CS60045 Artificial Intelligence Autumn 2023

Adversarial Search

Games

So far our search was carried out by a single intelligent agent.

What if there are **multiple agents**?

Some agents may be human. Others are machines (intelligent agents).

In many situations (like **economics**), the agents compete with one another.

Each agent tries to win by defeating others.

More complex situations like **coalitions** by groups of agents to defeat others are also practical.

Game theory deals with a formal study of such situations.

In our context, we will take the word **game** literally.

However, the techniques we will discuss apply to more general situations too.

Two-player games

Two professors Dr. H. Maxitra (MAX) and Dr. D. Minijit (MIN) play a competitive game.

We will support MAX. **All computations will be done by MAX.**

The game starts at an **initial state** (not necessarily the state at the very beginning of the game).

The **moves** of MAX and MIN alternate (a move is better called a **ply**). The intelligent agent springs into action before MAX's every move, and recommends what the best move for MAX is.

For every move, there is a set of **allowed actions** dictated by the rules of the game.

A **terminal situation** is one when the game ends.

The **utility** of a player *P* is his score at a terminal situation. It may be 1, –1, or 0 standing for win, loss, and draw. If the game involves points, the utility may be the total points accrued by *P* during the game, or *P*'s total points minus the other player's total points.

Types of games

Partially / Fully observable: Is there any information hidden from the players? Or not?

Tic-Tac-Toe, Boxes-and-Dots, Ludo, Chess are fully observable

Many card games and Scrabble are not fully observable, because no player knows the opponent's hand fully.

Deterministic / Random: Is there any source of randomness involved in the rules of the game?

Tic-Tac-Toe, Boxes-and-Dots, and Chess are deterministic.

Ludo involves throwing dices. The progress of the game depends on the outcomes of the throws, and cannot be predicted beforehand. Likewise, each Scrabble player picks his/her tiles blindfoldedly from the pool.

Game trees

A game involves a state space.

The initial state stands for the initial state of the game.

Each state transition is governed by one of the allowed actions during a move.

The complete state space is represented as a tree called the **game tree**.

The game tree includes all possible moves of both the players.

The same nodes may be generated multiple times, but we treat them as separate nodes (so game trees are not configured as graphs).

The game tree may be infinite if the state space itself is infinite, or the rules of the game allow cycles.

NAP: A fully observable deterministic two-player game

MAX and MIN start with the *N* integers 1, 2, 3, … , *N*. Each of these integers is initially unmarked.

Moves alternate between MAX and MIN. Assume that MAX makes the first move.

In each move, the player marks (like strikes off) a remaining unmarked integer. However, **N**o three of the marked integers can be in **A**rithmetic **P**rogression.

The player who fails to make a move loses.

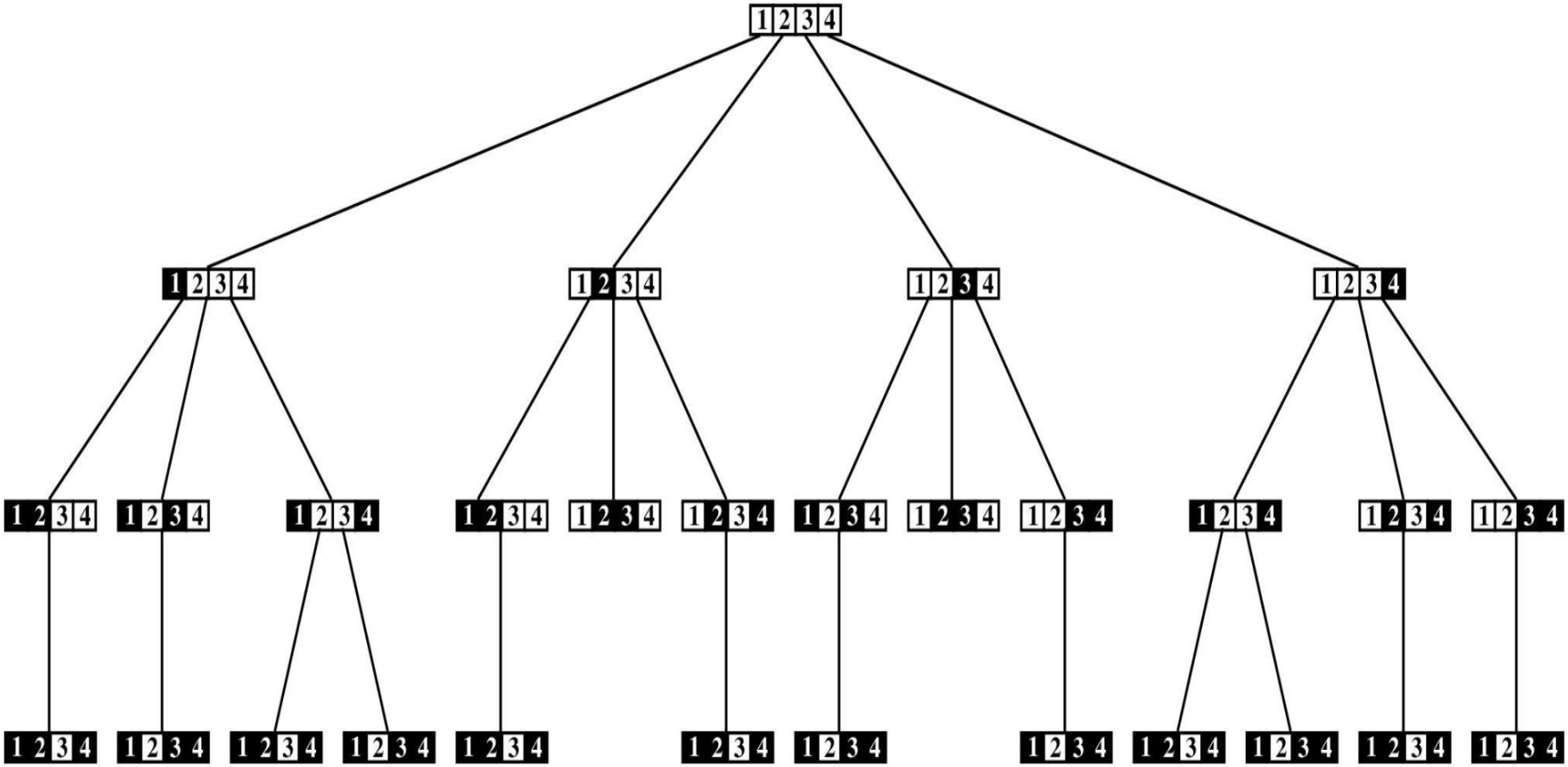
For *N* = 1, 2, 3, the game is trivial.

For *N* = 4, the NAP rule implies that three consecutive integers cannot be marked. That's all.

For *N* = 5, there is another restriction: 1, 3, and 5 cannot be all marked.

For larger values of *N*, there are many other forbidden marked triples like (3, 10, 17), (1, 19, 37).

The game tree for NAP with $N = 4$



What does the NAP(4) tree tell?

If MAX marks 1 or 4 in the first move, then no matter what MIN and MAX do next, MAX will win.

If MAX marks 2 or 3 in the first move, then MIN can win (unless MIN plays dumbly).

We say that MAX has a **winning strategy**.

Exercise: What is the winning strategy (if any) of MAX for $N = 5$? For $N = 6$?

Note: The size of the maximum AP-free subset of 1, 2, 3, …, N is called the **Salem–Spencer number** $SS(N)$. So $SS(N)$ is equal to the height of the NAP(N) game tree.

The game tree grows very rapidly with N, so it is very difficult to compute $SS(N)$ except for small N.

N	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
$SS(N)$	3	4	4	4	4	5	5	6	6	7	8	8	8	8	8	8	9

How can MAX find the winning strategy?

We assume (for now) that we have (well, MAX has) time to explore the entire game tree.

Both MAX and MIN will try to win and play optimally.

We look at the utilities at the terminal (end-of-game) nodes, and work up the tree to figure out the best move of MAX at the beginning (and also at every step).

The example on the next slide assumes that the utility of a terminal node is calculated as

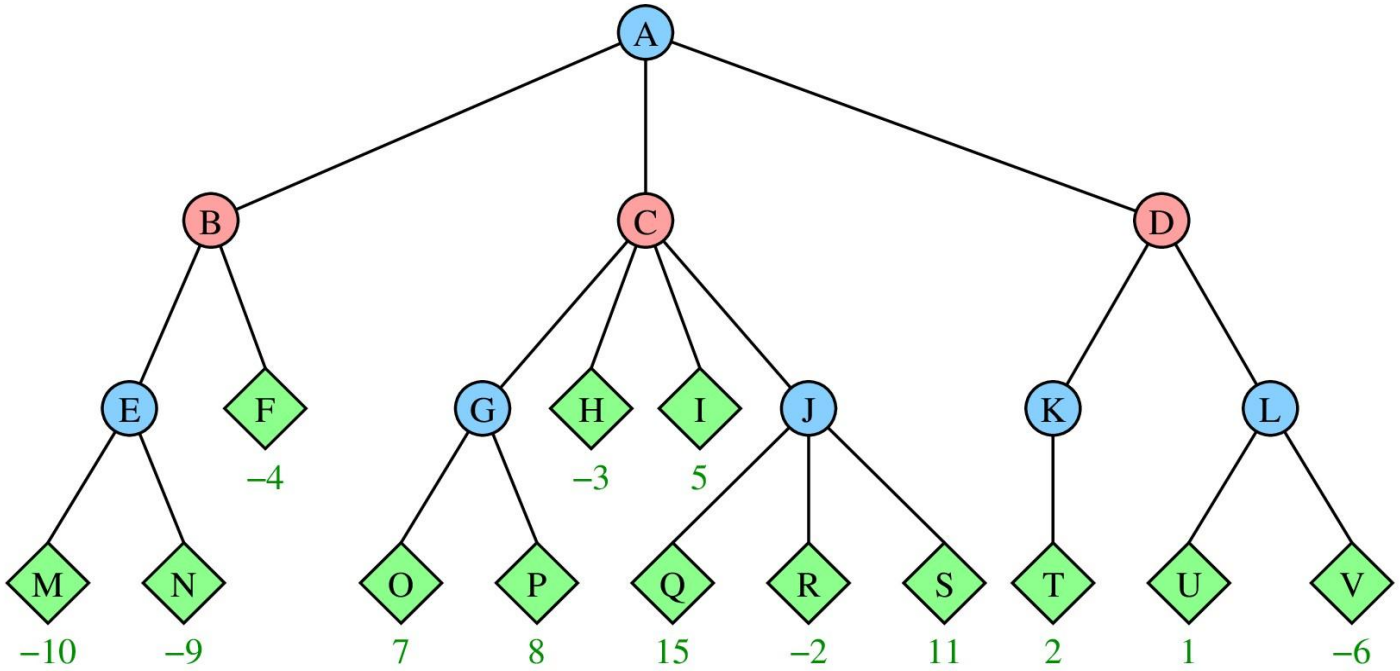
points(MAX) – points(MIN).

This is from the viewpoint of MAX.

MAX will try to maximize this quantity.

MIN will try to minimize this quantity. (For MIN, the utility is points(MIN) – points(MAX).)

Example of winning strategy



If MAX moves to B, then no matter what happens next, MIN will win.

If MAX moves to C, he can win only if MIN makes the bad move G, I or J. But MIN should choose H.

If MAX moves to D, then no matter what MIN does, MAX can win provided that MAX does not end up in V.

Minimax values

Let n be a node in the tree. Minimax(n) quantifies the odds that MAX can win from n.

If n is a terminal node, then

$$\text{Minimax}(n) = \text{Utility}(n).$$

Otherwise let $n_1, n_2, \ldots, n_k$ be the children of n in the game tree.

If it is MAX's turn to make a move from n, then

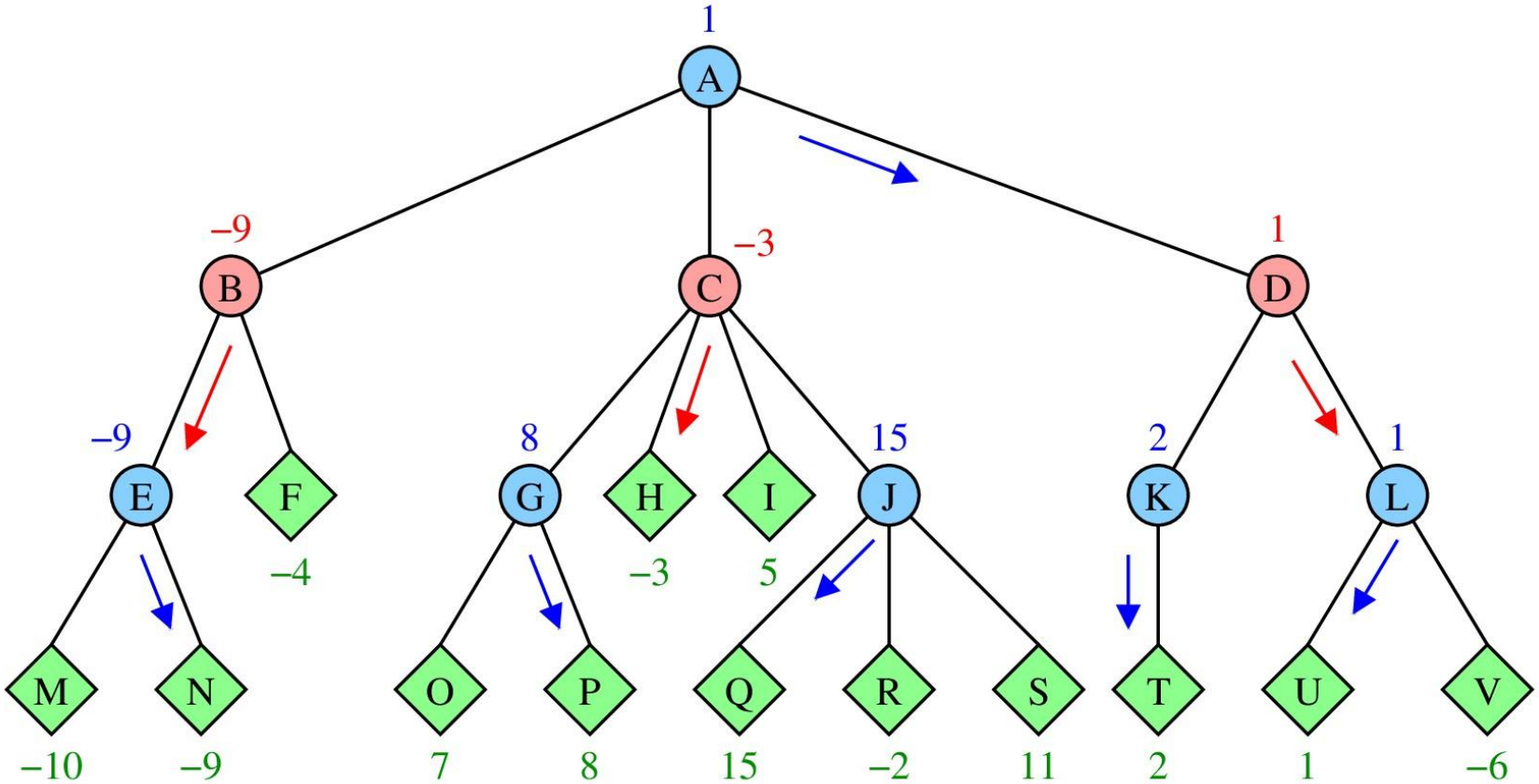
$$\text{Minimax}(n) = \textbf{max}\left(\text{Minimax}(n_1), \text{Minimax}(n_2), \ldots, \text{Minimax}(n_k)\right).$$

If it is MIN's turn to make a move from n, then

$$\text{Minimax}(n) = \textbf{min}\left(\text{Minimax}(n_1), \text{Minimax}(n_2), \ldots, \text{Minimax}(n_k)\right).$$

The optimal moves are those where the maximum or the minimum is attained.

Example of Minimax calculations



The Minimax Algorithm

Minimax (n)

If (n is a terminal node), then return (Utility(n), NoMove).
Expand n. Let n_1, n_2, …, n_k be all the children of n.
If (n is a MAX node)
    Initialize best = $-\infty$ and bestchild = none.
    For i = 1, 2, …, k, repeat:
        Get (cval, cmove) = Minimax(n_i).
        If (cval > best), update best = cval and bestchild = n_i.
else (that is, if n is a MIN node)
    Initialize best = $+\infty$ and bestchild = none.
    For i = 1, 2, …, k, repeat:
        Get (cval, cmove) = Minimax(n_i).
        If (cval < best), update best = cval and bestchild = n_i.
Return (best, bestchild).

Minimax algorithm: Discussion

This algorithm is run by MAX before making **every** move.

This algorithm is based upon the assumption that both MAX and MIN play optimally.

The algorithm prepares MAX for the worst-case scenario.

If one player does not play optimally, then the other player will not play worse than in the worst case.

The algorithm gives MAX not only a winning strategy, but a way to bag the maximum points too.

In reality, if MAX sees no chance to win from a given position, he may choose a non-optimal move, and MIN, if deprived of enough computing resources, may play non-optimally too.

Let h be the height of the game tree, and b the branching factor.
Time complexity is $O(b^h)$. If the whole tree is remembered, the space complexity is $O(b^h)$ too.

If only the topmost recommendation at the root node is used, then the tree can be forgotten, and the space usage is only $O(hb)$ (or only $O(h)$ if the child nodes are generated one at a time). But then, before every move, the Minimax algorithm is called. That is not that bad (and is in fact the **preferred strategy**), because as the game progresses, the tree keeps on becoming smaller (but this is not the main reason).

Pruning: No need to explore the entire game tree

The space requirement is low if the exploration is done in the depth-first fashion.

But the running time $O(b^h)$ may be prohibitively high.

There is a necessity to cut down *unnecessary* work.

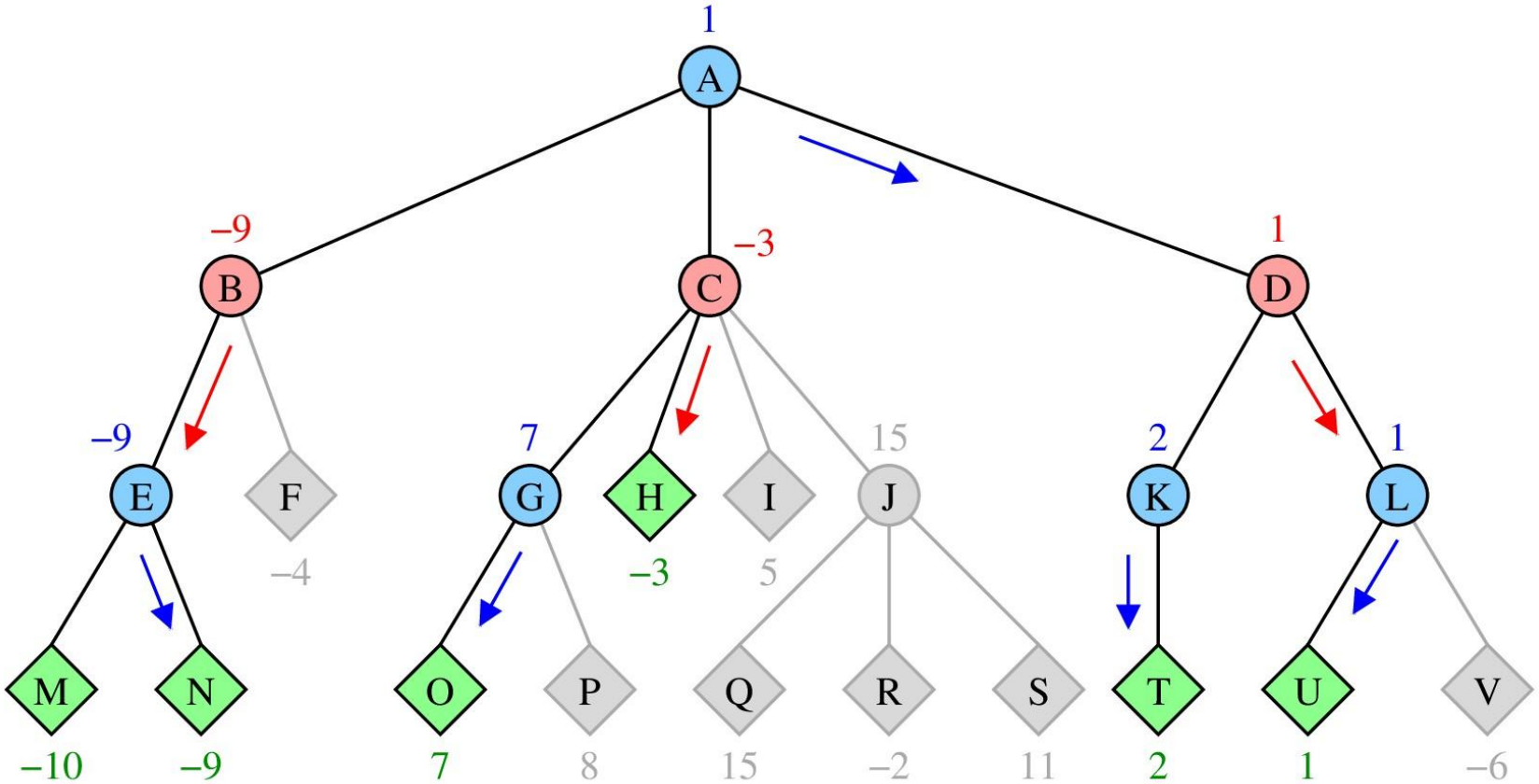
For now, assume that the only goal of each player is to win.

If a winning strategy is already found at any node, there is no need to continue exploring further.

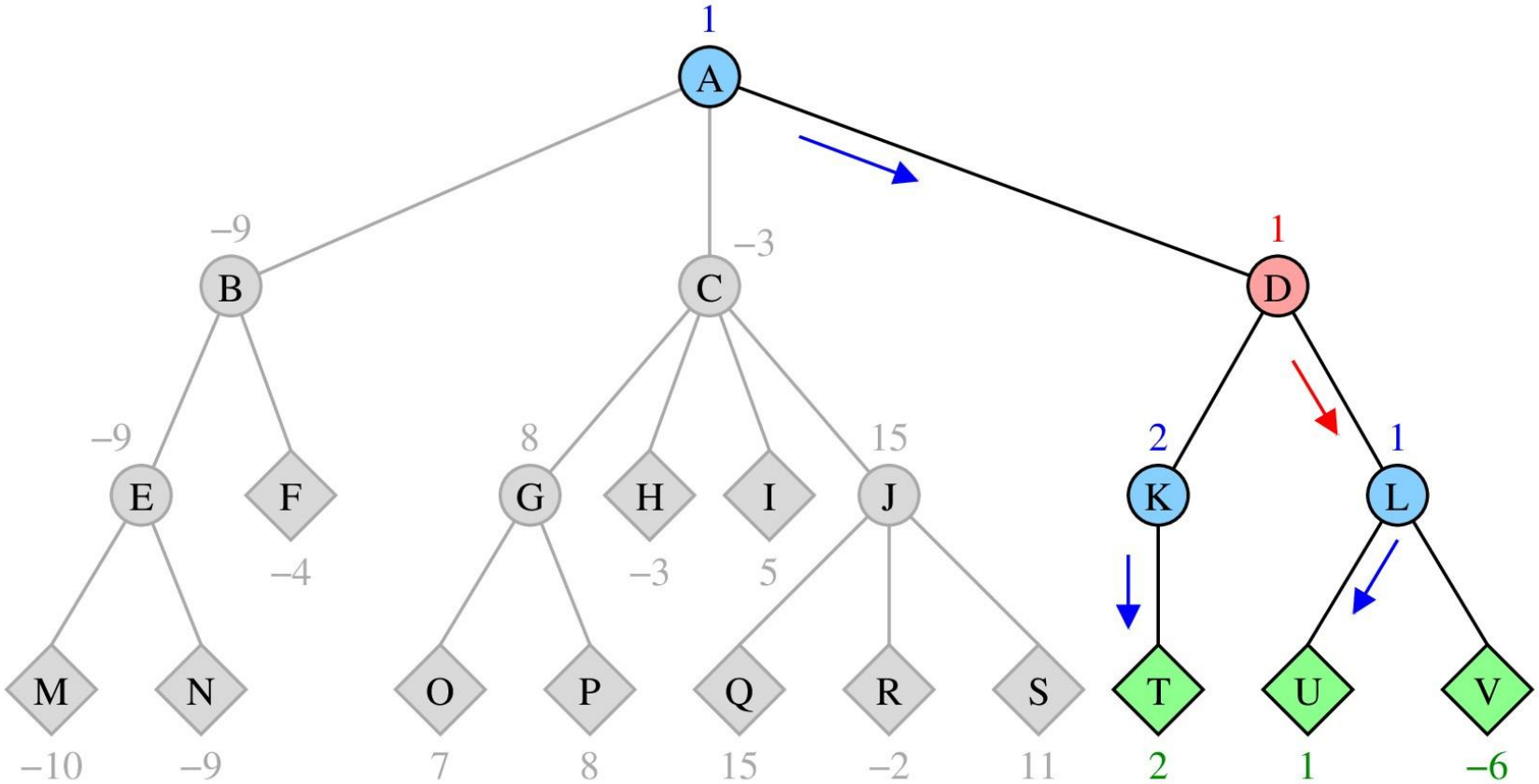
Suppose that we are expanding a node *n*. If a child node gives a winning strategy, follow that. There is no need to find out better winning strategies.

This is true for both MAX and MIN.

Pruning for winning (child nodes are explored left to right)



Node ordering is important (child nodes are explored right to left)



Pruning for finding the best move

The last two examples illustrate the situation when each player is happy as soon as a single winning strategy at any point is found.

This strategy will not work if the best move is to be determined. Even in this new situation, some pruning can be done.

A strategy called **α-β pruning** is used. The algorithm is run by MAX. Two values α and β are passed to the recursive calls. These values are not directly associated with the minimax values to be returned.

Suppose that we are going to expand a node *n*. The node receives α and β from its parent.

α is a lower bound on what MAX can get, and β is an upper bound on what MIN can get.

If *n* is a MAX node, the algorithm tries to increase α.

If *n* is a MIN node, the algorithm tries to decrease β.

In either case, the condition $\alpha \geqslant \beta$ implies that there is no point exploring further from *n*.

Case 1: *n* is a MAX node

Let $n_1, n_2, \ldots, n_k$ be all the children of n as prescribed by the rules of the game.

We try to locate the child n_i with the largest Minimax value.

From the viewpoint of MAX, we have:

- α is a lower bound on what MAX can get.
- β is an upper bound on what MIN can get.

With this information in mind, MAX will keep on making recursive calls on its children to get their Minimax values cval, and updating α = max(α, cval).

If the *i*-th child results in $\alpha \geqslant \beta$, then Minimax(n) $\geqslant \beta$ (because n takes the maximum of the child values).

There is no point exploring the remaining children $n_{i+1}, n_{i+2}, \ldots, n_k$. **[Pruning]**

Case 2: *n* is a MIN node

Let $n_1, n_2, \ldots, n_k$ be all the children of n as prescribed by the rules of the game.

We try to locate the child n_i with the smallest Minimax value.

From the viewpoint of MAX, we have:

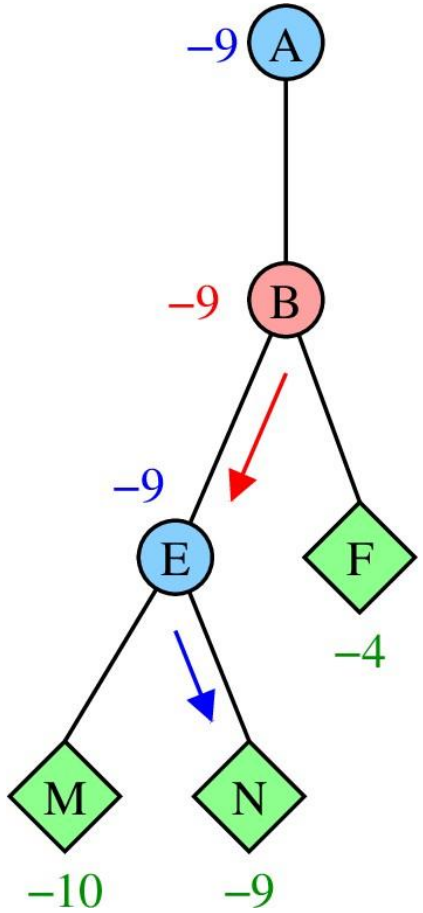
- α is a lower bound on what MAX can get.
- β is an upper bound on what MIN can get.

With this information in mind, MAX will keep on making recursive calls on its children to get their Minimax values cval, and updating β = min(β, cval).

If the i-th child results in $\beta \leqslant \alpha$, then Minimax(n) $\leqslant \alpha$ (because n takes the minimum of the child values).

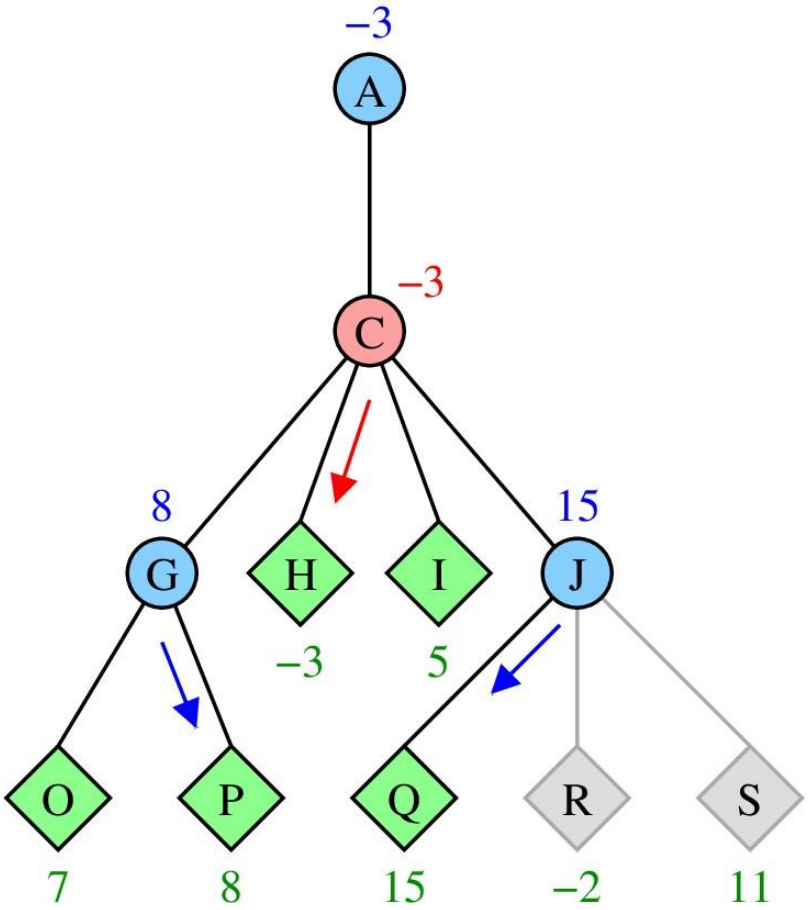
There is no point exploring the remaining children $n_{i+1}, n_{i+2}, \ldots, n_k$. **[Pruning]**

Example: Begin with the first subtree of A



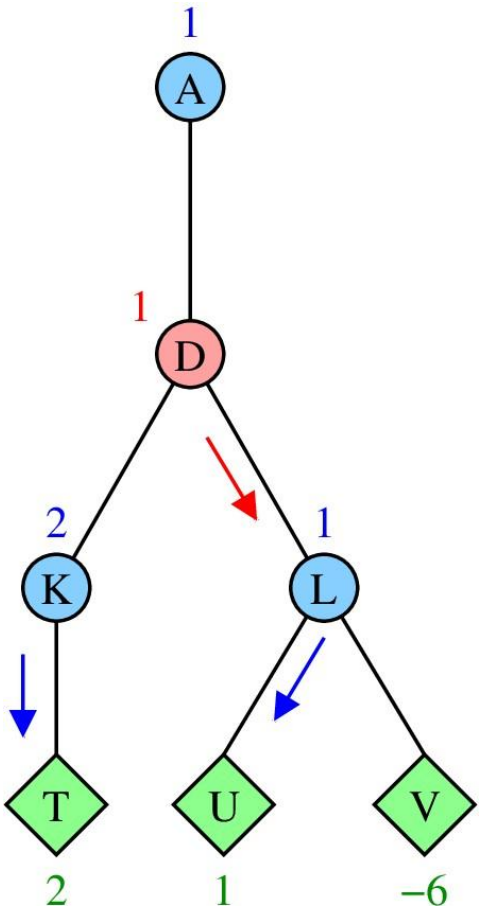
Event	α	β	Return
Init by A	–∞	+∞	
A → B	–∞	+∞	
B → E	–∞	+∞	
E → M	–10	+∞	–10
E → N	–9	+∞	–9
E → B	–∞	–9	–9
B → F	–∞	–9	–4
B → A	–9	+∞	–9

Example: Continue with the second subtree of A



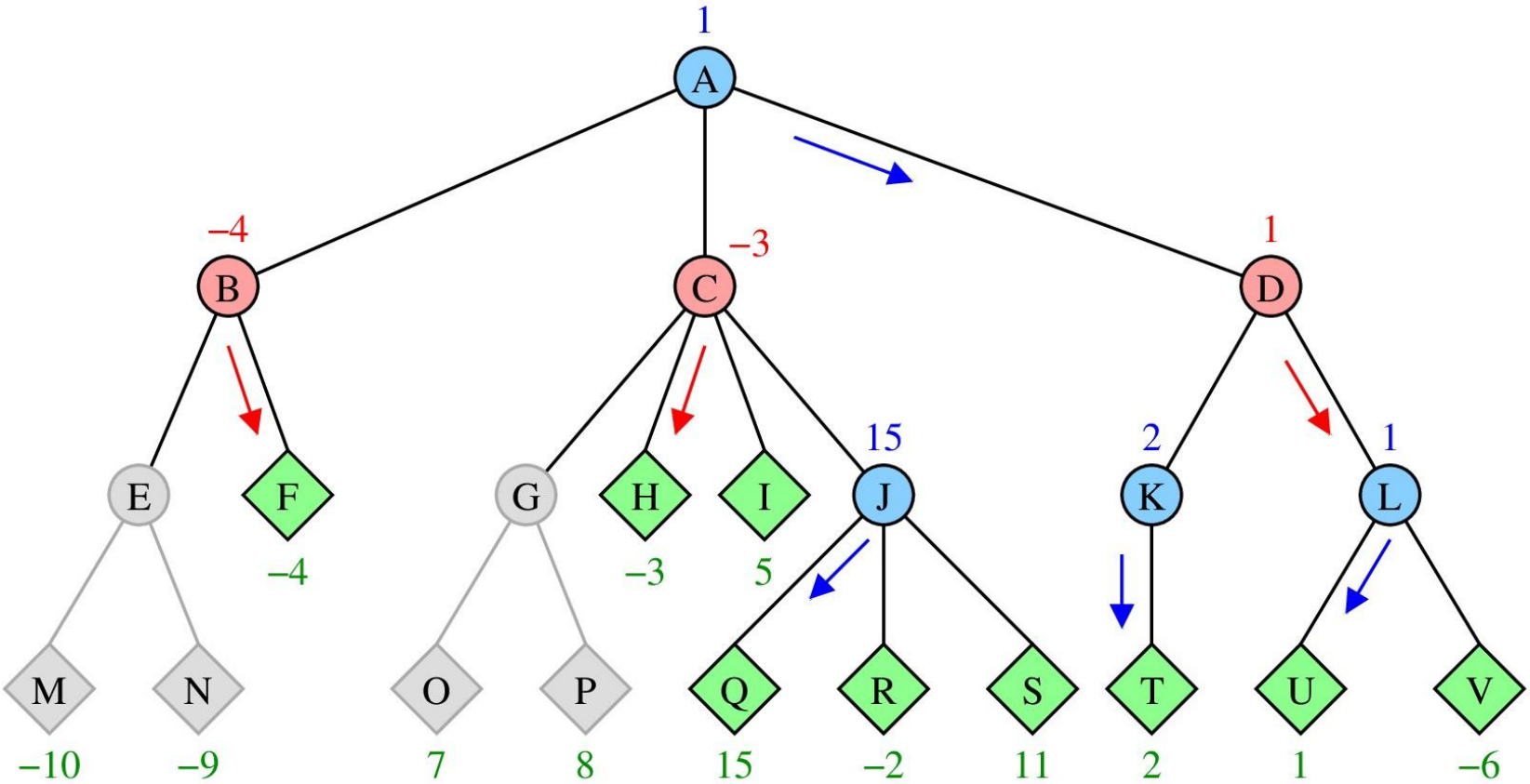
Event	α	β	Return
A has	–9	+∞	
A → C	–9	+∞	
C → G	–9	+∞	
G → O	7	+∞	7
G → P	8	+∞	8
G → C	–9	8	8
C → H	–9	–3	–3
C → I	–9	–3	5
C → J	–9	–3	
J → Q	**15**	**–3**	15
J → C	–9	–3	15
C → A	–3	+∞	–3

Example: Finish with the third subtree of A



Event	α	β	Return
A has	–3	+∞	
A → D	–3	+∞	
D → K	–3	+∞	
K → T	2	+∞	2
K → D	–3	2	2
D → L	–3	2	
L → U	1	2	1
L → V	1	2	–6
L → D	–3	1	1
D → A	1	+∞	1

Node ordering matters (Child nodes are explored from right to left)



Minimax search with α-β pruning: The algorithm

AlphaBeta (n, alpha, beta)

If (*n* is a terminal node), then return (Utility(*n*), NoMove).

Expand n. Let n_1, n_2, …, n_k be all the children of *n*.

If (*n* is a MAX node)

Initialize best = –∞ and bestchild = none.

For *i* = 1, 2, …, *k*, repeat:

Get (cval, cmove) = AlphaBeta(n_i, alpha, beta).

If (cval > best), update best = cval and bestchild = n_i.

Set alpha = max(alpha, best). If alpha ⩾ beta, return (best, bestchild).

else (that is, if *n* is a MIN node)

Initialize best = +∞ and bestchild = none.

For *i* = 1, 2, …, *k*, repeat:

Get (cval, cmove) = AlphaBeta(n_i, alpha, beta).

If (cval < best), update best = cval and bestchild = n_i.

Set beta = min(beta, best). If alpha ⩾ beta, return (best, bestchild).

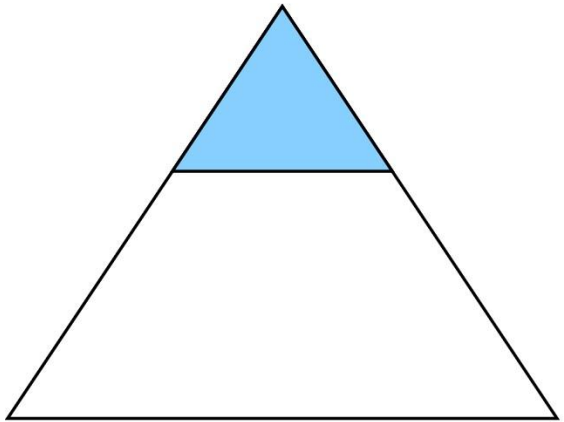
Return (best, bestchild).

This is not the end of the story

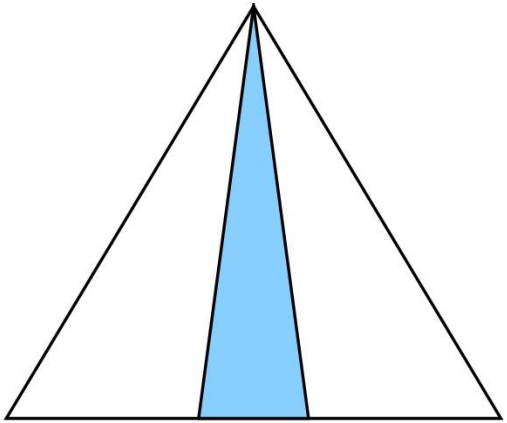
Minimax calculations (with or without pruning) are done in the depth-first fashion, so memory is not a problem. The running time becomes prohibitive if the search needs to explore to a very large depth.

Two types of search to cut down the effort [Shannon, 1950]

Depending on the game, the preferred search type is used.



Type A Strategy



Type B Strategy

Heuristic evaluation

Type A strategy

Terminal nodes are not reached.

A heuristic function is used to estimate a node's Minimax value.

At the maximum allowed depth, this heuristic estimate is used as the utility value of that node.

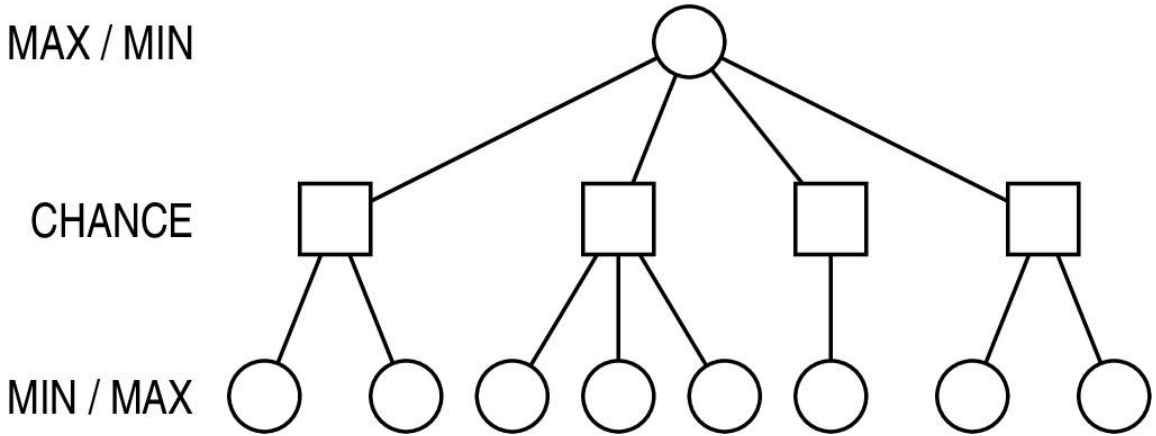
Type B strategy

A heuristic evaluation function is needed to identify the best nodes to explore.

Stochastic games

Each move is preceded by a random event (like throwing a dice, as in Ludo).

Depending upon the outcome of the random event, one of a set of allowed moves is made.



Expectiminimax algorithm

If n is a terminal node,

E-Minimax(n) = Utility(n).

If n is a chance node with k child nodes $n_1, n_2, \ldots, n_k$ appearing with probabilities $p_1, p_2, \ldots, p_k$, then

E-Minimax(n) = $\Sigma_i \, p_i \times$ E-Minimax(n_i).

If n is a MAX node with k child nodes $n_1, n_2, \ldots, n_k$, then

E-Minimax(n) = $\max_i$ (E-Minimax(n_i)).

If n is a MIN node with k child nodes $n_1, n_2, \ldots, n_k$, then

E-Minimax(n) = $\min_i$ (E-Minimax(n_i)).

Expectiminimax example

Each player tosses a fair coin before choosing his move.