CHAPTER 6

Small Multiple Player Games

6.1 Introduction

Having dealt with small single-player games, we turn now to small multiple-player games. In most cases, the other players are general game playing programs or humans. However, in some cases, the other players represent uncertainty in the game itself. For example, it is common to model some card games by representing a randomly shuffled deck of cards as an additional player in the game, one that deals or reveals cards as the game progresses.

Multiple-player games are more complicated than single-player games because the state resulting from a player's actions can depend on the actions of the other players. No player can directly control the actions of other players; and so, in making its choices, a player must consider all possible actions of the other players.

Before proceeding, it is worth emphasizing that games need not be *fixed sum*. In a fixed sum game, the total number of points is fixed. (When this number is zero, such games are usually said to be *zero-sum*.) In order for one player to get more points, some other player must lose points. For this reason, fixed sum games are necessarily competitive. In general game playing, there is no such restriction. Some games are competitive; but others are cooperative - it may be that the only way for one player to get a higher reward is to help the other players get higher rewards as well.

While it is possible, in some multiple-player games, to find sequential plans that produce maximal rewards, this is rarely the case. In order to achieve an optimal reward, it is frequently necessary for a player to conditionalize its actions on the state of the game. This is a situation where compulsive deliberation works well.

In this chapter, as in the preceding chapter, we look at settings in which there is sufficient time for players to search the game tree entirely. That said, as in single-player games, it is sometimes possible to find optimal actions even without searching the entire game tree.

We begin this chapter with a procedure called *Minimax*, and we then consider a more efficient variation called *Bounded Minimax*. We then turn to an even more efficient procedure called *Alpha-Beta Search*, which produces the same results but eliminates some of the needless computation of minimax.

6.2 Minimax

In general game playing, a player may choose to make assumptions about the actions of the other players. For example, a player might want to assume that the other players are behaving rationally. By eliminating irrational actions on the part of the other players, a player can decrease the number of possibilities, it needs to consider.

Unfortunately, in general game playing, as currently constituted, no player knows the identity of the other players. The other players might be irrational or they might behave the same as the player itself. Since there is no information about the other players, many general game players take a pessimistic approach - they assume that the other players will perform the worst possible actions. This pessimistic approach is the basis for a game-playing technique called *Minimax*. Note that Minimax is also used in other fields such as Economics or more traditional AI. The definition of Minimax in this chapter exactly matches the ones in the other fields in the case of a 2-player, constant-sum, alternating moves game.

In the case of a one-step game, Minimax chooses an action such that the value of the resulting state for *any* opponent action is greater than or equal to the value of the resulting state for any other action. In the case of a multiple-step game, Minimax goes to the end of the game and *backs up* values.

We can think about Minimax as search of a bipartite tree consisting of alternating *max nodes* and *min nodes*. See the example shown below. The max nodes (shown in beige) represent the choices of the player while the min nodes (shown in grey) represent the choices of the other players.

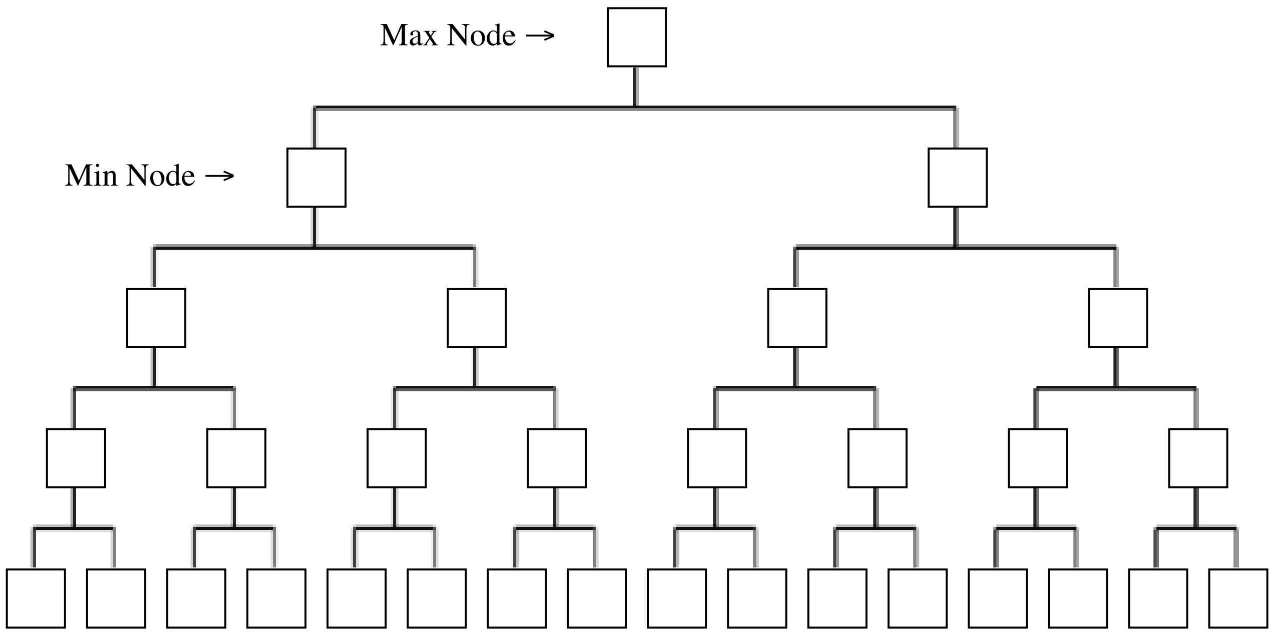
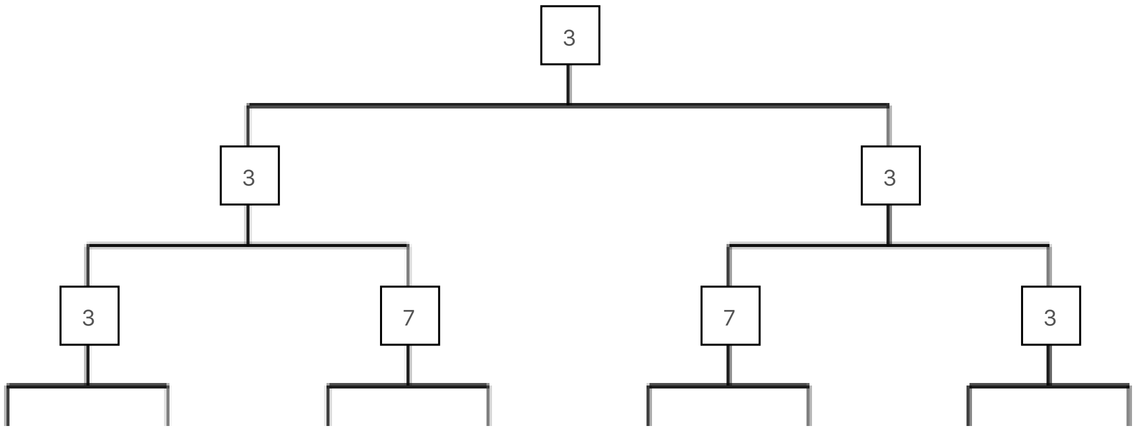


Figure 6.1 - A minimax tree

Note that, although we have separated the choices of the player and its opponents, this does not mean that play alternates between the opponents or that the opponents know the player's action. The player and its opponents make their choices simultaneously, without knowledge of each other's choices.

The value of a max node for a player is either the utility of that state if it is terminal or the maximum of all values for the min nodes that result from its legal actions. The value of a min node is the minimum value that results from any legal opponent action.

The following game tree illustrates this. The nodes at the bottom of the tree are terminal states, and the values are the player's goal values for those states. The values shown in the other nodes are computed according to the rules just stated. For example, the value of the minnode at the lower left is 1 because that is the minimum of the values of the maxnodes below it, viz. 1 and 2. The value of the minnode next to that minnode is 3 because that is the minimum of the values of the two maxnodes below it, viz. 3 and 4. The value of the maxnode above these two minnodes is 3 because that is the maximum of the values of the two minnodes. And so forth.



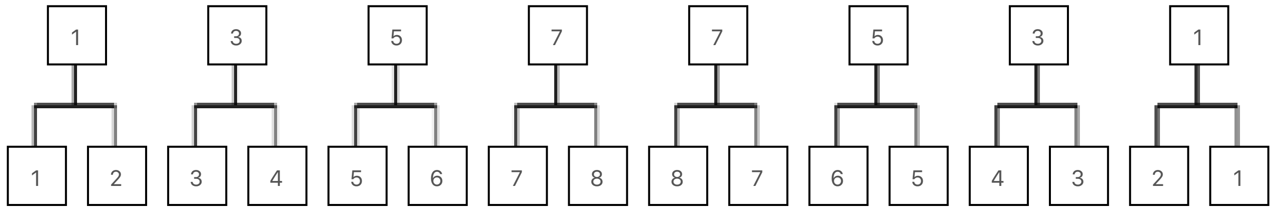


Figure 6.2 - A minimax tree with values

The following procedure is a simple implementation of a player that uses minimax to evaluate states. The implementation is similar to that of the compulsive deliberation player introduced in the preceding chapter. The `start` handler merely records relevant information in the player's global variables. The `play` handler simulates the previous move to obtain the current state and then finds the best action for it to perform in that state. In this case, it uses the `bestmove` subroutine to obtain this action.

```
function start (id,player,rules,sc,pc)
 {game = rules;
  role = player;
  roles = findroles(game);
  state = findinits(game);
  startclock = sc;
  playclock = pc;
  return 'ready'}

function play (id,move)
 {state = simulate(move,state);
  return bestmove(role,state)}
```

The main difference between the `bestmove` subroutine for single-player games and the `bestmove` for multiple-player games is the way scores are computed. Rather than comparing subsequent states, it compares min nodes as described above.

```
function bestmove (role,state)
 {var actions = findlegals(role,state,game);
  var action = actions[0];
  var score = 0;
  for (var i=0; i<actions.length; i++)
      {var result = minscore(role,actions[i],state);
       if (result>score) {score = result; action = actions[i]}};
  return action}
```

The `minscore` subroutine takes a role, an action of that role, and a state as arguments and produces the minimum values for the given role associated with the given action for any of the opponent's legal actions in the given state. The definition here is designed for two player games only. The code for games with more than two players is analogous. (The findopponent subroutine here uses the game description to compute the other player in a two-player game.)

```
function minscore (role,action,state)
 {var opponent = findopponent(role,game);
  var actions = findlegals(opponent,state,game);
  var score = 100;
  for (var i=0; i<actions.length; i++)
      {var move;
       if (role==roles[0]) {move = [action,actions[i]]}
          else {move = [actions[i],action]}
       var newstate = findnext(move,state,game);
       var result = maxscore(role,newstate);
       if (result<score) {score = result}};
  return score}
```

The `maxscore` subroutine, which is called by `minscore`, takes a role and a state as arguments. It conducts a recursive exploration of the game tree below the given state. If the state is terminal, the output is just the role's reward for that state. Otherwise, the output is the maximum of the utilities of the min nodes associated with the player's legal actions in the given state.

```
function maxscore (role,state)
 {if (findterminalp(state,game)) {return findreward(role,state,game)};
  var actions = findlegals(role,state,game);
  var score = 0;
  for (var i=0; i<actions.length; i++)
      {var result = minscore(role,actions[i],state);
       if (result>score) {score = result}};
  return score}
```

6.3 Bounded Minimax Search

One disadvantage of the Minimax procedure described in the preceding section is that it examines the entire game tree in all cases. While this is sometimes necessary, there are cases where it is possible to get the same result without examining the entire game tree. For example, if in processing a state the `maxscore` subroutine finds an action that produces 100 points, it does not need to look at any additional actions since it cannot do better; and if the `minscore` subroutine finds an action that produces 0 points, it does not need to look at any additional actions since it cannot get the score any lower.

Bounded Minimax is just the Minimax procedure just discussed. Rather than processing all actions on every node, it checks first for these bounds; and, if they occur on any node, it terminates its examination and returns the corresponding value.

As an example of this, consider the game tree shown below. The nodes with values are those examined by Bounded Minimax. The other nodes are not examined at all and do not need to be examined.

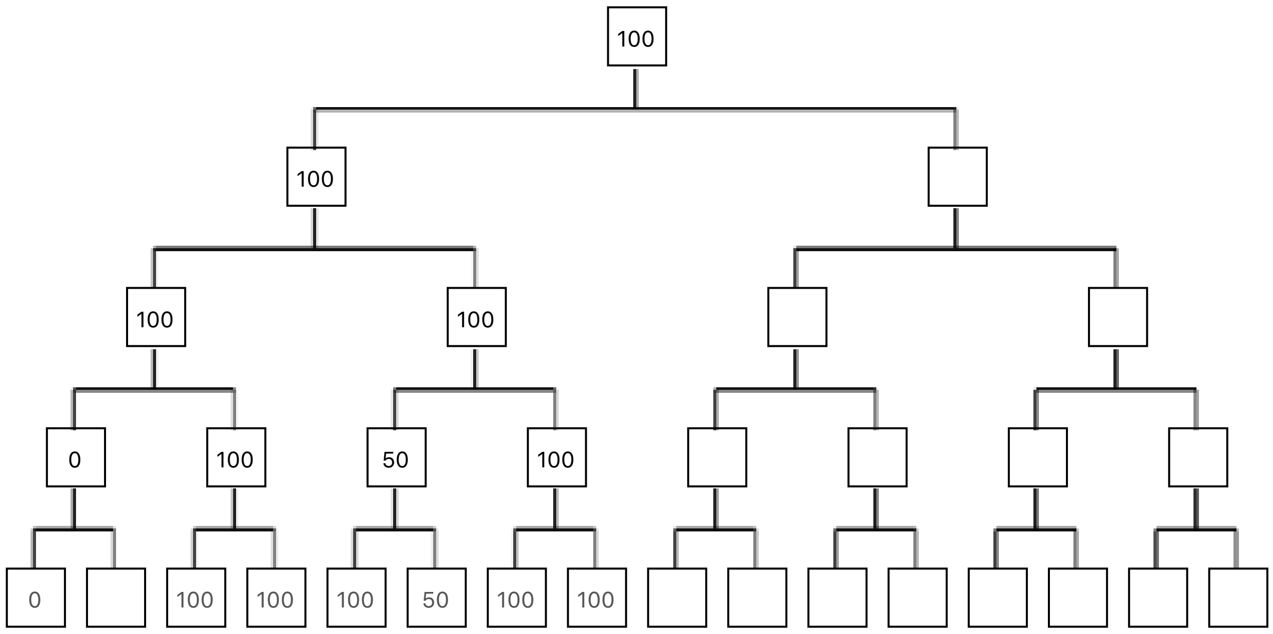


Figure 6.3 - A minimax tree with values computed by Bounded Minimax

It is easy to adapt the basic Minimax code to do Bounded Minimax. All we need to do is to put conditionals in the inner loops of `bestmove` and `maxscore` and `minscore`, as shown below.

```
function bestmove (role,state)
 {var actions = findlegals(role,state,game);
  var action = actions[0];
  var score = 0;
  for (var i=0; i<actions.length; i++)
      {var result = minscore(role,actions[i],state);
       if (result==100) {return actions[i]};
```

```
          if (result>score) {score = result; action = actions[i]}};
      return action}

    function minscore (role,action,state)
     {var opponent = findopponent(role,game)
      var actions = findlegals(opponent,state,game);
      var score = 100;
      for (var i=0; i<actions.length; i++)
          {var move;
           if (role==roles[0]) {move = [action,actions[i]]}
              else {move = [actions[i],action]}
           var newstate = findnext(move,state,game);
           var result = maxscore(role,newstate);
           if (result==0) {return 0};
           if (result<score) {score = result}};
      return score}

    function maxscore (role,state)
     {if (findterminalp(state,game)) {return findreward(role,state,game)};
      var actions = findlegals(role,state,game);
      var score = 0;
      for (var i=0; i<actions.length; i++)
          {var result = minscore(role,actions[i],state);
           if (result==100) {return 100};
           if (result>score) {score = result}};
      return score}
```

Note that 100 and 0 are not the only values that can be used here. For example, if a player is in a satisficing game, where it needs to get a certain minimum score, then it can use that threshold rather than 100. If a player simply wants to win a fixed sum game, then it can use 51 as the threshold, knowing that if it gets this amount it has won the game.

6.4 Alpha-Beta Search

While Bounded Minimax helps avoid some wasted work, we can do even better. Consider the game tree shown below.

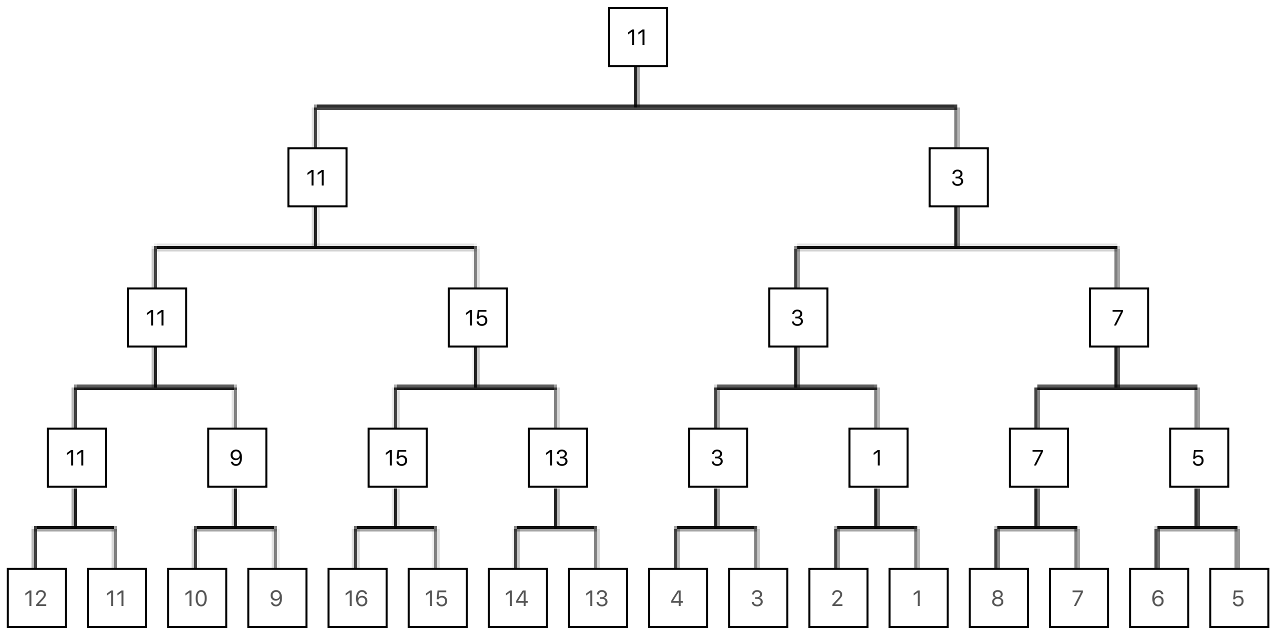


Figure 6.4 - A sample minimax tree

In this case, unlike the examples seen earlier, there are many terminal values that are not 0 or 100. In determining its maximum score for the top node of this tree, a Minimax player, even a Bounded Minimax player, would examine the entire tree. However, not all of this work is necessary.

Alpha-Beta Search is a variation on Bounded Minimax that eliminates such wasted work by computing bounds dynamically and passing them along as parameters. One bound, called alpha, is the best score the player has seen thus far. The other bound, called beta, is the worst score the player has seen. In examining new nodes, alpha-beta search uses these bounds to decide whether to look at further nodes.

If the partial result at a min node is less than alpha, then there is no point in examining other descendants of that node since it could only decrease this value and the player would not take that choice given that it has a higher value elsewhere.

Analogously, if the partial result at a max node is greater than beta, then there is no point in considering other options since they can only increase the score and the player's opponents would not allow that since they know they can keep the value to no more than beta.

The following is an implementation of `maxscore` and `minscore` for an alpha-beta player.

```
function maxscore (role,state,alpha,beta)
 {if (findterminalp(state,game)) {return findreward(role,state,game)};
  var actions = findlegals(role,state,game);
  for (var i=0; i<actions.length; i++)
      {var result = minscore(role,actions[i],state,alpha,beta);
       alpha = max(alpha,result);
       if (alpha>=beta) then {return beta}};
  return alpha}

function minscore (role,action,state,alpha,beta)
 {var opponent = findopponent(role,game);
  var actions = findlegals(opponent,state,game);
  for (var i=0; i<actions.length; i++)
      {var move;
       if (role==roles[0]) {move = [action,actions[i]]}
          else {move = [actions[i],action]}
       var newstate = findnext(move,state,game);
       var result = maxscore(role,newstate,alpha,beta);
       beta = min(beta,result);
       if (beta<=alpha) then {return alpha}};
   return beta}
```

Now let's apply the `maxscore` procedure to the tree shown above with initial value 0 and 100 for alpha and beta. In the tree below, we have written in values produced by the alpha-beta procedure in this case, and we have left the other nodes blank.

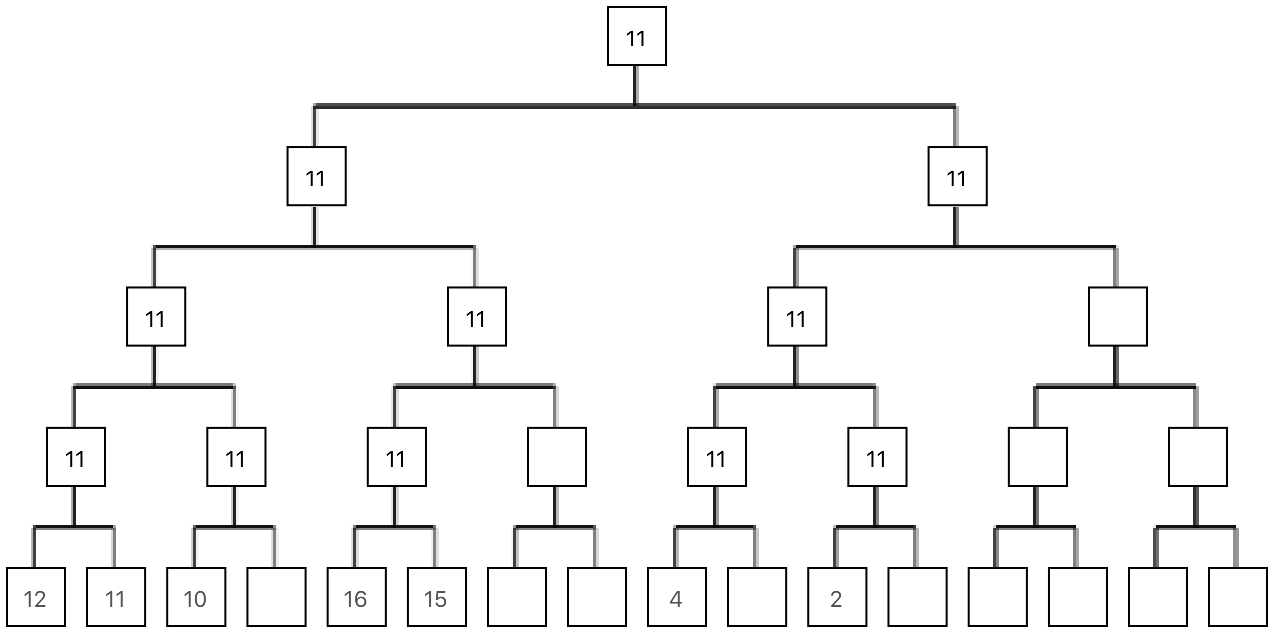
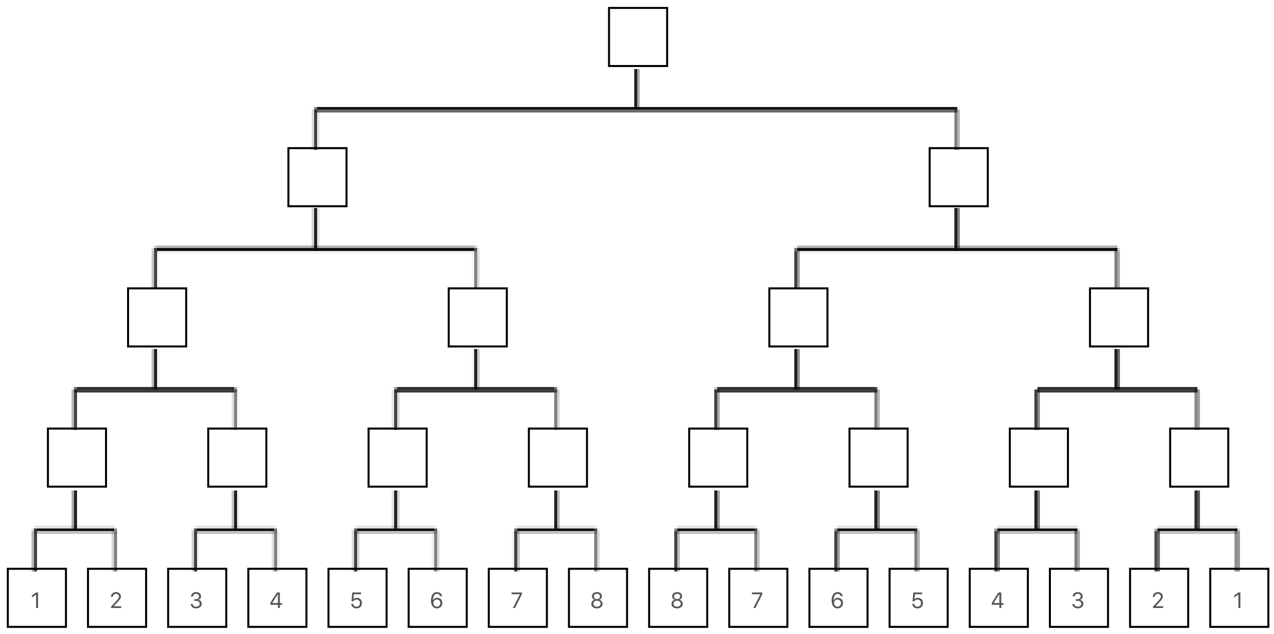


Figure 6.5 - A minimax tree with values computed by Alpha-Beta Search

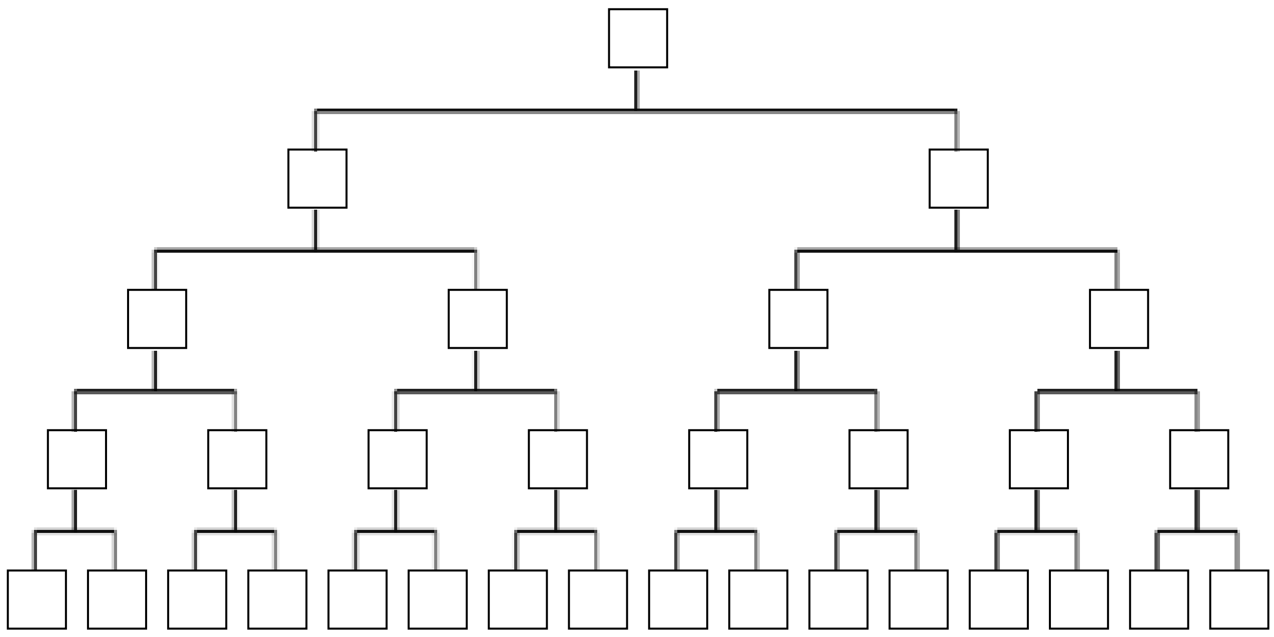
In this particular case, the improvement of Alpha-Beta over Minimax is modest. However, in general, Alpha-Beta Search can save a significant amount of work over full Minimax. In the best case, given a tree with branching factor b and depth d, Alpha-Beta Search needs to examine at most $O(b^{d/2})$ nodes to find the maximum score instead of $O(b^d)$. This means that an Alpha-Beta player can look ahead twice as far as a Minimax player in the same amount of time. Looked at another way, the effective branching factor of a game in this case is *sqrt*(b) instead of b. It would be the equivalent of searching a tree with just 5 moves instead 25 moves.

Exercises

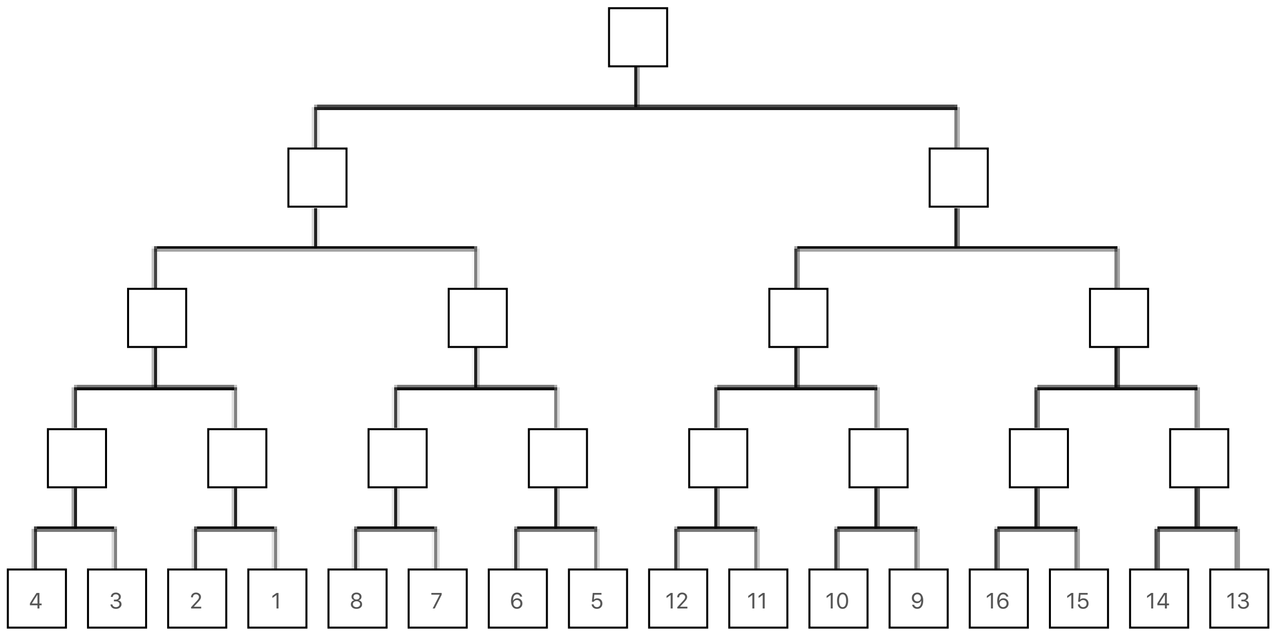
Exercise 6.1: Fill in the minimax values for the non-terminal nodes in the following game tree. Max nodes are initially beige; min nodes are grey.



Exercise 6.2: Assign the utility values 1, 2, ..., 16 to the 16 terminal nodes in the following game tree so that (1) no two terminal nodes have the same value and (2) the minimax value of the top node is 8.



Exercise 6.3: Fill in the alpha-beta values for the non-terminal nodes in the following game tree. Put an X in any node that alpha-beta does not examine.


4
3
2
1
8
7
6
5
12
11
10
9
16
15
14
13