C H A P T E R n

Reformulation

1. Introduction

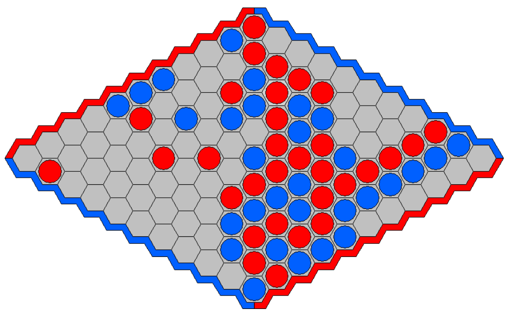
The optimization methods described in the preceding chapter produce game descriptions that are logically equivalent to the game descriptions they are given as inputs. The output descriptions use the same vocabulary; and, in all cases, they logically entail the same conclusions. The methods do no more than drop subgoals or rules or rearrange them in ways that make evaluation more efficient.

Restricting one's self to logically equivalent transformations is safe, but it is not the only way to go. By changing the concepts used in descriptions and, more interestingly, adding or dropping base relations, it is sometimes possible to produce more dramatic efficiency improvements while describing the same game trees.

In the next section, we look at an interesting example of this sort. Then in the sections following that, we look at some methods that can be combined to accomplish this sort of reformulation. First, we look at a method for converting view relations into base relations, a process called *materialization*. After that, we look at the reverse process, viz. *dematerialization*, i.e. demoting base relations to ordinary view relations. And, finally, we look at some interesting techniques for inventing entirely new relations that can then be used either as view relations or base relations.

2. Example

As example of game reformulation, consider the game of Hex. It is an alternating move game played on an 11x11 hexagonally connected board like the one shown here. On each step, one of the players places a token of his color on an open cell. The first to make connect one side of the board to the other is the winner. Here, red has won with a circuitous path.



The key to the game is the concept of connection. See below for one way to axiomatize this notion. Red has a win if and only there is a cell on the left side of the board and there is a red path leading from that cell to a cell on the right side of the board. There is a red path from a cell X to itself if and only if X contains a red token *or* if X contains a red token and is adjacent to a cell with a red path to Z.

```
redwin :- left(X) &  redpath(X,Y) & right(Y)
redpath(X,X) :- cell(X,red)
redpath(X,Z) :- cell(X,red) & next(X,Y) & redpath(Y,Z)
```

Mathematically, this definition is correct. Unfortunately, it is not so good computationally. If there is a path, this is very expensive to compute, since it goes back over old ground again and again. In fact, when there is no path, it can cause some logic interpreters, such as the Prolog interpreter, to run forever.

One way to fix this problem is to include a path argument in `redpath`. The path is augmented on recursive calls and search terminates whenever it encounters a node that has already been seen.

```
redwin :- left(X) &  redpath(X,Y,nil) & right(Y)
redpath(X,X,nil) :- cell(X,red)
redpath(X,Z,P) :-
  cell(X,red) & next(X,Y) & ~in(Y,P) & redpath(Y,Z,cons(Y,P))
```

This version prevents infinite loops, but there is still some cost in maintaining and checking the list of visited cells.

Luckily, we can do even better. For each connected patch of cells we can associate a representative element and maintain a table of connections between this element and the members of the connected patch of cells. To check whether two cells are connected we just check to see whether than have the same representative element.

```
redwin :- left(X) &  rep(X,K) &  rep(Y,K) & right(Y)
```

The upshot is that we can significantly improve the cost of computation. Moreover, this is not that difficult a transformation. It can be found in any good textbok on algorithms as a minimal spanning tree transformation. A player that does this transformation is likely to play faster by several orders of magnitude.

In the next few sections, we look at some methods that, when combined, can produce game reformulations of this sort.

3. Materialization

Forthcoming.

4. Dematerialization

Forthcoming.

5. Conceptual Reformulation

Forthcoming.