CHAPTER 10

General Game Playing With Propnets

10.1 Introduction

As we saw in Chapter 4, a game player is typically implemented as a web service that 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 using 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.

Building a game player using a propnet is essentially the same. In fact, all of the methods we have described so far still apply. The main difference is that we replace the subroutines that work directly on game descriptions with subroutines that operate on propnets. For example, in place of the `findlegals` subroutine we used earlier, we now use a `proplegals` subroutine that computes legal actions using the propnet; in place of the `findterminalp` subroutine we used earlier, we now use a `propterminalp` subroutine that computes whether or not a state is terminal using the propnet; and so forth.

We begin this chapter by presenting a representation of propnets as data structures. We then define the basic subroutines for marking propnets and reading marks on propnets. Then we use these subroutines to build up the General Game Playing subroutines needed by our players (e.g. `proplegals`, `propterminalp`, and so forth).

10.2 Propositional Nets as Data Structures

Before we can define methods for processing propnets, we need a representation of those propnets as data structures. In the simple approach taken here, we represent each propnet component as a structured object with various components depending on the type of the object. The connectivity of the propnet is captured by values of these components.

A propnet as a whole is represented by an object of the form shown below. The type is `propnet`. The `roles` component is a sequence of the roles in the game. The `inputs` component is a sequence of sequences of input nodes, one sequence for each role; the `bases` component is a sequence of the base nodes in the game; and the `view` component is a sequence of the view nodes in the game. The `legals` component is a sequence of sequences of legal nodes, one sequence for each role; the `rewards` component is a sequence of sequences of reward nodes; and the terminal component is the terminal node of the game.

```
{type:'propnet',
 roles:roles, inputs:inputs, bases:bases, views:views,
 legals:legals, rewards:rewards, terminal:terminal}
```

An input node is a structured object of the form shown below. The `type` component is `input`; the `name` is the GDL representation of the input; and the mark is an indication of whether the node is true or false in the current state (as determined by the input marking current at each point in time).

```
{type:'input',name:name,mark:boolean}
```

A base node is a structured object of the form shown below. The `type` component is `base`; the `name` is the GDL representation of the node; the source is the transition that leads to the base node;

and the mark is an indication of whether the node is true or false in the current state (as determined by the base marking current at each point in time).

`{type:'base',name:`*name*`,source:`*transition*`,mark:`*boolean*`}`

A view node is a structured object of the form shown below. The `type` component is `view`; the `name` is the GDL representation of the node; the source is the connective that leads to the view node.

`{type:'view',name:`*name*`,source:`*connective*`}`

A negation is a structured object of the form shown below. The `type` component is `negation`, and the source is the proposition that leads to the connective.

`{type:'negation',source:`*source*`}`

A conjunction is a structured object of the form shown below. The `type` component is `conjunction`, and the sources component is a sequence of propositions that leads to the connective.

`{type:'conjunction',source:`*sources*`}`

A disjunction is a structured object of the form shown below. The `type` component is `disjunction`, and the sources component is a sequence of propositions that leads to the connective.

`{type:'disjunction',sources:`*sources*`}`

A transition is a structured object of the form shown below. The `type` component is `transition`, and the source is the proposition that leads to the transition.

`{type:'transition',source:`*source*`}`

As an example, consider the propnet presented in the preceding chapter. For convenience, it appears here again as Figure 10.1. In what follows, we see how to assemble this propnet manually. In a general game player, it would be automatically generated from the GDL game description.

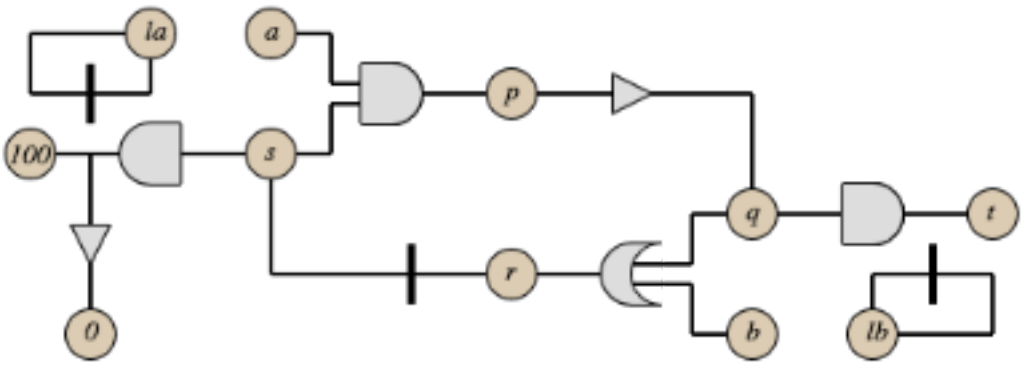

Figure 10.1 - Propnet for a simple game

First of all, we create input, base, and view nodes for the propositions in the propnet. Since we do not yet have data structures for our connectives, we initialize the sources of our propositions with `null`.

```
a = {type:'input',name:'a'}
b = {type:'input',name:'b'}
p = {type:'view',name:'p',source:null,mark:false}
q = {type:'view',name:'q',source:null,mark:false}
r = {type:'view',name:'r',source:null,mark:false}
s = {type:'base',name:'s',source:null,mark:false}

la = {type:'base',name:'la',source:null,mark:false}
```

```
lb = {type:'base',name:'lb',source:null,mark:false}
g100 = {type:'view',name:'100',source:null,mark:false}
g0 = {type:'view',name:'0',source:null,mark:false}
t = {type:'view',name:'terminal',source:null,mark:false}
```

Next, we create connectives and insert the propositions as components.

```
a1 = {type:'conjunction',sources:[a,s]}
a2 = {type:'conjunction',sources:[q,q]}
a3 = {type:'conjunction',sources:[s,s]}
i1 = {type:'negation',source:p}
i2 = {type:'negation',source:g100}
o1 = {type:'disjunction',sources:[q,b]}
t1 = {type:'transition',source:r}
t2 = {type:'transition',source:la}
t3 = {type:'transition',source:lb}
```

Now, we go back and insert the connectives into the proposition nodes.

```
p.source = a1
q.source = i1
r.source = o1
s.source = t1

la.source = t2
lb.source = t3
g100.source = a3
g0.source   = i2
t.source = a2
```

Finally, we insert all of these things into the components of `mypropnet`.

```
mypropnet =
 {type:'propnet',
  inputs:[a,b],
  bases:[s, la, lb],
  views:[p,q,r],
  rewards:[g100,g0],
  terminal:t}
```

Once we have created our propnet, either manually or automatically from the GDL for a game, we can access the components by simply naming the propnet and one of its components and extracting the desired component. For example, to get a sequence of the base propositions in `mypropnet`, we would ask for `mypropnet.base`; and to get the source of proposition `r`, we would ask for `r.source`.

10.3 Marking and Reading Propositional Nets

In our approach to General Game Playing using propnets, we use input markings in place of moves and base markings in place of states. In order to compute the various attributes of a game on a given step, we typically mark the input propositions and base propositions of the propnet and then compute the corresponding view marking. We then read the view marking to compute the desired attributes. In this section, we discuss the details of marking and reading propnets, and in the next section we show how they are used.

There are two different approaches to marking and reading propnets. In the *forward propagation* approach, we mark propositions and then propagate the values to compute the values of view propositions dependent on those marks. To answer questions, we then just read the values of the relevant view propositions. In the *backward reasoning* approach, we mark our base and input propositions but do not propagate. Instead, when we need a value for a view proposition, we work backward from the desired view proposition to determine that value. The forward method saves

redundant computation. The backward methods is a little simpler and not much slower. We show the backward method here.

Marking propositions is easy. See the subroutines below. We simply iterate through the base propositions or input propositions marking those nodes with values from the input or base vectors of booleans.

```
function markbases (vector,propnet)
 {var props = propnet.bases;
  for (var i=0; i<props.length; i++)
      {props[i].mark = vector[i]};
  return true}

function markactions (vector,propnet)
 {var props = propnet.actions;
  for (var i=0; i<props.length; i++)
      {props[i].mark = vector[i]};
  return true}

function clearpropnet (propnet)
 {var props = propnet.bases;
  for (var i=0; i<props.length; i++)
      {props[i].mark = false};
  return true}
```

Computing the values of view propositions is accomplished by working backwards from the propositions of interests. If the proposition is an input or base proposition, we simply return the mark on that proposition. In the case of a view proposition, we compute the values of propositions for the inputs to the connective feeding the view proposition and combine those values in accordance with the type of the connective.

```
function propmarkp (p)
 {if (p.type=='base') {return p.mark};
  if (p.type=='input') {return p.mark};
  if (p.type=='view') {return propmarkp(p.source)};
  if (p.type=='negation') {return propmarknegation(p)};
  if (p.type=='conjunction') {return propmarkconjunction(p)};
  if (p.type=='disjunction') {return propmarkdisjunction(p)};
  return false}

function propmarknegation (p)
 {return !propmarkp(p.source)}

function propmarkconjunction (p)
 {var sources = p.sources;
  for (var i=0; i<sources.length; i++)
      {if (!propmarkp(sources[i])) {return false}};
  return true}

function propmarkdisjunction (p)
 {var sources = p.sources;
  for (var i=0; i<sources.length; i++)
      {if (propmarkp(sources[i])) {return true}};
  return false}
```

10.4 Computing Game Playing Basics

Now that we have some tools for marking and reading propnets, let's see how we can use them to define the basic subroutines used in the game playing methods we defined in previous chapters.

In order to compute the legal actions for a role in a given state using a propnet, we first mark the base propositions of the propnet using the information in the given state. We then check each of the legality nodes for the given role in the propnet. If the node is true, we put the corresponding input node on the list. When we are done, we return the list of input nodes that we have

accumulated in this process. (Note that, to return one of these actions to a game manager, we would need to extract the `name` component from the corresponding input node.)

```
function proplegals (role,state,propnet)
 {markbases(state,propnet);
  var roles = propnet.roles;
  var legals = seq();
  for (var i=0; i<roles.length; i++)
      {if (role==roles[i]) {legals = propnet.legals[i]; break}};
  var actions = seq();
  for (var i=0; i<legals.length; i++)
      {if (propmarkp(legals[i]))
          {actions[actions.length]=legals[i]}};
  return actions}
```

The `propnext` subroutine shown below computes the next state for a given move and a given state using a given propnet. (Note that the move here is assumed to be an input marking, not just a sequence of actions. To simulate a move supplied by a game manager, we would need to convert to an input marking before using `propnext`.) In executing the `propnext` subroutine, we first mark the input propositions using the given move, and we mark the base propositions of the propnet using the information in the given state. We then check each of the base propositions in the propnet, collecting those that are true in the next state.

```
function propnext (move,state,propnet)
 {markactions(move,propnet);
  markbases(state,propnet);
  var bases = propnet.bases;
  var nexts = seq();
  for (var i=0; i<bases.length; i++)
      {nexts[i] = propmarkp(bases[i].source.source)};
  return nexts}
```

To compute the reward for a given role in a given state using a propnet, we first mark the base propositions as before. We then check each of the reward propositions for the given role until we find one that is true.

```
function propreward (role,state,propnet)
 {markbases(state,propnet);
  var roles = propnet.roles;
  var rewards = seq();
  for (var i=0; i<roles.length; i++)
      {if (role==roles[i]) {rewards = propnet.rewards[i]; break}};
  for (var i=0; i<rewards.length; i++)
      {if (propmarkp(rewards[i])) {return rewards[i].name}};
  return 0}
```

Computing whether or not a state is terminal is particularly easy. We mark the base propositions as before and check the terminal node for the propnet.

```
function propterminalp (state,propnet)
 {markbases(state,propnet);
  return propmarkp(propnet.terminal)}
```