Chapter 2

CSP solving — An overview

2.1 Introduction

This chapter gives an overview of CSP solving techniques, which can roughly be classified into three categories: *problem reduction*, *search* and *solution synthesis*. We shall also analyse the general characteristics of CSPs, and explain how these characteristics could be exploited in solving CSPs. Finally, we shall look at features of CSPs, as some of them could be exploited to develop specialized techniques for solving CSPs efficiently.

2.1.1 Soundness and completeness of algorithms

Definition 2-1:

> An algorithm is **sound** if every result that is returned by it is indeed a solution; in CSPs, that means any compound label which is returned by it contains labels for every variable, and this compound label satisfies all the constraints in the problem. ■

Definition 2-2:

> An algorithm is **complete** if every solution can be found by it. ■

Soundness and completeness are desirable properties of algorithms. Most of the algorithms described in this book are sound and complete unless otherwise specified. However, it is worth pointing out that some real life problems are intractable. In that case, incomplete (and sometimes even unsound) but efficient algorithms are sometimes considered acceptable. Examples of incomplete strategies are hill-climbing algorithms, which we discuss in Chapters 8 and 10.

2.1.2 Domain specific vs. general methods

It is generally believed that efficiency can be gained by encoding domain specific knowledge into the problem solver. For example, after careful analysis of the *N*-queens problem, one can find algorithms which solve it very efficiently [AbrYun89, Bern91]. However, there are good reasons for studying general algorithms. First, tailor made algorithms are costly. Second, tailored algorithms are limited to the problems for which they are designed. A slight change of the problem specification would render the algorithm inapplicable. Finally, general algorithms can often form the basis for the development of specialized algorithms.

The CSP is worth studying because it appears in a large number of applications. It also has specific characteristics which can be exploited for the development of specialized algorithms. This book is mainly concerned with CSP solving algorithms.

2.2 Problem Reduction

Problem reduction is a class of techniques for transforming a CSP into problems which are hopefully easier to solve or recognizable as insoluble. Although problem reduction alone does not normally produce solutions, it can be extremely useful when used together with search or problem synthesis methods. As we shall see in later chapters, problem reduction plays a very significant role in CSP solving.

2.2.1 Equivalence

Definition 2-3:

We call two CSPs **equivalent** if they have identical sets of variables and identical sets of solution tuples:

$$\forall\ \text{csp}((Z, D, C)), \text{csp}((Z', D', C')):$$
$$\text{equivalent}((Z, D, C), (Z', D', C')) \equiv$$
$$Z = Z' \wedge \forall\ T : (\text{solution\_tuple}(T, (Z, D, C)) \Leftrightarrow \text{solution\_tuple}(T, (Z', D', C'))) \blacksquare$$

Definition 2-4:

A problem $P = (Z, D, C)$ is **reduced** to $P' = (Z', D', C')$ if (a) P and P' are equivalent; (b) every variable's domain in D' is a subset of its domain in D; and (c) C' is more restrictive than, or as restrictive as, C (i.e. all compound labels that satisfies C' will satisfy C). We write the relationship between P and P' as *reduced*(P, P'):

$\forall$ csp((*Z*, *D*, *C*)), csp((*Z'*, *D'*, *C'*)):
reduced((*Z*, *D*, *C*), (*Z'*, *D'*, *C'*)) $\equiv$
equivalent((*Z*, *D*, *C*), (*Z'*, *D'*, *C'*)) $\wedge$
$(\forall\ x \in Z: D'_x \subseteq D_x) \wedge$
$(\forall\ S \subseteq Z: (C_S \in C \Rightarrow C'_S \in C' \wedge C'_S \subseteq C_S))$ ■

Since we see constraints as sets of compound labels, reducing a problem means removing elements from the constraints those compound labels which appear in no solution tuples. If constraints are seen as functions, then reducing a problem means modifying the constraint functions. For convenience, we define redundancy of values and redundancy of compound labels below.

Definition 2-5:

A value in a domain is redundant if it is not part of any solution tuple:

$\forall$ csp((*Z*, *D*, *C*)): $\forall\ x \in Z$: $\forall\ v \in D_x$:
redundant(*v*, *x*, (*Z*, *D*, *C*)) $\equiv$
$\neg\ \exists\ T$: (solution_tuple(*T* , (*Z*, *D*, *C*)) $\wedge$ projection(*T* , (<*x*,*v*>))) ■

Such values are called “redundant” because the removal of them from their corresponding domains does not affect the set of solution tuples in the problem.

Definition 2-6:

A compound label in a constraint is redundant if it is not a projection of any solution tuple:

$\forall$ csp((*Z*, *D*, *C*)): $\forall\ C_S \in C$: $\forall\ cl \in C_S$:
redundant(*cl* , (*Z*, *D*, *C*)) $\equiv$
$\neg\ \exists\ T$: (solution_tuple(*T* , (*Z*, *D*, *C*)) $\wedge$ projection(*T* , *cl*)) ■

Similarly, such compound labels are called redundant because the removal of them from their corresponding constraints does not affect the set of solution tuples in the problem.

2.2.2 Reduction of a problem

Problem reduction techniques transform CSPs to equivalent but hopefully easier problems by reducing the size of the domains and constraints in the problems. Problem reduction is possible in CSP solving because the domains and constraints are specified in the problems, and that constraints can be propagated.

Problem reduction involves two possible tasks: (1) removing redundant values from

the domains of the variables; and (2) tightening the constraints so that fewer compound labels satisfy them; if constraints are seen as sets, then this means removing redundant compound labels from the constraints. If the domain of any variable or any constraint is reduced to an empty set, then one can conclude that the problem is insoluble. Reduced problems are possibly, though not necessarily, easier to solve for the following reasons. Firstly, the domains of the variables in the reduced problem are no larger than the domains in the original problem. This leaves us with fewer labels to consider. Secondly, the constraints of the reduced problem are at least as tight as those in the original problem. This means that fewer compound labels need to be considered in the reduced problem.

Problem reduction requires one to be able to recognize redundant values and redundant compound labels. Such information can be deduced from the constraints. For example, if x and y are variables, and a constraint requires x to be greater than y in value, then we can remove from the domain of x all the values which are smaller than the smallest value in the domain of y. Similarly, we can remove from the domain of y all the values which are greater than the greatest value of x. Problem reduction algorithms will be discussed in Chapter 4.

Problem reduction is often referred to as *consistency maintenance* in the literature. Maintaining consistency of a problem means reducing a problem to one which has certain properties. Maintaining a different *consistency* means maintaining different properties in the problem, which will be explained in Chapter 3.

The use of the term *consistency* in the CSP literature may need some clarification. In logical systems, we call a system inconsistent if absurdity can be derived from it; for example, if $x < 0$ and $x > 4$ must both hold. In the CSP literature, a problem is called inconsistent with regard to a property when that property does not hold in the problem. Therefore, being inconsistent does not prevent a problem from being solvable. The use of the term *inconsistency* in these two different contexts should not be confused.

2.2.3 Minimal problems

Definition 2-7:

> A graph which is associated to a binary CSP is called a **minimal graph**[1] if no domain contains any redundant values and no constraint contains any redundant compound labels. In other words, every compound label in every binary constraint appears in some solution tuples:

1. Montanari [1974] defines *minimal networks* in binary constraint problems. According to our definitions of a graph and a network in Chapter 1, a binary CSP is in general associated with a constraint graph rather than a constraint network. Therefore, we shall use the term *minimal graph* instead of *minimal network.*

$\forall$ csp((*Z*, *D*, *C*)):
  minimal_graph((*Z*, *D*, *C*)) ≡
    ($\forall$ $x \in Z$: ($\neg \exists$ $v \in D_x$: redundant(*v*, *x*, (*Z*, *D*, *C*)))) ∧
      ($\forall$ $C_{y,z} \in C$: ($\neg \exists$ $cl \in C_{y,x}$: redundant(*cl* , (*Z*, *D*, *C*)))) ■

Montanari points out that although reducing a problem to its minimal graph is intractable in general, it may be feasible to reduce it to an approximation of its minimal graph — where some redundant values and redundant compound labels are removed.

Graphs can only represent binary CSPs. General CSPs (CSPs with general constraints) must be represented by hypergraphs. Therefore, we extend the concept of minimal graphs to general CSPs.

Definition 2-8:

A CSP is called a **minimal problem** if no domain contains any redundant values and no constraint contains any redundant compound labels:

$\forall$ csp((*Z*, *D*, *C*)):
  minimal_problem((*Z*, *D*, *C*)) ≡
    ($\forall$ $x \in Z$: ($\neg \exists$ $v \in D_x$: redundant(*v*, *x*, (*Z*, *D*, *C*)))) ∧
      ($\forall$ $C_S \in C$: ($\neg \exists$ $cl \in C_S$: redundant(*cl* , (*Z*, *D*, *C*)))) ■

In principle, there is nothing to stop one from reducing a problem to its minimal problem. This can be done by creating dummy constraints for all combinations of variables whenever necessary, and tightening each constraint to the set of compound labels which satisfy all the constraints (by checking all the compound labels in all constraints). When that is done, the constraint C_Z, where *Z* is the set of all variables, contains nothing but solution tuples. However, doing so is in general NP-hard, so most problem reduction algorithms limit their efforts to removing only those redundant values and compound labels which can be recognized relatively easily. Only in special cases will solutions be found by problem reduction alone. A number of algorithms combine problem reduction and searching, and these are discussed in Chapters 5.

2.3 Searching For Solution Tuples

Probably more research effort in CSP research has been spent on searching than in other approaches. In this section, we first describe a basic search algorithm, then analyse the properties of CSPs. Specialized search algorithms can be designed to solve CSPs efficiently by exploiting those properties.

2.3.1 Simple backtracking

The basic algorithm to search for solution tuples is **simple backtracking**, which is a general search strategy which has been widely used in problem solving (e.g. Prolog uses simple backtracking to answer queries). In the CSP context, the basic operation is to pick one variable at a time, and consider one value for it at a time, making sure that the newly picked label is compatible with all the labels picked so far. Assigning a value to a variable is called **labelling**. If labelling the current variable with the picked value violates certain constraints, then an alternative value, when available, is picked. If all the variables are labelled, then the problem is solved. If at any stage no value can be assigned to a variable without violating any constraints, the label which was last picked is revised, and an alternative value, when available, is assigned to that variable. This carries on until either a solution is found or all the combinations of labels have been tried and have failed. Figure 2.1 shows the control of BT.

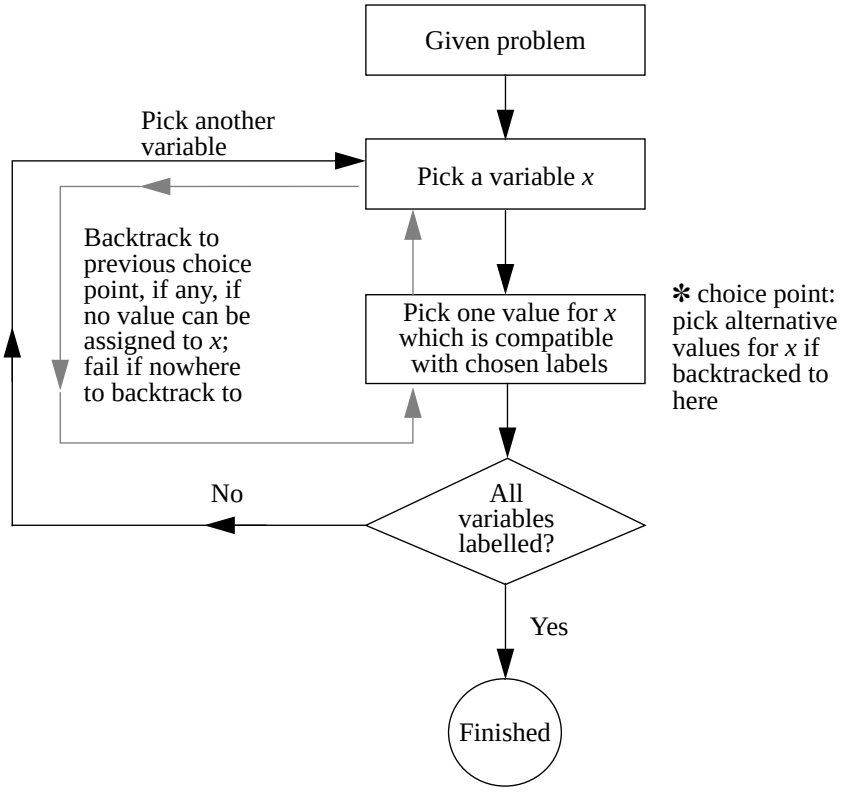


Figure 2.1 Control of the chronological backtracking (BT) algorithm

Since the BT algorithm will always backtrack to the last decision when it becomes unable to proceed, it is also called *chronological backtracking*. The pseudo code for the simple backtracking algorithm is shown in the Chronological_Backtracking and BT-1 procedures below.

```
PROCEDURE Chronological_Backtracking( Z, D, C );
BEGIN
    BT-1( Z, { }, D, C );
END


PROCEDURE BT-1( UNLABELLED, COMPOUND_LABEL, D, C );
/* UNLABELLED is a set of variables to be labelled; */
/* COMPOUND_LABEL is a set of labels already committed to */
BEGIN
  IF (UNLABELLED = { }) THEN return(COMPOUND_LABEL)
  ELSE BEGIN
    Pick one variable x from UNLABELLED;
    REPEAT
      Pick one value v from Dx;
      Delete v from Dx;
      IF COMPOUND_LABEL + {<x,v>} violates no constraints
      THEN BEGIN
        Result ←
          BT-1(UNLABELLED – {x}, COMPOUND_LABEL +
            {<x,v>}, D, C);
        IF (Result ≠ NIL) THEN return(Result);
      END
    UNTIL (Dx = { });
    return(NIL); /* signifying no solution */
  END /* of ELSE */
END /* of BT-1*/
```

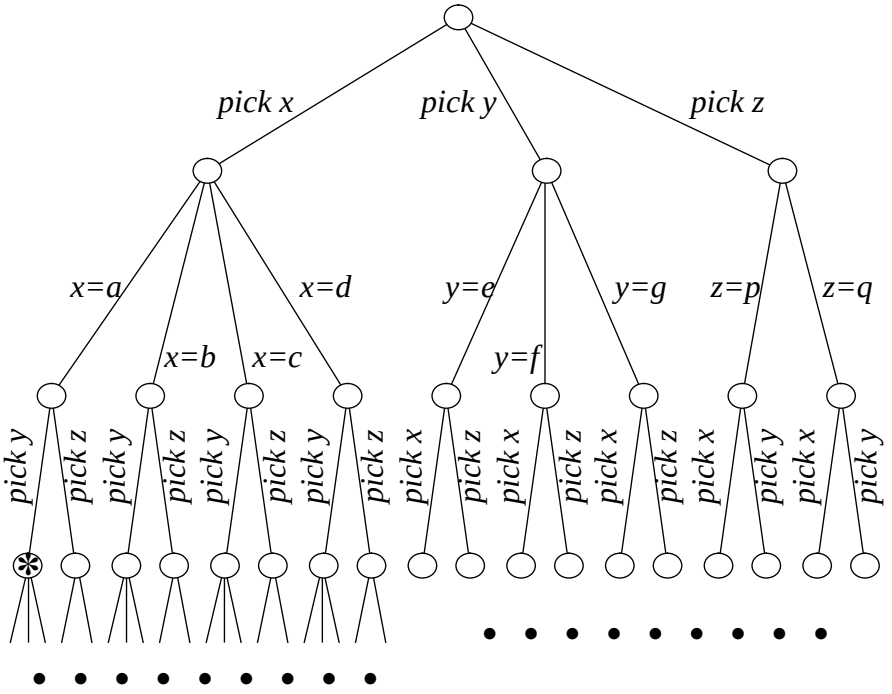
Let n be the number of variables, e be the number of constraints, and a be the domain sizes of the variables in a CSP. Since there are altogether a^n possible combinations of n-tuples (candidate solutions), and for each candidate solution all the constraints must be checked once in the worst case, the time complexity of this backtracking algorithm is $O(a^n e)$.

To store the domains of the problem requires $O(na)$ space. The BT algorithm does not require more temporary memory than $O(n)$ to store the compound label. Therefore, the space complexity of Chronological_Backtracking is $O(na)$.

The time complexity above shows that search efficiency could be improved if a can be reduced. This could be achieved by problem reduction techniques, as mentioned in the previous section. The combination of problem reduction and searching will be discussed in greater detail later.

2.3.2 Search space of CSPs

The search space is the space of all those states which a search could possibly arrive at. Since different sets of variables and search paths could be introduced in different problem formalization, the search space could be different from one formalization to another. The BT algorithm searches in the space of all compound labels. Its search space is shown in Figure 2.2.



✻ represents the state in which $x = a$ and y is picked for labelling

Figure 2.2 Search space of BT in a CSP (Z, D, C) when the variables are not ordered; here: $Z = \{x, y, z\}$, $D_x = \{a,b,c,d\}$, $D_y = \{e, f, g\}$ and $D_z = \{p, q\}$

The node marked ✻ in Figure 2.2 represents a state in the search space in which x is labelled with a, and y is picked for labelling but not yet assigned a value. Note that the constraints play no part in the definition of the search space, although, as it will become clear later, it affects the search space that needs to be explored by an algorithm. If we assume a fixed ordering among the variables in our search, then the search space of BT is the tree shown in Figure 2.3.

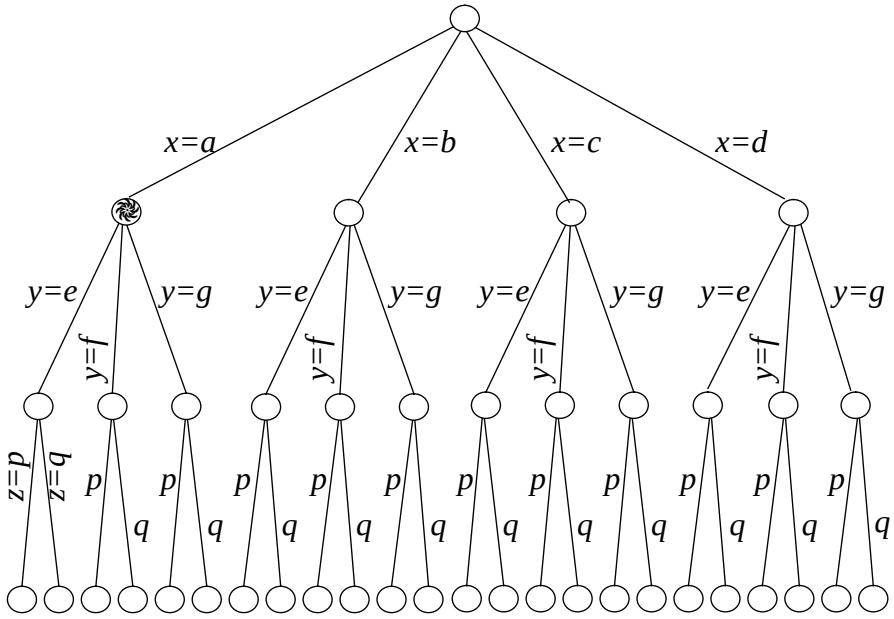


Figure 2.3 Search space for a CSP (Z, D, C), given the ordering (x, y, z), where $Z = \{x, y, z\}$, $D_x = \{a, b, c, d\}$, $D_y = \{e, f, g\}$ and $D_z = \{p, q\}$

Each node X in the search space represents a state in which a compound label is committed to, and each of its children represents a state in which an extra label is added into the compound label in X. For example, the node marked ✻ in Figure 2.3 represents a state in the search space in which x is assigned the value a, and y and z are yet to be labelled. The search space is different if the variables are searched in another fixed order. For example, Figure 2.4 shows the search space under the variable ordering (z, y, x).

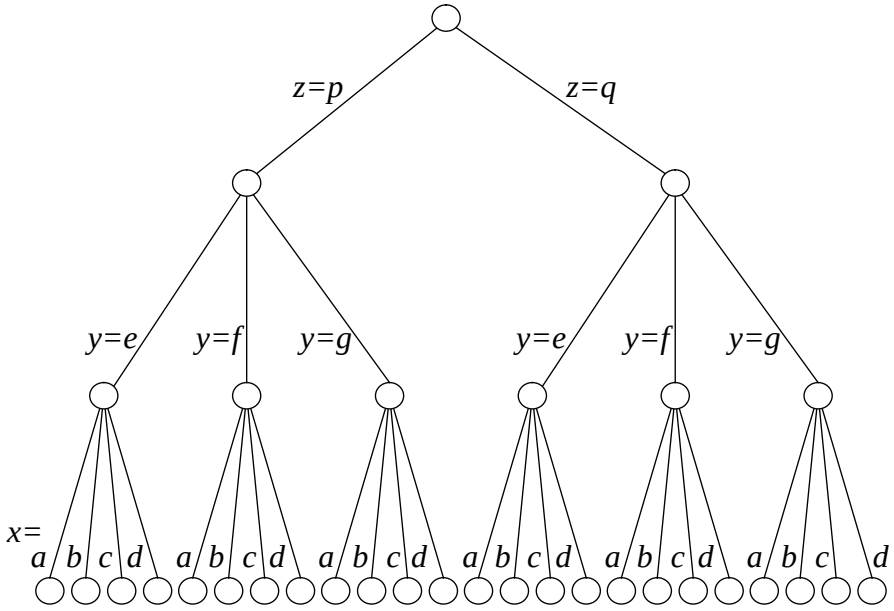


Figure 2.4 Alternative organization of the search space for the csp (Z, D, C) in Figure 2.3, given the ordering (z, y, x), where Z = {x, y, z}, D_x = {a, b, c, d}, D_y = {e, f, g} and D_z = {p, q}

2.3.3 General characteristics of CSP's search space

There are properties of CSPs which differentiate them from general search problems. It is the presence of these properties that makes problem reduction possible in CSPs. Besides, specialized search techniques can be, and have been, developed to exploit these properties so as to solve CSPs more efficiently. These algorithms will be described in detail later in the book. To help understand why those techniques work, we shall describe these properties here:

(1) The size of the search space is finite

The number of leaves of the search tree is $L = \left|D_{x_1}\right| \cdot \ldots \cdot \left|D_{x_n}\right|$, where D_{x_i} is the domain of variable x_i, and $\left|D_{x_i}\right|$ is the size of this domain. Notice that L is not affected by the ordering in which we decide to label the variables. However, this ordering does affect the number of internal nodes in the search

space. For example, the total number of internal nodes in the search space in Figure 2.3 is 16, as opposed to 8 in Figure 2.4. In general, if we assume that the variables are ordered as x_1, x_2, ..., x_n, the number of nodes in the search tree is:

$$1 + \sum_{i=1}^{n} (|D_{x_1}| \cdot \ldots \cdot |D_{x_i}|), \quad \text{or} \quad 1 + \sum_{i=1}^{n} (\prod_{j=1}^{i} |D_{x_j}|)$$

With this formula, together with our examples in Figures 2.3 and 2.4, it is not difficult to see that if the variables were ordered by their domain sizes in descending order, then the number of nodes in the search space would be maximal. That should also be the upper bound of the size of the search space. If the variables were ordered by their domain sizes in ascending order, then the number of nodes in the search space would be minimal. However, the size of the problem is dominated by the last and most significant term, $|D_{x_1}| \cdot \ldots \cdot |D_{x_n}|$.

(2) The depth of the tree is fixed

When the variables are ordered, the depth of the search tree is always equal to the number of variables in the problem regardless of the ordering. In both Figures 2.3 and 2.4, the depth of the search tree is 3. When the ordering of the variables is not fixed, the depth of the tree is exactly *2n*, where *n* is the number of variables (see Figure 2.2).

(3) Subtrees are similar

If we fix the ordering of the variables, then the subtrees under each branch of the same level are identical in their topology. In Figures 2.3 and 2.4, the same choices are available in all sibling subtrees. Figure 2.2 shows that even when the variables are not given a fixed ordering, similar choices are available in sibling subtrees.

The fact that the subtrees are similar means that experiences in searching one subtree may be useful in subsequently searching its siblings. This makes learning possible (discussed in Chapter 5).

2.3.4 Combining problem reduction and search

Efficiency of a backtracking search can be improved if one can prune off search spaces that contain no solution. This is precisely where problem reduction can help. Earlier we said that problem reduction reduces the size of domains of the variables and tightens constraints. Reducing the domain size of a variable is effectively the same as pruning off branches in the search space. Tightening constraints potentially helps us to reduce the search space at a later stage of the search. Problem reduction could be performed at any stage of the search. Various search strategies combine problem reduction and search in various ways (described in detail in Chapters 5 to 7). Some of these strategies have been proved to be extremely effective.

In general, the more redundant values and compound labels one attempts to remove, the more computation is required. One the other hand, the less redundant values and compound labels one removes, the more time one is likely to spend in backtracking. One often has to find a balance between the efforts made and the potential gains in problem reduction. Figure 2.5 roughly shows the relationship between the two.

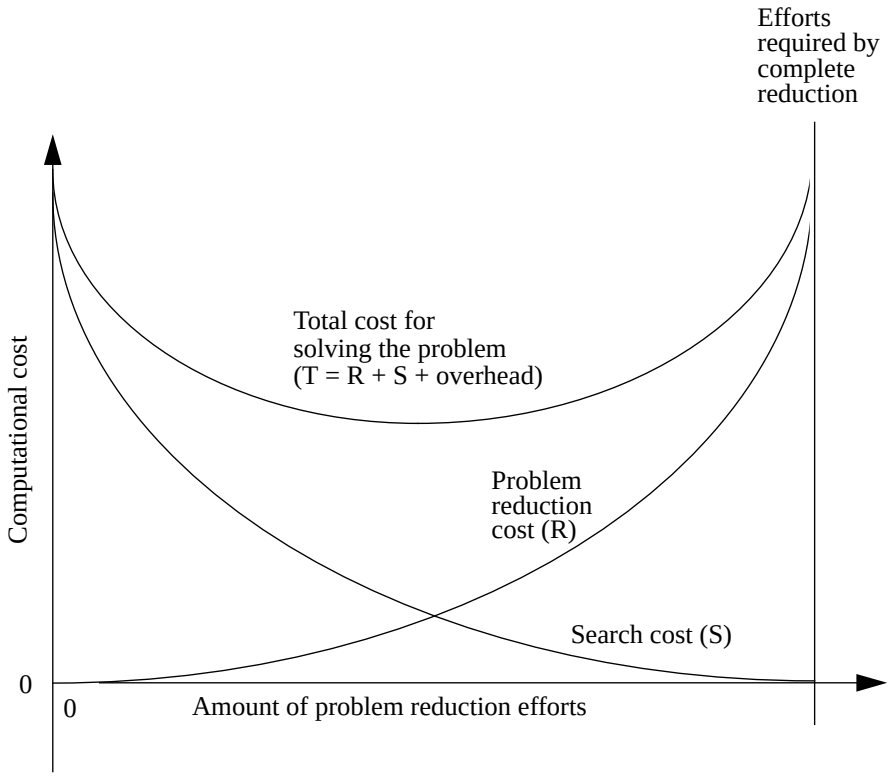


Figure 2.5 Cost of problem reduction vs. cost of backtracking (the more effort one spends on problem reduction, the less effort one needs in searching)

2.3.5 Choice points in searching

There are three sets of choice points in the chronological backtracking algorithm above:

(1) which variable to look at next?
(2) which value to look at next?
(3) which constraint to examine next?

The first two choice points are shown in the flow chart in Figure 2.1. Different search space will be explored under different ordering among the variables and values. Since constraints can be propagated, the different orderings in which the variables and values are considered could affect the efficiency of a search algorithm. This is especially significant when a search is combined with problem reduction, as committing to different branches of the search tree may cause different amounts of the search space to be pruned off.

For problems in which a single solution is required, search efficiency could be improved by the use of *heuristics* — rules which guide us to look at those branches in the search space that are more likely to lead to solutions.

In some problems, checking whether a constraint is satisfied is itself computation expensive. In that case, the ordering in which the constraints are examined could significantly affect the efficiency of an algorithm. If the situation is over-constrained, then the sooner the violated constraint is examined, the more computation one could save.

2.3.6 Backtrack-free search

In Chapter 1, we defined the basic concepts of constraints and satisfiability. In this section, we extend our discussion on these concepts.

Definition 2-9:

> A **constraint expression** on a set of variables *S*, which we denote by *CE(S)*, is a collection of constraints on *S* and its subset of variables. ■

Definition 2-10:

> A **constraint expression** on a subset of variables *S* in a CSP P, denoted *CE(S,* P*)*, is the collection of all the relevant constraints in P on *S* and its subset of variables:
>
> $\forall$ csp((*Z*, *D*, *C*)): $\forall$ $S \subseteq Z$: (CE(*S*, (*Z*, *D*, *C*)) $\equiv$ $\{C_Y \mid Y \subseteq S \wedge C_Y \in C\}$) ■

It should not be difficult to see that the problem designation (*Z*, *D*, *C*) can be written as (*Z*, *D*, CE(*Z*, (*Z*, *D*, *C*))).

Definition 2-11:

A **compound label** *CL* **satisfies a constraint expression** *CE* if *CL* satisfies all the constraints in *CE*:

$\forall$ csp((*Z*, *D*, *C*)): $\forall$ $x_1, x_2, ..., x_k \in Z$: ($\forall$ $v_1 \in D_{x_1}$, $v_2 \in D_{x_2}$, ...,$v_k \in D_{x_k}$:
$\forall$ $S \subseteq \{x_1, x_2, ..., x_k\}$:
satisfies((<x_1,v_1><x_2,v_2>...<x_k,v_k>), CE(*S*, (*Z*, *D*, *C*))) ≡
$\forall$ C_R: ($C_R \in$ CE(*S*, (*Z*, *D*, *C*)) ⇒
satisfies((<x_1,v_1><x_2,v_2>...<x_k,v_k>), C_R))) ■

Definition 2-12:

A search in a CSP is **backtrack-free** in a depth first search under an ordering of its variables if for every variable that is to be labelled, one can always find for it a value which is compatible with all the labels committed to so far:

$\forall$ csp((*Z*, *D*, *C*)): ($\forall$ <: total_ordering(*Z*, <):
backtrack-free((*Z*, *D*, *C*), <) ≡
($\forall$ $x_1, x_2, ..., x_m \in Z$: ($x_1 < x_2 < ... < x_m$ ⇒
($\forall$ $v_1 \in D_{x_1}$, $v_2 \in D_{x_2}$, ..., $v_m \in D_{x_m}$:
(satisfies((<x_1,v_1>... <x_m,v_m>), CE({x_1,...,x_m}, (*Z*, *D*, *C*))) ⇒
($\forall$ $y \in Z$: ($x_m < y$ ⇒ $\exists$ $a \in D_y$:
satisfies((<x_1,v_1>...<x_m,v_m><y,a>),
CE({x_1,...,x_m,y}, (*Z*, *D*, *C*))))))) ■

A number of strategies have been developed to make search backtrack-free in CSPs. They will be discussed later in this book.

2.4 Solution Synthesis

In this section, we shall give an overview of the solution synthesis approach in CSP solving. Solution synthesis can be seen as search algorithms which explore multiple branches simultaneously. It can also be seen as problem reduction in which the constraint for the set of all variables (i.e. the *n*-constraint for a problem with *n* variables) is created, and reduced to such a set that contains all the solution tuples, and solution tuples only. The distinctive feature of solution synthesis is that solutions are constructively generated.

In searching, one partial solution (which is a compound label) is looked at at a time. A compound label is extended by adding one label to it at a time, until a solution tuple is found or all the compound labels have been exhausted. The basic idea of

solution synthesis is to collect the sets of all legal labels for larger and larger sets of variables, until this is done for the set of all variables. To ensure soundness, a solution synthesis algorithm has to make sure that all illegal compound labels are removed from this set. To ensure completeness, the algorithm has to make sure that no legal compound label is removed from this set. A naive solution synthesis algorithm is shown in the pseudo code Naive_synthesis, and the synthesis process is shown in Figure 2.6.

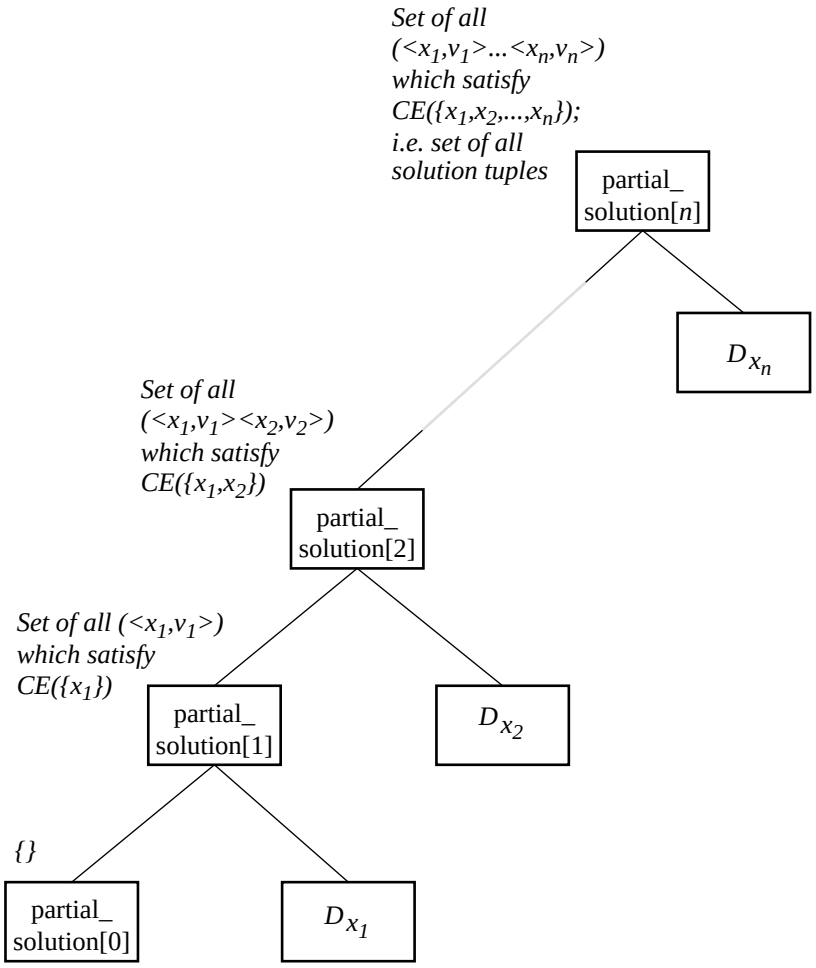


Figure 2.6 A naive solution synthesis approach

```
PROCEDURE Naive_synthesis(Z, D, C)
BEGIN
    order the variables in Z as x1, x2, ..., xn;
    partial_solution[0] ← {()};
    FOR i = 1 to n DO
        BEGIN
            partial_solution[i] ← { cl + <xi, vi> | cl ∈ partial_solution[i–1]
                ∧ vi ∈ Dxi ∧ cl + <xi, vi> satisfies all the constraints on
                variables_of(cl) + xi };
        END
    return(partial_solution[n]);
END /* of Naive_synthesis */
```

Solution synthesis was first introduced by Freuder [1978]. Freuder's algorithm and other solution synthesis algorithms will be explained in detail in Chapter 9.

2.5 Characteristics of Individual CSPs

CSPs are NP-hard in general, but every CSP is unique, and it is quite possible to develop specialized techniques to exploit the specific features of individual CSPs. Indeed, some such techniques have been developed, and they will be explained in Chapter 7. In this section, we shall list some of the most commonly studied characteristics of CSPs. This will help us to relate the different CSP solving techniques to the problems for which they are particularly effective when these techniques are introduced in subsequent chapters.

2.5.1 Number of solutions required

Some applications require a single solution and some require all solutions to be found. Examples of problems which require single solutions are scene labelling in vision, scheduling jobs to meet deadlines, and constructive proof of the consistency of a temporal constraints network. Examples of problems where all solutions are required are logic programming where all variable bindings are to be returned, and scheduling where all possible schedules are to be returned for comparison.

Problems which require a single solution favour techniques which have a better chance of finding solutions at an earlier stage. The ordering of the variables and the values in searching is especially significant in solving such problems. Consequently, heuristics for ordering variables and values could play an important role in solving them. Solution synthesis techniques are normally used to generate all solutions.

2.5.2 Problem size

The size of a problem could be measured by the number of variables, the domain sizes, the number of constraints, or a combination of all three.

We mentioned earlier that the number of variables determines the depth of the search tree. The domain sizes determine the branching factors (number of branches) at the nodes. The number of leaves in the search tree, $\prod_{\forall (x \in Z)} |D_x|$, dominates the size of the search tree (see Section 2.3.3). This is probably the most commonly used criteria for measuring the size of a problem, but the number of constraints in a problem should not be overlooked. The more constraints there are in a problem, the more compatibility checks one is likely to require in solving it. On the other hand, constraints could help one to prune off part of the search space, and therefore reduce the total number of consistency checks.

Small problems are only difficult when compatibility checks are computationally expensive. For such problems, techniques which minimize the number of compatibility checks necessary should be favoured.

2.5.3 Types of variables and constraints

The type of variable affects the techniques that one can apply. Most of the techniques described in this book focus on symbolic variables. If all the variables in a problem are numbers and all the constraints are conjunctive linear inequalities, then *integer programming* or *linear programming*, both studied extensively in operations research (OR), are appropriate tools for handling it.

A number of CSP solving techniques have been developed for binary CSPs. Normally, constraints can be propagated more effectively through binary constraints rather than through general constraints. For example, if $X + Y < 10$ is a constraint, then the values of X and Y determine one another. If one commits to $X = 2$, then one can immediately remove from the domain of Y all the values which are greater than 7. But if the constraint is $A + B + C < 10$, then committing to $A = 2$ leaves us with $B + C < 8$, which will not allow us to reduce the domains of B and C until either B or C is fixed.

2.5.4 Structure of the constraint graph in binary-constraint-problems

We mentioned in Chapter 1 that associated to each CSP is a hypergraph. Associated to each binary CSP is a graph. We also mentioned that the efficiency of a search is affected by the ordering of the variables in the search. In fact, the efficiency of a search in an ordering is significantly affected by the connectivity of the nodes in the constraint hypergraph. In Chapters 6 and 7, we shall explain that when the con-

straint graph/hypergraph is or can be transformed into a tree, the problem can be solved in polynomial time. Some heuristics also exploit the connectivity of the nodes.

Definition 2-13:

A **complete graph** is a graph in which an edge exists between every two nodes:

$\forall$ graph((V, E)): complete_graph((V, E)) $\equiv E = \{ (x, y) \mid x, y \in V \}$ ■

When the graph is not complete, the connectivity of the nodes (i.e. the structure of the graph) can be exploited to improve search efficiency. One well known heuristic is the *adjacency heuristic*, which suggests that after labelling a variable X, one should choose a variable which is connected to X as the next variable to label. This idea is extended to more complex heuristics such as the *minimal bandwidth ordering* heuristics, which will be discussed in Chapter 6 alongside other variable ordering techniques.

2.5.5 Tightness of a problem

Problems can be characterized by their **tightness**, which could be measured under the following definition.

Definition 2-14:

The **tightness of a constraint** C_S is measured by the number of compound labels satisfying C_S over the number of all compound labels on S:

$\forall$ csp((Z, D, C)): $\forall C_{x_1, x_2, \ldots, x_k} \in C$:

$$\text{tightness}(C_{x_1, x_2, \ldots, x_k}, (Z, D, C)) \equiv \frac{s}{T}$$

where:

s = number of compound labels satisfying $C_{x_1, \ldots, x_k}$

$= \left| \{(<x_1,v_1>...<x_k,v_k>) \mid \text{satisfies}((<x_1,v_1>...<x_k,v_k>), C_{x_1, \ldots, x_k})\} \right|$

T = maximum number of compound labels for $x_1, .. x_k = \prod_{i=1}^{k} \left|D_{x_i}\right|$ ■

Definition 2-15:

The **tightness of a CSP** is measured by the number of solution tuples over

the number of all distinct compound labels for all variables:

$$\forall\ \text{csp}((Z, D, C))\text{: tightness}((Z, D, C)) \equiv \frac{|S|}{(\prod_{\forall\,(x\,\in\,Z)} |D_x|)}$$

where S = the set of all solution tuples = $\{T \mid \text{solution\_tuples}(T, (Z,D,C))\}$ ■

Tightness is a relative measure. Some CSP solving techniques are more suitable for tighter problems, while others are suitable for looser problems. In principle, the tighter the constraints, the more effectively can one propagate the constraints, which makes problem reduction more effective. Partly because of this, problems with tighter constraints need not be harder to solve than loosely constrained problems. Whether a problem is easier or harder to solve depends on the tightness of the problem combined with the number of solutions required.

For loose problems, many leaves of the search space represent solutions. Therefore, a simple backtracking algorithm like Chronological_Backtracking would not require much backtracking before a solution can be found. A strategy which combines searching and problem reduction is likely to spend its efforts unnecessarily in attempting to reduce the problem. However, if all solutions are required, then a loosely constrained problem becomes harder by its very nature. This is because of the fact that, since the problem is loosely constrained, a large proportion of the search space lead to solutions. Since all solutions are required, a larger search space has to be explored.

The tighter a problem is, the more backtracking a naive backtracking algorithm is likely to require to find solutions. Therefore, tighter problems are harder to solve if a single solution is required. However, when all solutions are required, looser problems becomes harder to solve. The tighter a problem is, the more likely it becomes that domains can be reduced through constraint propagation (see problem reduction strategies in Chapters 3 and 4); consequently, a smaller space needs to be searched to find all the solutions. Table 2.1 summarizes the conclusions made in this section.

2.5.6 Quality of solutions

In applications such as industrial scheduling, the objective is often to find single solutions. However, not all solution tuples are as good as one another. For example, assigning different machines (value) to the same job (variables) could incur different costs. It might also affect the production time. Given an optimization function (for instance, to minimize the cost or the production time) the requirement is to find the optimal or near-optimal solution tuple(s), rather than finding any solution tuple. If the variables are numbers, and the constraints are inequalities, then linear programming or integer programming may be useful for finding optimal solutions.

Table 2.1 Relating difficulty of problems, tightness and number of solutions

Solutions required	Tightness of the problem	
	Loosely constrained	Tightly constrained
Single solution required	Solutions can easily be found by simple backtracking, hence such problems are easy	Simple backtracking may require a lot of backtracking, hence harder compared with loose problems
All solutions required	More space needs to be searched, hence such problems could be harder than tightly constrained problems	Less space needs to be searched, hence, given the right tools, could be easier than loosely constrained problems

A CSP in which the optimal solution is required is akin to a problem in which all solutions are required. A naive approach is to look at all the solutions in order to choose the best. However, in some applications one may be able to find heuristics to help pruning off search space which has no hope of containing solutions that are better than the best solution found so far. When such heuristics are available, which is the case in many applications, we can use search strategies called *branch and bound* to solve the problem without looking at all solutions.

In industrial scheduling, the environment changes dynamically (e.g. machines may break down from time to time, different jobs may be given different priority at different times, etc.). Under such situations, near-optimal solutions are often sufficient because optimal solutions at the point when it is generated may become suboptimal very soon. For such applications, stochastic search techniques are often used. Techniques for finding optimal and near-optimal solutions will be discussed in Chapters 8 and 10.

2.5.7 Partial solutions

Not every CSP is solvable. In many applications, problems are mostly over-constrained. When no solution exists, there are basically two things that one can do. One is to relax the constraints, and the other is to satisfy as many of the requirements as possible. The latter solution could take different meanings. It could mean labelling as many variables as possible without violating any constraints. It could also mean labelling all the variables in such a way that as few constraints are violated as possible. Such compound labels are actually useful for constraint relaxation because they indicate the minimum set of constraints which need to be violated. Furthermore, weights could be added to the labelling of each variable or each constraint violation. In other words, the problems are:

(1) to maximize the number of variables labelled, where the variables are possibly weighted by their *importance*;
(2) to minimize the number of constraints violated, where the constraints are possibly weighted by their *costs*.

These are optimization problems, which are different from the standard CSPs defined in Definition 1-12. This class of problems is called the Partial CSP (PCSP), and will be discussed in Chapter 10.

2.6 Summary

We have given an overview of Constraint Satisfaction Problem solving approaches, and have proposed the classification of techniques in CSP solving into three categories:

(1) problem reduction: to reduce the problem to problems which are hopefully easier to solve or recognizable as insoluble;
(2) search: to enumerate combinations of labels so as to find solutions. It is often used together with problem reduction;
(3) solution synthesis: to construct and extend partial solutions in order to generate the set of all solution tuples.

Since domains in a CSP are known in advance, constraints can be used to identify redundant values and redundant compound labels — values and compound labels which will never appear in any solutions. Problem reduction is concerned with the removal of redundant values and redundant compound labels (i.e. to tighten constraints). The reduced CSP is hopefully easier to solve.

Search is probably the most studied approach in CSP research. We have pointed out that specialized search techniques can be developed to take advantage of properties that are special to CSPs. Such properties include the fact that choice points are known in advance (because the variables and their domains are fixed given any CSP). This allows one to fix or shape the search space before searching starts. Besides, in a search tree, all the sibling subtrees under a choice point are very similar. Since the search space is known in advance, it is possible to prune off search spaces after committing to a certain branch (constraint propagation). This leads to *look ahead algorithms*. Since sibling subtrees are very similar, one can *learn* from failures in a search. Both look ahead algorithms and learning algorithms are explained in Chapter 5.

Understandably, every CSP solving technique is applicable to and effective in a subset of CSPs. In this chapter we have listed some of the most studied problem-specific characteristics of CSPs, and outlined the classes of techniques which are relevant to each of them. Being able to relate CSP solving techniques to problem-

specific characteristics is important, because it allows one to pick the most relevant techniques for a given problem. When discussing the CSP solving techniques in detail, we shall also identify the problem-specific characteristics which when present make these techniques applicable or effective.

2.7 Bibliographical Remarks

Meseguer [1989] and Kumar [1992] give overviews of CSP solving. The minimal network (which we refer to as the minimal graph) concept is introduced by Montanari [1974]. Mackworth [1977] elaborates on Montanari's work, and introduces a number of problem reduction strategies and algorithms. Later work such as Freuder [1978, 1982, 1990], Mackworth & Freuder [1985], Cooper [1989] and Tsang [1989] analyse and extend problem reduction concepts and techniques. Haralick & Elliott [1980] summarize some of the most important search strategies for CSP solving. Work on search techniques for CSPs is abundant (see bibliographical remarks in Chapters 5 to 8).

Backtrack-free search is studied by Dechter & Pearl [1988a]. Freuder [1985] extends the concept of backtrack-free search to backtrack-bound search. Examples of incomplete search strategies are *hill-climbing* (e.g. see Nilsson, 1980 and Minton *et al.*, 1992), *staged search* [DorMic66], *beam search*, *wave search* [Fox87] and *stochastic search* [Glov89,90, TsaWar90, WanTsa91,92, TsaWan92]. Work on solution synthesis include [Freu78], [Seid81] and [TsaFos90]. Bibel [1988] tackles CSPs from a deductive viewpoint, which is closely related to solution synthesis. Recently, Vempaty [1992] proposed tackling CSPs using finite state automata.