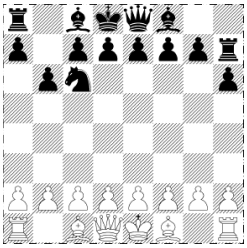
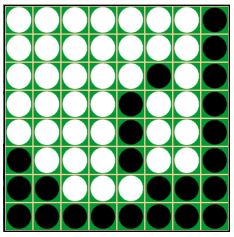
General Game Playing

Small Multiple Player Games

Multiple Player Games

Example:


More complicated that single-player games

Changes depend on actions of others

and those actions cannot be controlled

So player must consider all possible actions of others

Having discussed single-player games, we turn now to multiple player games, such as Chess and Othello and Chinese Checkers. In most cases, the other players in these games 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.

Fixed Sum and Variable Sum Games

Fixed Sum Games

- Total reward in all states is the same
- For one player to get more, the others must get less

- Also called zero sum in cases where sum is 0
- Note that, in GGP, all rewards are non-negative.
- However, can be scaled - to 50, for example.

- Many common games are zero sum
- e.g. Chess - winner and loser

Variable Sum Games

- Possible for one player to get more without other players getting less
- Some games are even cooperative

Before proceeding, it is worth emphasizing that multiple player 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.

Small Games

Resources

Sufficient time and space
to search the entire game tree

Results

Players can find optimal *strategy*
Not necessarily *sequential*, as with single player games
Plans may be *conditional* on the actions others
NB: *Sometimes* possible without searching entire tree

Small size rules out games like Chess and Othello, which require more complicated techniques, some of which are discussed in subsequent lessons.

In this lesson, as in the preceding lesson, 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.

Programme

Approaches covered

- Minimax and Bounded Minimax
- Alpha-Beta Search

Interesting Approach not covered:

- Conditional Planning

We begin this lesson with a procedure called Minimax and 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. There is also an analog to sequential planning, called conditional planning. However, it is a little complicated and is not used all that often; so we bypass that for now.


GENERAL
GAME
PLAYING


GENERAL
GAME
PLAYING

Minimax

Opponent

Opponent Modeling
- May assume opponent will play rationally
  but
- Assumption may be wrong

- Also, players do not know identities of other players

Common Alternative - pessimistic / conservative
- Assume the other player will do the worst possible thing

In general game playing, a player may choose to make assumptions about the actions of the other players. For example, a player might 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.

Basic Idea

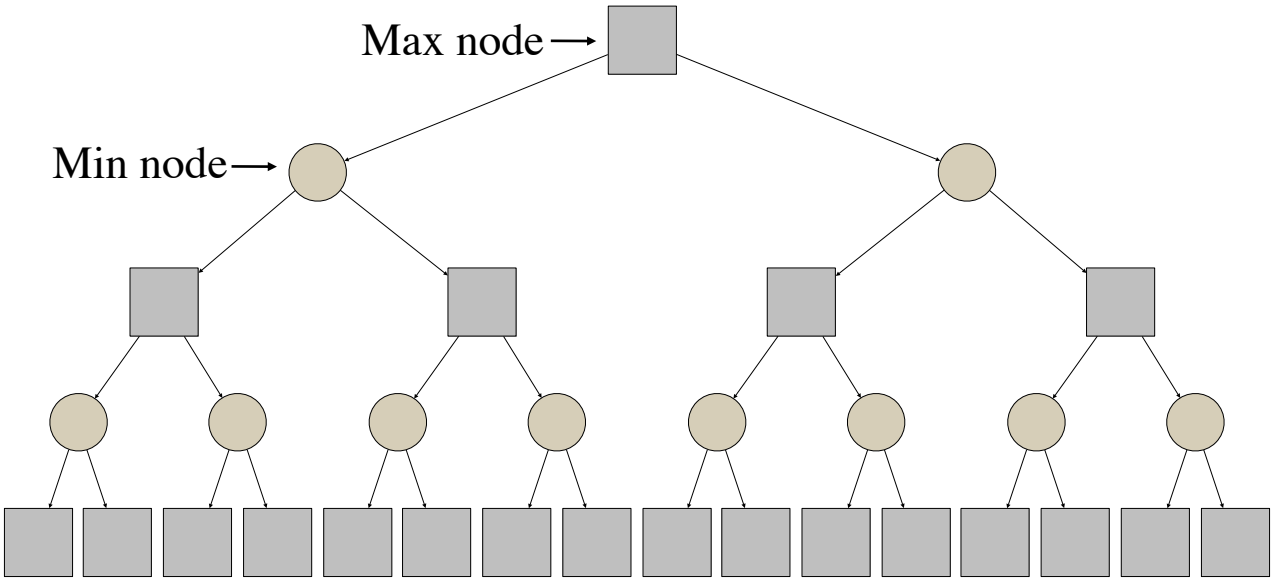
Intuition - Select a move that is guaranteed to produce the highest possible return no matter what the opponents do.

In the case of a one move game, a player should choose 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 and opponent action.

In the case of a multi-move game, minimax goes to the end of the game and "backs up" values.

The basic idea of Minimax is to select moves that are guaranteed to produce the highest possible return no matter what the opponents do. The player tries to maximize its own value and assumes that the opponents are trying to minimize its value. Hence the name Minimax.

Bipartite Game Tree



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 multi-step game, Minimax goes to the end of the game and backs up value. In general, we can think about Minimax as search of a bipartite tree consisting of alternating max nodes (shown here as grey squares) and min nodes (shown here as beige circles). The max nodes represent the choices of the player while the min nodes represent the choices of the other players.

Note that, in the case of games with more than two players, there can be multiple layers of min nodes between each layer of max node, one layer for each opponent. Note also 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.

State Value

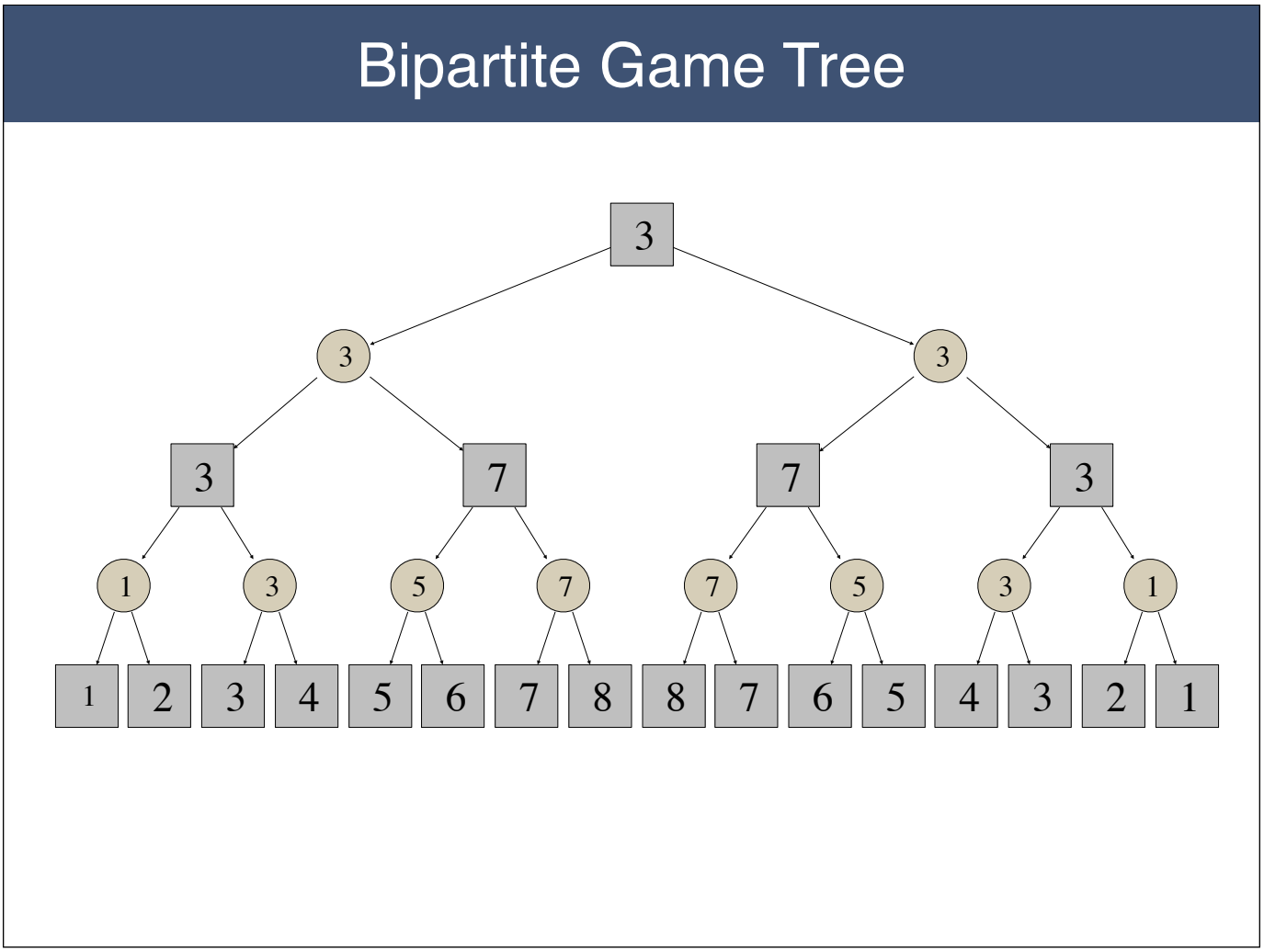
The *value* of a *max* node for player *p* is either the utility of that state if it is terminal or the maximum of all *values* for the *min* nodes that result from its legal actions.

$$value(p,x) = goal(p,x) \text{ if } terminal(x)$$
$$max(\{value(p,minnode(p,a,x)) \mid legal(p,a,x)\})$$

The *value* of a *min* node is the minimum value that results from any legal opponent action.

$$value(p,n) = min(\{value(p,maxnode(P\text{-}p,b,n)) \mid legal(P\text{-}p,b,n)\})$$

The value of a max node for a player is the utility of the corresponding state if the state is terminal. Otherwise, it is the maximum of the 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 the values of its maxnodes below it, viz. 1 and 2. The value of the minnode next to that minnode is 3 because that is the value 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.

Implementation

```
var ruleset;
var role;
var roles;
var state;

function info (id)
  {return 'ready'}

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

function play (id,actions)
 {var move = doesify(roles,actions);
  if (move!=‘nil’) {state=findnexts(move,state,ruleset)};
  return undoesify(bestmove(state))}

function stop (id,move)
  {return 'done'}

function abort (id)
  {return 'done'}
```

Here is an implementation of a minimax player. This is identical to the implementation of compulsive deliberation for single player games *except* that it has a different bestmove procedure.

bestmove

```
function bestmove (state)
 {var actions = findlegals(role,state,ruleset);
  var action = actions[0];
  var score = 0;
  for (var i=0; i<actions.length; i++)
      {var result = minscore(actions[i],state);
       if (result>score)
          {score = result; action = actions[i]}};
  return action[2]}
```


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.

minscore and maxscore

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


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

The minscore subroutine for minimax takes an action and a state as arguments and produces the minimum values for the given role associated with the given player action for any of the opponent's legal actions in the given state. The maxscore subroutine, which is called by minscore, takes a state as argument and 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.

Bounded Minimax

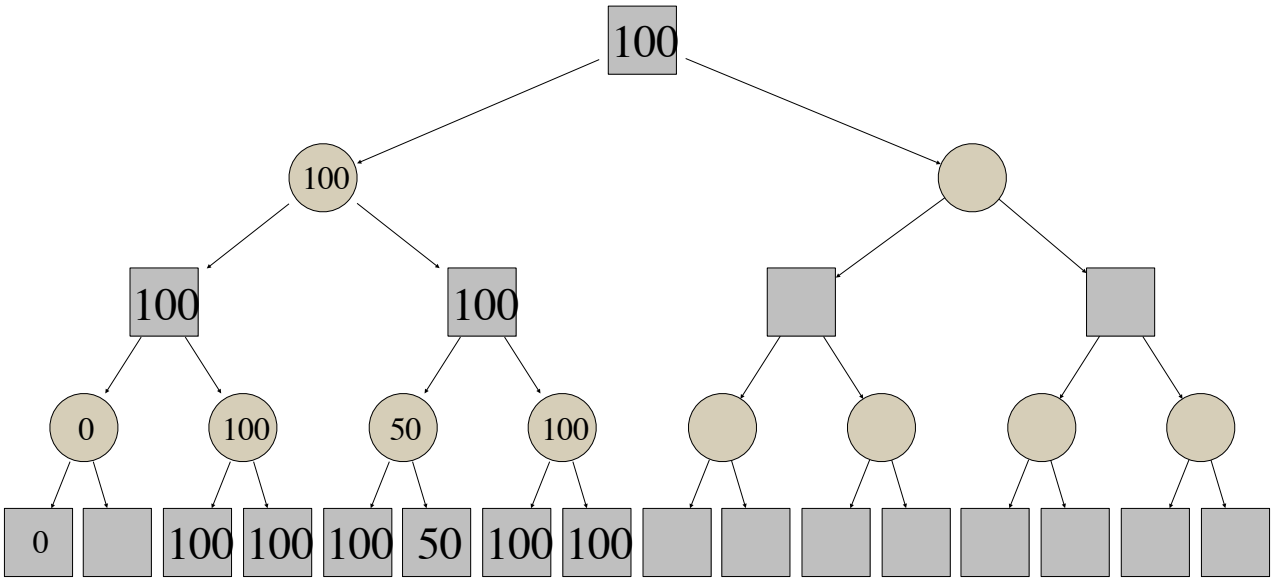
If the minvalue for an action is determined to be 100, then there is no need to consider other actions.

In computing the minvalue for a state if the value to the first player of an opponent's action is 0, then there is no need to consider other possibilities.

One disadvantage of the Minimax procedure 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 with two minor changes. Rather than processing all actions on every node, maxscore and minscore checks first for these bounds; and, if they occur on any node, they terminate their search and return the corresponding values.

Bounded Minimax Example



Here is an example. The nodes in this tree with values are those examined by Bounded Minimax. The other nodes are not examined at all and do not need to be examined. In this case, more than half of the tree is pruned from consideration.

Generalization

As stated

100 is the limiting case for maxscore

0 is the limiting case for minscore

Other possibilities

Satsificing - fixed minimal score all that is needed

Fixed sum game - 51 is sufficient

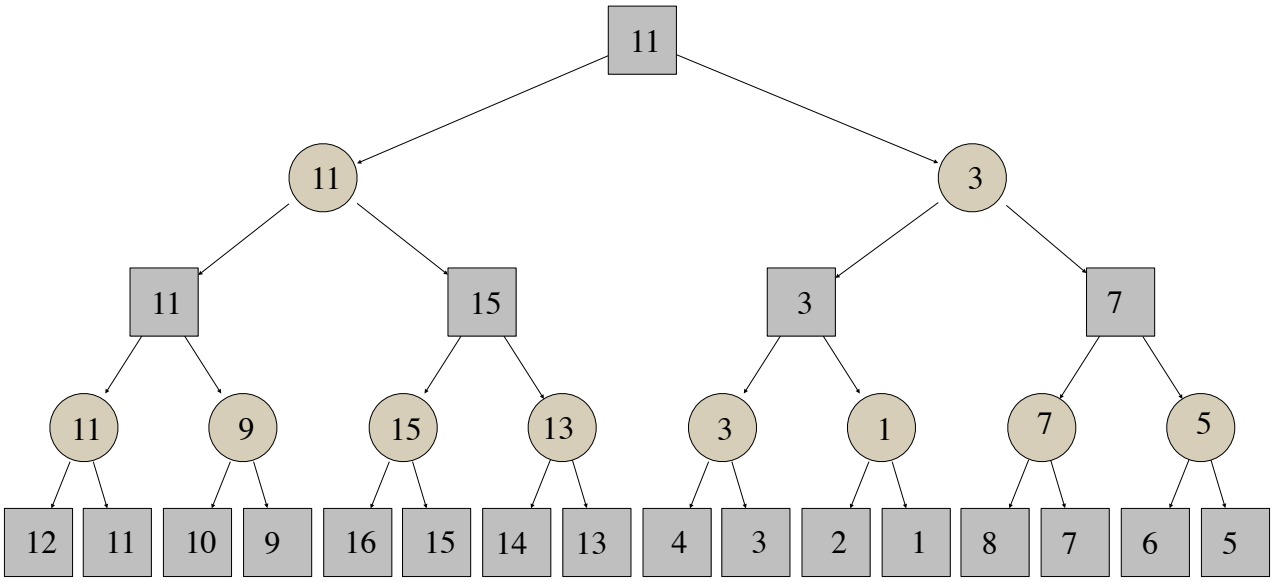
Note that 100 and 0 are not the only values that can be used here. For example, if a player is in a satisficing game, where it just needs to get a certain minimum score, then it can use that threshold rather than 100. If a player simply wants to win a fixed sum game, then it can use 51 as the threshold,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


GENERAL
GAME
PLAYING


GENERAL
GAME
PLAYING

Alpha Beta Pruning

Alpha-Beta Example



24

While Bounded Minimax helps avoid some wasted work, we can do even better. Consider the game tree shown here. In this case, unlike the earlier examples, 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. As an example, consider the fourth terminal node on the lower left. Even before that node is examined, the player knows that its opponent can keep it to at most 10 points on that branch (based on the third node), and it already has a move of its own that gets it 11 points. Exploring the fourth node can only decrease the min node's score, and the player is not going to choose it anyway, so there is no point in examining it. In this case, the saving is only one node; in other cases, it can allow a player to prune whole subtrees, as we shall see in just a bit.

Alpha-Beta Search

Alpha-Beta Search - Same as Bounded Minimax except that bounds are computed dynamically and passed along as parameters.

If partial result of min node less than alpha, can only decrease score and player need not consider.

If partial result of max node greater than beta, can only increase score and opponent will not allow.

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.

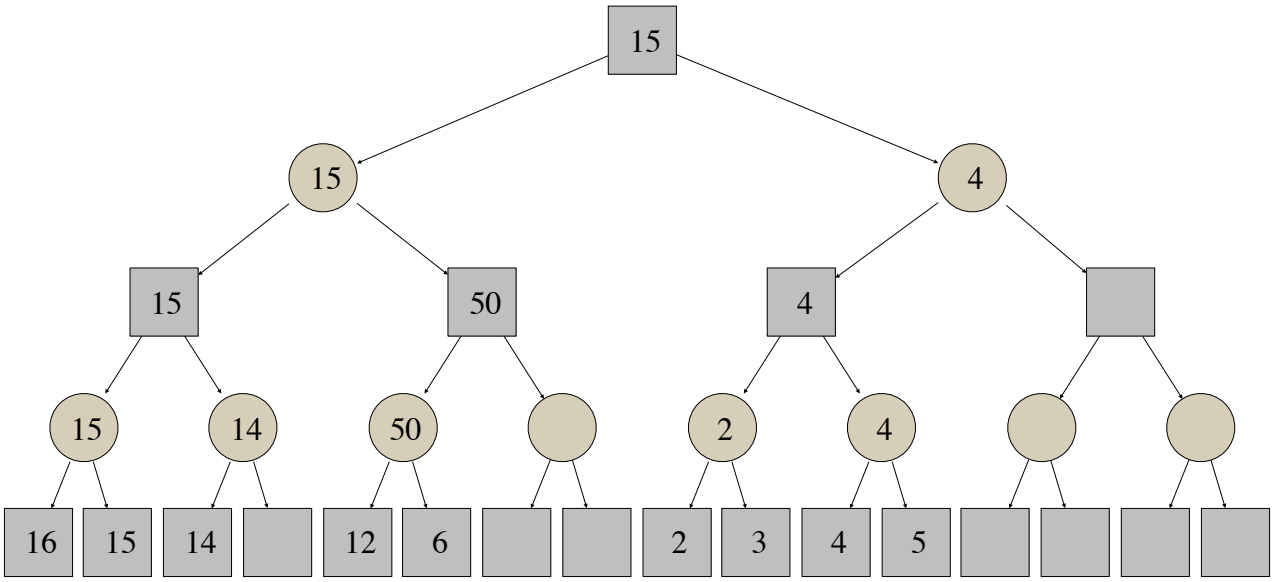
maxscore and minscore

```
function maxscore (state,alpha,beta)
  {if (findterminalp(state,ruleset))
      {return reward(role,state,ruleset)};
   for (var action in findlegals(role,state,ruleset))
       {var minval = minscore(action,state,alpha,beta);
        alpha = max(alpha,minval);
        if (alpha>=beta) then {return beta}};
   return alpha}


function minscore (action,state,alpha,beta)
  {var actions = findlegals(opponent,state,ruleset);
   for (i=0; i<actions.length; i++)
       {var move = [action, actions[i]];
        var newstate = findnexts(move,state,ruleset);
        var maxval = maxscore(newstate,alpha,beta);
        beta = min(beta,maxval);
        if (beta<=alpha) then {return alpha}};
   return beta}
```

Here is an implementation of maxscore and minscore for an alpha-beta player. In computing the maxscore of a max node, the player takes the max of alpha and the minscore of the node obtained by performing any legal action in the corresponding state. To compute the minscore of a node, the player takes the minimum of beta and the maxscore of the node for the state obtained by executing the joint move for any legal action.

Alpha-Beta Example



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 shown here, we have written in values produced by the alpha-beta procedure in this case, and we have left the other nodes blank. As you can see there is substantial saving.

Benefits

Best case of alpha-beta pruning can reduce search space to square root of the unpruned search space, thereby dramatically increasing depth searchable within given time bound.

For example, it could in some cases reduce a tree with branching factor of 25 to branching factor of 5.

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.


GENERAL
GAME
PLAYING


GENERAL
GAME
PLAYING

Caching

Virtual Game Graph

The game graph is virtual - it is computed on the fly from the game description. Hence, our subroutines are not cheap.

May be good to cache results (build an explicit game graph) rather than call repeatedly.

Benefit - Avoiding Duplicate Work

Consider a game in which the actions allow one to explore an 8x8 board by making moves left, right, up, and down.

Game tree has branching factor 4 and depth 16. Fringe of tree has 4^16 = 4,294,967,296 nodes.

Only 64 states. Space searched much faster if duplicate states detected.

Disadvantage - Extra Work

Finding duplicate states takes time proportional to the log of the number of states and takes space linear in the number of states.

Work wasted when duplicate states rare. (We shall see two common examples of this next week.)

Memory might be exhausted. Need to throw away old states.


GENERAL
GAME
PLAYING


Celebrating AAAI's 25th
Anniversary