Calhoun
Institutional Archive of the Naval Postgraduate School

Calhoun: The NPS Institutional Archive

DSpace Repository

Faculty and Researchers | Faculty and Researchers' Publications

1988

Artificial Intelligence through Prolog by Neil C. Rowe

Rowe, Neil C.

Prentice-Hall

http://hdl.handle.net/10945/36984

Downloaded from NPS Archive: Calhoun

DUDLEY KNOX LIBRARY

Calhoun is a project of the Dudley Knox Library at NPS, furthering the precepts and goals of open government and government transparency. All information contained herein has been approved for release by the NPS Public Affairs Officer.

Dudley Knox Library / Naval Postgraduate School
411 Dyer Road / 1 University Circle
Monterey, California USA 93943

http://www.nps.edu/library

Problems with many constraints

Some problems have lots of conditions or *constraints* that must be satisfied. When expressed as Prolog queries, these problems appear as many predicate expressions "and"ed together. Processing these queries can be awfully slow if we don't do it right.

Special techniques are often used for such many-constraint problems. We'll talk about three such ideas in this chapter: rearranging queries to make them easier to solve, smarter ("dependency-based") backtracking, and relaxation (reasoning about possibility lists).

Two examples

Two good examples of many-constraint applications are automatic scheduling and computer vision.

Suppose we want to schedule a class to meet five times a week. Assume meeting times must be on the hour, from 9 A.M. to 4 P.M., and on weekdays. Suppose we have ten students, each of which is taking other already-scheduled classes, and we can't have our class meet when any of those meets. Furthermore, suppose we don't want our class to meet two successive hours on any day, and we don't want our class to meet more than two hours total on any day. Figure 13-1 gives an example situation and an example solution; "occupied" represents times that aren't available because of the other commitments of the ten students, and "class" represents a solution time proposed for our class. (Blank spaces are available times we don't need to use.)

This problem can be stated as a query finding values of five variables **T1**, **T2**, **T3**, **T4**, and **T5**, each value of which is a two-item list **[<day>,<hour>]**, so **[tuesday,3]** means a meeting on Tuesday at 3 P.M. Suppose meeting times of other classes are represented as **occupied([<day>,<hour>])** facts. Then the following query would solve our problem:

```
?- classtime(T1), classtime(T2), classtime(T3),
  classtime(T4), classtime(T5),
  not(occupied(T1)), not(occupied(T2)), not(occupied(T3)),
  not(occupied(T4)), not(occupied(T5)),
  not(two_consecutive_hours([T1,T2,T3,T4,T5]),
  not(three_classes_same_day([T1,T2,T3,T4,T5]).
```

For this we need the following definitions:

```
classtime([Day,Hour]) :-
  member(Day,[monday,tuesday,wednesday,thursday,friday]),
  member(Hour,[9,10,11,12,1,2,3,4]).


two_consecutive_hours(TL) :- member([Day,Hour1],TL),
  member([Day,Hour2],TL), H2 is Hour1+1, H2=Hour2.


three_classes_same_day(TL) :- member([Day,Hour1],TL),
  member([Day,Hour2],TL), member([Day,Hour3],TL),
  not(Hour1=Hour2), not(Hour1=Hour3), not(Hour2=Hour3).
```

```
member(X,[X|L]).
member(X,[Y|L]) :- member(X,L).
```

Here we use the backtracking version of the **member** predicate to both generate class times and to select class times from a list.

Computer vision often involves applying many constraints to interpret some picture. Preliminary visual processing (for example, the techniques of Section 9.16) groups parts of a picture into regions based on similarities of brightness, color, and texture. The remaining problem is to decide what each region of the picture represents. For this, variables correspond to regions, and values of the variables correspond to labels (from a finite label set) for the identity of each region. Additional constraints (conditions) affect which regions can border others, or be above others, or be inside others.

Consider the aerial photograph shown in Figure 13-2. Assume regions can be labeled either **grass**, **water**, **pavement**, **house**, or **vehicle**. Let's give some reasonable constraints about relationships of regions in an aerial photo.

> --a region cannot border or be inside a region with the same label (else we couldn't see the boundary line);
> --houses and vehicles cannot be next to or surrounded by water regions (we assume no water is deep enough for boats in this part of the world);
> --vehicles must be next to or surrounded by pavement, but pavement cannot be inside them;
> --vehicles cannot be inside houses;
> --grass cannot be inside vehicles or houses;
> --pavement cannot be completely inside another region (pavement nearly always connects to other pavement);
> --only houses, vehicles, and pavement are regular (straight-edged) regions;
> --only grass and water are irregular regions;
> --on a scale of 1 inch = 20 feet, vehicles cannot exceed an area of one quarter of a square inch.

These constraints might be violated, but aren't likely to be--remember, artificial intelligence programs don't try to be perfect, just "intelligent". We'll often know when they are violated, by being unable to find a solution. Then we can remove the most questionable constraint and try again, and then the second most questionable, and so on until we do find a solution.

Examining the photo in Figure 13-2, we see five regions; we've written names on them for convenience. R5 is the only region not "large" by the definition. Regions R3 and R5 are the only regions that are definitely regular; regions R1 and R2 are definitely irregular; but R3 is hard to classify so we won't state that it is either regular or irregular. R1 borders R2, and R2 borders R4; R3 is inside R2, and R5 is inside R4. So the photo interpretation problem can be described in a query this way:

```
?- label(R1), label(R2), label(R3), label(R4), label(R5),
  borders(R1,R2), borders(R2,R4), inside(R3,R2), inside(R5,R4),
  large(R1), large(R2), large(R3), large(R4),
  regular(R3), regular(R5), irregular(R2), irregular(R1).
```

with the following definitions:

```
label(grass).
label(water).
```

```
label(pavement).
label(house).
label(vehicle).


borders(A1,A2) :- not(A1=A2), not(water_constraint(A1,A2)),
  not(vehicle_constraint(A1,A2)).


inside(A1,A2) :- not(A1=A2), not(water_constraint(A1,A2)),
  not(vehicle_constraint2(A1,A2)),
  not(grass_constraint(A1,A2)), not(A1=pavement).

water_constraint(water,house).
water_constraint(water,vehicle).
water_constraint(house,water).
water_constraint(vehicle,water).


vehicle_constraint(A1,vehicle) :- not(A1=pavement).
vehicle_constraint(vehicle,A2) :- not(A2=pavement).
vehicle_constraint2(A1,vehicle).
vehicle_constraint2(vehicle,A2) :- not(A2=pavement).


grass_constraint(grass,house).
grass_constraint(grass,vehicle).


large(A) :- not(A=vehicle).


regular(house).
regular(vehicle).
regular(pavement).


irregular(grass).
irregular(water).
```

Here **water_constraint**, **vehicle_constraint**, and **grass_constraint** predicates specify permissible combinations of adjacent regions.

Rearranging long queries without local variables

Before processing a long query, we can often rearrange it into a more efficient form. It's usually best in an "and" to put first the predicate expressions hardest to satisfy. We can use probabilities to explain why. Take the query

```
?- a, b, c.
```

and suppose **a** is true with probability 0.9, **b** with probability 0.5, and **c** with probability 0.8. First, assume these probabilities are independent of each other. Then the preceding query has a probability of success of |0.9 * 0.5 * 0.8 = 0.36|, and a probability of failure of |0.64|. So it will more often fail than succeed. When it does fail, we would like to know as soon as possible. With the order as given, 0.1 of the time we will

recognize a failure from evaluating just the **a**, and |0.9 * 0.5 = 0.45| from evaluating both **a** and **b**. That's a total of 0.55. But if we rearrange the query as

```
?- b, c, a.
```

then 0.5 of the time we will know a failure from examining the first predicate expression, and |0.5 * 0.2 = 0.1| from examining the first two. That's a total of 0.6. So we're finding the failure earlier, meaning a little less work for us.

In general: if we have a bunch of predicate expressions "and"ed together, we should order them by *increasing* probability of success. That way, if the whole thing is going to fail, we find out as soon as possible. This also applies to all rules without local variables. For instance:

```
f(X) :- b(X), c(X), a(X).
```

If for a random **X**, **b** succeeds with probability 0.5, **c** with probability 0.8, and **a** with probability 0.5, and these probabilities are all independent of each other, then this is the best arrangement of the right side of the rule for all queries of **f** with a bound argument, as for example

```
?- f(foo).
```

There is a corresponding idea for "or"s (rules "or"ed together). Consider:

```
r :- a.
r :- b.
r :- c.
```

Here we should try the rule most likely to succeed first, because if it succeeds then we're done. And we should try second the rule second-most-likely to succeed. The same applies to the rules

```
r(X) :- a(X).
r(X) :- b(X).
r(X) :- c(X).
```

provided **r** is only queried with **X** bound. So we should order "or"s by *decreasing* probability of success, just the opposite of "and"s.

Some mathematics

We can give general mathematical formulas for this rearrangement problem. They apply to any queries or rules without local variables, without "side effect" predicates like **asserta** and **consult**, and for which the success probabilities of expressions are independent.

First, suppose we have some expressions "and"ed together, where each has a probability of success |p sub i| and an execution cost |c sub i| (measured in arbitrary nonnegative units, perhaps time, perhaps the number of database queries made). Number the expressions from left to right. Consider two adjacent ones |i| and |i+1|. Interchanging them will improve computation cost if

```
c sub i   +   p sub i c sub i+1   >   c sub i+1   +   p sub i+1 c sub i
```

Manipulating the inequality we get:

```
( 1 - p sub i+1 ) c sub i   >   ( 1 - p sub i ) c sub i+1
```

or:

```
( 1 - p sub i ) / c sub i   <   ( 1 - p sub i+1 ) / c sub i+1
```

In other words, we can use the ratio of probability of failure to the execution cost as a sorting criterion for adjacent expressions. Since repeated interchanges of adjacent expresssions are sufficient to sort a list (this is how the famous "bubble sort" sorting algorithm works), we should sort the expressions by this criterion to get the best order in an "and".

A similar formula applies to "or"s. We should interchange adjacent expressions |i| and |i+1| in an "or" if

```
c sub i + ( 1 - p sub i ) c sub i+1   >   c sub i+1
   +   ( 1 - p sub i+1 ) c sub i
```

or:

```
p sub i / c sub i   <   p sub i+1 / c sub i+1
```

Here's an example. Take the query

```
?- a, b.
```

where **a** has probability 0.6 of success, and **b** has probability 0.8. If it costs the same amount of time to query either, **a** should be first. We can see this with the "and" formula because

```
( 1 - 0.6 ) / 1   >   ( 1 - 0.8 ) / 1   ,     "or"     0.4 > 0.2
```

But suppose (without changing its probability) that **a** is defined as

```
a :- c, d.
```

where **c** and **d** are only expressible as facts. Now it requires more work to find the truth of **a**: a query of **a**, followed by queries of **c** and **d**. If each query costs about the same, that's three times as much work. So now

```
(1 - 0.6 ) / 3   <   ( 1 - 0.8 ) / 1   ,     "since"     0.1333   "<"   0.2
```

and **b** is better to put first in the original query.

Rearranging queries with local variables

Local variables cause problems for the preceding reasoning. Consider:

```
?- a(X), b(X).
```

Here **X** is local to the query, and will be bound in **a**; in queries like these, **a** isn't expected to fail. But if we try to rearrange the query to

```
?- b(X), a(X).
```

Now **b** is the predicate that binds **X**, and it becomes unlikely to fail. So predicate expressions that bind local variables to values must be analyzed specially.

For certain expressions we can ignore this problem, those that require bound variable values. Consider:

```
?- a(X), b(Y), X>Y.
?- a(X), Y is X*X.
?- a(X), square(X,Y).
```

where **square** in the last query is defined by

```
square(X,Y) :- Y is X*X.
```

In each of these queries, putting last predicate expression first will cause an error message about trying to refer to an unbound variable. Things are almost as bad with

```
?- a(X), b(Y), not(c(X,Y)).
```

which, while not giving an error message, gives completely different answers when the last expression is put first (see a similar use in the **done** predicate for forward chaining in Chapter 7):

```
?- not(c(X,Y)), a(X), b(Y).
```

The first predicate expression here asks whether there are any two-argument **c** facts, and doesn't connect its **X** and **Y** to those in the **a** and **b** predicates.

But assuming we *can* move expressions to a front binding position, which one should we put there? Consider

```
?- a(X), b(X).
```

Suppose there are only a finite number of **X** values satisfying this query, say 5. Suppose predicate **a** is satisfied by 20 values of **X**, and predicate **b** by 100 values. If we put first **a**, then 5 of its 20 solutions will subsequently satisfy **b**. But if we put first **b**, then 5 of its 100 solutions will subsequently satisfy **a**. So if **X** values occur in a random way, we'll need more backtracking to answer the query the second way than the first. The amount of backtracking is the important criterion, because probabilities and costs don't apply here. Probabilities aren't relevant because we don't usually expect a variable-binding predicate to fail, and costs aren't relevant because binding a variable doesn't cost much.

To summarize: when you have several predicate expressions in an "and" that could bind a variable were they moved to the left in a query, move the one with the fewest possible bindings. Then sort the remaining expressions by increasing probability of success, as before.

Rearranging queries based on dependencies

Besides the preceding criteria, we should also use *dependencies* as criteria for rearranging queries. A dependency holds between two predicate expressions if they share a variable. Generally speaking, you should try to maximize the number of dependencies between adjacent expressions in an "and", while staying consistent with the previous criteria. This is because when two adjacent expressions don't have variables in common, the variable bindings in the first are irrelevant to the success or failure of the second, and backtracking may be wasting its time going to the second when a failure is due to the first.

For example, consider

```
?- a(X), b(Y), c(X), e(X,Y), d(X), f(Y), g(Y).
```

Here the **b** expression has no dependency on the **a** expression, the **c** none on the **b**, and the **f** none on the **d**, though the other adjacent expressions do have dependencies. A better rearrangement would be (assuming that this has about the same average overall cost due to expression probabilities and costs)

```
?- a(X), c(X), d(X), b(Y), e(X,Y), f(Y), g(Y).
```

where there is only one non-dependency between adjacent expressions, between **d** and **b**. Now when **g** fails, backtracking only needs to go back three expressions, whereas before it needed to go back five.

Summary of guidelines for optimal query arrangements

We can summarize what we have concluded about query and rule rearrangement by the following guidelines. (They're like heuristics because they can be wrong.)

1. For an "and" that does not bind local variables, sort the predicate expressions by decreasing values of the ratio of the failure probability for the expression to its execution cost.

2. For an "or" or for rules with the same left side (an implicit "or"), sort them by decreasing values of the ratio of the success probability for the rule to its execution cost.

3. For an "and" that binds local variables, choose to make binding for each variable the predicate expression that mentions it, among those that can bind it, both legally and without changing the predicate meaning, and that has the fewest database matches when queried alone. Then rearrange the rest of the expressions by guidelines 1 and 3 while freezing the position of these binding expressions.

4. Maximize the number of dependencies between adjacent expressions in an "and", as long as you don't violate the binding choices made according to guideline 3, or significantly violate the probability-cost ratio ordering in guideline 1.

Let's apply these guidelines to the scheduling example of Section 13.1. It originally looked like this:

```
classtime(T1), classtime(T2), classtime(T3),
  classtime(T4), classtime(T5),
  not(occupied(T1)), not(occupied(T2)), not(occupied(T3)),
  not(occupied(T4)), not(occupied(T5)),
  not(two_consecutive_hours([T1,T2,T3,T4,T5]),
  not(three_classes_same_day([T1,T2,T3,T4,T5]).
```

For guideline 3 the question is which expressions should bind variables **T1**, **T2**, **T3**, **T4**, and **T5**. But every expression besides the **classtime** ones has a **not** in front of it, so this is the only arrangement that will satisfy the conditions of guideline 3.

To better satisfy guideline 4, we can move the **not** expressions to immediately after the expressions binding them.

```
classtime(T1), not(occupied(T1)),
classtime(T2), not(occupied(T2)),
classtime(T3), not(occupied(T3)),
classtime(T4), not(occupied(T4)),
classtime(T5), not(occupied(T5)),
```

```
not(two_consecutive_hours([T1,T2,T3,T4,T5]),
not(three_classes_same_day([T1,T2,T3,T4,T5]).
```

It's hard to guess success probabilities here--this varies enormously with the number of **occupied** facts we have--so we just use computation cost to order by guideline 1. So we should keep the last two expressions last because they both involve computation (calls to rule definitions), and the other **not** expressions don't. Similarly, **two_consecutive_hours** should precede **three_classes_same_day** because the first has three expressions in its definition, the second six.

Rearrangement and improvement of the photo interpretation query

The photo interpretation query of Section 13.1 requires more care to rearrange. Here it is:

```
?- label(R1), label(R2), label(R3), label(R4), label(R5),
  borders(R1,R2), borders(R2,R4), inside(R3,R2), inside(R5,R4)
  large(R1), large(R2), large(R3), large(R4),
  regular(R3), regular(R5), irregular(R2), irregular(R1).
```

All five variables (**R1**, **R2**, **R3**, **R4**, and **R5**) are bound locally in this query. So apply guideline 3 to decide which expressions to put first. The definitions of **borders**, **inside**, and **large** all have **not**s containing variables that would become unbound if those expressions were put first in the query. So the only expressions that can bind variables are those for **label**, **regular**, and **irregular**. Among them, **irregular** is best by guideline 3 because it has only two argument possibilities, **regular** is second best with three possibilities, and **label** is third with five (see Figure 13-3). That suggests this rearrangement:

```
?- irregular(R2), irregular(R1), regular(R3), regular(R5),
  label(R1), label(R2), label(R3), label(R4), label(R5),
  borders(R1,R2), borders(R2,R4), inside(R3,R2), inside(R5,R4)
  large(R1), large(R2), large(R3), large(R4).
```

But now **label** can never fail for **R2**, **R1**, **R3**, and **R5** (see Figure 13-3); any value bound in the first line will satisfy the **label** predicate. So we can eliminate four predicate expressions to give the equivalent shorter query:

```
?- irregular(R2), irregular(R1), regular(R3), regular(R5),
  label(R4), borders(R1,R2), borders(R2,R4), inside(R3,R2),
  inside(R5,R4) large(R1), large(R2), large(R3), large(R4).
```

(We must still keep **label(R4)** because there's no **regular** or **irregular** fact to bind **R4**.)

So now the first line does all the variable binding. Let's now apply guideline 1 (the probability-to-cost ratio ordering) to the rest of the query, as suggested in the discussion of guideline 3. To simplify analysis, we'll assume all labels are equally likely, and we'll measure processing cost by the number of queries needed. Predicate **large** (see Figure 13-3) succeeds except if its argument is **vehicle**, so it then succeeds with probability 0.8; it requires two query lookups. This means a guideline-1 ratio of $|(1 - 0.8) / 2 = 0.1|$. Predicates **borders** and **inside** require more complicated analysis, however. Here again are their definitions:

```
borders(A1,A2) :- not(A1=A2), not(water_constraint(A1,A2)),
  not(vehicle_constraint(A1,A2)).
inside(A1,A2) :- not(A1=A2), not(water_constraint(A1,A2)),
  not(vehicle_constraint2(A1,A2)), not(grass_constraint(A1,A2)),
  not(A1=pavement).
```

Counting both left and right sides of rules, **borders** requires four query lookups for its four expressions, and **inside** requires six. (We'll ignore the added complication of lowered cost due to incomplete rule processing for some rule failures.) From Figure 13-4, we can estimate the probability of success of **borders** as |12 / 25 = 0.48|, and the probability of success of **inside** as |7 / 25 = 0.28|. Then the ratios needed for guideline 1 are |(1 - 0.48) / 4 = 0.13| for **borders** and |(1 - 0.28) / 6 = 0.18| for **inside**. By these rough numbers, **inside** expressions should come first, then **borders** expressions, and then **large** expressions. So the query should be rearranged to:

```
?- irregular(R2), irregular(R1), regular(R3), regular(R5),
  label(R4), inside(R3,R2), inside(R5,R4), borders(R1,R2),
  borders(R2,R4), large(R1), large(R2), large(R3), large(R4).
```

Studying Figure 13-3, we see that **large** succeeds whenever **irregular** succeeds. So we can eliminate the **large(R1)** and **large(R2)** to get an equivalent query:

```
?- irregular(R2), irregular(R1), regular(R3), regular(R5),
  label(R4), inside(R3,R2), inside(R5,R4), borders(R1,R2),
  borders(R2,R4), large(R3), large(R4).
```

Now we can apply guideline 4, trying to group expressions mentioning the same variable together. One way is to move the **inside**, **borders**, and **large** expressions back into the first line, preserving their order at the expense of the order of the first line (since guideline 1 order is more important than guideline 3 order):

```
?- irregular(R2), regular(R3), inside(R3,R2),
  regular(R5), label(R4), inside(R5,R4),
  irregular(R1), borders(R1,R2), borders(R2,R4),
  large(R4), large(R3).
```

Here we've had to move **irregular(R1)** considerably to the right, but that's the only expression moved significantly right.

So now we've probably got a much faster query. We emphasize "probably", since we had to make a number of simplifying assumptions about probabilities and costs. (If these assumptions seemed too much, we can always do a more careful analysis using conditional probabilities and more subcases.)

Dependency-based backtracking

Our second technique for handling long queries is *dependency-based backtracking*. Consider the query:

```
?- a(X), b(X,Y), c(Z), d(X,Z), e(X).
```

Suppose the probability of any predicate expression succeeding is 0.1, and that these probabilities are independent of the success or failure of other expressions. On the average we'll need to generate 10 **X** values before we find one that works, 10 **Y** values, and 10 **Z** values. But for each **X** value we must try about 10 **Y** values, for about 100 **Y** values in all; and for each **Y** value we must try about 10 **Z** values, for about 1000 about **Z** values in all. So there will be a lot of backtracking to answer this query with Prolog's standard backward-chaining control structure.

An alternative is *nonchronological* or *dependency-based* backtracking. That is, backing up to the expression that did the last variable binding that could have affected the failure, and finding a new binding there. That's

not necessarily the previous expression, so dependency-based backtracking is "smarter" backtracking, backtracking that notices and takes obvious shortcuts. Dependencies (indications of which expressions have common variables) help this considerably, so that's why they're in the name of the technique.

Let's apply dependency-based backtracking to the previous query. First, we need to tabulate the predicate expressions in the query that bind, and what expressions have dependencies on what other expressions; see Figure 13-5. Assume as a database for this query:

```
a(1).
a(2).
a(3).
b(A,B) :- B is 3*A.
c(4).
c(1).
d(A,B) :- A>B.
e(3).
```

Then what happens in executing the query with dependency-based backtracking is shown in Figure 13-6. Notice that it's identical to regular Prolog backtracking except at three places, after steps 7, 14, and 16:

> --After step 7, we backtrack from the third predicate expression directly to the first, since variable **X** must be responsible for the earlier failure of the fourth expression, and variable **Y** has nothing to do with that failure.
>
> --After step 14, we backtrack from the last expression directly to the first, since variable **X** is the only possible cause of the failure, and the first expression binds it.
>
> --After step 16, we skip over the third expression in moving back to the right, since it didn't fail and it doesn't include any variables whose values were changed since the last time it was visited.

So the effect of dependency-based backtracking is to save us 6 actions (right and left movements) out of 24 total. That's not much, but other queries show more dramatic savings.

Here's an algorithm for dependency-based backtracking:

> 1. Mark every predicate expression in the query as "active". Set P to the leftmost predicate in the query.
>
> 2. Execute repeatedly until you reach the right end of the query, or backtrack off the left end:
>
> > (a) Execute expression P, and mark it as "inactive".
> >
> > (b) If P succeeds and does not bind variables, set P to the next expression to the right that is marked "active".
> >
> > (c) If P succeeds and binds variables, mark as "active" all other query expressions to the right containing those variables if (1) they are not so marked already and (2) the values are different from the previous values of those variables. Set P to the next expression to the right that is marked "active".

(d) If P fails, mark as "active" all the expressions to the left of P that bind variables mentioned in P. Set P to the first expression to the left that is currently marked "active" (which is not necessarily one just marked, for it may have been marked previously).

There are further improvements on dependency-based backtracking that we don't have space for here. The theory is quite elegant, and this is an active area of research, also called *truth maintenance*.

Reasoning about possibilities

There's a third way to efficiently handle long queries. The idea is to abandon considering variable bindings one at a time, and to reason about possibilities for each variable. Consider the query of the last section:

```
?- a(X), b(X,Y), c(Z), d(X,Z), e(X).
```

Suppose there are only a finite number of ways to satisfy each predicate expression. We could figure out all the possible values for **X** that satisfy **a(X)**, and then check which of them also satisfy e(X). Then for each remaining choice we could check which could satisfy **b(X,Y)**, crossing out ones that can't. Then we could figure out the possible values of **Z** that satisfy **c(Z)**, and which of those **Z** values for which there is a corresponding **X** value remaining that can satisfy **d(X,Z)**. Sherlock Holmes might put the approach this way: when you have eliminated the impossible, whatever remains, however improbable, must include the truth.

Why reason about possibilities for variables? So we don't need to work as hard to answer the query. Early elimination of impossible values for variables reduces fruitless backtracking. Sometimes we can even reduce possibilities to one for each variable, and we don't need to do any further work. The idea is called *relaxation*--a technical term that has nothing to do with vacations on beaches. Artificial intelligence uses in particular *discrete relaxation*, relaxation involving variables with finitely-long possibility lists.

Using relaxation for the photo interpretation example

We'll present an algorithm for relaxation in the next section. It's a little complicated. To work our way up to it, let's see how we can reason about possibility lists for the photo interpretation problem of Section 13.1, the one with the query:

```
?- label(R1), label(R2), label(R3), label(R4), label(R5),
  borders(R1,R2), borders(R2,R4), inside(R3,R2), inside(R5,R4),
  large(R1), large(R2), large(R3), large(R4), regular(R3),
  regular(R5), irregular(R2), irregular(R1).
```

To get possibility lists for each variable, we must extract the single-variable expressions referring to each variable:

```
variable R1: label(R1), large(R1), irregular(R1).
variable R2: label(R2), large(R2), irregular(R2).
variable R3: label(R3), large(R3), regular(R3).
variable R4: label(R4), large(R4).
variable R5: label(R5), regular(R5).
```

Reexamining Figure 13-3, we can see what values (labels) are consistent with the expressions. The possible labels are grass, water, pavement, house, and vehicle. The **large** predicate rules out the vehicle label. The

regular predicate specifies house, vehicle, or pavement, and **irregular** specifies grass or water. So the initial possibility lists are:

```
variable R1: grass or water
variable R2: grass or water
variable R3: house or pavement
variable R4: house or pavement or grass or water
variable R5: house or vehicle or pavement
```

Now let's eliminate or "cross out" possibilities using the remaining multivariable predicate expressions (constraints) of the original query:

```
borders(R1,R2), borders(R2,R4), inside(R3,R2), inside(R5,R4)
```

We can pick a variable and a possibility, and check if the possibility works. Suppose we pick **R1** and **grass**. The **borders(R1,R2)** mentions **R1**, so we see if some **R2** value will work with **R1=grass**--that is, whether there's some label that could border grass. Yes, **R2=water** works. Check the definition of **borders**:

```
borders(A1,A2) :- not(A1=A2), not(water_constraint(A1,A2)),
  not(vehicle_constraint(A1,A2)).
```

For **A1=grass** and **A2=water**, **A1** is not equal to **A2**. The **water_constraint** only applies when one label is a house or vehicle, as you can see from its definition:

```
water_constraint(water,house).
water_constraint(water,vehicle).
water_constraint(house,water).
water_constraint(vehicle,water).
```

And **vehicle_constraint** only applies when **A1** or **A2** is a vehicle:

```
vehicle_constraint(A1,vehicle) :- not(A1=pavement).
vehicle_constraint(vehicle,A2) :- not(A2=pavement).
```

So **R1=grass** will work for at least one value of **R2**, which is all we need to keep us from crossing it out.

But if we examine a different variable and value, things can be different. Suppose we pick the **R2** variable and **water**. The **inside(R3,R2)** constraint mentions **R2**. But there's no way to label **R3** to satisfy the second expression on the right side of the **inside** definition. Here's that definition:

```
inside(A1,A2) :- not(A1=A2), not(water_constraint(A1,A2)),
  not(vehicle_constraint2(A1,A2)), not(grass_constraint(A1,A2)),
  not(A1=pavement).
```

The only possibilities for **R3** from the single-variable constraints are house and vehicle, and water_constraint succeeds for both. Hence the **not** fails, and **inside** fails too because this is the only rule for **inside**. In other words, a vehicle or house can't be inside water, a reasonable assumption for most pictures, though you can probably imagine a few exceptions. So **R2** cannot be water. The only remaining possibility is grass, so **R2** must be grass.

Once we've eliminated possibilities for a variable, we can often eliminate possibilities for a variable appearing in constraints with it. Consider variable **R1** which we didn't have any success with originally. Now it's no longer possible that **R1** is grass, because then **R1** would be the same as **R2**, and the first condition in

the **borders** definition prohibits that. It says two adjacent regions can't have the same label since we couldn't then see the boundary. There's only one remaining label for **R1**, **water**, so **R1** must be that.

Our possibility lists are now:

```
variable R1: water
variable R2: grass
variable R3: house or pavement
variable R4: house or pavement or grass or water
variable R5: house or vehicle or pavement
```

Now consider **R3**. It can't be pavement because **R3** is inside **R2**, and the last condition in the definition of **inside** says that the thing inside can't be pavement. That is, pavement must connect to other pavement--a reasonable, though sometimes false, assumption. So **R3** must be a house.

Consider **R4**. It can't be grass because it borders **R2**, and **R2** is already grass. It can't be water because water can't contain (as **R5**) a house or a vehicle by **water_constraint**, and water can't contain pavement by the last condition in the **inside** definition. That leaves house or pavement for **R4**.

Now consider **R5**. It can't be pavement because pavement can't be contained in anything. So it must be a house or vehicle.

Now consider **R4** again. Suppose it's a house. Then **R5** can't be a house too because that would make two adjacent regions with the same label. But **R5** couldn't be a vehicle either, because a vehicle can't be inside a house by the **vehicle_constraint2** definition. So **R4** can't be a house, and must be pavement.

No further eliminations can be done, so the final possibility lists are:

```
variable R1: water
variable R2: grass
variable R3: house
variable R4: pavement
variable R5: house or vehicle
```

Quantifying the effect (*)

To show the savings in considering possibility lists instead of individual values, let's use some numbers. Suppose there are |n sub Y| possibilities for a variable **Y** . Suppose the expression **d(X,Y)** is satisfied by fraction p of all possible **XY** pairs. If we can assume satisfaction pairs for d are randomly distributed over all **XY** pairs, then the probability that some **X** value r has at least one corresponding **Y** that satisfies d can be approximated by the binomial series:

```
1 - ( 1 - p ) sup { n sub Y }  =    { n sub Y } p   -
{ n sub Y } ( { n sub Y }  - 1 ) p sup 2 / 2   +
{ n sub Y } ( { n sub Y } - 1 ) ( { n sub Y }  - 2 ) p sup 3 / 6   -   . . .
```

And the expected number of such matched **X** values in the |n sub X| values for **X** will be just |n sub X| times this. Figure 13-7 shows the probability computed for sample values of p and n. When p is small (as a rule of thumb, whenever p < 0.2) we can approximate the binomial distribution by the Poisson distribution, and the probability is

```
1 - e sup { - n sub Y p }
```

And the expected number of **X** values remaining from |n sub X| originally, after examining predicate **d(X,Y)** which has probability of success p and |n sub Y| values of **Y** that can match it is

```
n sub X ( 1 - e sup { - n sub Y p } )
```

The smaller this number, the better is reasoning about possibilities first before working Prolog-style.

If several constraints mention a variable, each can independently eliminate possibilities, further reducing their number. Consider this query:

```
?- a(R), b(S), c(T), d(R,S), e(S,T).
```

Suppose there are 10 ways to satisfy predicate **a**, 20 ways to satisfy **b**, and 50 ways to satisfy **c**. Suppose further that the probability is 0.1 that a random pair of values satisfy **d**, and 0.01 that a random pair of values will satisfy **e**, and that probabilities are all independent. Then:

> --From Figure 13-7 and our formula we can expect |10 * 0.88 = 8.8| possibilities for **R** that satisfy **d**, on the average.
>
> --And we can expect |50 * 0.39 = 19.5| possibilities for **T** that satisfy **e**, on the average.
>
> --**S** occurs in both the two-argument predicates **d** and **e**. For the occurrence in **d** we can expect |20 * (1 - e sup {-8.8 * 0.1}) = 11.7| possibilities remaining.
>
> --The predicate **e** can only remove further **S** possibilities of these. If satisfaction of **e** is probabilistically independent of satisfaction of **d**, for instance, we expect |11.7 * (1 - e sup {-19.5 * 0.01}) = 2.1| possibilities remaining for **S**. That's a lot better than the initial 20.

Formalization of pure relaxation

Here's an algorithm for the simplest kind of relaxation method for answering many-constraint queries: *pure* relaxation. Note for efficiency in this algorithm that it's important to index all variable names appearing in constraint expressions.

> 1. Create possibility lists for each variable using all single-variable query predicate expressions. (Single-argument expressions with a variable argument are single-variable, but others like **p(X,a,X)** can be too.) In Prolog, **bagof** will help do this.
>
> 2. Mark each variable in the original query as "active".
>
> 3. Do the following step repeatedly until no more "active" variables remain.
>
> > (a) Choose some "active" variable A. (Heuristics may help choose, like "focus-of-attention" heuristics.) For each current possible value V for A:
> >
> > > For each constraint C mentioning variable A:
> > >
> > > > Check to see if when variable A has value V, constraint C

can be satisfied some way (binding other variables of C as necessary). If not, remove V from the possibility list for A; otherwise do nothing. Throw away previous bindings for other constraints when considering each new constraint (this makes the relaxation "pure").

(b) Mark variable A as "inactive", and mark as "active" all variables (i) mentioned in constraints with A, and (ii) marked "inactive", and (iii) having more than one possibility remaining. Eliminate any constraints for which all the variables have unique values now.

4. If there is now a unique possibility for each variable, stop and give those values as the query answer. Otherwise, run the Prolog interpreter on the original query, drawing values from the possibility lists.

We emphasize that for every constraint we consider, we ignore bindings of its variables made when those variables occurred in other constraints. This "pure" relaxation idea may seem wasteful, but if we didn't do it we would just be doing Prolog-style backward chaining in disguise. But compromises between the two approaches are possible, like the *double relaxation* we'll show later.

One nice thing about this relaxation algorithm is its good potential for concurrency. We can work in parallel on different constraints, different values, or even different variables.

Another relaxation example: cryptarithmetic

No artificial intelligence book would be complete without an analysis of a good puzzle problem. Analysis of "brain-teaser" puzzles stimulated much of the early development of the field.

A classic application of relaxation is to puzzles in which numbers are represented by letters (*cryptarithmetic*). Here the variables are the letters and the possibility lists are numbers those letters can be (often, the possibilities are restricted to single digits). Constraints come from arithmetic relationships between the digits. Here's a famous example, where every letter stands for a different digit, and every occurrence of a letter is the same digit.

```
    S E N D
+ M O R E
_____________________
M O N E Y
```

Note that's the letter "O", not the number zero. To make the problem easier, suppose we are told that E=5. (Otherwise we can't solve it completely with pure relaxation.)

So from the single-variable constraints, initial possibility lists for this problem are:

```
M: [1,2,3,4,5,6,7,8,9]
S: [1,2,3,4,5,6,7,8,9]
O: [0,1,2,3,4,5,6,7,8,9]
E: [5]
N: [0,1,2,3,4,5,6,7,8,9]
R: [0,1,2,3,4,5,6,7,8,9]
```

```
D: [0,1,2,3,4,5,6,7,8,9]
Y: [0,1,2,3,4,5,6,7,8,9]
```

Each letter represents a variable for this problem. Note M and S cannot be 0 because they are the first digits in numbers, and numbers don't start with 0. On the other hand, we can't cross out the 5's yet because we need to use a multi-variable constraint for them.

But there are also some important additional variables buried in the problem: the carries from each column. We'll label these C1, C10, and C100, representing the carries out of the 1's column, the 10's column, and the 100's column respectively. Their possibilities for two-number addition are:

```
C1: [0,1]
C10: [0,1]
C100: [0,1]
```

Here are the constraints on the variables. The first four come from the rules of column addition, and the last says that digit assignments must be unique. While these constraints could also be satisfied for integers greater than 10 or real numbers, the initial possibility lists will prevent such happenings.

Con1: D + E = Y + (10 * C1)
Con2: N + R + C1 = E + (10 * C10)
Con3: E + O + C10 = N + (10 * C100)
Con4: S + M + C100 = O + (10 * M)
Con5: all_different_numbers(S,E,N,D,M,O,R,Y)

Since E=5 we can substitute that value into the constraints to simplify them:

Con1: D + 5 = Y + (10 * C1)
Con2: N + R + C1 = 5 + (10 * C10)
Con3: 5 + O + C10 = N + (10 * C100)
Con4: S + M + C100 = O + (10 * M)
Con5: all_different_numbers(S,5,N,D,M,O,R,Y)

That covers step 1 of the relaxation algorithm, so we proceed to step 2 and mark every letter and carry as "active". Now we start step 3, the big step. We will follow a "leftmost-first" heuristic that says to choose to work on the active variable representing the digit or carry farthest left in the sum display.

1. We can pick either M or C1000 as the first variable according to our heuristic; suppose we pick M. Only constraints Con4 and Con5 mention M. And the only value of M satisfying Con4 is 1, since 0 is not on M's original possibility list, and a value for M of 2 or greater would make the right side of Con4 at least 20, and there's no combination of S and C100 drawn from their possibility lists whose sum plus 2 (for M=2) could equal or surpass 20. We mark M inactive.

2. Using our "leftmost active" choice heuristic, we pick S next. It's only mentioned in Con4 and Con5. The only way to satisfy Con4 is if S is 8 or 9, because M is 1 and the right side is at least 10. Con5 doesn't further eliminate possibilities for S. We mark S inactive.

3. We pick O next. It's mentioned in Con3, Con4, and Con5. We can't do anything with Con3, but by Con4 we see O must be 0 or 1 since M=1 and S is either 8 or 9. But turning now to Con5,

O cannot be 1 (with M already 1). So O=0. We mark S active and O inactive.

4. We pick S next. We can't conclude anything about it, so we mark it inactive. (Nothing is marked active because we didn't rule out any possibilities.)

5. We pick C100 next. It's mentioned in Con3 and Con4. It must be 0, since |5 + 0 + C10 < 7|. We mark S active, and C100 inactive.

6. We pick S next. Now by Con4 it must be 9, since |9 + 1 + 0 = 0 + (10 * 1)|. S is marked inactive.

Current possibility lists and activity markings are:

```
M: [1] (inactive)
S: [9] (inactive)
O: [0] (inactive)
E: [5] (inactive)
N: [0,1,2,3,4,5,6,7,8,9] (active)
R: [0,1,2,3,4,5,6,7,8,9] (active)
D: [0,1,2,3,4,5,6,7,8,9] (active)
Y: [0,1,2,3,4,5,6,7,8,9] (active)
C1: [0,1] (active)
C10: [0,1] (active)
C100: [0] (inactive)
```

And the sum looks like:

```
    9 5 N D
+ 1 0 R 5
-----------
1 0 N 5 Y
```

And the constraints are:

> **Con1: D + 5 = Y + (10 * C1)**
> **Con2: N + R + C1 = 5 + (10 * C10)**
> **Con3: 5 + C10 = N**
> **Con4: 10 = 10**
> **Con5: all_different_numbers(9,5,N,D,1,0,R,Y)**

We can just ignore Con4 from now on because it no longer has variables.

7. Pick N next. It's mentioned in Con2, Con3, and Con5. Con2 doesn't help, but Con3 says N must be 5 or 6. And then Con5 says N can't be 5 since E is already, so N=6. Mark N inactive.

8. Pick C10. It's mentioned in Con2 and Con3. But for Con3 it must be 1 since N=6. Mark C10 inactive.

9. Pick R. It's mentioned in Con2 and Con5. In Con2, |6 + R + C1 = 5 + (10 * 1)|, so R is 8 or 9. But by Con5, R cannot be 9 because S is already 9. So R=8. Mark R inactive.

10. Pick C1. It's mentioned in Con1 and Con2. By Con2, it must be 1. Mark C1 inactive.

11. Pick D. It's mentioned in Con1 and Con5. By Con1, |D + 5 = Y + (10 * 1)|. This means |D = Y + 5|, so D can only be 5, 6, 7, 8, or 9. But by Con5, the values 5, 6, 8, and 9 are ruled out because variables already have those values. So D=7. Mark D inactive.

12. Pick Y. By Con1, it must be 2. Mark Y inactive. Now every variable is marked inactive, so we can stop.

The final solution is:

```
M=1, S=9, O=0, E=5, N=6, R=8, D=7, Y=2, C1=1, C10=1, C100=0


    9 5 6 7
+ 1 0 8 5
_________________
 1 0 6 5 2
```

Notice relaxation is much like iterative methods for solving mathematical equations: "convergence" or progress towards the solution may be slow at times, but things speed up as the solution approaches.

Implementation of pure relaxation (*)

We can implement relaxation as we implemented the chaining programs (Chapter 7) and search programs (Chapter 10): with a problem-independent file and a problem-dependent file. The last part must specify the constraints. In it we'll use the following definitions (as either facts or rules):

choices(<variable_name>,<possible_value>)--states initial possibilities for <variable_name>, incorporating all the single-variable constraints

constraint(<type>,<argument_list>)--states multivariable constraints, classifying by type and arguments

satisfiable(<type>,<argument_list>,<bindings_found>)--determines whether a multi-variable constraint of a given type is actually satisfiable by some set of variable bindings

To circumvent the Prolog interpreter's usual binding of variables, "variable names" here must actually be constants (written in lower case). Note that because constraints have a type argument, all constraints of the same type can be checked by a single satisfiable predicate definition. Another reason for having both constraint and satisfiable is that relaxation can notice common variables between constraints without having to test constraints.

Here's an example problem-dependent file, the definitions for the photo interpretation problem considered earlier. Much of the code remains the same as before.

```
choices(R,CL) :- member(R,[r1,r2,r3,r4,r5]),
  bagof(C,choice(R,C),CL).
```

```
choice(R,grass) :- irregular(R).
choice(R,water) :- irregular(R).
choice(R,pavement) :- regular(R).
choice(R,house) :- regular(R).
choice(R,vehicle) :- regular(R), not(large(R)).
choice(R,L) :- not(regular(R)), not(irregular(R)), label(L).


label(grass).
label(water).
label(pavement).
label(house).
label(vehicle).


large(r1).
large(r2).
large(r3).
large(r4).


regular(r3).
regular(r5).


irregular(r2).
irregular(r1).

constraint(borders,[r1,r2]).
constraint(borders,[r2,r4]).
constraint(inside,[r3,r2]).
constraint(inside,[r5,r4]).


satisfiable(borders,Args,[A1,A2]) :- some_bindings(Args,[A1,A2]),
  not(A1=A2), not(water_constraint(A1,A2)),
  not(vehicle_constraint(A1,A2)).
satisfiable(inside,Args,[A1,A2]) :- some_bindings(Args,[A1,A2]),
  not(A1=A2), not(water_constraint(A1,A2)),
  not(vehicle_constraint2(A1,A2)),
  not(grass_constraint(A1,A2)), not(A1=pavement).


water_constraint(water,house).
water_constraint(water,vehicle).
water_constraint(house,water).
water_constraint(vehicle,water).


vehicle_constraint(A1,vehicle) :- not(A1=pavement).
vehicle_constraint(vehicle,A2) :- not(A2=pavement).
vehicle_constraint2(A1,vehicle).
vehicle_constraint2(vehicle,A2) :- not(A2=pavement).


grass_constraint(grass,house).
grass_constraint(grass,vehicle).
```

The problem-independent code translates the algorithm given in Section 13.12. Figure 13-8 gives the predicate hierarchy. The top-level predicate is relax of no arguments. It firsts converts all choices definitions into possibility lists. It then repeatedly chooses an active variable, until no more such variables exist. For that variable, it retrieves the possibility list and checks each possibility in turn, making a list of those possibilities that are satisfiable, using the bagof predicate defined in Section 10.6. If any previous possibilities are now impossible (meaning that the possibility list is shortened), it marks all variables in constraints with this one as active, and makes this variable inactive. If only one possibility remains for the variable, then (predicate check_unique) that value is substituted into the constraints. Any constraints no longer having variables in them, just constants, are removed.

Checking of possibilities is done by possible_value and constraint_violation. The first generates possibilities, and the second substitutes them into the constraints and calls satisfiable to check for satisfiability. Only constraints containing the variable under study are substituted into. Note it's important we check constraint violation and not constraint satisfaction; as soon as we find a constraint that can't be satisfied, we can stop and cross out a possibility.

Here's the program for pure relaxation:

```
/* Top-level routines */
relax :- setup, relax2.
relax :- listing(choice_list).


relax2 :- repeatactive, choice_list(O,CL), active(O),
  retract(active(O)), not(one_possibility(CL)),
  write('Studying variable '), write(O), nl,
  bagof(V,possible_value(O,CL,V),NCL), not(CL=NCL),
  retract(choice_list(O,CL)), asserta(choice_list(O,NCL)),
  write('Possibilities for '), write(O), write(' reduced to '),
  write(NCL), nl, update_activity(O), check_unique(O,NCL), fail.


/* Creation of the initial possibility lists */
setup :- abolish(choice_list,2), fail.
setup :- choices(O,VL), assertz(choice_list(O,VL)),
  assertz(active(O)), fail.
setup.


/* Analysis of a particular variable value */
possible_value(O,CL,V) :- member(V,CL),
  not(constraint_violation(V,O)).


constraint_violation(V,O) :- constraint(Type,Args), member(O,Args),
  substitute(O,V,Args,Args2), not(satisfiable(Type,Args2,Newargs)).


update_activity(O) :- constraint(T,A), member(O,A),
  member(O2,A), not(O2=O), choice_list(O2,L), not(active(O2)),
  not(one_possibility(L)), asserta(active(O2)), fail.
update_activity(O).
```

```
check_unique(O,[V]) :- constraint(Type,Args), member(O,Args),
  substitute(O,V,Args,Args2), retract(constraint(Type,Args)),
  some_var(Args2), asserta(constraint(Type,Args2)), fail.
check_unique(O,L).

/* Utility routines */
some_var([X|L]) :- choice_list(X,CL), !.
some_var([X|L]) :- some_var(L).


one_possibility([ZZZ]).


repeatactive.
repeatactive :- active(X), repeatactive.


member(X,[X|L]).
member(X,[Y|L]) :- member(X,L).


substitute(X,Y,[],[]).
substitute(X,Y,[X|L],[Y|L2]) :- substitute(X,Y,L,L2), !.
substitute(X,Y,[Z|L],[Z|L2]) :- substitute(X,Y,L,L2).


/* Note--the following is not necessary to run the program, but */
/* is provided as an aid to user definition of "satisfiable" */
some_bindings([],[]).
some_bindings([Arg|Args],[Substarg|Substargs]) :-
  choice_list(Arg,CL), !, member(Substarg,CL),
  substitute(Arg,Substarg,Args,Args2),
  some_bindings(Args2,Substargs).
some_bindings([Arg|Args],[Arg|Substargs]) :-
  some_bindings(Args,Substargs).
```

Here's the program working on our photo interpretation problem:

```
?- relax.
Studying variable r1
Studying variable r2
Possibilities for r2 reduced to [grass]
Studying variable r3
Possibilities for r3 reduced to [house]
Studying variable r4
Possibilities for r4 reduced to [pavement]
Studying variable r5
Possibilities for r5 reduced to [house,vehicle]
Studying variable r1
Possibilities for r1 reduced to [water]

choice_list(r1,[water]).
choice_list(r5,[house,vehicle]).
choice_list(r4,[pavement]).
choice_list(r3,[house]).
choice_list(r2,[grass]).
```

```
yes
```

Running a cryptarithmetic relaxation (*)

Here's a similar problem-dependent file defining the "send more money" cryptarithmetic problem:

```
choices(D,[0,1,2,3,4,5,6,7,8,9]) :- member(D,[n,d,r,o,y]).
choices(D,[1,2,3,4,5,6,7,8,9]) :- member(D,[s,m]).
choices(e,[5]).
choices(C,[0,1]) :- member(C,[c1,c10,c100]).


constraint(sum,[s,m,c100,o,m]).
constraint(sum,[e,o,c10,n,c100]).
constraint(sum,[n,r,c1,e,c10]).
constraint(sum,[d,e,0,y,c1]).
constraint(unique,[s,e,n,d,m,o,r,y]).


satisfiable(sum,L,[D1,D2,Carryin,Sum,Carryout]) :-
  some_bindings(L,[D1,D2,Carryin,Sum,Carryout]),
  S is D1 + D2 + Carryin,
  S2 is Sum + (Carryout * 10), S=S2.
satisfiable(unique,L,VL) :- not(duplication(L)),
  unique_values(L,VL,[]).


duplication([X|L]) :-  member(X,L), !.
duplication([X|L]) :- duplication(L).


unique_values([],[],KL).
unique_values([O|L],[V|L2],KL) :- choice_list(O,VL), !,
  unique_values(L,L2,KL), member(V,VL), not(member(V,L2)),
  not(member(V,KL)).
unique_values([K|L],[K|L2],KL) :- unique_values(L,L2,[K|KL]),
  not(member(K,L2)).
```

The satisfiable rule with sum as first argument checks column-sum constraints, and the satisfiable rule with unique checks unique-assignment constraints.

Here's a run. We've inserted extra blank lines for readability. Notice that the order of considering variables is different than for the problem analysis of Section 13.13, so possibility eliminations come in a different order.

```
?- relax.
Studying variable n
Possibilities for n reduced to [0,1,2,3,4,6,7,8,9]
Studying variable d
Possibilities for d reduced to [0,1,2,3,4,6,7,8,9]
Studying variable r
Possibilities for r reduced to [0,1,2,3,4,6,7,8,9]
Studying variable o
Possibilities for o reduced to [0,1]
Studying variable y
```

```
Possibilities for y reduced to [1,2,3,4,6,7,8,9]
Studying variable s
Possibilities for s reduced to [8,9]
Studying variable m
Possibilities for m reduced to [1]
Studying variable c1
Studying variable c10
Studying variable c100
Possibilities for c100 reduced to [0]

Studying variable s
Possibilities for s reduced to [9]
Studying variable y
Possibilities for y reduced to [2,3,4,6,7,8]
Studying variable o
Possibilities for o reduced to [0]
Studying variable r
Possibilities for r reduced to [2,3,4,6,7,8]
Studying variable d
Possibilities for d reduced to [2,3,7,8]
Studying variable n
Possibilities for n reduced to [6]
Studying variable c1
Possibilities for c1 reduced to [1]
Studying variable c10
Possibilities for c10 reduced to [1]

Studying variable d
Possibilities for d reduced to [7,8]
Studying variable r
Possibilities for r reduced to [8]
Studying variable y
Possibilities for y reduced to [2,3]
Studying variable d
Possibilities for d reduced to [7]
Studying variable y
Possibilities for y reduced to [2]

choice_list(y,[2]).
choice_list(d,[7]).
choice_list(r,[8]).
choice_list(c10,[1]).
choice_list(c1,[1]).
choice_list(n,[6]).
choice_list(o,[0]).
choice_list(s,[9]).
choice_list(c100,[0]).
choice_list(m,[1]).
choice_list(e,[5]).
```

Implementing double relaxation (*)

But pure relaxation can't solve the "send more money" problem unless we're told E=5 in advance. A better relaxation program could consider pairs of constraints together when it stops making progress analyzing single constraints. That is, a program could use the same bindings for variables common to

two constraints. This "double relaxation" will be slower than pure relaxation, since there are more pairs than singles and it's doing more work with each pair, so it should only be used as a last resort. Only when both methods fail should relaxation stop.

```
/* Top-level routines */
relax :- setup, relax2.


relax2 :- single_pred_relax.
relax2 :- done.
relax2 :- double_pred_relax.
relax2 :- active(X), relax2.
relax2 :- not(active(X)), listing(choice_list).


done :- not(possibilities_remain), listing(choice_list).


possibilities_remain :- choice_list(O,CL),
  not(one_possibility(CL)).


single_pred_relax :- write('Single predicate relaxation begun.'),
  nl, repeatactive, choice_list(O,CL), active(O),
  retract(active(O)), not(one_possibility(CL)),
  write('Studying variable '), write(O), nl,
  bagof(V,possible_value(O,CL,V),NCL), not(CL=NCL),
  new_possibilities(O,CL,NCL), fail.


double_pred_relax :- write('Double predicate relaxation begun.'),
  nl, choice_list(O,CL), not(one_possibility(CL)),
  write('Studying variable '), write(O), nl,
  bagof(V,possible_value2(O,CL,V), NCL),
  not(CL=NCL), new_possibilities(O,CL,NCL), fail.


new_possibilities(O,CL,NCL) :- retract(choice_list(O,CL)),
  asserta(choice_list(O,NCL)), write('Possibilities for '),
  write(O), write(' reduced to '), write(NCL), nl,
  update_activity(O), check_unique(O,NCL).


/* Creation of the initial possibility lists */
setup :- abolish(choice_list,2), fail.
setup :- choices(O,VL), assertz(choice_list(O,VL)),
  assertz(active(O)), fail.
setup.


/* Single-relaxation analysis of a particular value for a particular variable */
possible_value(O,CL,V) :- member(V,CL),
  not(constraint_violation(V,O)).


constraint_violation(V,O) :- constraint(Type,Args),
  member(O,Args), substitute(O,V,Args,Args2),
```

```
  not(satisfiable(Type,Args2,Newargs)).


update_activity(O) :- constraint(T,A), member(O,A), member(O2,A),
  not(O2=O), choice_list(O2,L), not(active(O2)),
  not(one_possibility(L)), asserta(active(O2)), fail.
update_activity(O).


check_unique(O,[V]) :- constraint(Type,Args), member(O,Args),
  substitute(O,V,Args,Args2), retract(constraint(Type,Args)),
  some_var(Args2), asserta(constraint(Type,Args2)), fail.
check_unique(O,L).


some_var([X|L]) :- choice_list(X,CL), !.
some_var([X|L]) :- some_var(L).


one_possibility([ZZZ]).

/* Double-relaxation analysis of a particular value */
/* for a particular variable */
possible_value2(O,CL,V) :- member(V,CL),
  bagof([T,A],constraint(T,A),ConL),
  not(constraint_violation2(V,O,ConL)).


constraint_violation2(V,O,ConL) :-
  twoconstraints(ConL,Type,Args,Type2,Args2),
  member(O,Args), argsoverlap(Args,Args2),
  substitute(O,V,Args,XArgs),
  substitute(O,V,Args2,XArgs2),
  not(double_satisfiable(Type,XArgs,Type2,XArgs2)).


double_satisfiable(Type,Args,Type2,Args2) :-
  constraint_preference(Type2,Type),
  !, double_satisfiable(Type2,Args2,Type,Args).
double_satisfiable(Type,Args,Type2,Args2) :-
  satisfiable(Type,Args,Newargs),
  bind(Args,Newargs,Args2,Result),
  satisfiable(Type2,Result,Newargs2).


twoconstraints([[T,A]|ConL],T,A,T2,A2) :- member([T2,A2],ConL).
twoconstraints([[T,A]|ConL],T1,A1,T2,A2) :-
  twoconstraints(ConL,T1,A1,T2,A2).


argsoverlap(L1,L2) :- member(X,L1), member(X,L2),
  choice_list(X,CL), !.


bind([],[],A2,A2).
bind([V|A],[V|NA],A2,R) :- bind(A,NA,A2,R), !.
bind([O|A],[V|NA],A2,R) :- substitute(O,V,A2,NA2),
  bind(A,NA,NA2,R), !.
```

```
/* Utility functions */
repeatactive.
repeatactive :- active(X), repeatactive.


member(X,[X|L]).
member(X,[Y|L]) :- member(X,L).


substitute(X,Y,[],[]).
substitute(X,Y,[X|L],[Y|L2]) :- substitute(X,Y,L,L2), !.
substitute(X,Y,[Z|L],[Z|L2]) :- substitute(X,Y,L,L2).


some_bindings([],[]).
some_bindings([Arg|Args],[Substarg|Substargs]) :-
  choice_list(Arg,CL), !, member(Substarg,CL),
  substitute(Arg,Substarg,Args,Args2),
  some_bindings(Args2,Substargs).
some_bindings([Arg|Args],[Arg|Substargs]) :-
  some_bindings(Args,Substargs).
```

And here's the script of a run on the "send more money" problem, using the problem-dependent file of the last section, without E=5, and plus one additional fact which says sum constraints should be used before unique constraints (since sum requires less computation to check):

```
constraint_preference(sum,unique).
```

Extra blank lines were inserted in the following printout to make it easier to read.

```
?- relax.
Single predicate relaxation begun.
Studying variable e
Studying variable n
Studying variable d
Studying variable o
Possibilities for o reduced to [0,1]
Studying variable r
Studying variable y
Studying variable s
Possibilities for s reduced to [8,9]
Studying variable m
Possibilities for m reduced to [1]
Studying variable c1
Studying variable c10
Studying variable c100

Studying variable s
Studying variable o
Possibilities for o reduced to [0]
Studying variable e
Possibilities for e reduced to [2,3,4,5,6,7,8,9]
Studying variable n
Possibilities for n reduced to [2,3,4,5,6,7,8,9]
Studying variable d
Possibilities for d reduced to [2,3,4,5,6,7,8,9]
Studying variable r
```

```
Possibilities for r reduced to [2,3,4,5,6,7,8,9]
Studying variable y
Possibilities for y reduced to [2,3,4,5,6,7,8,9]
Studying variable c1
Studying variable c10
Studying variable c100
Possibilities for c100 reduced to [0]

Studying variable r
Studying variable d
Studying variable n
Studying variable e
Studying variable s
Possibilities for s reduced to [9]
Studying variable c10
Studying variable y
Possibilities for y reduced to [2,3,4,5,6,7,8]
Studying variable r
Possibilities for r reduced to [2,3,4,5,6,7,8]
Studying variable d
Possibilities for d reduced to [2,3,4,5,6,7,8]
Studying variable n
Possibilities for n reduced to [2,3,4,5,6,7,8]
Studying variable e
Possibilities for e reduced to [2,3,4,5,6,7,8]
Studying variable c1
Studying variable c10

Studying variable n
Studying variable d
Studying variable r
Studying variable y
Double predicate relaxation begun.
Studying variable e
Possibilities for e reduced to [2,3,4,5,6,7]
Studying variable n
Possibilities for n reduced to [3,4,5,6,7,8]
Studying variable d
Studying variable r
Studying variable y
Studying variable c1
Studying variable c10
Possibilities for c10 reduced to [1]

Single predicate relaxation begun.
Studying variable n
Studying variable e
Studying variable d
Studying variable r
Possibilities for r reduced to [3,4,5,6,7,8]
Studying variable y
Studying variable c1
Studying variable n
Studying variable e
Studying variable d

Double predicate relaxation begun.
```

```
Studying variable r
Possibilities for r reduced to [8]
Studying variable n
Possibilities for n reduced to [4,5,6,7]
Studying variable e
Possibilities for e reduced to [3,4,5,6]
Studying variable d
Possibilities for d reduced to [2,3,4,7]
Studying variable y
Possibilities for y reduced to [2,3,5,6,7]
Studying variable c1
Possibilities for c1 reduced to [1]

Single predicate relaxation begun.
Studying variable y
Possibilities for y reduced to [2,3]
Studying variable d
Possibilities for d reduced to [7]
Studying variable e
Possibilities for e reduced to [5,6]
Studying variable n
Possibilities for n reduced to [6]
Studying variable e
Possibilities for e reduced to [5]
Studying variable y
Possibilities for y reduced to [2]

choice_list(y,[2]).
choice_list(e,[5]).
choice_list(n,[6]).
choice_list(d,[7]).
choice_list(c1,[1]).
choice_list(r,[8]).
choice_list(c10,[1]).
choice_list(s,[9]).
choice_list(c100,[0]).
choice_list(o,[0]).
choice_list(m,[1]).

yes
```

.SH Keywords:

constraint
label
dependency
dependency-based backtracking
possibility list
relaxation
active variable

Exercises

13-1. (A) Suppose in an expert system written in Prolog we have the following rules for proving that some person X is an **a**:

```
a(X) :- b(X), c(X).
a(X) :- d(X).
a(X) :- e(X).
```

Suppose:

b is true with probability 0.8;
c is true with probability 0.7;
d is true with probability 0.1;
e is true with probability 0.6;
b, c, d, and e are probabilistically independent of one another.

Rearrange the three rules, and perhaps the expressions in the first rule, to give the fastest (most efficient) execution of the query

```
?- a(george).
```

13-2. Rearrange the query given in Figure 13-6 for most efficient execution using the suggested rearrangement guidelines. How does regular Prolog interpretation of the rearranged query compare with dependency-based backtracking on the original query?

13-3. (E) For some queries it is better to rearrange the query than to do dependency-based backtracking, but for other queries dependency-based backtracking will always result in fewer backtracks than any rearrangement of the query. Explain why the following exemplifies the latter situation:

```
?- a(X), b(Y), c(Z), d(X,Y), e(X,Z), f(Y,Z).
```

13-4. (A) Suppose we are doing relaxation for picture interpretation. It would be nice if we can determine unique variable assignments when done; that would be strong evidence that we found the correct interpretation of the picture. But what can we conclude if we get instead:

(a) No possible final interpretations?

(b) Two possible final interpretations?

(c) One million possible final interpretations?

13-5. (E) Discuss why relaxation might have more application to artistic creation than any of the other control structures we have discussed in this book.

13-6. (R,A) Budget General Hospital must schedule meals for its patients. Budget tries to save its patients money, so its food is cheap and tastes terrible, and there are only three meals offered:

Meal m1: gruel
Meal m2: dehydrated eggs
Meal m3: leftovers from nearby restaurants

Assist the hospital by scheduling six meals for a patient. The six meals are breakfast, lunch, and dinner on each of the two days. The following constraints apply:

C1. Breakfast on the first day must be Meal m1, since the patient will still be queasy from their operation.

C2. No two meals in succession may be the same--you don't want the patients to realize how cheap you are.

C3. Meal m3 is the most expensive, so it must only occur once. (It requires a trip to a restaurant.)

C4. Every meal on the second day must have a higher number than the meal at the corresponding time on the first day. For instance, lunch on the first day could be Meal 1 and lunch on the second day Meal m2. (This is because patients are harder to fool as they recover.)

Find a meal plan for the six meals using pure relaxation. After obtaining initial possibility lists, check the possibilities for breakfast on the second day, and proceed from there. Write out each step of your solution, giving constraints used and how they were used.

13-7. (E) Develop a relaxation approach to analyzing pictures like Figure 13-9. This picture was produced by a camera mounted on a robot moving around outdoors. The picture only indicates line boundaries between visual regions of abruptly different brightness. Assume all the lines represent sharp boundaries, and the camera was focussed far away so that everything seen was more than 30 feet away. Assume the camera was oriented so that horizontal is one third of the way down from the top of the picture.

(a) Define ten permissible labels for regions based on their shapes, the ten most reasonable ones you can think of.

(b) List impossible relationships between regions based on these labels (what regions cannot be next to a particular region, etc.) Use **above**, **left-of**, and **inside** for relationships, and any others you think useful. Treat combinations that are possible only rarely as impossible; don't worry about minor exceptions.

(c) Use your constraints to assign as narrow an interpretation as possible to the picture.

13-8. Relaxation can be used to solve an immensely important problem of large organizations: interpretation of acronyms. The full names of many organizations are so long that it's impossible to avoid abbreviating them, yet no one can keep track of all these acronyms. Sophisticated computer programs are maybe necessary.

Suppose for each letter of the alphabet we have a list of words that the letter could represent, in the context of names of government organizations. Suppose:

N can be national, naval, new, name
S can be school, staff, science
C can be center, committee, computer, crazy

Here are the constraints:

C1: a word at the right end of an acronym must be a noun. The nouns are "name", "science", "school", "staff", "center", "committee", and "computer".

C2: a word anywhere else must be an adjective. The adjectives are "national", "naval", "new",

"science", "computer", and "crazy".

C3: the same word cannot occur twice in an acronym.

C4: "national" and "naval" cannot be in the same acronym with "crazy" (no one ever admits something national or naval is crazy, even when it is).

C5: "staff" cannot be in the same acronym with "naval", "national", or "name".

C6: "new" cannot be in the same acronym with "science".

Interpret the acronym "NCSS" using these constraints.

(a) Use relaxation to reduce the possibilities as much as you can. Write out each step. Then write out the possible interpretations remaining.

(b) Suppose we assign probabilities to every word possibility, representing how likely they are to occur in an acronym. So "national" might be 0.5 and "new" might be 0.02 (rarely is anything new in government organizations). Describe a good way to combine these probabilities to choose the "best" one of several interpretations of some acronym, "best" according to a single derived number (give or refer to a formula).

13-9. (H) Try to interpret Figure 13-10 (label the regions) using constraint propagation. You won't be able to find a unique interpretation, but do as much as you can. Initial region labels are as follows:

The five small regions are either airplanes or ships; anything else is either water, marsh, land, or a cloud.

You have the following constraints:

C1. Airplane regions are the only ones that can be within cloud regions, but airplane regions can be within any region.
C2. Ship regions can only be within water regions.
C3. Two small regions touching one another must be ships.
C4. A "T" vertex (where three line segments meet, and two of them form an angle of approximately 180 degrees) not involving a ship or airplane means either

(a) that the region with the 180 degree angle is a cloud, or
(b) the region with the 180 degree angle is land and the other two regions are water and marsh.

C5. Every marsh region must have such a T vertex somewhere on its border.
C6. Water regions cannot border water regions.
C7. Marsh regions cannot border marsh regions.
C8. Land regions cannot border land regions.
C9. The outer (border) region is not a cloud.

Give every step in your relaxation.

13-10.(a) Suppose in relaxation there are ten variables with initially eight possible labels each. Suppose there is a unique solution. What is the size of the search space?

(b) Suppose each step that chooses a variable eliminates half the possibilities for that variable, on the average. What is approximately the average number of steps to reach the unique solution?

13-11. Consider a picture as a graph. Regions border edges, and there are vertices where edges meet. Suppose you do relaxation to interpret the picture, and constraints specify what labels adjacent regions can have. Suppose you draw another graph for which the vertices represent variables in the relaxation, with connections between vertices representing that the two variables both occur together in at least one constraint. What is the relationship of this new graph to the original picture graph?

13-12. (E) Suppose we are certain that there is one and only one solution to a relaxation problem. Suppose when we run this relaxation problem with all the constraints we can think of, we get several hundred possible interpretations. Suppose we can also formulate some "reasonable restrictions", restrictions usually but not always true (their probability being close to 1). Then we can try to get a solution by taking (assuming) some of the "reasonable restrictions" as constraints, and seeing if we get a unique solution to the relaxation. But we want to assume as few restrictions as possible, and we prefer to assume those that are most certain. Explain how finding a unique solution to a relaxation problem by assuming "reasonable restrictions" can be considered a search problem. Describe its characteristics, and recommend an appropriate search strategy.

13-13. (R,A,P) Consider the query:

```
?- n(X), n(Y), n(Z), X>Y, not(f(X,X)), g(X,Y,Z).
```

Suppose the database is:

```
n(1).
n(2).
n(3).
n(4).
n(5).


f(2,2).
f(2,4).
f(3,1).


g(1,3,2).
g(2,3,3).
g(3,3,4).
g(4,3,5).
```

(a) How many times does this query backtrack to get its first answer? Count each right-to-left movement from predicate expression to predicate expression as one backtrack.

(b) Rearrange the query in a form that leads to the fewest backtracks. How many times does it backtrack now? Show the query you used.

(c) Suggest a way to do dependency-based backtracking to answer the original query. Simulate it without the computer. How many times do you backtrack now?

(d) Now answer the original query by doing relaxation without the computer. Assume the standard form of relaxation that tries to satisfy each multivariable constraint separately, throwing away previously found bindings. Write out your steps.

13-14. (A) The demonstration of the pure relaxation program on the cryptarithmetic problem takes a considerable amount of time on the first few variables, then speeds up considerably. If instead of specifying E=5 we say that E is either 0 or 5, the program runs faster and reaches a solution faster.

(a) What accounts for this paradox?

(b) Suggest code that can be eliminated from the definition of this cryptarithmetic problem to significantly lessen or eliminate the paradoxical behavior.

(c) The code that you eliminated in part (b) must have been there for a reason. What was it?

13-15. (P) Suppose you have an eight-digit number whose digits are all different. Suppose that when you multiply this number by its rightmost (least) digit, you get a nine-digit number whose digits are all the leftmost (highest) digit of the original number. Suppose further that the second-highest digit of the original number is 2. Set this problem up in a problem-dependent file to run with the pure relaxation program. Run it and see what you get.

13-16. (H,P) Generalize the "double relaxation" program to "N-relaxation". In other words, take three constraints together if you don't get anywhere with two constraints together, and four constraints together if you don't get anywhere with three, and so on up to the total number of constraints. This will require recursion. The program should degrade to standard Prolog querying in the worst case. Try your program on the cryptarithmetic problem CROSS + ROADS = DANGER.

13-17. (H,P) Besides double relaxation, there is an alternative improvement on pure relaxation: keep possibility lists for pairs, triples, etc. of variables instead of single variables. So you can solve harder problems this way than with pure relaxation, but you need more possibility lists. Write a "pair-relaxation program" that reasons about pairs of values this way.

13-18. (A) We can see relaxation as a search problem with a single operator: picking a variable and a value, and trying to verify that value is still possible.

(a) What is the search space for this problem? Remember, a state for a search problem must include everything about the problem in one bundle.

(b) Which search strategy, of those mentioned in Chapter 9, is most like that used by the algorithm?

(c) Is this search decomposable about intermediate states into simpler subproblems?

(d) How does the branching factor change as search proceeds?

(e) Give an upper bound on the size of the search space.

Go to book index