CHAPTER x

Optimization

1. Introduction

The game playing methods described in the preceding chapters provide a variety different ways to create and search game trees. All of these methods are designed to operate at runtime, i.e. while the play clock is running. In this chapter and the next few chapters, we look at some methods to reformulate game descriptions so that the methods we have seen can execute more efficiently. Unlike the methods we have already seen, these methods are typically executed during the start clock. If one of these methods manages to produce an improved game description, the player can then supply that description to its runtime method in place of the original description. If not, the player simply passes along the original game description.

In this chapter, we look at various methods for logical optimization of game descriptions. The optimizations are termed logical because the resulting game descriptions are logically equivalent to the original descriptions. As we shall see, these optimizations are are similar to logical reformulations (described in the next chapter) in that they both generate game trees that are isomorphic to the game trees of the original descriptions. However, logical reformulations are different from logical optimizations in that they involve different base relations and, therefore, are not logically equivalent.

There is extensive literature on methods for logical optimization. In this chapter, we look at just a few of these methods - subgoal ordering, redundant subgoal removal, and redundant rule removal. All are fairly simple to understand in and of themselves. The most interesting part of implementing these optimizations is finishing with sufficient time left over to set up the runtime player to use the new game description (if one is found) in place of the original description.

Note that the value of logical optimization depends on the exact method used to evaluate queries. What works for one method may not be effective for another. In our discussion here, we assume a standard top-down, left-to-right evaluation procedure with no indexing or caching, like the one described in chapter 4.

2. Subgoal Ordering

One very common source of inefficiency in logical evaluation stems from non-optimal ordering of subgoals within rules. The good news is that it is often possible to find better orderings just by looking at the form of the rules involved even without looking at the data to which the rules are applied.

As an example of inefficiency due to subgoal ordering, consider the rule shown below. Relation `s` is true of `x` and `y` if `p` is true of `x` and `r` is true of `x` and `y` and `q` is true of `x`.

```
s(X,Y) :- p(X) & r(X,Y) & q(X)
```

Intuitively, this seems like a bad way to write this rule for our usual evaluation procedure. It seems as though the `q` condition should come before the `r` condition, as in the following rule.

```
s(X,Y) :- p(X) & q(X) & r(X,Y)
```

In fact, there is good reason for this intuition. For our standard evaluation procedure, the worst-case cost of evaluating the first rule is n^4, where n is the size of the domain of objects. By contrast,

the worst-case cost of computing the relation given the second definition is just n^3.

Let's look at these two cases in more detail. In the worst case, there are $n^2 + 2n$ facts in the database, where n is the number of objects in the domain.

In evaluating the first rule, our algorithm would first examine all n^2 + 2n facts to find those that match p(X). There would be at most 4 answers. For each of these, the algorithm would again look at n^2 + 2n facts to find those that match r(X,Y). There would be 4 for each value of X. Finally, for each of these pairs, the algorithm would again examine at n^2 + 2n facts to find those that match q(X). The grand total is shown below.

$$(n^2 + 2n) + n*((n^2 + 2n) + n*(n^2 + 2n)) = n^4 + 3n^3 + 3n^2 + 2n$$

In evaluating the second rule, the algorithm would again examine all n^2 + 2n facts to find those that match p(X). There would be at most 4 answers. For each of these, the algorithm would again look at n^2 + 2n facts to find those that match q(X). There would be at lost one for each value of X. Finally, for each such X, the algorithm would examine n^2 + 2n facts to find those that match r(X,Y). The grand total is shown below.

$$(n^2 + 2n) + n*((n^2 + 2n) + 1*(n^2 + 2n)) = 2n^3 + 5n^2 + 2n$$

Suppose, for example, there were 4 objects in the domain. In this case, there would be at most 4 p facts and 4 q facts and 16 r facts. Evaluating the first rule would require 504 unifications, while evaluating the second rule would require only 208.

In the presence of indexing, the asymptotic complexity is the same for both orderings. However, the lower degree terms for the second ordering suggest that it is still the better ordering. Moreover, it is possible to show that, averaging over all possible databases, the second rule is better than the first.

Fortunately, there is a simple method for reordering subgoals in situations like this. The basic idea is to assemble a new body for the rule incrementally, picking a subgoal on each step and removing it from the list of remaining subgoals to be considered. In making its choice, the method examines the remaining subgoals in left-to-right order. If it encounters a subgoal all of whose variables are bound by subgoals already chosen, then that subgoal is added to the new rule and removed from the list of remaining subgoals. If not, the method removes the first remaining subgoal from the list, adds it to the new rule, updates its list of bound variables, and moves on to the next step.

The code for the method is shown below. The code utilizes three subroutines not defined here. The subroutine `varsexp` takes an expression and a variable list as arguments and adds to the list any variables in the expression that are not already on the list. The subroutine `varp` takes an expression as argument and returns true if the expression is a variable, and otherwise it returns false. The subroutine `adjoin` adds an item to a sequence if it is not already a member of that sequence.

```
function reorder (rule)
 {var vl = seq();
  var sl = rule.slice(2);
  var newrule = seq('rule',rule[1]);
  while (sl.length>0)
   {var ans = getbest(sl,vl);
    newrule.push(ans);
    vl = varsexp(ans,vl)};
  return newrule}

function getbest (sl,vl)
 {var varnum = 10000;
  var best = 0;
  for (var i=0; i<sl.length; i++)
      {var dum = unboundvarnum(sl[i],vl);
       if (dum<varnum) {varnum = dum; best = i}};
```

```
  var ans = sl[best];
  sl.splice(best,1);
  return ans}

function unboundvarnum (x,vs)
 {return unboundvars(x,seq(),vs).length}

function unboundvars (x,us,vs)
 {if (varp(x)) {if (find(exp,vs)) {return us} else {return adjoin(x,us)}};
  if (symbolp(x)) {return us};
  for (var i=0; i<x.length; i++)
      {us = unboundvars(x[i],us,vs)};
  return us}
```

As an example of this method in action, consider the first rule shown above. At the start, the list of remaining subgoals consists of all three subgoals in the rule. At this point, none of the three subgoals is ground, so the method chooses the first subgoal `p(X)`, adds it to its new rule, and puts `X` on the list of bound variables. On the second step, the method looks at the remaining two subgoals. The subgoal `r(X,Y)` contains the unbound variable `Y` and so it is not chosen. By contrast, all of the variables in the subgoal `q(X)` are bound, so the method outputs this subgoal next. On the third step, the final subgoal is added forming the second rule shown above.

3. Redundant Subgoal Removal

Another common source of inefficiency in logical evaluation stems from the presence of redundant subgoals within rules. In many cases, it is possible to detect and eliminate such redundancies.

As a simple example of the problem, consider the rule shown below. Relation `r` is true of `X` and `Y` if `p` is true of `X` and `Y` and `q` is true of `Y` and `q` is also true of some `Z`.

```
r(X,Y) :- p(X,Y) & q(Y) & q(Z)
```

It should be clear that the subgoal `q(Z)` is redundant here. If there is a value for `Y` such that `q(Y)` is true, then that value for `Y` also works as a value for `Z`. Consequently, we can drop the `q(Z)` subgoal (while retaining `q(Y)`), resulting in the rule shown below.

```
r(X,Y) :- p(X,Y) & q(Y)
```

Note that the opposite is not true. If we were to drop `q(Y)` and retain `q(Z)`, we would lose the constraint on the second argument of `p`, which is also an argument to `r`.

Fortunately, it is easy to determine which subgoals can be removed and which need to be retained. The code for the method is shown below. The code utilizes one new subroutine not defined here. The subroutine `sublis` takes an expression and a binding list as arguments and returns a copy of the expressions with variables on the binding list replaced with their values.

```
function prunesubgoals (rule)
 {var vl = vars(rule[1]);
  var newrule = seq('rule',rule[1]);
  for (var i=2; i<rule.length; i++)
      {var sl = newrule.slice(2).concat(rule.slice(i+1));
       if (!pruneworthyp(sl,rule[i],vl)) {newrule.push(rule[i])}};
  return newrule}

function pruneworthyp (sl,p,vl)
 {vl = varsexp(sl,vl.slice(0));
  var al = seq();
  for (var i=0; i<vl.length; i++)
      {al[vl[i]] = 'x' + i};
  var facts = sublis(sl,al);
  var goal = sublis(p,al);
```

```
        return compfindp(goal,facts,seq())}
```

As an illustration of this method in action, consider the example shown above. The method first computes the variables in the head of the rule, viz. `[X,Y]` and initializes the variable `newrule` to a new rule with the same head as the original rule. It then iterates over the body of the rule, adding subgoals to the new rule once they are checked for redundancy.

On the first step of the iteration, the method focusses on `p(X,Y)`. It creates a dataset consisting of instances of the remaining subgoals, viz. `q(x1)` and `q(x2)`; it then tries to prove `p(x0,x1)`; and, in this case, it fails. So `p(X,Y)` is added to the new rule.

On the second step, the method focusses on `q(Y)`. It creates a dataset consisting of instances of the remaining subgoals, viz. `p(x0,x1)` and `q(x2)`; it then tries to prove `q(x1)`; and, once again, it fails. So `q(Y)` is added to the new rule.

Finally, the method focusses on `q(Z)`. It creates a dataset consisting of instances of the other subgoals, viz. `p(x0,x1)` and `q(x1)`; it then tries to prove `q(Z)`. Note that `Z` is not bound here since it does not occur as a head variable or a variable in any of the other subgoals. In this case, the test succeeds; and so `q(Z)` is *not* added to the new rule.

This method is sound in that any subgoal it removes is truly redundant. As a result, any rule produced by this method is equivalent to the rule it is given as input.

Unfortunately, the method is not complete. There are redundant subgoals that it is does not detect. The problem arises when multiple redundant subgoals share variables that prevent the method from detecting the redundancy.

As an example, consider the rule shown below. The relation `r` of `X` and `Y` if `p` is true of `X` and `Y` and `q` is true of `X` and `Y` and `p` is true of `X` and `Z` and `q` is true of `X` and `Z`.

```
r(X) :- p(X,Y) & q(X,Y) & p(X,Z) & q(X,Z)
```

Clearly the last two subgoals are redundant with the first two subgoals. Unfortunately, our method does not detect that either of the subgoals in either pair is redundant because of the variables shared with the other subgoal of the pair. Try it.

Detecting this sort of redundancy can be done mechanically by considering subsets of subgoals and not just individual subgoals. However, this is more expensive than the simple method outlined above.

4. Redundant Rule Removal

An analogous form of inefficiency in logical evaluation stems from the presence of redundant rules. As with redundant subgoals within rules, it is often easy to detect and eliminate such redundancies.

As an example of the problem, consider the rules shown below. Relation `s` is true of `X` if `p` is true of `X` and `Y` and `q` is true of `b` and `r` is true of `Z`. Relation `s` is true of `X` if `p` is true of `X` and `Y` and `q` is true of `Y`.

```
s(X):-p(X,b) & q(b) & r(Z)
s(X):-p(X,Y) & q(Y)
```

Any answer produced by the first rule here is also produced by the second rule, so the first rule is redundant and can be eliminated. The trick to detecting such redundancies is to recognize that the second rule *subsumes* the first - if we replace some or all of the variables in the second rule, then

the heads are the same and all of its subgoals are members of the body of the first rule. In this case, all we need to do is to replace `Y` by `b`, and we get the rule `s(X):-p(X,b) & q(b)`, which is the same as the first rule except with fewer subgoals. Every output of the first rule is, therefore, an output of the second; and, consequently, teh first rule can be dropped.

As with redundant subgoal removal, it is easy to determine which rules can be removed and which need to be retained. The code for the method is shown below.

```
function prunerules (rules)
 {var newrules = seq();
  for (var i=0; i<rules.length; i++)
      {if (!subsumedp(rules[i],newrules) &&
           !subsumedp(rules[i],rules.slice(i+1)))
          {newrules.push(rules[i])}};
  return newrules}

function subsumedp (rule,rules)
 {for (i=0; i<rules.length; i++)
      {if (subsumesp(rules[i],rule)) {return true}};
  return false}

function subsumesp (p,q)
 {if (equalp(p,q)) {return true};
  if (symbolp(p) || symbolp(q)) {return false};
  if (p[0]==='rule' && q[0]==='rule')
     {var al = matcher(p[1],q[1]);
      if (al!==false  && subsumesexp(p.slice(2),q.slice(2),al))
         {return true}};
  return false};

function subsumesexp (pl,ql,al)
 {if (pl.length===0) {return true};
  for (var i=0; i<ql.length; i++)
      {var bl = match(pl[0],ql[i],al);
       if (bl!==false && subsumesexp(pl.slice(1),ql,bl))
          {return true}};
  return false}
```

The code utilizes two new subroutines not defined here. The subroutine `matcher` takes two expressions as arguments. If the first expression can be made to look like the second by binding the variables in the first expression, then the method returns a binding list for those variables; otherwise, it returns false. The subroutine `match` takes two expressions and a binding list as arguments and does the same except that it starts with the bindings on the given binding list.

Exercises

Exercise op.1: Consider the rule `s(X,Y,Z) :- p(X,Y) & q(Y,Z)`. Select the expression that captures the worst case complexity of our standard evaluation algorithm without indexing. The symbol *n* represents the total number of objects in the domain.

(*a*) $2n^2 + 2n^3$

(*b*) $2n^2 + 2n^4$

(*c*) $2n^2 + n^3 + n^4$

Exercise op.2: For each of the following sets of equivalent rules, say which rule is best in terms of worst case evaluation complexity using our standard algorithm without indexing.

(*a*)
```
s(X,Y,Z) :- p(X,Y) & q(X,X) & r(X,Y,Z)
s(X,Y,Z) :- p(X,Y) & r(X,Y,Z) & q(X,X)
```

```
s(X,Y,Z) :- q(X,X) & p(X,Y) & r(X,Y,Z)
s(X,Y,Z) :- q(X,X) & r(X,Y,Z) & p(X,Y)
s(X,Y,Z) :- r(X,Y,Z) & p(X,Y) & q(X,X)
s(X,Y,Z) :- r(X,Y,Z) & q(X,X) & p(X,Y)
```

(*b*)
```
s(X,Y,Z) :- p(X,Y) & q(a,b) & r(X,Y,Z)
s(X,Y,Z) :- p(X,Y) & r(X,Y,Z) & q(a,b)
s(X,Y,Z) :- q(a,b) & p(X,Y) & r(X,Y,Z)
s(X,Y,Z) :- q(a,b) & r(X,Y,Z) & p(X,Y)
s(X,Y,Z) :- r(X,Y,Z) & p(X,Y) & q(a,b)
s(X,Y,Z) :- r(X,Y,Z) & q(a,b) & p(X,Y)
```

(*c*)
```
s(X,Y,Z) :- p(X,Y,Z) & q(X) & ~r(X,Y)
s(X,Y,Z) :- p(X,Y,Z) & ~r(X,Y) & q(X)
s(X,Y,Z) :- q(X) & p(X,Y,Z) & ~r(X,Y)
s(X,Y,Z) :- q(X) & ~r(X,Y) & p(X,Y,Z)
s(X,Y,Z) :- ~r(X,Y) & q(X) & p(X,Y,Z)
s(X,Y,Z) :- ~r(X,Y) & p(X,Y,Z) & q(X)
```

Exercise op.3: For each of the rules shown below, select the expression that captures the worst case complexity of our standard evaluation algorithm without indexing. The symbol n represents the total number of objects in the domain.

```
r(X,Y) :- p(X,Y) & q(Y) & q(Z)
```
(*a*) $n^4 + n^3 + n^2 + n$
(*b*) $2n^4 + 2n^3 + n^2 + n$
(*c*) $2n^4 + 2n^3$

```
r(X,Y) :- p(X,Y) & q(Y)
```
(*a*) $n^4 + n^3 + n^2 + n$
(*b*) $2n^4 + 2n^3 + n^2 + n$
(*c*) $2n^4 + 2n^3$

Exercise op.4: For each of the following rules, select the alternative that is equivalent to the original.

(*a*)
```
r(X) :- p(X,Y) & q(Y) & q(Z)
r(X) :- p(X,Y)
r(X) :- p(X,Y) & q(Y)
r(X) :- p(X,Y) & q(Z)
```

(*b*)
```
r(X) :- p(X) & q(X) & q(W)
r(X) :- p(X)
r(X) :- p(X) & q(X)
r(X) :- p(X) & q(W)
```

(*c*)
```
r(X,Y,Z) :- p(X,Y) & q(Y) & q(Z) & q(W)
r(X,Y,Z) :- p(X,Y) & q(Y) & q(Z)
r(X,Y,Z) :- p(X,Y) & q(Y) & q(W)
r(X,Y,Z) :- p(X,Y) & q(Z) & q(W)
```

Exercise op.5: For each of the following pairs of rules, say whether the first rule subsumes the second (meaning that the second rule can be dropped).

(*a*)
```
r(X) :- p(X,Y)
r(X) :- p(X,a) & p(X,b)
```

(*b*)
```
r(X) :- p(X,a)
r(X) :- p(X,Y)
```

(*c*)
```
r(X) :- p(X,Y) & p(X,Z)
r(X) :- p(X,b) & q(b)
```