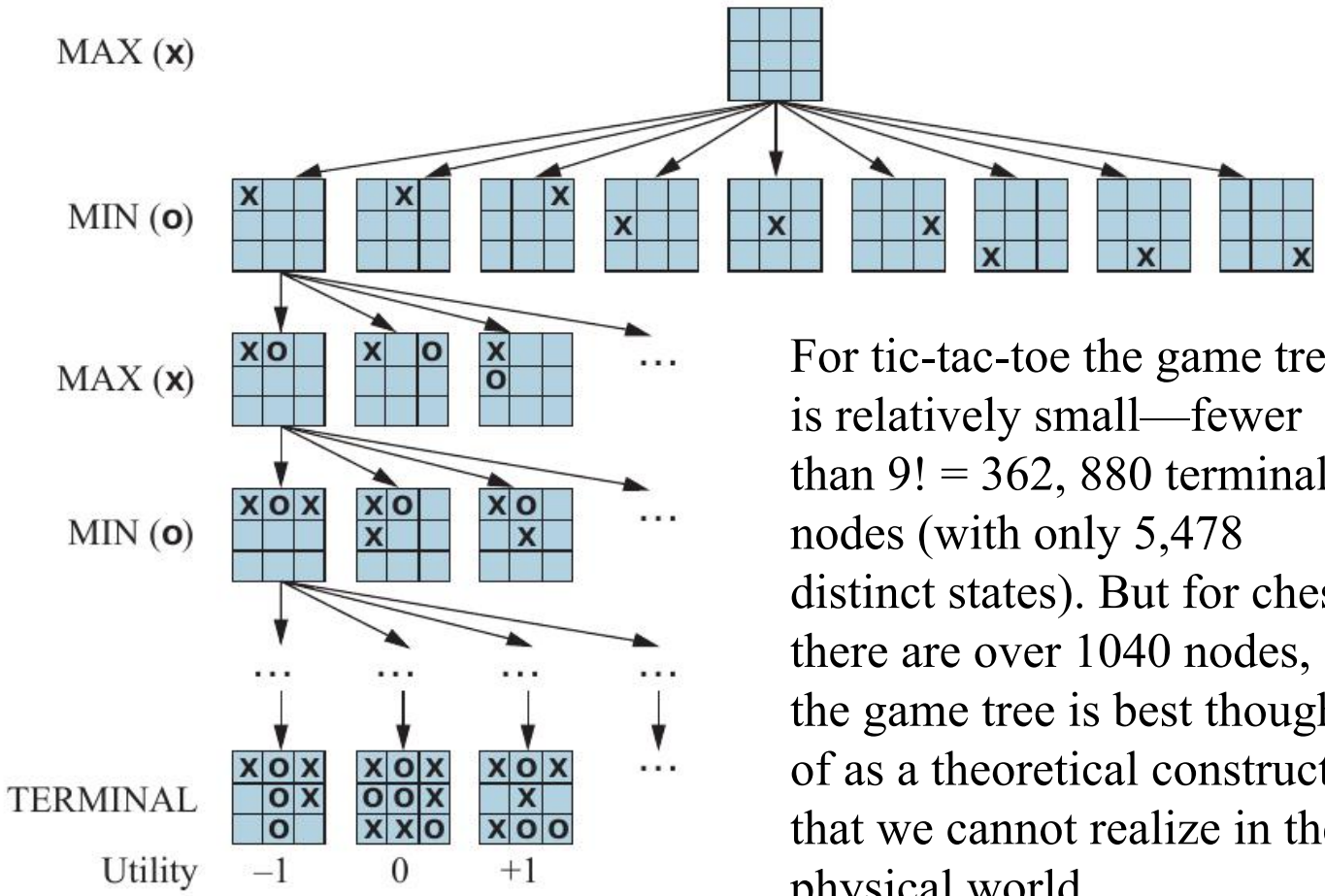
1. Game Theory
Two-Player Zero-Sum Games

- The games most commonly studied within AI (such as chess and Go) are what game theorists call *deterministic, two-player, turn-taking, perfect information, zero-sum games*.
  - "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.
- For games we often use the term move as a synonym for "action" and position as a synonym for "state."
- We will call our two players MAX and MIN. MAX moves first, and then the players 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.

1. Game Theory
Two-Player Zero-Sum Games

- 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. Some games have a wider range of possible outcomes—for example, the payoffs in backgammon range from 0 to 192.

1. Game Theory
Two-Player Zero-Sum Games



For tic-tac-toe the game tree is relatively small—fewer than 9! = 362, 880 terminal nodes (with only 5,478 distinct states). But for chess there are over 1040 nodes, so the game tree is best thought of as a theoretical construct that we cannot realize in the physical world.

- Figure 6.1 shows part of the game tree for tic-tac-toe (noughts and crosses).
- 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 such that one player has three squares in a row or all the squares are filled.
- The number on each leaf node indicates the utility value of the terminal state from the point of view of MAX; high values are good for MAX and bad for MIN (which is how the players get their names).

2. Optimal Decisions in Games

- MAX wants to find a sequence of actions leading to a win, but MIN has something to say about it.
- This means that MAX's strategy must be a conditional plan—a contingent strategy specifying a response to each of MIN's possible moves.
- In games that have a binary outcome (win or lose), we could use AND–OR search to generate the conditional plan.
  - In fact, for such games, the definition of a winning strategy for the game is identical to the definition of a solution for a nondeterministic planning problem: in both cases the desirable outcome must be guaranteed no matter what the "other side" does.
- For games with multiple outcome scores, we need a slightly more general algorithm called **minimax search**.

2. Optimal Decisions in Games

- Given a game tree, the optimal strategy can be determined by working out the **minimax** value of each state in the tree, which we write as MINIMAX(*s*).
- The minimax value is the utility (for MAX) of being in that state, assuming that both players play optimally from there to the end of the game.
- The minimax value of a terminal state is just its utility.
- In a non-terminal state, MAX prefers to move to a state of maximum value when it is MAX's turn to move, and MIN prefers a state of minimum value (that is, minimum value for MAX and thus maximum value for MIN).
  - $$MINIMAX(s) = \begin{cases} UTILITY(s, MAX) & if\ IS-TERMINAL(s) \\ max_{a\ \in\ Actions(s)} MINIMAX(RESULT(s,a)) & if\ TO-MOVE(s) = MAX \\ min_{a\ \in\ Actions(s)} MINIMAX(RESULT(s,a)) & if\ TO-MOVE(s) = MIN \end{cases}$$

2. Optimal Decisions in Games

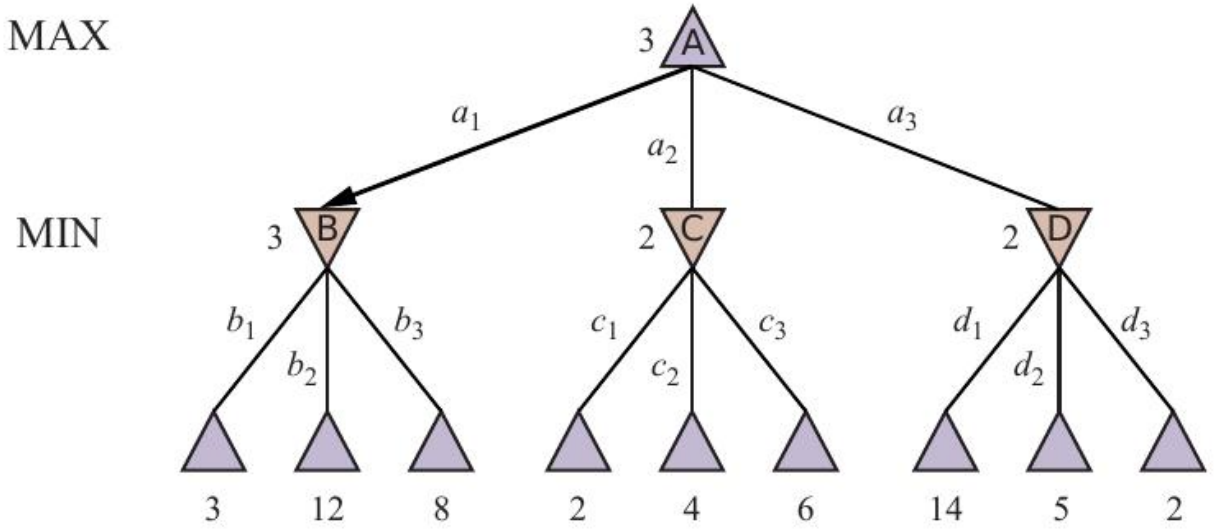


Figure 6.2 A two-ply game tree. The △ nodes are "MAX nodes," in which it is MAX's turn to move, and the ▽ nodes are "MIN nodes." The terminal nodes show the utility values for MAX; the other nodes are labeled with their minimax values. MAX's best move at the root is a_1, because it leads to the state with the highest minimax value, and MIN's best reply is b_1, because it leads to the state with the lowest minimax value.

- The **minimax decision** at the root:
  - Action a_1 is the optimal choice for MAX because it leads to the state with the highest minimax value.

2. Optimal Decisions in Games
The Minimax Search Algorithm

```
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
```

- **Figure 6.3** 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 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 and the move to get there.

2. Optimal Decisions in Games
The Minimax Search Algorithm

- The minimax algorithm performs a complete depth-first exploration of the game tree.
- If the maximum depth of the tree is m and there are b legal moves at each point, then the time complexity of the minimax algorithm is $O(b^m)$.
  - The exponential complexity makes MINIMAX impractical for complex games.
    - Chess has a branching factor of about 35 and the average game has depth of about 80 ply, and it is not feasible to search $35^{80} \approx 10^{123}$ states.
  - MINIMAX does, however, serve as a basis for the mathematical analysis of games. By approximating the minimax analysis in various ways, we can derive more practical algorithms.

2. Optimal Decisions in Games

Optimal Decisions in Multiplayer Games

- Many popular games allow more than two players.
- Let us examine how to extend the minimax idea to multiplayer games.
  - Replace the single value for each node with a *vector* of values.
    - For example, in a three-player game with players A, B, and C, a vector $<v_A, v_B, v_C>$ is associated with each node.
  - For terminal states, this vector gives the utility of the state from each player's viewpoint. The UTILITY function returns a vector of utilities.
  - For nonterminal states, the backed-up value of a node n is the utility vector of the successor state with the highest value for the player choosing at n.

2. Optimal Decisions in Games

Optimal Decisions in Multiplayer Games

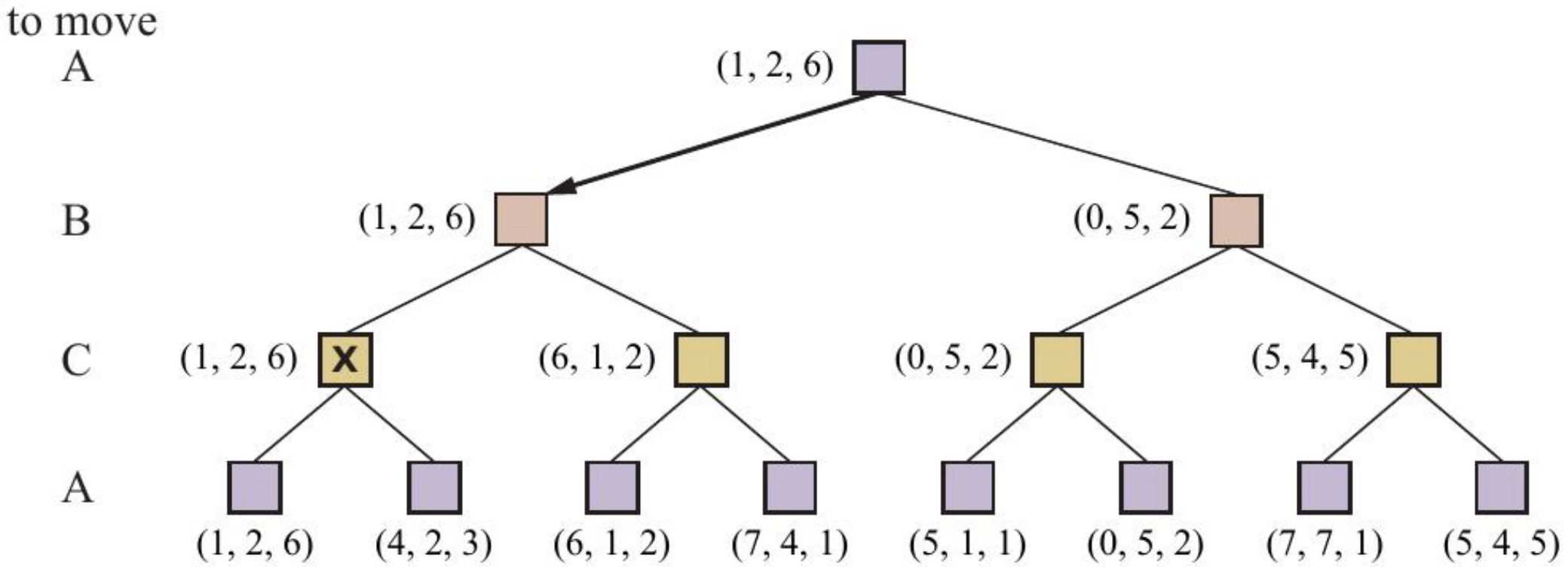


Figure 6.4 The first three ply of a game tree with three players (*A*, *B*, *C*). Each node is labeled with values from the viewpoint of each player. The best move is marked at the root.

2. Optimal Decisions in Games

Optimal Decisions in Multiplayer Games

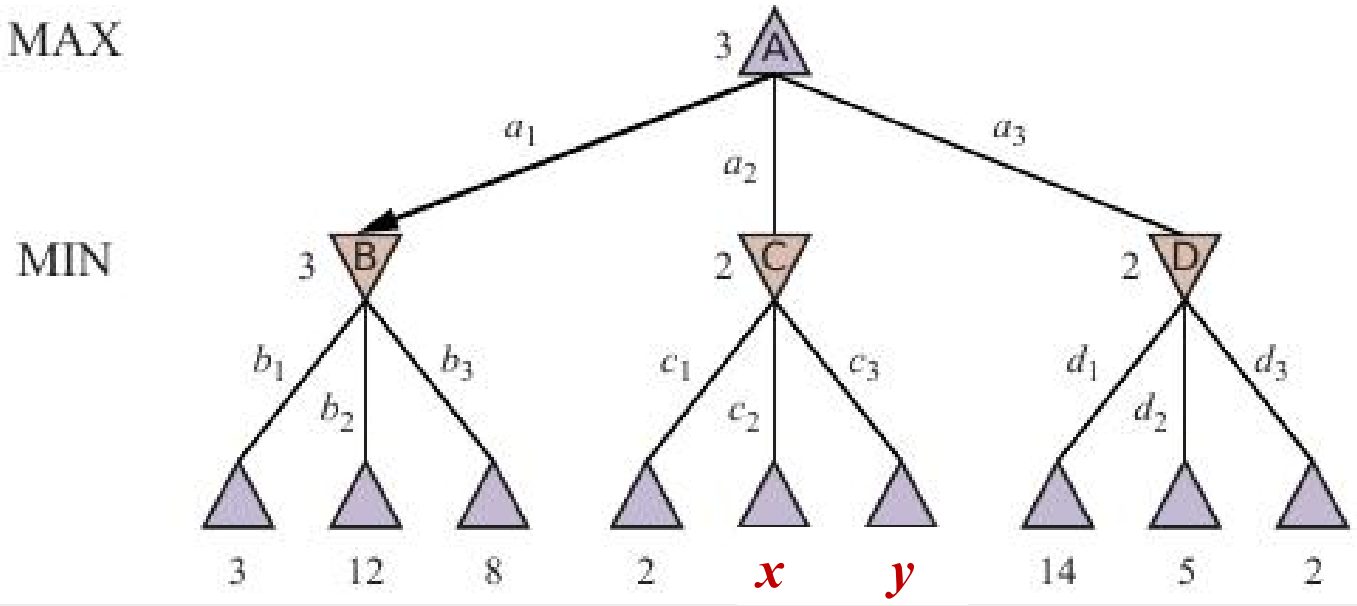
- If the game is not zero-sum, then collaboration can also occur with just two players.
- Suppose, for example, that there is a terminal state with utilities $<v_A = 1000, v_B = 1000>$ and that 1000 is the highest possible utility for each player. Then the optimal strategy is for both players to do everything possible to reach this state—that is, the players will automatically cooperate to achieve a mutually desirable goal.

2. Optimal Decisions in Games
Alpha-Beta Pruning

- The number of game states is exponential in the depth of the tree. No algorithm can completely eliminate the exponent, but we can sometimes cut it in half, computing the correct minimax decision without examining every state by **pruning** large parts of the tree that make no difference to the outcome.
  - **Alpha–Beta pruning**.

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

root A and hence the minimax decision are *independent*
he leaves x and y, and therefore they can be pruned.

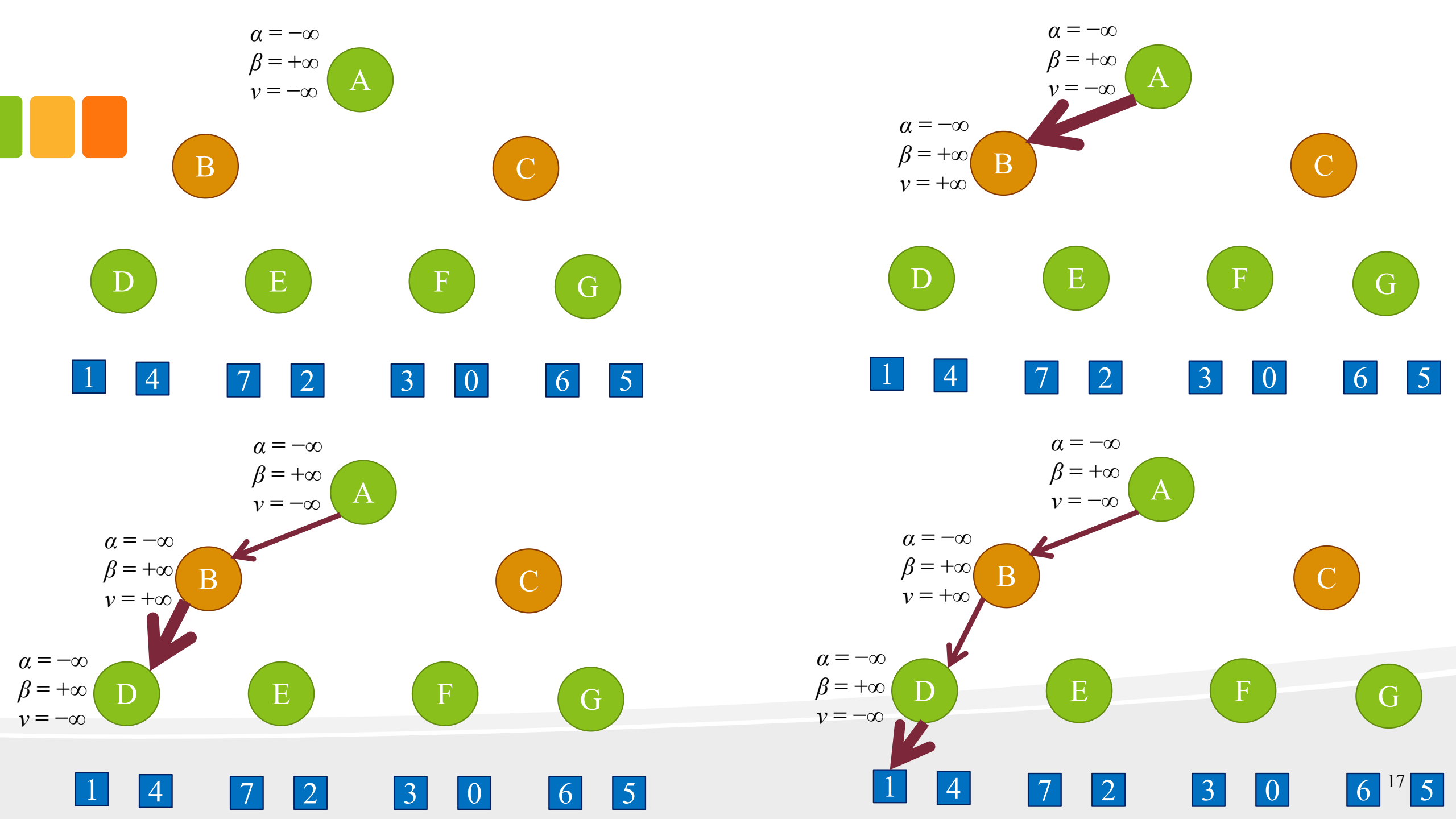


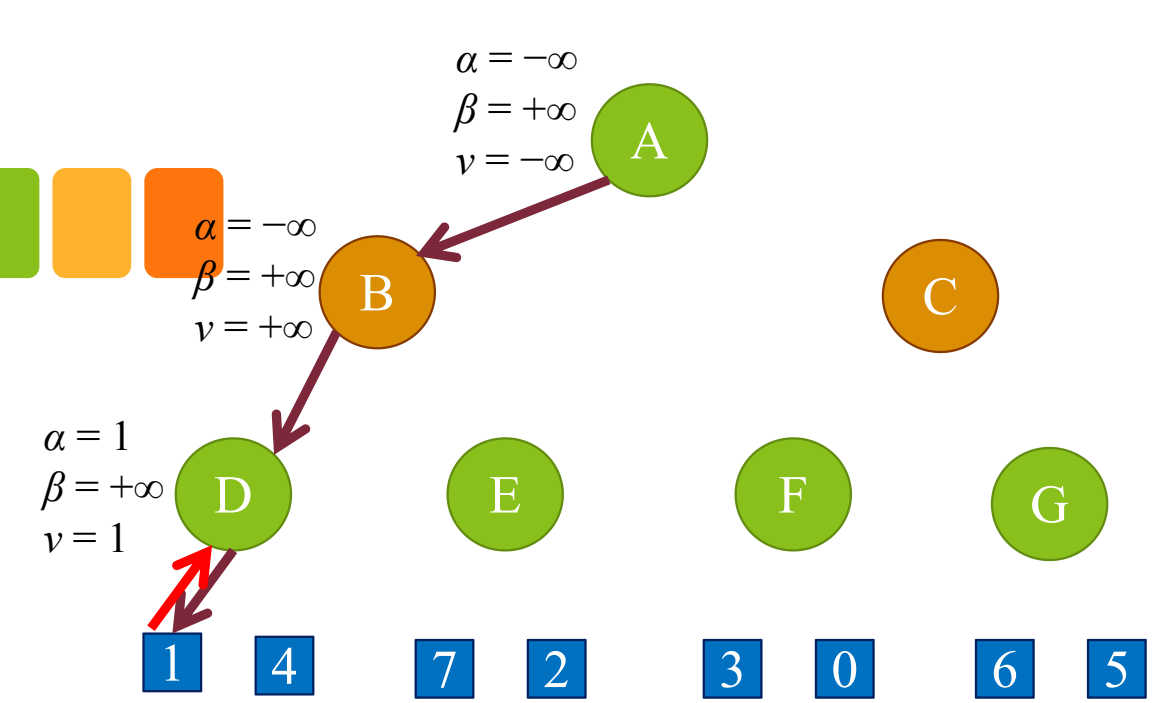
2. Optimal Decisions in Games
Alpha-Beta Pruning

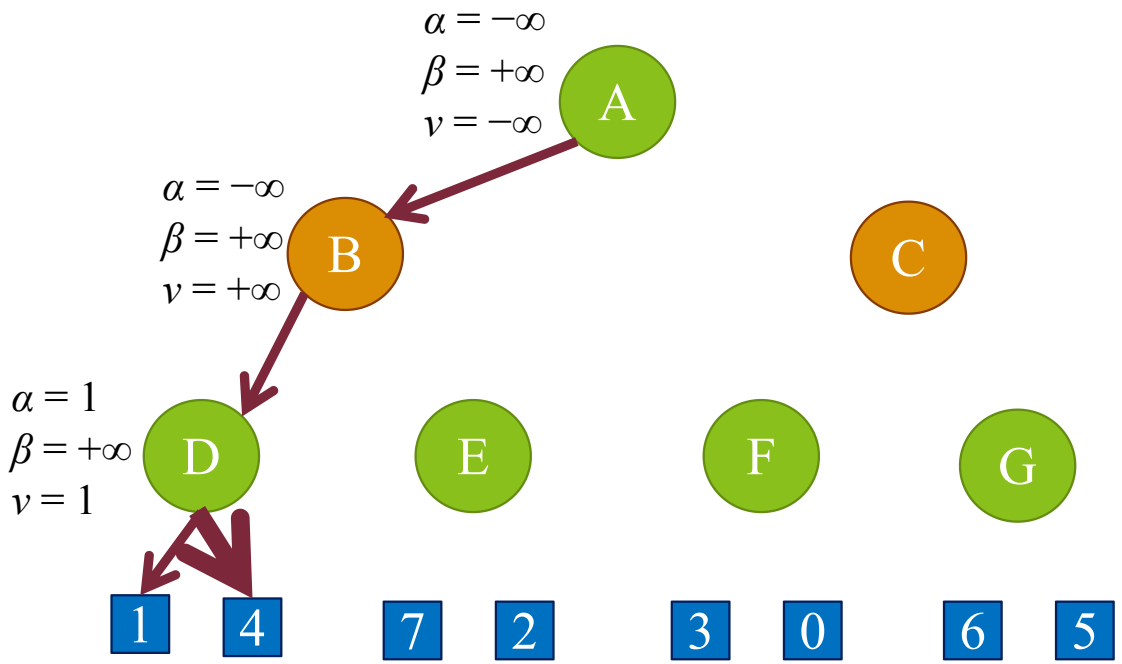
- https://www.scaler.com/topics/artificial-intelligence-tutorial/alpha-beta-pruning/
- Parameters of the Alpha-Beta Pruning Algorithm
  - α: The best choice (highest utility/value) found till the current state on the path traversed by the MAX.
  - β: The best choice (lowest utility/value) found till the current state on the path traversed by the MIN.
- The initialization of the parameters
  - $\alpha = -\infty, \beta = +\infty$
- Updating the parameters
  - α is updated only by the MAX at its turn.
  - β is updated only by the MIN at its turn.

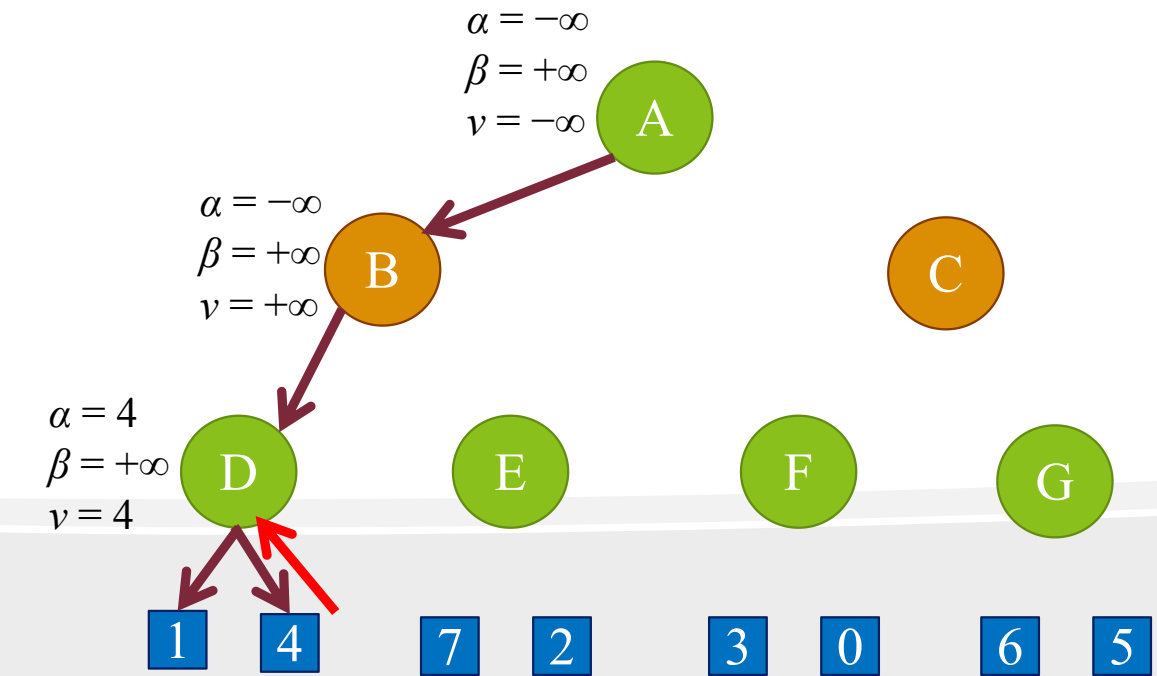
2. Optimal Decisions in Games
Alpha-Beta Pruning

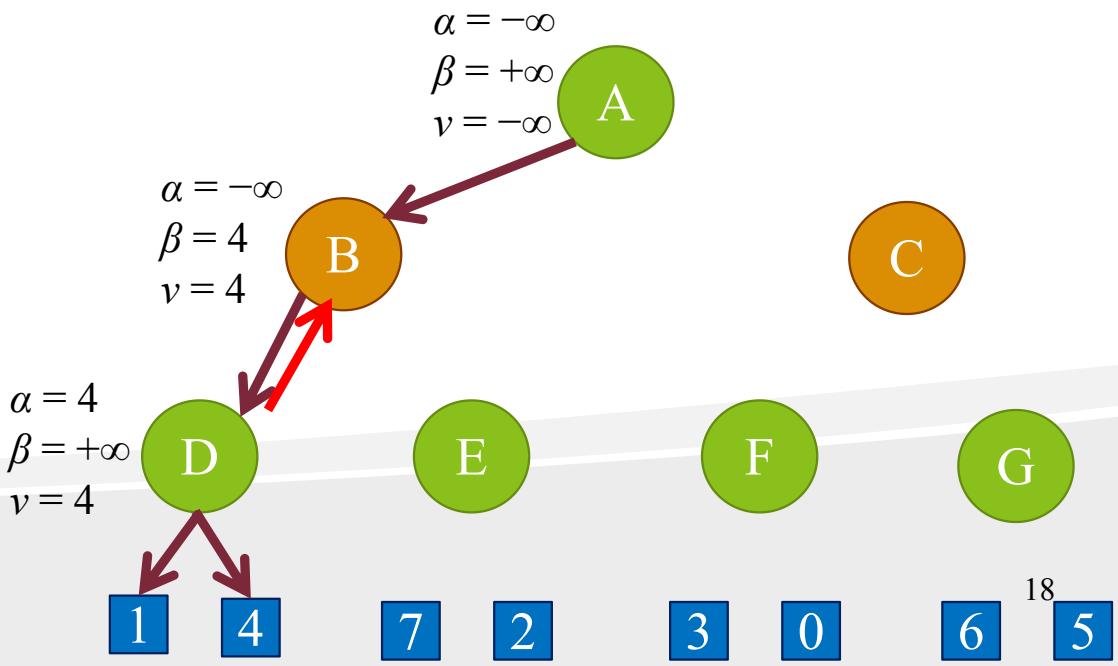
- Passing the parameters
  - α, β are passed on to only child nodes.
  - While bactracking the game tree, the node values are passed to parent nodes.
- Pruning condition
  - The child sub-trees which are not yet traversed are pruned if the condition $\alpha \geq \beta$ hold.

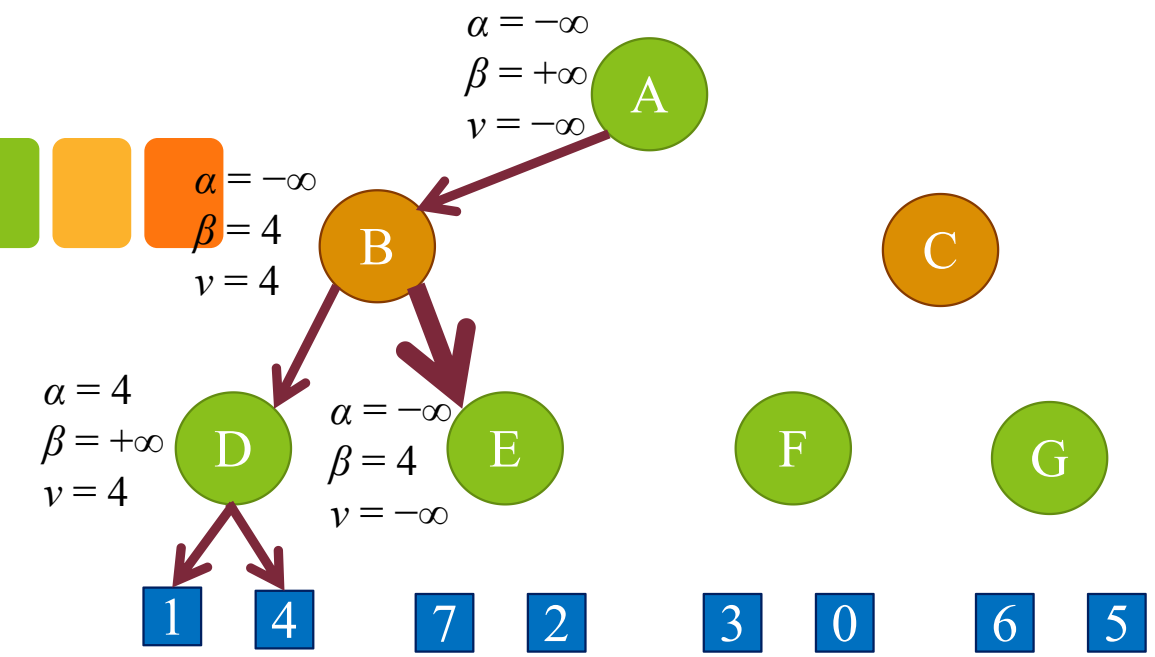

α = −∞
β = +∞
v = −∞
A
B
C
D
E
F
G
1 4 7 2 3 0 6 5
α = −∞
β = +∞
v = +∞
B
α = −∞
β = +∞
v = −∞
D

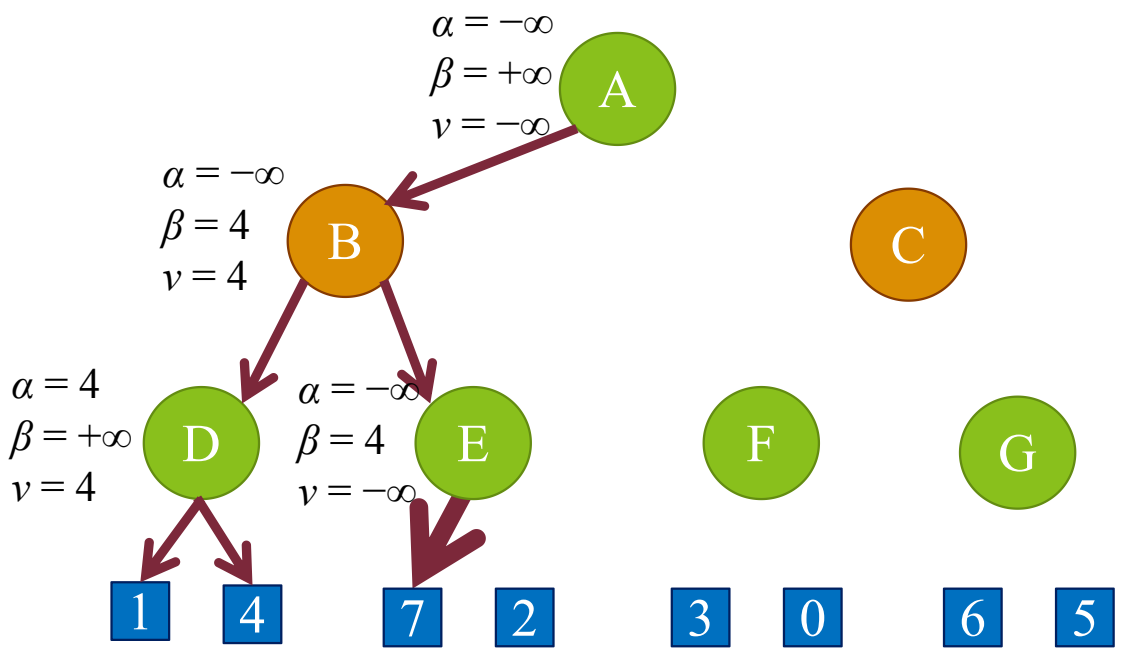

α = −∞
β = +∞
v = −∞
A
α = −∞
β = +∞
v = +∞
B
C
α = 1
β = +∞
v = 1
D
E
F
G
1
4
7
2
3
0
6
5

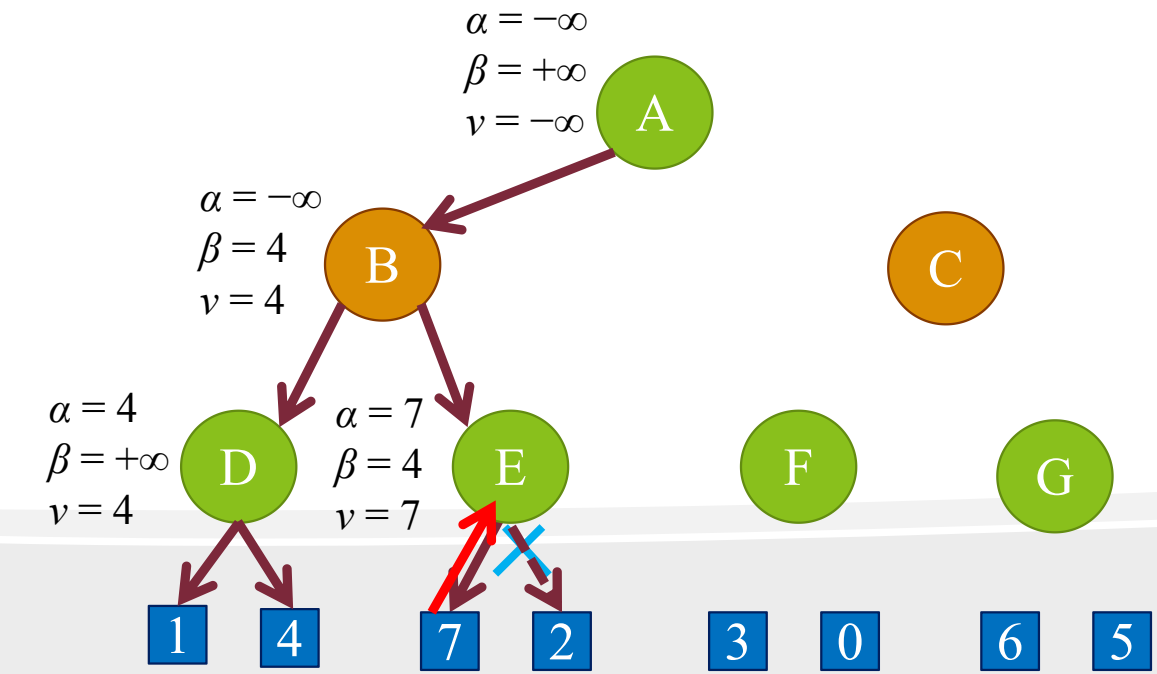

α = −∞
β = +∞
v = −∞
A
α = −∞
β = +∞
v = +∞
B
C
α = 1
β = +∞
v = 1
D
E
F
G
1
4
7
2
3
0
6
5

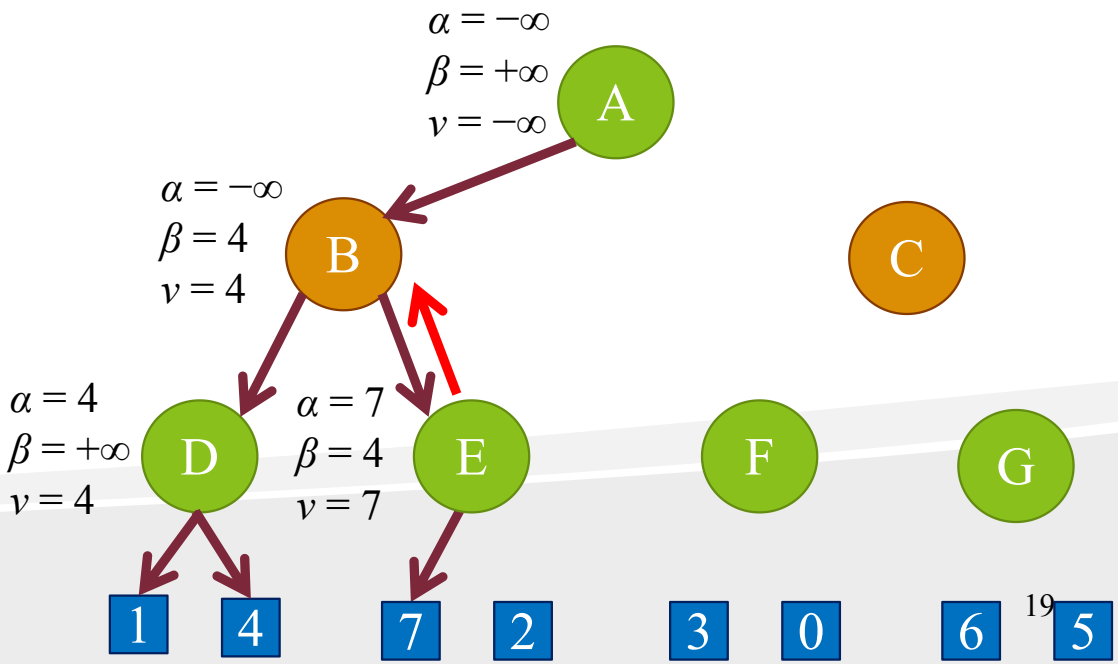

α = −∞
β = +∞
v = −∞
A
α = −∞
β = +∞
v = +∞
B
C
α = 4
β = +∞
v = 4
D
E
F
G
1
4
7
2
3
0
6
5

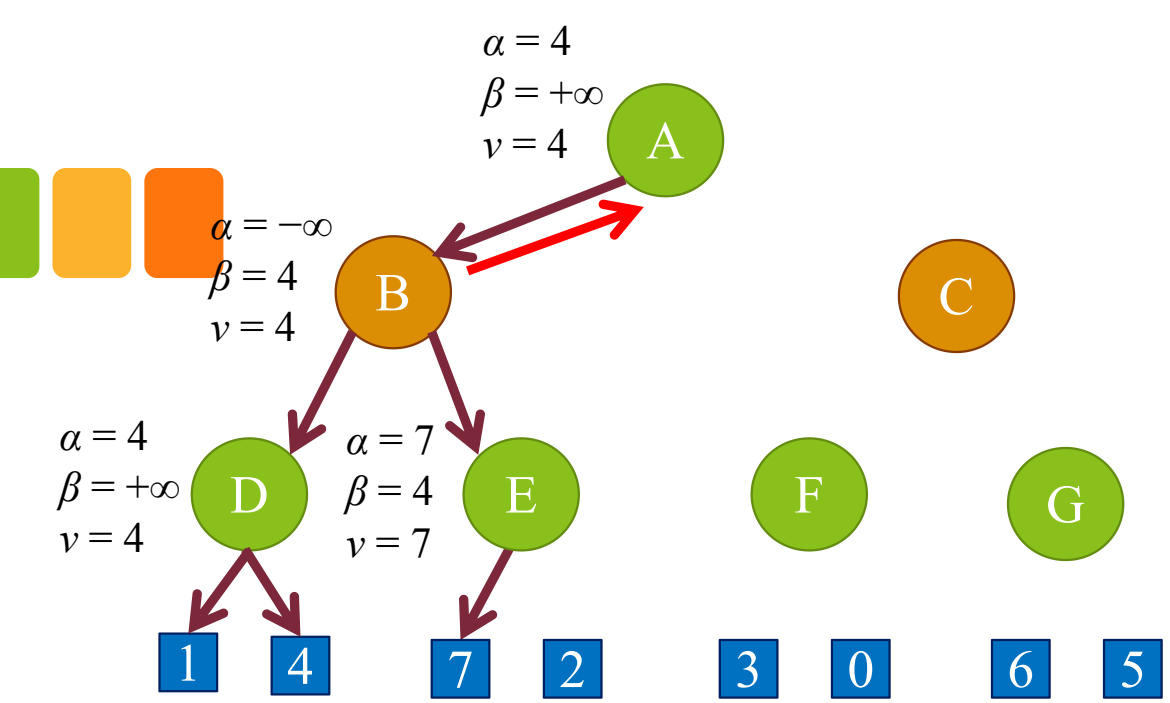

α = −∞
β = +∞
v = −∞
A
α = −∞
β = 4
v = 4
B
C
α = 4
β = +∞
v = 4
D
E
F
G
1
4
7
2
3
0
6
5

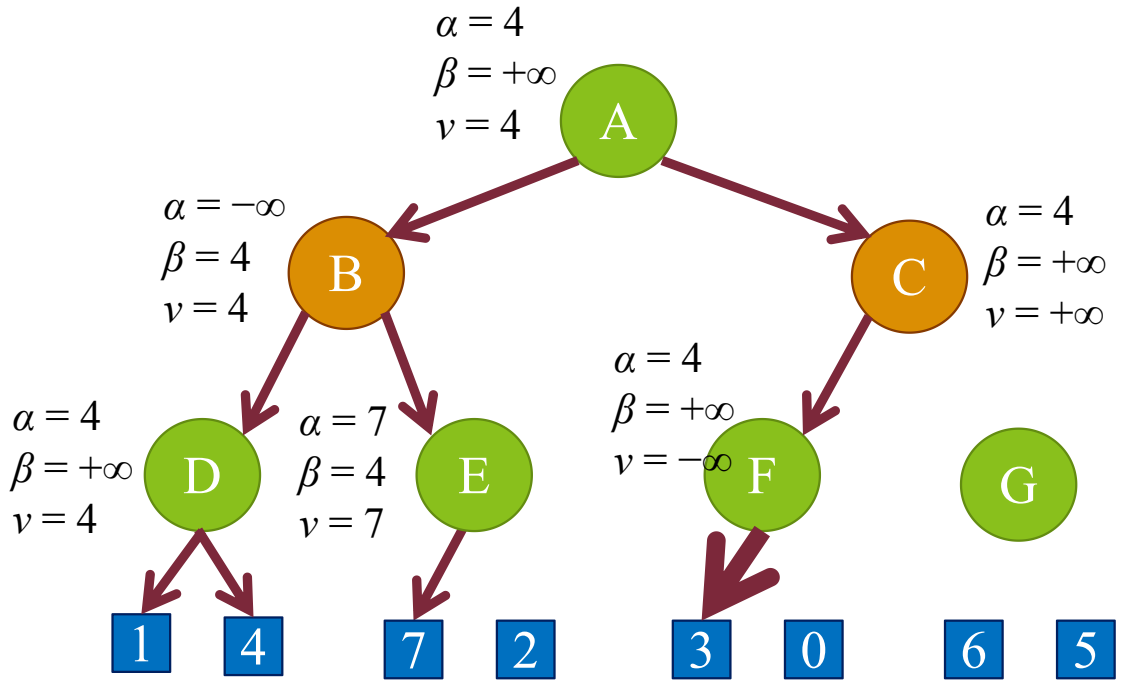

α = −∞
β = +∞
v = −∞
A
α = −∞
β = 4
v = 4
B
C
α = 4
β = +∞
v = 4
D
α = −∞
β = 4
v = −∞
E
F
G
1
4
7
2
3
0
6
5

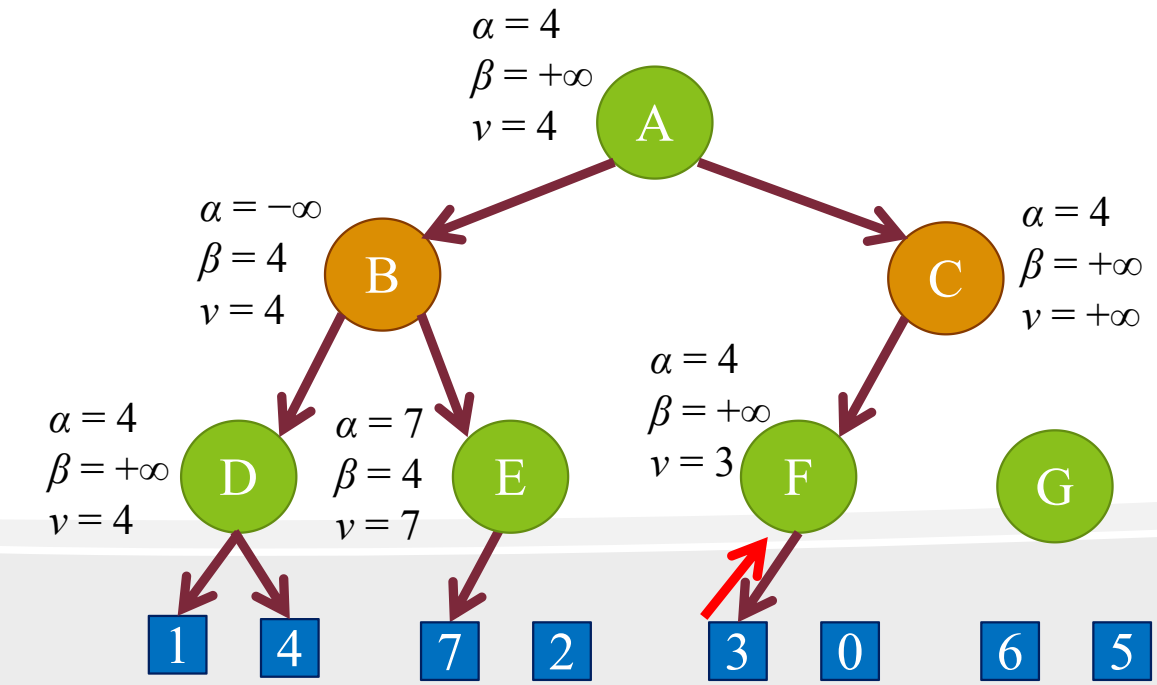

α = −∞
β = +∞
v = −∞
A
α = −∞
β = 4
v = 4
B
C
α = 4
β = +∞
v = 4
D
α = −∞
β = 4
v = −∞
E
F
G
1
4
7
2
3
0
6
5

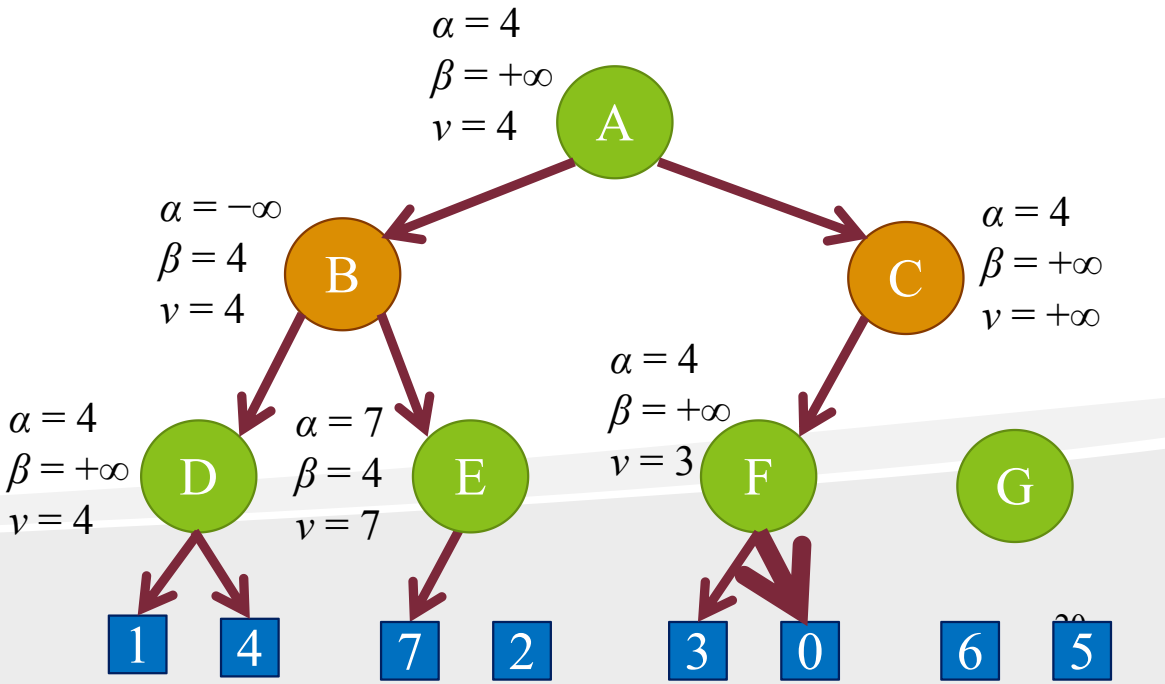

α = −∞
β = +∞
v = −∞
A
α = −∞
β = 4
v = 4
B
C
α = 4
β = +∞
v = 4
D
α = 7
β = 4
v = 7
E
F
G
1
4
7
2
3
0
6
5

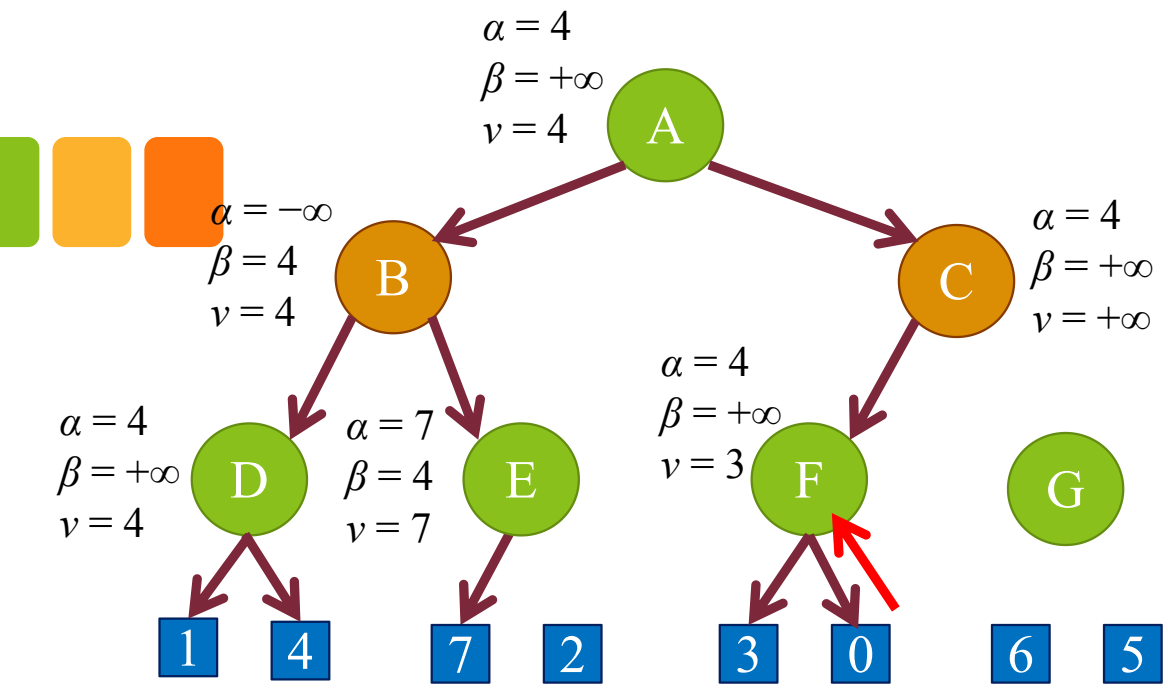

α = −∞
β = +∞
v = −∞
A
α = −∞
β = 4
v = 4
B
C
α = 4
β = +∞
v = 4
D
α = 7
β = 4
v = 7
E
F
G
1
4
7
2
3
0
6
5

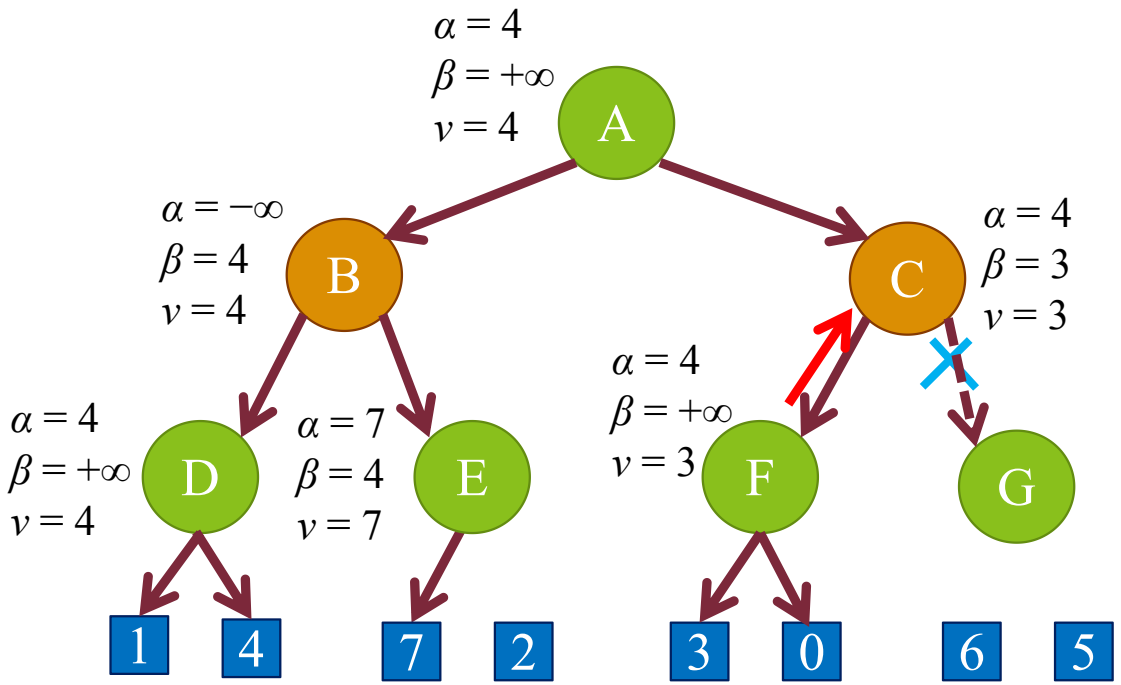

α = 4
β = +∞
v = 4
A
α = −∞
β = 4
v = 4
B
C
α = 4
β = +∞
v = 4
D
α = 7
β = 4
v = 7
E
F
G
1
4
7
2
3
0
6
5

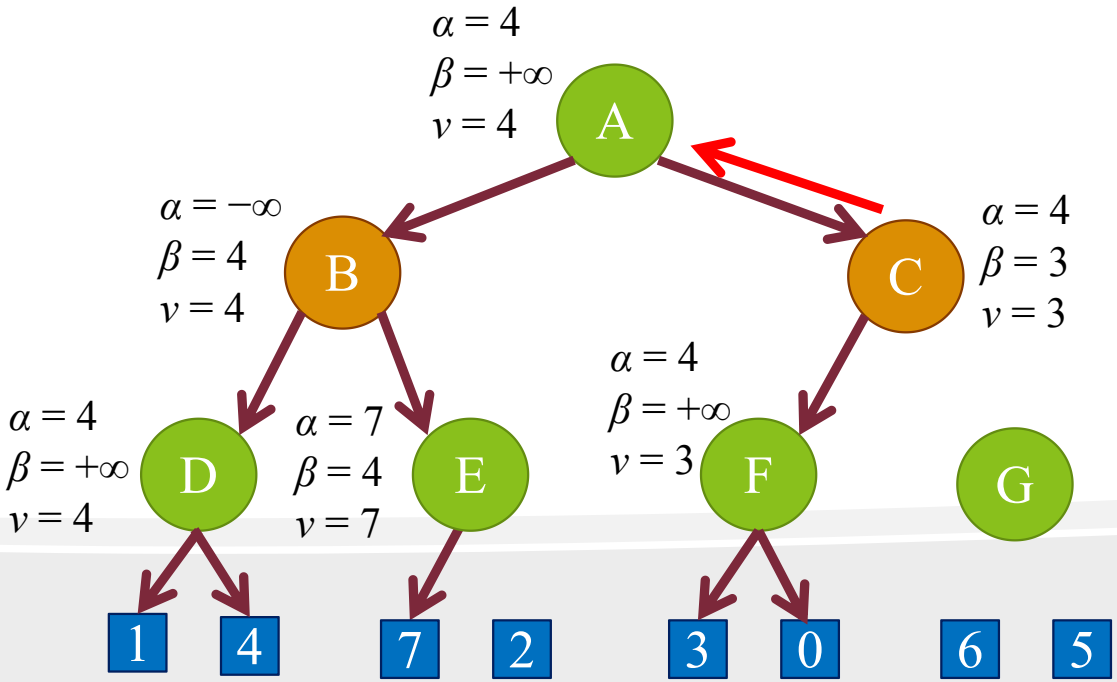

α = 4
β = +∞
v = 4
A
α = −∞
β = 4
v = 4
B
C
α = 4
β = +∞
v = +∞
α = 4
β = +∞
v = 4
D
α = 7
β = 4
v = 7
E
α = 4
β = +∞
v = −∞
F
G
1
4
7
2
3
0
6
5


α = 4
β = +∞
v = 4
A
α = −∞
β = 4
v = 4
B
C
α = 4
β = +∞
v = +∞
α = 4
β = +∞
v = 4
D
α = 7
β = 4
v = 7
E
α = 4
β = +∞
v = 3
F
G
1
4
7
2
3
0
6
5


α = 4
β = +∞
v = 4
A
α = −∞
β = 4
v = 4
B
C
α = 4
β = +∞
v = +∞
α = 4
β = +∞
v = 4
D
α = 7
β = 4
v = 7
E
α = 4
β = +∞
v = 3
F
G
1
4
7
2
3
0
6
5


α = 4
β = +∞
v = 4
A
α = −∞
β = 4
v = 4
B
C
α = 4
β = +∞
v = +∞
α = 4
β = +∞
v = 4
D
α = 7
β = 4
v = 7
E
α = 4
β = +∞
v = 3
F
G
1
4
7
2
3
0
6
5


α = 4
β = +∞
v = 4
A
α = −∞
β = 4
v = 4
B
C
α = 4
β = 3
v = 3
α = 4
β = +∞
v = 4
D
α = 7
β = 4
v = 7
E
α = 4
β = +∞
v = 3
F
G
1
4
7
2
3
0
6
5


α = 4
β = +∞
v = 4
A
α = −∞
β = 4
v = 4
B
C
α = 4
β = 3
v = 3
α = 4
β = +∞
v = 4
D
α = 7
β = 4
v = 7
E
α = 4
β = +∞
v = 3
F
G
1
4
7
2
3
0
6
5

2. Optimal Decisions in Games

Alpha-Beta Pruning

```
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
```

Figure 6.7 The alpha–beta search algorithm.

- We maintain bounds in the variables α and β, and use them to cut off search when a value is outside the bounds.

2. Optimal Decisions in Games
Moving Ordering

- The effectiveness of alpha–beta pruning is highly dependent on the order in which the states are examined.
  - Case-1 (Worst Performance): If the best sequence of actions is on the right of the game tree, then there will be no pruning, and all the nodes will be traversed, leading to even worse performance than the Minimax algorithm because of the overhead of computing α and β.
  - Case-2 (Best Performance): If the best sequence of actions is on the left of the game tree, there will be a lot of pruning, and only about half of the nodes will be traversed to get the optimal result.
- If this could be done perfectly, alpha–beta would need to examine only $O(b^{m/2})$ nodes to pick the best move, instead of $O(b^m)$ for minimax.

3. Heuristic Alpha-Beta Tree Search

- To limit computation time, we can cut off the search early and apply a **heuristic evaluation function** to states, effectively treating nonterminal nodes as if they were terminal.
- In other words, we replace the UTILITY function with EVAL, which estimates a state's utility.
- We also replace the terminal test by a **cutoff test**, which must return true for terminal states, but is otherwise free to decide when to cut off the search, based on the search depth and any property of the state that it chooses to consider.
- That gives us the formula H-MINIMAX(s, d) for the heuristic minimax value of state s at search depth d:

$$\text{H-MINIMAX}(s,d) = \begin{cases} \text{EVAL}(s,\text{MAX}) & \text{if IS-CUTOFF}(s,d) \\ \max_{a \in Actions(s)} \text{H-MINIMAX}(\text{RESULT}(s,a), d+1) & \text{if TO-MOVE}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{H-MINIMAX}(\text{RESULT}(s,a), d+1) & \text{if TO-MOVE}(s) = \text{MIN}. \end{cases}$$

3. Heuristic Alpha-Beta Tree Search

Evaluation Functions

- A heuristic evaluation function EVAL(s, p) returns an *estimate* of the expected utility of state s to player p, just as the heuristic functions return an estimate of the distance to the goal.
- For terminal states: EVAL(s, p) = UTILITY(s, p)
- For nonterminal states: The evaluation must be somewhere between a loss and a win:

$$\text{UTILITY}(loss, p) \leq \text{EVAL}(s, p) \leq \text{UTILITY}(win, p).$$

3. Heuristic Alpha-Beta Tree Search
Evaluation Functions

- What makes for a good evaluation function?
  - First, the computation must not take too long!
  - Second, the evaluation function should be strongly correlated with the actual chances of winning. One might well wonder about the phrase "chances of winning."
  - After all, chess is not a game of chance: we know the current state with certainty, and no dice are involved; if neither player makes a mistake, the outcome is predetermined.
    - But if the search must be cut off at nonterminal states, then the algorithm will necessarily be uncertain about the final outcomes of those states (even though that uncertainty could be resolved with infinite computing resources).

4. Monte Carlo Tree Search

- The game of Go illustrates two major weaknesses of heuristic alpha–beta tree search:
  - First, Go has a branching factor that starts at 361, which means alpha–beta search would be limited to only 4 or 5 ply.
  - Second, it is difficult to define a good evaluation function for Go because material value is not a strong indicator and most positions are in flux until the endgame.
- In response to these two challenges, modern Go programs have abandoned alpha–beta search and instead use a strategy called **Monte Carlo tree search** (MCTS).

4. Monte Carlo Tree Search

- The basic MCTS strategy does not use a heuristic evaluation function. Instead, the value of a state is estimated as the average utility over a number of **simulations** of *complete games starting from the state*.
- A simulation (also called a **playout** or **rollout**) chooses moves first for one player, then for the other, repeating until a terminal position is reached. At that point the rules of the game determine who has won or lost, and by what score.
- For games in which the only outcomes are a win or a loss, "average utility" is the same as "win percentage."

4. Monte Carlo Tree Search

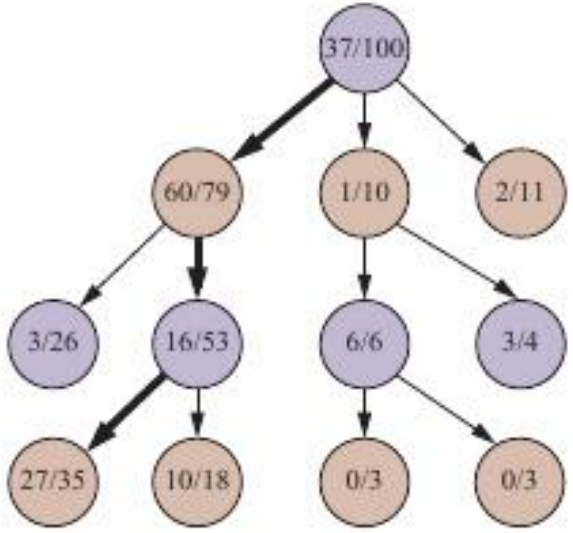
- How do we choose what moves to make during the playout?
  - If we just choose randomly, then after multiple simulations we get an answer to the question “what is the best move if both players play randomly?”
  - For some simple games, that happens to be the same answer as “what is the best move if both players play well?,” but for most games it is not.
  - To get useful information from the playout we need a **playout policy** that biases the moves towards good ones.
    - For Go and other games, playout policies have been successfully learned from self-play by using neural networks. Sometimes game-specific heuristics are used, such as “consider capture moves” in chess or “take the corner square” in Othello.

4. Monte Carlo Tree Search

- Given a playout policy, we next need to decide two things:
  - From what positions do we start the playouts ?
  - How many playouts do we allocate to each position?
- The simplest answer: called **pure Monte Carlo search**
  - Do *N* simulations starting from the current state of the game, and track which of the possible moves from the current position has the highest win percentage.
  - For some stochastic games this converges to optimal play as *N* increases, but for most games it is not sufficient.
- **Selection policy:**
  - It balances two factors **exploration** of states that have had few playouts, and **exploitation** of states that have done well in past playouts, to get a more accurate estimate of their value.

4. Monte Carlo Tree Search

- Monte Carlo tree search does that by maintaining a search tree and growing it on each iteration of the following four steps:
  - **Selection**: Starting at the root of the search tree, we choose a move (guided by the selection policy), leading to a successor node, and repeat that process, moving down the tree to a leaf.

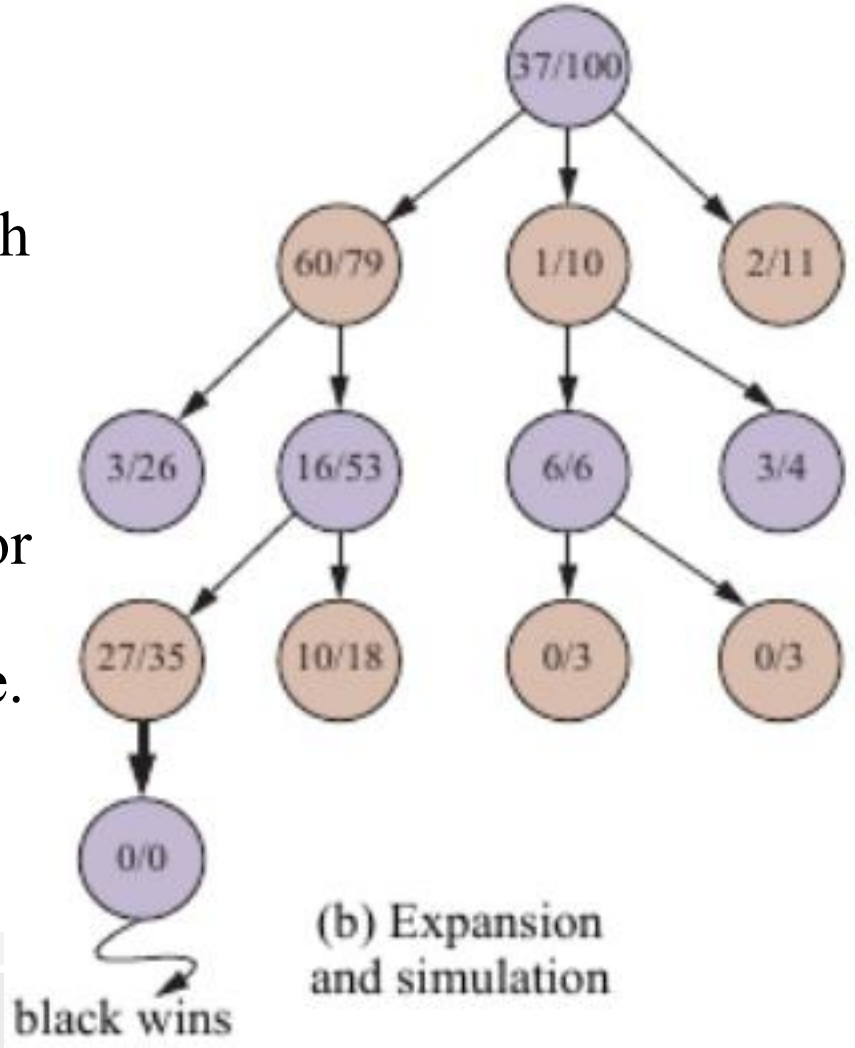


(a) Selection

- Figure 6.10(a) shows a search tree with the root representing a state where *white has just moved*, and white has won 37 out of the 100 playouts done so far.
- The thick arrow shows the selection of a move by black that leads to a node where black has won 60/79 playouts. This is the best win percentage among the three moves, so selecting it is an example of exploitation. But it would also have been reasonable to select the 2/11 node for the sake of exploration—with only 11 playouts, the node still has high uncertainty in its valuation, and might end up being best if we gain more information about it.
- Selection continues on to the leaf node marked 27/35.

4. Monte Carlo Tree Search

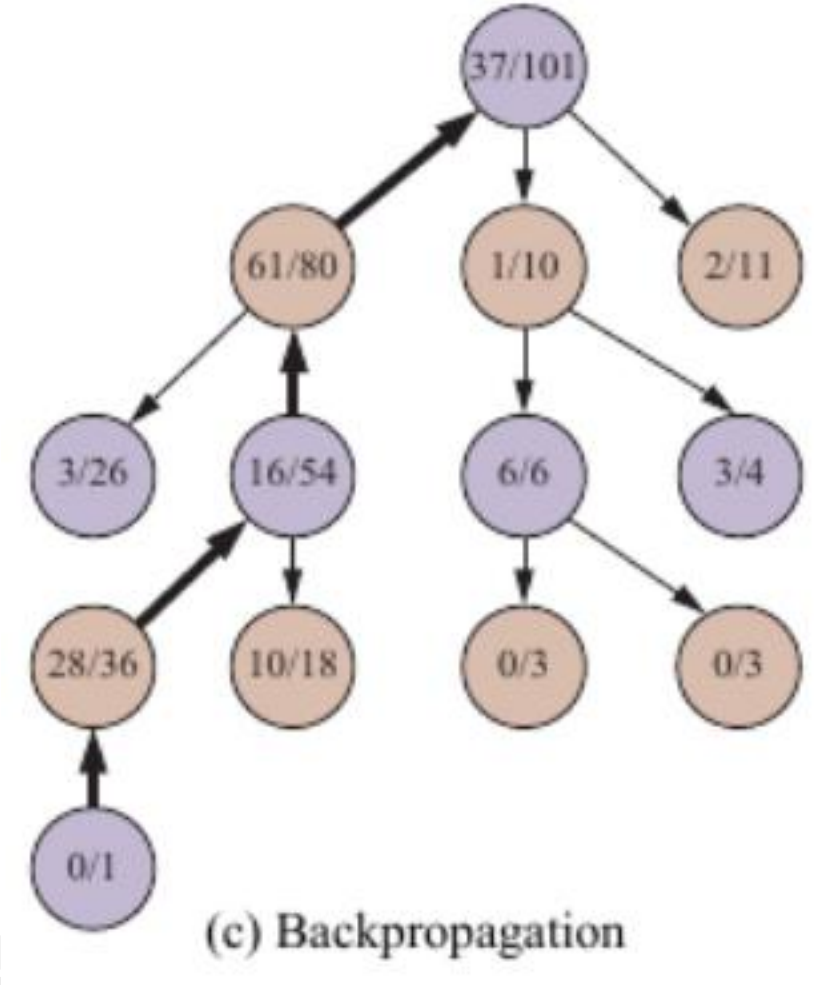
- **Expansion**: We grow the search tree by generating a new child of the selected node; Figure 6.10(b) shows the new node marked with 0/0. (Some versions generate more than one child in this step.)
- **Simulation**: We perform a playout from the newly generated child node, choosing moves for both players according to the playout policy. These moves are not recorded in the search tree. In the figure, the simulation results in a win for black.



4. Monte Carlo Tree Search

- **Back-propagation**: We now use the result of the simulation to update all the search tree nodes going up to the root. Since black won the playout, black nodes are incremented in both the number of wins and the number of playouts, so 27/35 becomes 28/36 and 60/79 becomes 61/80. Since white lost, the white nodes are incremented in the number of playouts only, so 16/53 becomes 16/54 and the root 37/100 becomes 37/101.

- We repeat these four steps either for a set number of iterations, or until the allotted time has expired, and then return the move with the highest number of playouts.



(c) Backpropagation

4. Monte Carlo Tree Search

```
function MONTE-CARLO-TREE-SEARCH(state) returns an action
    tree ← NODE(state)
    while IS-TIME-REMAINING() do
        leaf ← SELECT(tree)
        child ← EXPAND(leaf)
        result ← SIMULATE(child)
        BACK-PROPAGATE(result, child)
    return the move in ACTIONS(state) whose node has highest number of playouts
```

Figure 6.11 The Monte Carlo tree search algorithm. A game tree, *tree*, is initialized, and then we repeat a cycle of SELECT / EXPAND / SIMULATE / BACK-PROPAGATE until we run out of time, and return the move that led to the node with the highest number of playouts.

4. Monte Carlo Tree Search

- One very effective selection policy is called “upper confidence bounds applied to trees” or UCT. The policy ranks each possible move based on an upper confidence bound formula called UCB1. For a node *n*, the formula is:

$$UCB1(n) = \frac{U(n)}{N(n)} + C \times \sqrt{\frac{\log N(\text{PARENT}(n))}{N(n)}}$$