CHAPTER 7

Heuristic Search

7.1 Introduction

In the last two chapters, 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 chapter, we look at a variety of techniques for incomplete search. We begin with Depth-Limited Search, then Fixed-Depth Heuristic Search, and finally Variable-Depth Heuristic Search. In the next chapter, we examine probabilistic methods for dealing with incomplete search.

7.2 Depth-Limited Search

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.

The implementation of Depth-Limited Search is a simple variation of the implementation of the minimax player described in the preceding chapter. See below. One difference is the addition of a level parameter to `maxscore` and `minscore`. This parameter is incremented on each recursive call in `minscore`. If the player reaches the depth-limit at a non-terminal state, it simply returns 0, a conservative lower bound on its utility.

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

function minscore (role,action,state,level)
 {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,level+1);
       if (result==0) {return 0};
       if (result<score) {score = result}};
  return score}
```

The most obvious problem with Depth-Limited Search is that the conservative estimate of utility 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 discriminatory value. We discuss some ways of dealing with this problem in the next sections and the next chapter.

Another problem with Depth-Limited Search is that a player may not be able to determine a suitable depth-limit in advance. Too low and the player will not search as much as it could. Too high and the search may not terminate in time.

One solution to this problem is to use breadth-first search rather than depth-first search. The downside of this is the amount of space that this consumes on very large trees, in many cases exceeding the storage limits of the computer.

An alternative solution to this problem is to use an iterative deepening approach to game tree exploration, exploring the game tree repeatedly at increasing depths until time runs out. As usual with iterative deepening, this is wasteful in 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.

7.3 Fixed-Depth Heuristic Search

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 fixed-depth 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.

Examples of such 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 it 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 structure of the game is not known in advance. Rather, the game player must analyze the game itself in order to find a useful evaluation function. In a later chapter, 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.

The implementation of a general game player based on fixed-depth heuristic search is a simple variation of the fixed-depth search player just described. See below. The difference comes in the `maxscore` procedure. Rather than returning 0 on non-terminal states, the procedure returns the value of a subroutine `evalfn`, which gives a heuristic value for the state.

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

There are various ways of defining `evalfn`. In the following paragraphs, we look at just a few of these - mobility, focus, and goal proximity.

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 multiple action sequences could lead to the same state. All roads lead to Rome.)

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

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

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.

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

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 that 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 utility of the given state as a measure of progress toward the goal, with the idea being that the higher utility, the closer the goal. Of course, this is not always true. However, in many games the goal values are indeed *monotonoic*, 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 chapters.

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. See the formula below. Here each f_i is a heuristic function, and w_i is the corresponding weight.

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

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.

7.4 Variable Depth Heuristic Search

As we discussed in the preceding section, heuristic 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, for example white capturing two pawns and black capturing a pawn and a bishop. 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.

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.

In Chess, a good example of this is to look for quiescence, i.e. a state in which there are no immediate captures.

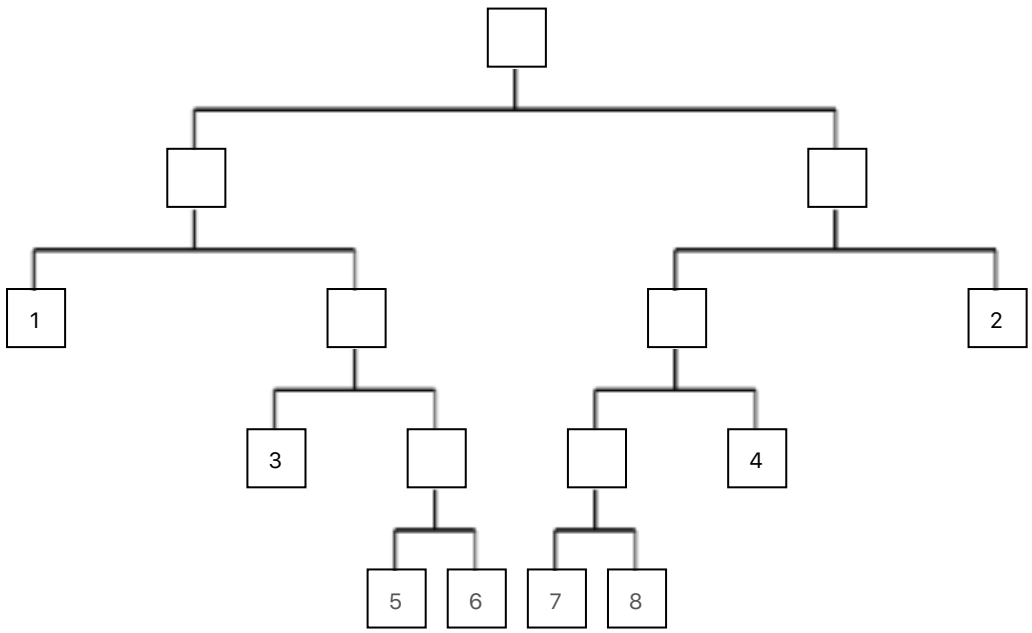
The following is an implementation of a variable-depth heuristic. This version of `maxscore` differs from the fixed-depth version in that there is a subroutine (here called `expfn`) that is called to determine whether the current state and/or depth meets an appropriate condition. If so, the tree expansion terminates; otherwise, the player expands the state.

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

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.

Exercises

Exercise 7.1: Consider the single-player game tree shown below. and answer the questions that follow.



(*a*) What is the minimax value of the tree?

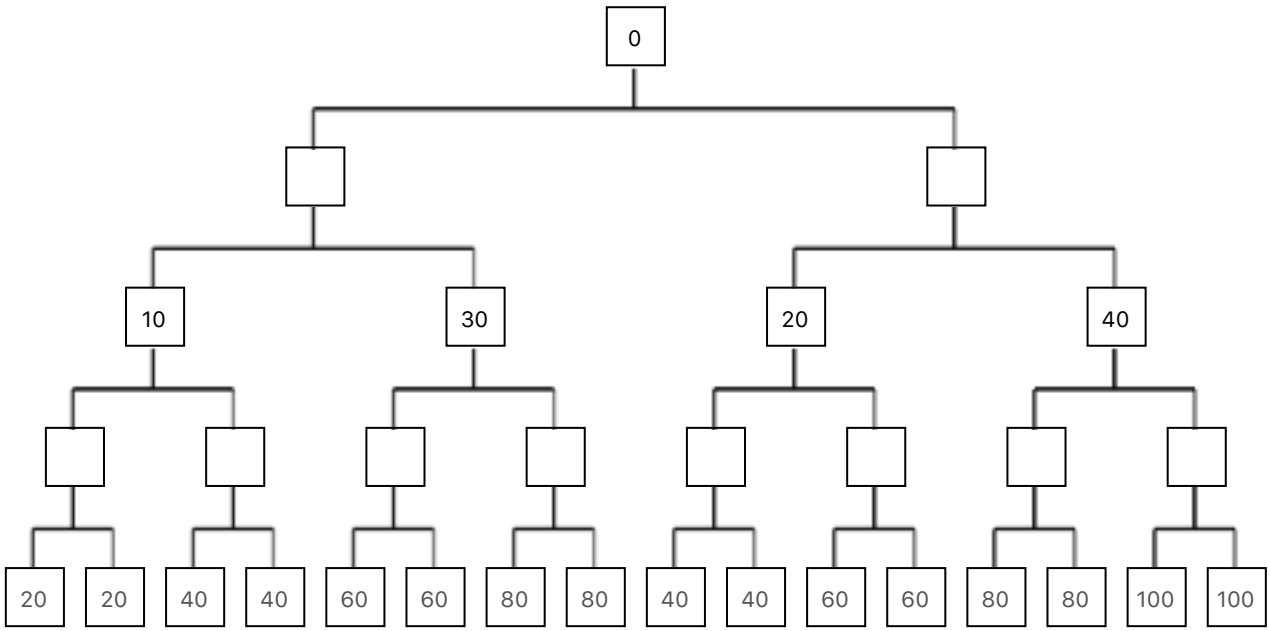
(*b*) What is the value returned by Depth-Limited Search with a depth limit of 3?

(*c*) How many nodes are examined by depth-first search with a depth-limit of 3, i.e. how many times is `maxscore` called?

(*d*) How many nodes are examined by breadth-first search with a depth-limit of 3, i.e. how many times is `maxscore` called?

(*e*) How many nodes are examined by iterative-deepening search with a depth-limit of 3 and a depth increment of 1, i.e. how many times is `maxscore` called?

Exercise 7.2: Consider the two-player game tree shown below. The values on the max nodes are actual goal values for the associated states as given in the game description, *not* state utilities determined by game tree search.



(*a*) What is the state utility of the top of the tree (as determined, for example, by Minimax)?

(*b*) Now consider a player using fixed-depth heuristic search with depth limit 1. How many max nodes are searched in evaluating the top node in this tree (i.e. how many times is maxscore called)?

(*c*) Suppose the player uses a goal proximity heuristic with state reward as the heuristic value for non-terminal states. What is the minimum final reward for this player in this game?