COMP338: **ARTIFICIAL INTELLIGENCE**

AI & Games

Dr. Radi Jarrar
Department of Computer Science

AI & Games

- Competitive environment in a multi-agent environment
- There is an **opponent,** we can’t control, planning again us!
  - Each agent needs to consider actions of other agents and they might affect its own welfare
- The unpredictability of the other agents might introduce some contingencies into the agent's problem-solving process
- In competitive environments, the goals are in conflict

AI & Games

- Game vs. search: optimal solution is not a sequence of actions but a **strategy** (policy)
  - If opponent does *a*, agent does *b*, else if opponent does *c*, agent does *d*, etc.
- Tedious and fragile if hard-coded (i.e., implemented with rules)
- Good news: games are modeled as **search problems** and use **heuristic evaluation** functions.
- Heuristic evaluation functions allow us to approximate the true utility of a state without doing a complete search.

Games

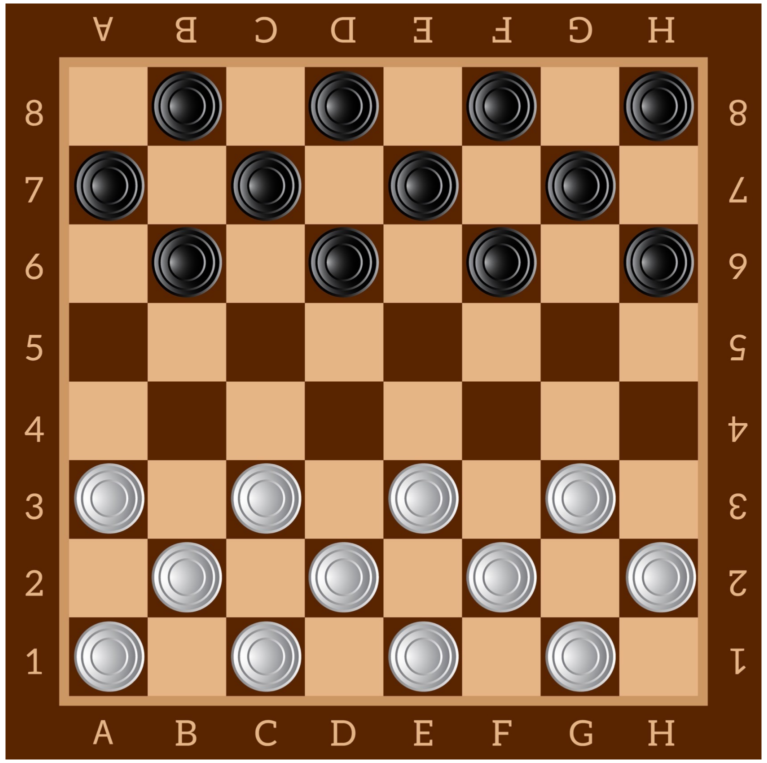
- Games are a big deal in AI
- Games are interesting to AI because they are too hard to solve
- We need to make some decision even when the optimal decision is infeasible

Games

- Adversarial search (which is known as Games)
- Mathematical **game theory**, a branch of economics, views any multiagent environment as a game, provided that the impact of each agent on the others is “significant,” regardless of whether the agents are cooperative or competitive

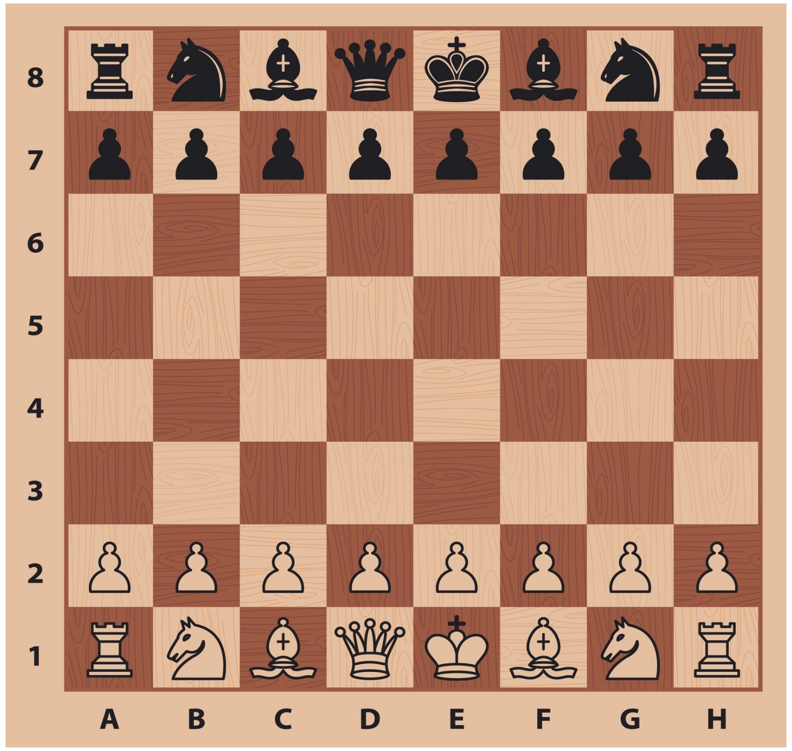
Games

- Chinook (a computer program that plays checkers) ended 40-year-reign of human world champion Marion Tinsley in 1994.
- Used an endgame database defining perfect play for all positions involving 8 or fewer pieces on the board, a total of 443,748,401,247 positions.



Games

- In 1949, Claude E. Shannon in his paper "Programming a Computer for Playing Chess", suggested *Chess* as an AI problem for the community
  - Deep Blue defeated human world champion Gary Kasparov in a six-game match in 1997
  - In 2006, Vladmir Kramnik, the undisputed world champion, was defeated 4-2 by Deep Fritz



Games

- Google Deep mind through its Project *AlphaGo* could beat both Fan Hui, the European Go champion and Lee Sedol the worlds best player in the game Go in the year 2016

- Othello: human champions refuse to compete against computers. Computers are too good.

Types of Games

	deterministic	chance
perfect information	chess, checkers, go, othello	backgammon monopoly
imperfect information	battleships, blind tictactoe	bridge, poker, scrabble nuclear war

- We are mostly interested in deterministic games, fully observable environments, zero-sum, where two agents act alternately.

Games

- For example, if one player wins a game of chess, the other player necessarily loses. It is this opposition between the agents' utility functions that makes the situation adversarial
- A zero-sum game is one in which the gain of one player is balanced exactly by the loss of the other player

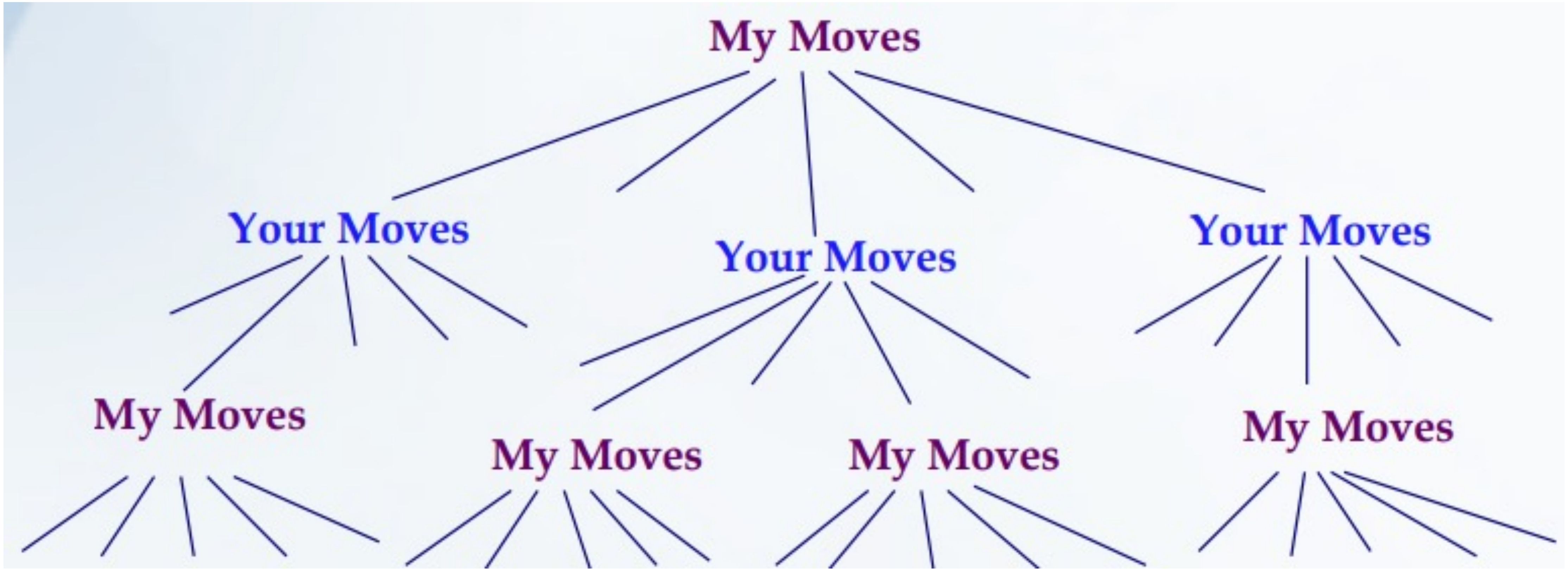
Games

- Games are interesting because they are hard to solve (too hard, for that matter!)
  - E.g., chess has a branching factor of average of 35
  - A game often go to 50 moves by each player
  - The search space has 35^{100} nodes
- Games, like the real world, therefore require the ability to make *some* decision even when calculating the *optimal* decision is infeasible
- Games also penalize inefficiency

Games

- How would a computer play a game?
  - Consider all legal moves you can make
  - Compute new positions that would result from each new moves
  - Evaluate each new position and determine the best
  - Move!
  - Wait for the opponent’s move, and repeat
- Issues to be considered
  - Representing the game (i.e., board)
  - Generating all next legal moves and evaluating the moves

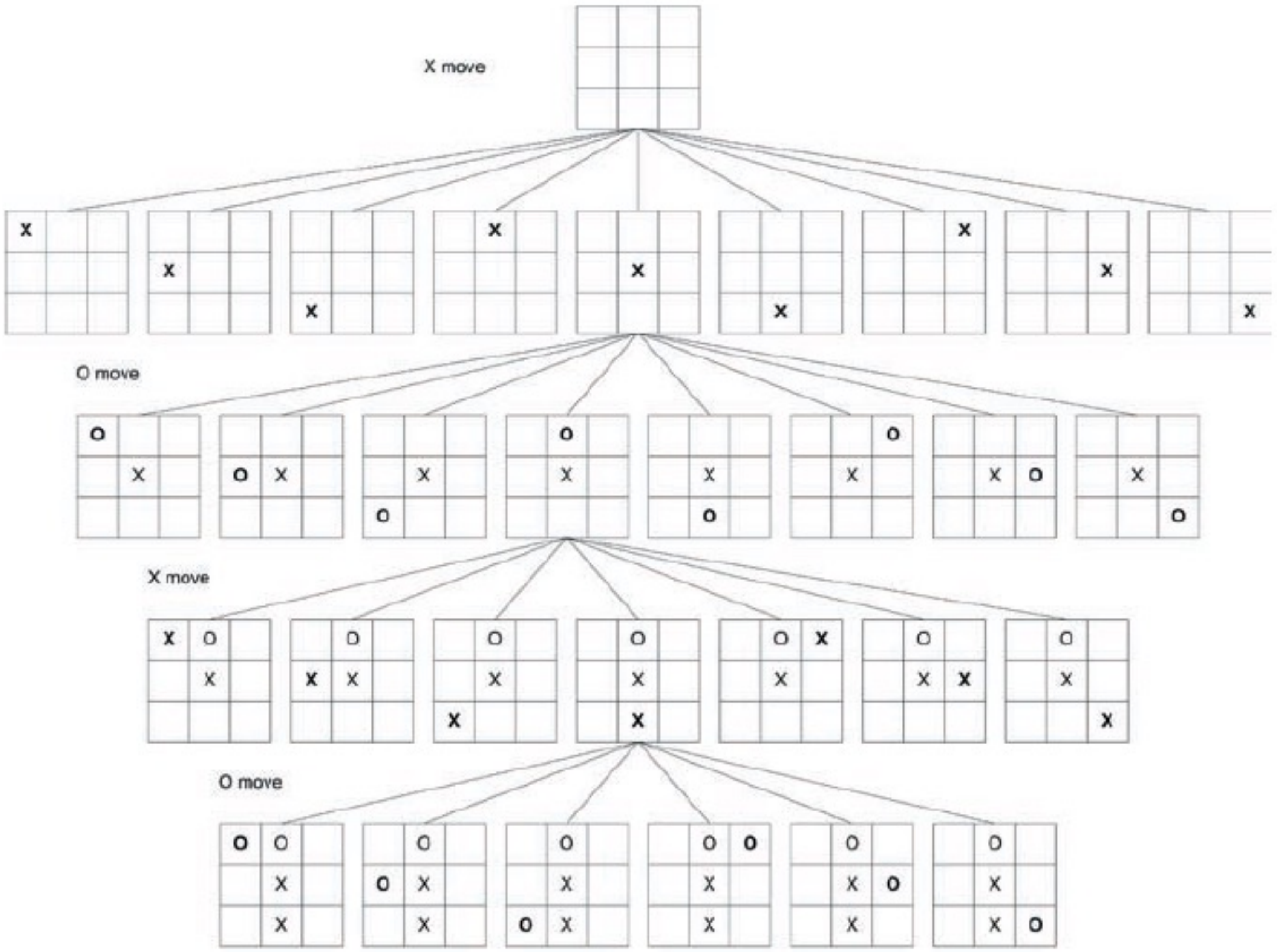
Games


My Moves
Your Moves
Your Moves
Your Moves
My Moves
My Moves
My Moves
My Moves

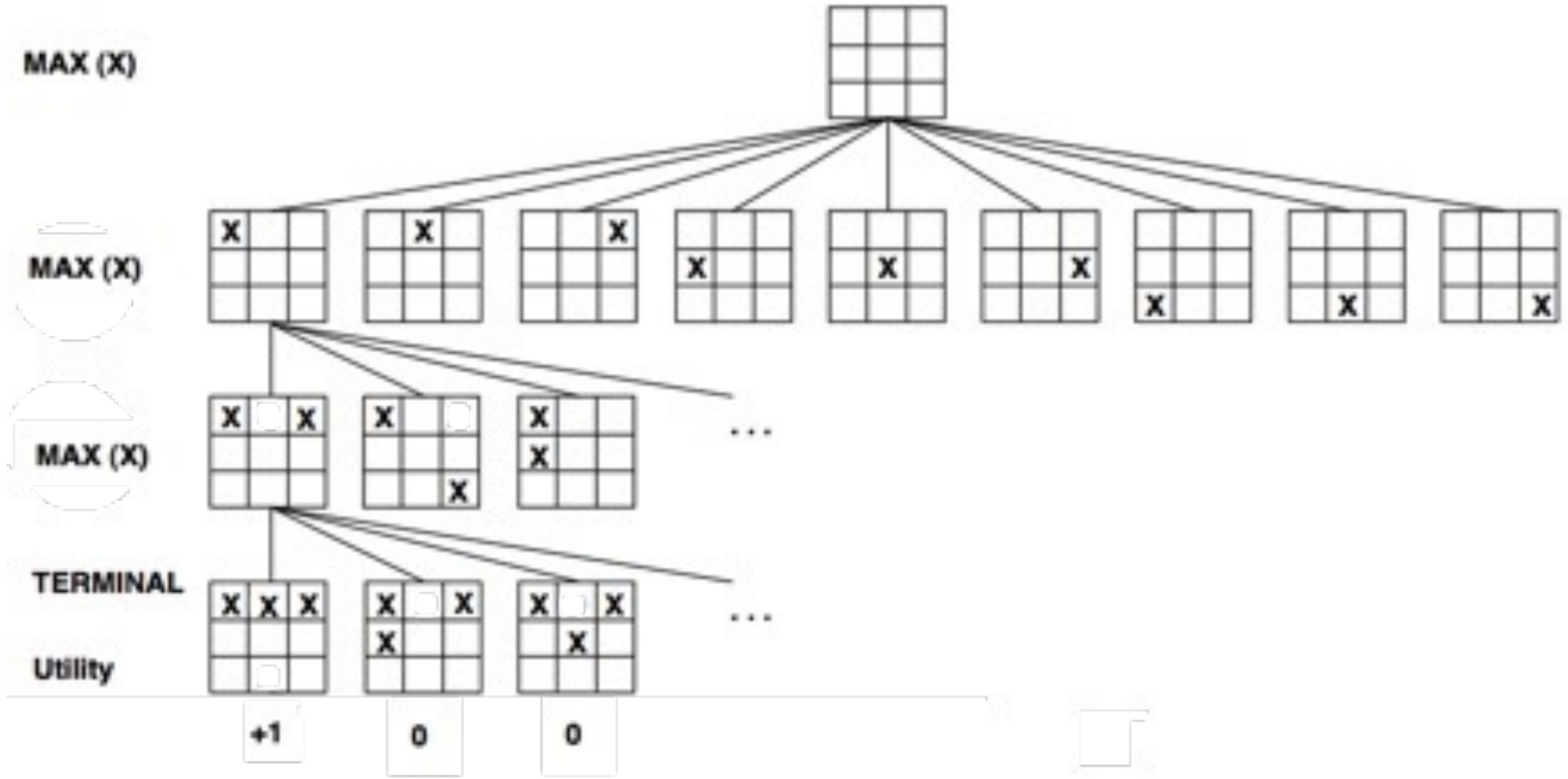
Games

- Consider the two-player game Tic-Tac-Toe
- Players alternate moves, and as each move is made, the possible moves are constrained

Games



Games



- The game is seen as a tree
- The values at the bottom are calculated by a utility function, based on the game.

Games

- A move is selected such that it leads to a win by traversing all moves that are constrained by this move
- Also, by traversing the tree for a given move, we can choose the move that leads to the win in the shallowest depth (minimal number of moves)
- In Tic-Tac-Toe, the maximum number of moves is small in comparison to other more complex games (such as checkers or chess)

Games

- So, games can be formulated as search problems
- The zero-sum utility function leads to an adversarial situation: for one agent to win, the other necessarily has to lose
- Factors that make the search complicated:
  - Potentially huge search space
  - Time limits
  - Multi-player games (teams)
  - Elements of chance

Games

- In single-person games, a sequence of moves that lead to a win is identified
- Easier than two player games
- Traditional search methods only consider how close the agent to the goal state (i.e., BFS, DFS, …)
- In two player games, decisions of both agents should be taken into account as the decision of one agent will affect the resulting search space that the other agent would further explore

Minimax

- The search space of a two player game can be formalised through an agent that is called MAX and the opponent is called MIN
- Players alternate turns
- Max moves first
- Max maximizes results
- Min minimizes the result
- Compute each node's minimax value's the best achievable utility against an optimal adversary
- Minimax value = best achievable payoff against best play

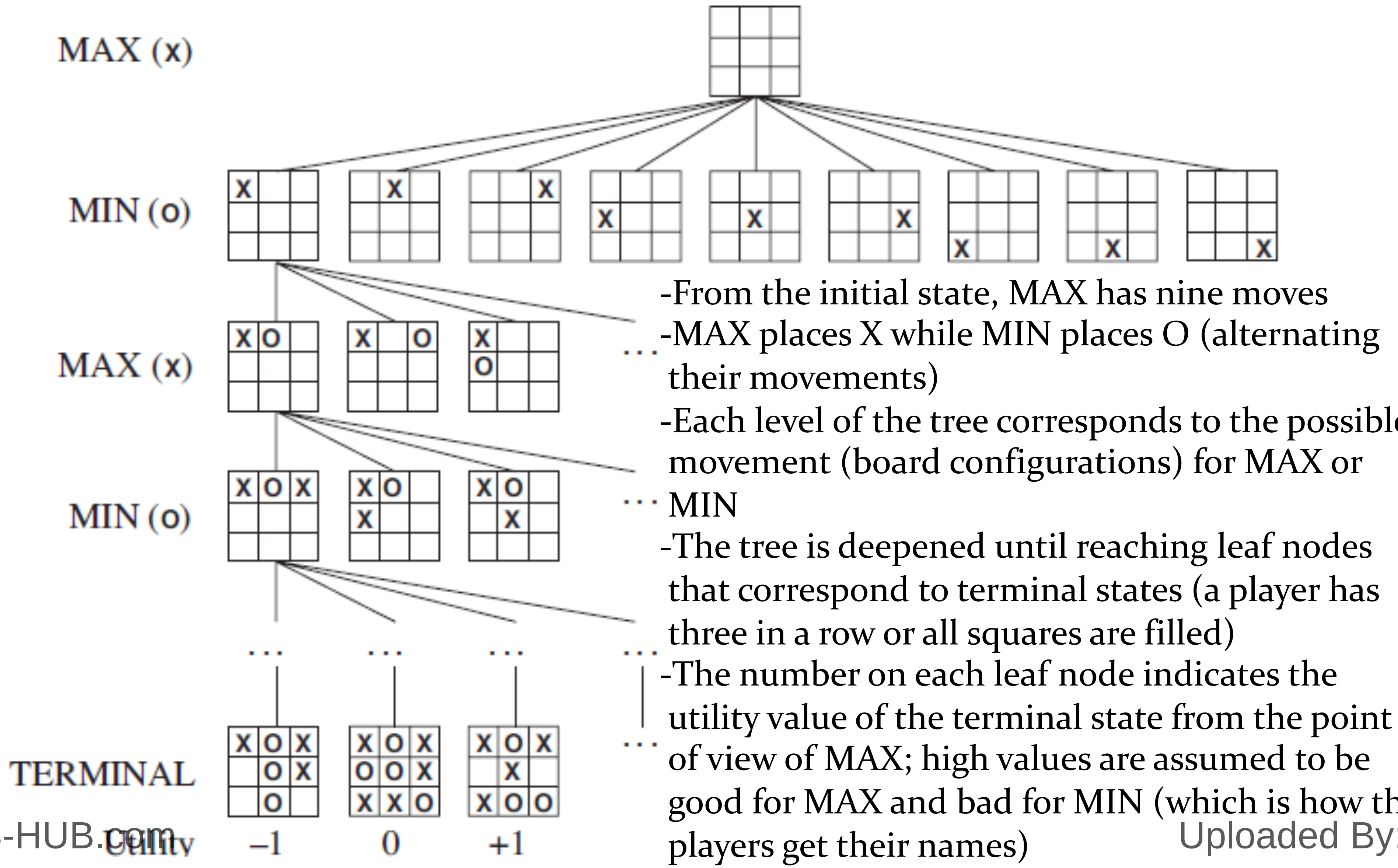
Minimax – Problem Formulation

- Initial state: board configurations and the player to move.
- Successor function: list of pairs (move, e) specifying legal moves and their resulting states. (moves + initial state = game tree)
- A terminal test: true when the game is over and false otherwise.
- A utility function: a utility function (an objective function) that defines the final numeric value for a game that ends in terminal state . For example (Chess) the outcome = win/loss/draw, with values +1, -1, 0.

Minimax – Problem Formulation

- The initial state, Actions function, and Result function define the **game tree**
- In the game tree, the nodes are states and the edges are moves

Minimax



-From the initial state, MAX has nine moves
-MAX places X while MIN places O (alternating their movements)
-Each level of the tree corresponds to the possible movement (board configurations) for MAX or MIN
-The tree is deepened until reaching leaf nodes that correspond to terminal states (a player has three in a row or all 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 assumed to be good for MAX and bad for MIN (which is how the players get their names)

MiniMax Algorithm

- Create a start node and as the MAX node with the initial configurations
- Expand nodes down to some depth in the game
- Apply the evaluation function to each leaf node
- Propagate the values of non-leaf nodes until a value is computed for the root node
  - At MAX nodes, the propagated value is the maximum of the values associated with its children
  - At MIN nodes, the propagated value is the minimum of the values associated with its children
- Select the operator that associated with the child node whose propagated value determined the value at the root

MiniMax Algorithm

- Find the optimal strategy for Max:
  - Depth-first search of the game tree
  - An optimal leaf node could appear at any depth of the tree
  - Minimax principle: compute the utility of being in a state assuming both players play optimally from there until the end of the game
  - Propagate minimax values up the tree once terminal nodes are discovered
    - If state is terminal node: *Value is utility(state)*
    - If state is MAX node: *Value is highest value of all successor node values (children)*
    - If state is MIN node: *Value is lowest value of all successor node values (children)*

Algorithm of Minimax game tree search

```
/* Find the child state with the lowest utility value */

function MINIMIZE(state)
    returns TUPLE of ⟨STATE, UTILITY⟩ :

    if TERMINAL-TEST(state):
        return ⟨NULL, EVAL(state)⟩

    ⟨minChild, minUtility⟩ = ⟨NULL, $\infty$⟩

    for child in state.children():
        ⟨ _, utility⟩ = MAXIMIZE(child)

        if utility $<$ minUtility:
            ⟨minChild, minUtility⟩ = ⟨child, utility⟩

    return ⟨minChild, minUtility⟩
```

```
/* Find the child state with the highest utility value */

function MAXIMIZE(state)
    returns TUPLE of ⟨STATE, UTILITY⟩ :

    if TERMINAL-TEST(state):
        return ⟨NULL, EVAL(state)⟩

    ⟨maxChild, maxUtility⟩ = ⟨NULL, $-\infty$⟩

    for child in state.children():
        ⟨ _, utility⟩ = MINIMIZE(child)

        if utility $>$ maxUtility:
            ⟨maxChild, maxUtility⟩ = ⟨child, utility⟩

    return ⟨maxChild, maxUtility⟩
```

```
/* Find the child state with the highest utility value */

function DECISION(state)
    returns STATE :

    ⟨child, _⟩ = MAXIMIZE(state)

    return child
```

MiniMax Algorithm

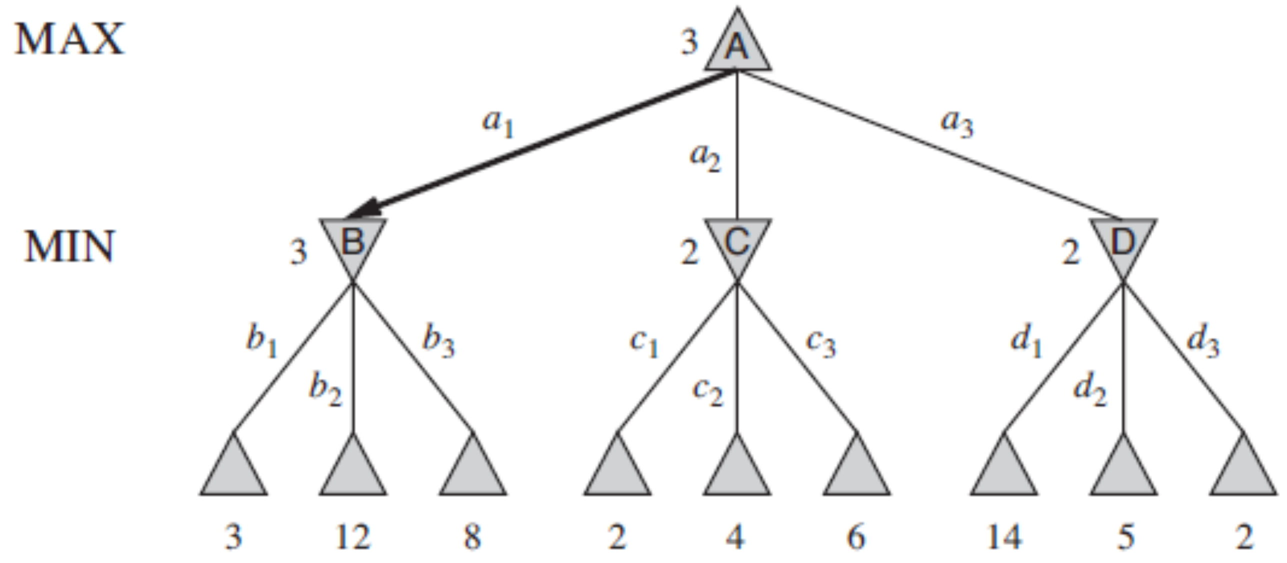
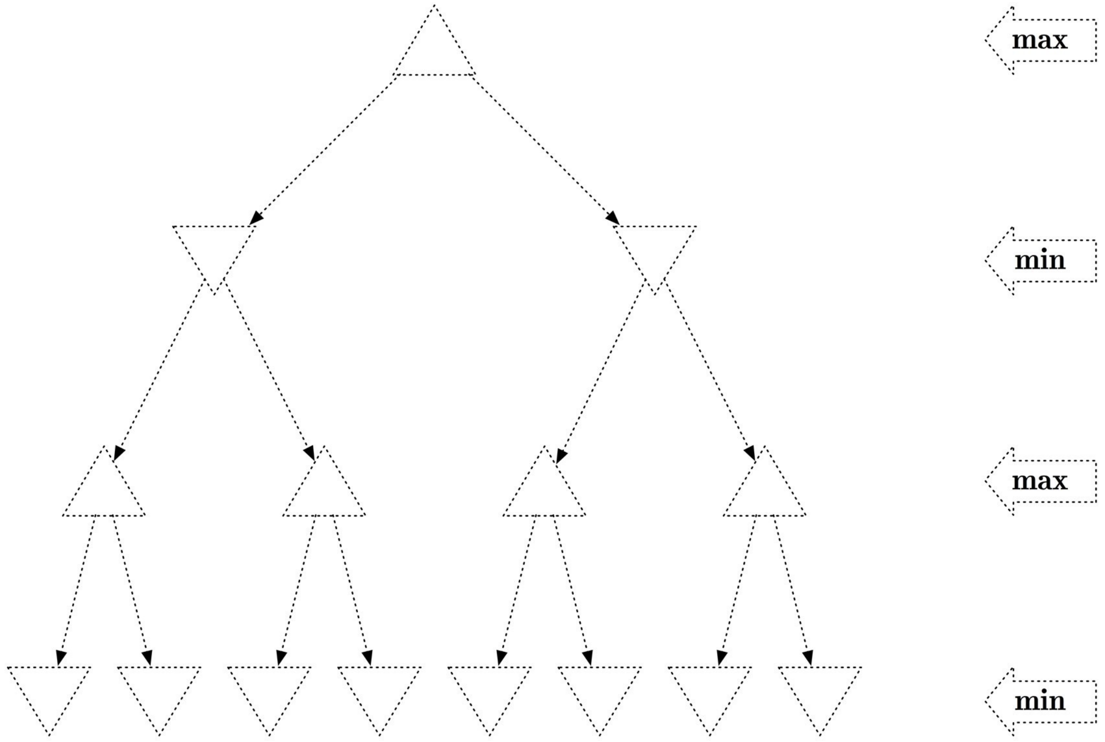


Figure 5.2 A two-ply game tree. The △ nodes are "MAX nodes," in which it is MAX's turn to move, and the ▽ nodes are "MIN nodes." The terminal nodes show the utility values for MAX; the other nodes are labeled with their minimax values. MAX's best move at the root is a_1, because it leads to the state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

MiniMax - Example

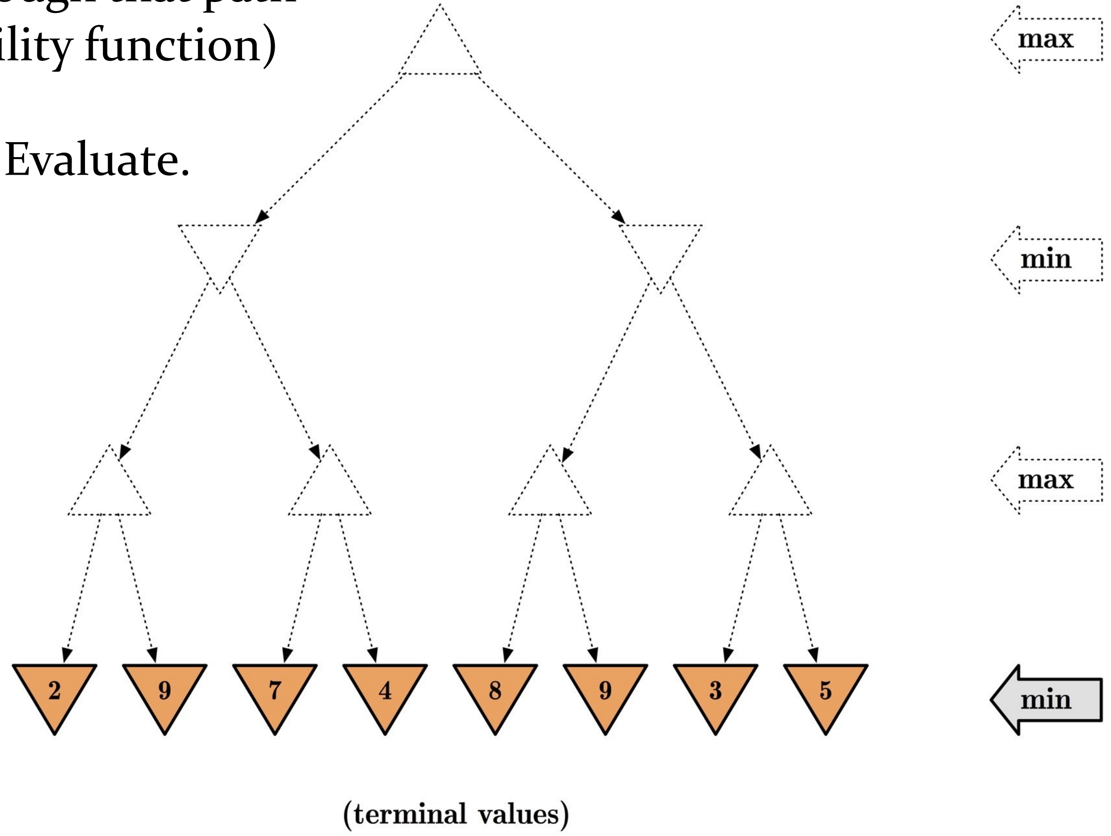
Here is a search tree. We want to select the best moves to make Max win the game.



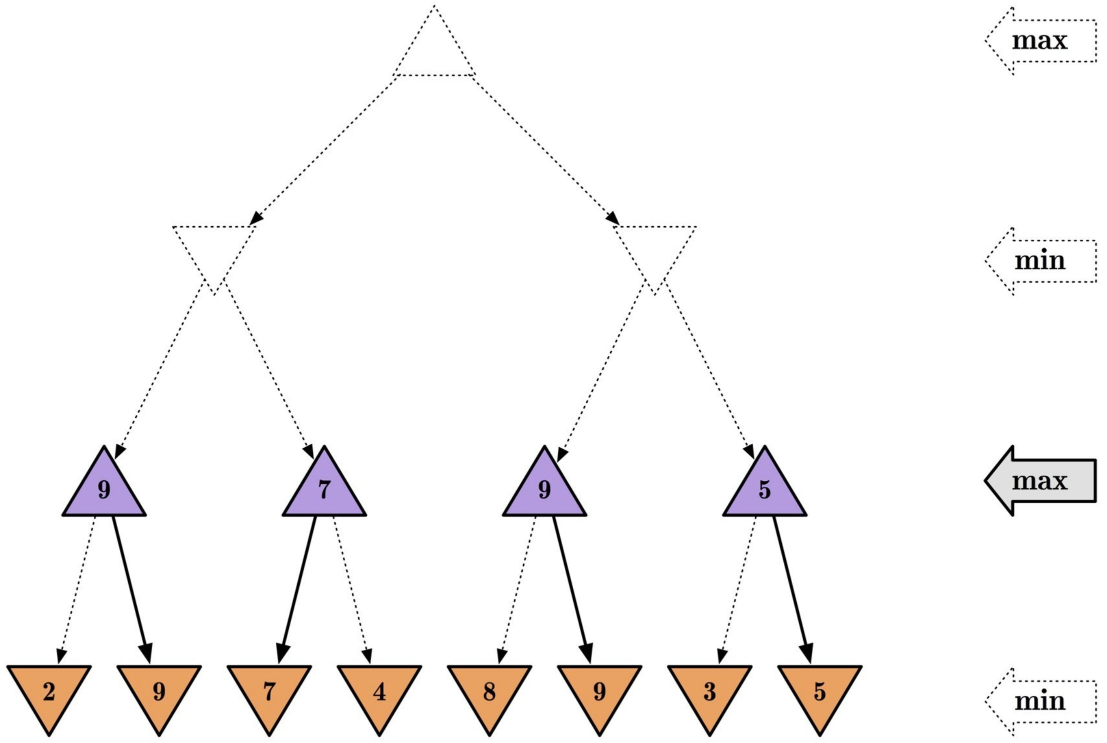
MiniMax - Example

The terminal values are some heuristic values that evaluate the game if we go through that path (values calculated from the utility function)

Start from the Root, DFS, and Evaluate.

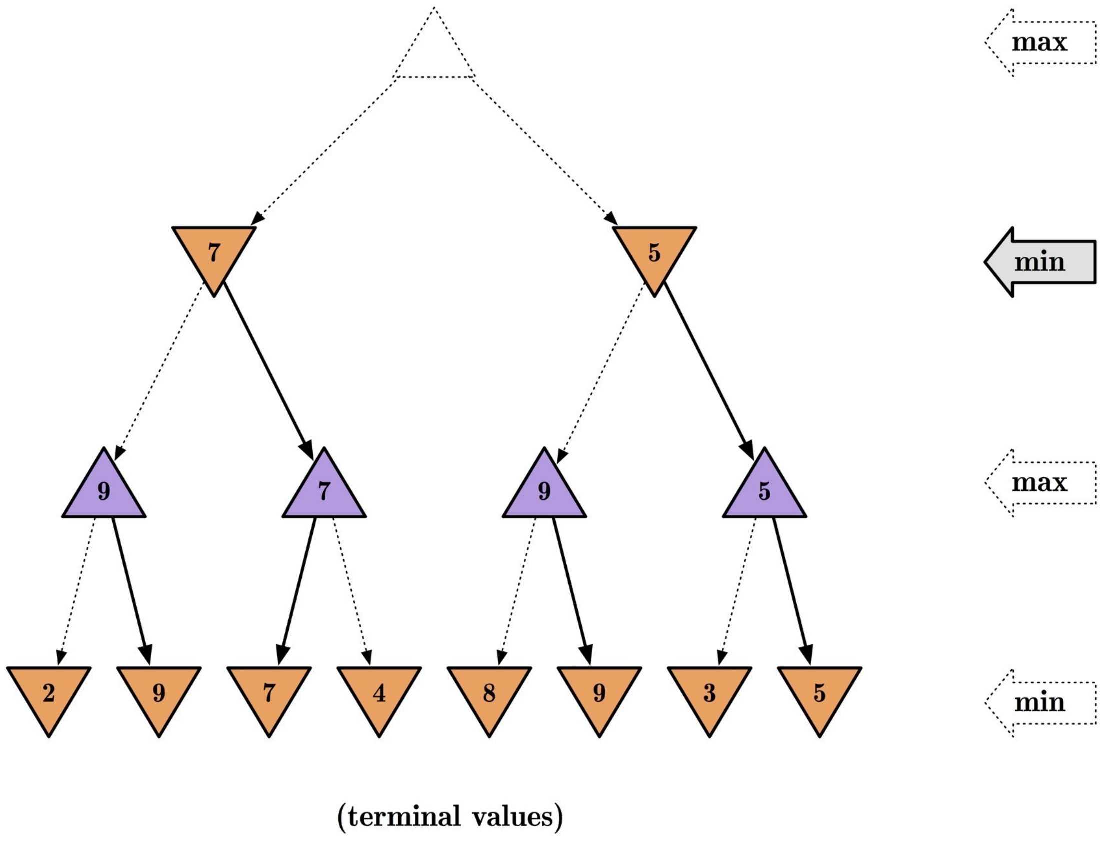


MiniMax - Example

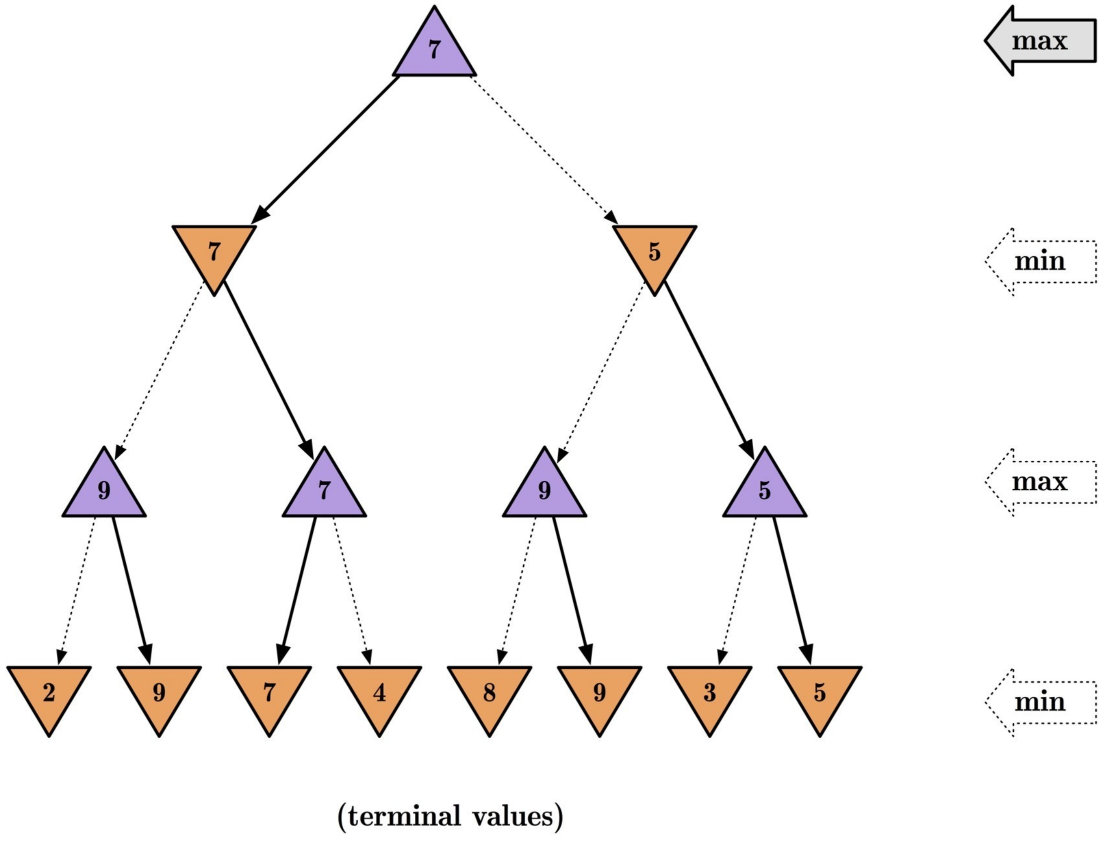


(terminal values)

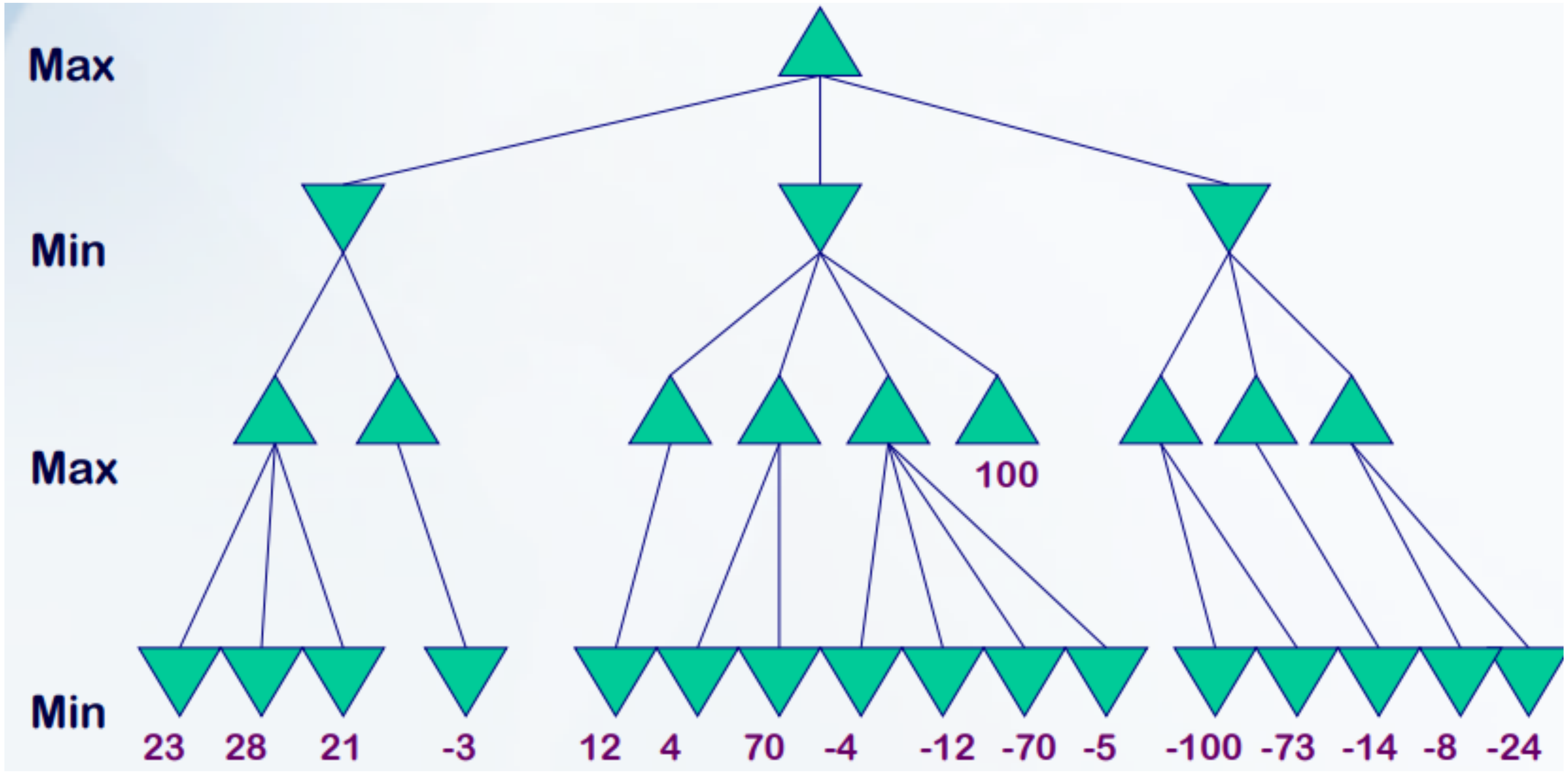
MiniMax - Example



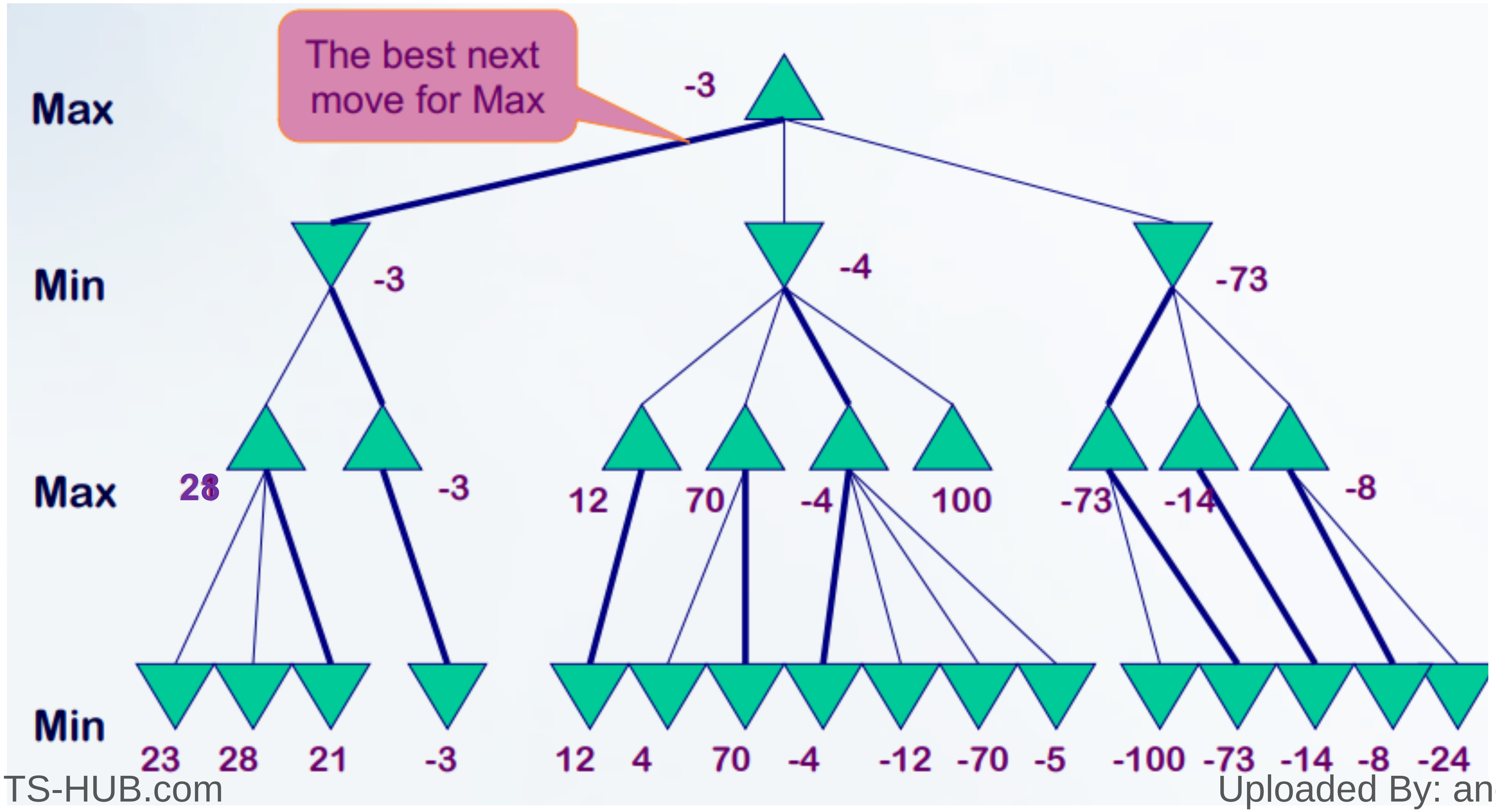
MiniMax - Example



MiniMax - Example



MiniMax - Example



Properties of Minimax Algorithm

- Complete: Yes (if tree is finite)
- Optimal: Yes (against an optimal opponent)
- Time complexity: A complete evaluation takes time b^m
- Space complexity: A complete evaluation takes space bm (depth-first exploration)
- For example, in chess, $b \approx 35$, $m \approx 100$ for "reasonable" games
  - The exact solution completely infeasible, since it is too big Instead, we limit the depth based on various factors, including time available.

* b is the number of legal moves at each point
* m is the maximum depth of the tree

Case of Limited Resources

- Problem: In real games, we are limited in time, so we can't search the leaves.
- To be practical and run in a reasonable amount of time, minimax can only search to some depth (especially with complex games)
- Solution:
  1. Replace terminal utilities with an evaluation function for non-terminal positions.
  2. Use Iterative Deepening Search (IDS).
  3. Use pruning: eliminate large parts of the tree.

ALPHA-BETA PRUNING ALGORITHM

MiniMax with Alpha-Beta Pruning

- A modification over Minimax algorithm
- The goal is to find a solution without expanding the entire tree. Thus it requires less memory and should be faster
- You don’t need to examine all branches of the tree to find the solution
- The higher the number of pruned branches, the greater effect in minimizing the search space of the tree

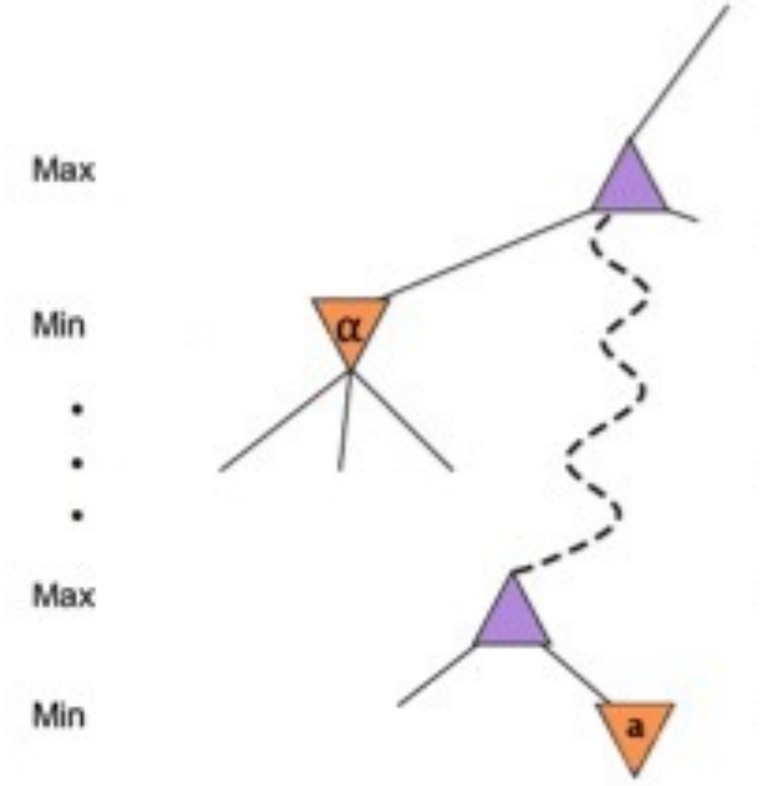
MiniMax with Alpha-Beta Pruning

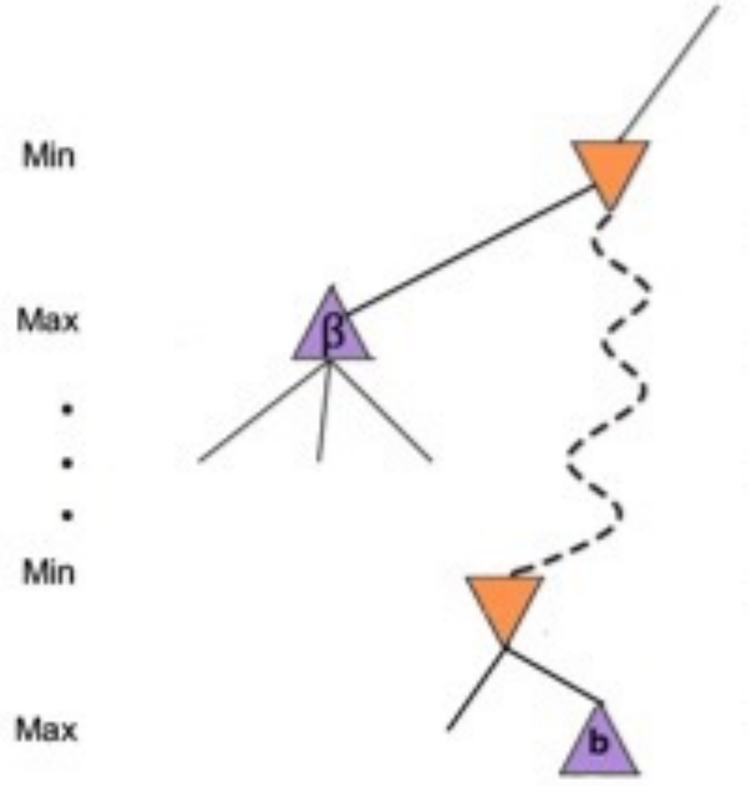
- During the (DFS) of the game tree, two variables are calculated: *alpha* and *beta*
- The alpha variable defines the best move that can be made to maximize (our best move)
- Beta variable defines the best move that can be made to minimize (the opposing best move)
- Prune the tree where $\alpha \geq \beta$

MiniMax with Alpha-Beta Pruning

- Strategy: Just like minimax, it performs a DFS.
- Parameters: Keep track of two bounds
  - α: largest value for Max across seen children (current lower bound on MAX's outcome)
  - β: lowest value for MIN across seen children (current upper bound on MIN's outcome)
- **Initialization**: $\alpha = -\infty, \quad \beta = \infty$
- **Propagation**: Send $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ values *down* during the search to be used for pruning.
  - Update $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ values by *propagating upwards* values of terminal nodes.
  - Update $\boldsymbol{\alpha}$ only at Max nodes and update $\boldsymbol{\beta}$ only at Min nodes.
- **Pruning**: Prune any remaining branches whenever $\alpha \geq \beta$

MiniMax with Alpha-Beta Pruning





- If α is better than node a for Max, then Max will avoid it, that is prune that branch.
- If β is better than node b for Min, then Min will avoid it, that is prune that branch.

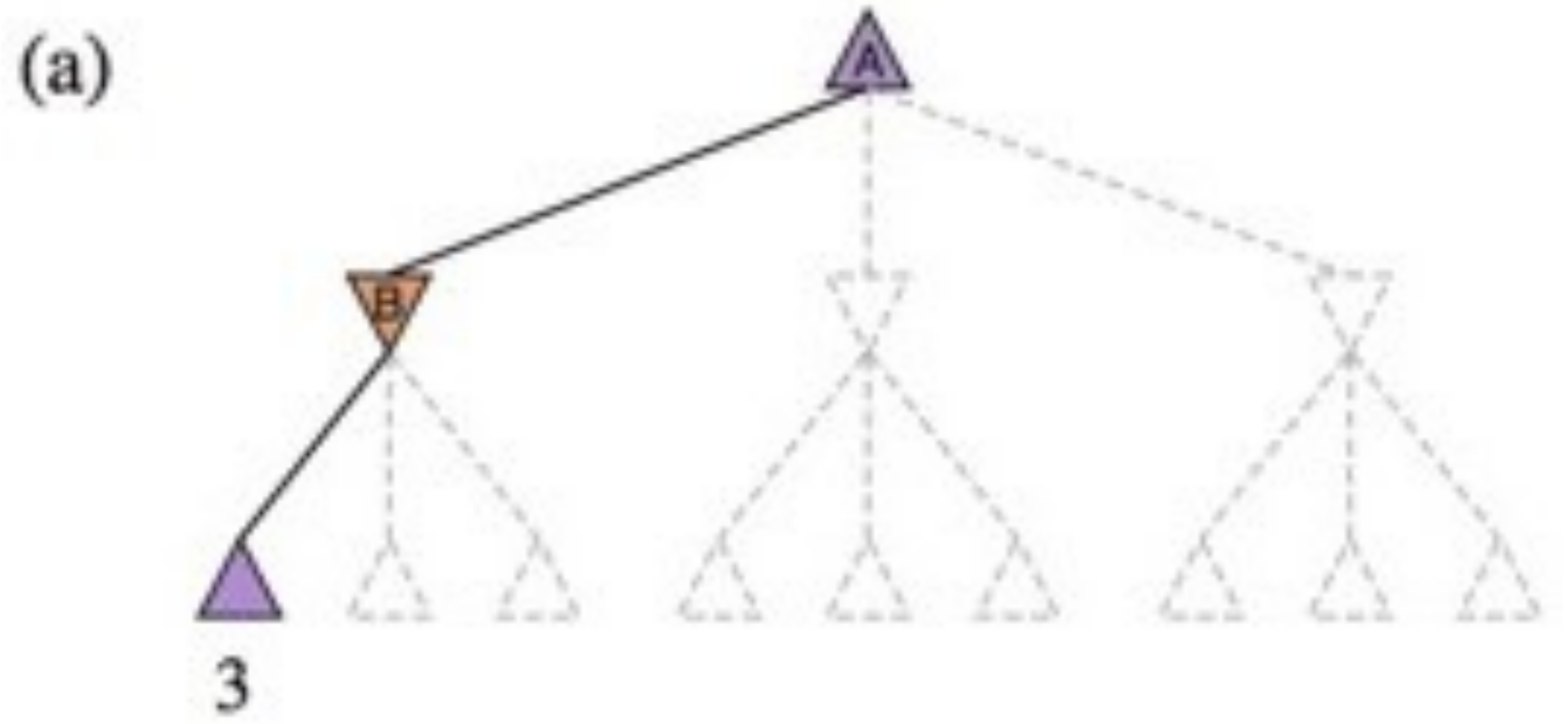
Alpha-Beta Pruning - Algorithm

```
/* Find the child state with the lowest utility value */

function MINIMIZE(state, $\alpha$, $\beta$)
    returns TUPLE of ⟨STATE, UTILITY⟩ :

    if TERMINAL-TEST(state):
        return ⟨NULL, EVAL(state)⟩

    ⟨minChild, minUtility⟩ = ⟨NULL, $\infty$⟩

    for child in state.children():
        ⟨ _, utility⟩ = MAXIMIZE(child, $\alpha$, $\beta$)

        if utility $<$ minUtility:
            ⟨minChild, minUtility⟩ = ⟨child, utility⟩

        if minUtility $\leq \alpha$:
            break

        if minUtility $< \beta$:
            $\beta$ = minUtility

    return ⟨minChild, minUtility⟩
```

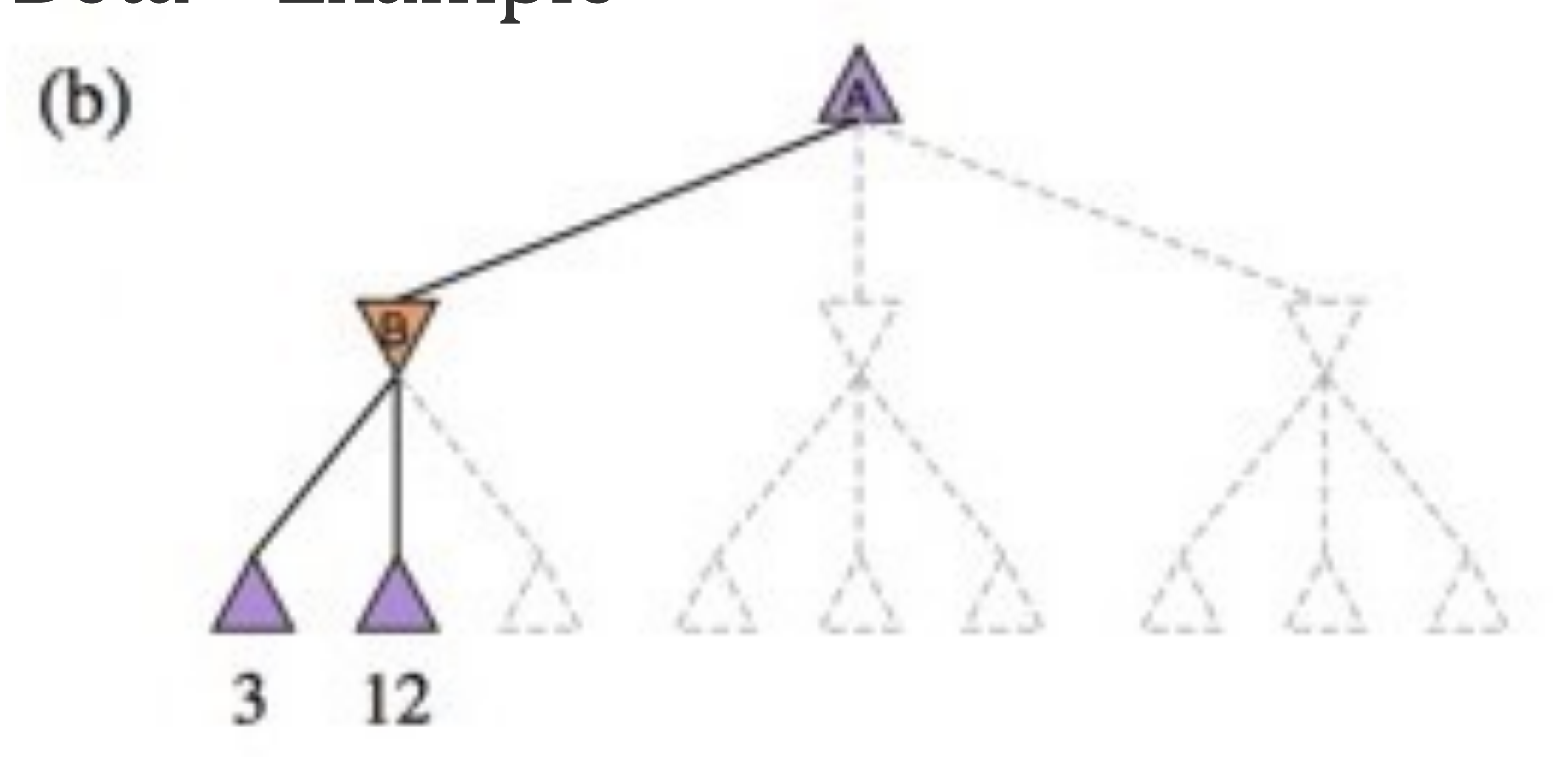
```
/* Find the child state with the highest utility value */

function MAXIMIZE(state, $\alpha$, $\beta$)
    returns TUPLE of ⟨STATE, UTILITY⟩ :

    if TERMINAL-TEST(state):
        return ⟨NULL, EVAL(state)⟩

    ⟨maxChild, maxUtility⟩ = ⟨NULL, $-\infty$⟩

    for child in state.children():
        ⟨ _, utility⟩ = MINIMIZE(child, $\alpha$, $\beta$)

        if utility $>$ maxUtility:
            ⟨maxChild, maxUtility⟩ = ⟨child, utility⟩

        if maxUtility $\geq \beta$:
            break

        if maxUtility $> \alpha$:
            $\alpha$ = maxUtility

    return ⟨maxChild, maxUtility⟩
```

```
/* Find the child state with the highest utility value

function DECISION(state)
    returns STATE :

    ⟨child, _⟩ = MAXIMIZE(state, $-\infty$, $\infty$)

    return child
```

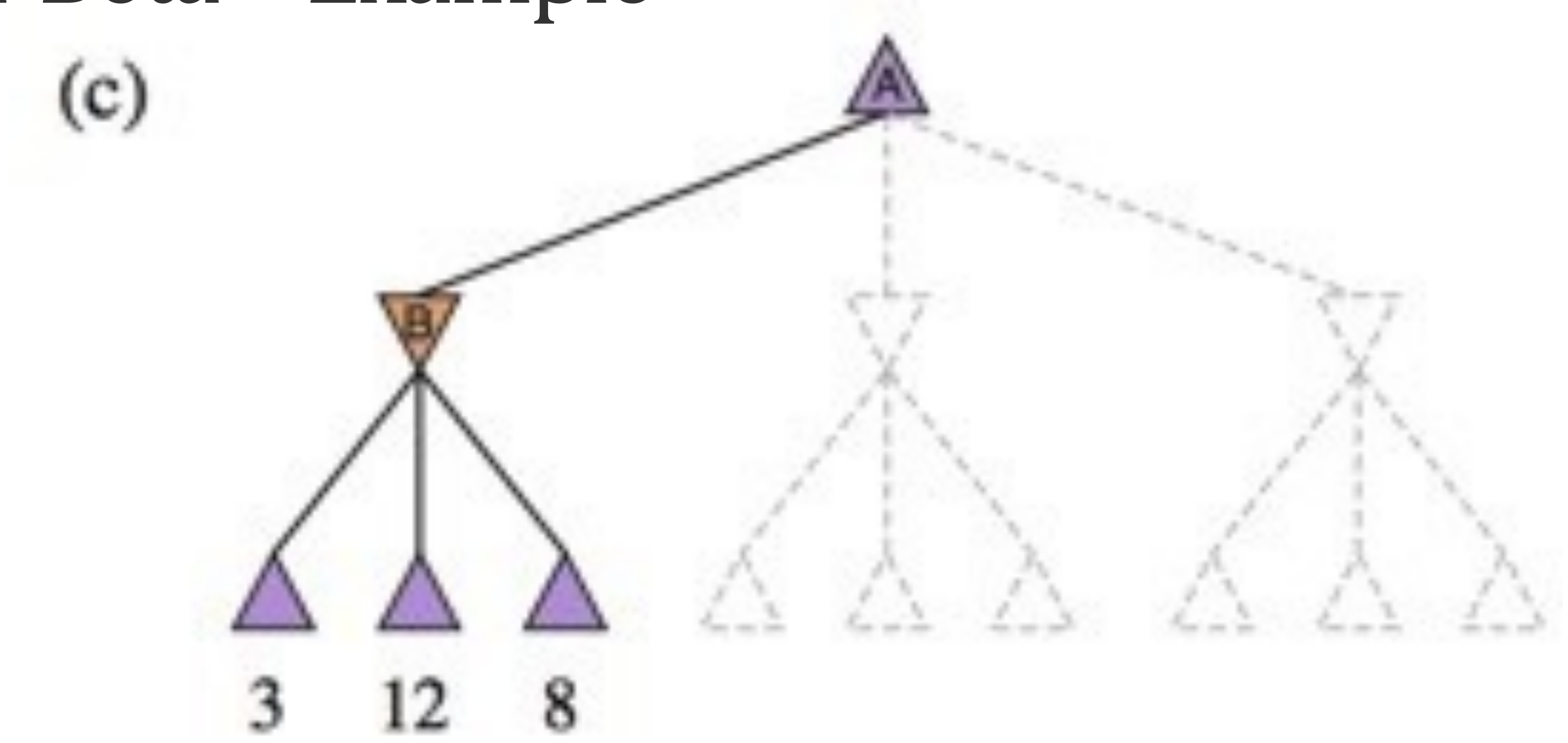
Alpha-Beta – Example



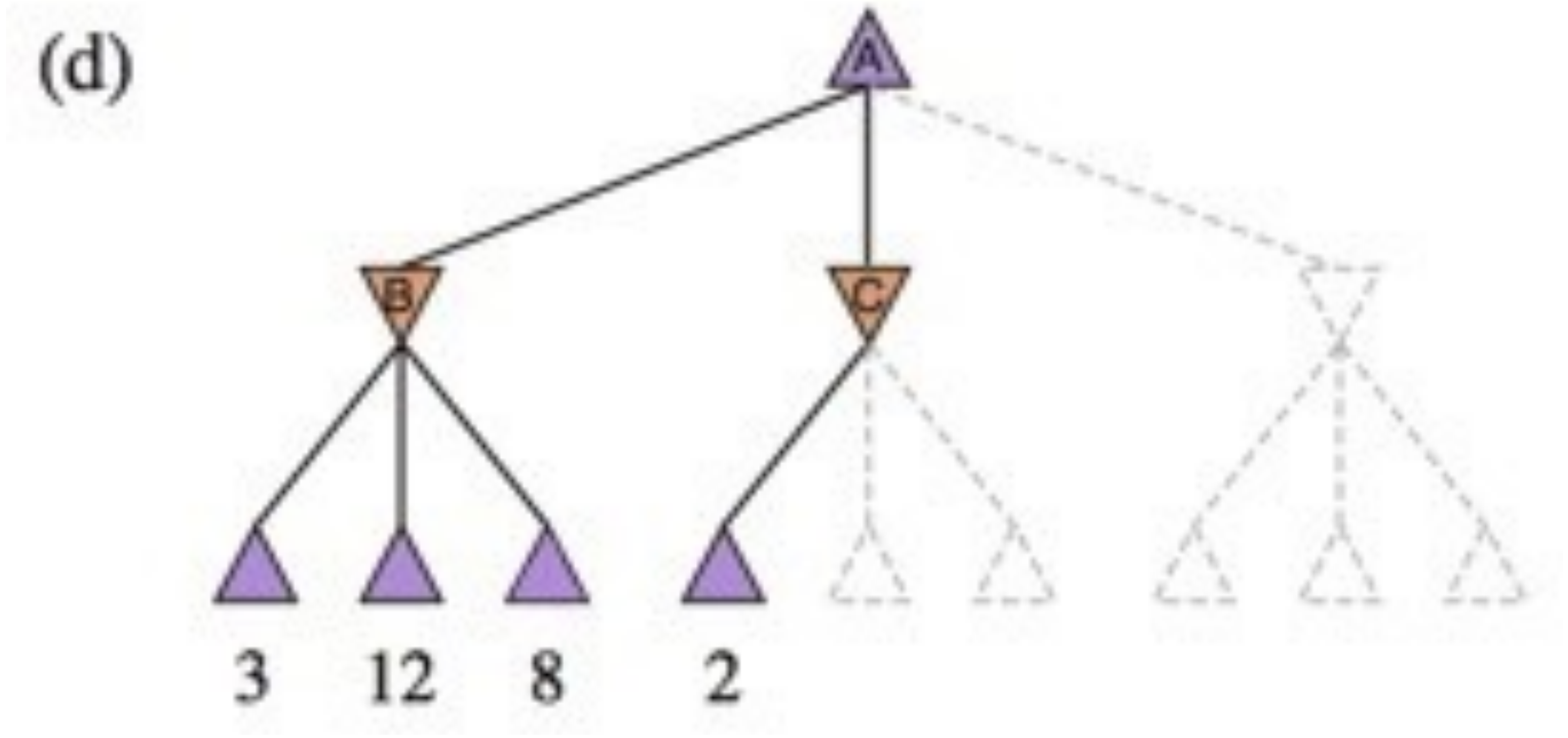
Alpha-Beta – Example



Alpha-Beta – Example

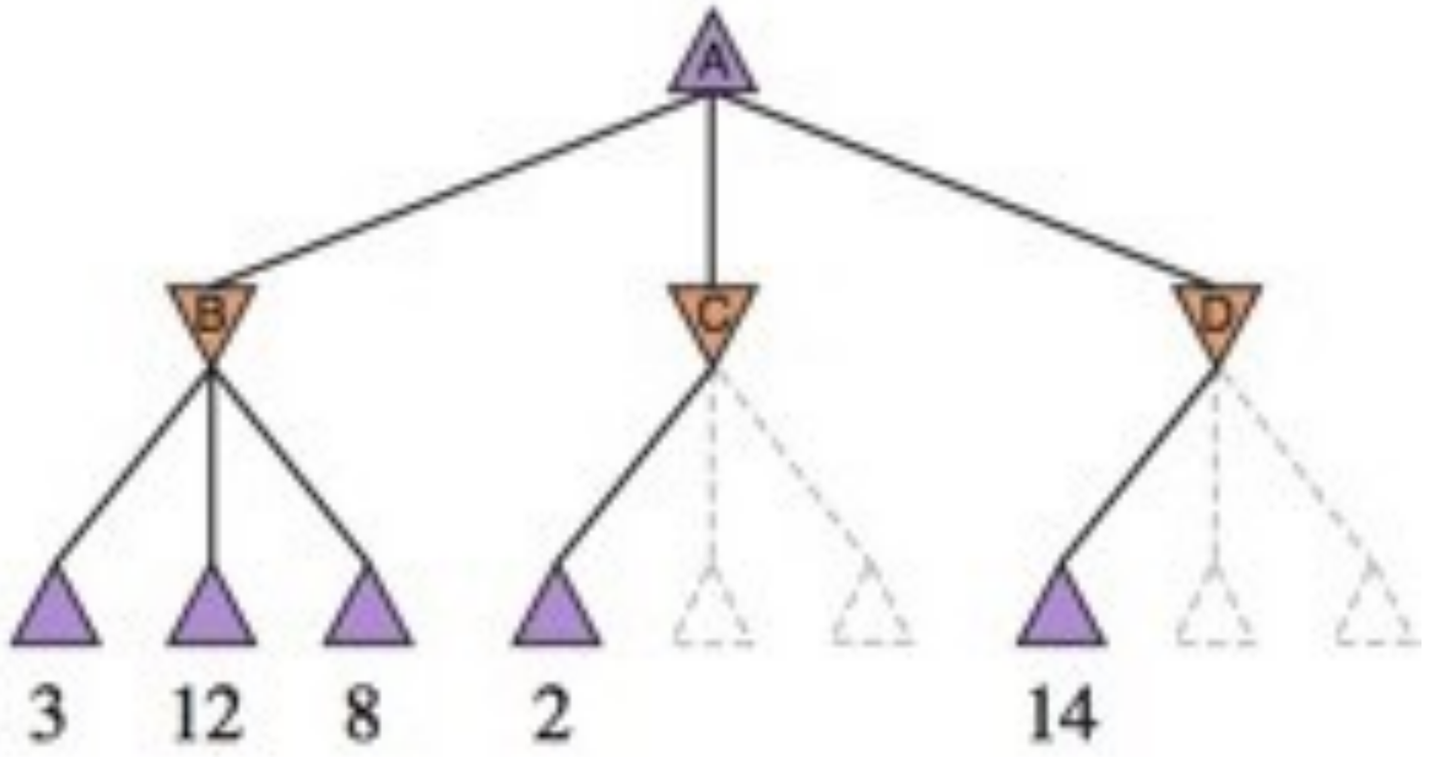


Alpha-Beta – Example

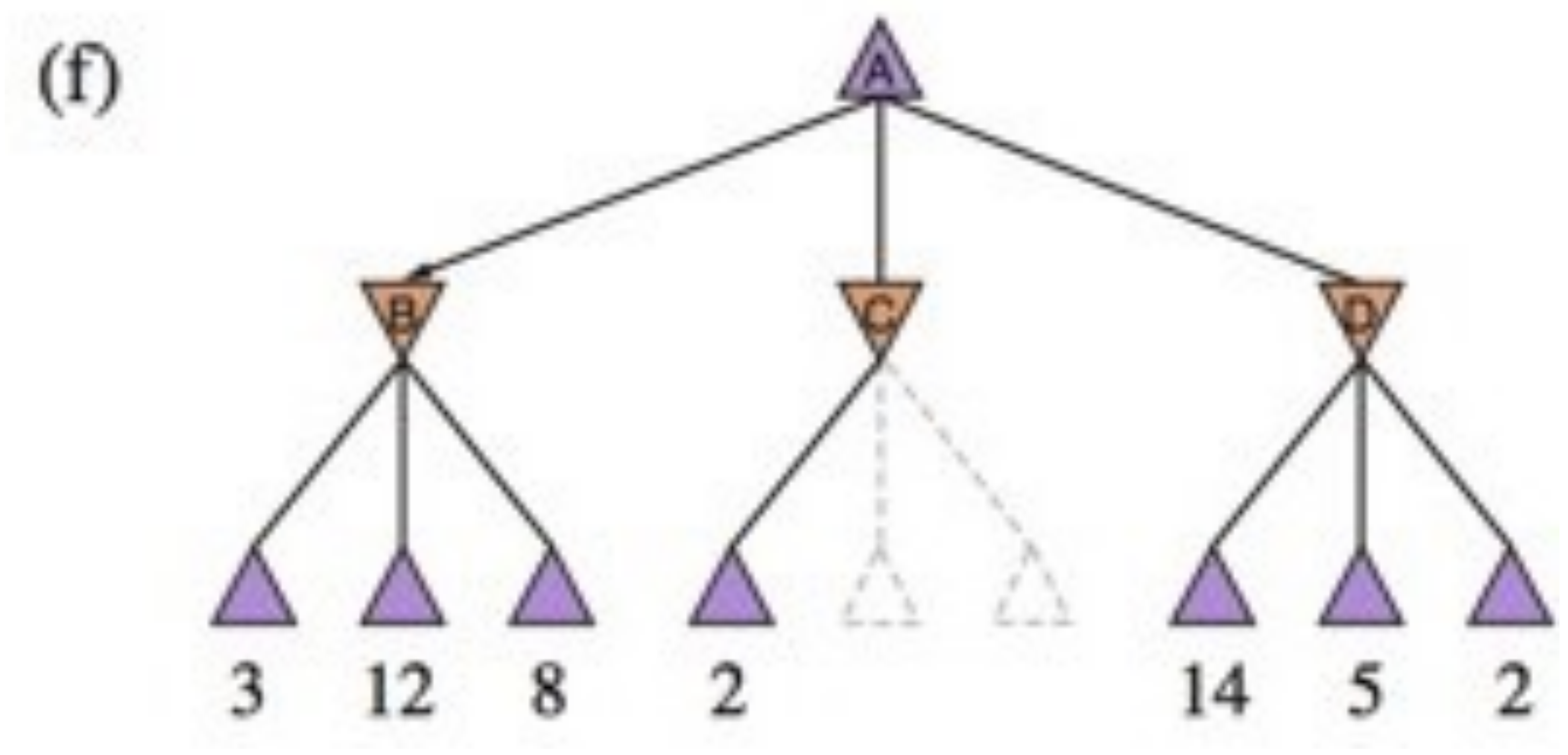


Alpha-Beta – Example

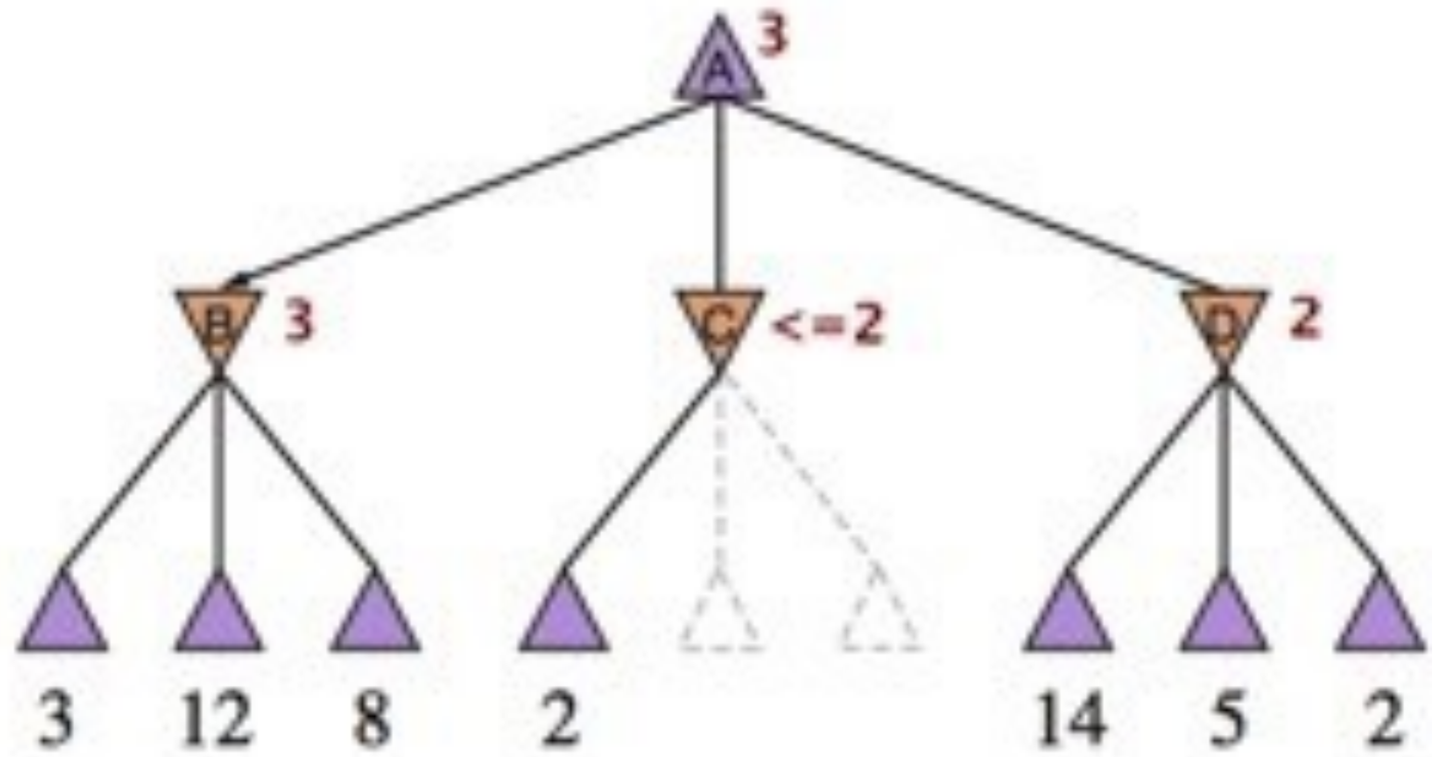
(e)



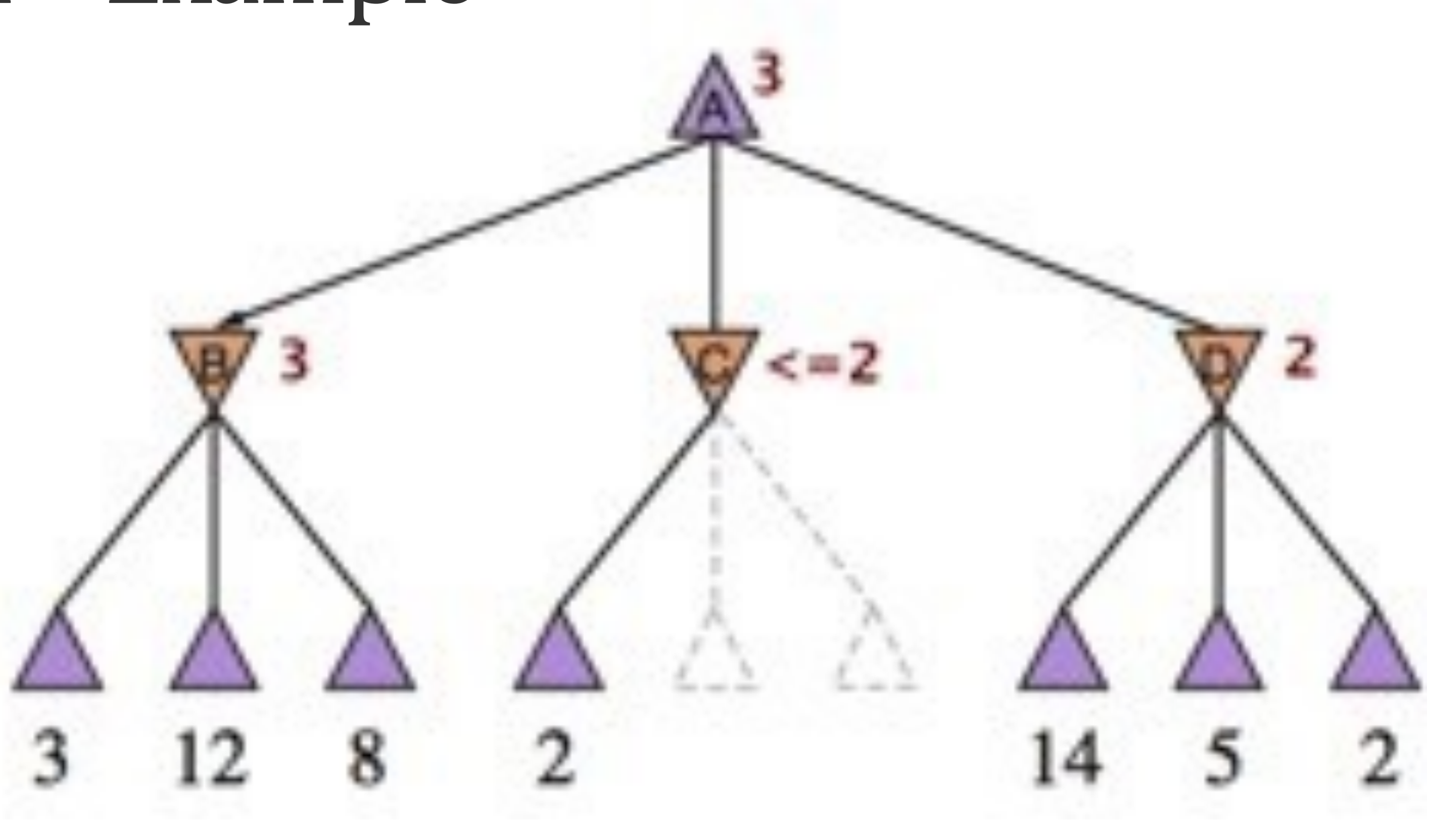
Alpha-Beta – Example



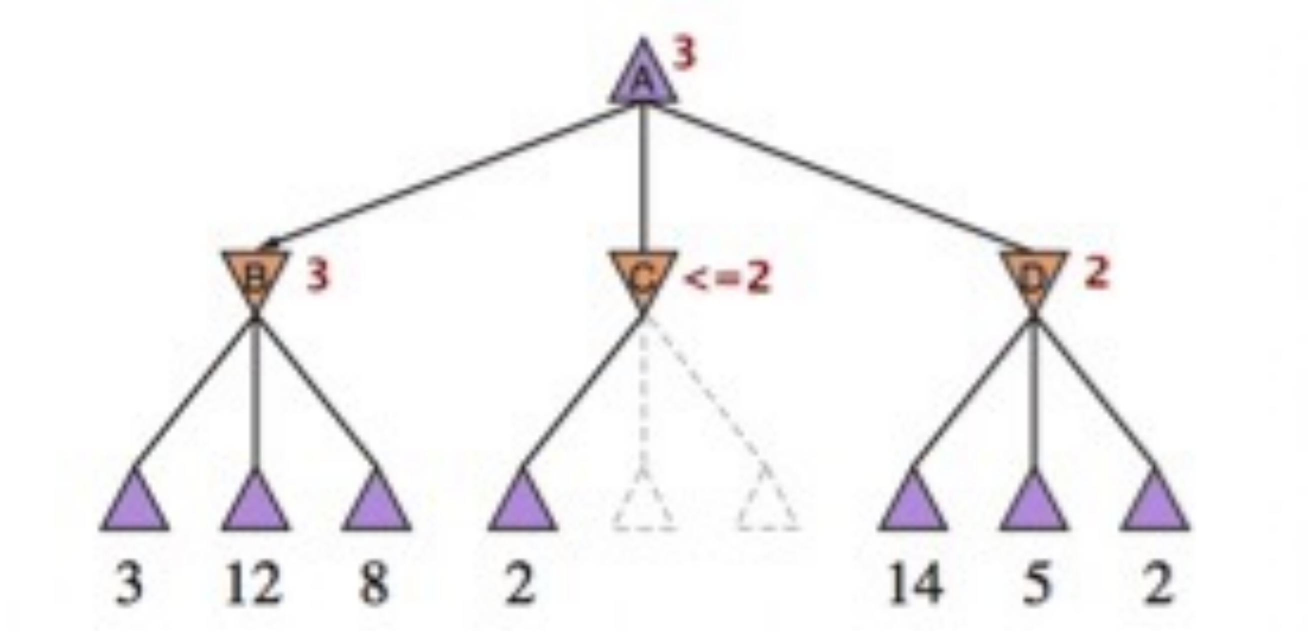
Alpha-Beta – Example



Alpha-Beta – Example



Move Order

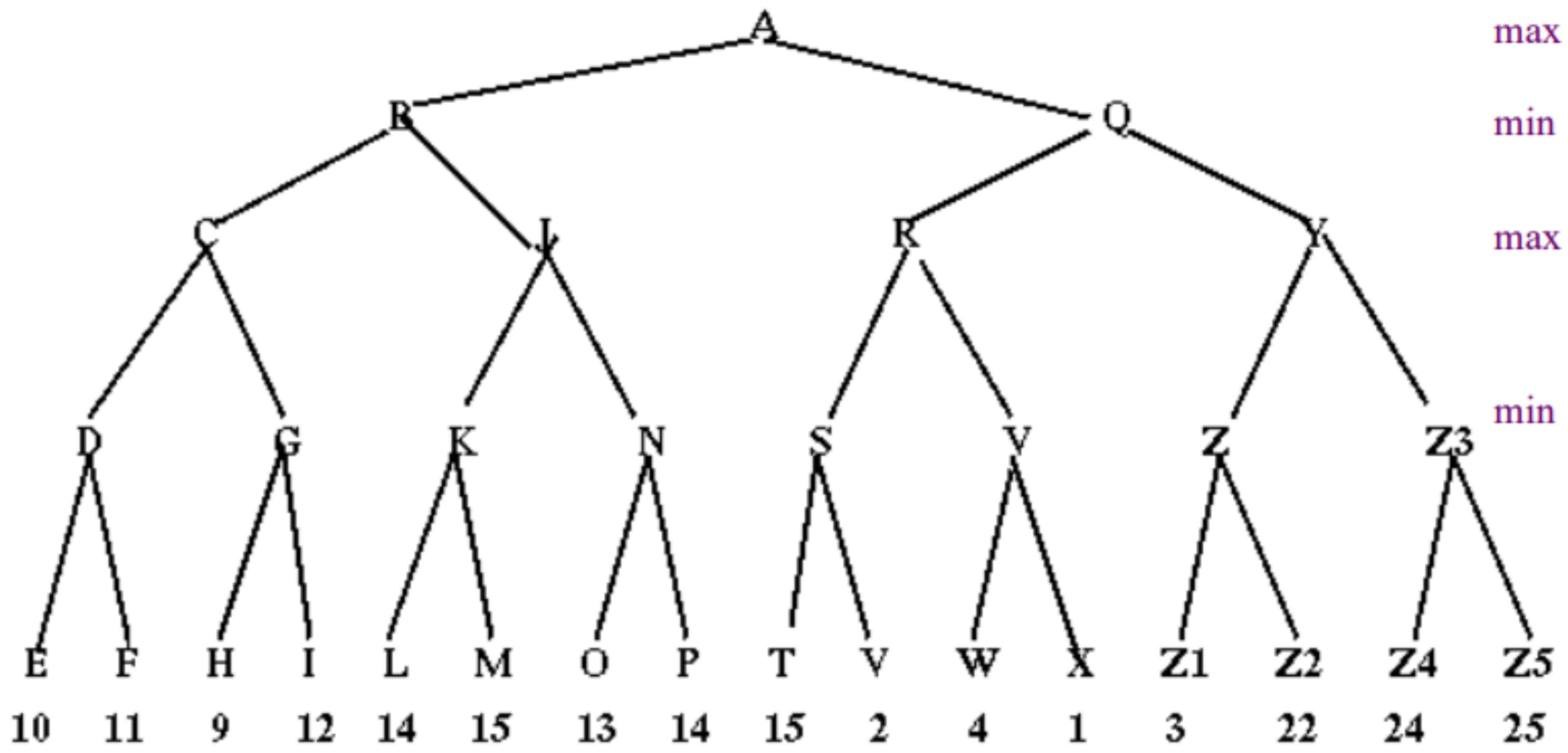


- It does matter as it affects the effectiveness of $\boldsymbol{\alpha - \beta}$ pruning.
- Example: We could not prune any successor of D because the worst successors for Min were generated first. If the third one (leaf 2) was generated first we would have pruned the two others (14 and 5).
- Idea of ordering: examine first successors that are likely best

Move Order

- Worst ordering: no pruning happens (best moves are on the right of the game tree). Complexity O(bm).
- Ideal ordering: lots of pruning happens (best moves are on the left of the game tree). This solves tree twice as deep as minimax in the same amount of time. Complexity $O(b^{m/2})$ (in practice). The search can go deeper in the game tree.
- How to find a good ordering?
  - Remember the best moves from shallowest nodes.
  - Order the nodes so as the best are checked first.
  - Use domain knowledge: e.g., for chess, try order: captures first, then threats, then forward moves, backward moves.
  - Bookkeeping the states, they may repeat!

Alpha-Beta – Example



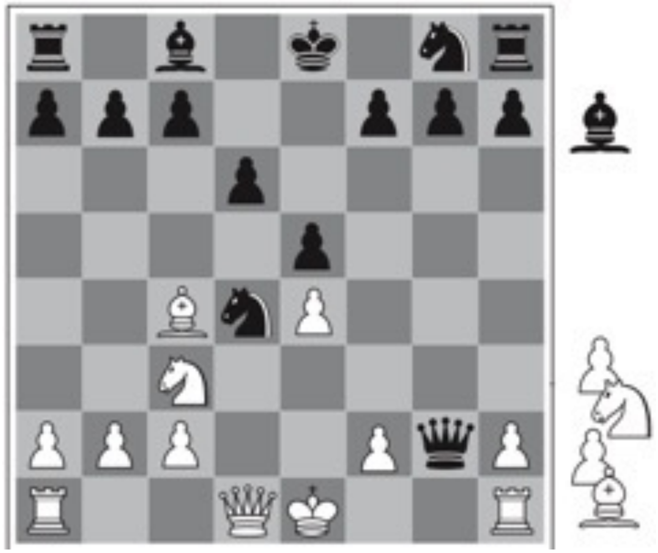
Real-Time Decision

- Minimax: generates the entire game search space
- Alpha-Beta algorithm: prune large chunks of the trees
- Alpha-Beta still has to go all the way to the leaves
- Impractical in real-time (moves has to be done in a reasonable amount of time)
- Solution:
  - bound the depth of search (cut search) and
  - **replace** *utiliy*(*s*) **with** *eval*(*s*), an evaluation function to **estimate** value of current board configurations

Real-Time Decision

- eval(s) is a heuristic at state s
- E.g., Chess: Value of all white pieces Value of all black pieces turn non-terminal nodes into terminal leaves!
- An ideal evaluation function would rank terminal states in the same way as the true utility function; but must be fast
- Typical to define features, make the function a linear weighted sum of the features
- Use domain knowledge to craft the best and useful features.

Real-Time Decision



- How does it works?
- Select useful features $f_1, \ldots, f_n$
  e.g., Chess: # pieces on board, value of pieces (1 for pawn, 3 for bishop, etc.)
- Weighted linear function: $eval(s) = w_1 f_1 + w_2 f_2 + \ldots + w_n f_n$, which is a combination of features and are weighted by their relevance
- Weights are learnt from the examples
- Deep blue uses about 6,000 features!
- This is referred to as Utility Evaluation Function

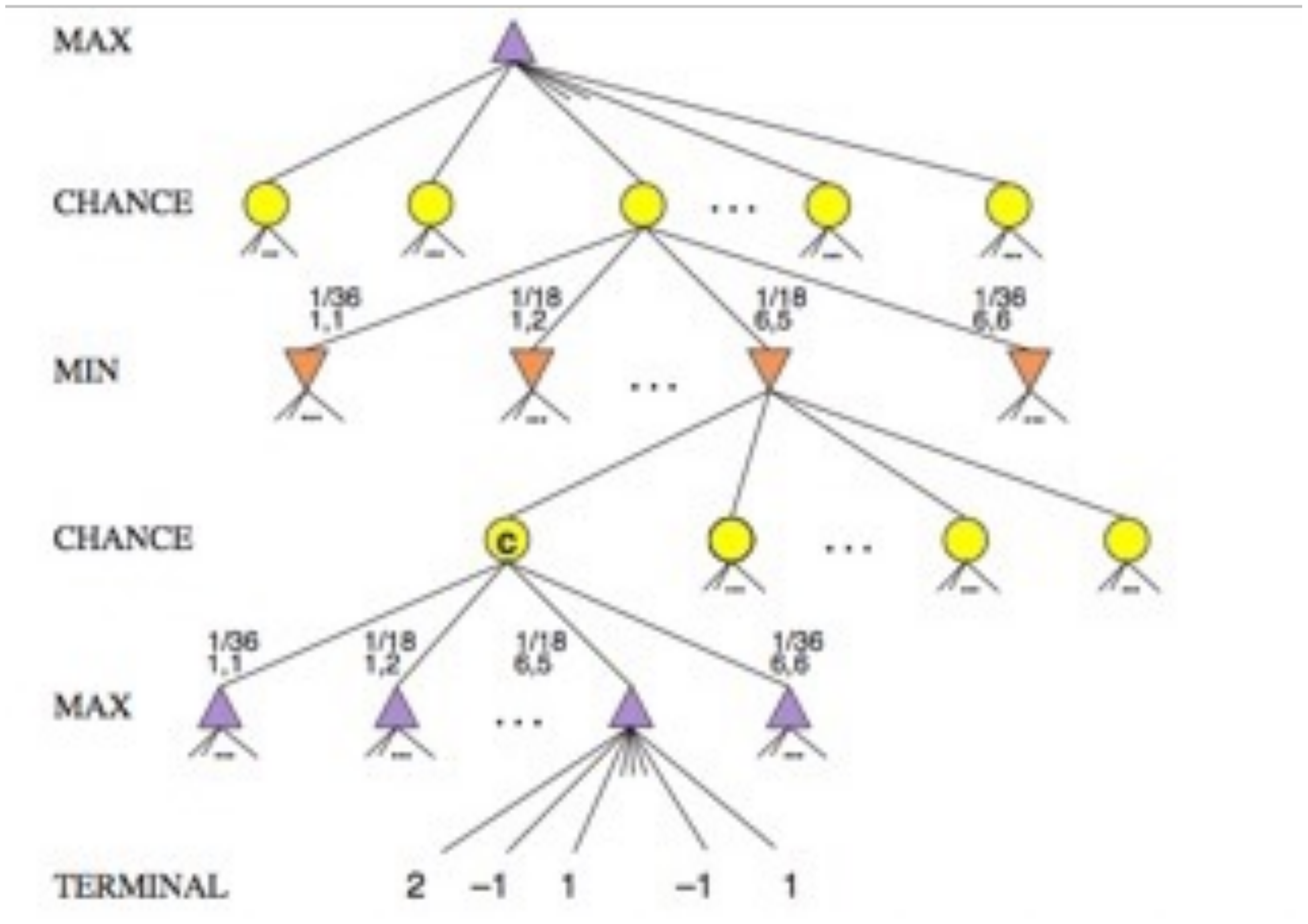
Utility Evaluation Function

- Determines the performance of a game-playing program
- Game-specific!
- Values for terminal nodes (or at least their order) must be the same
- The function should reflect the actual chances of winning
- Notice: quality of utility function is based on:
  - What features are evaluated
  - How those features are scored
  - How the scores are weighted/combined
- Absolute utility value doesn’t matter – relative value does.

Stochastic Games

- Include a random element (e.g., throwing a die).
- Include chance nodes.
- Backgammon: old board game combining skills and chance.
- The goal is that each player tries to move all of his pieces off he board before his opponent does.

Stochastic Games



- Partial game tree for Backgammon.

Stochastic Games - Algorithm

- *Expectminimax* generalized Minimax to handle chance nodes as follows:
- If state is a Max node then
  return the highest *Expectminimax-Value* of Successors(state)
- If state is a Min node then
  return the lowest *Expectminimax-Value* of Successors(state)
- If state is a chance node then
  return average of *Expectminimax-Value* of Successors(state)

Conclusion

- Games are modeled in AI as a search problem and use heuristic to evaluate the game.
- Minimax algorithm choses the best most given an optimal play from the opponent.
- Minimax goes all the way down the tree which is not practical give game time constraints.
- Alpha-Beta pruning can reduce the game tree search which allow to go deeper in the tree within the time constraints.
- Pruning, bookkeeping, evaluation heuristics, node re-ordering and IDS are effective in practice.