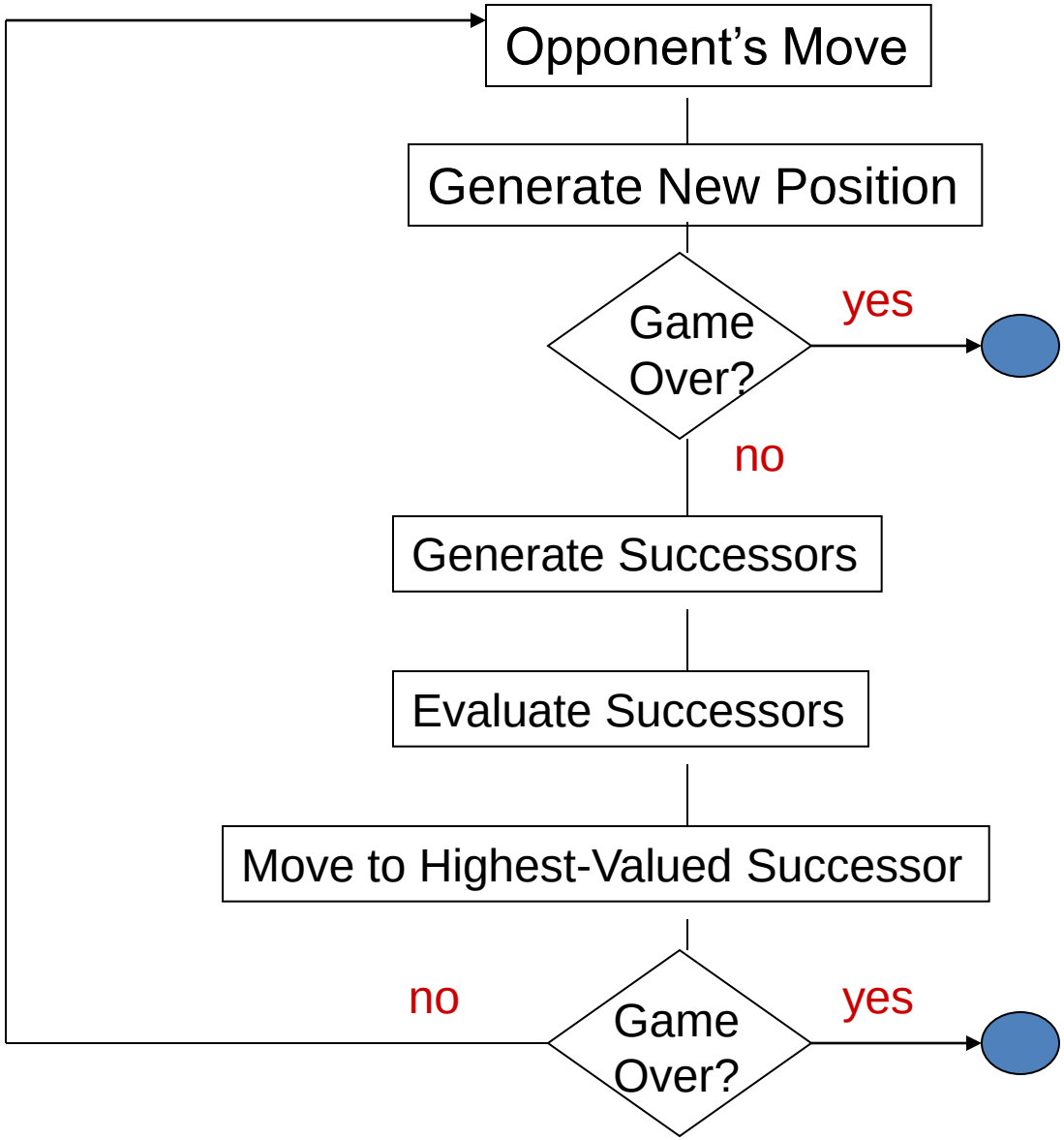
Unit 5

Building games with AI

Minimax and Alpha-Beta

When can we use Minimax?

- Game Properties required for Minimax
  - Two players
  - No chance (such as coin flipping)
  - Perfect information
    - No hidden cards or hidden Chess pieces…
- Non-Minimax Games
  - Poker (or any game that involves bluffing or somehow outwitting your opponent)
  - Arcade Games…


Opponent's Move
Generate New Position
Game Over?
yes
no
Generate Successors
Evaluate Successors
Move to Highest-Valued Successor
Game Over?
no
yes

Quiz

Supervised learning is not suitable due to

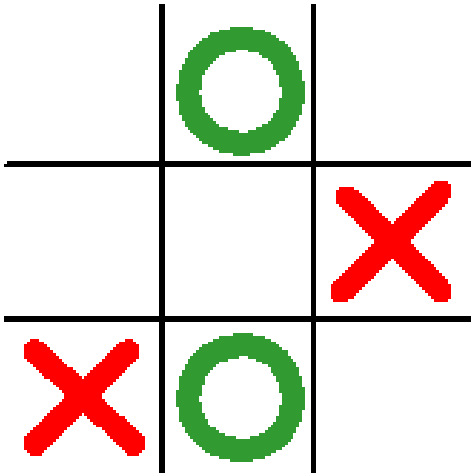
A. Tagged corpus is hard to generate
B. Tagged corpus is easily available
C. Does not require input
D. All of these

Quiz

Supervised learning is not suitable due to

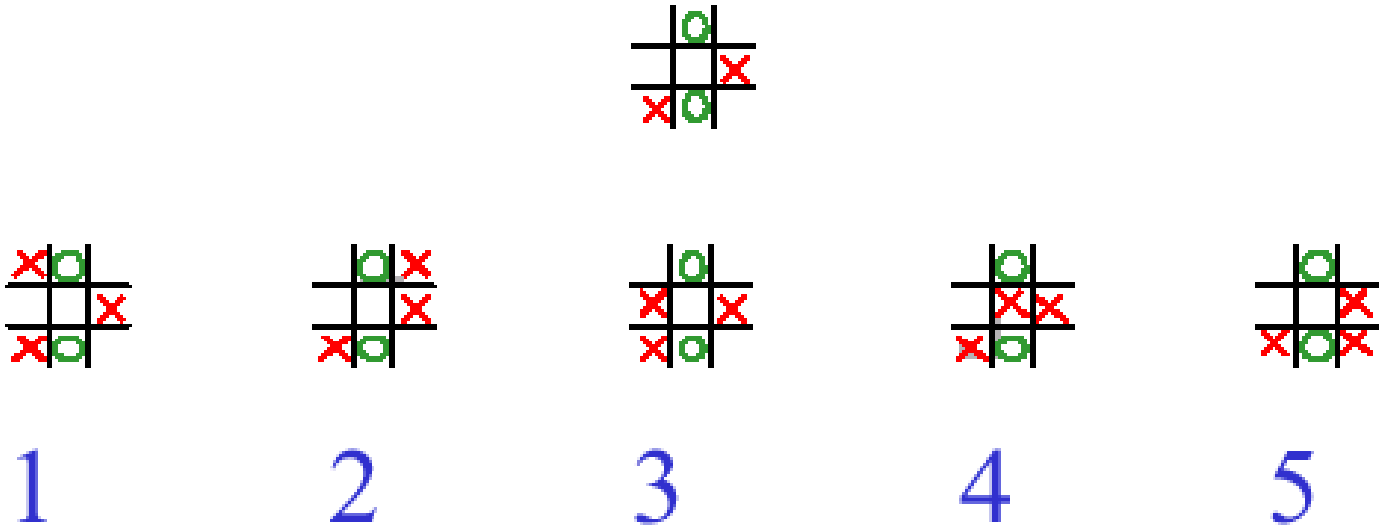
A. Tagged corpus is hard to generate
B. Tagged corpus is easily available
C. Does not require input
D. All of these

Illustration : Tic-Tac-Toe



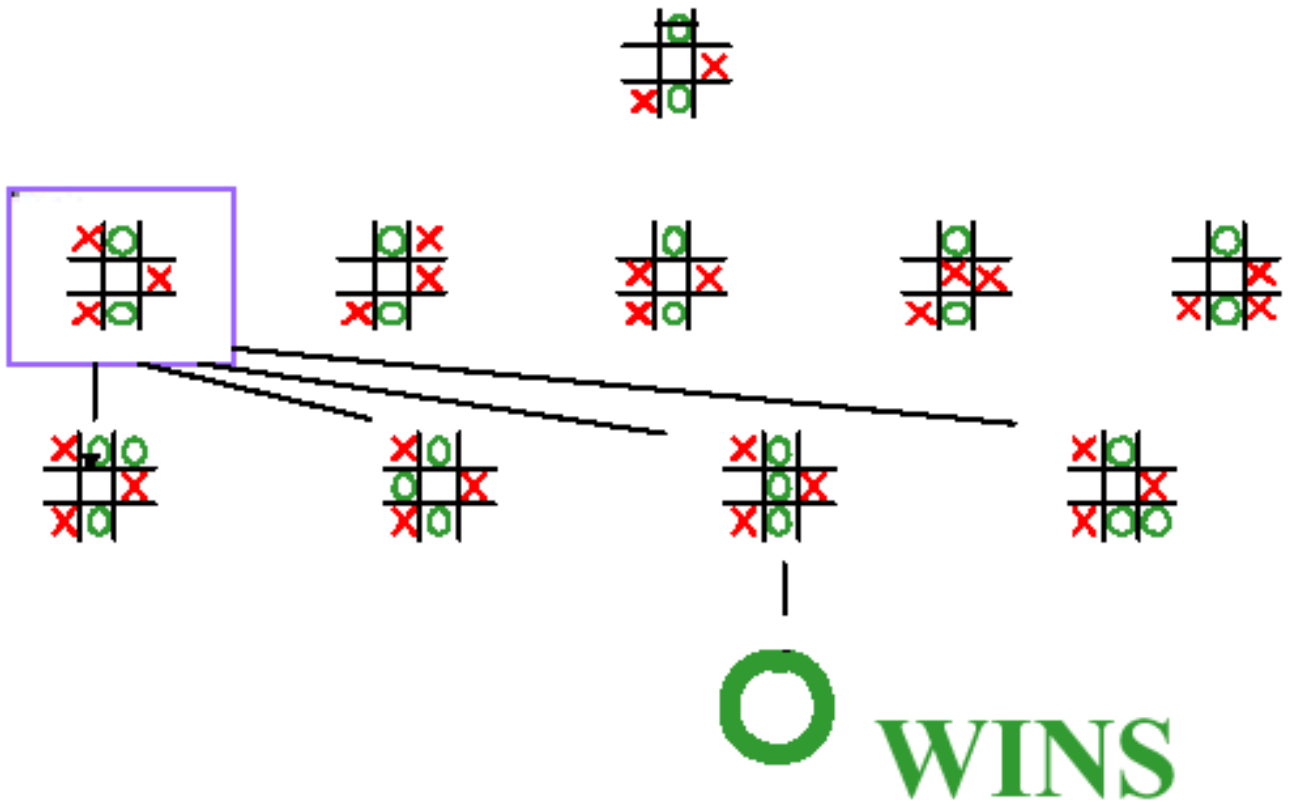
We are playing X, and it is now our turn.

Let’s write out all possibilities



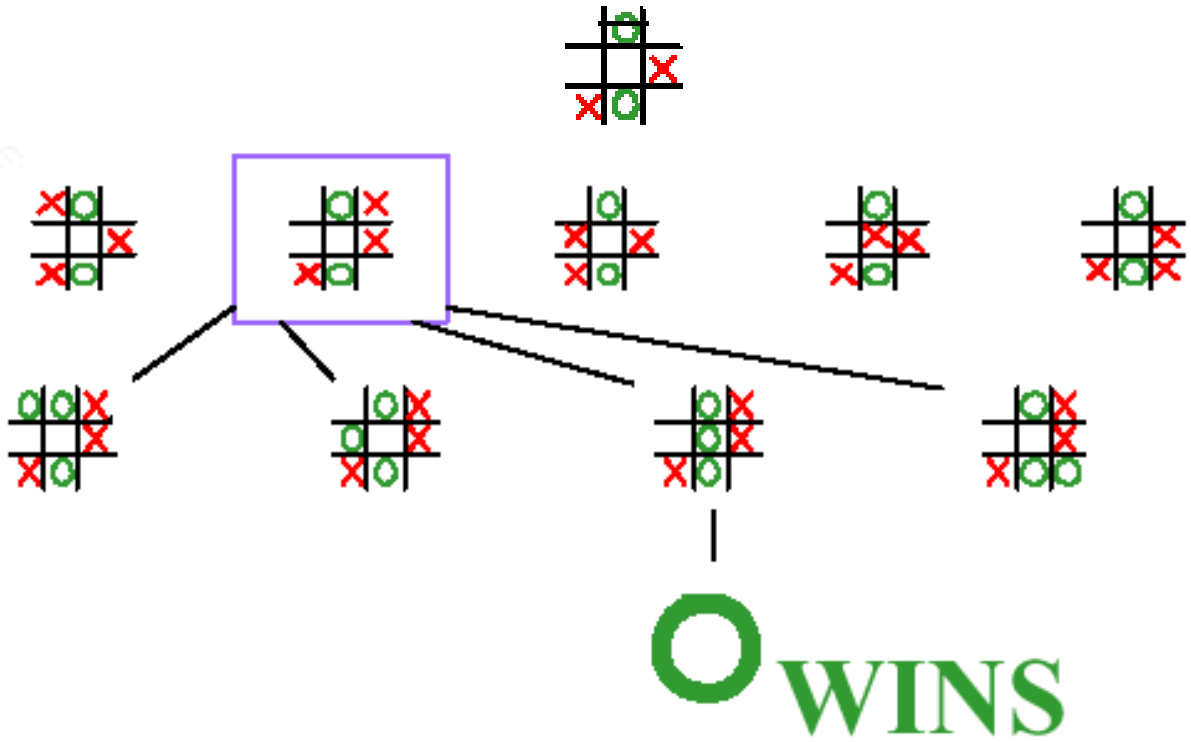
Each number represents a position after each legal move we have.

Now let's look at their options



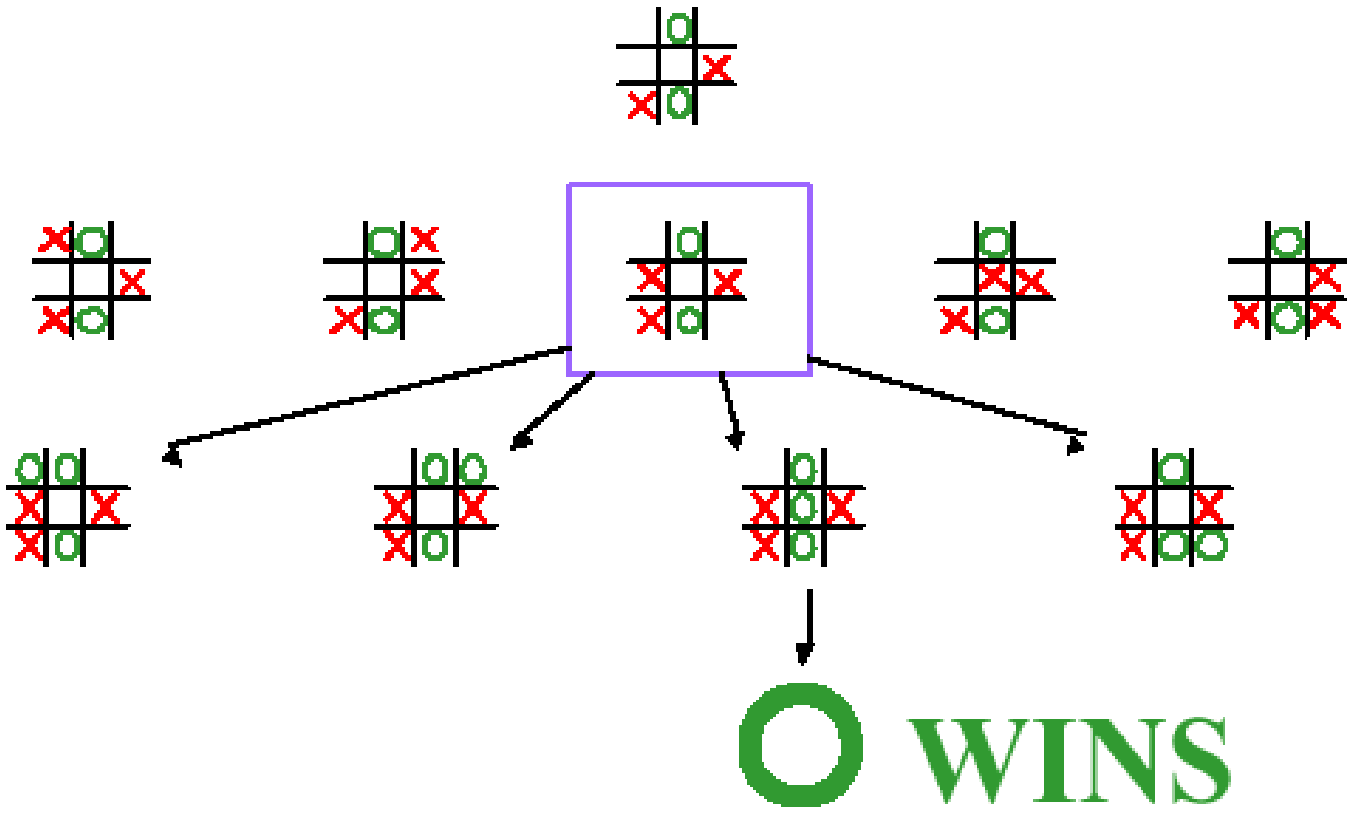
Here we are looking at all of the opponent responses to the first possible move we could make.

Now let's look at their options



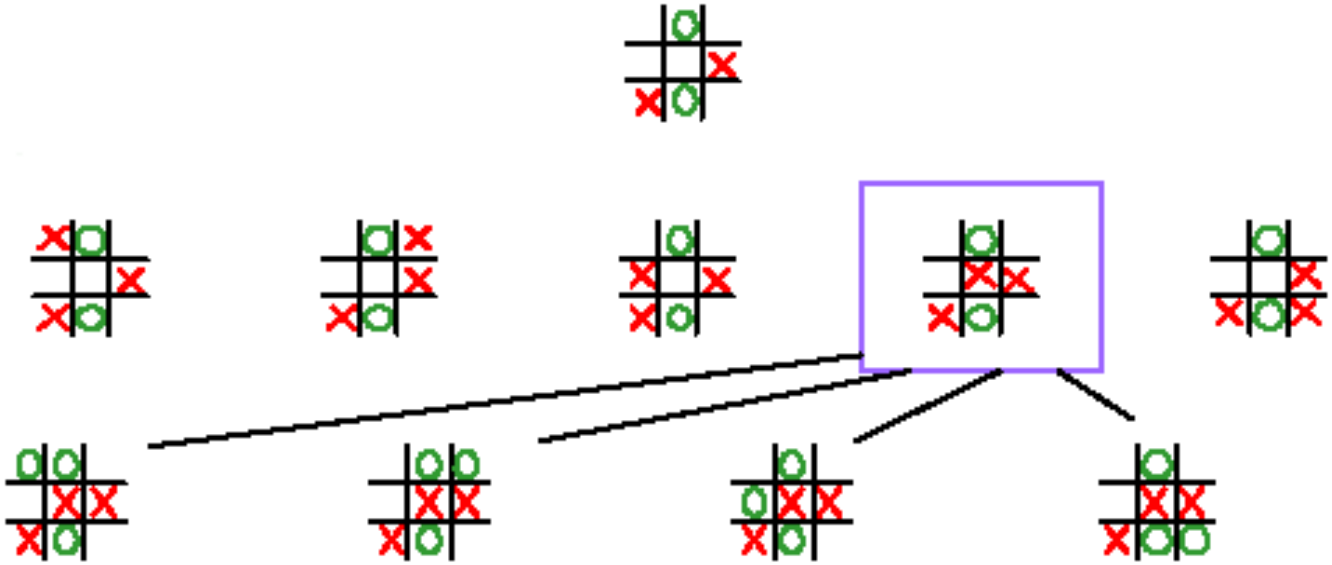
Opponent options after our second possibility. Not good again...

Now let's look at their options



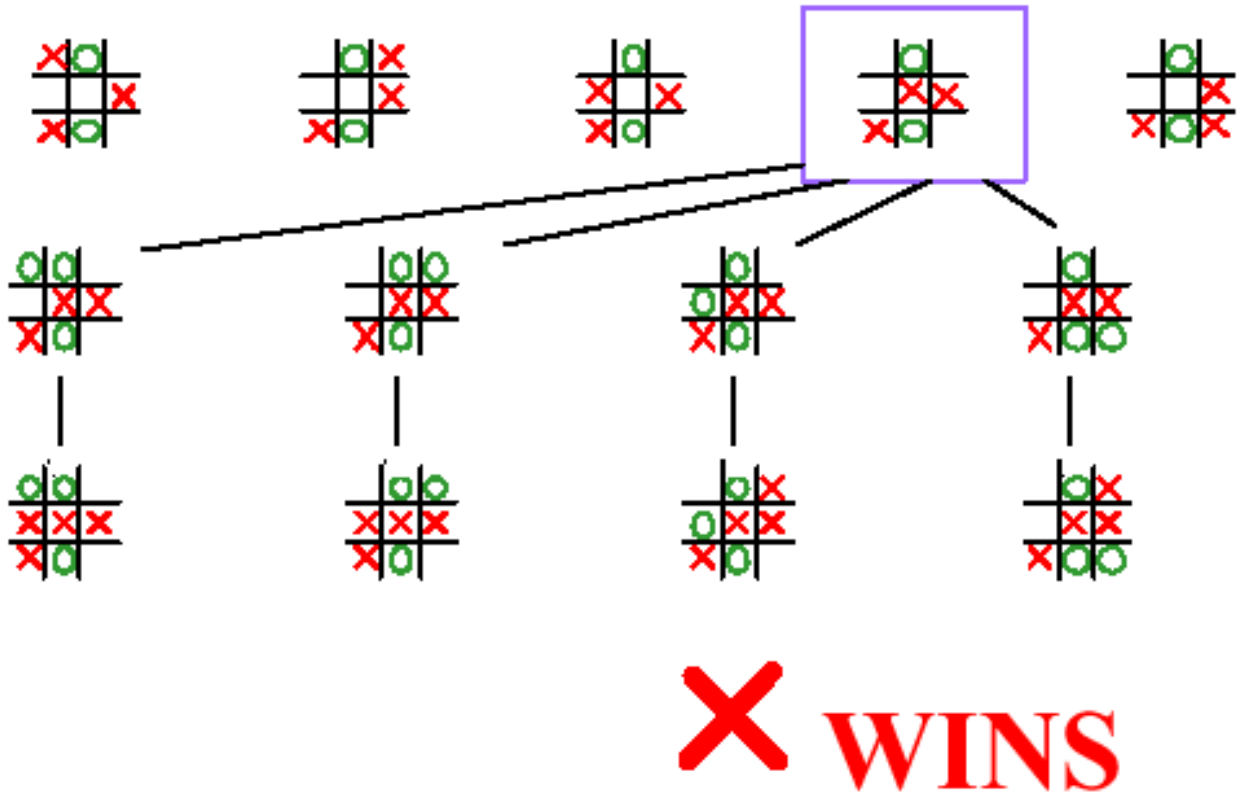
Struggling…

More interesting case



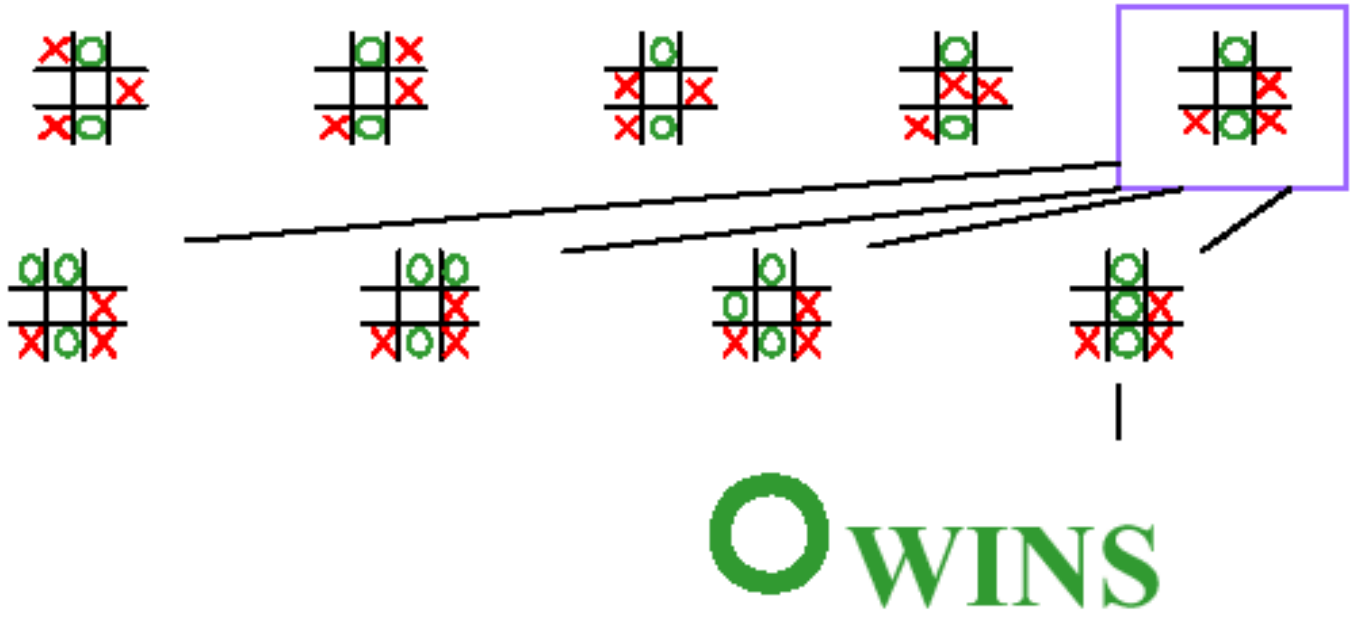
Now they don’t have a way to win on their next move. So now we have to consider our responses to their responses.

Our options



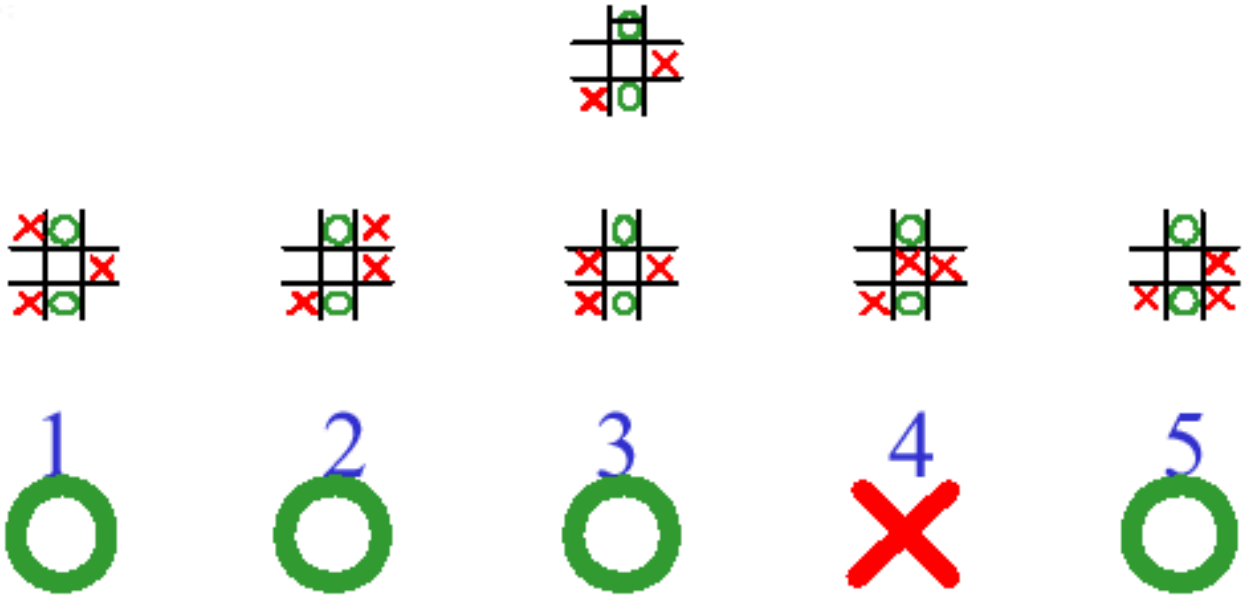
We have a win for any move they make.
So the original position in purple is an X win.

Finishing it up…



They win again if we take our fifth move.

Summary of the Analysis



So which move should we make?

Looking closer at the process

- Traverse the “game tree”.
  - Enumerate all possible moves at each node. The children of that node are the positions that result from making each move. A **leaf** is a position that is **won** or **drawn** for some side.
- Make the assumption that we pick the best move for us, and the opponent picks the best move for him (causes most damage to us)
- Pick the move that maximizes the minimum amount of success for our side.
- This process is known as the Minimax algorithm.

Game Playing and AI

Why would game playing be a good problem for AI research?

- game playing is non-trivial
  - players need “human-like” intelligence
  - games can be very complex (e.g. chess, go)
  - requires decision making within limited time
- games often are:
  - well-defined and repeatable
  - easy to represent
  - fully observable and limited environments
- can directly compare humans and computers

Game Playing as Search

- Consider two-player, turn-taking, board games
  - e.g., tic-tac-toe, checkers, chess
  - adversarial, zero-sum
  - board configs: unique arrangements of pieces

- Representing these as search problem:
  - **states:** board configurations
  - **edges:** legal moves
  - **initial state:** start board configuration
  - **goal state:** winning/terminal board configuration

Game Playing as Search: Complexity

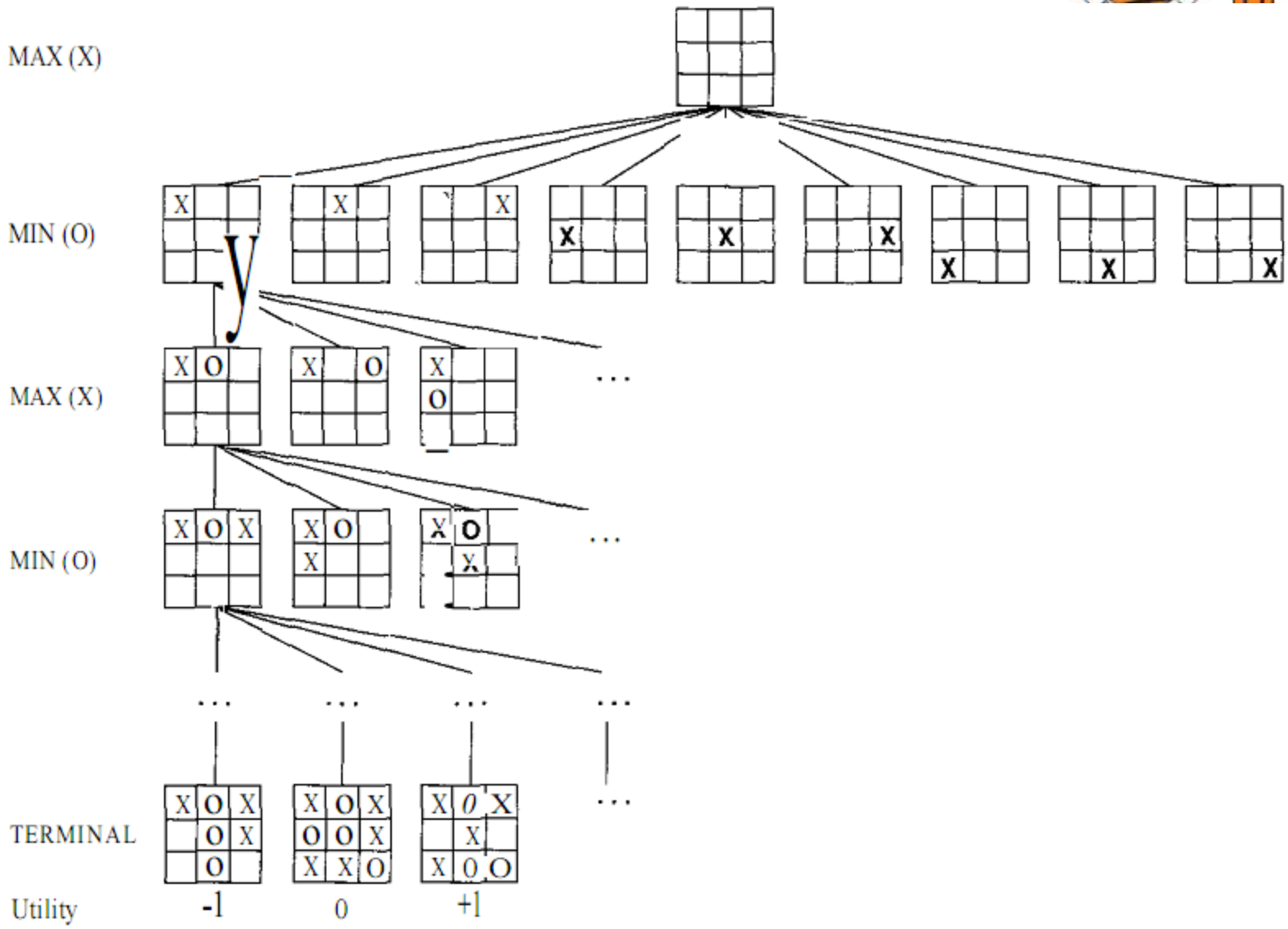
- Assume the opponent's moves *can* be predicted given the computer's moves.

- How complex would search be in this case?
  - worst case: $O(b^d)$ $b$ranching factor, $d$epth
  - **Tic-Tac-Toe:** ~5 legal moves, 9 moves max game
    - 5^9 = 1,953,125 states
  - **Chess:** ~35 legal moves, ~100 moves per game
    - 35^{100} ~10^{154} states, but only ~10^{40} legal states

MiniMax Algorithm

A game can be formally defined as a kind of search problem with the following components:

- **The initial state,** which includes the board position and an indication of whose move it is.
- **A set** of **operators,** which define the legal moves that a player can make.
- A **terminal test,** which determines when the game is over. States where the game has ended are called **terminal states.**
- A **utility function** (also called a **payoff function),** which gives a numeric value for the outcome of a game. In chess, the outcome is a win, loss, or draw, which we can represent by the values +1, −1, or 0. Some games have a wider variety of possible outcomes; for example, the payoffs in backgammon range from +192 to −192.


MAX (X)
MIN (O)
MAX (X)
MIN (O)
TERMINAL
Utility
-1
0
+1

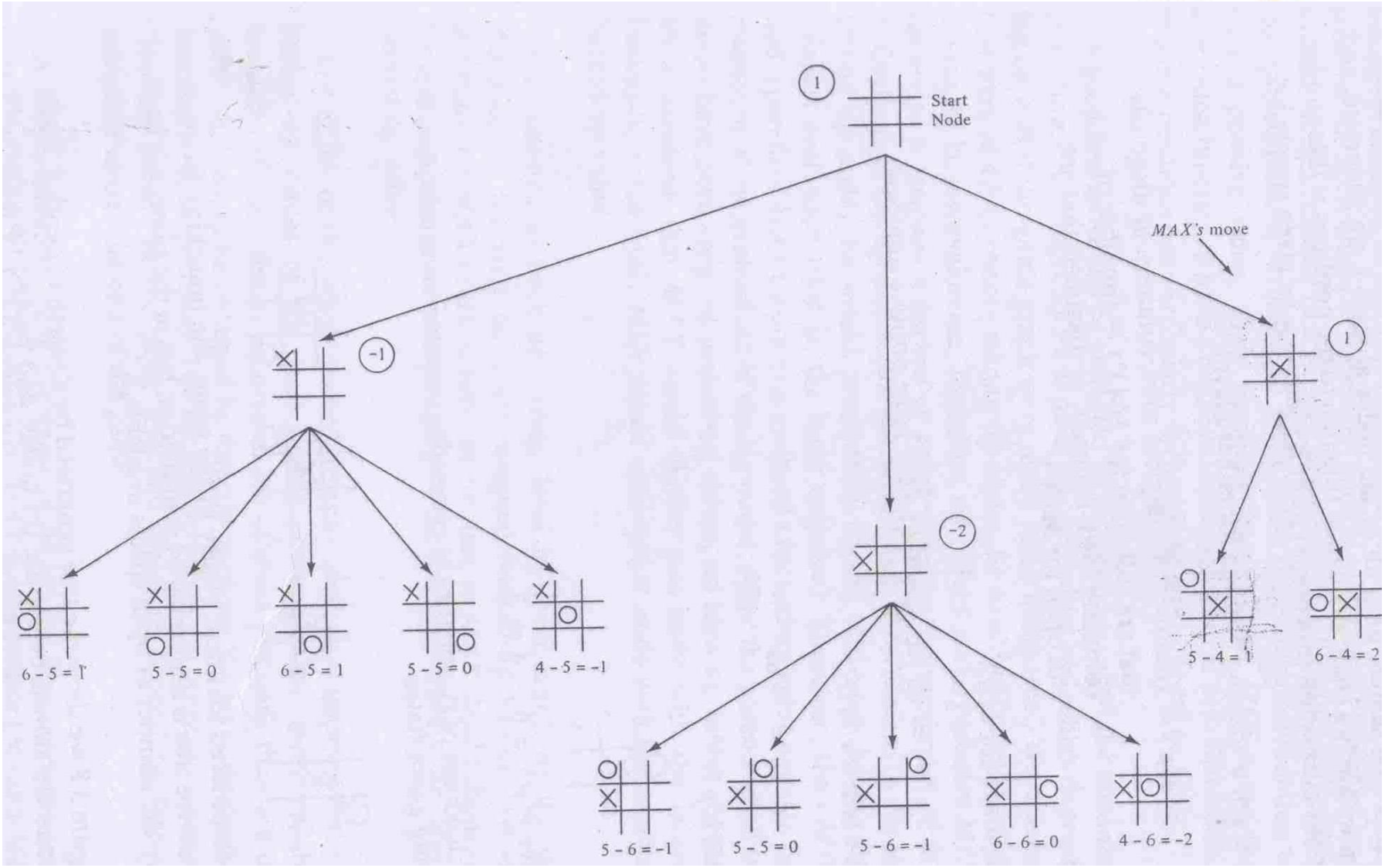
Tic-Tac-Toe

X	O	

e(p) = 6 - 5 = 1

- **Initial State**: Board position of 3x3 matrix with 0 and X.
- **Operators:** Putting 0's or X's in vacant positions alternatively
- **Terminal test:** Which determines game is over
- **Utility function**:

e(p) = (No. of complete rows, columns or diagonals are still open for player) – (No. of complete rows, columns or diagonals are still open for opponent)


1
Start
Node
MAX's move
-1
1
-2
6 – 5 = 1
5 – 5 = 0
6 – 5 = 1
5 – 5 = 0
4 – 5 = -1
5 – 4 = 1
6 – 4 = 2
5 – 6 = -1
5 – 5 = 0
5 – 6 = -1
6 – 6 = 0
4 – 6 = -2

Quiz

I got late. There was fatal accident happened on road.

A. Parts of action
B. Parts of entity
C. Elements of sets
D. Causal chain

Quiz

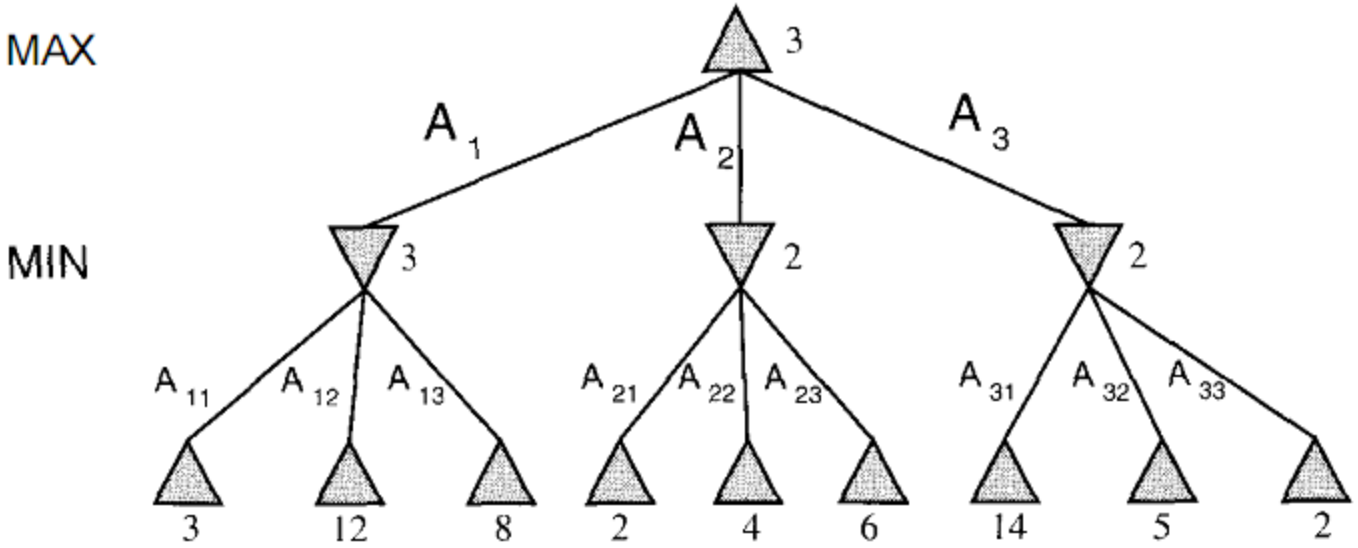
I got late. There was fatal accident happened on road.

A. Parts of action

B. Parts of entity

C. Elements of sets

D. Causal chain



Minmax(s)=

Utility(s)		**if terminal(s)**
$\max_{a \in action(s)}$	**minmax(result(s,a)**	**if player(s)=max**
$\min_{a \in action(s)}$	**minmax(result(s,a)**	**if player(s)=min**

Process of MINIMAX Algorithm

The **minimax** algorithm is designed to determine the optimal strategy for MAX, and thus to decide what the best first move is. The algorithm consists of five steps:

- Generate the whole game tree, all the way down to the terminal states.
- Apply the utility function to each terminal state to get its value.
- Use the utility of the terminal states to determine the utility of the nodes one level higher up in the search tree. Consider the leftmost three leaf nodes in Figure 5.2. In the V node above it, MIN has the option to move, and the best MIN can do is choose A_{11}, which leads to the minimal outcome, 3. Thus, even though the utility function is not immediately applicable to this V node, we can assign it the utility value 3, under the assumption that MIN will do the right thing. By similar reasoning, the other two V nodes are assigned the utility value 2.
- Continue backing up the values from the leaf nodes toward the root, one layer at a time.
- Eventually, the backed-up values reach the top of the tree; at that point, MAX chooses the move that leads to the highest value. In the topmost A node of Figure 5.2, MAX has a choice of three moves that will lead to states with utility 3, 2, and 2, respectively. Thus, MAX's best opening move is $A1_1$. This is called the **mininiax decision,** because it maximizes the utility under the assumption that the opponent will play perfectly to minimize it.

Minimax: Algorithm Complexity

Assume all terminal states are at depth d

☞ Space complexity?

depth-first search, so $\boldsymbol{O(bd)}$

☞ Time complexity?

given branching factor b, so $\boldsymbol{O(b^d)}$

✱ *Time complexity is a major problem! Computer typically only has a finite amount of time to make a move.*

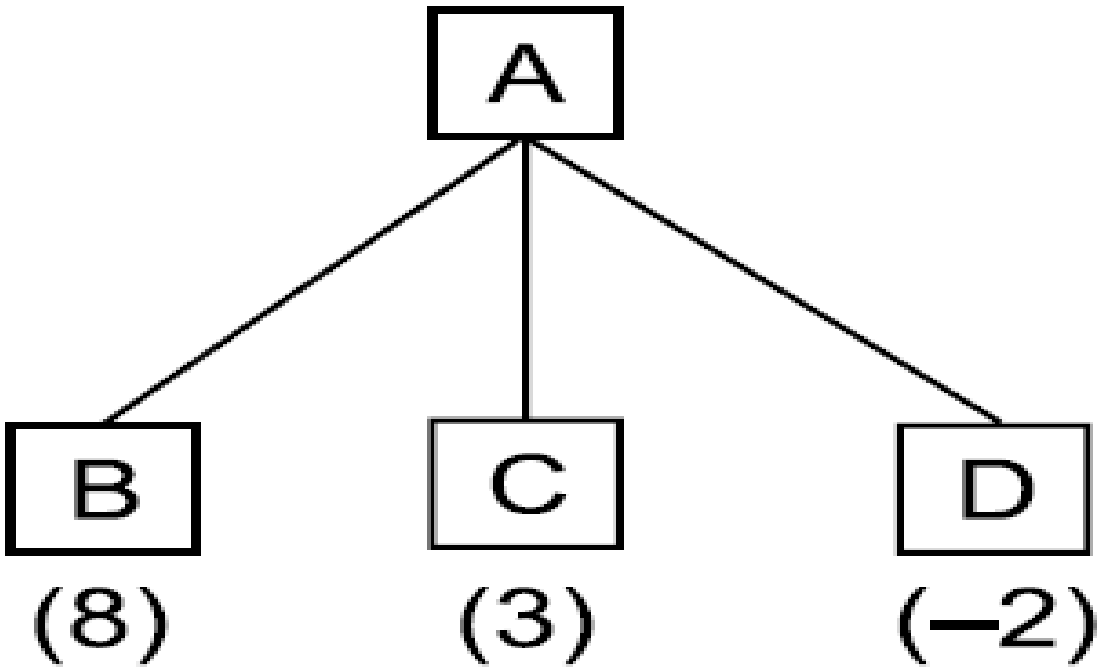
Minimax: Algorithm Complexity

- Direct minimax algo. is impractical in practice
  - instead do depth-limited search to ply (depth) ***m***

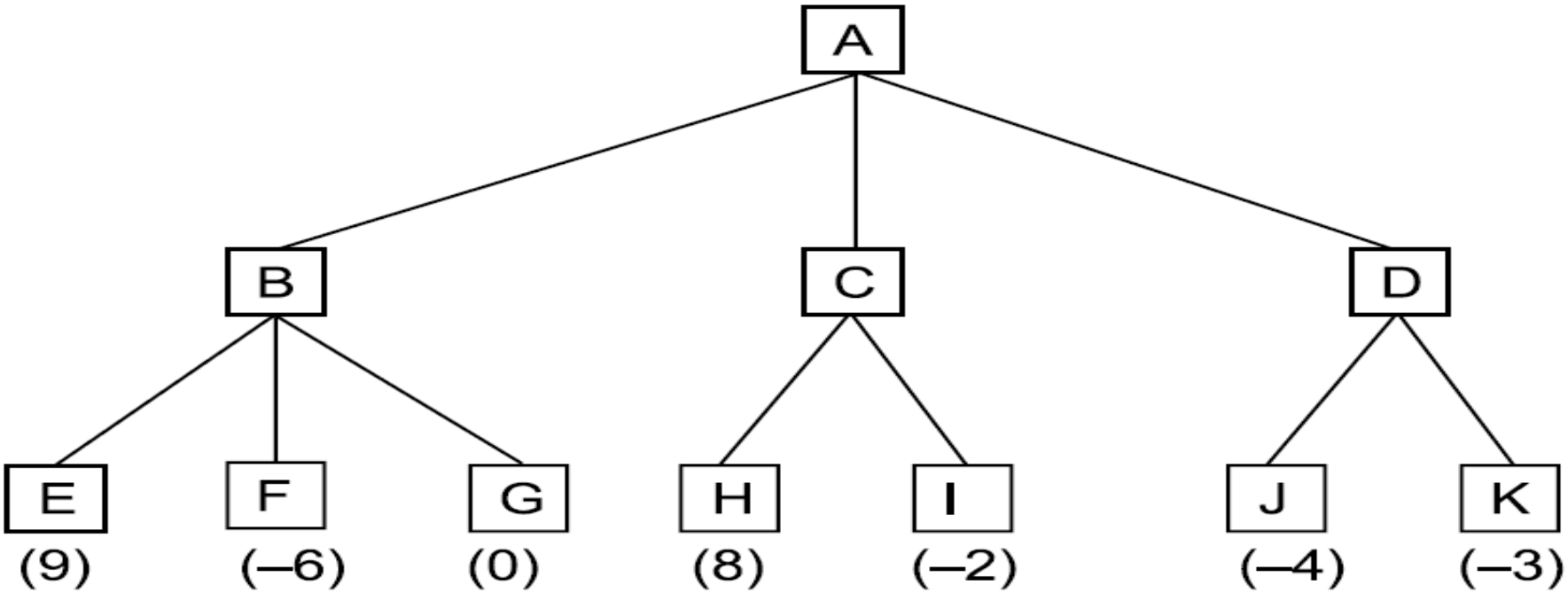
☞ What's the problem for stopping at *any* ply?
  - evaluation defined only for terminal states
  - we need to know the value of non-terminal states

✱*Static board evaluator (SBE) function uses heuristics to estimate the value of non-terminal states.*

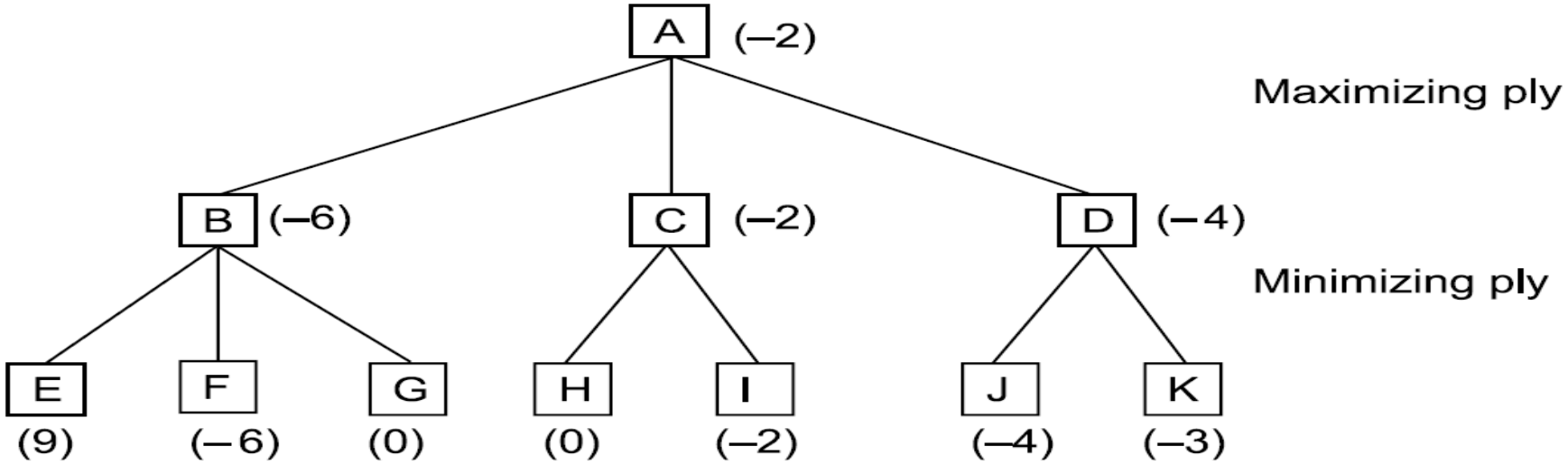
One – Ply Search



Two – Ply Search



Backing Up the Values of a Two – Ply Search



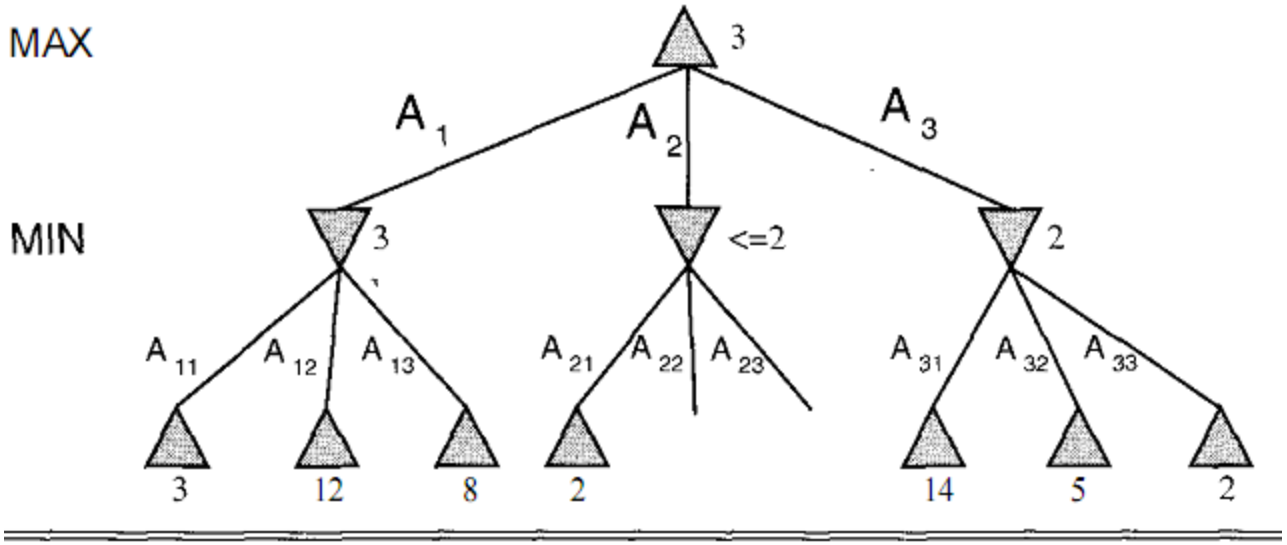
```
function MINIMAX-DECISION(state) returns an action
    v ← MAX-VALUE(state)
    return the action in SUCCESSORS(state) with value v

function MAX-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← −∞
    for a, s in SUCCESSORS(state) do
        v ← MAX(v, MIN-VALUE(s))
    return v

function MIN-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← ∞
    for a, s in SUCCESSORS(state) do
        v ← MIN(v, MAX-VALUE(s))
    return v
```

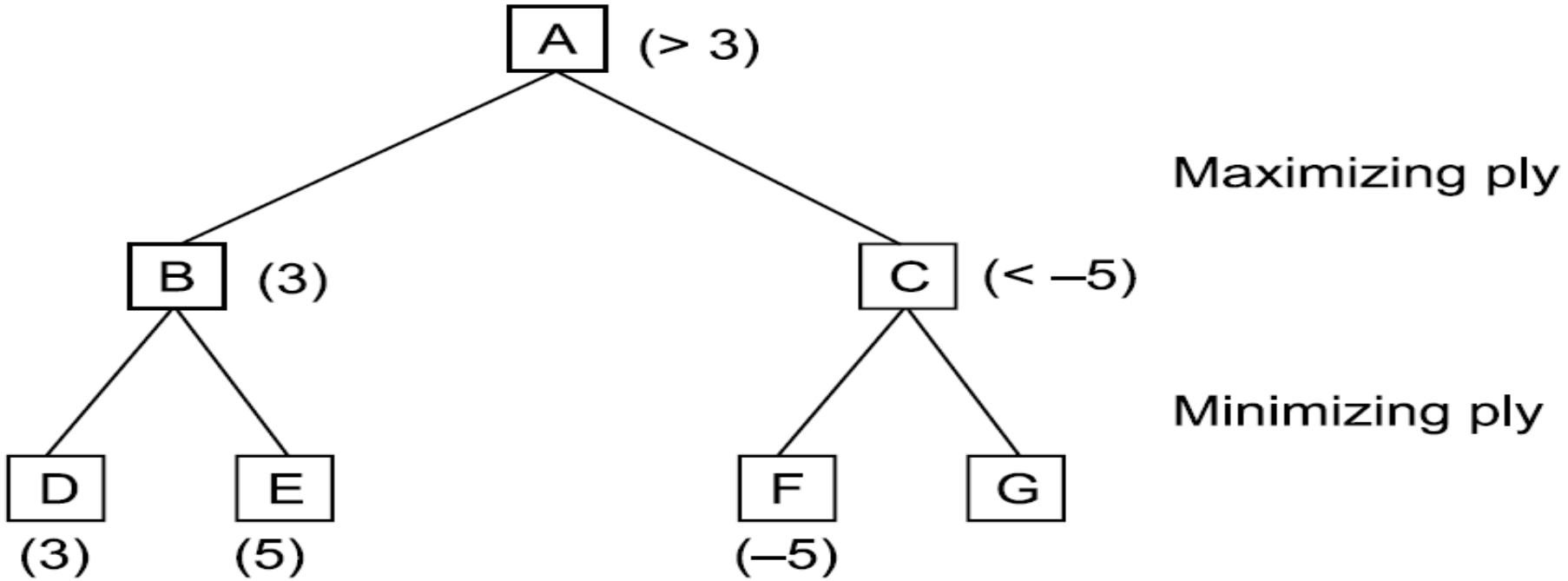
ALPHA-BETA PRUNING

Fortunately, it is possible to compute the correct minimax decision without looking at every node in the search tree. The process of eliminating a branch of the search tree from consideration without examining it is called **pruning** the search tree. The particular technique we will examine is called **alpha-beta pruning.** When applied to a standard minimax tree, it returns the same move as minimax would, but prunes away branches that cannot possibly influence the final decision.

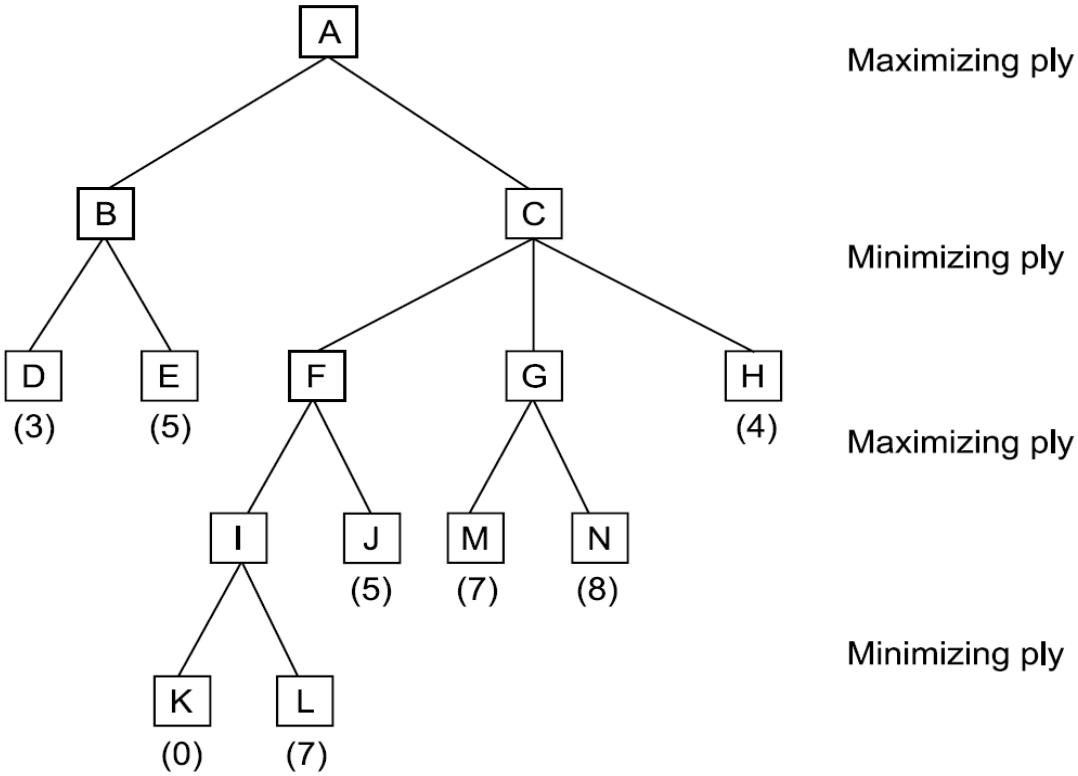


The two-ply game tree as generated by alpha-beta.

Alpha – Beta Pruning



Alpha and Beta Cutoffs



The general principle is this. Consider a node *n* somewhere in the tree (see Figure 5.7), such that Player has a choice of moving to that node. 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.*

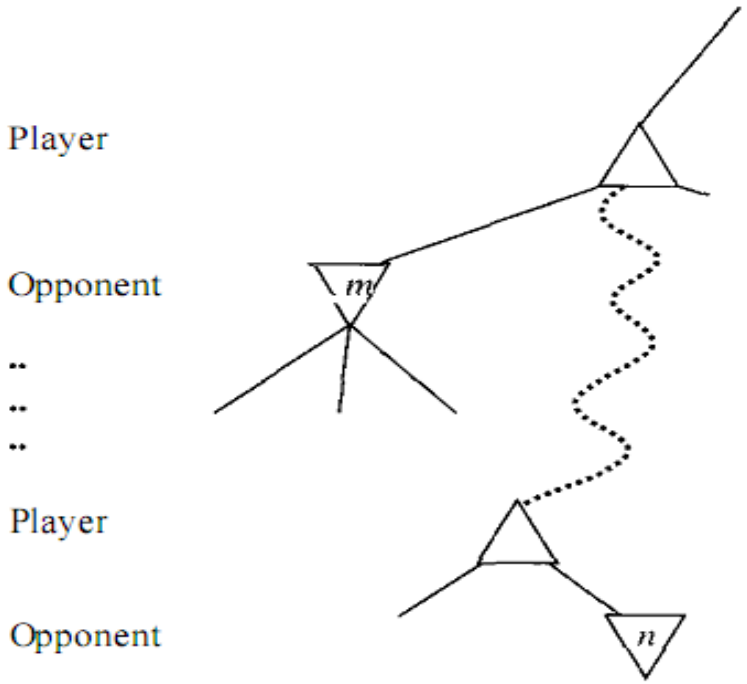


Figure 5.7 Alpha-beta pruning: the general case. If *m* is better than *n* for Player, we will never get to *n* in play.

The α-β algorithm

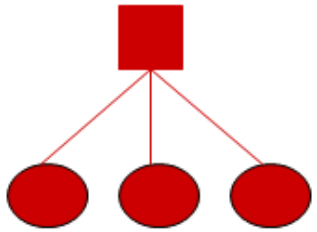
function ALPHA-BETA-SEARCH(*state*) **returns** *an action*
  inputs: *state*, current state in game

  $v \leftarrow$ MAX-VALUE($state, -\infty, +\infty$)
  return the *action* in SUCCESSORS(*state*) with value v

---

function MAX-VALUE($state, \alpha, \beta$) **returns** *a utility value*
  inputs: *state*, current state in game
    α, the value of the best alternative for MAX along the path to *state*
    β, the value of the best alternative for MIN along the path to *state*

  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow -\infty$
  for a, s in SUCCESSORS(*state*) **do**
    $v \leftarrow$ MAX(v, MIN-VALUE(s, α, β))
    if $v \geq \beta$ **then return** v cutoff
    $\alpha \leftarrow$ MAX(α, v)
  return v



The α-β algorithm

function MIN-VALUE($state, \alpha, \beta$) **returns** *a utility value*

inputs: *state*, current state in game

α, the value of the best alternative for MAX along the path to *state*

β, the value of the best alternative for MIN along the path to *state*

if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)

$v \leftarrow +\infty$

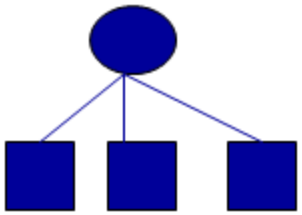
for a, s in SUCCESSORS(*state*) **do**

  $v \leftarrow$ MIN(v, MAX-VALUE(s, α, β))

  if $v \leq \alpha$ **then return** v cutoff

  $\beta \leftarrow$ MIN(β, v)

return v



Alpha Beta Pruning

- Does Alpha Beta ever return a different root value than Minimax?
  - No! Alpha Beta does the same thing Minimax does, except it is able to detect parts of the tree that make no difference. Because it can detect this it doesn't evaluate them.
- What is the speedup?
  - The optimal Alpha Beta search tree is $O(b^{d/2})$ nodes or the square root of the number of nodes in the regular Minimax tree.
  - The speedup is *greatly* dependent on the order in which you consider moves at each node. Why?

Quiz

Which is false

A. Production rules are used in expert system
B. Three player game uses minmax
C. Minmax explores all nodes
D. All of these

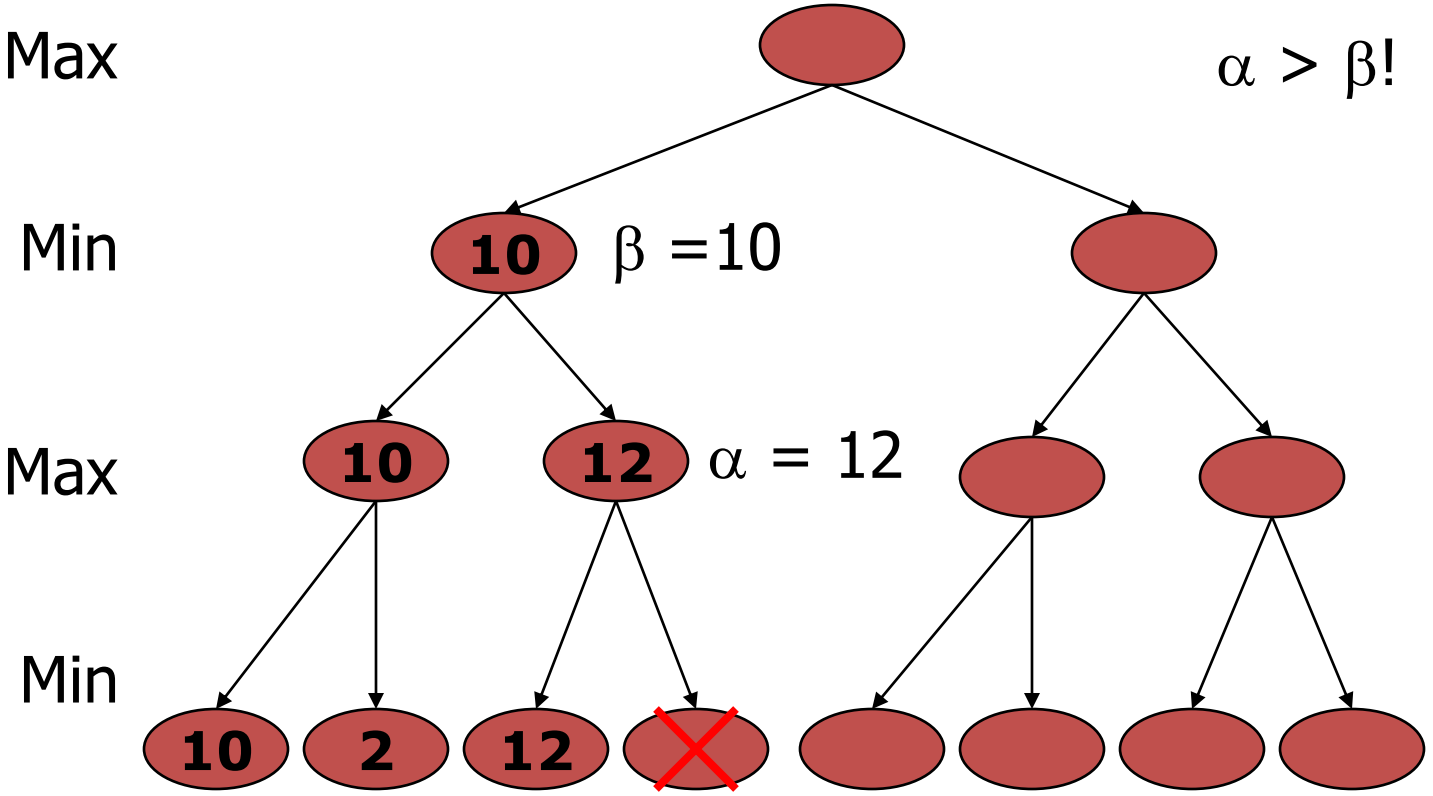
Quiz

Which is false

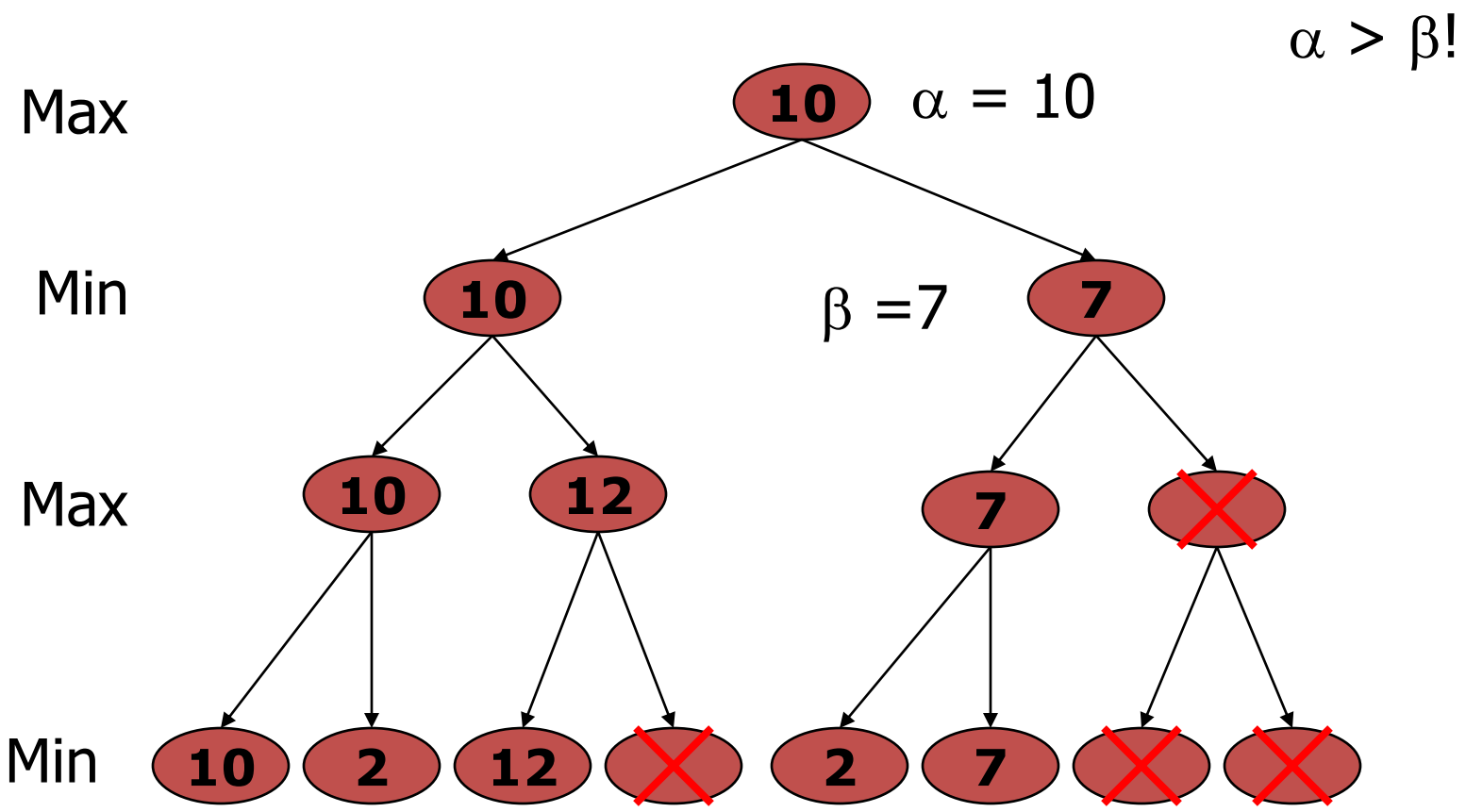
A. Production rules are used in expert system
B. Three player game uses minmax
C. Minmax explores all nodes
D. All of these

Example

Alpha Beta Example

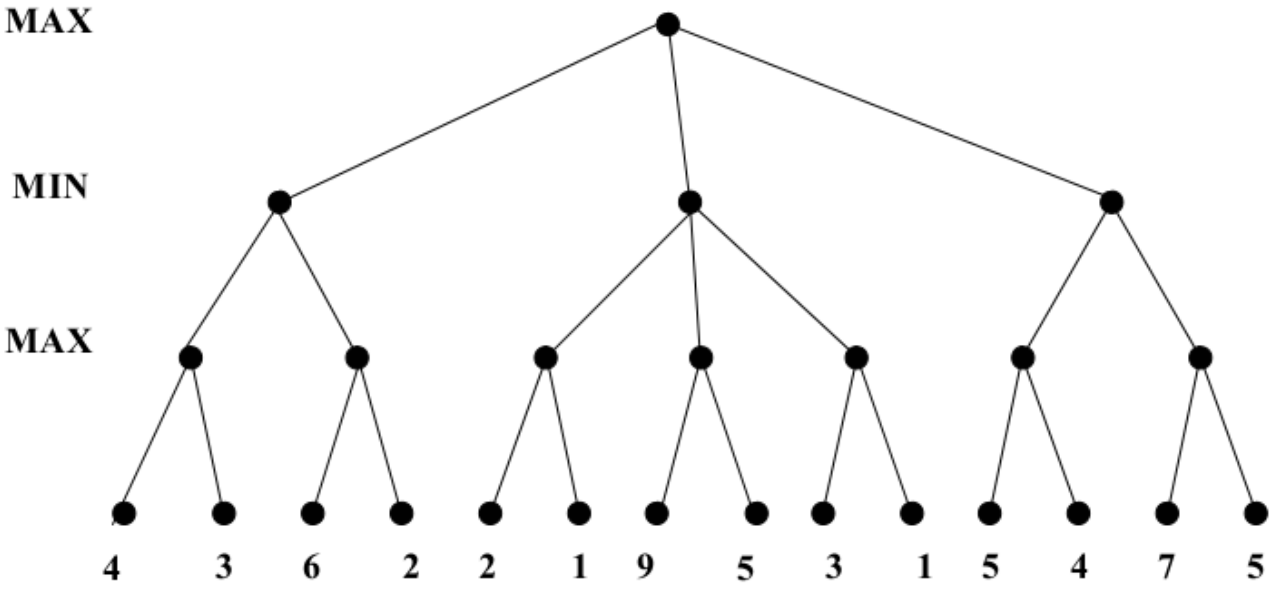


Alpha Beta Example

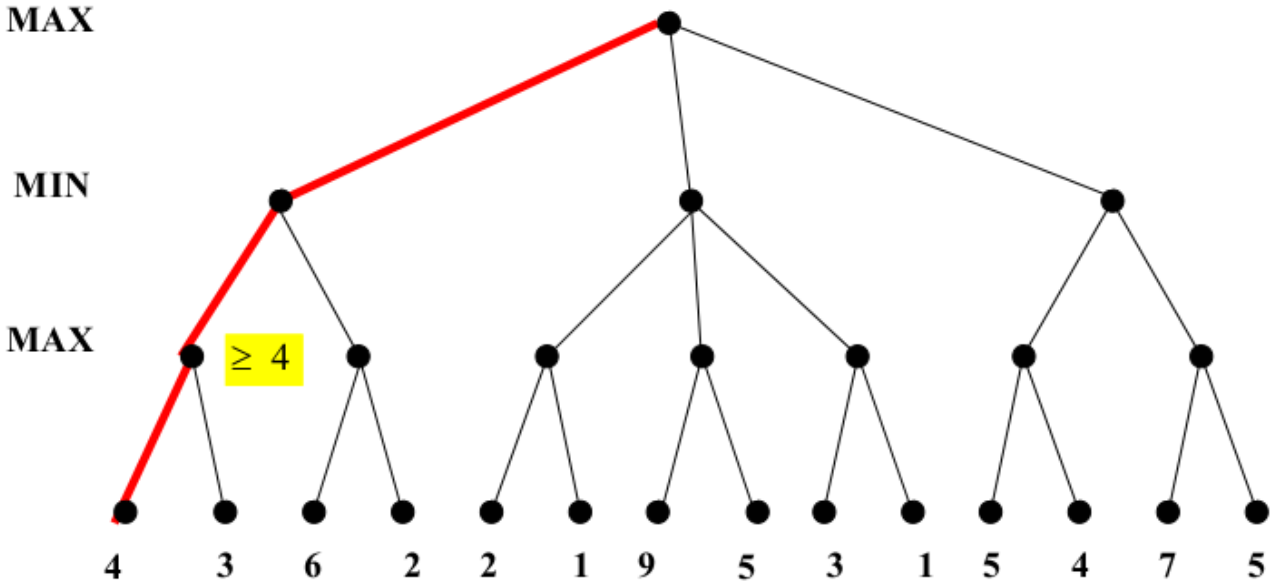


Next Example

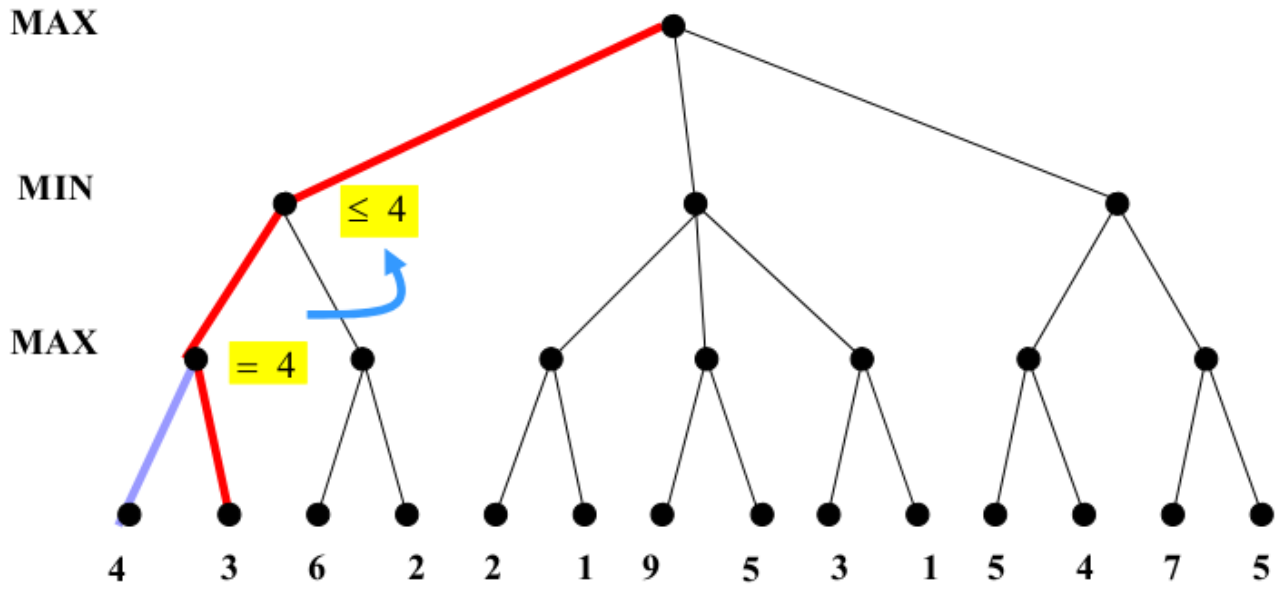
Alpha beta pruning. Example



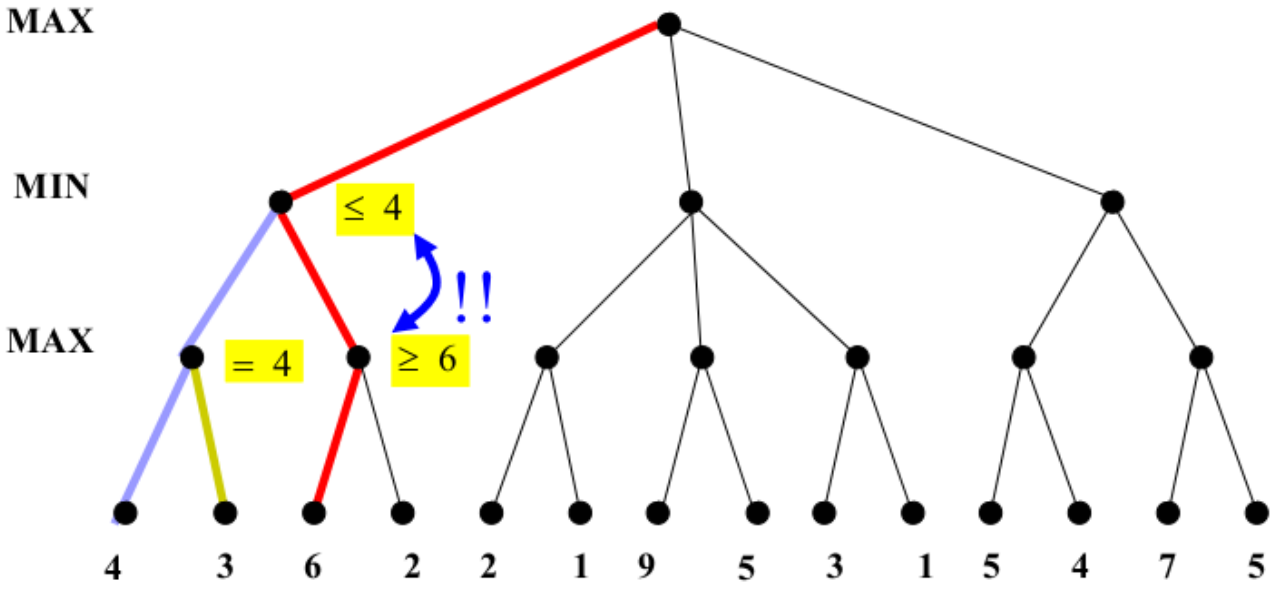
Alpha beta pruning. Example



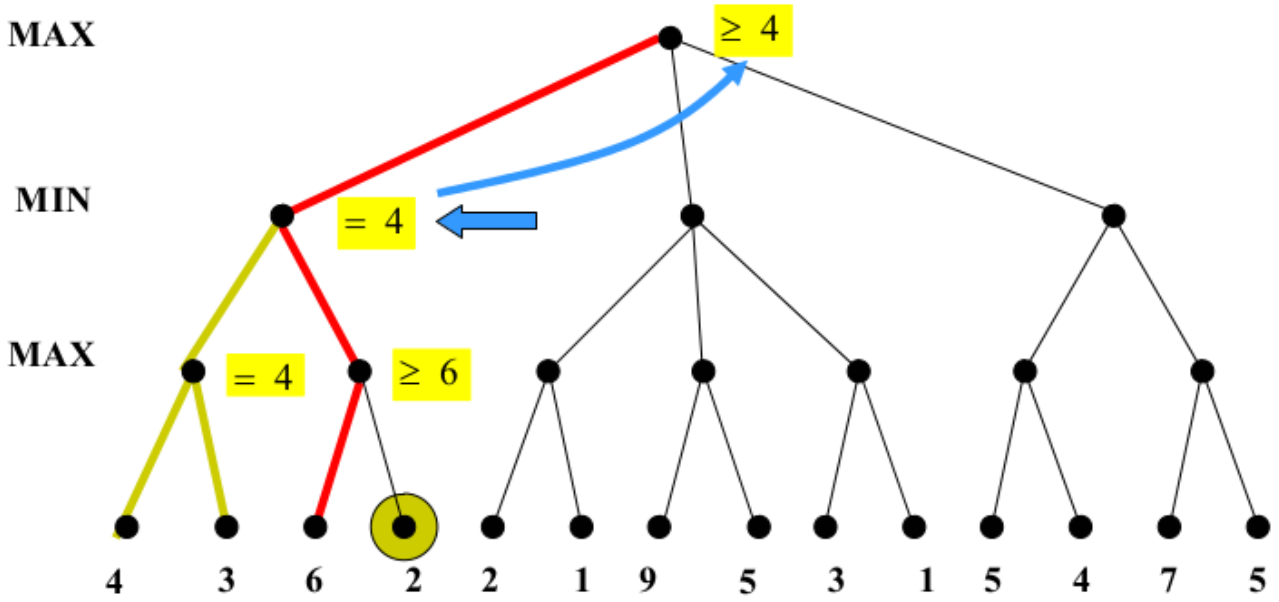
Alpha beta pruning. Example



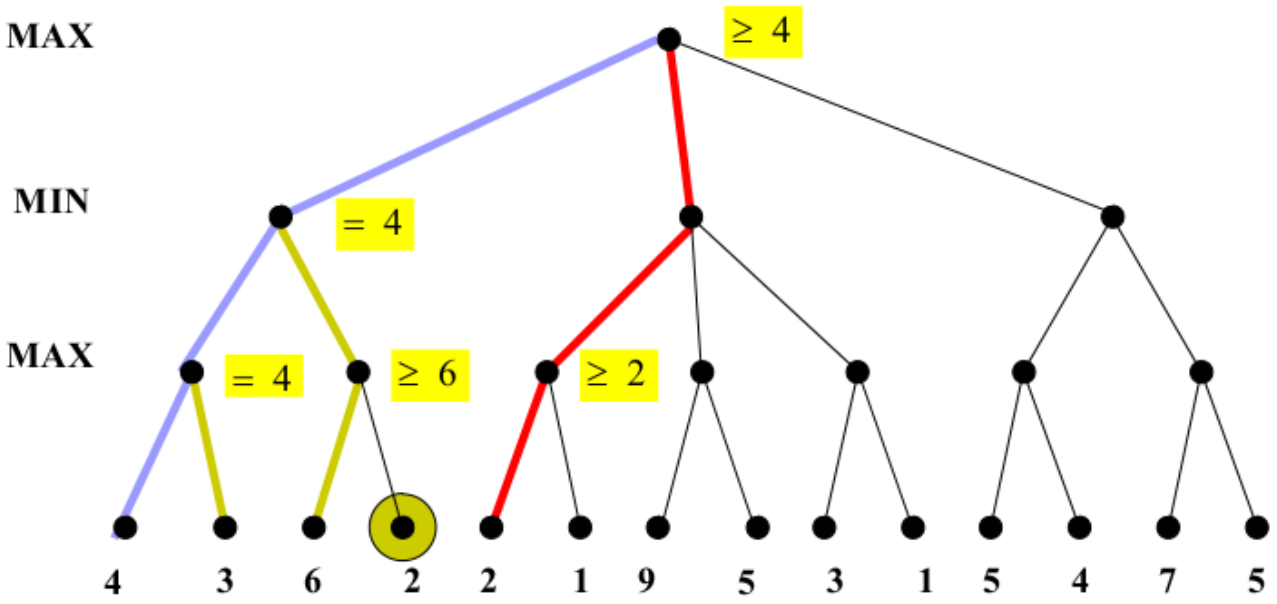
Alpha beta pruning. Example



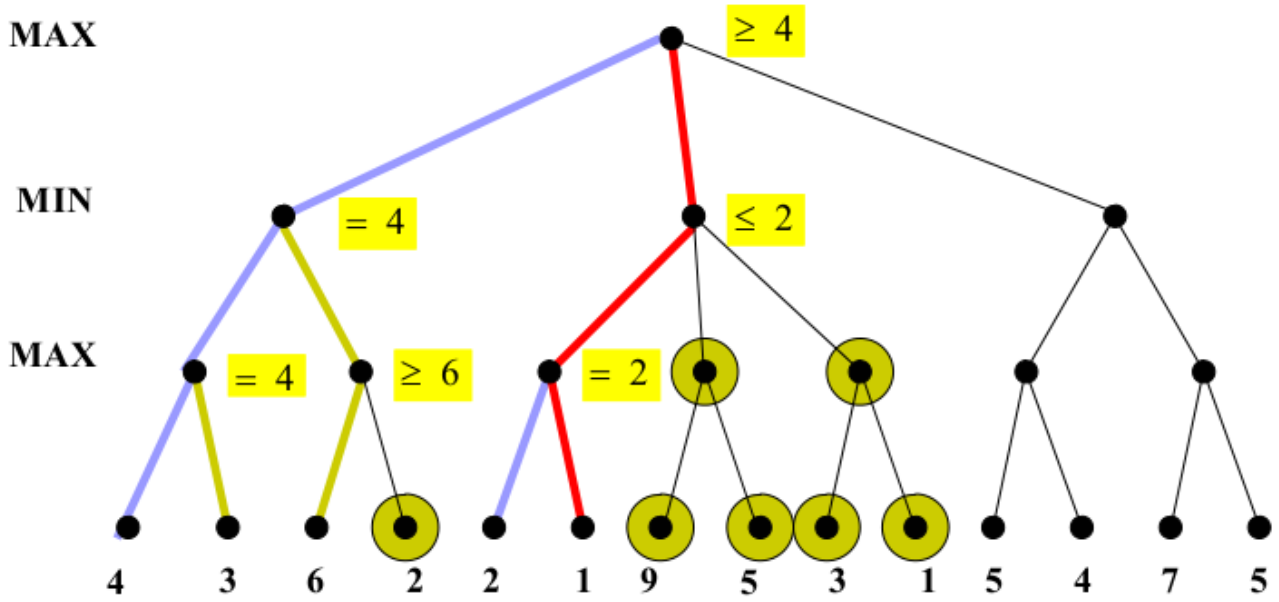
Alpha beta pruning. Example



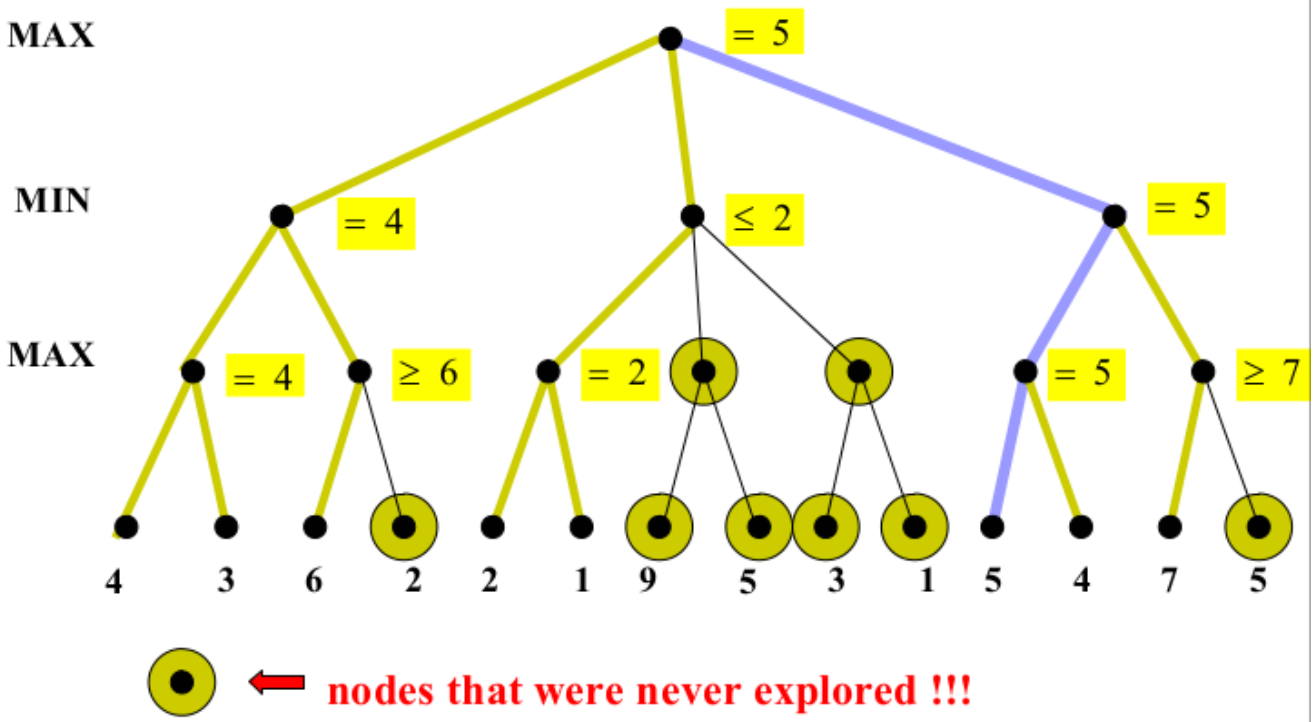
Alpha beta pruning. Example



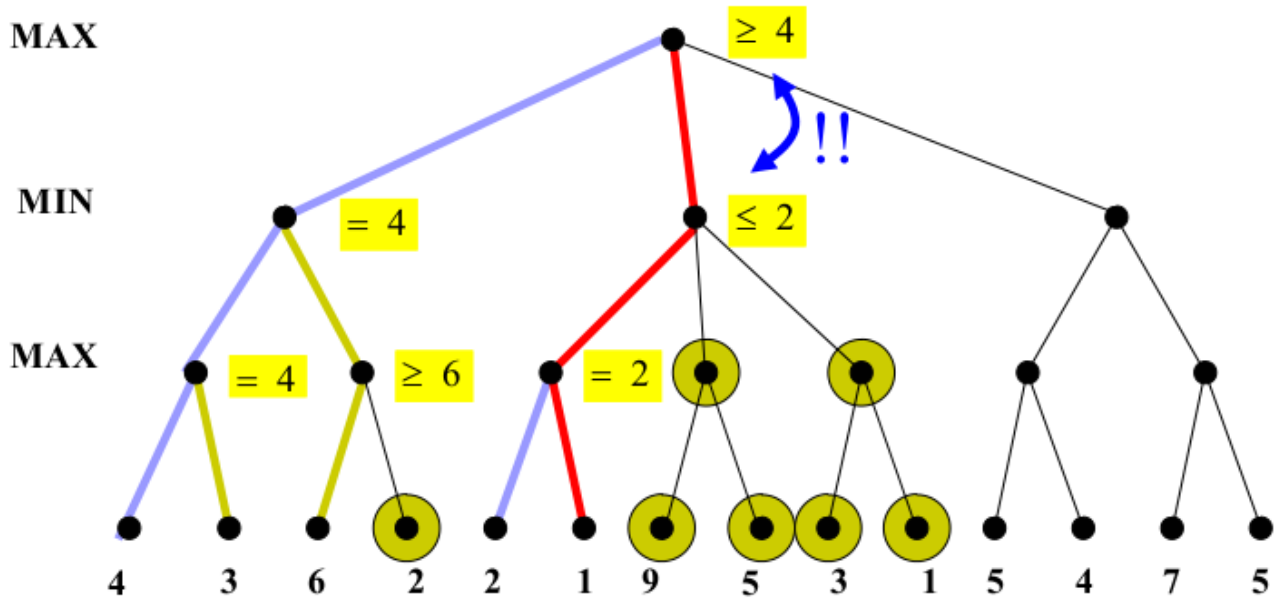
Alpha beta pruning. Example



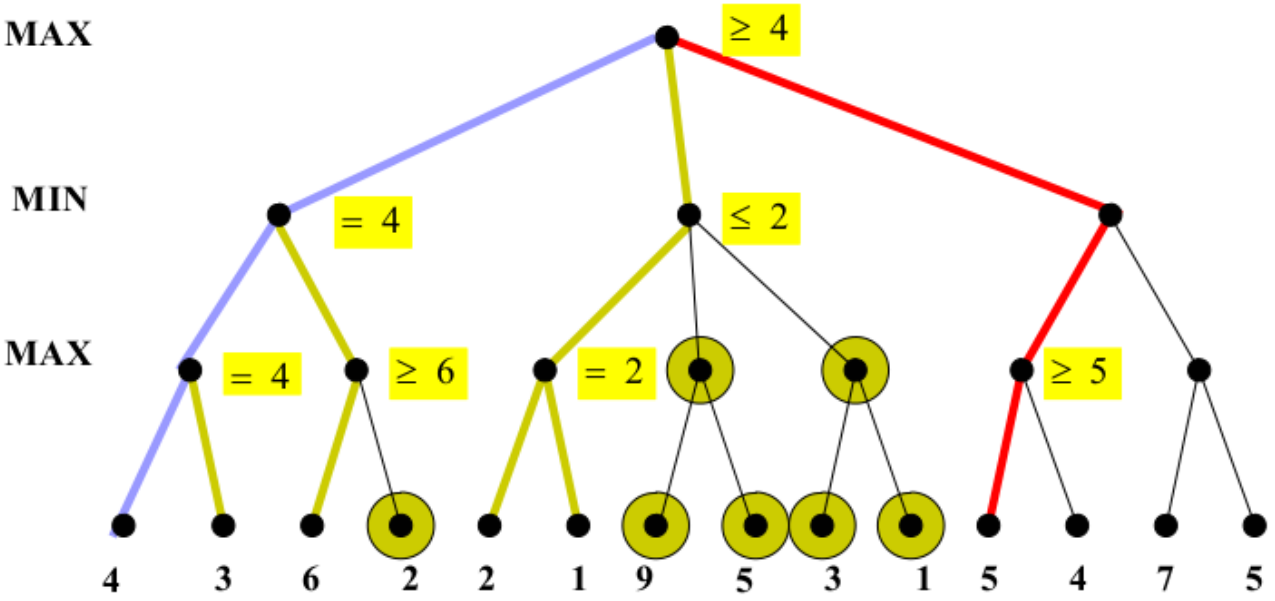
Alpha beta pruning. Example



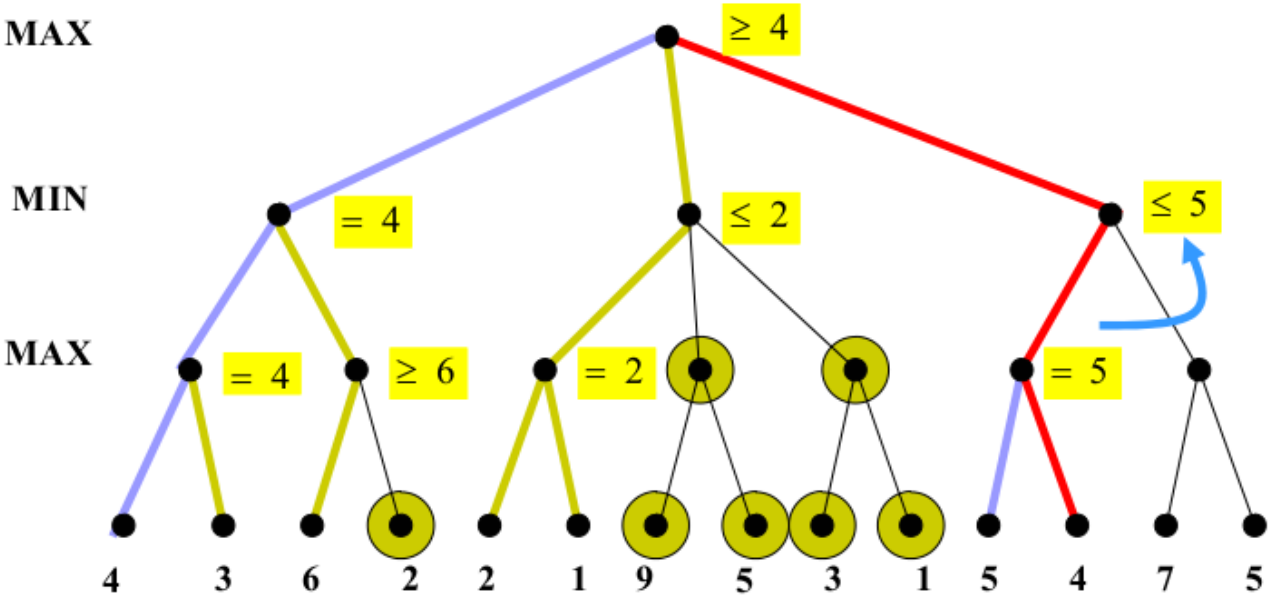
Alpha beta pruning. Example



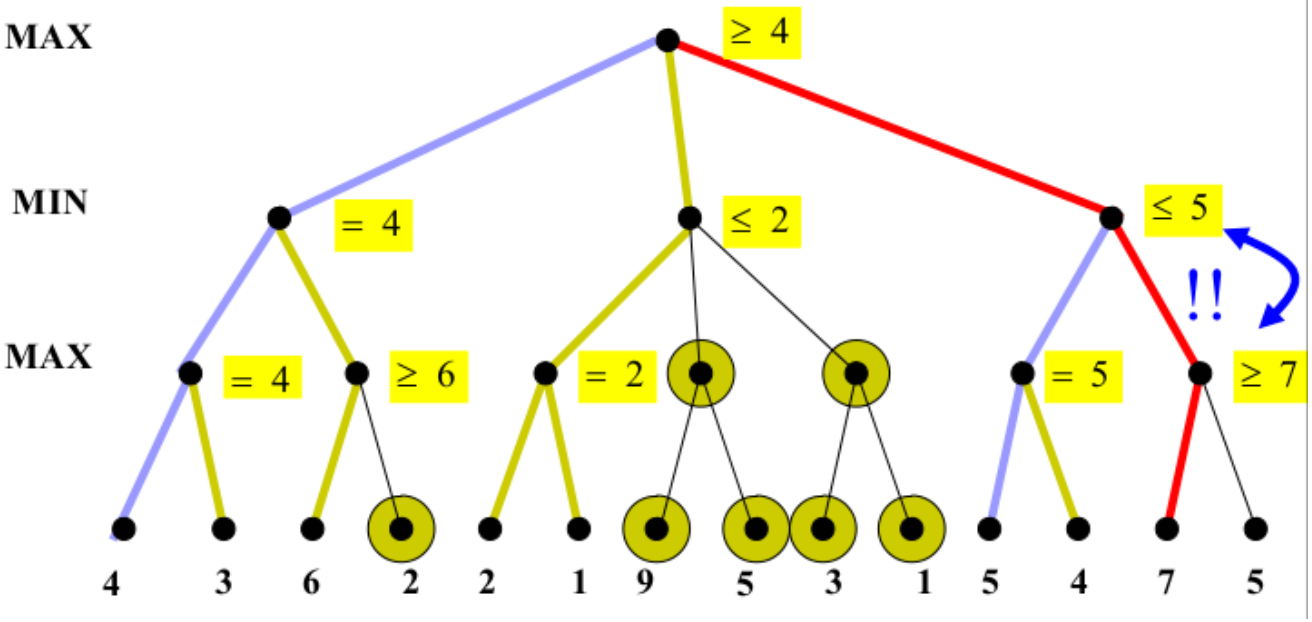
Alpha beta pruning. Example



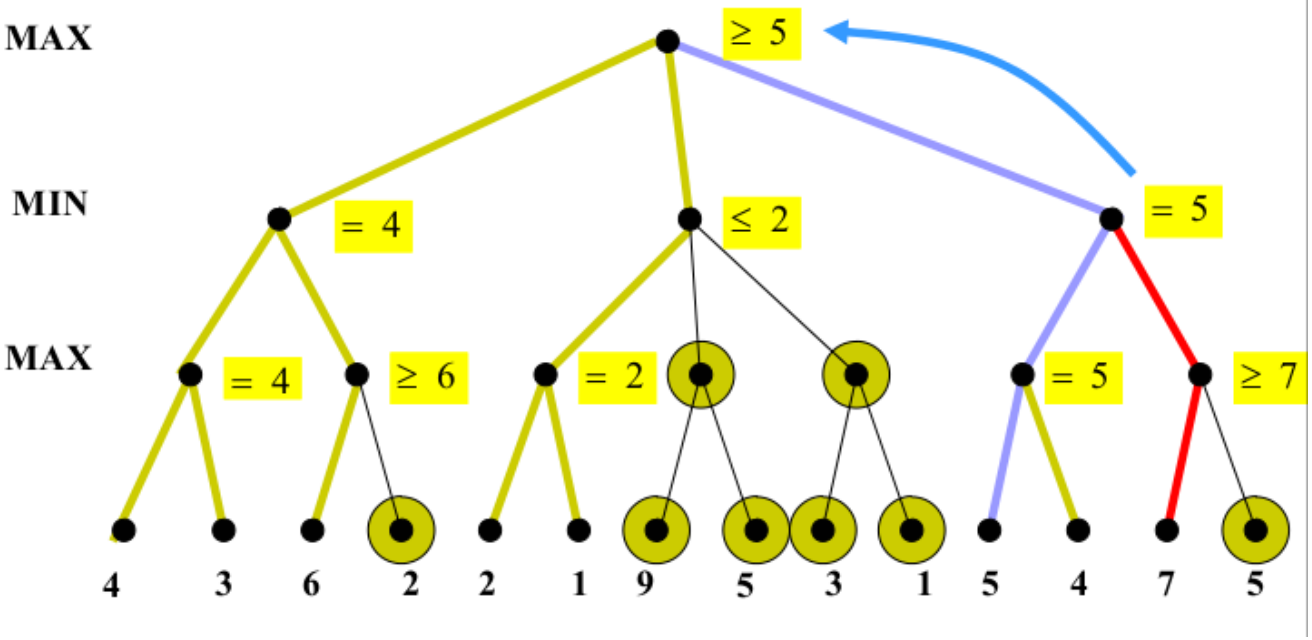
Alpha beta pruning. Example



Alpha beta pruning. Example



Alpha beta pruning. Example



NegaMax

NegaMax Algorithm

- In case of MinMax algorithm two different functions are used such as Max and Min
- Max and Min functions are doing same thing, selecting best possible values for the players.

NegaMax Algorithm

- In NegaMax only one function is used in place of two.
- In contrast to MinMax, NegaMax always selects maximum value out of possible values.
- When NegaMax selects any value than that the value is multiplied by -1.

```
// Search game tree to given depth, and return evaluation of
// root node.
int NegaMax(gamePosition, depth)
{
        if (depth=0 || game is over)
                // evaluate leaf gamePositionition from
                // current player's standpoint
                return Eval (gamePosition);
         // present return value
        score = - INFINITY;
        // generate successor moves
        moves = Generate(gamePosition);
        // look over all moves
        for i =1 to sizeof(moves) do
        {
                // execute current move
                Make(moves[i]);
                // call other player, and switch sign of
                // returned value
                cur = -NegaMax(gamePosition, depth-1);
                 // compare returned value and score
                // value, update it if necessary
                if (cur > score) score = cur;
                // retract current move
                Undo(moves[i]);
        }
        return score;
}
```

Fig. 3. NegaMax Algorithm Pseudo Code

Note:-

1. Depth variable stores the levels that one want to generate

Quiz

Twitter data is hard to process due to

A. Annotation is required
B. Twitter does not give data
C. Due to network issue
D. All of these

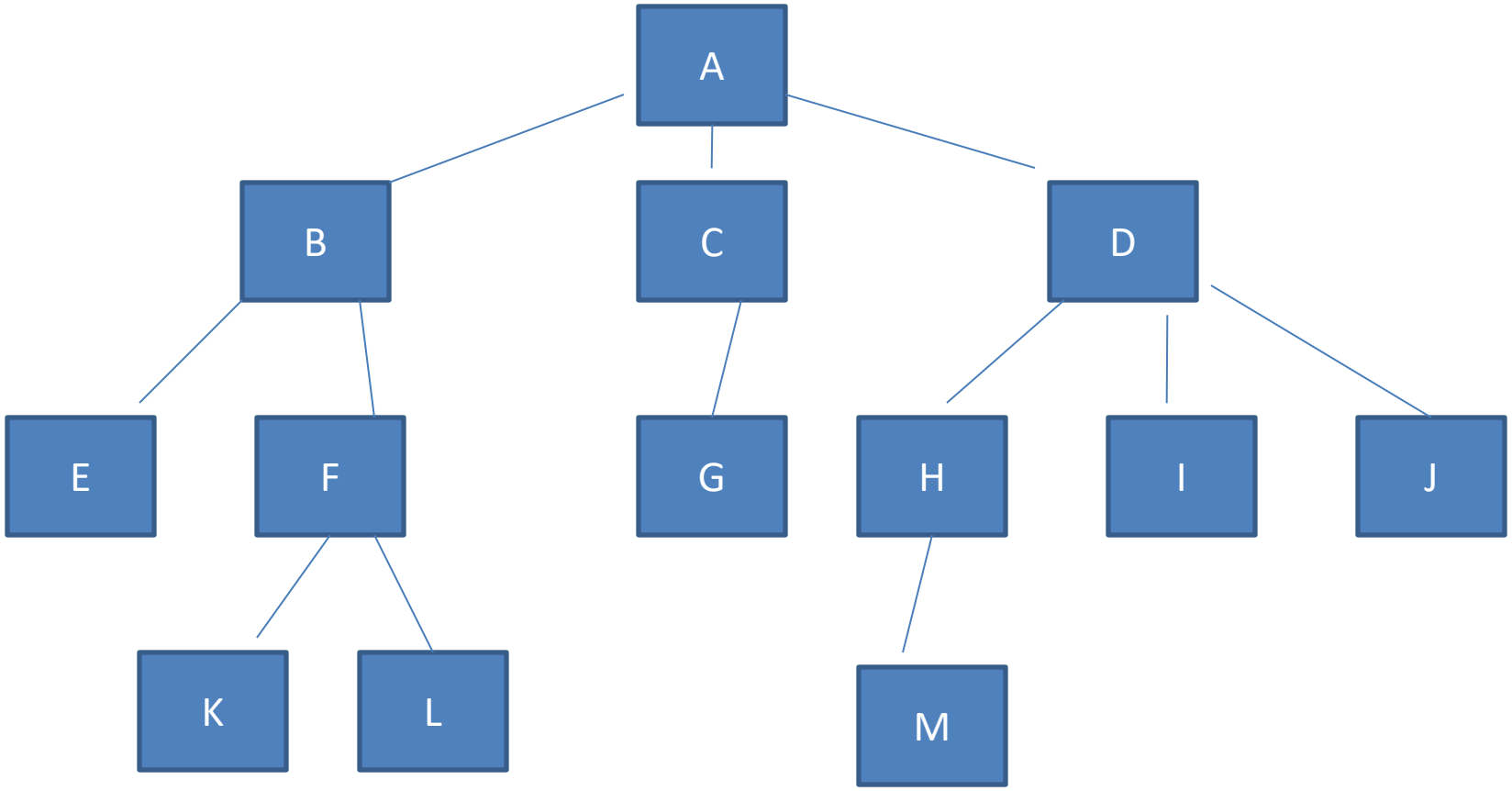
Quiz

Twitter data is hard to process due to

A. Annotation is required
B. Twitter does not give data
C. Due to network issue
D. All of these

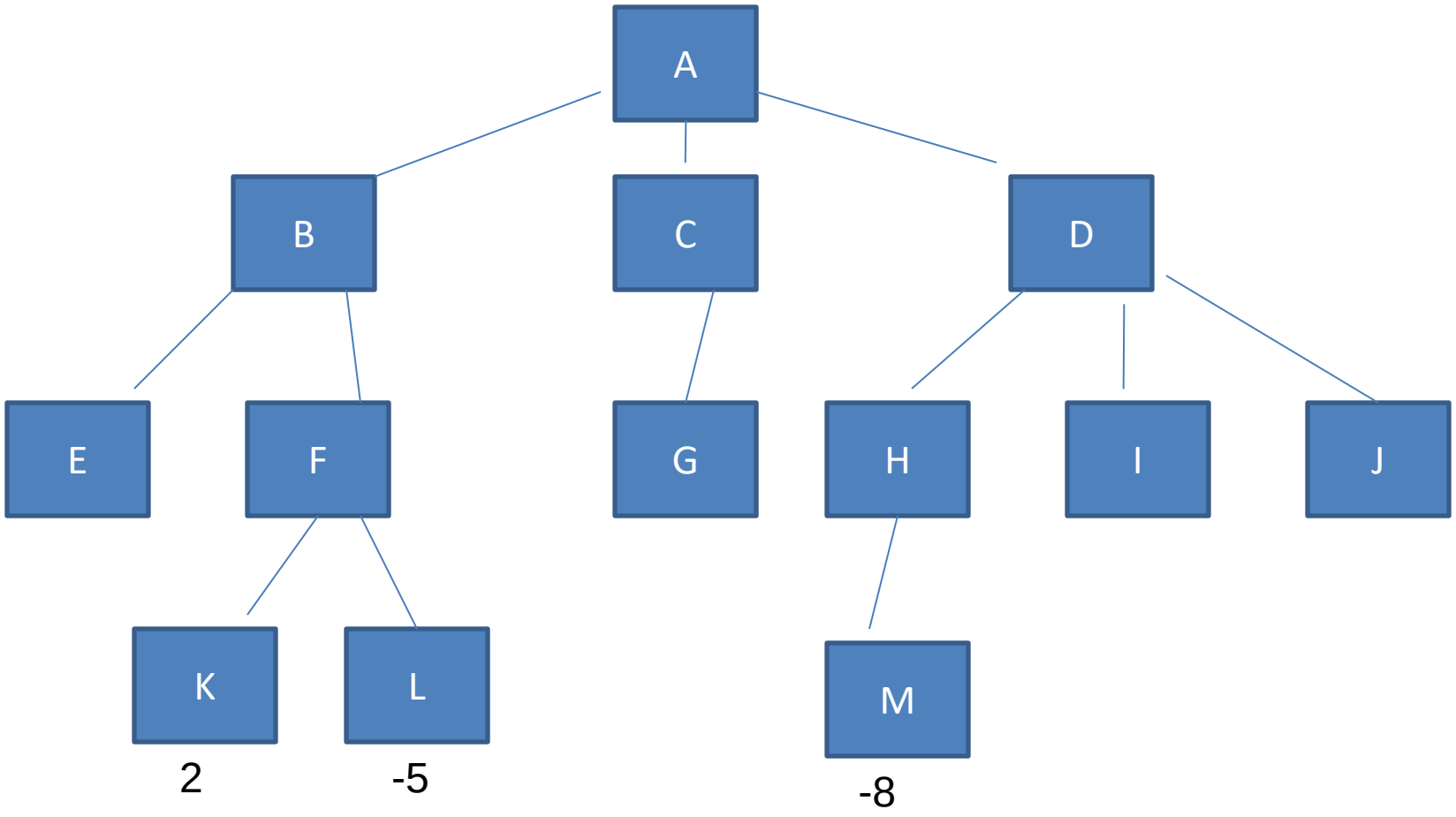
NegaMax Algorithm

Let us consider this example



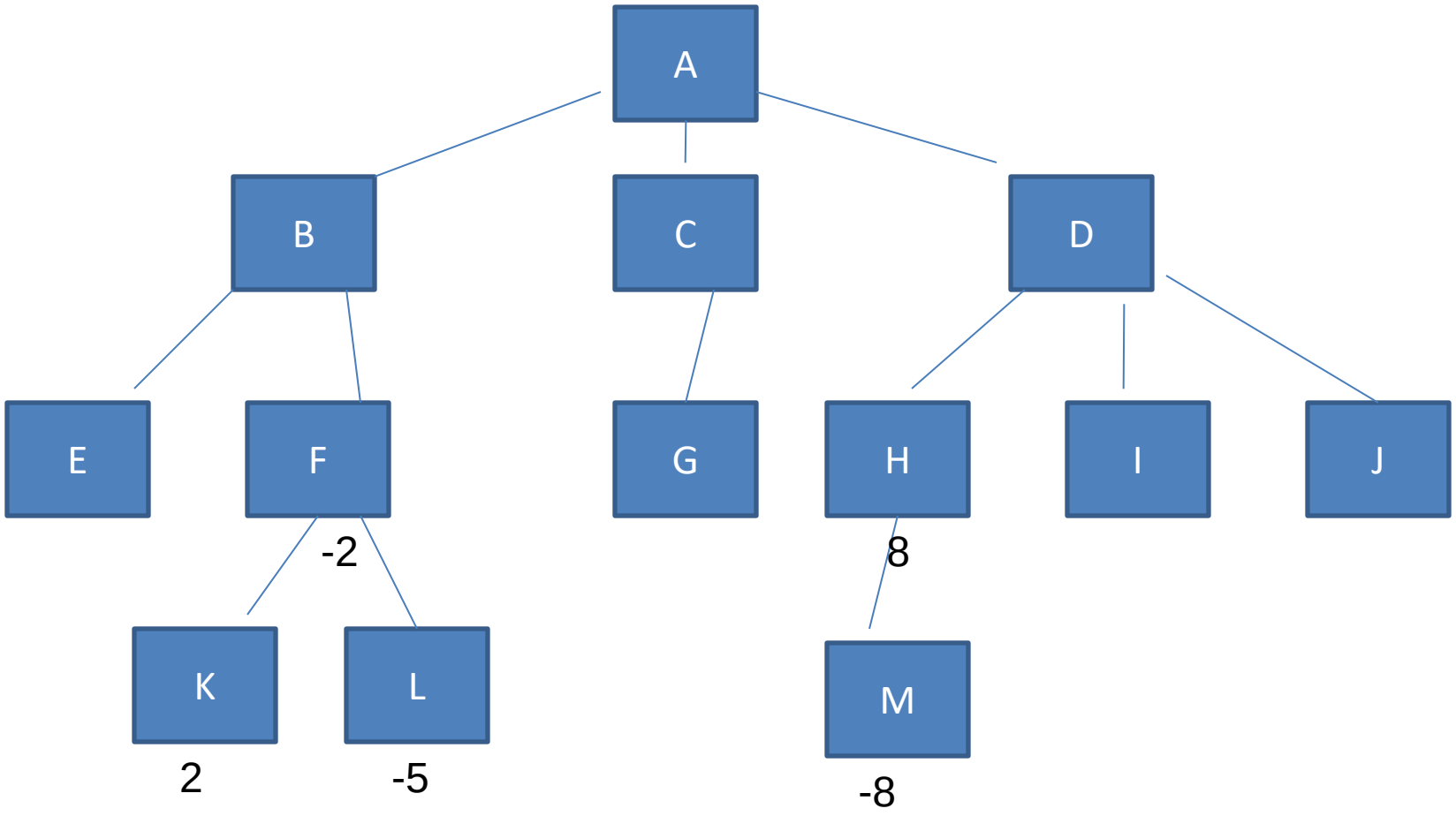
NegaMax Algorithm

Let us consider this example



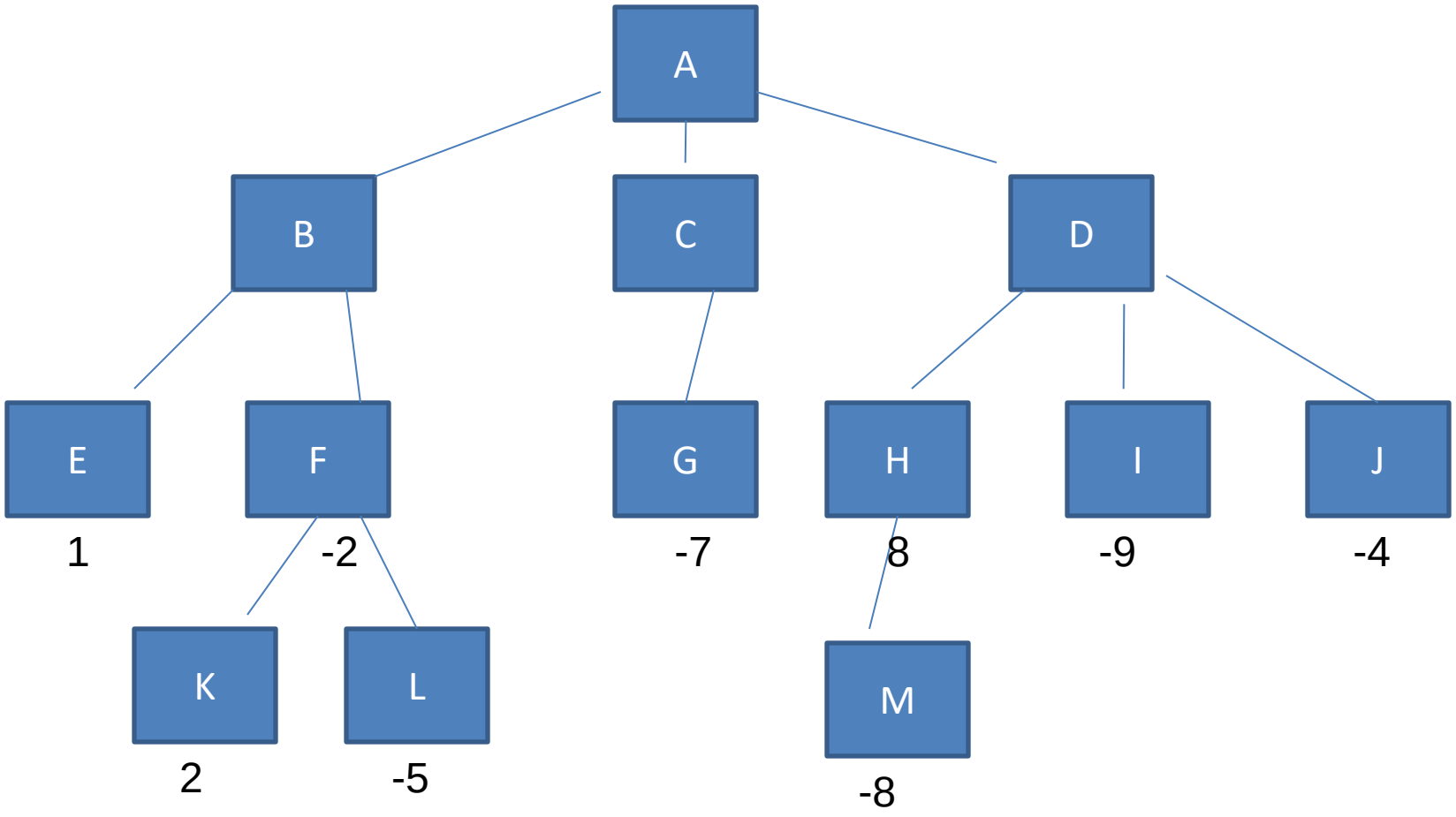
NegaMax Algorithm

Let us consider this example



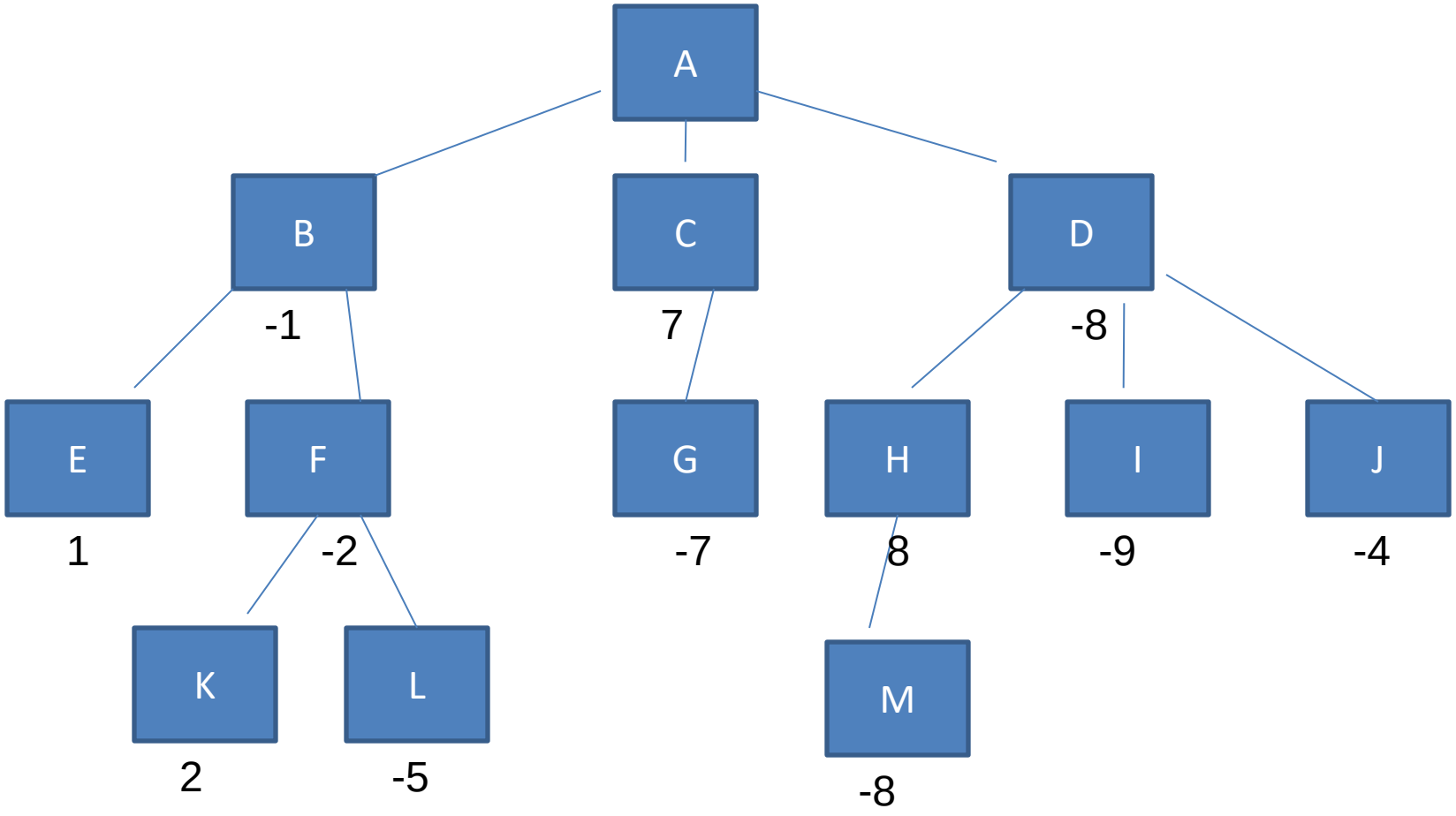
NegaMax Algorithm

Let us consider this example



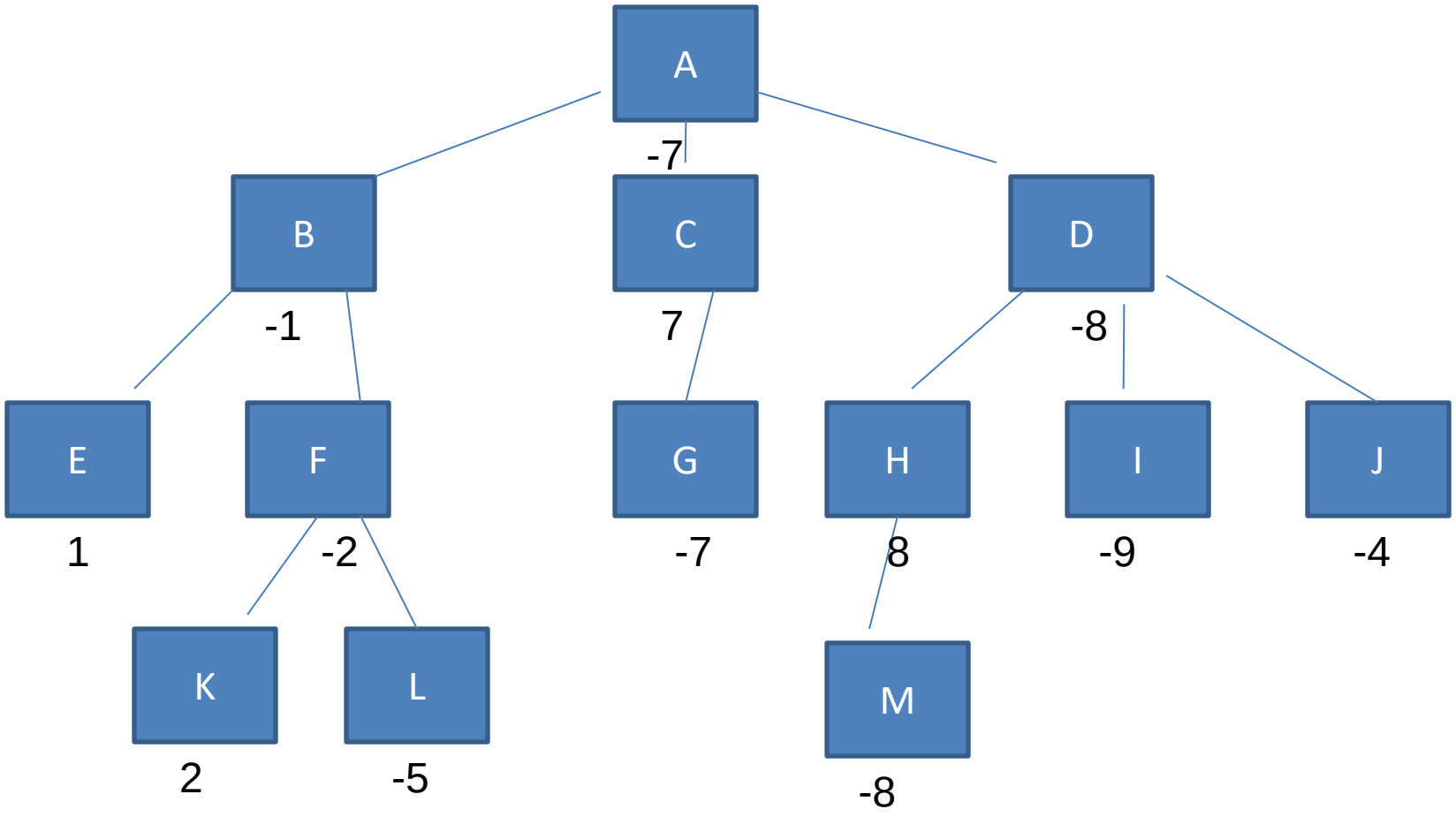
NegaMax Algorithm

Let us consider this example



NegaMax Algorithm

Let us consider this example



Quiz

Utility function can be applied

A. Last level of tree
B. Root node
C. Intermediate levels
D. None of these

Quiz

Utility function can be applied

A. Last level of tree
B. Root node
C. Intermediate levels
D. None of these

HEXAPAWN GAME

Hexapawn Rules

Hexapawn is a two-player game proposed by Martin Gardner in the 1960's. The game is played on a 3×3 board in which three white pawns and three black pawns are placed on opposite ends, as shown below:

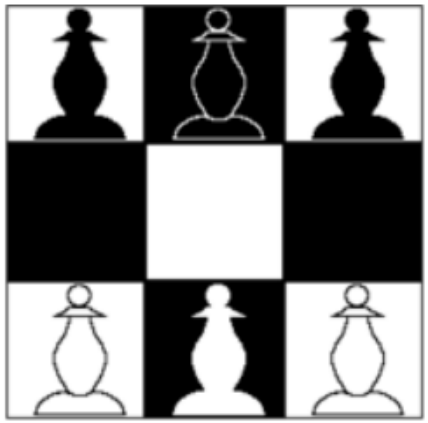


Figure 1: A starting 3×3 Hexapawn board.

Players take turns moving one of their pawns, which can be moved just like in Chess: a pawn can either be moved forward a single square (assuming the square ahead is unoccupied), or can capture an opposing pawn by moving a single square diagonally. The end-game conditions are slightly different from Chess – the game ends if any of the following events occur:

- A pawn reaches the opposite end of the board.
- A player loses all of their pawns.
- A player cannot make any further moves.

In all cases, the final player to move is the winner. Finally, note that while a standard Hexapawn board is 3×3, you can also consider a Hexapawn game on a different board size, such as 4×3 (opposing pawns are further apart) or 3×4 (four pawns on each side instead of three).

Quiz

Minmax algorithm can be applied

A. Single player game
B. Two players game
C. Three players game
D. Multiplayer's game

Quiz

Minmax algorithm can be applied

A. Single player game
B. Two players game
C. Three players game
D. Multiplayer's game