CHAPTER 5
Resolution Strategies

ONE OF THE DISADVANTAGES of using the resolution rule in an unconstrained manner is that it leads to many useless inferences. Some inferences are redundant in that their conclusions can be derived in other ways. Some inferences are irrelevant in that they do not lead to derivations of the desired result.

As an example, consider the resolution trace in Figure 5.1. Clauses 9, 11, 14, and 16 are redundant; clauses 10 and 13 are redundant; clauses 12 and 15 are redundant; all these redundancies lead to subsequent redundancies at the next level of deduction. We can remove duplicate clauses and thereby prevent the propagation of redundant conclusions. However, their initial generation is an indication of inefficiency in the unconstrained use of the resolution principle.

This chapter presents a number of strategies for eliminating useless work. In reading the chapter, it is important to bear in mind that we are concerned here not with the order in which inferences are done, but only with the size of a resolution graph and with ways of decreasing that size by eliminating useless deductions.

5.1 Deletion Strategies

A *deletion strategy* is a restriction technique in which clauses with specified properties are eliminated before they are ever used. Since those clauses

1.	{P,Q}	Δ
2.	{¬P,R}	Δ
3.	{¬Q,R}	Δ
4.	{¬R}	Γ
5.	{Q,R}	1, 2
6.	{P,R}	1, 3
7.	{¬P}	2, 4
8.	{¬Q}	3, 4
9.	{R}	3, 5
10.	{Q}	4, 5
11.	{R}	3, 6
12.	{P}	4, 6
13.	{Q}	1, 7
14.	{R}	6, 7
15.	{P}	1, 8
16.	{R}	5, 8
17.	{}	4, 9
18.	{R}	3, 10
19.	{}	8, 10
20.	{}	4, 11
21.	{R}	2, 12
22.	{}	7, 12
23.	{R}	3, 13
24.	{}	8, 13
25.	{}	4, 14
26.	{R}	2, 15
27.	{}	7, 15
28.	{}	4, 16
29.	{}	4, 18
30.	{}	4, 21
31.	{}	4, 23
32.	{}	4, 26

Figure 5.1 Example of unconstrained resolution.

are unavailable for subsequent deduction, this can lead to computational savings.

A literal occurring in a database is *pure* if and only if it has no instance that is complementary to an instance of another literal in the database.

A clause that contains a pure literal is useless for the purposes of refutation, since the literal can never be resolved away. Consequently, we can safely remove such a clause. Removing clauses with pure literals defines a deletion strategy known as *pure-literal elimination.*

The database that follows is unsatisfiable. However, in proving this we can ignore the second and third clauses, since they both contain the pure literal S.

$$\{\neg P, \neg Q, R\}$$
$$\{\neg P, S\}$$
$$\{\neg Q, S\}$$
$$\{P\}$$
$$\{Q\}$$
$$\{\neg R\}$$

Note that, if a database contains no pure literals, there is no way we can derive any clauses with pure literals using resolution. The upshot is that we do not need to apply the strategy to a database more than once, and in particular we do not have to check each clause as it is generated.

A *tautology* is a clause containing a pair of complementary literals. For example, the clause $\{P(F(A)), \neg P(F(A))\}$ is a tautology. The clause $\{P(x), Q(y), \neg Q(y), R(z)\}$ also is a tautology, even though it contains additional literals.

As it turns out, the presence or absence of tautologies in a set of clauses has no effect on that set's satisfiability. A satisfiable set of clauses remains satisfiable, no matter what tautologies we add. An unsatisfiable set of clauses remains unsatisfiable, even if we remove all tautologies. Therefore, we can remove tautologies from a database, because we need never use them in subsequent inferences. The corresponding deletion strategy is called *tautology elimination.*

Note that the literals in a clause must be exact complements for tautology elimination to apply. We cannot remove nonidentical literals, just because they are complements under unification. For example, the clauses $\{\neg P(A), P(x)\}$, $\{P(A)\}$, and $\{\neg P(B)\}$ are unsatisfiable. However, if we were to remove the first clause, the remaining set would be satisfiable.

In *subsumption elimination*, the deletion criterion depends on a relationship between two clauses in a database. A clause Φ *subsumes* a clause Ψ if and only if there exists a substitution σ such that $\Phi\sigma \subseteq \Psi$. For example, $\{P(x), Q(y)\}$ subsumes $\{P(A), Q(v), R(w)\}$, since there is a substitution $\{x/A, y/v\}$ that makes the former clause a subset of the latter.

If one member in a set of clauses is subsumed by another member, then the set remaining after eliminating the subsumed clause is satisfiable if and

only if the original set is satisfiable. Therefore, subsumed clauses can be eliminated. Since the resolution process itself can produce tautologies and subsuming clauses, we need to check for tautologies and subsumptions as we perform resolutions.

5.2 Unit Resolution

A *unit resolvent* is one in which at least one of the parent clauses is a *unit clause*; i.e., one containing a single literal. A *unit deduction* is one in which all derived clauses are unit resolvents. A *unit refutation* is a unit deduction of the empty clause {}.

As an example of a unit refutation, consider the following proof. In the first two inferences, unit clauses from the initial set are resolved with binary clauses to produce two new unit clauses. These are resolved with the first clause to produce two additional unit clauses. The elements in these two sets of results are then resolved with each other to produce the contradiction.

1.	{P,Q}	Δ
2.	{¬P,R}	Δ
3.	{¬Q,R}	Δ
4.	{¬R}	Γ
5.	{¬P}	2, 4
6.	{¬Q}	3, 4
7.	{Q}	1, 5
8.	{P}	1, 6
9.	{R}	3, 7
10.	{}	6, 7
11.	{R}	2, 8
12.	{}	5, 8

Note that the proof contains only a subset of the possible uses of the resolution rule. For example, clauses 1 and 2 can be resolved to derive the conclusion {Q,R}. However, this conclusion and its descendants are never generated, since neither of its parents is a unit clause.

Inference procedures based on unit resolution are easy to implement and are usually quite efficient. It is worth noting that, whenever a clause is resolved with a unit clause, the conclusion has fewer literals than the parent does. This helps to focus the search toward producing the empty clause and thereby improves efficiency.

Unfortunately, inference procedures based on unit resolution generally are not complete. For example, the clauses {P,Q}, {¬P,Q}, {P,¬Q}, and {¬P,¬Q} are inconsistent. Using general resolution, it is easy to derive the

empty clause. However, unit resolution fails in this case, since none of the initial propositions is a single literal.

On the other hand, if we restrict our attention to Horn clauses (i.e., clauses with at most one positive literal), the situation is much better. In fact, it can be shown that there is a unit refutation of a set of Horn clauses if and only if it is unsatisfiable.

5.3 Input Resolution

An *input resolvent* is one in which at least one of the two parent clauses is a member of the initial (i.e., input) database. An *input deduction* is one in which all derived clauses are input resolvents. An *input refutation* is an input deduction of the empty clause {}.

As an example, consider clauses 6 and 7 in Figure 5.1. Using unconstrained resolution, these clauses can be resolved to produce clause 14. However, this is not an input resolution, since neither parent is a member of the initial database.

Note that the resolution of clauses 1 and 2 is an input resolution but not a unit resolution. On the other hand, the resolution of clauses 6 and 7 is a unit resolution but not an input resolution. Despite differences such as this one, it can be shown that unit resolution and input resolution are equivalent in inferential power in that there is a unit refutation from a set of sentences whenever there is an input refutation and vice versa.

One consequence of this fact is that input resolution is complete for Horn clauses but incomplete in general. Again, the unsatisfiable set of propositions {P,Q}, {¬P,Q}, {P,¬Q}, and {¬P,¬Q} provides an example of a deduction on which input resolution fails. An input refutation must (in particular) have one of the parents of {} be a member of the initial database. However, to produce the empty clause in this case, we must resolve either two single literal clauses or two clauses having single-literal factors. None of the members of the base set meet either of these criteria, so there cannot be an input refutation for this set.

5.4 Linear Resolution

Linear resolution (also called *ancestry-filtered resolution*) is a slight generalization of input resolution. A *linear resolvent* is one in which at least one of the parents is either in the initial database or is an ancestor of the other parent. A *linear deduction* is one in which each derived clause is a linear resolvent. A *linear refutation* is a linear deduction of the empty clause {}.

Linear resolution takes its name from the linear shape of the proofs it generates. A linear deduction starts with a clause in the initial database (called the *top clause*) and produces a linear chain of resolutions such as that shown in Figure 5.2. Each resolvent after the first one is obtained from

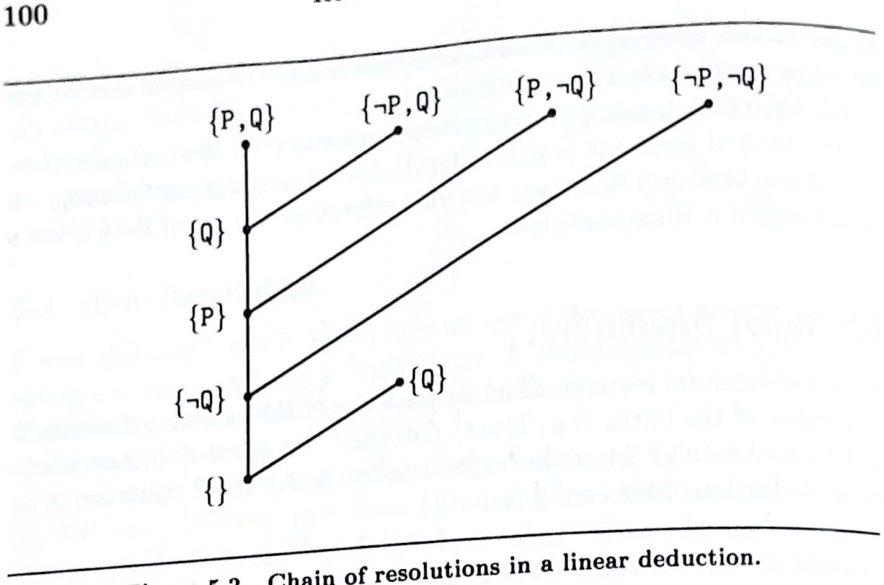


Figure 5.2 Chain of resolutions in a linear deduction.

the last resolvent (called the *near parent*) and some other clause (called the *far parent*). In linear resolution, the far parent must either be in the initial database or be an ancestor of the near parent.

Much of the redundancy in unconstrained resolution derives from the resolution of intermediate conclusions with other intermediate conclusions. The advantage of linear resolution is that it avoids many useless inferences by focusing deduction at each point on the ancestors of each clause and on the elements of the initial database.

Linear resolution is known to be refutation complete. Furthermore, it is not necessary to try every clause in the initial database as top clause. It can be shown that, if a set of clauses Γ is satisfiable and $\Gamma \cup \{\phi\}$ is unsatisfiable, then there is a linear refutation with ϕ as top clause. So, if we knows that a particular set of clauses is consistent, one need not attempt refutations with the elements of that set as top clauses.

A *merge* is a resolvent that inherits a literal from each parent such that this literal is collapsed to a singleton by the most general unifier. The completeness of linear resolution is preserved even if the ancestors that are used are limited to merges. Note that, in this example, the first resolvent (i.e., clause $\{Q\}$) is a merge.

5.5 Set of Support Resolution

If we examine resolution traces such as the one shown in Figure 5.1, we notice that many conclusions come from resolutions between clauses contained in a portion of the database that we know to be satisfiable. For

example, in Figure 5.1, the set Δ is satisfiable, yet many of the conclusions in the trace are obtained by resolving elements of Δ with other elements of Δ. As it turns out, we can eliminate these resolutions without affecting the refutation completeness of resolution.

A subset Γ of a set Δ is called a *set of support* for Δ if and only if $\Delta - \Gamma$ is satisfiable. Given a set of clauses Δ with set of support Γ, a *set of support resolvent* is one in which at least one parent is selected from Γ or is a descendant of Γ. A *set of support deduction* is one in which each derived clause is a set of support resolvent. A *set of support refutation* is a set of support deduction of the empty clause {}.

The following trace is a set of support refutation for the example in Figure 5.1, with the singleton set {¬R} as the set of support. The clause {¬R} resolves with {¬P,R} and {¬Q,R} to produce {¬P} and {¬Q}. These then resolve with clause 1 to produce {Q} and {P}, which resolve to produce the empty clause.

1.	{P,Q}	Δ
2.	{¬P,R}	Δ
3.	{¬Q,R}	Δ
4.	{¬R}	Γ
5.	{¬P}	2, 4
6.	{¬Q}	3, 4
7.	{Q}	1, 5
8.	{P}	1, 6
9.	{R}	3, 7
10.	{}	6, 7
11.	{R}	2, 8
12.	{}	5, 8

Obviously, this strategy would be of little use if there were no easy way of selecting the set of support. Fortunately, there are several ways this can be done at negligible expense. For example, in situations where we are trying to prove conclusions from a consistent database, the natural choice is to use the clauses derived from the negated goal as the set of support. This set satisfies the definition as long as the database itself is truly satisfiable. With this choice of set of support, each resolution must have a connection to the overall goal, so the procedure can be viewed as working "backward" from the goal. This is especially useful for databases in which the number of conclusions possible by working "forward" is larger. Furthermore, the goal-oriented character of such refutations often makes them more understandable than refutations using other strategies.

5.6 Ordered Resolution

Ordered resolution is a very restrictive resolution strategy in which each clause is treated as a linearly ordered set. Resolution is permitted only on the first literal of each clause; i.e., the literal that is least in the ordering. The literals in the conclusion preserve the order from their parent clauses with the literals from the positive parent followed by the literals from the negative parent (i.e., the one with the negated atom).

The following trace is an example of an ordered refutation. Clause 5 is the only ordered resolvent of clauses 1 through 4. Clauses 1 and 3 do not resolve, since the complementary literals are not first in each clause. Clauses 2 and 4 do not resolve for the same reason, nor do clauses 3 and 4. Once clause 5 is generated, it resolves with clause 3 to produce clause 6, which resolves with clause 4 to produce the empty clause.

1.	{P,Q}	Δ
2.	{¬P,R}	Δ
3.	{¬Q,R}	Δ
4.	{¬R}	Γ
5.	{Q,R}	1, 2
6.	{R}	3, 5
7.	{}	4, 6

Ordered resolution is extremely efficient. In this case, the empty clause is produced at the third level of deduction, and the inference space through that level of deduction includes only three resolvents. By comparison, general resolution through that level results in 24 resolvents.

Unfortunately, ordered resolution is not refutation complete. However, if we restrict our attention to Horn clauses, refutation completeness is guaranteed. Furthermore, we can get refutation completeness in the general case by considering resolvents in which the remaining literals from the positive parent follow the remaining literals from the negative parent, as well as the other way around.

5.7 Directed Resolution

Directed resolution is the use of ordered resolution in an important but restricted set of deductions. In directed resolution, the query takes the form of a conjunction of positive literals, and the database consists entirely of *directed clauses*. A directed clause is a Horn clause in which the positive literal occurs either at the beginning or the end of the clause. The goal is to find bindings for the variables so that the conjunction resulting from the substitution of these bindings is provable from the database.

In looking at directed resolution, we can use a bit of syntactic sugar. Since all the clauses are directional, we can write them in *infix form*. We write clauses with the positive literal at the end using the $\Rightarrow$ operator. We write clauses in which the positive literal is at the beginning using the reverse implication operator $\Leftarrow$. We let the literal in a positive unit clause represent the clause as a whole. We write the negative literals in clauses without positive literals as the antecedents of either implication operator.

$$\{\neg\phi_1, \ldots, \neg\phi_n, \psi\} \leftrightarrow \phi_1, \ldots, \phi_n \Rightarrow \psi$$
$$\{\psi, \neg\phi_1, \ldots, \neg\phi_n\} \leftrightarrow \psi \Leftarrow \phi_1, \ldots, \phi_n$$
$$\{\neg\phi_1, \ldots, \neg\phi_n\} \leftrightarrow \phi_1, \ldots, \phi_n \Rightarrow$$
$$\{\neg\phi_1, \ldots, \neg\phi_n\} \leftrightarrow \Leftarrow \phi_1, \ldots, \phi_n$$

The distinguishing feature of directed resolution is the directionality of the clauses in the database. Some clauses give rise to forward resolution, in which positive conclusions are derived from positive data. Other clauses give rise to backward resolution, in which negative clauses are derived from other negative clauses. As suggested by the preceding equivalences, the directionality of a clause is determined by the position of the positive literal in the clause.

A *forward clause* is one in which the positive literal comes at the end. In directed resolution, forward clauses give rise to forward resolution. To see why this is so, consider the following proof. Using ordered resolution on the first two clauses leads to the conclusion `P(A)`, and then this conclusion is resolved with the negative unit to derive the empty clause. Putting the positive literal at the end makes it possible to work forward to the positive intermediate conclusion (clause 4), but makes it impossible to work backward from the negative clause (clause 3).

1. {¬M(x),P(x)}	M(x) ⇒ P(x)
2. {M(A)}	M(A)
3. {¬P(z)}	P(z) ⇒
4. {P(A)}	P(A)
5. {}	{}

Symmetrically, if the positive literal is put at the front of a clause, the clause is *backward*. If we rewrite the previous clauses in this way, we get the opposite behavior. In the following proof, the negative clause is resolved with the first clause to produce the intermediate negative conclusion {¬M(z)}, then this result is resolved with the second clause to derive the empty clause.

1. {P(x),¬M(x)}	P(x) ⇐ M(x)
2. {M(A)}	M(A)
3. {¬P(z)}	⇐ P(z)

4. {¬M(z)} ⇐ M(z)

5. {} ⇐

By making some clauses forward and others backward, we can get a mixture of forward and backward resolution. As an example, consider the following proof. The positive data first resolve with forward clause 2 to produce more positive results. These results then resolve with clause 1 to produce some intermediate results. These results resolve with backward clause 3 to produce two subgoals involving N. One of these succeeds, leading to the positive result {R(B)}. This then resolves with clause 7 to produce the empty clause.

1. {¬P(x),¬Q(x),R(x)}	P(x),Q(x) ⇒ R(x)
2. {¬M(x),P(x)}	M(x) ⇒ P(x)
3. {Q(x),¬N(x)}	Q(x) ⇐ N(x)
4. {M(A)}	M(A)
5. {M(B)}	M(B)
6. {N(B)}	N(B)
7. {¬R(z)}	R(z) ⇒
8. {P(A)}	P(A)
9. {P(B)}	P(B)
10. {¬Q(A),R(A)}	Q(A) ⇒ R(A)
11. {¬Q(B),R(B)}	Q(B) ⇒ R(B)
12. {¬N(A),R(A)}	N(A) ⇒ R(A)
13. {¬N(B),R(B)}	N(B) ⇒ R(B)
14. {R(B)}	R(B)
15. {}	⇒

The possibility of controlling the direction of resolution by positioning the positive literal at one or the other end of a clause raises the question of which direction is more efficient. For the purpose of comparison, consider the following set of sentences.

Insect(x) ⇒ Animal(x)

Mammal(x) ⇒ Animal(x)

Ant(x) ⇒ Insect(x)

Bee(x) ⇒ Insect(x)

Spider(x) ⇒ Insect(x)

Lion(x) ⇒ Mammal(x)

```
Tiger(x) ⇒ Mammal(x)
Zebra(x) ⇒ Mammal(x)
```

Assuming that Zeke is a zebra, is Zeke an animal? The following proof shows that the search space in this case is quite small.

1. {Zebra(Zeke)}
2. {¬Animal(Zeke)}

---

3. {Mammal(Zeke)}

---

4. {Animal(Zeke)}

---

5. {}

Unfortunately, things are not always so pleasant. As an example, consider the following database of information about zebras. Zebras are mammals, striped, and medium in size. Mammals are animals and warm-blooded. Striped things are nonsolid and nonspotted. Things of medium size are neither small nor large.

```
Zebra(x) ⇒ Mammal(x)
Zebra(x) ⇒ Striped(x)
Zebra(x) ⇒ Medium(x)
Mammal(x) ⇒ Animal(x)
Mammal(x) ⇒ Warm(x)
Striped(x) ⇒ Nonsolid(x)
Striped(x) ⇒ Nonspotted(x)
Medium(x) ⇒ Nonsmall(x)
Medium(x) ⇒ Nonlarge(x)
```

The following proof shows that the search space in this case is somewhat larger than in the previous example. The reason is that we can derive more than one conclusion from each clause than we manage to derive.

1. {Zebra(Zeke)}
2. {¬Nonlarge(Zeke)}

---

3. {Mammal(Zeke)}
4. {Striped(Zeke)}

5. {Medium(Zeke)}

---

6. {Animal(Zeke)}
7. {Warm(Zeke)}
8. {Nonsolid(Zeke)}
9. {Nonstriped(Zeke)}
10. {Nonsmall(Zeke)}
12. {Nonlarge(Zeke)}

---

13. {}

Now consider what would happen if we were to reverse the direction of the clauses, as follows.

```
Mammal(x) ⇐ Zebra(x)
Striped(x) ⇐ Zebra(x)
Medium(x) ⇐ Zebra(x)
Animal(x) ⇐ Mammal(x)
Warm(x) ⇐ Mammal(x)
Nonsolid(x) ⇐ Striped(x)
Nonspotted(x) ⇐ Striped(x)
Nonsmall(x) ⇐ Medium(x)
Nonlarge(x) ⇐ Medium(x)
```

The following proof shows that the search space of backward resolution in this case is much smaller than that for forward resolution.

1. {Zebra(Zeke)}
2. {¬Nonlarge(Zeke)}

---

3. {¬Medium(Zeke)}

---

4. {¬Zebra(Zeke)}

---

5. {}

Unfortunately, like forward resolution, backward resolution has its drawbacks. As an example, consider the backward version of the clauses in the animal problem.

```
Animal(x) ⇐ Insect(x)
Animal(x) ⇐ Mammal(x)
```

```
Insect(x) ⇐ Ant(x)
Insect(x) ⇐ Bee(x)
Insect(x) ⇐ Spider(x)
Mammal(x) ⇐ Lion(x)
Mammal(x) ⇐ Tiger(x)
Mammal(x) ⇐ Zebra(x)
```

The following proof shows that the search space for the backward direction is much larger than it is for the forward direction.

1. {Zebra(Zeke)}
2. {¬Animal(Zeke)}

---

3. {¬Insect(Zeke)}
4. {¬Mammal(Zeke)}

---

5. {¬Ant(Zeke)}
6. {¬Bee(Zeke)}
7. {¬Spider(Zeke)}
8. {¬Lion(Zeke)}
9. {¬Tiger(Zeke)}
10. {¬Zebra(Zeke)}

---

11. {}

The fact is that forward resolution is best for some clause sets, and backward resolution is best for others. To determine which is best for which, we need to look at the branching factor of the clauses. In the preceding examples, the search space branches backward in the animal problem and forward in the zebra problem. Consequently, we should use forward resolution in the animal problem and backward resolution in the zebra problem.

Of course, things are not always this simple. Sometimes, it is best to use some clauses in the forward direction and others in the backward direction; deciding which clauses to use in which direction to get optimal performance is a computationally difficult problem. The problem can be solved in polynomial time, if we restrict our attention to *coherent databases*; i.e., those in which all clauses that can be used to prove a literal in the antecedent of a forward clause are themselves forward clauses. In general, however, the problem is NP-complete.

5.8 Sequential Constraint Satisfaction

Sequential constraint satisfaction is the use of ordered resolution in the solution of another restricted but important class of fill-in-the-blank questions. Like directed resolution, the query is posed as a conjunction of positive literals, containing some number of variables. However, unlike directed resolution, the database consists entirely of positive ground literals. The task is to find bindings for the variables such that, after substitution into the query, each of the resulting conjuncts is identical to a literal in the database.

As an example, consider the following database. Art and Ann are the parents of Jon; Bob and Bea are the parents of Kim; and Cap and Coe are the parents of Lem. Ann and Cap are carpenters; Jon and Kim are U.S. senators.

```
P(Art,Jon)      Carpenter(Ann)      Senator(Jon)
P(Ann,Jon)      Carpenter(Cap)      Senator(Kim)
P(Bob,Kim)
P(Bea,Kim)
P(Cap,Lem)
P(Coe,Lem)
```

The following conjunction is a typical query for a database of this sort. We are looking for bindings for the variables `x` and `y` such that `x` is the parent of `y`, `x` is a carpenter, and `y` is a senator.

$$\texttt{P(x,y)} \wedge \texttt{Carpenter(x)} \wedge \texttt{Senator(y)}$$

To use resolution on this problem, we need to negate the query, to convert to clausal form, and to add an appropriate answer literal. This results in the following clause:

$$\{\neg\texttt{P(x,y)}, \neg\texttt{Carpenter(x)}, \neg\texttt{Senator(y)}, \texttt{Ans(x,y)}\}$$

We then use ordered resolution to derive an answer. The following sequence of deductions shows a trace of this strategy in solving this query using the preceding data.

1. $\{\neg\texttt{P(x,y)}, \neg\texttt{Carpenter(x)}, \neg\texttt{Senator(y)}, \texttt{Ans(x,y)}\}$

---

2. $\{\neg\texttt{Carpenter(Art)}, \neg\texttt{Senator(Jon)}, \texttt{Ans(Art,Jon)}\}$
3. $\{\neg\texttt{Carpenter(Ann)}, \neg\texttt{Senator(Jon)}, \texttt{Ans(Ann,Jon)}\}$
4. $\{\neg\texttt{Carpenter(Bob)}, \neg\texttt{Senator(Kim)}, \texttt{Ans(Bob,Kim)}\}$

5. {¬Carpenter(Bea),¬Senator(Kim),Ans(Bea,Kim)}
6. {¬Carpenter(Cap),¬Senator(Lem),Ans(Cap,Lem)}
7. {¬Carpenter(Coe),¬Senator(Lem),Ans(Coe,Lem)}

---

8. {¬Senator(Jon),Ans(Ann,Jon)
9. {¬Senator(Lem),Ans(Cap,Lem)

---

10. {Ans(Ann,Jon)}

From the standpoint of efficiency, one of the key questions in sequential constraint satisfaction is the order of the literals in the query. Although there is some search involved in the preceding example, it is not great. By comparison, it is interesting to consider what happens with a somewhat larger database and a slightly different ordering of the literals in the query.

To be specific, consider a census database with the following properties. There are 100 U.S. senators; so, if the database is complete and nonredundant, there are 100 solutions to queries of the form `Senator(`ν`)`, where ν is a variable. Similarly, there are several hundred thousand carpenters and, therefore, several hundred thousand solutions to queries of the form `Carpenter(`ν`)`. There are several hundred million parent–child pairs and, therefore, several hundred million solutions to queries of the form `P(`μ,ν`)` involving two variables. However, there are only two solutions to queries of the form `P(`ν,γ`)`, where ν is a variable and γ is a constant, since each person has only two parents. Similarly, there are only a few solutions to queries of the form `P(`γ,ν`)`, since each person has at most a few children. We indicate the sizes of these solution sets as follows, where the notation `||Q(x)||` is used to denote the number of instances of `Q(x)` in the database.

$$||\texttt{Senator}(\nu)|| = 100$$
$$||\texttt{Carpenter}(\nu)|| \approx 10^5$$
$$||\texttt{P}(\mu,\nu)|| \approx 10^8$$
$$||\texttt{P}(\nu,\gamma)|| = 2$$
$$||\texttt{P}(\gamma,\nu)|| \approx 3$$

Consider the difficulty of answering the preceding query with this expanded database. As before, working on the literals in the order given results in an enumeration of all parent–child pairs, except in this case the search space includes several hundred million possibilities.

A much better way to answer the query is to reorder the literals as shown below. Since there are only 100 senators and only two parents for each senator, this ordering limits the search space to at most 200 possibilities.

```
Senator(y) ∧ P(x,y) ∧ Carpenter(x)
```

This example suggests a useful heuristic for sequential constraint satisfaction, known as the *cheapest first rule*, which states that one should process the literals in a query in order of increasing solution-set size. Unfortunately, the rule does not always produce the optimal ordering. As an example, consider the following problem.

$$\texttt{P(x)} \wedge \texttt{Q(y)} \wedge \texttt{R(x,y)}$$

Assume the database has the characteristics shown below. The symbols μ and ν here refer to arbitrary variables, and γ is a constant.

$$||\texttt{P}(\nu)|| = 1000$$
$$||\texttt{Q}(\nu)|| = 2000$$
$$||\texttt{R}(\mu, \nu)|| = 100{,}000$$
$$||\texttt{R}(\gamma, \nu)|| = 100$$
$$||\texttt{R}(\mu, \gamma)|| = 10$$

In this case, `P(x)` is the literal with the smallest solution set; therefore, using the cheapest first rule, we enumerate its solutions first, a total of 1000 possibilities. Next we compare the set sizes of the remaining two literals for the case where `x` is known. There are 2000 solutions to the `Q` literals but only 100 solutions to the `R` literal, if `x` is known. So the `R` literal is processed next, leading to a total search space of 100,000.

The problem is that there is a better ordering. Working first on `Q(y)` produces an initial search space of 2000 possibilities. However, given a value for `y`, there are only 10 solutions for the `R` literal, leading to a search space of only 20,000, a factor of 5 smaller than the ordering suggested by the cheapest first rule.

One way of guaranteeing the optimal ordering for a set of literals is to search through all possible orderings. For each ordering, we can compute the expected cost. Then, we can compare orderings and select the one that is cheapest.

The following equations show the cost estimates for the six orderings of the literals in the preceding problem. From these estimates, it is easy to see that it is best to process the `Q` literal first, followed by `R`, and then `P`.

$$||\texttt{P(x),Q(y),R(x,y)}|| = 2{,}000{,}000$$
$$||\texttt{P(x),R(x,y),Q(y)}|| = 100{,}000$$
$$||\texttt{Q(y),P(x),R(x,y)}|| = 2{,}000{,}000$$
$$||\texttt{Q(y),R(x,y),P(x)}|| = 20{,}000$$

$$||\texttt{R(x,y)},\texttt{P(x)},\texttt{Q(y)}|| = 100{,}000$$
$$||\texttt{R(x,y)},\texttt{Q(y)},\texttt{P(x)}|| = 100{,}000$$

The problem with enumerating and comparing all possible orderings is inefficiency. For a set of n literals, there are $n!$ possible orderings. Although there are only six possible orderings for three literals, the number jumps to over 40,000 for eight literals.

Fortunately, there are some results that help in cutting down the search necessary to find the optimal ordering. The adjacency theorem (Theorem 5.1) is an example.

Given a set of literals $l_1, \ldots, l_n$, we define the situated literal l_i^j to be the literal obtained by substituting into l_i ground terms for the variables in $l_1, \ldots, l_j$. For example, given the query $\texttt{P(x)} \wedge \texttt{Q(x,y)} \wedge \texttt{R(x,y)}$, the situated literal $\texttt{P(x)}^0$ is just $\texttt{P(x)}$. The situated literal $\texttt{Q(x,y)}^0$ is $\texttt{Q(x,y)}$, but the situated literal $\texttt{Q(x,y)}^1$ is $\texttt{Q}(\gamma,\texttt{y})$, where γ is a ground term. The situated literal $\texttt{R(x,y)}^0$ is $\texttt{R(x,y)}$; $\texttt{R(x,y)}^1$ is $\texttt{R}(\gamma,\texttt{y})$; and $\texttt{R(x,y)}^2$ is $\texttt{R}(\gamma_1,\gamma_2)$.

THEOREM 5.1 (Adjacency Theorem) *If $l_1, \ldots, l_n$ is an optimal literal ordering, then $||l_i^{i-1}|| \leq ||l_{i+1}^{i-1}||$ for all i between 1 and $n-1$.*

This theorem supports our intuitions about literal ordering in the simple cases covered by the following corollaries.

COROLLARY 5.1 *The most expensive conjunct should never be done first.*

COROLLARY 5.2 *Given a conjunct sequence of length two, the less expensive conjunct should always be done first.*

The upshot of the adjacency theorem is that we need not search through all the possible orderings to find one guaranteed to be optimal. For example, in the preceding problem, we need look at only two orderings. In this case, we can eliminate two-thirds of the possibilities. As the number of literals grows, the savings becomes more substantial. A short analysis shows that the number of possible orderings that must be considered is

Table 5.1 Reduction of search space by adjacency restriction

n	$G(n,0)$	$n!$
1	1	1
2	1	2
3	2	6
4	5	24
5	16	120
6	61	720
7	272	5040
8	1385	40,320
9	7936	362,880
10	50,521	3,628,800

bounded by $G(n,0)$, where n is the number of literals and G is defined recursively, as shown.

$$G(n,d) = \begin{cases} 0 & \text{if } n = d \\ 1 & \text{if } n = 1, d = 0 \\ \sum_{i=0}^{n-d-1} G(n-1,i) & \text{otherwise} \end{cases}$$

Here, d can be thought of as the number of remaining literals that cannot appear as the next literal because of the adjacency restriction. Note that, if the second argument to G is ignored, the formula reduces to $n!$, as expected.

Table 5.1 shows some of the values for this function by comparison to the total number of orderings of n literals. For three literals, the adjacency restriction reduces the search space to only two orderings. For eight literals, the space is reduced from over 40,000 possibilities to fewer than 1400.

The adjacency theorem is an example of a *reduction theorem*. It reduces the space of possible orderings that must be searched to find an optimal ordering, and thereby makes the process of optimization more efficient.

5.9 Bibliographical and Historical Remarks

Many restriction strategies for resolution refutations are discussed in detail by Loveland [Loveland 1978], by Chang and Lee [Chang 1973], and by Wos et al. [Wos 1984a].

Ordered resolution is similar to *lock resolution*, which was originally proposed by Boyer [Boyer 1971], and to *SL-resolution*, which was explored by Kowalski [Kowalski 1971]. Depth-first backward resolution is the strategy used in PROLOG [Clocksin 1981, Sterling 1986], as well as in

numerous expert systems. Moore [Moore 1975] was one of the first people to point out the efficiencies to be gained by choosing the appropriate direction for reasoning. Treitel and Genesereth explored the problem of automatically determining optimal directionality [Treitel 1987]. The adjacency theorem for optimal literal ordering was proved by Smith and Genesereth [Smith 1985]. A variety of additional strategies for resolution are discussed in [Kowalski 1970, 1971, 1972, Minker 1973, 1979, Smith 1986].

Although not discussed in this book, it is often helpful to precompute all the possible resolutions that can be performed among a set of clauses and to store the results in a *connection graph.* The actual search for a refutation can then be described in terms of operations on this graph. The use of connection graphs was first proposed by Kowalski [Kowalski 1975]. Other authors who have used various forms of connection graphs are Sickel [Sickel 1976], Chang and Slagle [Chang 1979a, 1979b], and Stickel [Stickel 1982].

Several extremely efficient resolution refutation systems have been written that are able to solve large, nontrivial reasoning problems, including some open problems in mathematics [Winker 1982, Wos 1984b]. A typical challenge problem for testing and illustrating the features of theorem-proving programs is the so-called *Schubert steamroller problem* [Stickel 1986].

Several other nonresolution theorem-proving systems also have been developed. Examples include those of Bledsoe [Bledsoe 1977, Ballantyne 1977], and of Boyer and Moore [Boyer 1979]. Shankar used the Boyer–Moore theorem prover in verifying steps in the proof of Gödel's incompleteness theorem [Shankar 1986].

Exercises

1. *Deletion strategies.* Consider the problem of showing that the clauses {P,Q}, {¬P,Q}, {P,¬Q}, and {¬P,¬Q} are not simultaneously satisfiable.

   a. Show a resolution trace for this problem using tautology elimination.

   b. Show a resolution trace for this problem using subsumption.

2. *Linear resolution.* Use linear resolution to show that the following set of clauses is unsatisfiable.

   {P,Q}

   {Q,R}

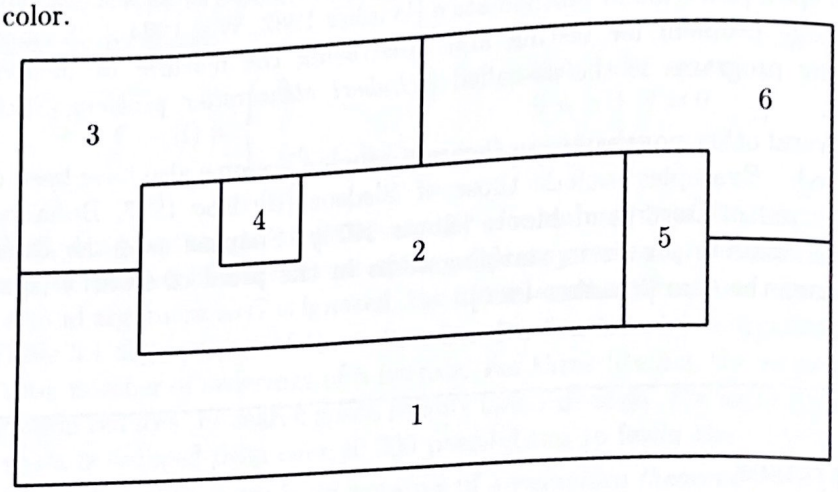
   {R,W}

$$\{\neg R, \neg P\}$$
$$\{\neg W, \neg Q\}$$
$$\{\neg Q, \neg R\}$$

3. *Combination strategies.* We know that unit resolution is not complete, but there are some problems for which it is able to derive the empty clause. If we combine unit resolution with ordered resolution, does this make it impossible to prove some things that are provable by unit resolution alone? If so, give an example. If not, prove that there is no difference.

4. *Combination strategies.* Give a counterexample to show that the combination of ordered resolution and set of support resolution is not complete.

5. *Map coloring.* Consider the problem of coloring the following map, using only four colors, such that no two adjacent regions share the same color.



This problem can be set up as a constraint satisfaction problem. Write down the database and the query.