CHAPTER 4
Game Playing

4.1 Introduction

The simplest sort of general game player is one that explores the game tree implicit in a game description. In this chapter, we talk about some infrastructure that frames the problem more precisely. We then consider a couple of search-free uses of this infrastructure, viz. legal players and random players. In Chapters 5-8, we look at complete search techniques (which are appropriate for small game trees) as well as incomplete search techniques (which are necessary for very large game trees). In Chapters 9-12, we examine some game playing techniques based on properties of states. Finally, in chapters 13-16, we show ways that game descriptions can be used to deduce general properties of games without explicitly enumerating states or properties.

4.2 Infrastructure

A game player is typically implemented as a web service. The service receives messages from a Game Manager and replies appropriately. Building a player means writing event handlers for the different types of messages in the GGP communication protocol.

To make things concrete, let us assume that we have at our disposal a code base that (1) includes a listener to call these event handlers on receipt of messages from a Game Manager and (2) includes subroutines for processing game descriptions and match data. Our job in building a player is to use the available subroutines to implement event handlers for the various GGP messages.

Once running, the listener enters a loop listening for messages from a Game Manager. Upon receipt of a message, the listener calls the appropriate handler. When the handler is finished, the listener sends the returned value to the Game Manager. The handlers called by the listener are listed below.

`info()`

`start(`*id*`,`*role*`,`*rules*`,`*startclock*`,`*playclock*`)`

`play(`*id*`,`*move*`)`

`stop(`*id*`,`*move*`)`

`abort(`*id*`)`

In order to facilitate the implementation of these message handlers, we assume that our code base contains definitions for the subroutines described below. There are subroutines for computing most of the components of a match.

`findroles(`*game*`)` - returns a sequence of roles.

`findpropositions(`*game*`)` - returns a sequence of propositions.

`findactions(`*role*`,`*game*`)` - returns a sequence of actions for a specified role.

`findinits(`*game*`)` - returns a sequence of all propositions that are true in the initial state.

`findlegalx(`*role*`,`*state*`,`*game*`)` - returns the first action that is legal for the specified role in the specified state.

`findlegals(`*role*`,`*state*`,`*game*`)` - returns a sequence of all actions that are legal for the specified role in the specified state.

`findnext(`*roles*`,`*move*`,`*state*`,`*game*`)` - returns a sequence of all propositions that are true in the state that results from the specified roles performing the specified move in the specified state.

`findreward(`*role*`,`*state*`,`*game*`)` - returns the goal value for the specified role in the specified state.

`findterminalp(`*state*`,`*game*`)` - returns a boolean indicating whether the specified state is terminal.

That's it. As mentioned above, our job is to use these subroutines to write the handlers called by the listener. In the remainder of this chapter, we look at a couple of simple approaches to doing this.

4.3 Creating a Legal Player

A *legal player* is the simplest form of game player. In each state, a legal player selects an action based solely on its legality, without consideration of the consequences. Typically, the choice of action is consistent - it selects the same action every time it finds itself in the same state. (In this way, a legal player differs from a random player, which selects different legal actions on different occasions.)

Legal play is not a particularly good general game playing strategy. However, it is a worthwhile exercise to build a legal player (and a random player) just to get familiar with the concepts described above and to have a basis of comparison for more intelligent players.

Using the basic subroutines provided in the GGP starter pack, building a legal player is very simple. We start by setting up some global variables to maintain information while a match is in progress. (Properly, we should create a data structure for each match; and we should attach these values to this data structure. However, we are striving for simplicity of implementation in these notes. This does not mean that you should do the same.)

```
var game;
var role;
var roles;
var state;
```

Next, we define a handler for each type of message. The `info` handler simply returns `ready`.

```
function info ()
 {return 'ready'}
```

The `start` event handler assigns values to `game` and `role` based on the incoming `start` message; it uses `findroles` to compute the roles in the game; it uses `findinits` to compute the initial state; and it returns `ready`, as required by the GGP protocol.

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

The `play` event handler takes a match identifier and a move as arguments. It first uses the `simulate` subroutine to compute the current state. If the move is `nil`, the subroutine returns the current state. Otherwise, it uses `findnext` to compute the state resulting from the specified move and the specified state. Once our player has the new state, it uses `findlegalx` to compute a legal move.

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

function simulate (move,state)
 {if (move=='nil') {return state};
  return findnext(roles,move,state,game)}
```

The `abort` event handler for our player does nothing. It ignores the inputs and simply returns `done` as required by the GGP protocol.

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

Like the `abort` message handler, the `stop` event handler for our legal player also does nothing. It simply returns `done`.

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

Just to be clear on how this works, let's work through a short Tic Tac Toe match. When a player is initialized, it sets up data structures to hold the game description, the role, and the state. These are initially empty.

Let's assume that our player receives a `start` message from a Game Manager of the sort shown below. The match identifier is `m23`. Our player is asked to be the `x` player. There are the usual axioms of Tic Tac Toe. The start clock and play clock are both 10 seconds.

```
start(m23, white, [role(white),role(black),...], 10, 10)
```

On receipt of this message, our listener calls the `start` handler. This sets the global variables accordingly. The returned value `ready` is then sent back to the Game Manager.

Once the Game Manager is ready, it sends a suitable `play` message to all players. See below. Here we have a request for each player to choose an action for match m23. The argument `nil` signifies that this is the first step of the match.

```
play(m23,nil)
```

On receipt of this message, our listener invokes the `play` handler with the arguments passed to it by the Game Manager. Since the move is `nil`, our player computes the current state by calling `inits` on the game description. This results in the dataset shown below.

```
true(cell(1,1,b))
true(cell(1,2,b))
true(cell(1,3,b))
true(cell(2,1,b))
true(cell(2,2,b))
true(cell(2,3,b))
true(cell(3,1,b))
true(cell(3,2,b))
true(cell(3,3,b))
true(control(white))
```

Using this state, together with the role and game description associated with this match, our player then computes the first legal move, i.e. `mark(1,1)` and returns that as answer.

The Game manager checks that the actions of all players are legal, simulates their effects and updates the state of the game, and then sends `play` messages to the players to solicit their next actions. In this case, our player would receive the message shown below.

```
play(m23,[mark(1,1),noop])
```

Again, our player invokes its `play` handler with the arguments passed to it by the Game Manager. This time, the move is not `nil`, and so our player uses `nexts` to compute the next state. This results in the dataset shown below.

```
true(cell(1,1,x))
true(cell(1,2,b))
true(cell(1,3,b))
true(cell(2,1,b))
true(cell(2,2,b))
true(cell(2,3,b))
true(cell(3,1,b))
true(cell(3,2,b))
true(cell(3,3,b))
true(control(black))
```

Using this state, our player then computes the first legal move, its only legal move, viz. `noop`, and returns that as answer.

This process then repeats until the end of the game, at which point our player receives a message like the one shown below.

```
stop(m23,[mark(3,3),noop])
```

While some players are able to make use of the information in a stop message, our legal player simply ignores this information and returns `done`, terminating its activity on this match.

4.4 Creating a Random Player

A *random player* is similar to a legal player in that it selects an action for a state based solely on its legality, without consideration of the consequences. A random player differs from a legal player in that it does not simply take the first legal move it finds but rather selects randomly from among the legal actions available in the state, usually choosing a different move on different occasions.

The implementation of a random player is almost identical to the implementation of a legal player. The only difference is in the `play` handler. In selecting an action, our player first computes all legal moves in the given state and then randomly selects from among these choices (using the `randomelement` subroutine). One way of writing the code for the play handler is shown below.

```
function play (id,move)
 {state = simulate(move,state);
  var actions=findlegals(role,state,game);
  return randomelement(actions)}
```

Random players are no smarter than legal players. However, they often appear more interesting because they are unpredictable. Also, they sometimes avoid traps that befall consistent players like legal, which can sometimes maneuver themselves into a corner and be unable to escape. They are also used as standards to show that general game players or specific methods perform better than chance.

A random player consumes slightly more compute time than a legal player, since it must compute all legal moves rather than just one. For most games, this is not a problem; but for games with a large number of possible actions, the difference can be noticeable.