CHAPTER 12

Discovery of Heuristics

12.1 Introduction

The biggest open problem in General Game Playing is the discovery of game-specific pruning rules and/or evaluation functions. In many games, it is possible to find such heuristics. The challenge to find such heuristics without searching the entire game tree.

In this chapter, we look at just one technique of this sort. We start by defining latches and inhibitors and discuss how to find them. Then we show how the concepts work together in a useful technique for removing states from the game tree and ordering states within the game tree.

12.2 Latches

A *latch* is a proposition that, once it becomes true (or, respectively, false), remains true (or, respectively, false) no matter what the players do. A latch is not the same as a constant in that it may change value during the game; but, once the value changes, it keeps the new value for all remaining steps.

Almost all of the propositions in our usual description of Tic Tac Toe are latches. Once a square is marked, it keeps that mark for the rest of the game; and once a cell ceases to be blank, it can never again become blank. In other words, propositions like `cell(1,2,x)` and `cell(2,3,o)` and `cell(3,3,b)` are all latches.

Given a propnet, it is straightforward to find latches. We enumerate values for all base markings and check the value of the candidate proposition. If the value remains the same, then the proposition is a latch.

One feature of this approach is that we do need to consider arbitrary action histories, only possible combinations of actions. The downside is that we still need to cycle through all possible states of the propnet as well. And thus the technique is prohibitively expensive in general.

A partial solution to this problem is to restrict one's attention to those base and input propositions that determine the proposition being checked. A set of base and input propositions determine all and only the propositions in the propnet that are downstream from it, i.e. the propositions themselves or the outputs of connectives for which they are the inputs and so forth. In the worst case, this may be as bad as the full search just described, but in many cases it is much less expensive.

Note that the idea of latches can be extended to groups of propositions. While no individual proposition in the group is a latch, it is possible that the group as a whole might constitute a latch. Consider, for example, a game in which one of `p` or `q` is always true. Then neither `p` nor `q` is a latch but their disjunction does form a latch, even if that disjunction is not a node in the propnet. Obviously, finding composite latches is more expensive than finding individual latches, but it may be helpful in some games.

12.3 Inhibitors

A proposition `p` inhibits a proposition `q` if and only if `p` must be false to achieve `q` or retain `q`. In other words, whenever `p` is true in a state, `q` is false in the next state. When this occurs, we say that

p is an inhibitor of q.

Some inhibitors can be detected by a simple scan of the propnet for a game. If every path from the base propositions and input propositions to q involve the negation of p, then p is an inhibitor of q.

12.4 Dead State Removal

Dead state removal is the removal of dead states from a game tree, i.e. states that cannot lead to satisfactory terminal states, i.e. those that can give a player an adequate score (e.g. 100 points, points above a desired threshhold, or a plurality of points in a zero sum game). Pruning the game tree below dead states can save lots of computation.

In a propnet, if the truth (or falsity) of a proposition makes it a dead state, then we should strive to ensure that the that proposition never becomes true (or, respectively false). So the technique could equally well be called dead proposition detection.

The trick in dead state removal is to find such states without actually searching the game tree starting with those states. Many ways of doing this have been proposed. In this section, we examine one such technique based on the concepts of latches and inhibitors.

As an example, consider a game with the following behavioral rule. White gets 100 points if both q and r are true. If we assume that this is the only way in which white can get 100 points, then both q and r are goal inhibitors.

```
goal(white,100) :- true(q) & true(r)
```

Now let's add in the rule shown below. If q ever becomes true, it remains true forever. In other words, q is a latch. Obviously, in a situation like this, we would like to ensure that q never becomes true or we lose the possibility of getting 100 points.

```
next(q) :- true(q)
```

Generalizing this gives us our rule for dead state removal / dead proposition setting - if a latch inhibits the goal in a game, then we should not consider states in which that proposition is set to true.

As an example of dead state removal in action, consider Untwisty Corridor. It is a single player game. There are nine propositions - p and q1,..., q8. And there are four action - a, b, c, and d. q1 is true in the initial state, and the goal is to turn on q8. Actions a, b, and c all turn on p. If a q proposition is true, then action d turns on the next in sequence, provided that p is false; otherwise it does nothing. There is a step counter that terminates the game after 7 steps.

The game tree for Untwisty Corridor has 349,525 states. However, most of those states are dead states. Recognizing that p is a latch and a goal inhibitor allows us to prune all of these dead states, leaving a total of 8 states to be considered, a considerable savings.

Note that this basic idea can be generalized in various ways. For example, we can look for latches that inhibit 90 point states and latches that inhibit 80 point states and so forth; and we can use our discoveries to order states depending on the values of the goals that are affected.