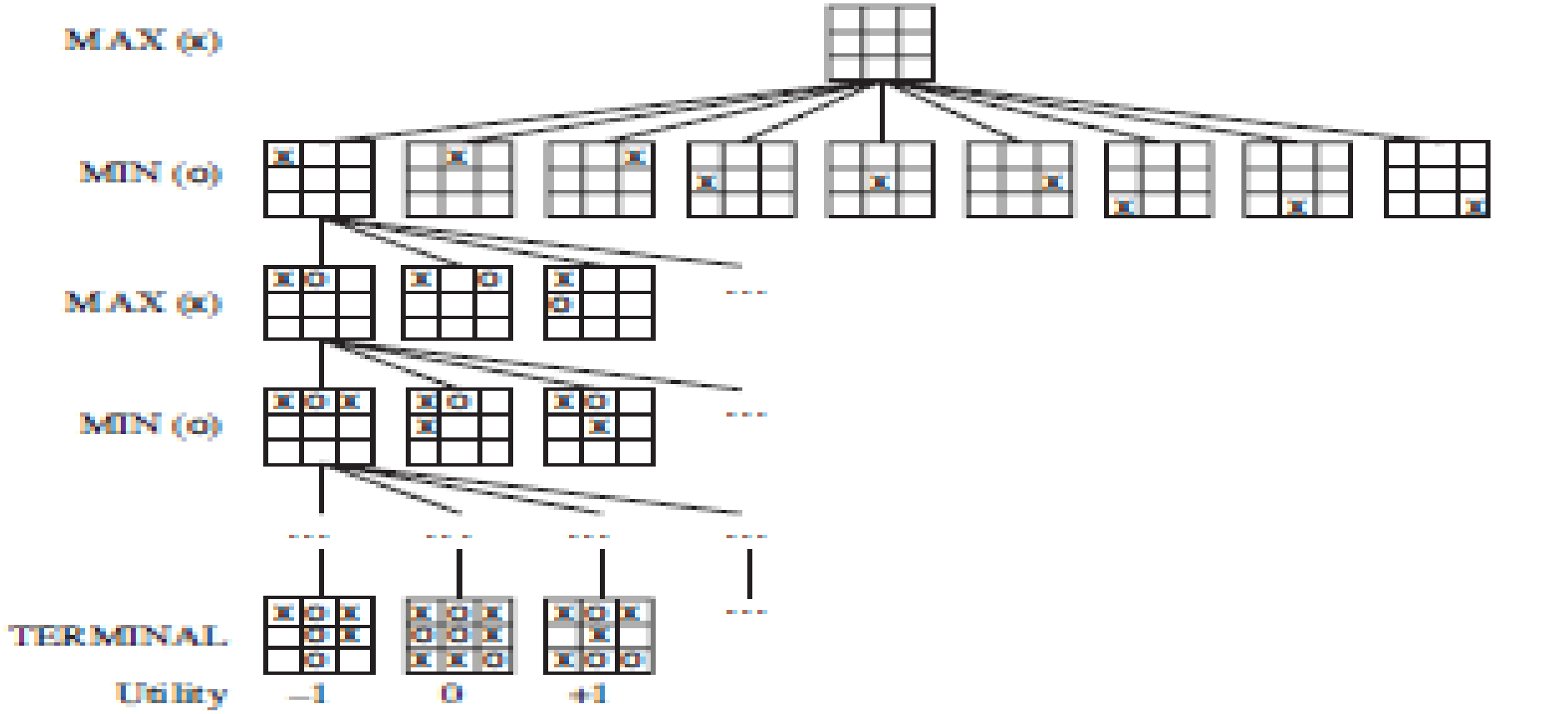
A (partial) game tree for the game of tic-tac-toe.



The top node is the initial state, and MAX moves first, placing an X in an empty square. part of the tree, giving alternating moves by MIN (O) and MAX (X), until, eventually, terminal states are reached, which can be assigned utilities according to the rules of the game.

State space for a variant of nim.

Each state partitions the seven matches into one or more piles.

To play this game, a number of tokens are placed on a table between the two opponents;

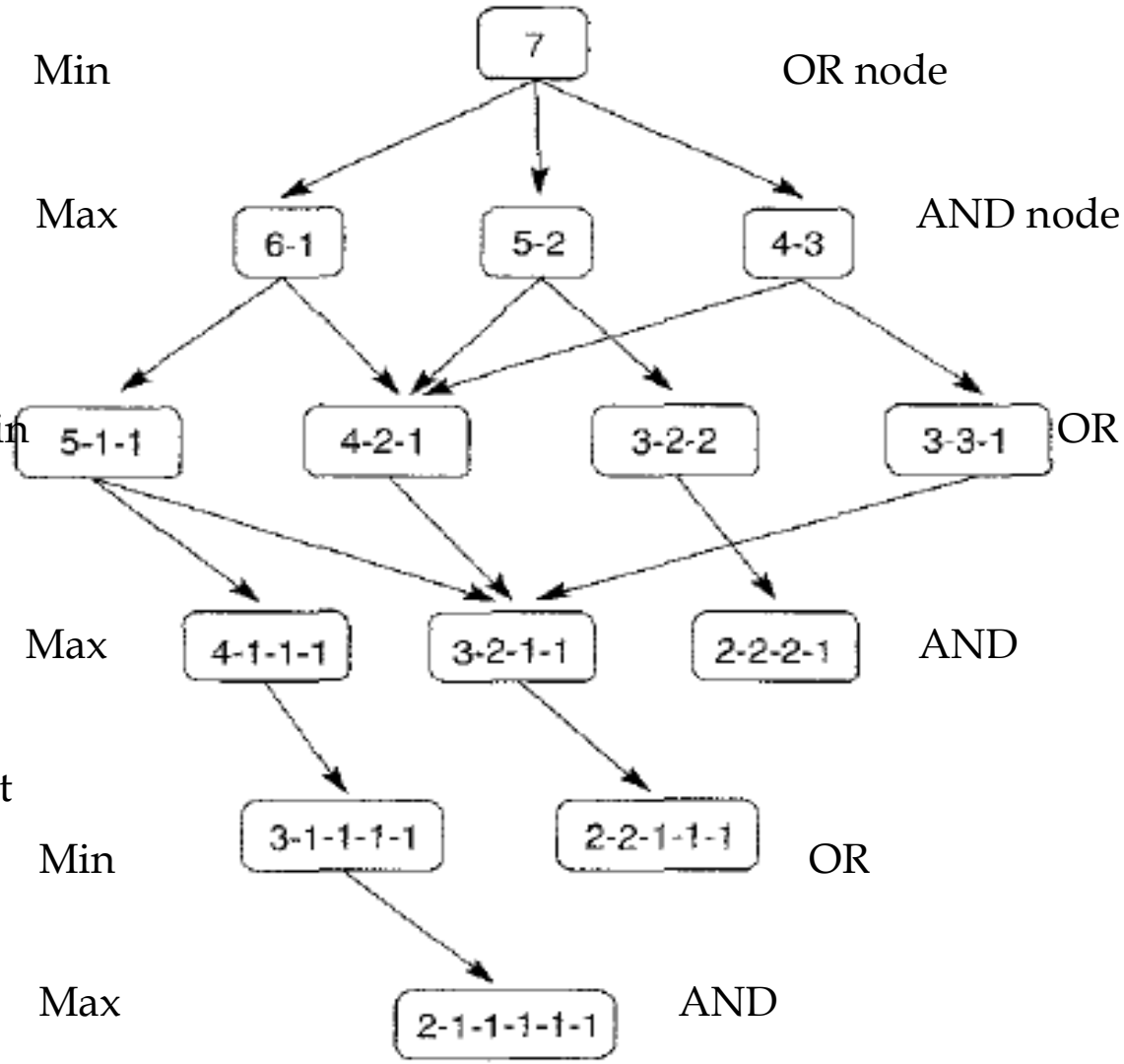
at each move, the player must divide a pile of tokens into two nonempty piles of different sizes.

Thus, 6 tokens may be divided into piles of 5 and 1

Or 4 and 2, but not 3 and 3.

The first player who can no longer make a move loses the game.

your opponent uses the same knowledge of the state space as you use and applies that knowledge in a consistent effort to win the game.



Game Definition as a search Problem

- S0: The initial state, which specifies how the game is set up at the start.
- PLAYER(s): Defines which player has the move in a state.
- ACTIONS(s): Returns the set of legal moves in a state.
- RESULT(s, a): The transition model, which defines the result of a move.
- TERMINAL-TEST(s): A terminal test, which is true when the game is over and false otherwise. States where the game has ended are called terminal states.
- UTILITY(s, p): A utility function (also called an objective function or payoff function),

  defines the final numeric value for a game that ends in terminal state s for a player p. In chess, the outcome is a win, loss, or draw, with values +1, 0, or ½ + ½ Some games have a wider variety of possible outcomes.

  payoffs in backgammon range from 0 to +192.

MiniMax

The opponents in a game are referred to as MIN and MAX.

MAX represents the player trying to win, or to MAXimize his/her advantage.

MIN is the opponent who attempts to MINimize MAX's score.

MIN uses the same information and

Always attempts to move to a state that is worst for MAX.

label each level in the search space according to whose Move it is at that point in the game, MIN or MAX.

Each leaf node is given a value of 1 or 0, depending on whether it is a win for MAX or for MIN.

MiniMax Procedure

Exhaustive minimax for the game of nim. Bold lines indicate forced win for MAX.

Each node is marked with its derived value

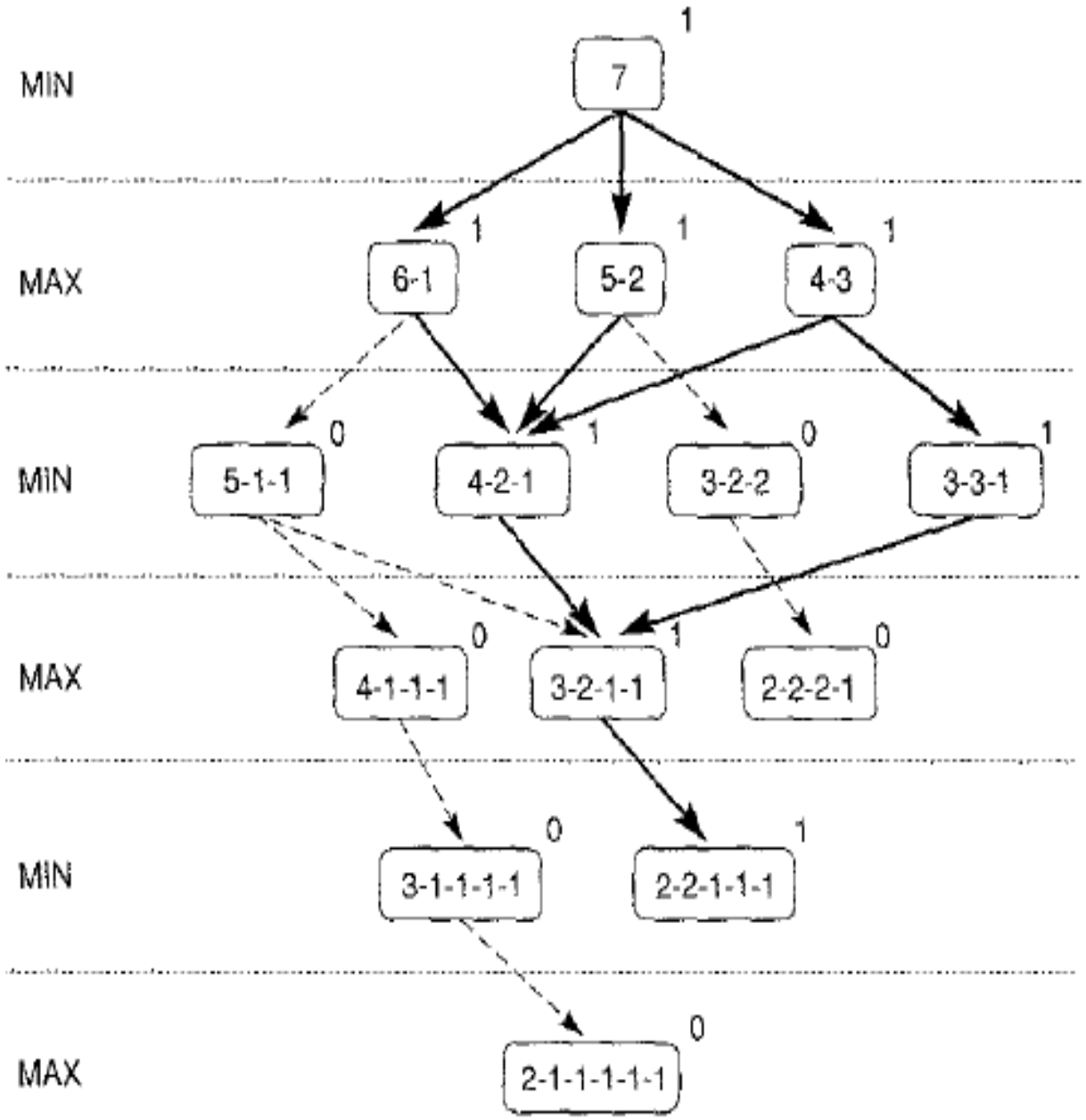
(0 or 1) under minimax.

Each leaf node is given a value of 1 or 0, depending on whether it is a win for MAX or for MIN.

Minimax propagates these values up the graph through successive parent nodes according to the rule:

If the parent state is a MAX node, give it the maximum value among its children.

If the parent is a MIN node, give it the minimum value of its children.



MiniMax

The value that is thus assigned to each state indicates the value of the best state
that this player can hope to achieve
assuming the opponent plays as predicted by the minimax algorithm
These derived values are used to choose among possible moves.

The values of the leaf nodes are propagated up the graph using minimax.
Because all of MIN's possible first moves lead to nodes with a derived value of 1,
the second player, MAX, always can force the game to a win, regardless of MIN's first move.
MIN could win only if MAX played foolishly.

Games Playing

A two-ply game tree.

△ nodes are "MAX nodes," in which it is MAX's turn to move

▽ nodes are "MIN nodes."

terminal nodes show utility values for MAX

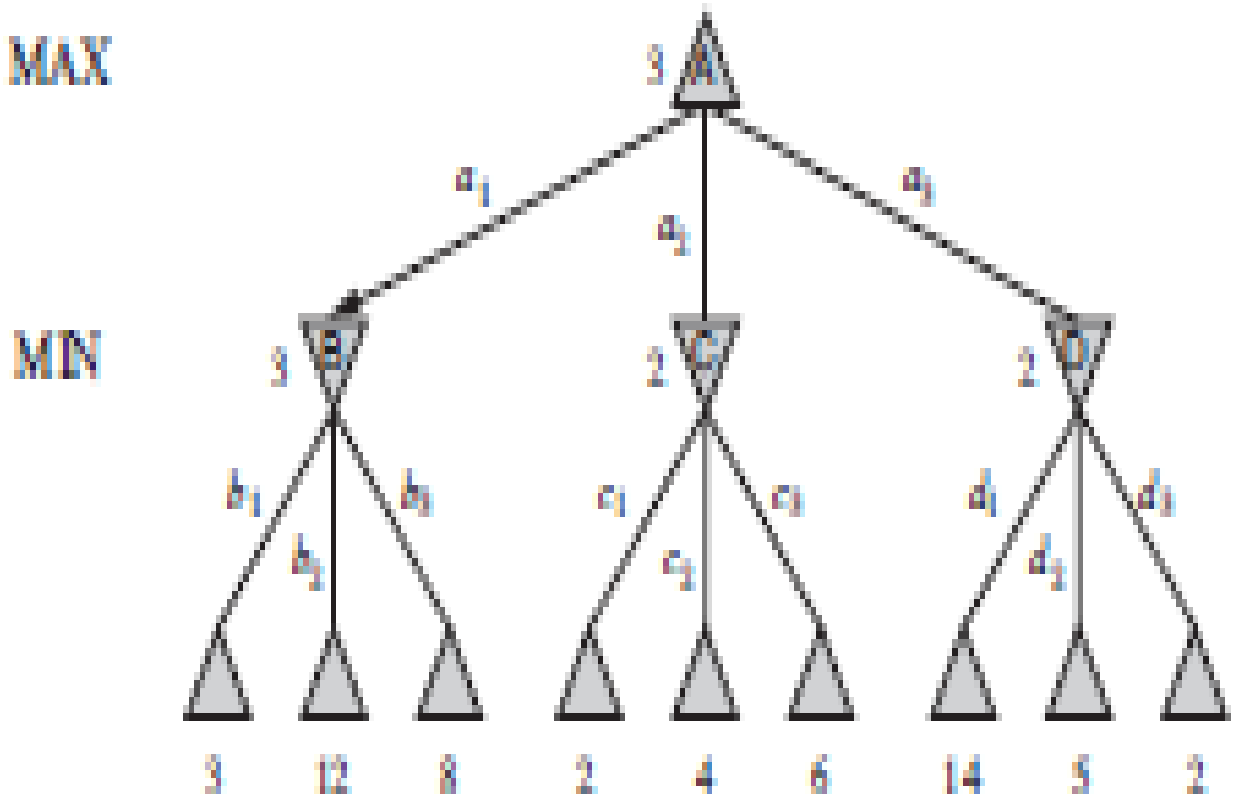
other nodes are labeled with their minimax values.

MAX's best move at the root is a1,

because it leads to the state with the highest minimax value,

MIN's best reply is b1,

because it leads to the state with the lowest minimax value.



MINMAX

MINIMAX(s) =

UTILITY(s)	if TERMINAL-TEST(s)
$\max_{a \in Actions(s)}$ MINIMAX(RESULT(s, a))	if PLAYER(s) = MAX
$\min_{a \in Actions(s)}$ MINIMAX(RESULT(s, a))	if PLAYER(s) = MIN

Algorithm to calculate MINIMAX decisions

function MINIMAX-DECISION(state) **returns** an action

return argmax $_{a \in \text{ACTIONS}(s)}$ MIN-VALUE(RESULT(state, a))

---

function MAX-VALUE(state) **returns** a utility value

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

v ← −∞

for each a **in** ACTIONS(state) **do**

v ←MAX(v, MIN-VALUE(RESULT(s, a)))

return v

---

function MIN-VALUE(state) **returns** a utility value

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

v← ∞

for each a **in** ACTIONS(state) **do**

v ←MIN(v, MAX-VALUE(RESULT(s, a)))

return v

Algorithm to calculate MINIMAX decisions

function MINIMAX-DECISION(state) returns the action corresponding to the best possible move –
the move that leads to the outcome with the best utility,
under the assumption that the opponent plays to minimize utility.

The functions MAX-VALUE and MIN-VALUE go through the whole game tree,
all the way to the leaves, to determine the backed-up value of a state.

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

Minimax - hypothetical state space. Fixed Ply Depth

Leaf states show heuristic values;

internal states show backed-up values

n-ply look-ahead Strategy:

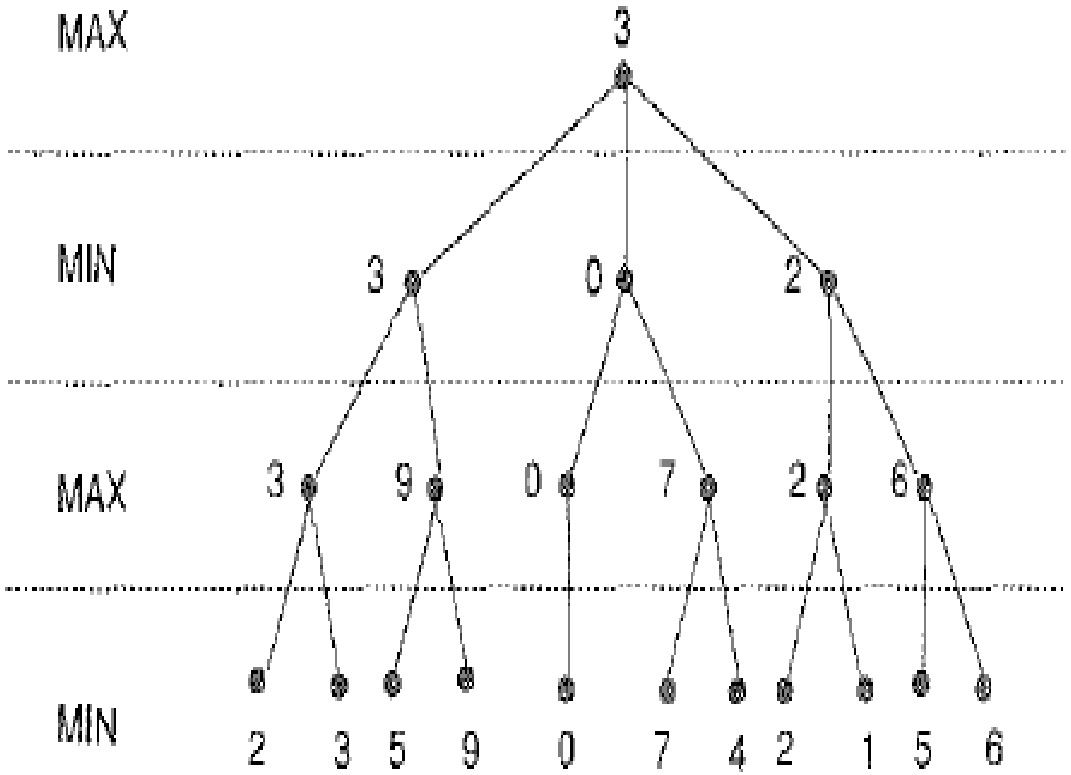
state space is searched to a predefined number of levels, as determined by available resources of time and memory

leaves of this subgraph are not final states of the game,

each leaf node is given a value according to some heuristic evaluation function.

value that is propagated back to the root node is not an indication of whether or not a win can be achieved

It is a heuristic value of the best state that can be reached in n moves from the root.



Game heuristics

Game heuristics directly measure the advantage of one player over another.

In checkers or chess,

piece advantage is important, so a simple heuristic might take the difference in the number of pieces belonging to MAX and MIN and try to maximize the difference between these piece measures.

A more sophisticated strategy might assign different values to the pieces, depending on their value (e.g., queen vs. pawn or king vs. ordinary checker) or location on the board.

Game graphs are searched by level, or ply.

MAX and MIN alternately select moves.

Each move by a player defines a new ply of the graph

Game playing programs typically look ahead a fixed ply depth,

The states on that ply are measured heuristically and

The values are propagated back up the graph using minimax.

The search algorithm then uses

these derived values to select among possible next moves.

MiniMax

After assigning an evaluation to each state on the selected ply, the program propagates a value up to each parent state.

If the parent is on a MIN level, the minimum value of the children is backed up.

If the parent is a MAX node, minimax assigns it the maximum value of its children.

Maximizing for MAX parents and minimizing for MIN, the values go back up the graph to the children of the current state.

These values are then used by the current state to select among its children

horizon effect.

evaluations to any (previously decided) fixed ply depth may be seriously misleading.

When a heuristic is applied with limited look-ahead, it is possible the depth of the look-ahead

may not detect that a heuristically promising path leads to a bad situation later in the game.

Selective deepening of search

searching several plies deeper from states that look exceptionally good.

evaluations that take place very deep in the space can be biased by their very depth

Heuristics for Tic Tac Toe

complex heuristic:

measure the conflict in the game.

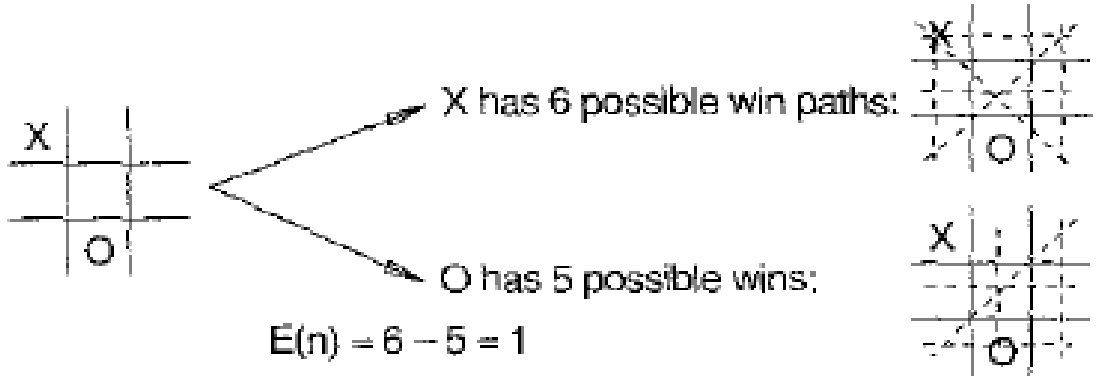
The heuristic takes a state that is to be measured, counts all winning lines open to MAX, and then subtracts the total number of winning lines open to MIN.

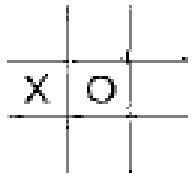
The search attempts to maximize this difference.

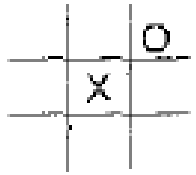
If a state is a forced win for MAX, it is evaluated as $+\infty$; a forced win for MIN, as $-\infty$

Tic tac toe

Heuristic measuring conflict applied to states of tic-tac-toe.







Heuristic is E(n) = M(n) − O(n)

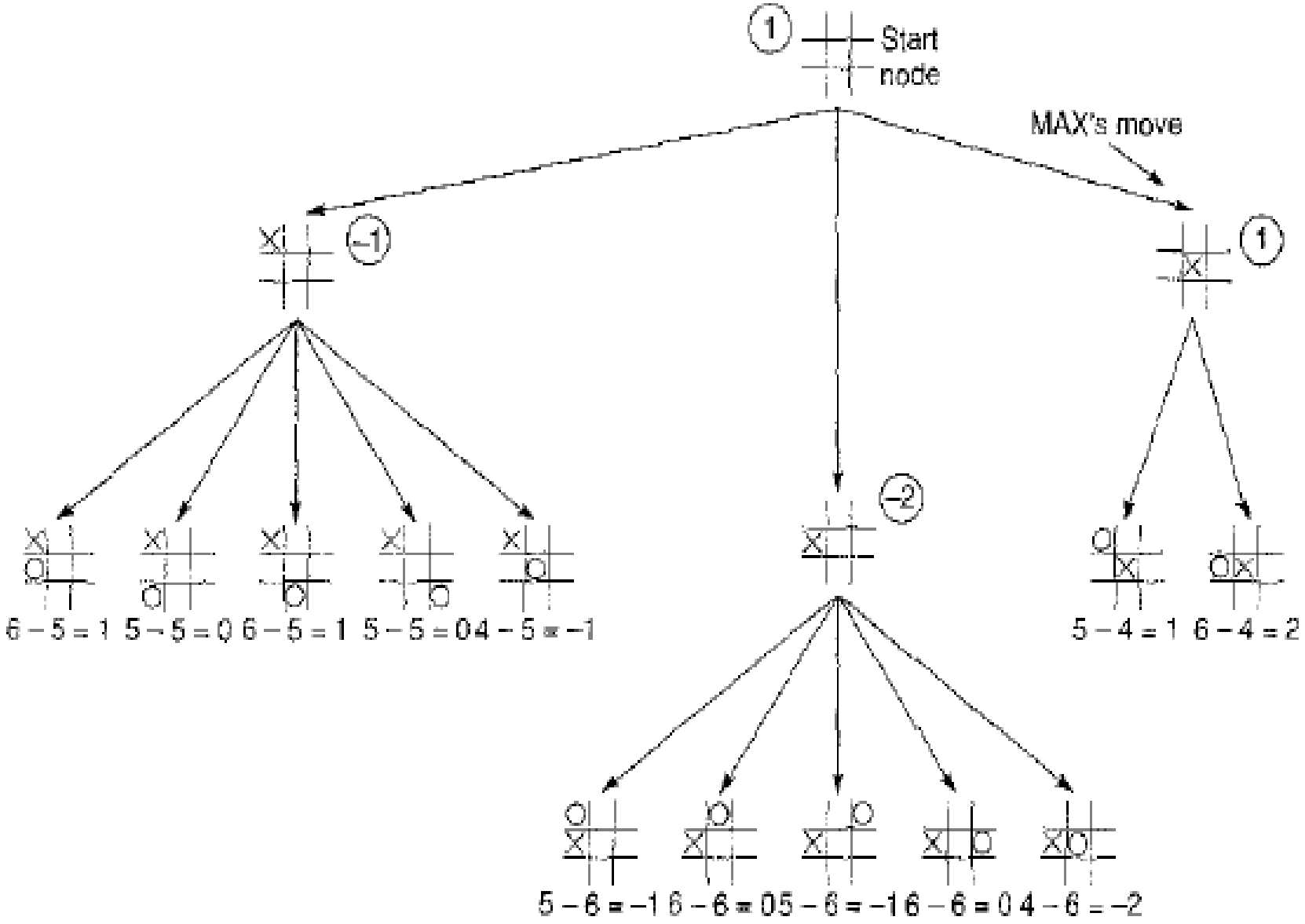
where M(n) is the total of My possible winning lines

O(n) is total of Opponent's possible winning lines

E(n) is the total Evaluation for state n

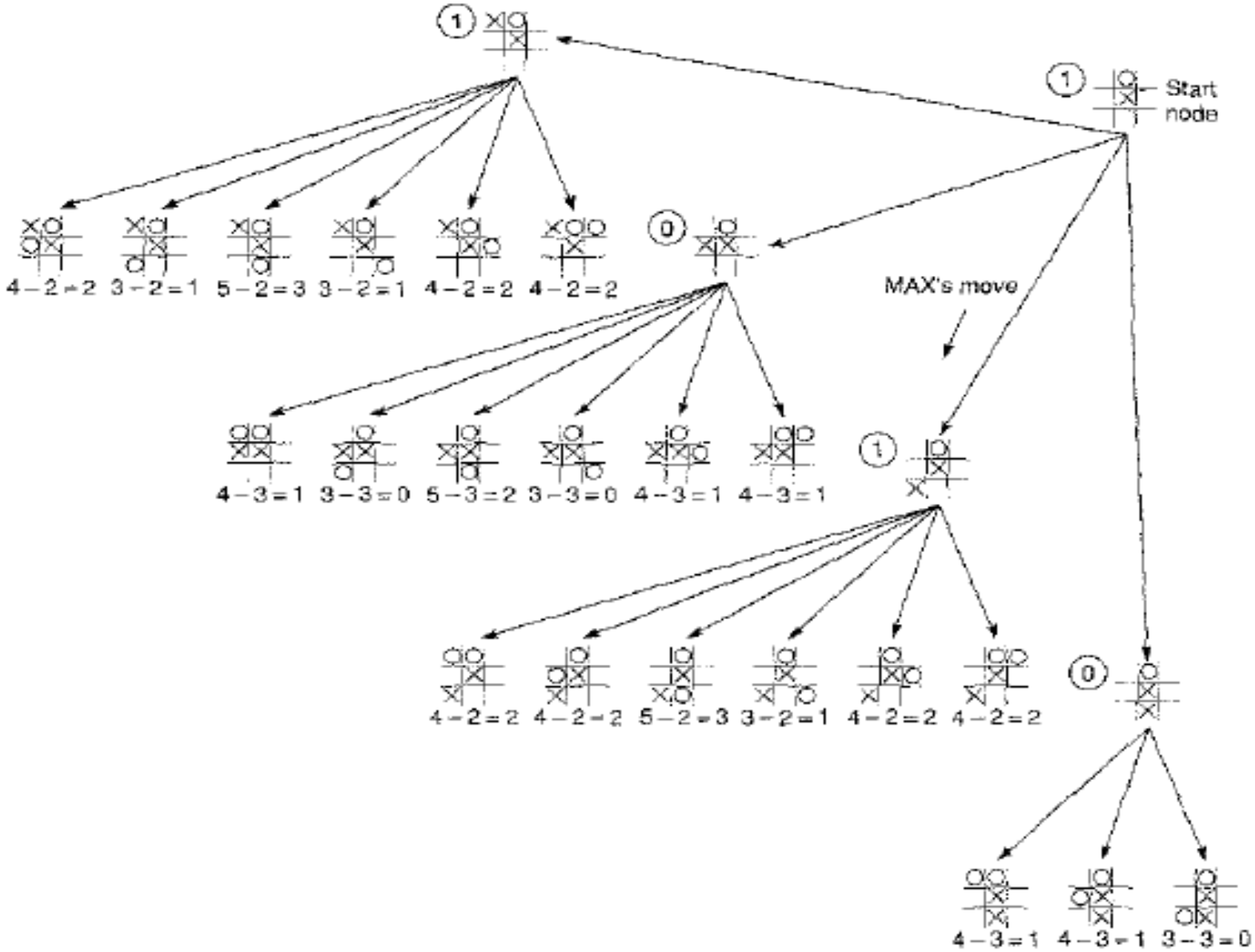
Tic tac toe

Two-ply minimax applied to the opening move of tic-tac-toe



Tic tac toe

Two-ply minimax and one of two possible MAX second moves



Complexity

difficult aspect of combinatorial problems is that
the "explosion" often takes place
without program designers realizing that it is happening

"If only I had a larger (or faster _ highly parallel) computer my problem would be solved!“

Estimated number of states produced by a full search of the space of possible chess moves is about 10^{120}

it is comparable to the number of molecules in the universe or the number of nanoseconds since the "big bang.“

branching factor: average number of branches (children) that are expanded from any state in the space.
The number of states at depth n of the search is equal to the branching factor raised to the nth power.

Alpha Beta Pruning

A has $\beta = 3$ (A will be no larger than 3)

B is β pruned, since 5 > 3

C has $\alpha = 3$ (C will be no smaller than 3)

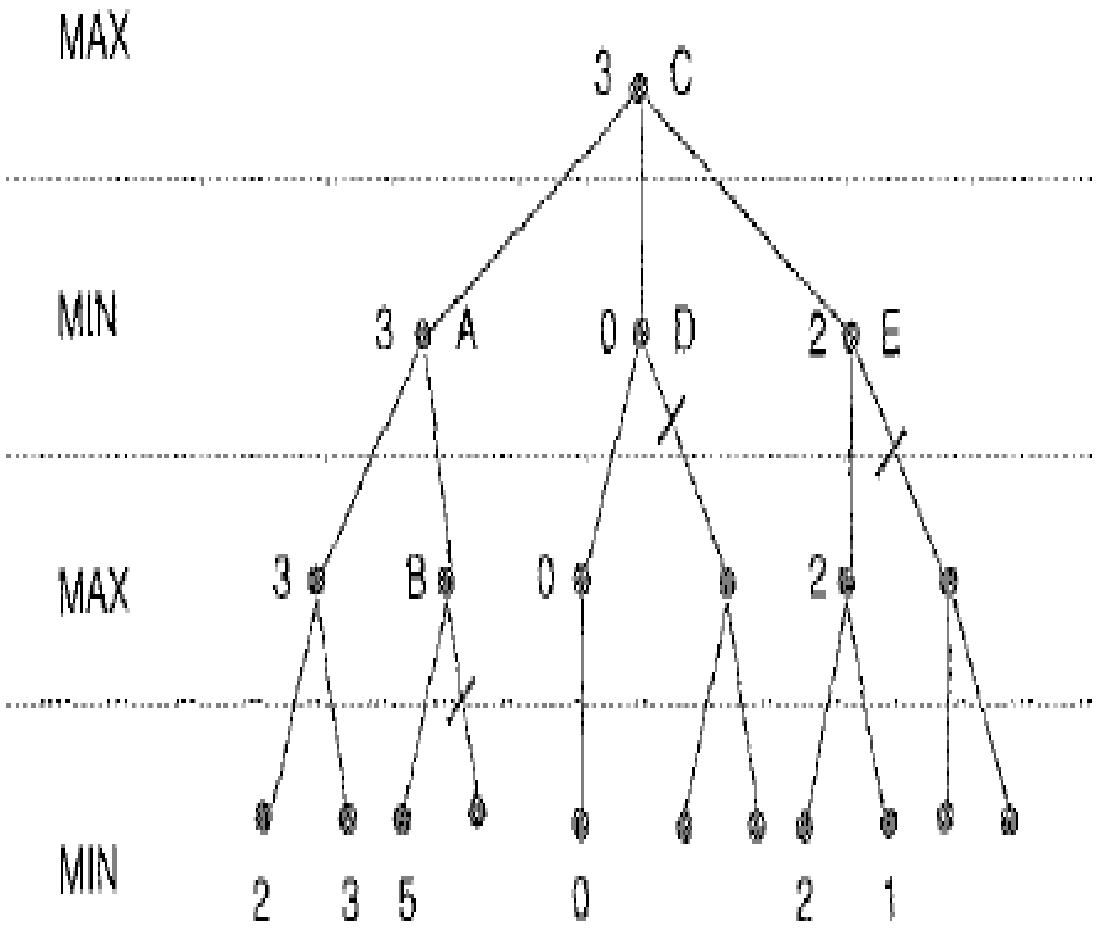
D is α pruned, since 0 < 3

E is α pruned, since 2 < 3

C is 3

Alpha-beta pruning applied to hypothetical state space.

States without numbers are not evaluated.



Alpha Beta Pruning

minimax requires a two-pass analysis of the search space

the first to **descend to the ply depth** and there apply the heuristic and

the second to **propagate values back** up the tree.

Minimax **pursues all branches in the space**, including many that could be ignored

or pruned by a more intelligent algorithm.

alpha-beta pruning

class of search techniques improve search efficiency in two-person games

rather than searching the entire space to the ply depth, alpha-beta search proceeds in a depth-first fashion.

Two values, called alpha and beta, are created during the search.

The alpha value, associated with MAX nodes, can never decrease, and

the beta value, associated with MIN nodes, can never increase.

Suppose a MAX node's alpha value is 6. Then MAX need not consider any backed-up value less than or equal to 6 that is associated with any MIN node below it.

Alpha is the worst that MAX can "score" given that MIN will also do its "best.“

if MIN has beta value 6, it does not need to consider any MAX node below that has a value of 6 or more.

Alpha Beta Pruning

alpha-beta search:

descend to full ply depth in a depth-first fashion and

apply heuristic evaluation to a state and all its siblings. Assume these are MIN nodes.

The maximum of these MIN values is then backed up to the parent (a MAX node).

This value is then offered to the grandparent of these MINs as a potential beta cutoff.

Next,

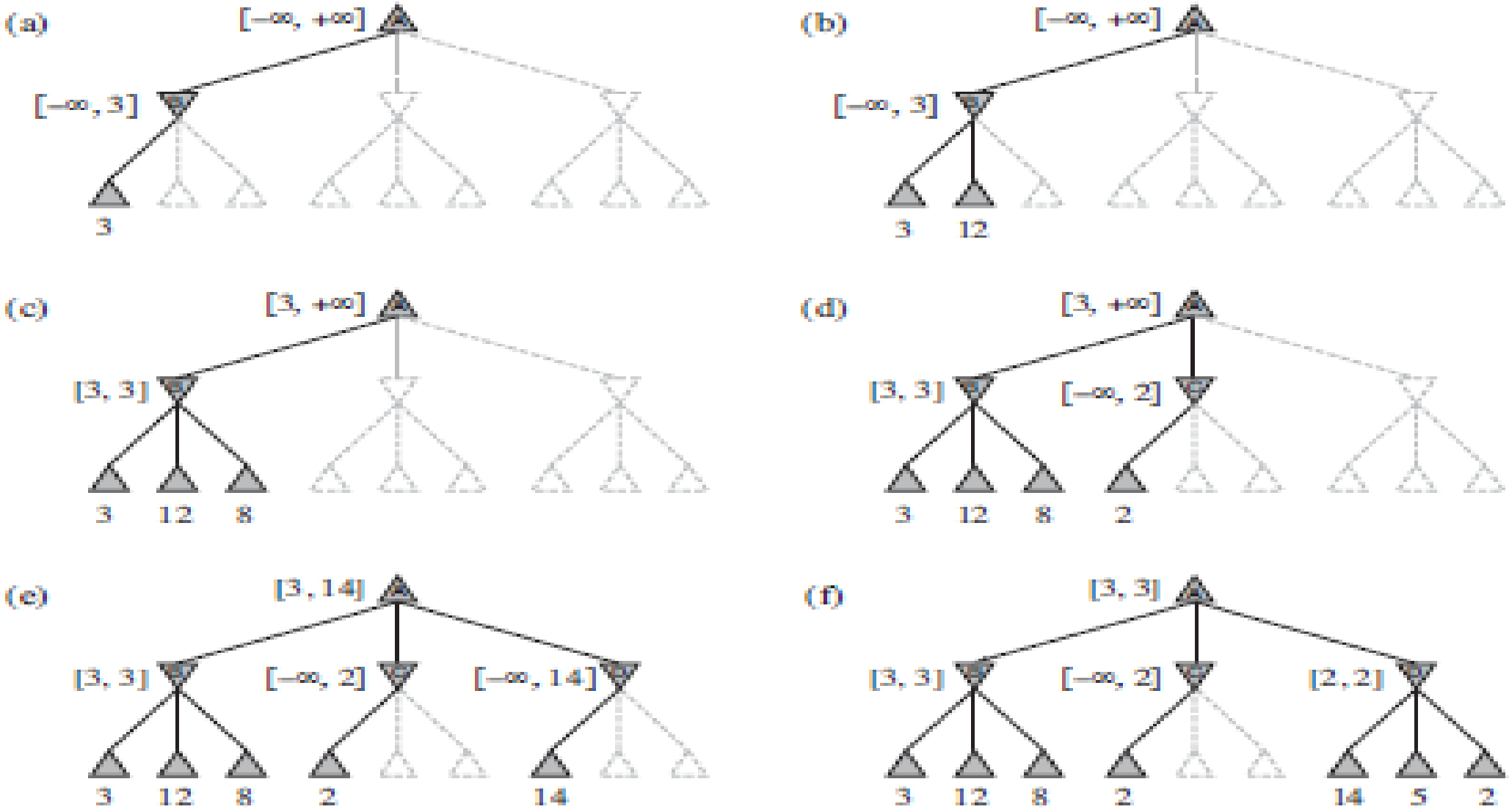
the algorithm descends to other grandchildren and terminates exploration of their parent

if any of their values is equal to or larger than this beta value.

Two rules for terminating search, based on alpha and beta values:

1. Search can be stopped below any MIN node having a beta value less than or equal
   to the alpha value of any of its MAX ancestors.
2. Search can be stopped below any MAX node having an alpha value greater than
   or equal to the beta value of any of its MIN node ancestors.

Alpha Beta Pruning



Alpha Beta Pruning

Stages in the calculation of the optimal decision for the game tree:

At each point, range of possible values for each node. Are shown:

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

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

(c) The third leaf below B has a value of 8; we have seen all B's successor states, so the value of B is exactly 3. Now, we can infer that the value of the root is at least 3, because MAX has a choice worth 3 at the root.

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

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

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

Alpha Beta Pruning

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

= max(3, min(2, x, y), 2)

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

= 3

Alpha Beta Pruning – Move ordering

effectiveness of alpha–beta pruning is highly dependent on **the order in which the states are examined.**

try to examine first the **successors that are likely to be best.**

With **Move ordering**

alpha–beta needs to examine only $O(b^{m/2})$ nodes to pick the best move, instead of $O(b^m)$ for minimax.

Implies -

Effective branching factor becomes $\sqrt{b}$ instead of b—for chess, about 6 instead of 35.

If successors are examined **in random order rather than best-first**, the total number of nodes examined will be roughly $O(b^{3m/4})$ for moderate b

For chess,

a fairly simple ordering function - trying captures first, then threats, then forward moves, and then backward moves) gets you to within about a factor of 2 of the best-case $O(b^{m/2})$ result.

dynamic move-ordering schemes - trying first the moves that were found

to be best in the past – **iterative deepening search** – use the recorded path to inform move ordering

Killer Move heuristic: try first killer moves(best moves)

Alpha Beta Pruning

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

Alpha Beta Pruning – Move ordering

repeated states in the search tree can cause an exponential increase in search cost.

In games, repeated states occur frequently because of **transpositions**—different permutations of the move sequence that end up in the same position.

Ex:

if White has one move, a1, that can be answered by Black with b1 and

an unrelated move a2 on the other side of the board that can be answered by b2, then

the sequences [a1, b1, a2, b2] and [a2, b2, a1, b1] both end up in the same position.

Transposition table - (identical to explored list in GRAPH-SEARCH)

Store the evaluation of the resulting position in a hash table the first time it is encountered so that

we don't have to re-compute it on subsequent occurrences.

The hash table of previously seen positions is called a transposition table

Imperfect real-Time Decisions

minimax algorithm generates the entire game search space,

alpha–beta algorithm allows us to prune large parts of it.

alpha–beta still has to search all the way to terminal states for at least a portion of the search space.

This depth is not practical,

As moves must be made in a reasonable amount of time—a few minutes at most.

Claude Shannon – “Programming a computer for playing chess” proposal -

cut off the search earlier and apply a heuristic evaluation function to states in the search, effectively turning nonterminal nodes into terminal leaves

Imperfect real-Time Decisions

H-MINIMAX(s, d) =

EVAL(s)	if CUTOFF-TEST(s, d)
$\max_{a \in Actions(s)}$ H-MINIMAX(RESULT(s, a), d + 1)	if PLAYER(s) = MAX
$\min_{a \in Actions(s)}$ H-MINIMAX(RESULT(s, a), d + 1)	if PLAYER(s) = MIN.

alter minimax or alpha–beta in two ways:

replace the utility function by a heuristic evaluation function EVAL, which estimates the position's utility, and

replace the terminal test by a cutoff test that decides when to apply EVAL.

Evaluation functions

evaluation function returns an estimate of the expected utility of the game from a given position,

just as the heuristic functions

performance of a game-playing program depends strongly on the quality of its evaluation function.

- evaluation function should order the terminal states in the same way as the true utility function:

states that are wins must evaluate better than draws, which in turn must be better than losses.

- computation must not take too long!

- for nonterminal states, the evaluation function should be strongly correlated with the actual chances of winning.

Evaluation functions

Calculating various features of the state— Ex: Chess

features for the number of white pawns, black pawns, white queens, black queens, and so on.

features, taken together, define various **categories** or equivalence classes of states:

the states in each category have the same values for all the features.

one category may contain all two-pawn vs. one-pawn endgames.

Any given category, generally speaking, will contain some states

that lead to wins, some that lead to draws, and some that lead to losses

Evaluation function cannot know which states are which, but it can return a single value that reflects the

proportion of states with each outcome.

Ex:

72% of the states encountered in the two-pawns vs. one-pawn category lead to a win (utility +1);

20% to a loss (0), and 8% to a draw (1/2).

Then a reasonable evaluation for states in the category is

the expected value: $(0.72 \times +1) + (0.20 \times 0) + (0.08 \times 1/2) = 0.76$.

Evaluation functions

introductory chess books give an approximate material value for each piece:

each pawn is worth 1, a knight or bishop is worth 3, a rook 5, and the queen 9.

features like "good pawn structure" and "king safety" might be worth half a pawn

feature values are added up to obtain the evaluation of the position.

evaluation function - a weighted linear function

$$\text{EVAL}(s) = w_1 f_1(s) + w_2 f_2(s) + \cdots + w_n f_n(s) = \sum_{i=1}^{n} w_i f_i(s)$$

where each w_i is a weight and each fi is a feature of the position.

For chess, f_i could be the numbers of each kind of piece on the board, and w_i could be the values of the pieces (1 for pawn, 3 for bishop, etc.).

strong assumption:

contribution of each feature is independent of the values of the other features.

assigning the value 3 to a bishop ignores the fact

That bishops are more powerful in the endgame, when they have a lot of space to maneuver.

programs for chess and other games also use **nonlinear combinations of features**

Search Cut off

Replace the two lines with TERMINAL-TEST with

if CUTOFF-TEST(state, depth) then return EVAL(state)

depth d is chosen so that a move is selected within the allocated time

Or apply iterative deepening

When time runs out, the program returns the move selected by the deepest completed search.

These simple approaches can lead to errors due to the approximate nature of the evaluation function.

Quiescence search

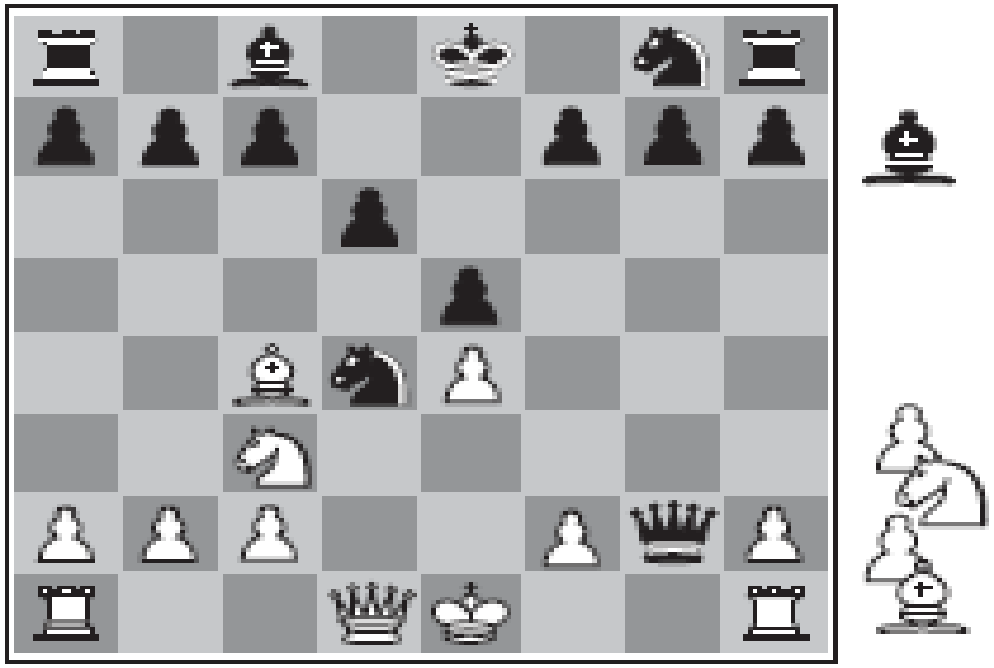
The evaluation function should be

applied only to positions that are quiescent—

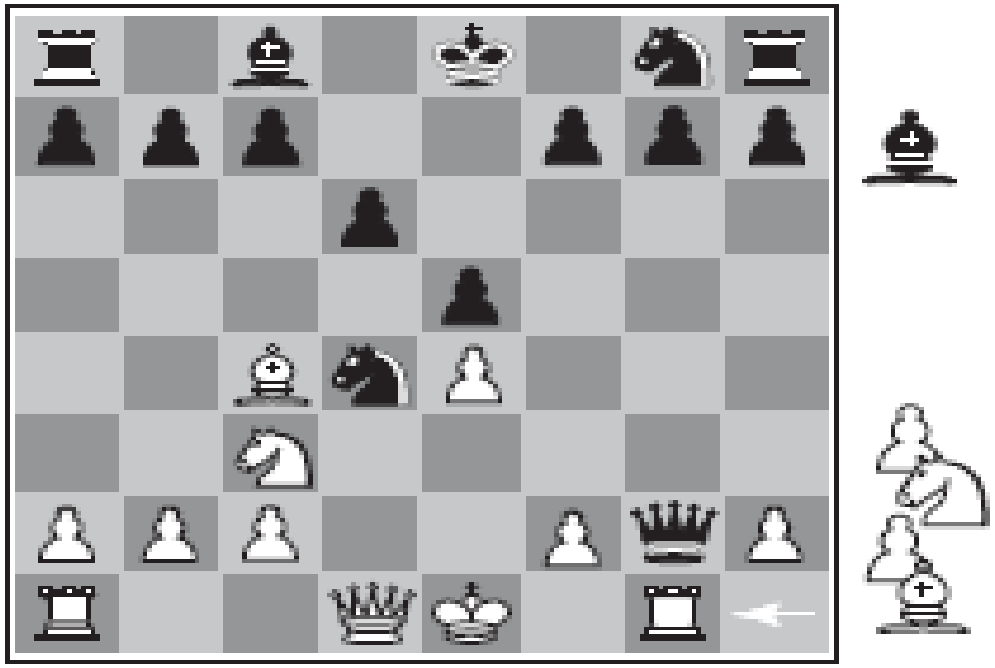
that is, unlikely to exhibit wild swings in value in the near future.

Non quiescent positions can be expanded further until quiescent positions are reached

Chess – Quiescent position



(a) White to move



(b) White to move

Two chess positions that differ only in the position of the rook at lower right.

(a), Black has an advantage of a knight and two pawns, which should be enough to win the game.

(b), White will capture the queen, giving it an advantage that should be strong enough to win.

Evaluation Functions - Chess

Horizon effect

when the program is facing an opponent's move that causes serious damage and is ultimately unavoidable, but can be temporarily avoided by delaying tactics.

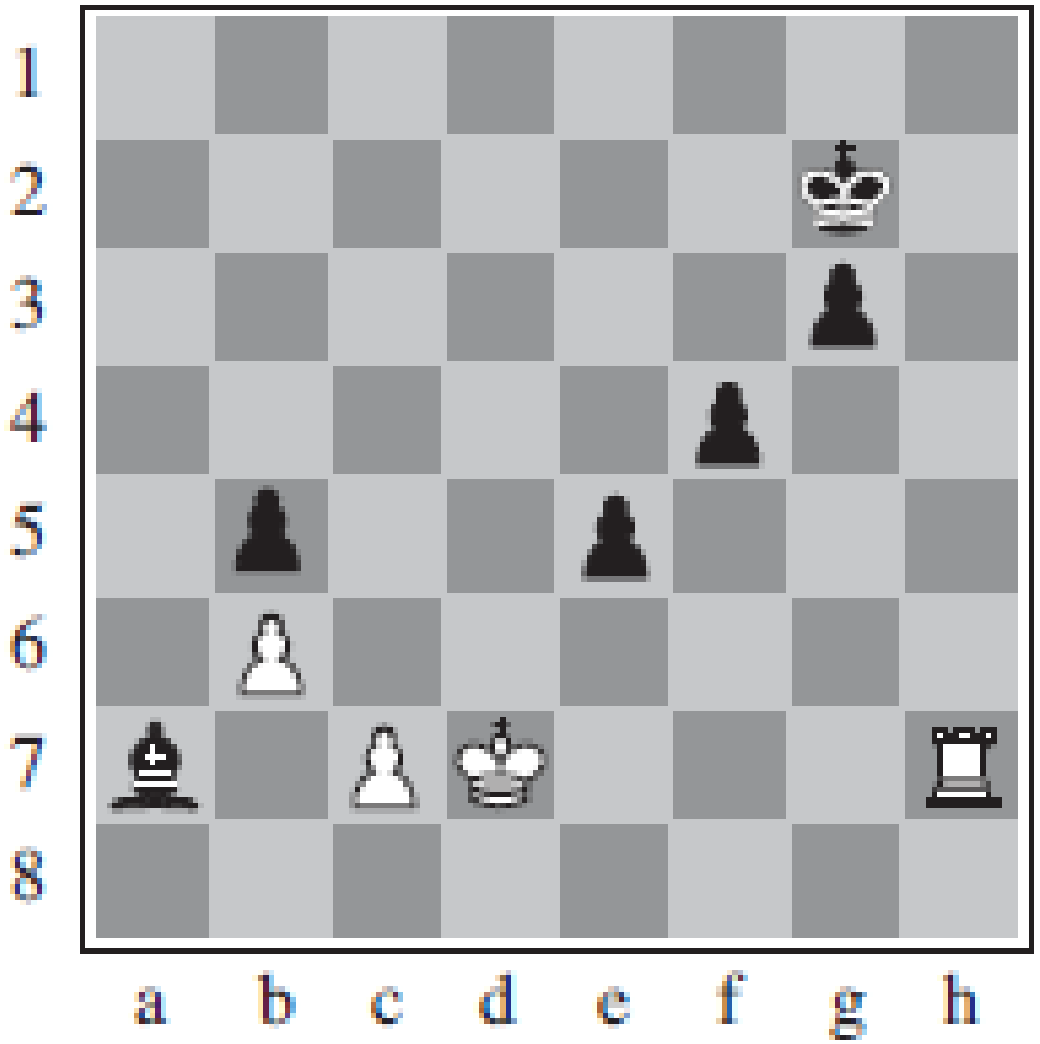
The horizon effect. With Black to move, the black bishop is surely doomed.

But Black can forestall that event by checking the white king with its pawns, forcing

the king to capture the pawns. This pushes the inevitable loss of the bishop over the horizon, and

thus the pawn sacrifices are seen by the search algorithm as good moves rather than bad ones.

Singular extension - to mitigate the horizon effect.



Singular Extension, Forward pruning

Singular extension - a move that is "clearly better" than all other moves in a given position.

Once discovered anywhere in the tree in the course of a search, this singular move is remembered.

When the search reaches the normal depth limit, the algorithm checks to see if the singular extension is a legal move; if it is, the algorithm allows the move to be considered.

This makes the tree deeper, but because there will be few singular extensions, it does not add many total nodes to the tree.

forward pruning

some moves at a given node are pruned immediately without further consideration

beam search

One approach to forward pruning is beam search:

on each ply, consider only a "beam" of the n best moves (according to the evaluation function)

rather than considering all possible moves.

this approach is rather dangerous because there is no guarantee

that the best move will not be pruned away

Search Cut off – PROBCUT (Buro, 1995)

PROBCUT - forward-pruning version of alpha–beta search

probabilistic cut, algorithm uses statistics gained from prior experience to lessen the chance that the best move will be pruned.

Alpha–beta search prunes any node that is **provably outside the current (α, β) window**.

PROBCUT also prunes nodes that are **probably outside the window**.

It computes this probability by doing a shallow search to compute the backed-up value v of a node and then using past experience to estimate how likely it is that a score of v at depth d in the tree would be outside (α, β).

Lookup Tables

overkill for a chess program to start a game by considering a tree of a billion game states,

only to conclude that it will move its pawn to e4.

many game-playing programs use table lookup rather than search for the opening and ending of games.

computers can also gather statistics from a database

of previously played games to see which opening sequences most often lead to a win.

In the early moves there are few choices, and thus

much expert commentary and past games on which to draw.

Usually after ten moves player ends up in a rarely seen position, and

the program must switch from table lookup to search.

Near the end of the game there are again fewer possible positions, and thus

more chance to do lookup.

Alpha Beta Pruning – modified to call Heuristic EVAL Function

```
function ALPHA-BETA-SEARCH(state) returns an action
    v ←MAX-VALUE(state,−∞,+∞)
    return the action in ACTIONS(state) with value v
-----------------------------------------------------------------------------------------
function MAX-VALUE(state,α, β) returns a utility value
    if CUTOFF-TEST(state, depth) then return EVAL(state)
    v ←−∞
    for each a in ACTIONS(state) do
        v ←MAX(v, MIN-VALUE(RESULT(s,a),α, β))
        if v ≥ β then return v
        α←MAX(α, v)
    return v
-----------------------------------------------------------------------------------------
function MIN-VALUE(state,α, β) returns a utility value
    if CUTOFF-TEST(state, depth) then return EVAL(state)
    v ←+∞
    for each a in ACTIONS(state) do
        v ←MIN(v, MAX-VALUE(RESULT(s,a) ,α, β))
        if v ≤ α then return v
        β←MIN(β, v)
    return v
```

STOCHASTIC GAMES

unpredictable external events can put us into unforeseen situations

Stochastic games

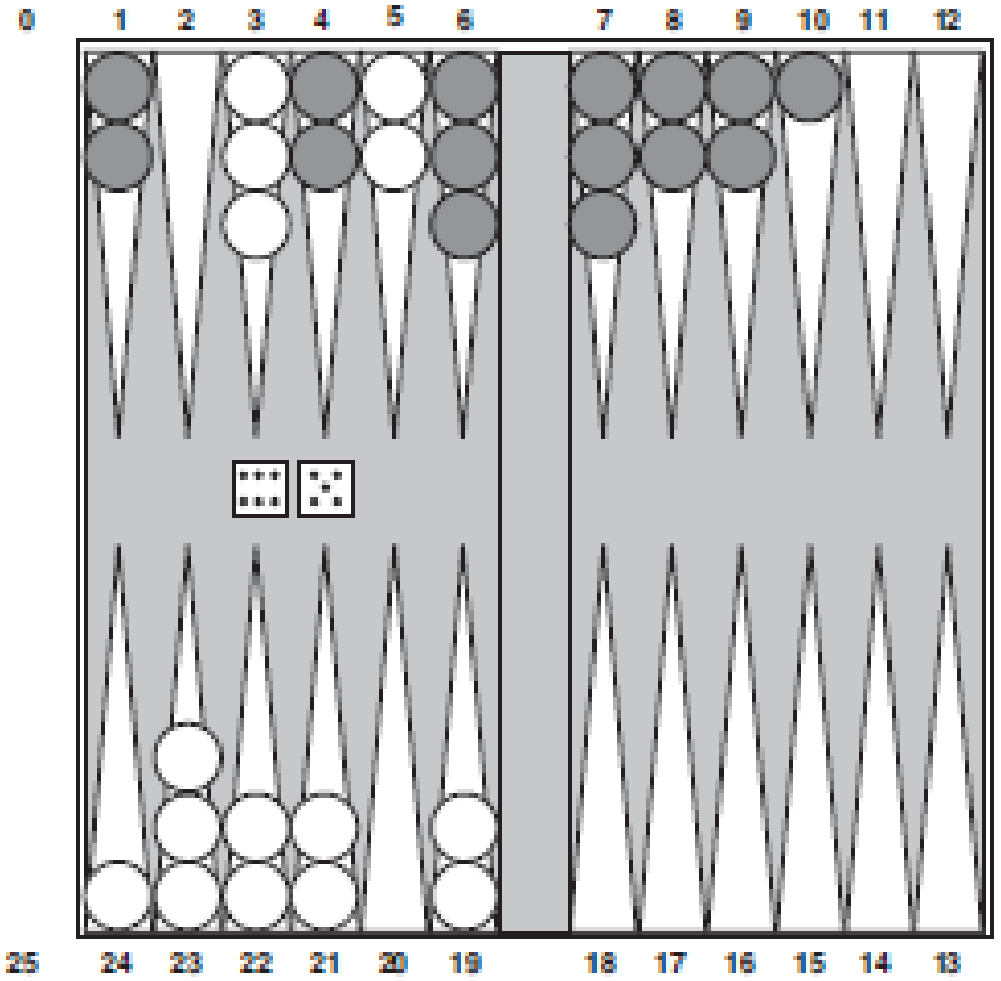
games mirror unpredictability by including a random element - throwing of dice.

A backgammon position. The goal of the game is to move all one's pieces off the board. White moves clockwise toward 25, Black moves counterclockwise toward 0.

A piece can move to any position unless multiple opponent pieces are there; if there is one opponent, it is captured and must start over.

In the position shown, White has rolled 6–5 and must choose among four legal moves: (5–10,5–11), (5–11,19–24), (5–10,10–16), and (5–11,11–16),

(5–11,11–16) means move one piece from position 5 to 11, and then move a piece from 11 to 16.



STOCHASTIC GAMES

White knows what his or her own legal moves are,

White does not know what Black is going to roll and so, does not know what Black's legal moves will be.

White cannot construct a standard game tree of the sort we saw in chess and tic-tac-toe.

A game tree in backgammon must include chance nodes in addition to MAX and MIN nodes.

The branches leading from each chance node denote the possible dice rolls

There are 36 ways to roll two dice, each equally likely;

but because a 6–5 is the same as a 5–6, there are only 21 distinct rolls.

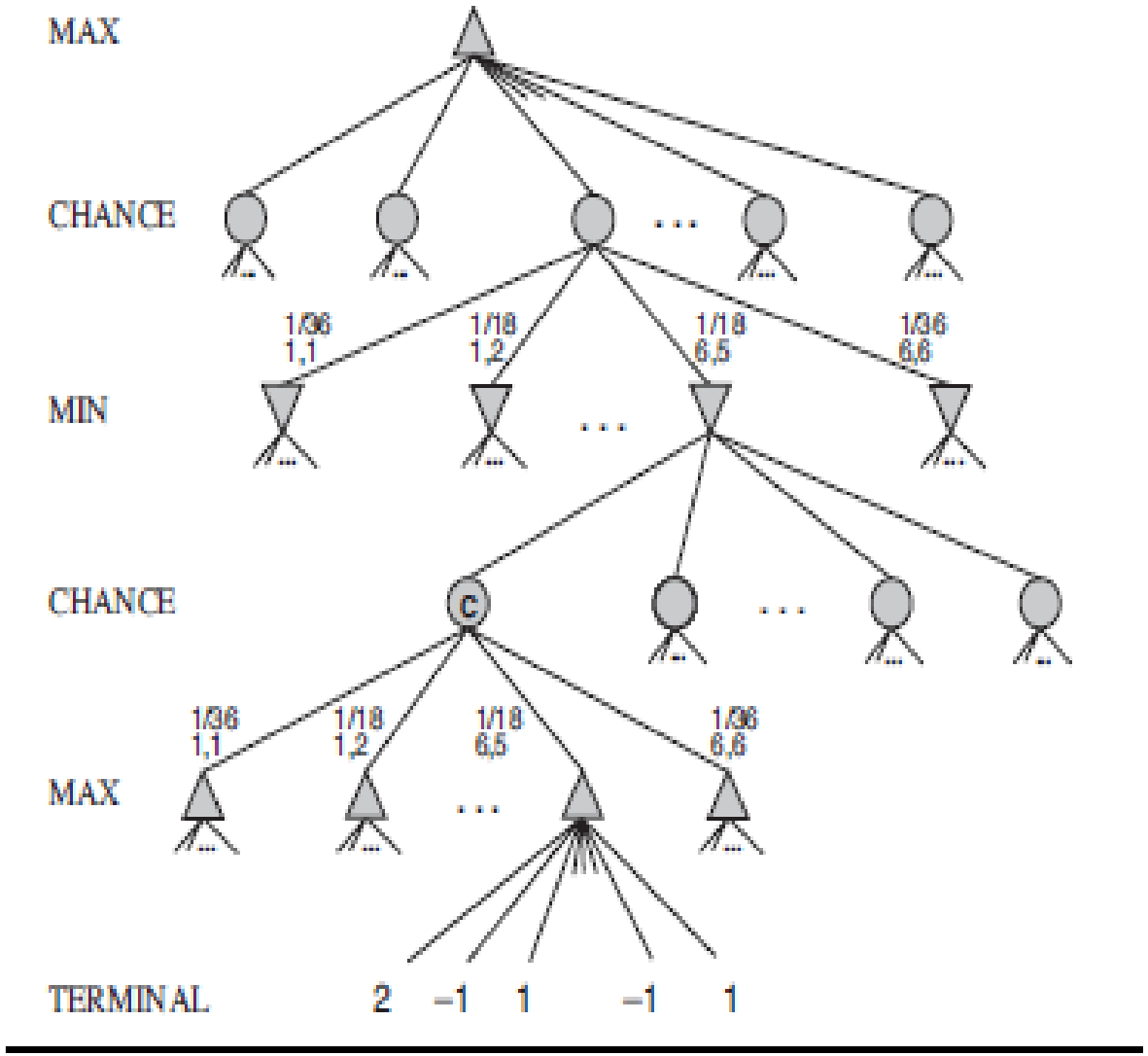
The six doubles (1–1 through 6–6) each have a probability of 1/36, so we say P(1–1) = 1/36.

The other 15 distinct rolls each have a 1/18 probability.

STOCHASTIC GAMES

Schematic game tree for a backgammon position.

The branches leading from each chance node denote the possible dice rolls; each branch is labeled with the roll and its probability.



STOCHASTIC GAMES

how to make correct decisions?

Need to pick the move that leads to the best position

positions do not have definite minimax values.

we can only calculate the expected value of a position:

the average over all possible outcomes of the chance nodes.

generalize the **minimax value** for deterministic games to

expectiminimax value for games with chance nodes.

Terminal nodes and MAX and MIN nodes (for which the dice roll is known) work exactly the same way as before.

For chance nodes we compute the expected value,

which is the sum of the value over all outcomes,

weighted by the probability of each chance action:

STOCHASTIC GAMES

how to make correct decisions?

Need to pick the move that leads to the best position

positions do not have definite minimax values.

we can only calculate the expected value of a position:

the average over all possible outcomes of the chance nodes.

EXPECTIMINIMAX(s) =

UTILITY(s) if TERMINAL-TEST(s)

$\max_a$ EXPECTIMINIMAX(RESULT(s, a)) if PLAYER(s)= MAX

$\min_a$ EXPECTIMINIMAX(RESULT(s, a)) if PLAYER(s)= MIN

$\sum_r$ P(r)EXPECTIMINIMAX(RESULT(s, r)) if PLAYER(s)= CHANCE

r represents a possible dice roll (or other chance event) and

RESULT(s, r) is the same state as s, with the additional fact that the result of the dice roll is r.

Move ordering, Evaluation Function for games of chance

with an evaluation function that assigns the values [1, 2, 3, 4] to the leaves, move a1 is best;

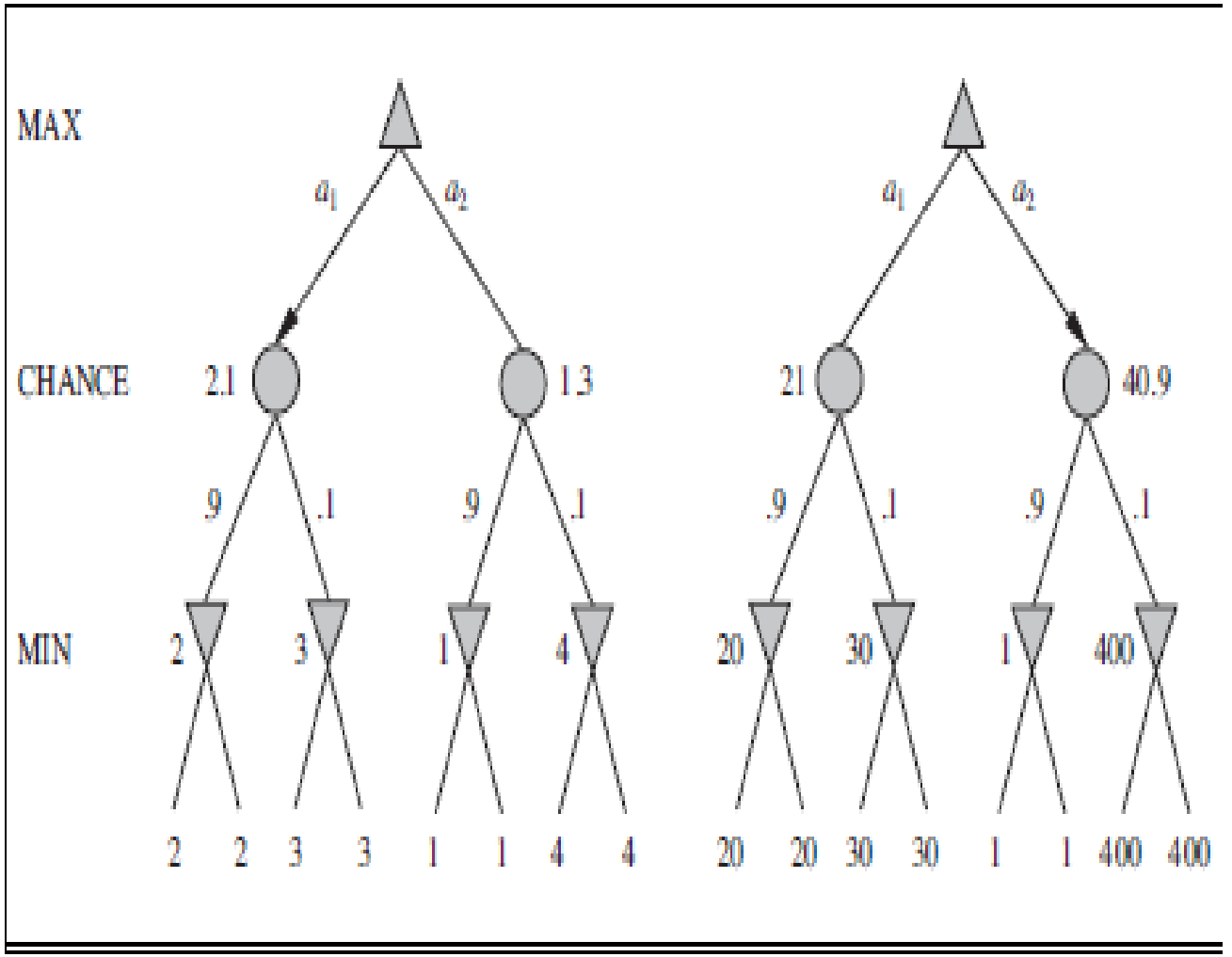
with values [1, 20, 30, 400], move a2 is best.

the program behaves totally differently if we make a change in the scale of some evaluation values!

to avoid this sensitivity -

the evaluation function must be a positive linear transformation of the probability of winning from a position

order-preserving transformation on leaf values changes the best move.



Evaluation Functions for games of chance

If the program knew in advance all the dice rolls that would occur for the rest of the game,

solving a game with dice would be just like solving a game without dice,

which minimax does in $O(b^m)$ time,

where b is the branching factor and m is the maximum depth of the game tree.

Because expectiminimax is also considering all the possible dice-roll sequences,

it will take $O(b^m n^m)$,

where n is the number of distinct rolls.

Even if the search depth is limited to some small depth d, the extra cost compared with that of minimax makes it unrealistic to consider looking ahead very far in most games of chance.

if we put bounds on the possible values of the utility function, then we can arrive at bounds for the average without looking at every number and prune the chance nodes.

An alternative is to do **Monte Carlo simulation** to evaluate a position – **Roll out**

Start with an alpha–beta (or other) search algorithm. From a start position, have the algorithm play thousands of games against itself, using random dice rolls. In the case of backgammon, the resulting win percentage has been shown to be a good approximation of the value of the position

State of the Art Game Programs – IBM's Deep Blue

Chess: IBM's Deep Blue chess program (retired)

H/W: 30 IBM RS/6000 processors

Search – alpha-beta search

Uniqueness:

configuration of 480 custom VLSI chess processors that performed move generation and move ordering for the last few levels of the tree, and evaluated the leaf nodes

Deep Blue searched up to 30 billion positions per move, reaching depth 14 routinely.

key to its success seems to have been

its ability to generate singular extensions beyond the depth limit

for sufficiently interesting lines of forcing/forced moves.

State of the Art Game Programs – IBM's Deep Blue

In some cases the search reached a depth of 40 plies.

The evaluation function had over 8000 features,

many of them describing highly specific patterns of pieces.

An "opening book" of about 4000 positions

a database of 700,000 grandmaster games from which consensus recommendations could be extracted.

large endgame database of solved positions containing

all positions with five pieces and many with six pieces.

Endgame database had the effect of substantially extending the effective search depth, allowing Deep Blue to play perfectly in some cases even when it was many moves away from checkmate.

The success of DEEP BLUE reinforced the widely held belief that progress in computer

game-playing has come primarily from ever-more-powerful hardware—

a view encouraged by IBM.

State of the Art Game Programs –

algorithmic improvements have allowed programs running on standard PCs to win World Computer Chess Championships.

A variety of pruning heuristics are used to reduce the effective branching factor to less than 3 (compared with the actual branching factor of about 35).

null move heuristic generates a good lower bound on the value of a position,

using a shallow search in which the opponent gets to move twice at the beginning.

This lower bound often allows alpha–beta pruning without the expense of a full-depth search.

futility pruning helps decide in advance which moves will cause a beta cutoff in the successor nodes

HYDRA

HYDRA successor to DEEP BLUE.

H/W: 64-processor cluster with 1 gigabyte per processor

and with custom hardware in the form of FPGA (Field Programmable Gate Array) chips.

HYDRA reaches 200 million evaluations per second, about the same as Deep Blue,

HYDRA reaches 18 plies deep rather than just 14

aggressive use of the **null move heuristic and forward pruning.**

RYBKA (winner of the 2008 and 2009 World Computer Chess Championships)

H/W: an off-the-shelf 8-core 3.2 GHz Intel Xeon processor

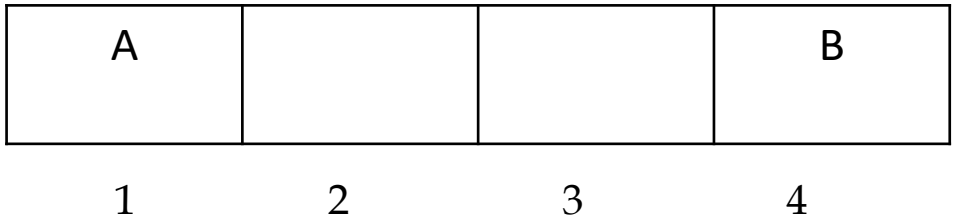
little is known about the design of the program.

RYBKA's main advantage appears to be its evaluation function,

which has been tuned by its main developer,

International Master Vasik Rajlich, and at least three other grandmasters.

Assignment/Homework



The starting position of a simple game. Player A moves first. The two players take turns moving, and each player must move his token to an open adjacent space in either direction.

If the opponent occupies an adjacent space, then a player may jump over the opponent to the next open space if any.

(For example, if A is on 3 and B is on 2, then A may move back to 1.)

The game ends when one player reaches the opposite end of the board.

If player A reaches space 4 first, then the value of the game to A is +1;

if player B reaches space 1 first, then the value of the game to A is −1.

Assignment/Homework - 1

Consider the two-player game described in the previous slide

a. Draw the complete game tree, using the following conventions:

- Write each state as (s_A, s_B), where s_A and s_B denote the token locations.
- Put each terminal state in a square box and write its game value in a circle.
- Put loop states (states that already appear on the path to the root) in double square boxes. Since their value is unclear, annotate each with a "?" in a circle.

b. Now mark each node with its backed-up minimax value (also in a circle). Explain how you handled the "?" values and why.

c. Explain why the standard minimax algorithm would fail on this game tree and briefly sketch how you might fix it, drawing on your answer to (b).

Does your modified algorithm give optimal decisions for all games with loops?

d. This 4-square game can be generalized to n squares for any n > 2. Prove that A wins if n is even and loses if n is odd.

Assignment/Homework - 2

This problem exercises the basic concepts of game playing, using tic-tac-toe (noughts and crosses) as an example. We define Xn as the number of rows, columns, or diagonals with exactly n X's and no O's. Similarly, On is the number of rows, columns, or diagonals with just n O's. The utility function assigns +1 to any position with X3 = 1 and −1 to any position with O3 = 1. All other terminal positions have utility 0. For nonterminal positions, we use a linear evaluation function defined as Eval (s) = 3X2(s)+X1(s)−(3O2(s)+O1(s)).

a. Approximately how many possible games of tic-tac-toe are there?
b. Show the whole game tree starting from an empty board down to depth 2 (i.e., one X and one O on the board), taking symmetry into account.
c. Mark on your tree the evaluations of all the positions at depth 2.
d. Using the minimax algorithm, mark on your tree the backed-up values for the positions at depths 1 and 0, and use those values to choose the best starting move.
e. Circle the nodes at depth 2 that would not be evaluated if alpha–beta pruning were applied, assuming the nodes are generated in the optimal order for alpha–beta pruning.

Assignment/Homework - 3

Develop a general game-playing program, capable of playing a variety of games.

a. Implement move generators and evaluation functions for one or more of the following games: chess.

b. Construct a general alpha–beta game-playing agent.

c. Compare the effect of increasing search depth, improving move ordering, and improving the evaluation function. How close does your effective branching factor come to the ideal case of perfect move ordering?