5 ADVERSARIAL SEARCH

Competitive environments, in which the agent's goals are in conflict, give rise to **adversarial search** problems – often known as **games.**

5.1 Games

Mathematical **Game Theory**, a branch of economics, views any **multiagent environment** as a **game** provided that the impact of each agent on the other is "significant", regardless of whether the agents are cooperative or competitive. In, **AI**, "games" are deterministic, turn-taking, two-player, zero-sum games of perfect information. This means deterministic, fully observable environments in which there are two agents whose actions must alternate and in which the **utility values** at the end of the game are always equal and opposite. For example, if one player wins the game of chess(+1),the other player necessarily loses(-1). It is this opposition between the agents' utility functions that makes the situation **adversarial.**

Formal Definition of Game

We will consider games with two players, whom we will call **MAX** and **MIN**. MAX moves first, and then they take turns moving until the game is over. At the end of the game, points are awarded to the winning player and penalties are given to the loser. A **game** can be formally defined as a **search problem** with the following components :

- The **initial state**, which includes the board position and identifies the player to move.
- A **successor function**, which returns a list of (*move*,*state*) pairs, each indicating a legal move and the resulting state.
- A **terminal test**, which describes when the game is over. States where the game has ended are called **terminal states**.
- A **utility function** (also called an objective function or payoff function),which give a numeric value for the terminal states. In chess, the outcome is a win, loss, or draw, with values +1,-1,or 0. he payoffs in backgammon range from +192 to -192.

Game Tree

The **initial state** and **legal moves** for each side define the **game tree** for the game. Figure 2.18 shows the part of the game tree for tic-tac-toe (noughts and crosses). From the initial state,MAX has nine possible moves. Play alternates between MAX's placing an X and MIN's placing a 0 until we reach leaf nodes corresponding to the terminal states such that one player has three in a row or all the squares are filled. He number on each leaf node indicates the utility value of the terminal state from the point of view of MAX; high values are assumed to be good for MAX and bad for MIN. It is the MAX's job to use the search tree(particularly the utility of terminal states) to determine the best move.

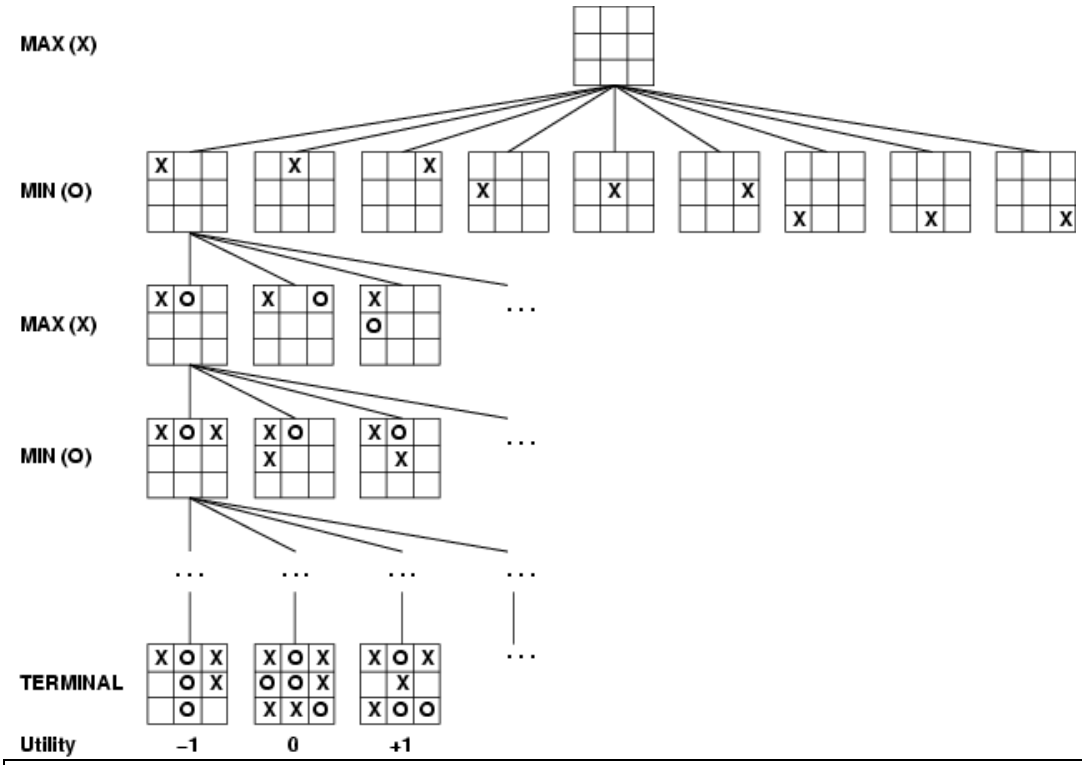


Figure 2.18 A partial search tree . The top node is the initial state,and MAX move first,placing an X in an empty square.

5.2 Optimal Decisions in Games

In normal search problem,the **optimal solution** would be a sequence of move leading to a **goal state** – a terminal state that is a win. In a game,on the other hand,MIN has something to say about it,MAX therefore must find a contingent **strategy**,which specifies MAX's move in the **initial state**,then MAX's moves in the states resulting from every possible response by MIN,then MAX's moves in the states resulting from every possible response by MIN those moves,and so on. An **optimal strategy** leads to outcomes at least as good as any other strategy when one is playing an infallible opponent.

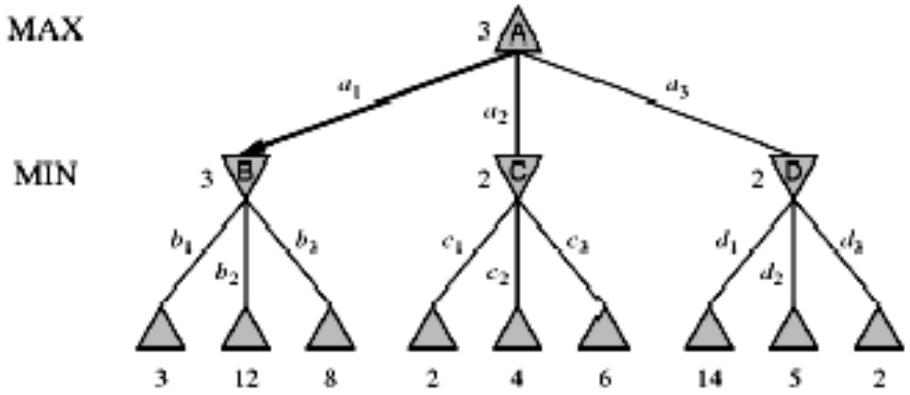


Figure 2.19 A two-ply game tree. The △ nodes are "MAX nodes",in which it is AMX's turn to move,and the ▽ nodes are "MIN nodes". The terminal nodes show the utility values for MAX; the other nodes are labeled with their minimax values. MAX's best move at the root is a_1,because it leads to the successor with the highest minimax value,and MIN's best reply is b_1,because it leads to the successor with the lowest minimax value.

Minimax Search: Algorithm

```
function MINIMAX-DECISION(state) returns an action
  inputs: state, current state in game

  v ← MAX-VALUE(state)
  return the action in SUCCESSORS(state) with value v

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

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

Figure 2.20 An algorithm for calculating minimax decisions. It returns the action corresponding to the best possible move,that is,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 minimax Algorithm

The minimax algorithm(Figure 2.20) computes the minimax decision from the current state. It uses a simple recursive computation of the minimax values of each successor state,directly implementing the defining equations. The recursion proceeds all the way down to the leaves of the tree ,and then the minimax values are **backed up** through the tree as the recursion unwinds**.** For example in Figure 2.19,the algorithm first recourses down to the three bottom left nodes,and uses the utitliy function on them to discover that their values are 3,12,and 8 respectively. Then it takes the minimum of these values,3,and returns it as the backed-up value of node B. A similar process gives the backed up values of 2 for C and 2 for D. Finally,we take the maximum of 3,2,and 2 to get the backed-up value of 3 at the root node. The minimax algorithm performs a complete depth-first exploration of the game tree. If the maximum depth of the tree is *m*,and there are b legal moves at each point,then the time

complexity of the minimax algorithm is $O(b^m)$. The space complexity is O(bm) for an algorithm that generates successors at once.

5.3 Alpha-Beta Pruning

The problem with minimax search is that the number of game states it has to examine is **exponential** in the number of moves. Unfortunately,we can't eliminate the exponent,but we can effectively cut it in half. By performing **pruning,**we can eliminate large part of the tree from consideration. We can apply the technique known as **alpha beta pruning** ,when applied to a minimax tree ,it returns the same move as **minimax** would,but **prunes away** branches that cannot possibly influence the final decision.

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

- α : the value of the best(i.e.,highest-value) choice we have found so far at any choice point along the path of MAX.
- β: the value of best (i.e., lowest-value) choice we have found so far at any choice point along the path of MIN.

Alpha Beta search updates the values of α and β as it goes along and prunes the remaining branches at anode(i.e.,terminates the recursive call) as soon as the value of the current node is known to be worse than the current α and β value for MAX and MIN,respectively. The complete algorithm is given in Figure 2.21.

The effectiveness of alpha-beta pruning is highly dependent on the order in which the successors are examined. It might be worthwhile to try to examine first the successors that are likely to be the best. In such case,it turns out that alpha-beta needs to examine only $O(b^{d/2})$ nodes to pick the best move,instead of $O(b^d)$ for minimax. This means that the effective branching factor becomes sqrt(b) instead of b – for chess,6 instead of 35. Put anotherway alpha-beta cab look ahead roughly twice as far as minimax in the same amount of time.

```
function ALPHA-BETA-SEARCH(state) returns an action
   inputs: state, current state in game
   v ← MAX-VALUE(state, −∞, +∞)
   return the action in SUCCESSORS(state) with value v

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
  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 ← MIN(v, MAX-VALUE(s, α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v
```

Figure 2.21 The alpha beta search algorithm. These routines are the same as the minimax routines in figure 2.20,except for the two lines in each of MIN-VALUE and MAX-VALUE that maintain α and β

5.4 Imperfect ,Real-time Decisions

The minimax algorithm generates the entire game search space,whereas the alpha-beta algorithm allows us to prune large parts of it. However,alpha-beta still has to search all the way to terminal states for atleast a portion of search space. Shannon's 1950 paper,Programming a computer for playing chess,proposed that programs should **cut off** the search earlier and apply a heuristic **evaluation function** to states in the search,effectively turning nonterminal nodes into terminal leaves. The basic idea is to alter minimax or alpha-beta in two ways :

(1) The utility function is replaced by a heuristic evaluation function EVAL,which gives an estimate of the position's utility,and

(2) the terminal test is replaced by a **cutoff test** that decides when to apply EVAL.

5.5 Games that include Element of Chance

Evaluation functions

An evaluation function returns an estimate of the expected utility of the game from a given position,just as the heuristic function return an estimate of the distance to the goal.

Games of imperfect information

- Minimax and alpha-beta pruning require too much leaf-node evaluations.
  May be impractical within a reasonable amount of time.
- SHANNON (1950):
  - Cut off search earlier (replace TERMINAL-TEST by CUTOFF-TEST)
  - Apply heuristic evaluation function EVAL (replacing utility function of alpha-beta)

Cutting off search

Change**:**

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

  into

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

Introduces a fixed-depth limit *depth*

- Is selected so that the amount of time will not exceed what the rules of the game allow.

When cuttoff occurs, the evaluation is performed.

Heuristic EVAL

Idea: produce an estimate of the expected utility of the game from a given position.

Performance depends on quality of EVAL.

Requirements:

- EVAL should order terminal-nodes in the same way as UTILITY.
- Computation may not take too long.
- For non-terminal states the EVAL should be strongljy correlated with the actual chance of winning.

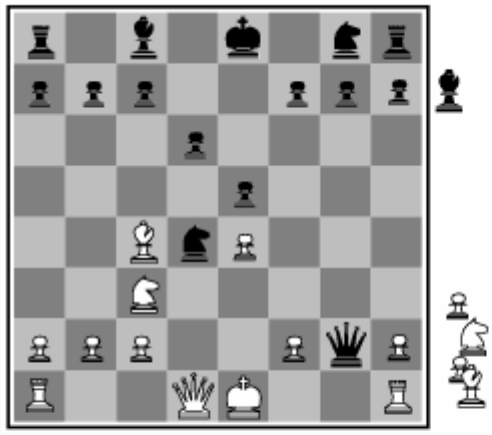
Only useful for quiescent (no wild swings in value in near future) states

Weighted Linear Function

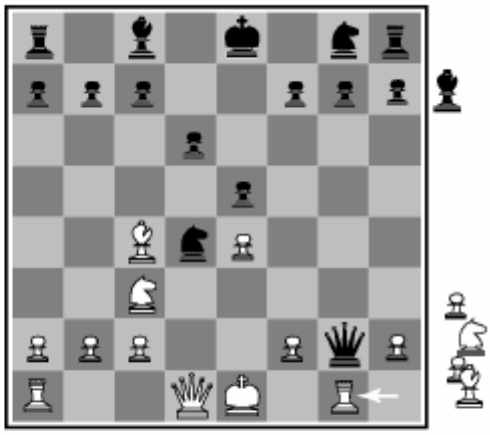
The introductory chess books give an approximate material value for each piece : each pawn is worth 1,a knight or bishop is worth 3,a rook 3,and the queen 9. These feature values are then added up toobtain the evaluation of the position. Mathematically,these kind of evaluation fuction is called weighted linear function,and it can be expressed as :

$$Eval(s) = w1\ f1(s) + w2\ f2(s) + \ldots + wn\ fn(s)$$

- e.g., w1 = 9 with
  f1(s) = (number of white queens) – (number of black queens), etc.



(a) White to move



(b) White to move

(a) Black has an advantage of a knight and two pawns and will win the game.
(b) Black will lose after white captures the queen.
