General Game Playing

Heuristic Search

Search

Complete Search
- guarantees optimal play
- works only for games with small game trees

Incomplete search
- Depth-Limited Search
- Fixed-Depth Heuristic Search
- Variable-Depth Heuristic Search
- Statistical Methods (next lesson)

In the last two lessons, we looked at approaches to playing *small* games, i.e. games for which there is sufficient time for a complete search of the game tree. Unfortunately, most games are not so small, and complete search is usually impractical. In this lesson, we look at a variety of techniques for incomplete search. We begin with simple Depth-Limited Search. After that, we turn to two variations, first fixed-depth heuristic search and then variable depth heuristic search. In the next lesson, we examine statistical methods for dealing with incomplete search.

Depth-Limited Search

Idea - search tree to some depth-limit
for terminal nodes, return goal values
for non-terminal nodes, return 0

Legal and random players are degenerate depth-limited search procedures with depth 0.

The simplest way of dealing with games for which there is insufficient time to search the entire game tree is to limit the search in some way. In depth-limited search, the player explores the game tree to a given depth. A legal player is a special case of depth-limited search where the depth is effectively zero.

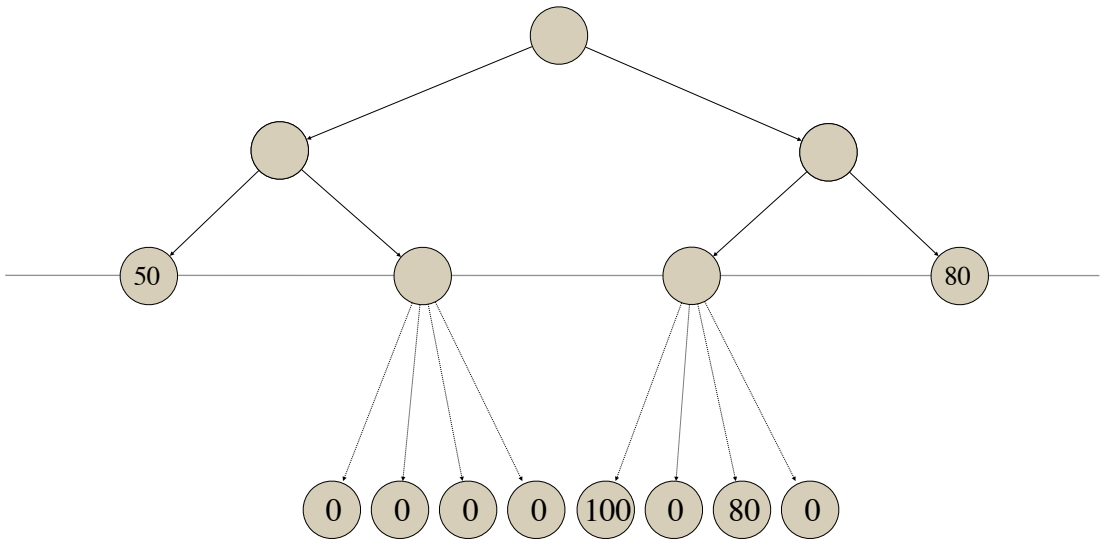
Implementation

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

function minscore (action,state,level)
 {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,level+1);
       if (result<score) {score = result}};
  return score}
```

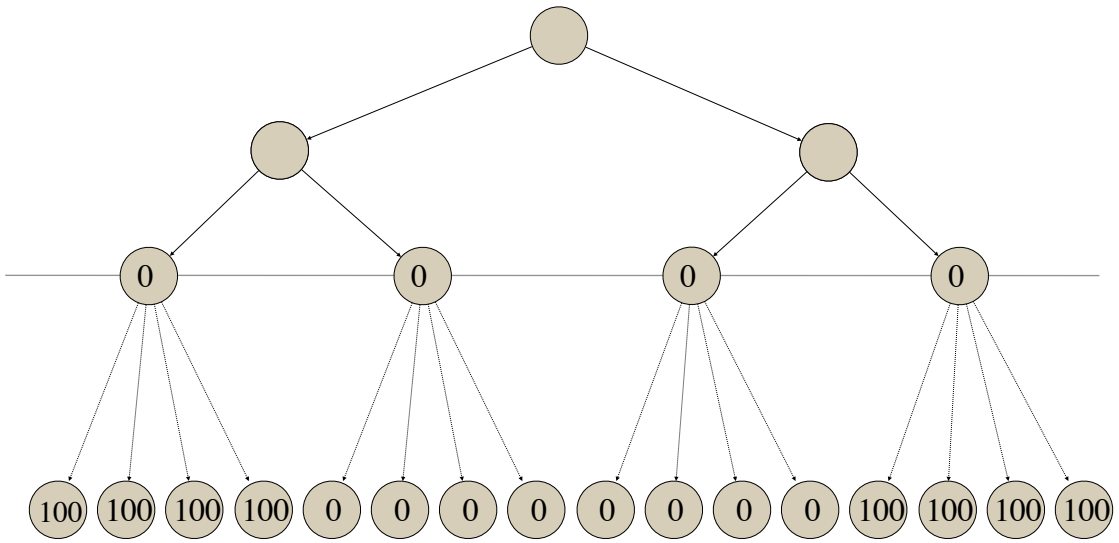
The implementation of depth-limited search is a small variation of the implementation of the minimax player described in the preceding lesson. The only difference is the addition of a level parameter to maxscore and minscore. This parameter is incremented in minscore on each recursive call in maxscore, as we see here at the botton. If the level of any particular non-terminal node in the game tree exceeds a pre-determined depth limit, rather than expanding, maxscore simply returns 0, a conservative lower bound on the utility of the corresponding state.

Example



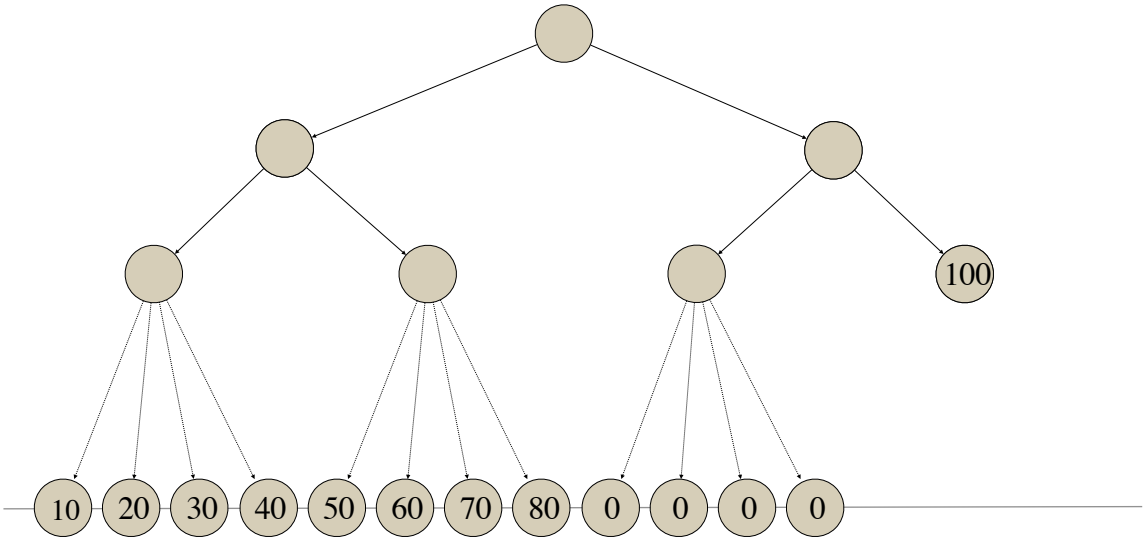
Here is an example of Depth-Limited Search. By limiting the depth, the player does not need to explore the entire tree. Yet it is still able to find a solution that nets it 80 points. Not as good as 100 points but better than 50 and a lot better than 0.

Problem - Insufficient Depth



The most obvious problem with depth-limited search is that the conservative estimate of 0 for non-terminal states is not very informative. In the worst case, none of the states at a given depth may be terminal, in which case the search provides no real value. We discuss some ways of dealing with this problem in the next segment and the next lesson.

Problem - Excessive Depth



There is also the opposite problem. The depth may be set at too great a level, forcing the player to search deep into the tree before finding an answer at a shallow level. This problem is serious if along the way the player runs out of time before encountering the shallow solution.

Breadth-First Search

Explore all nodes to level 1
Explore all nodes at level 2
Explore all nodes at level 3
And so forth

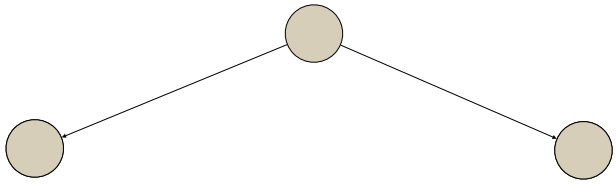
While time permits
Choose action that gives maximal value

One solution to this problem is to use breadth-first search rather than depth-first search.

Breadth-First Search

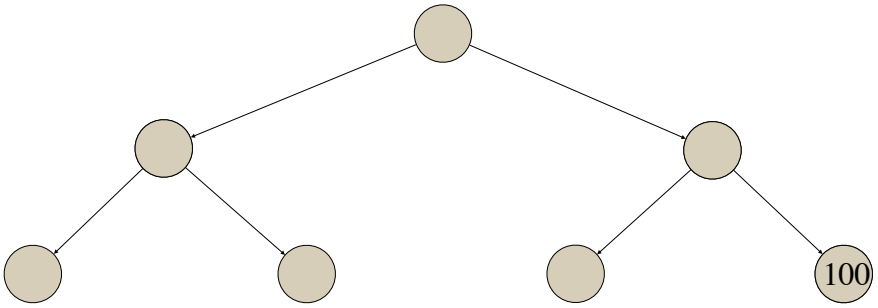
We start off at the root.

Breadth-First Search



Expand one level, checking each node for termination.

Breadth-First Search



Halt when encounter a terminal state with a sufficiently large value. If time expires before this happens, we simply choose a branch that leads to node with highest terminal value. Breadth-first search has the merit that it finds the shortest path to a maximal goal. It has the disadvantage that it consumes space that is proportional to the size of the expanded game tree, which can be exponential in depth. Using this approach, in some cases, a player can run out of memory and crash.

Iterative Deepening

Use depth-limited search to explore entire tree to level 1
Use depth-limited search to explore entire tree to level 2
Use depth-limited search to explore entire tree to level 3
And so forth

While time permits
Choose action that gives maximal value

An alternative solution to this problem is to use an iterative deepening approach to game tree exploration, exploring the entire game tree repeatedly at increasing depths until time runs out. We search the tree to depth 1, then we search the entire tree to depth 2, then to depth 3, and so forth, using depth-limited search on each iteration.

Advantages and Disadvantages

Advantages

requires storage linear to depth rather than exponential

still finds shortest path to an optimal solution

Disadvantages (?)

Repeated work

but

Cost only a constant factor more than depth-first search

Why? Tree is growing exponentially, so fringe of tree and size of tree above fringe are approximately same

The primary advantage of this approach is that it requires space that is linear in the depth of the explored portion of the tree. The entire subtree does not need to be stored. Yet it still finds the shortest path to an optimal solution.

The downside is that portions of the tree may be explored multiple times. However, as usual with iterative deepening, this waste is usually bounded by a small constant factor. Why? The size of the fringe of the tree is approximately the same as the size of the tree above the fringe. So searching it each additional time requires only the same amount of work as exploring the fringe at the next level. This means extra work but it is bounded by a constant factor.

Heuristic Search

Heuristic Search

Alternative is to arbitrary 0 value for non-terminal states is to apply a heuristic *evaluation function* to fringe states (whether terminal or non-terminal).

One way of dealing with the conservative nature of depth-limited search is to improve upon the arbitrary 0 value returned for nonterminal states. In heuristic search, this is accomplished by applying a heuristic evaluation function to non-terminal states. Such functions are based on features of states, and so they can be computed without examining entire game tree.

Implementation

Depth-Limited Search:

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

Heuristic Depth-Limited Search

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

The implementation of Fixed-depth Heuristic Search is easy. At the top here, we have the implementation of depth-limited search that we saw earlier. At the bottom, we have an implementation of Heuristic fixed-depth search. The only difference is that we have replaced the default value of 0 with the result of calling the evaluation subroutine evalfun on the state being considered whenever the depth limit is exceeded. The tough part of the implementation is figuring out how to evaluate non-terminal states.

Evaluation Functions

Chess examples:

- Piece count
- Board control

Comments

- Not necessarily successful
- Usually work on specific games
- However, there are some interesting general heuristics

Fortunately, examples of heuristic functions abound. For example, in chess, we often use piece count to compare states, with the idea that, in the absence of immediate threats, having more material is better than having less material. Similarly, we sometimes use board control, with the idea that having control of the center of the board is more valuable than controlling the edges or corners.

The downside of using heuristic functions is that they are not necessarily guaranteed to be successful. They may work in many cases but they can occasionally fail, as happens, for example in chess, when a player is checkmated even though he has more material and a better board control. Still, games often admit heuristics that are useful in the sense that they work more often than not.

While, for specific games, such as chess, programmers are able to build in evaluation functions in advance, this is unfortunately not possible for general game playing, since the rules of the game are not known in advance. Rather, the game *player* must analyze the game itself in order to find a useful evaluation function. In an upcoming lesson, we discuss how to find such heuristics.

That said, there are some heuristics for game playing that have arguable merit across all games. In this section, we examine some of these heuristics. We also show how to build game players that utilize these general heuristics.

Example - Mobility

Mobility is a measure of the number of things a player can do.

Basis - number of actions in a state or number of states reachable from that state.

Horizon - current state or *n* moves away.

Mobility is one such heuristic. Mobility is a measure of the number of things a player can do. This could be the number of actions available in the given state or n steps away from the given state. Or it could be the number of states reachable within n steps. (This could be different from the number of actions since different action sequences can lead to the same state.)

Implementation

```
function mobility (state)
 {var actions = findlegals(role,state,ruleset);
  var feasibles = findinputs(role,ruleset);
  return (actions.length/feasibles.length * 100)}
```

A simple implementation of the mobility heuristic is shown here. The method simply computes the number of actions that are legal in the given state and returns as value the percentage of all possible actions represented by this set of legal actions.

Example - Focus

Focus is a measure of the narrowness of the search space. It is the inverse of mobility.

Sometimes it is good to focus to cut down on search space.

Often better to restrict opponents' moves while keeping one's own options open.

Focus is another heuristic. Focus is a measure of the narrowness of the search space. It is the inverse of mobility. Sometimes it is good to focus to cut down on search space. Often, it is better to restrict opponents' moves while keeping one's own options open.

Implementation

```
function focus (state)
 {var actions = findlegals(role,state,ruleset);
  var feasibles = findinputs(role,ruleset);
  return (100 - actions.length/feasibles.length * 100)}
```

A simple implementation of the focus heuristic is shown here. It is the dual of mobility. Again we divide the number of legal actions by the number of all possible actions; but, in this case instead of returning that value, we return the result of subtracting it from 100.

Goal Proximity

Goal proximity is a measure of the proximity of a state to a goal state.

Number of propositions shared between state and some goal state. Problem - difficult to find / enumerate terminal states.

Use goal value of non-terminal state as indicator of proximity to terminal state with high goal value. Problem - there may be no correlation (e.g. chess checkmates). Good news: Sometimes possible to find and prove a correlation.

Goal proximity is a measure of how similar a given state is to desirable terminal state. There are various ways this can be computed.
One common method is to count how many propositions are true in the current state are also true in a terminal state with adequate utility. The difficulty of implementing this method is obtaining a set of desirable terminal states with which the current state can be compared.
Another alternative is to use the goal value of a state as a measure of progress toward the goal, with the idea being that the goal value of a non-terminal state, the closer the actual goal. Of course, this is not always true. However, in many games the goal values are indeed *monotonic*, meaning that values do increase with proximity to the goal. Moreover, it is sometimes possible to compute this by a simple examination of the game description, using methods we describe in later lessons.

Weighted Linear Combinations

Definition

$$f(s) = w_1 \times f_1(s) + \ldots + w_n \times f_n(s)$$

Examples:

- Mobility
- Focus
- Goal proximity

Many players estimate weights by experimentation during the start clock.

None of these heuristics is guaranteed to work in all games, but all have strengths in some games. To deal with this fact, some designers of GGP players have opted to use a weighted combination of heuristics in place of a single heuristic. The equation shown here is a typical formula. Each fi here is a heuristic function (such as mobility or focus or goal proximity), and wi is the corresponding weight.
Of course, there is no way of knowing in advance what the weights should be, but sometimes playing a few instances of a game (e.g. during the start clock) can suggest weights for the various heuristics.

Horizon Problem

Horizon Problem

- white gains a rook but loses queen or loses game
- example - sequence of captures in chess

As mentioned earlier, depth-limited search is not guaranteed to succeed in all cases. Failing is never good. However, it is particularly embarrassing in situations where just a little more search would have revealed significant changes in the player's circumstances, for better or worse. In the research literature, this is often called a horizon problem. As an example of a horizon problem in Chess, consider a situation where the players are exchanging piece, with white capturing black's pieces and vice versa. Now imagine cutting off the search at an arbitrary depth, say 2 captures each. At this point, white might believe it has an advantage since it has more material. However, if the very next move by black is a capture of the white queen, this evaluation could be misleading.

Variable Depth Search

Idea - use e*xpansion* function in place of fixed depth as a termination criterion.

A common solution to this problem is to forego the fixed depth limit in favor of one that is itself dependent on the current state of affairs, searching deeper in some areas of the tree and searching less deep in other areas.

Variable-Depth Heuristic Search

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

Here is an implementation of variable depth heuristic search. This version of maxscore differs from the fixed-depth version in that there is a subroutine (here called expfn) that is called to determine whether the current state and/or depth meets an appropriate condition. If so, the tree expansion continues; otherwise, the player terminates the expansion and simply returns the result of applying its evaluation function to the non-terminal state.

Expansion Functions

Example - *Quiescence*

Evaluation function values changing slowly

The challenge in variable-depth heuristic search is finding an appropriate definition for expfn. One common technique is to focus on differentials of heuristic functions. For example, a significant change in mobility or goal proximity might indicate that further search is warranted whereas actions that do not lead to dramatic changes might be less important. In Chess, a good example of this is to look for quiescence, i.e. a state in which there are no immediate captures.


GENERAL
GAME
PLAYING


Celebrating AAAI's 25th
Anniversary