Intelligent Systems (IS)

Master's Degree in Informatics Engineering

Carlos Ansótegui

Área: Ciencias de la Computación e Inteligencia Artificial (CCIA)
Departamento de Informática e Ingeniería Industrial
Escuela Universitaria Politécnica
Universidad de Lleida

October 8, 2014

Index

Introduction

We examine the problems that arise when we try to plan ahead in a world where other agents are planning against us

- **Multiagent environments**: each agent needs to consider the actions of other agents and how they affect its own welfare
- The unpredictability of these other agents introduce **contingencies**
- We look into **competitive environments**, where agents' goals are in conflict, giving rise to **adversarial games**

Games

- Most common games are: deterministic, turn-taking, two-player, zero-sum games of perfect information (e.g. chess)
- Games are too hard to solve:
  Example: Chess average branching factor is 35.
  Games often go to 50 moves by each player.
  Therefore, search tree has about 35^{100} nodes
- Require the ability to make some decision even when calculating the optimal decision is infeasible
- Penalize inefficiency severely. An A* search that is half efficient will simply take twice time to execute. A chess program being half efficient can be beaten, being the rest of things equal

Types of Games

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

Two Player Games (Max-Min)

- Max moves first
- Players take turns alternatively until the game is over
- At the end, the points go to winner and penalties to loser

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

- S_0 : the initial state, specifies how the game is set up at the start
- PLAYER(s): defines which player has the move in a state s
- ACTIONS(s): returns the set of legal moves in a state s
- RESULT(s, a): the transition model, which defines the result of an action a from state s
- TERMINAL-TEST(s): a terminal test, which is true when the game is over and false otherwise

Two Player Games (Max-Min)

- UTILITY(s, p): A utility function, defines the final numeric value for a game that ends in terminal state s for a player p.

  In chess, outcome is a win, loss, or draw, with values $+1, 0$, or 1

  Some games have wider variety of possible outcomes; the payoffs in backgammon range from 0 to $+192$.

  A zero-sum game is (confusingly) defined as one where the total payoff to all players is the same for every instance of the game.

  Chess is zero-sum because every game has payoff of either $0 + 1$, $1 + 0$ or $\frac{1}{2} + \frac{1}{2}$

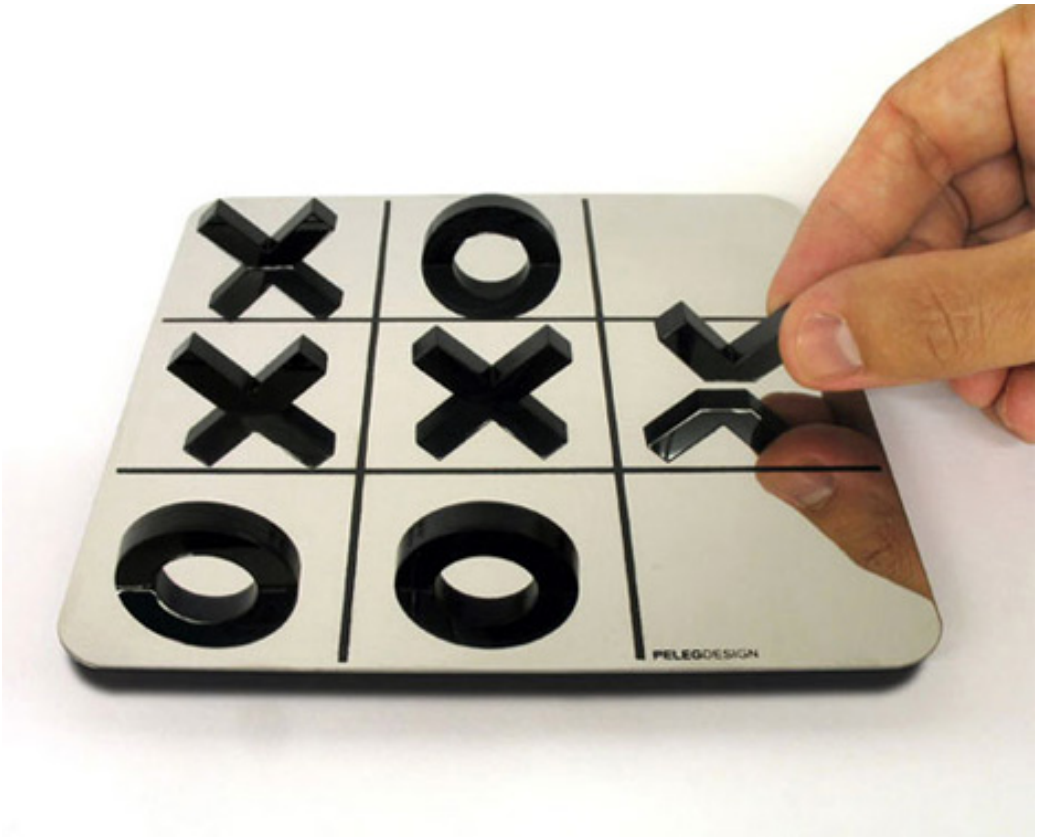
  "Constant-sum" is a better term. Zero-sum makes sense if you imagine each player is charged an entry fee of $\frac{1}{2}$.

  Exercise: Look for examples of non zero-sum games

The initial state, ACTIONS function, and RESULT function define the **game tree**, where nodes are game states and the edges are moves

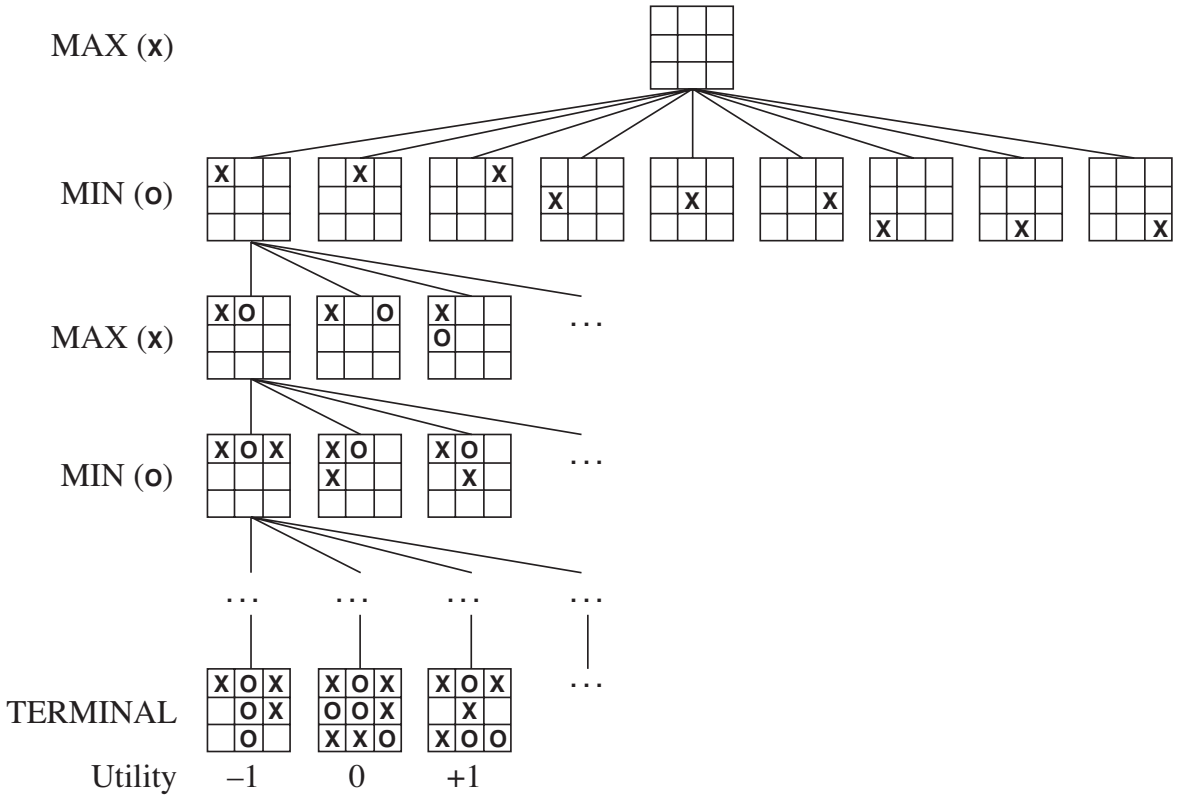
Two Player Games (Max-Min)

Exercise: define formally tic-tac-toe



Two Player Games (Max-Min)

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



Optimal Decision in Games

In a normal search problem, the optimal solution is a sequence of actions leading to a goal state (terminal state that is a win).

In adversarial search, an optimal solution for *MAX* is a tree of actions specifying what to do depending of what *MIN* plays.

Optimal Decision in Games

An example game tree:

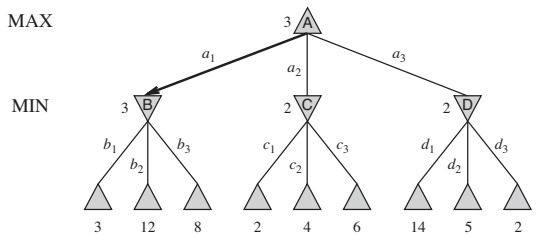


Figure 1: Two-ply game tree

The possible moves for MAX at root are a_1, a_2 and a_3.
The possible replies for a_1 are b_1, b_2 and b_3.
This game is one move deep. Consisting on two half-moves, each of which is called a **ply**.
The terminal nodes show the utility values for MAX.
The nonterminal nodes show the minimax values (see next two slides).

Optimal Decision in Games

Given a game tree, the optimal strategy can be determined from the minimax value of each node, which we write as $MINIMAX(n)$.

The minimax value of a node is the utility (for MAX) of being in the corresponding state, *assuming that both players play optimally* from there to the end of the game.

Obviously, the minimax value of a terminal state is just its utility.

Furthermore, given a choice, MAX prefers to move to a state of a maximum value whereas MIN prefers a state of minimum value

Optimal Decision in Games

MINIMAX(s) =

$$\begin{cases} UTILITY(s) & \text{if TERMINAL-TEST(s)} \\ max_{a \in ACTIONS(s)} MINIMAX(RESULT(s,a)) & \text{if } PLAYER(s) = MAX \\ min_{a \in ACTIONS(s)} MINIMAX(RESULT(s,a)) & \text{if } PLAYER(s) = MIN \end{cases}$$

In the previous example, the *minimax decision* at the root is a_1.

Exercise: substitute in the previous example the utility values for terminal nodes by: $7, 1, 5, 20, 13, 12, 7, 3, 2$. Determine MINIMAX value for states A, B, C and D.

Exercise: What if MIN does not play optimally? Show that MAX will do even better.

Other strategies against suboptimal opponents may do better than the minimax strategy, but these strategies necessarily do worse against optimal opponents.

The MiniMax Algorithm

Naive approach:

1. Expand the tree below root node n
2. Evaluate terminal nodes
3. Select a node n' without value such that all their children have been already evaluated.

   If such a node does not exist, return the value of n
4. if n' is MAX, assign as value the maximum value of its children
   if n' is MIN, assign as value the minimum value of its children

   GOTO 3

The MiniMax Algorithm

Naive approach:

1. Expand the tree below root node n
2. Evaluate terminal nodes
3. Select a node n' without value such that all their children have been already evaluated.

   If such a node does not exist, return the value of n
4. if n' is MAX, assign as value the maximum value of its children
   if n' is MIN, assign as value the minimum value of its children

   GOTO 3

This algorithm has a problem:

The MiniMax Algorithm

Naive approach:

1. Expand the tree below root node n
2. Evaluate terminal nodes
3. Select a node n' without value such that all their children have been already evaluated.

   If such a node does not exist, return the value of n
4. if n' is MAX, assign as value the maximum value of its children
   if n' is MIN, assign as value the minimum value of its children

   GOTO 3

This algorithm has a problem:

It is a Breadth First Search ... That implies exponential memory! 

The MiniMax Algorithm

Iterative version of DFS approach (L: stack)

1. $L := \langle n \rangle$
2. $x := L.top()$

   If $x = n$ and has a value, return the value of n
3. If x has value w, let node p parent of x with value v

   If p is MAX, assign $max(v, w)$ to p
   If p is MIN, assign $min(v, w)$ to p

   $L.pop()$
4. If x has no value and TERMINAL-TEST(x), assign UTILITY(x) to x
5. If x has no value and not TERMINAL-TEST(x),

   assign $-\infty$ if x is MAX, ∞ otherwise.
   For each child x' of x : $L.push(x')$
6. GOTO 2

The MiniMax Algorithm

For example, in Figure 1:

The algorithm first goes down to the three bottom-left nodes and uses UTILITY to discover values $3, 12$ and 8.

Then, takes the minimum of these values , 3 and returns it as the backed-up value of node B.

A similar process gives the back-up values of 2 for C and 2 for D.

Finally, we take the minimum of $3, 2$ and 2 to get the back-up value of 3 for the root node

Properties of the MiniMax Algorithm

Complete? Yes, if tree is finite (chess has specific rules for this).

Optimal? Yes, against an optimal opponent. Otherwise?.

Time complexity? $O(b^m)$.

Space complexity? $O(b \cdot m)$ (if all actions are generated all at once).

If actions are generated one at a time it is $O(m)$.

m: maximum depth of the tree is m.

b: number of legal moves at each point.

The MiniMax Algorithm

Second approach (recursive version):

function MINIMAX-DECISION(*state*) **returns** *an action*
  return $\arg\max_{a \in \text{ACTIONS}(s)}$ MIN-VALUE(RESULT(*state*, *a*))

---

function MAX-VALUE(*state*) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow -\infty$
  for each *a* **in** ACTIONS(*state*) **do**
    $v \leftarrow$ 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*)
  $v \leftarrow \infty$
  for each *a* **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MIN(v, MAX-VALUE(RESULT(s, a)))
  return v

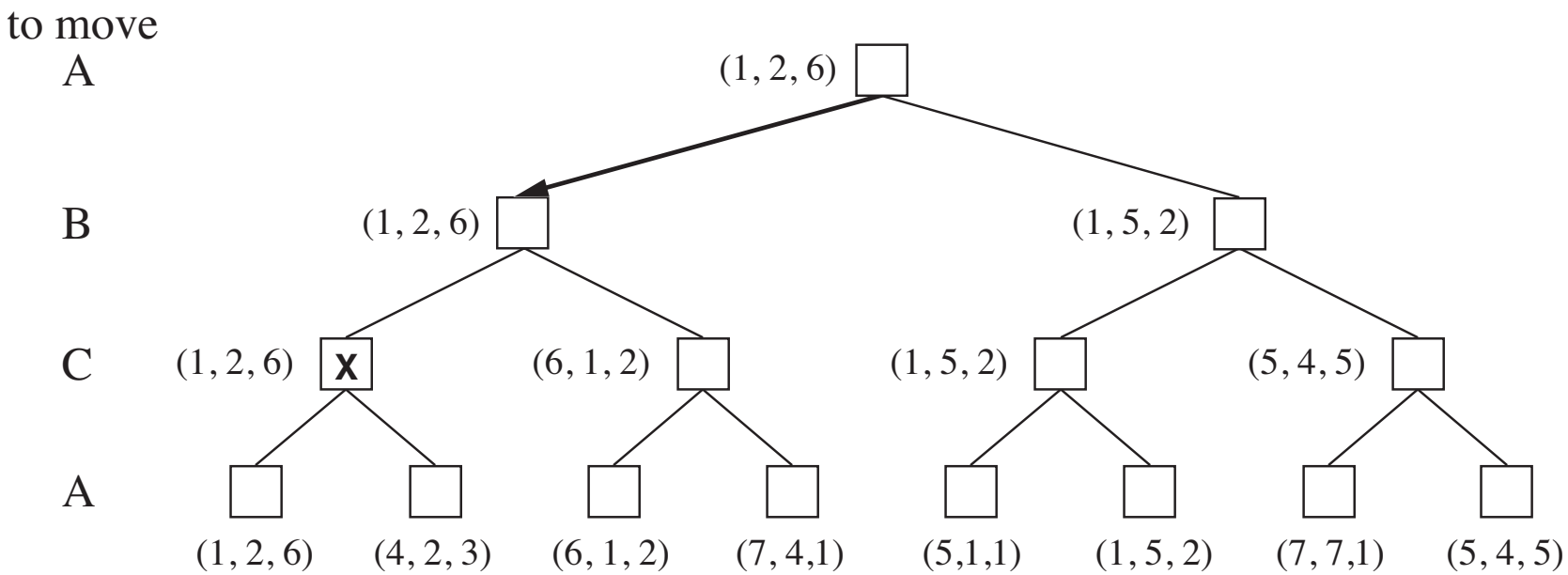
Optimal decisions in multiplayer games

Let us examine how to extend the minimax idea to multiplayer games:

- Replace the single value for each node with a vector of values
  For example, lets have three players A, B and C:
  a vector $\langle v_A, v_b, v_c \rangle$ is associated with each node
- For terminal states, the vector gives the utility for each player
- For nonterminal states, the back-up value of a node n is always the utility vector of the successor with the highest value for the player choosing at n

Optimal decisions in multiplayer games

Example of Minimax for multiplayer games:



Optimal decisions in multiplayer games

- Multiplayer games usually involve **alliances** (informal or formal)
- Alliances are made and broken as the game proceeds
- Are alliances a natural consequence of optimal strategies? They can be

  Example: A and B are in weak positions and C is in stronger position. Then, it is often optimal for *A* and *B* to attack *C*
- In some cases, alliances make concrete what would have happened anyway
- If the game is not zero-sum, collaboration can also occur with just two players

Alpha-Beta Pruning

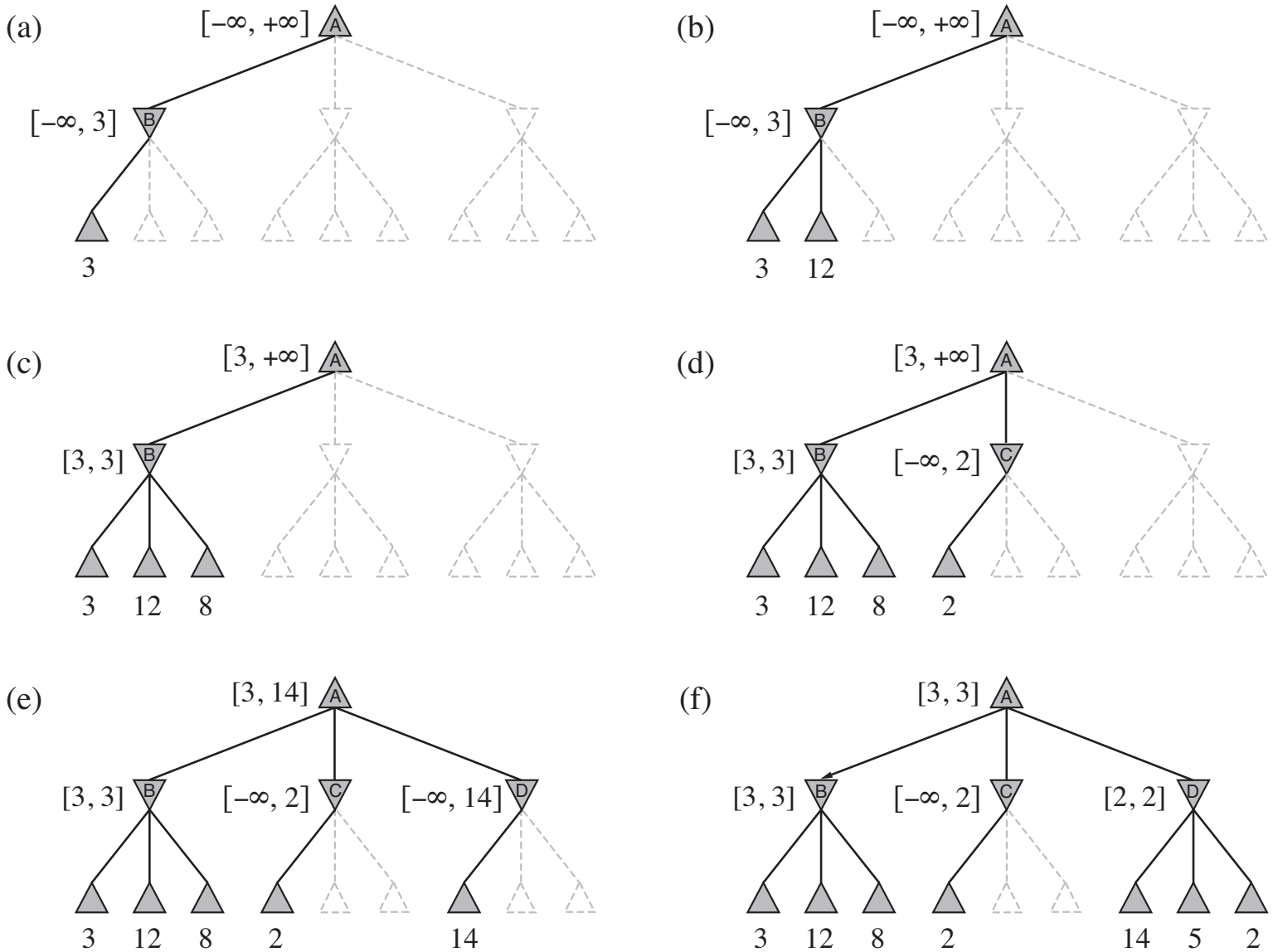
The problem with minimax is that number of game states it examines is exponential in the depth of the tree.

We can not eliminate the exponent, but we can effectively cut it in half!.

Idea: compute the correct minimax decision without looking at every node in the game tree.

We prune away branches that can not possibly influence the final decision.

Alpha-Beta Pruning



Alpha-Beta Pruning

Stages in the calculation of the optimal decision for the game tree. At each point, we show the range of possible values for each node.

(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

Alpha-Beta Pruning

(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 alphabeta pruning

(e) The first leaf below D has the value 14, so B 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

We can look at it as simplification of the formula for MINIMAX.
Let the two unevaluated successors of node C have values x and y.
The the value of the root node is given by:

$$MINIMAX(root) =$$
$$= max(min(3, 12, 8), min(2, x, y), min(15, 5, 2))$$
$$= max(3, min(2, x, y), 2)$$
$$= max(3, z, 2) \quad where\ z = min(2, x, y) \leq 2$$
$$= 3$$

The value of root is independent of the values of the pruned x and y

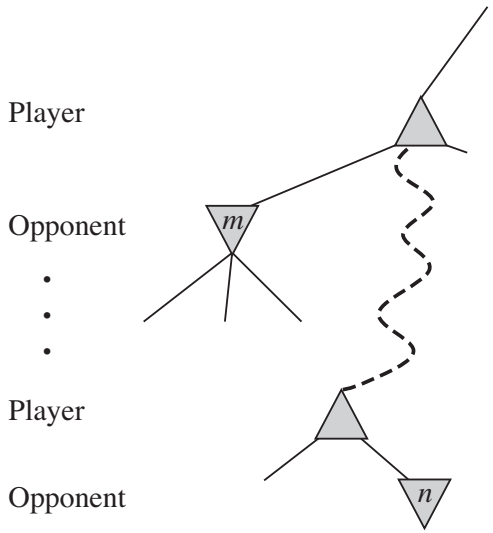
Alpha-Beta Pruning

General idea:

Consider a node *n* somewhere in the tree, 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, the *n will never be reached in actual play*.

Once we have found out enough about *n* (by examining some of its descendants) to reach this conclusion, we can prune it



Alpha-Beta Pruning (recursive version)

Minimax is Depth First, so at any one time we only have to consider the nodes along a single path in the tree.

Alpha-Beta gets its name from the following two parameters that describe bounds on the back-up values that appear anywhere along the path:

- $\alpha =$ value of best choice (highest value) along the path for MAX
- $\beta =$ value of best choice (lowest value) along the path for MIN

Alpha-Beta Pruning

Alpha-Beta search updates the value of α and β as it goes along and prunes the remaining branches at a node as soon as the value of the current node is known to be worse than the current α or β value for MAX or MIN, respectively.

Alpha-Beta Pruning

function ALPHA-BETA-SEARCH(*state*) **returns** an action
  $v \leftarrow$ MAX-VALUE($state, -\infty, +\infty$)
  return the *action* in ACTIONS(*state*) with value v

---

function MAX-VALUE($state, \alpha, \beta$) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow -\infty$
  for each a **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if $v \geq \beta$ **then return** v
    $\alpha \leftarrow$ MAX(α, v)
  return v

---

function MIN-VALUE($state, \alpha, \beta$) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow +\infty$
  for each a **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if $v \leq \alpha$ **then return** v
    $\beta \leftarrow$ MIN(β, v)
  return v

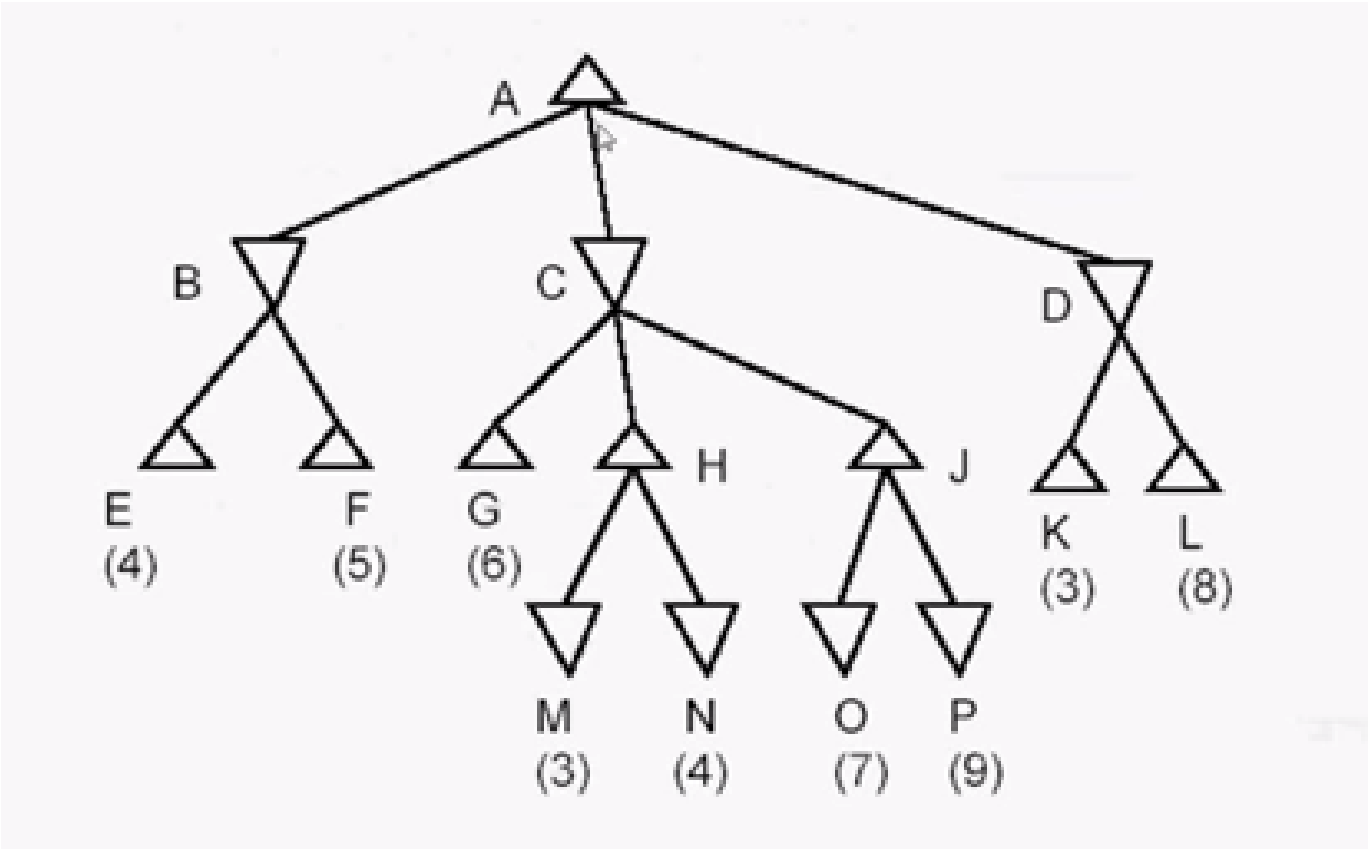
Alpha-Beta Pruning

Exercise: is this version better?

```
function ALPHA-BETA-DECISION(state) returns an action
    return the a in ACTIONS(state) maximizing MIN-VALUE(RESULT(a, state))

---

function MAX-VALUE(state, α, β) 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 ← −∞
    for a, s in SUCCESSORS(state) do
        v ← MAX(v, MIN-VALUE(s, α, β))
        if v ≥ β then return v
        α ← MAX(α, v)
    return v

---

function MIN-VALUE(state, α, β) returns a utility value
    same as MAX-VALUE but with roles of α, β reversed
```

Alpha-Beta Pruning

Exercise:



Alpha-Beta Pruning

Link **Solution**

Link **Another Example**

Exercise: write the iterative version of Alpha-Beta

Properties of Alpha-Beta

Pruning does not affect final result.

Good move ordering improves effectiveness of pruning.

Think about our example, for node D we were not able to prune anything because of the order of the successors.

With “perfect ordering" time complexity = $O(b^{m/2})$.

Dynamic ordering, such trying first the best movements in the past, bring us quite close to the theoretical limit!.

Properties of Alpha-Beta

Dynamic ordering, such trying first the best movements in the past, bring us quite close to the theoretical limit!.

A way to do this is to apply an iterative deepening approach.

Repeated states occur frequently because of **transpositions** ("different permutations of the move sequence that end up in the same position").

Use a hash table to store the evaluation of positions (called transposition table).

This is similar to graph search.

Imperfect RealTime Decisions

How do people manage to play a game like chess?.

We do not analyze it all the way to the end.

We just look far enough ahead so that we can *estimate* who is likely to win in some nonterminal position and then back-up the values from the nonterminal positions we have found.

Imperfect RealTime Decisions

Programs should cutoff search earlier and apply a heuristic **evaluation function** to states in the search.

Replace the utility function by a heuristic evaluation function EVAL, which estimates the position's utility.

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

H-MINIMAX(s) $=$

$$\begin{cases} EVAL(s) & \text{if CUTOFF-TEST(s)} \\ max_{a \in ACTIONS(s)} H - MINIMAX(RESULT(s,a)) & \text{if } PLAYER(s) = MAX \\ min_{a \in ACTIONS(s)} H - MINIMAX(RESULT(s,a)) & \text{if } PLAYER(s) = MIN \end{cases}$$

Evaluation functions

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

How exactly do we design good evaluation functions?

1. It should order the terminal states in the same way as the true utility function
2. The computation must not take too long!
3. For nonterminal states, it should be strongly correlated with the actual *chances of winning*

Evaluation functions

Most evaluation functions work by calculating various features of the state.

Example: in chess, we would have features for the number of white pawns, black pawns, white queens, black queens, and so on.

The features, taken together, define various categories or equivalence classes of states.

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

Example: one category contains all two-pawns vs. one pawn endgames.

Evaluation functions

Some states in a category will lead to win and some to looses.

The evaluation function can return a single value that reflects the proportion of states with each outcome.

Example: suppose 72% of the states in (two-pawns vs. one pawn) lead to a win (utility +1); 20% to a loss (0), and 8% to a draw (1/2).

The **expected value** is : $(0.72 \times +1) + (0.20 \times 0) + (0.08 \times 1/2) = 0.76$

For terminal states, the EVAL function does not need to return actual expected values a long as the *ordering* of the states is the same.

Evaluation functions

The previous kind of analysis require too many categories and hence too much experience to estimate all probabilities of wining.

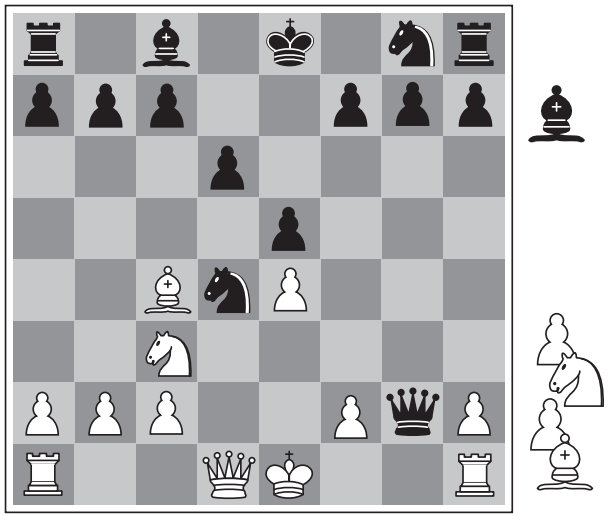
Most evaluation functions compute separate numerical contributions from each feature and then combine them.

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

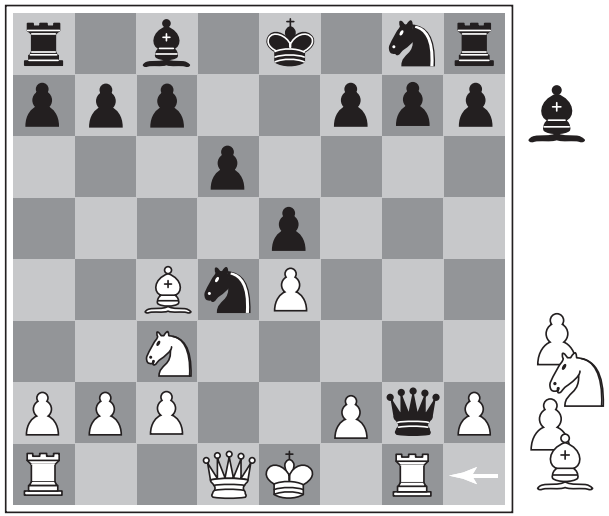
Other features such as *good pawn structure* and *king safety* might be worth half of a pawn.

Evaluation functions

Example: a secure advantage equivalent to pawn gives a substantial likelihood of winning, and a secure advantage equivalent to three pawns should give almost certain victory (see figure on the left):



(a) White to move



(b) White to move

Evaluation functions

Mathematically, the previous kind of computation is called a *weighted linear function* because ti can be expressed as:

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

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

Example: for chess, f_i could be the numbers of each kind of piece on the board and the w_i could be the values of the pieces.

Evaluation functions

Adding up the values involves a strong assumption assumption: the **contribution** of each feature is **independent** of the values of the other features.

Example: assigning the value of 3 to a bishop ignores the fact that bishops are more powerful in the end game, when they have a lot of space to maneuver.

Therefore, current programs for chess or other games also use **nonlinear** combinations for features.

Example: a pair of bishops might be worth slightly more than twice the value of a single bishop, and a bishop is worth more in the endgame (that is, when the *move number* feature is high or the number of remaining pieces feature is low)

Cutting off search

Idea: modify Alpha-Beta search such that it will call the heuristic EVAL function when it is appropriate to cut off the search. We replace in Alpha-Beta algorithm the two lines that mention TERMINAL-TEST with the following line:

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

First, we need to update *depth* properly.

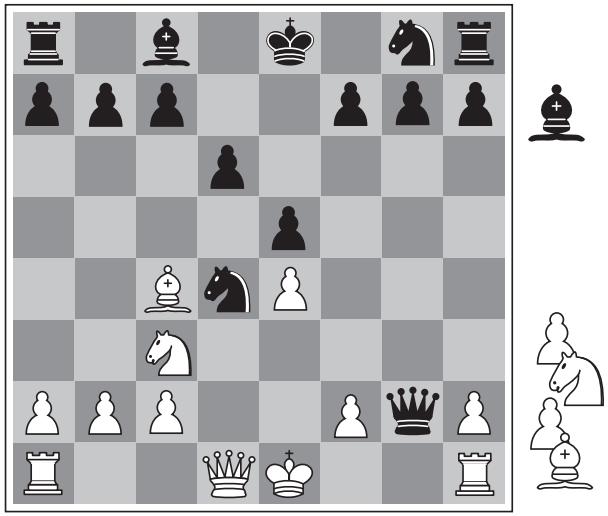
Second, to control the amount of search is to set a limit depth limit so that CUTOFF-TEST(state,depth) return true for all depth greater than some fixed depth d (also returns true for all terminal tests).

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

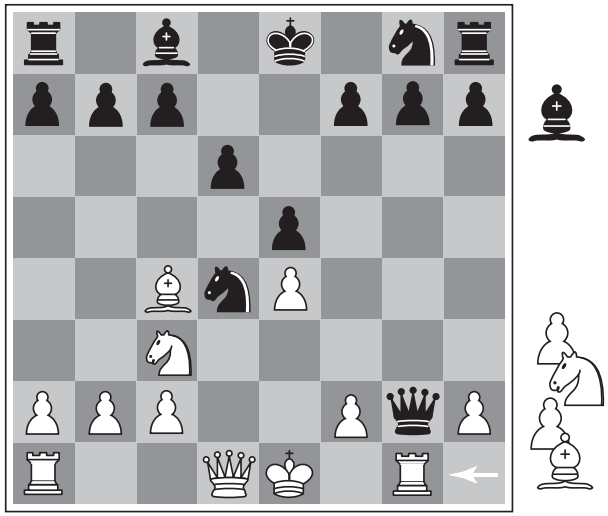
A more robust approach is to apply iterative deepening. When times runs out, the program returns the move selected by the deepest completed search (as a bonus it helps with the move ordering).

Cutting off search: common errors

Lets see common errors due to the approximate nature of the evaluation function.



(a) White to move



(b) White to move

Example: suppose we reach situation (b) at the depth limit. Back is ahead by a knight and two pawns. It would report a probable win for Black. However, next, White can capture Black's Queen with no compensation. Actually, the position in (b) is good for White!, but this can only be seen by looking ahead one more ply.

Cutting off search: quiescent search

The evaluation function should only be applied to positions that are **quiescent**, i.e., unlikely to exhibit wild swings in value in the near future.

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

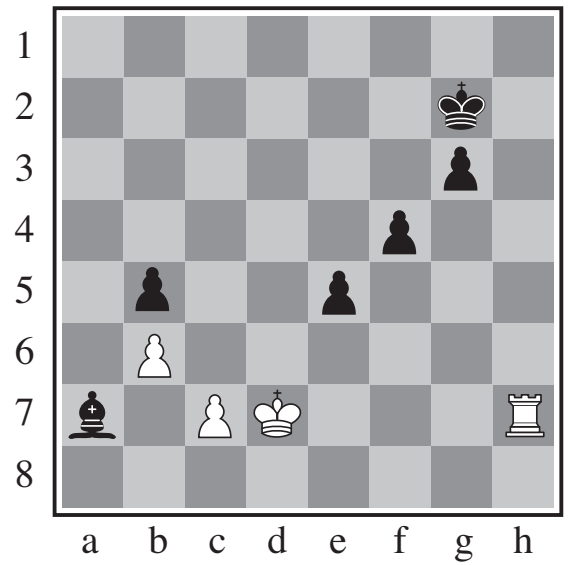
This extra search is called a **quiescent search**.

Cutting off search: horizon effect

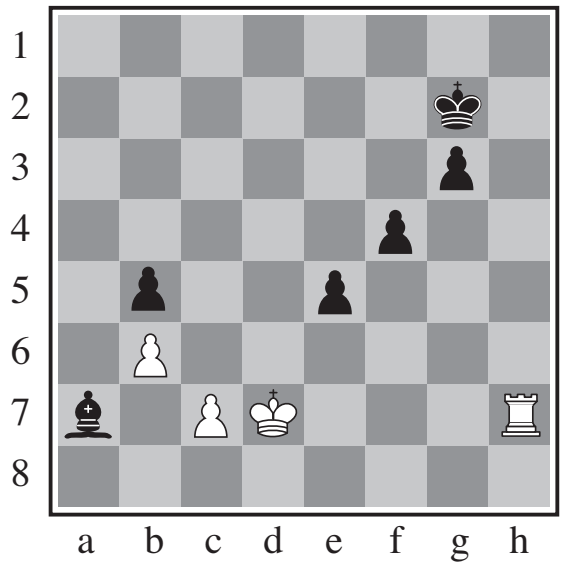
The **horizon effect** is more difficult to eliminate.

The program faces an opponent's move that causes serious damage and is ultimately unavoidable, but can be temporally avoided by delaying tactics.

Example:



Cutting off search: horizon effect



It is clear that there is no way for the Black bishop to escape.

But Black does have a sequence of moves that pushes the capture of the bishop *over the horizon*.

Therefore, Black may think that the line of play has saved the bishop, but he has pushed the inevitable capture beyond the horizon he sees.

Cutting off search: singular extension

One strategy to mitigate the horizon effect is the **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 can be applied.

This makes the tree deeper, but in general there are few singular extensions.

Forward pruning

We can prune some moves at a given node without further consideration.

Most humans playing chess only consider a few moves (at lest consciously).

One approach of forward pruning is **beam search**: on each ply, consider only a *beam* for the *n* best moves.

Caution: no guarantee that the best move will not be pruned away!

Forward pruning

PROBCUT (Michael Buro, 1994) is a forward pruning version of alpha-beta search that uses statistics gained from prior experience to lessen the chance that the best move will be pruned.

PROBCUT prunes nodes that are *probably* outside the current (α, β) window.

To compute the probability it applies a shallow search to compute the back-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).

Search vs. lookup

Consider openings in a chess game. Would it be make sense to start the game by considering a tree of one billion states, only to conclude *e*4?

We can simply look to a predefined set of openings in a Chess Book ...

Moreover, computers can gather statistics from a database of previously played games to see which *opening sequences* lead more often to a win.

Near the end of the game there are fewer positions, and thus more chance to lookup. It is here where the computer has the expertise.

Search vs. lookup

Computer analysis of end games goes far beyond anything achieved by humans!

Example: a human can tell the general strategy for playing a king-and-rook-versus-king (KRK) endgame.

Other endings, such as king, bishop, and knight versus king (KBNK) have not a succinct strategy description.

A computer can solve completely the game and produce a *policy*: mapping from every possible state to the best move in that state.

Then, in the next game we just need to lookup the best move!

Search vs. lookup

How big will be the lookup table for KBNK?

There are 462 ways two kings can be placed on the board without being adjacent. Then, there are 62 empty spaces for the bishop, 61 for the knight, and two possible payers to move next:

$462 \times 62 \times 61 \times 2 = 3494568$ possible positions.

Some are checkmates, and we mark them.

Then, we can apply a *retrograde* minimax search: reverse the rules of chess to do unmoves rather than moves.

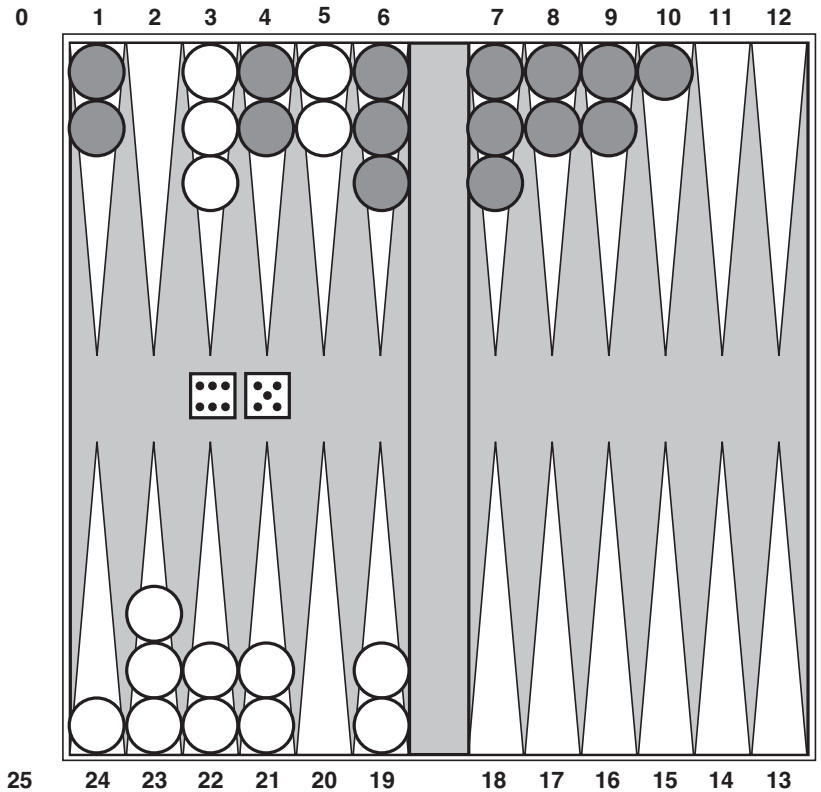
Any move by White, no matter what Black does respond with, ends up in a position marked as win, must also be a win.

Continue the search till the 3494568 positions are marked as win, loss or draw.

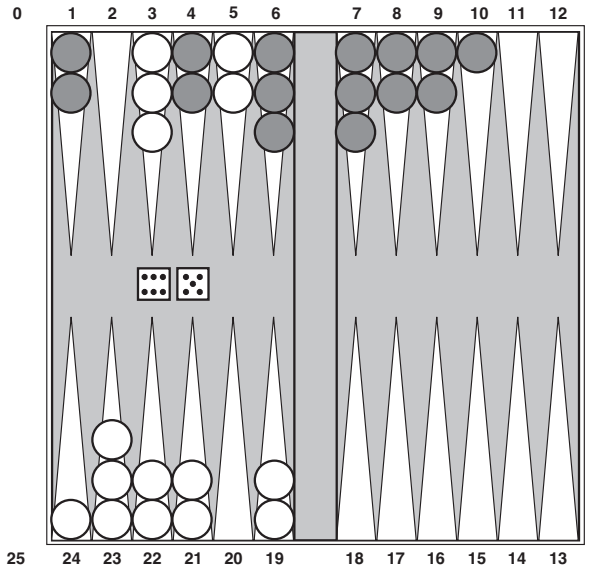
Stochastic Games

In real life, many unpredictable external events can put us into unforeseen situations.

Many games mirror this unpredictability by including a random element (e.g. throwing a dice).



Stochastic Games

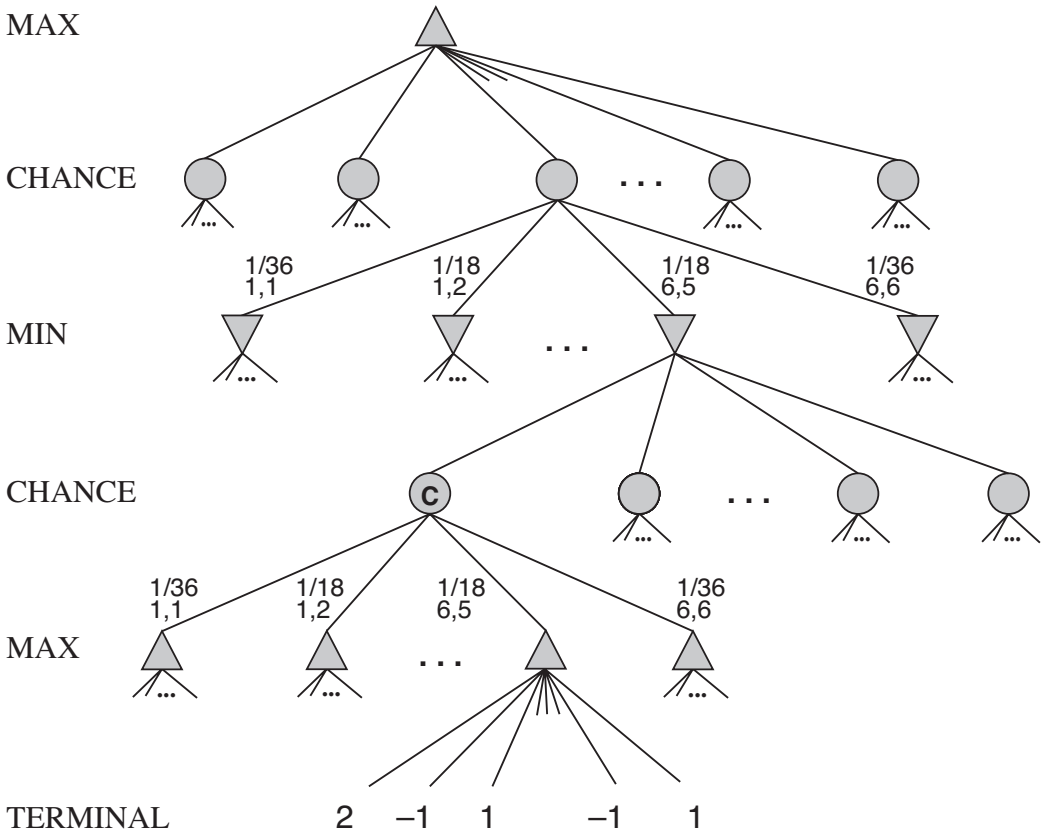


A typical backgammon position. The goal of the game is to move all one's pieces off the board. White moves clockwise toward 25, and Black moves counterclockwise toward 0. A piece can move to any position unless multiple opponent pieces are there; if there is one opponent, it is captured and must start 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), where the notation (5-11,11-16) means move one piece from position 5 to 11, and then move a piece from 11 to 16

Stochastic Games: Backgammon

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

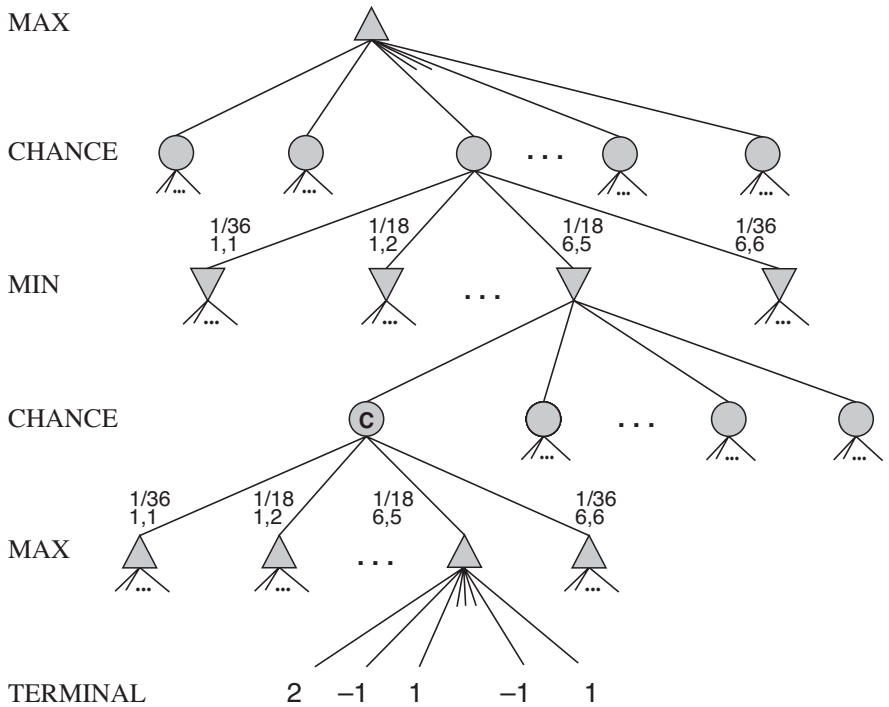
Therefore, a game tree in **backgammon** must include **chance nodes** in addition to MAX and MIX nodes.



Stochastic Games: Backgammon

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

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: Backgammon

Positions do not have definite minimax values.

Instead, we can only calculate the **expected value** of a position: the average over all possible outcomes of the chance nodes.

We generalize the **minimax value** for deterministic games to an **expectiminimax value** for games with chance nodes.

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

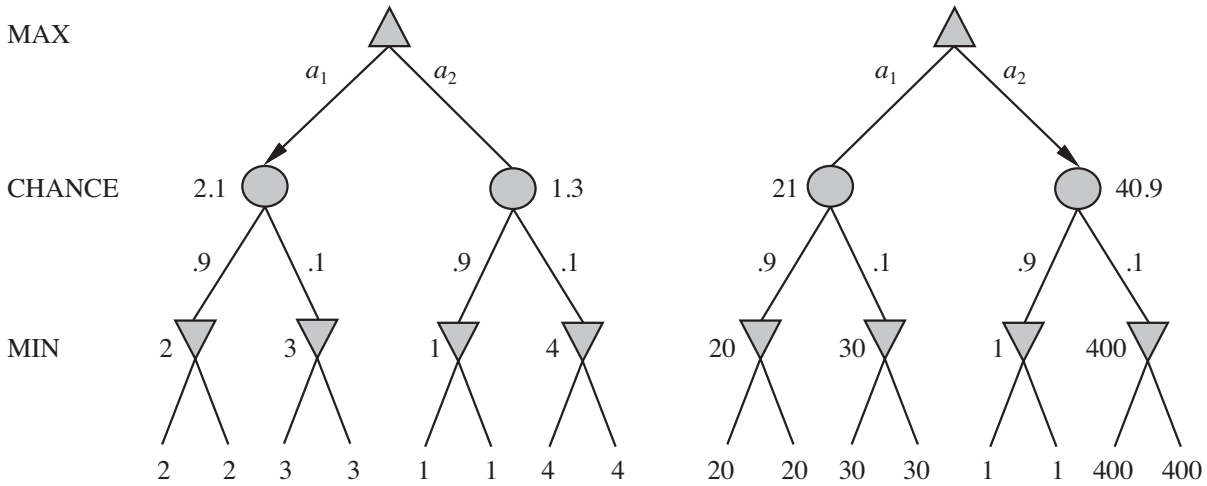
For chance nodes we compute the expected value, which is the sum of the value over all outcomes, weighted by the probability of each chance action:

Stochastic Games: Expectiminimax

$$\text{EXPECTIMINIMAX(s)} = \begin{cases} \mathit{UTILITY}(s) & \text{if TERMINAL-TEST(s)} \\ \max_{a \in \mathit{ACTIONS}(s)} \mathit{EXPECTIMINIMAX}(\mathit{RESULT}(s,a)) & \text{if } \mathit{PLAYER}(s) = \mathit{MAX} \\ \min_{a \in \mathit{ACTIONS}(s)} \mathit{EXPECTIMINIMAX}(\mathit{RESULT}(s,a)) & \text{if } \mathit{PLAYER}(s) = \mathit{MIN} \\ \sum_r P(r) \cdot \mathit{EXPECTIMINIMAX}(\mathit{RESULT}(s,r)) & \text{if } \mathit{PLAYER}(s) = \mathit{CHANCE} \end{cases}$$

Evaluation functions for games of chance

The presence of chance nodes means that one has to be more careful about what the evaluation values mean.



With an evaluation function that assigns the values [1,2,3,4] to leaves, move a_1 is the best. With values [1,20,30,400] move a_2 is the best!

In order to avoid this sensitivity, the evaluation function must be a positive linear transformation of the probability of winning from a position (or, more generally, of the expected utility of the position).

Complexity

If the program knew in advance all the dice rolls, solving a game would be like solving the game without dice, i.e., as minimax $O(b^m)$, where b is the branching factor and m the maximum depth.

Since expectiminimax is also considering all the possible dice-roll sequences, it is $O(b^m \cdot n^m)$ where n is the number of distinct rolls.

The extra cost compared with minimax makes it unrealistic to consider looking very far in most games of chance.

Example: in backgammon n is 21 and b is usually around 20, but in some situations can be as high as 4000 for dice rolls that are doubles. We can not manage efficiently more than three plies.

Complexity

We reacall that the advantage of AlphaBeta is that it ignores future developments that are no going to happen, given best play, i.e., it concentrates on *likely occurrences*.

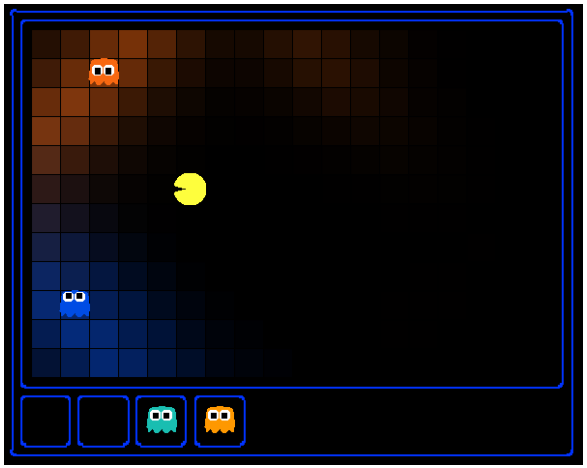
In games with dice, there are no likely sequences of moves, because for those moves to take place, the dice would have first to come out the right way to make them legal.

However, kind of AlphaBeta pruning can be applied if we can put bounds on the possible values of the utility function.

Partially Observable Games

Chess has often been described as war in miniature, but it lacks one major characteristic of real wars, namely, **partial observability**.

In the *fog of war* the existence and disposition of enemy units is often unknown until revealed by direct contact.



Kriegspiel: Partially observable chess

In deterministic partially observable games, uncertainty about the state of the board arises entirely from lack of access to the choices made by the opponent.

Rules of Kriegspiel:

White and Black each see a board containing only their own pieces.

A referee, who can see all the pieces, adjudicates the game and periodically makes announcements that are heard by both players.

On his turn, White proposes to the referee any move that would be legal if there were no black pieces.

If the move is in fact not legal (because of the black pieces), the referee announces ’illegal."

In this case, White may keep proposing moves until a legal one is found (and learns more about the location of Black’s pieces in the process).

Kriegspiel: Partially Observable Games

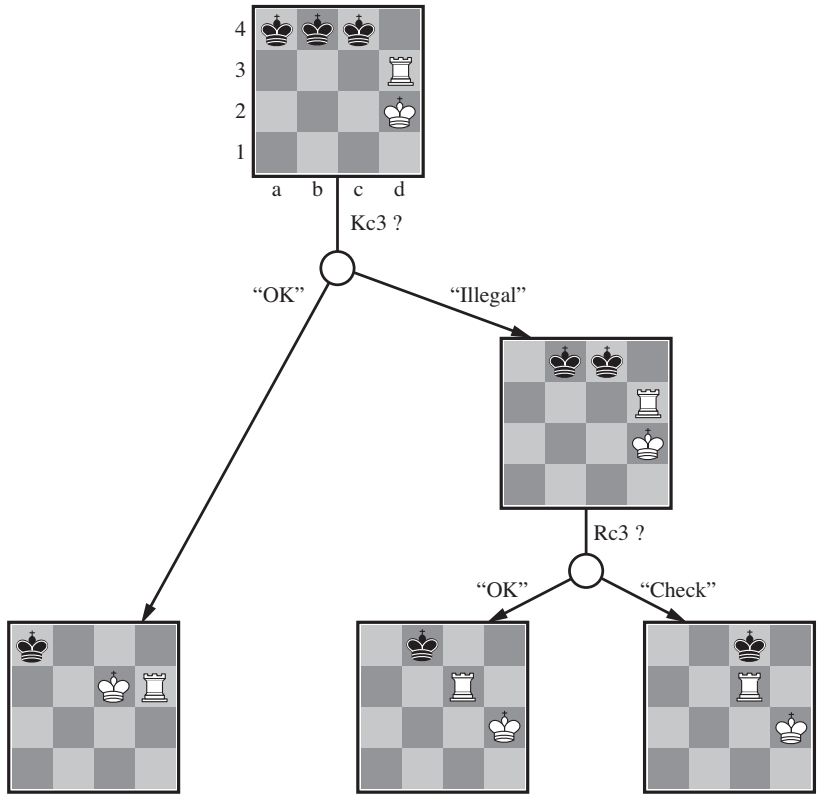
Once a legal move is proposed, the referee announces one or more of the following:

"Capture on square X" if there is a capture, and "Check by D" if the black king is in check, where D is the direction of the check, and can be one of "Knight," "Vertical," “Horizontal”," "Long diagonal," or "Short diagonal."

(In case of discovered check, the referee may make two "Check" announcements.)

If Black is checkmated or stalemated, the referee says so; otherwise, it is Black’s turn to move.

Kriegspiel: Partially Observable Games



Part of a guaranteed checkmate in the KRK endgame. In the initial belief state, Black's king is in one of three possible locations. By a combination of probing moves, the strategy narrows this down to one.

Exercise: Complete the checkmate.

Kriegspiel: Partially Observable Games

We need the notion of **belief state**: the set of all logically possible board states given the complete history of percepts to date.

Example: Initially, White's belief state is a singleton because Black's pieces haven't moved yet. After White makes a move and Black responds, White's belief state contains 20 positions because Black has 20 replies to any White move.

Keeping track of the belief state as the game progresses is exactly the problem of **state estimation**.

The notion of **strategy** is altered; instead of specifying a move to make for each possible move the opponent might make, we need a move for every possible *percept sequence* that might be received.

A winning strategy or **guaranteed checkmate**, is one that, for each possible percept sequence, leads to an actual checkmate for every possible board state in the current belief state, regardless of how the opponent moves.

Card Games

In many games, cards are dealt randomly at the beginning of the game, with each player receiving a hand that is not visible to the other players. Example: bridge, whist, hearts, and some sorts of poker.

Since all cards are dealt at the beginning one can consider all possible deals of the invisible cards; solve each one as if it were a fully observable game; and then choose the move that has the best outcome averaged over all the deals.

Suppose that each deal s occurs with probability $P(s)$; then the move we want is:

$$argmax_a \sum P(s) \cdot MINIMAX(RESULTS(s, a))$$

Card Games

In most card games, the number of deals is rather large!

Example: for bridge is 10400600. Solving even one deal is quite difficult, so solving a ten million is out if the question.

However, we can resort to a Monte Carlo approximation: instead of adding up all the deals, we take a random sample of N deals where the probability of deal s appearing in the sample is proportional to $P(s)$:

$$argmax_a \frac{1}{N} \sum_{i=1}^{N} P(s) \cdot MINIMAX(RESULTS(s_i, a))$$