Chapter 2

SEARCH

The main issue in search is deciding where to look next. There are many ways to do this, but exhaustive search methods have generally taken one of two forms—branching and constraint-directed search.

Branching is a divide-and-conquer strategy that is essentially guided by problem difficulty. If a problem is too hard to solve, it is broken into two or more subproblems that are more highly restricted and perhaps easier to solve. The process generates a search tree whose leaf nodes correspond to subproblems that can be solved or shown infeasible. The best solution found is optimal for the original problem.

Constraint-directed search is guided by past experience. Whenever a solution is examined, a nogood constraint is generated that excludes it and perhaps other solutions that can be no better. The next solution examined must satisfy the nogood constraints generated so far, to avoid covering the same ground. The search is over when there is no ground left to cover, and the best solution found is optimal.

This chapter explores the two basic search strategies and the roles played by inference and relaxation in each. But this is only part of its purpose. An equally important objective is to show how an integrated approach to optimization actually plays itself out in some concrete cases. To this end, it presents several small examples that cover a wide range of application areas. It each case it formulates a model that is appropriate for integrated solution, and in most cases it carries the solution procedure to completion.

The chapter examines three branching schemes: branch and infer, branch and relax, and branch and price. It then describes three types of constraint-directed search: constraint-directed branching, partial-order dynamic backtracking, and logic-based Benders decomposition. It con-

cludes by showing that branching and constraint-directed search schemes can be seen at work in heuristic as well as exact methods. This allows one to understand exhaustive and local search as belonging to a single algorithmic framework.

The chapter begins with a general description of the search-infer-and-relax solution process that is developed in this book. This provides an opportunity to introduce some of the terminology and notation used thereafter.

2.1 The Solution Process

For the purposes of this book, an optimization problem can be written

$$\begin{array}{l} \min\,(\text{or max})\ f(x) \\ \mathcal{S} \\ x \in D \end{array} \tag{2.1}$$

where $f(x)$ is a real-valued function of variable x and D is the *domain* of x. The function $f(x)$ is to be minimized (or maximized) subject to a set $\mathcal{S}$ of constraints, each of which is either satisfied or violated by any given $x \in D$. Generally, x is a tuple $(x_1, \ldots, x_n)$ and D is a Cartesian product $D_1 \times \cdots \times D_n$, where each $x_j \in D_j$.

Any $x \in D$ is a *solution* of (2.1). A *feasible* solution is one that satisfies all the constraints in $\mathcal{S}$, and the *feasible set* of (2.1) is the set of feasible solutions. A feasible solution x^* is *optimal* if $f(x^*) \leq f(x)$ for all feasible x. An *infeasible* problem is one with no feasible solution. If (2.1) has no feasible solution, it is convenient to say that it has optimal value ∞ (or $-\infty$ in the case of a maximization problem). The problem is *unbounded* if there is no lower bound on $f(x)$ for feasible values of x, in which case the optimal value is $-\infty$ (or ∞ for maximization).

It is assumed throughout this book that (2.1) is either infeasible, unbounded, or has a finite optimal value. Thus such problems as minimizing x subject to $x > 0$ are not considered. An optimization problem is considered to be *solved* when an optimal solution is found, or when the problem is shown to be unbounded or infeasible. In incomplete search methods that do not guarantee an optimal solution, the problem is solved when a solution is found that is acceptable in some sense, or when the problem is shown to be unbounded or infeasible.

A constraint C can be *inferred* from $\mathcal{S}$ if it is *valid* for $\mathcal{S}$, meaning that all $x \in D$ satisfying $\mathcal{S}$ satisfy C. A *relaxation* of the minimization problem (2.1) is obtained, roughly speaking, by dropping constraints or replacing the objective function with a lower bound. A more precise

definition allows for the possibility that a relaxed problem may introduce additional variables. Thus the problem

$$\begin{array}{l} \min\ f'(x,y) \\ \mathcal{S}' \\ x \in D,\ y \in D' \end{array} \tag{2.2}$$

is a relaxation of (2.1), if for any x feasible in (2.1) there is a $y \in D'$ such that (x, y) is feasible in (2.2) and $f'(x, y) \leq f(x)$. The optimal value of a relaxation is always a lower bound on the optimal value of the original problem.

Search, inference, and relaxation interact to provide a general scheme for solving (2.1). The search procedure solves a series of restrictions or special cases of the problem. If the set of restrictions is exhaustive, the best optimal solution of a restriction is optimal in the original problem. Inference and relaxation provide opportunities to exploit problem structure, a key element of any successful approach to solving combinatorial problems. Inference and relaxation also obtain information that helps direct the search.

2.1.1 Search

Search is carried out by solving a series of problem *restrictions*, which may be denoted $P_1, P_2, \ldots, P_m$. Each P_k is obtained by adding constraints to the original problem P. The search is *exhaustive* if the restrictions are *exhaustive*; that is, the feasible set of P is equal to the union of the feasible sets of $P_1, \ldots, P_m$.

The rationale for search is that restrictions may be easier to solve than the original problem. Thus it may be practical to solve P by solving $P_1, \ldots, P_m$ to optimality and picking the best solution. Any optimal solution of a restriction, which might be called a *candidate solution*, is feasible in P (some restrictions may be infeasible and have no solution). If the search is complete, the best candidate solution is an optimal solution for P. In incomplete search, the restrictions may not be solved to optimality, as is often the case in local search schemes.

The most basic kind of search simply enumerates elements of the domain D and selects the best feasible solution. This can be viewed as a search over problem restrictions P_k, each of which is defined by fixing x to a particular value. It is generally more practical, however, to define restrictions by branching, constraint-directed search, or an incomplete version of these. Table 2.1 indicates briefly how they fit into the search-infer-and-relax framework. The concept of a selection function is discussed below in connection with constraint-directed search.

Table 2.1. How branching, constraint-directed search, and heuristic methods fit into the search-infer-and-relax framework.

Solution method	*Restriction* P_k	*Relaxation* R_k	*Selection function* $s(R_k)$	*Inference*
Branching	Corresponds to node of search tree	LP, Lagrangean, convex NLP, domain store	Any optimal (feasible) solution of R_k	Domain filtering, cutting planes
Constraint-directed search	Original problem plus processed nogoods	Processed nogoods generated so far	Solution that results in easy R_{k+1}, $R_{k+2}, \ldots$	Nogood generation and processing
Heuristic method	Neighborhood of current solution	Problem specific	Random solution, best solution, etc.	Tabu list

Note: LP and NLP refer to linear programming and nonlinear programming, respectively.

2.1.2 Inference

Search can often be accelerated by inference—that is, by inferring new constraints from the constraint set of each P_k. The new constraints are then added to the constraint set, which may make the restrictions easier to solve by describing the feasible set more explicitly. Common forms of inference are domain filtering in constraint programming methods and cutting plane generation in integer programming methods. Adding inferred constraints may also result in a stronger relaxation, as in the case of cutting planes.

Inference is most effective when it exploits problem structure. When a group of constraints has special characteristics, the model can indicate this by combining the constraints into a single *metaconstraint*, known as a *global constraint* in constraint programming. This allows inference algorithms to exploit the global structure of the group in order to derive strong implications. For instance, if variables $x_1, \ldots, x_n$ must take distinct values, one can write a single all-different constraint for them rather than writing $x_i \neq x_j$ for each pair. Inference algorithms developed specifically for this metaconstraint might deduce, for example, that x_j must take a certain value, thus obviating the necessity of branching on x_j. Similarly, a group of inequalities having common structure may allow the inference of specialized cutting planes (implied inequalities).

Developing special-purpose inference procedures has been a major theme of research in both constraint programming and integer programming.

Inference procedures applied to individual constraints or metaconstraints can miss implications of the constraint set as a whole. For instance, the inequalities $x_1 + x_2 \geq 1$ and $x_1 - x_2 \geq 0$ (where each $x_j \in \{0, 1\}$) jointly imply $x_1 = 1$, a fact that follows from neither inequality alone. This implication will be overlooked if the inequalities are processed individually. The constraint programming community partially addresses this problem by collecting at least some of the implications derived from individual constraints into a *constraint store* $\mathcal{D}$. When inferences are drawn from a constraint C, they are actually drawn from C and the current constraint store—that is, from the constraint set $\{C\} \cup \mathcal{D}$. Processing a constraint may enlarge D and thereby strengthen the implications that can be derived from the next constraint. This is a form of *constraint propagation*.

Propagation of this sort is possible only if the constraint store contains very elementary constraints that all of the inference algorithms can accommodate. In constraint programming practice, the constraint store typically consists of domain constraints, namely constraints of the form $x_i \in D_i$ that restrict the domain of x_i. The constraint store is therefore typically a *domain store*. Since any practical inference algorithm considers the variable domains, it already accommodates constraints in the domain store. As a result of this practice, inference algorithms developed by constraint programmers are generally *domain reduction* or *filtering* algorithms that remove values from variable domains. These domain filters are no less useful for integrated problem solving than for constraint programming in particular.

2.1.3 Relaxation

Relaxation is the third element of the solution scheme. It is often used when the subproblems P_k are themselves hard to solve. A relaxation R_k of each P_k is normally created by dropping some constraints in such a way as to make R_k easier than P_k. For instance, one might form a continuous relaxation by allowing integer-valued variables to take any real value.

If P is a minimization problem, the optimal value of the relaxation R_k is a lower bound on the optimal value of P_k. By solving R_k and making use of this bound, one may be able to accelerate the search by avoiding solution of the generally harder problem P_k. For instance, if the optimal value v of R_k is greater than or equal to the value of the best candidate solution found so far, then there is no need to solve P_k,

since its optimal value can be no better than v. One can regard P_k as having been *enumerated*, even though it is not actually solved.

In many cases the solution of a relaxation also guides the search by helping to determine the next restriction P_{k+1} or the inferences that are derived from P_{k+1}.

Relaxation also provides a valuable opportunity to exploit special structure in individual constraints or groups of constraints, and this has been a perennial theme of the optimization literature. For example, strong implied inequalities (cutting planes) can be inferred from sets of inequalities that define certain types of polyhedra. The constraints that relax individual constraints or groups of constraints are pooled into a constraint set to obtain a relaxation of the original problem.

The relaxation, like the constraint store, must contain fairly simple constraints, but for a different reason: they must allow easy optimal solution of the relaxed problem. In traditional optimization methods, these are generally linear inequalities in continuous variables, or perhaps nonlinear inequalities that define a convex feasible set.

Both inference and relaxation provide the solver a somewhat global view over the entire problem even while exploiting local structure. The domain store allows propagation of constraints and therefore the deduction of some the global implications of the problem. The relaxation gathers individual constraint relaxations into a single optimization problem that reflects some of the problem's global characteristics. Although the constraint or domain store can be viewed as a relaxation of the problem, it plays a different role, because its constraints are an input to filtering, while the constraints in a relaxation are not.

2.1.4 Exercises

1 An optimization problem (2.1) can be viewed as defining a function $f : S \to \mathbb{R}$, where S is the feasible set of (2.1). The *epigraph* of f is the set $\{(z, x) \mid z \geq f(x),\ x \in S\}$. Show that (2.2) is a relaxation of (2.1) if and only if the projection of the epigraph of f' onto (z, x) contains the epigraph of f.[1]

2 A relaxation can have a different objective function than the problem relaxed. Can a restriction also be defined to allow this? How would exhaustive search be characterized? Hint: Use the concept of an epigraph.

[1]The projection of the epigraph E of f' onto (z, x) is $\{(z, x) \mid (z, x, y) \in E\}$.

2.2 Branching Search

Branching search uses a recursive divide-and-conquer strategy. If the original problem P is too hard to solve as given, the branching algorithm creates a series of restrictions $P_1, \ldots, P_m$ and tries to solve them. In other words, it *branches* on P. Each P_k is solved, if possible. If it is feasible, its solution becomes a candidate solution. If P_k is too hard to solve, the search procedure attacks P_k in a similar manner by branching on it, and so on, recursively.

A given restriction P_k is considered to be enumerated when it has been solved, or all of its restrictions have been enumerated. The search terminates when the original problem P has been enumerated. The optimal solution is the best candidate solution found.

The most popular branching mechanism is to branch on a variable x_j. The domain of x_j is divided into two or more subsets, and restrictions are created by successively restricting x_j to each of these subsets.

To ensure an exhaustive search, the restrictions $P_1, \ldots, P_m$ created at each branch should be exhaustive. Normally, their feasible sets also partition the feasible set of P (i.e., they are pairwise disjoint), which is unnecessary but more efficient, as it avoids covering the same ground more than once.

To ensure that the search terminates, the branching mechanism must be designed so that problems become easy enough to solve as they are increasingly restricted. For instance, if the variable domains are finite, then branching on variables will eventually reduce the domains to singletons, thus fixing the value of each x_j and making the restriction trivial to solve. Figure 2.1 displays a generic branching algorithm.

2.2.1 Branch and Infer

Inference may be combined with branching by inferring new constraints for each P_k before P_k is solved. When inference takes the form of do-

Let $S = \{P_0\}$ and $v_{\text{UB}} = \infty$.
While S is nonempty repeat:
  Select a problem restriction $P \in S$ and remove P from S.
  If P is too hard to solve then
    Define restrictions $P_1, \ldots, P_m$ of P and add them to S.
  Else
    Let v be the optimal value of P and let $v_{\text{UB}} = \min\{v, v_{\text{UB}}\}$.
The optimal value of P_0 is v_{UB}.

Figure 2.1. Generic branching algorithm for solving a minimization problem P_0. Set S contains the problem restrictions so far generated but not yet attempted, and v_{UB} is the best solution value obtained so far.

main filtering, for example, some of the variable domains are reduced in size. When one branches on variables, this tends to reduce the size of the branching tree because the domains are more rapidly reduced to singletons or the empty set.

2.2.2 Branch and Relax

Relaxation can also be combined with branching in a process that is known in the operations research community as *branch and bound* and in the constraint programming community as *branch and relax*. One solves the relaxation R_k of each restriction, rather than P_k itself. If the solution of R_k happens to be feasible in P_k, it is optimal for P_k and becomes a candidate solution. If R_k is infeasible, then so is P_k. Otherwise, the algorithm branches on P_k. To ensure termination, the branching mechanism must be designed so that solving R_k becomes sufficient to solve P_k if one descends deeply enough into the search tree.

The choice of how to branch depends on which constraint is violated by the solution of the relaxation. For example, if the domain of x_j is $\{1, 2, 3\}$, and the solution value x_j in the relaxation is 1.5, the domain constraint $x_j \in \{1, 2, 3\}$ is violated. This would trigger a branching scheme that the solver associates with the domain constraint, perhaps splitting the domain into $\{1\}$ and $\{2, 3\}$. If two or more constraints are violated, the solver must decide which one will control the branching.

Branch and relax also uses the bounding mechanism described earlier. If the optimal value of R_k is greater than or equal to the value of the best candidate solution found so far, then there is no point in solving P_k and no need to branch on P_k. In this case, the search tree is said to be "pruned" at P_k. Thus, P_k is enumerated when the solution of R_k is feasible in P_k, R_k is infeasible, or the tree is pruned at P_k.

Inference and relaxation can work together effectively in branch-and-relax methods. The addition of inferred constraints (e.g., cutting planes) to P_k can result in a tighter bound when one solves its relaxation R_k. Conversely, the solution of R_k can provide guidance for generating further constraints, as for instance when separating cuts are generated to exclude a solution of R_k that is infeasible in P_k. A generic branching algorithm with inference and relaxation appears in Figure 2.2.

The most widely used solvers in constraint programming, integer programming, and continuous global optimization combine inference and relaxation with branching. Table 2.2 shows how they fit into the search-infer-and-relax framework.

The following sections present six examples of branching methods. A freight transfer problem shows how inference and relaxation techniques from both constraint and integer programming can work together. A

Let $S = \{P_0\}$ and $v_{\text{UB}} = \infty$.
While S is nonempty repeat:
    Select a problem restriction $P \in S$ and remove P from S.
    Repeat as desired:
        Add inferred constraints to P.
        Let v_{R} be the optimal value of a relaxation R of P.
    If $v_{\text{R}} < v_{\text{UB}}$ then
        If R's optimal solution is feasible for P then let $v_{\text{UB}} = \min\{v_{\text{R}}, v_{\text{UB}}\}$.
        Else define restrictions $P_1, \ldots, P_m$ of P and add them to S .
The optimal value of P_0 is v_{UB}.

Figure 2.2. Generic branching algorithm, with inference and relaxation, for solving a minimization problem P_0. The inner repeat loop is typically executed only once, but it may be executed several times, perhaps until no more constraints can be inferred or R becomes infeasible. The inference of constraints can be guided by the solution of previous relaxations.

Table 2.2. How some selected branching methods fit into the search-infer-and-relax framework.

Solution method	*Restriction* P_k	*Relaxation* R_k	*Selection function* $s(R_k)$	*Inference*
Constraint programming	Created by splitting domain, etc.	Domain store	Any feasible solution of R_k	Domain filtering, cutting planes
MILP branch and cut	Created by branching on fractional variables	LP relaxation + cutting planes	Optimal solution of R_k	Cutting planes, preprocessing
Continuous global optimization	Created by splitting intervals	LP or convex NLP relaxation	Optimal solution of R_k	Interval propagation, Lagrangean bounding

Note: NLP refers to nonlinear programming.

production planning problem introduces disjunctive constraints. An employee scheduling problem illustrates several modeling and filtering concepts developed in constraint programming. A small example of continuous global optimization provides a glimpse of how global solvers combine domain filtering and relaxation. A fifth example shows how an integrated method can solve a realistic product configuration problem more efficiently than constraint or integer programming alone. A final

airline crew scheduling problem illustrates branch and price, a recently popular framework for combining constraint and integer programming.

2.2.3 Example: Freight Transfer

A simple freight transfer problem illustrates how inference and relaxation can interact with branching search. Forty-two tons of freight must be conveyed overland. The shipper has a choice of trucks from a fleet of twelve vehicles. The trucks come in four sizes, with three vehicles of each size (Table 2.3). No more than eight trucks may be used altogether. The problem is to select trucks to carry the freight at minimum cost.

Formulating the Problem

If x_i is the number of trucks of type i used, the requirement that 42 tons be transported can be written

$$7x_1 + 5x_2 + 4x_3 + 3x_4 \geq 42$$

This is a knapsack covering constraint, which in general has the form $ax \geq \alpha$, where each $a_i > 0$ and the variables x_i must take nonnegative integer values. Such a constraint can be interpreted as saying that one must select x_i knapsacks of each size a_i in such a way as to carry a total load of α. In the freight problem, the knapsacks are trucks.

One can also formulate a knapsack packing constraint of the form $bx \leq b_0$, where each $b_i > 0$. It arises when one wishes to pack a single knapsack with several types of items, and each type i consumes space b_i. The constraint says that the total space consumed must not exceed the capacity b_0 of the knapsack. In the freight problem, the limit of eight trucks can be conceived as a knapsack packing constraint

$$x_1 + x_2 + x_3 + x_4 \leq 8$$

in which the knapsack is a fleet of at most eight trucks.

Table 2.3. Data for a small instance of a freight transfer problem.

Truck type	Number available	Capacity (tons)	Cost per truck
1	3	7	90
2	3	5	60
3	3	4	50
4	3	3	40

The freight problem can now be formulated

$$\text{Integer linear: } \begin{cases} \min\ 90x_1 + 60x_2 + 50x_3 + 40x_4 \\ 7x_1 + 5x_2 + 4x_3 + 3x_4 \geq 42 & (a) \\ x_1 + x_2 + x_3 + x_4 \leq 8 & (b) \end{cases} \tag{2.3}$$

$$\text{Domains: } x_i \in \{0, 1, 2, 3\},\ \ i = 1, \ldots, 4$$

In accord with the spirit of revealing problem structure, the constraints are grouped by type to provide the solver guidance as to how to process them. Since the objective function becomes an integer linear inequality whenever it is bounded, it is grouped with the integer linear constraints to form an integer linear metaconstraint. An optimal solution is $(x_1, \ldots, x_4) = (3, 2, 2, 1)$, with a minimum cost of 530.

The problem may be solved by a branching algorithm that uses inference and relaxation. The following sections first show how to carry out the inference and relaxation steps and then how to conduct the search.

Inference: Bounds Propagation

Domain filtering is a popular form of inference, in part because the reduced domains inferred from one constraint can be used as a starting point for domain reduction in another constraint. One can, in principle, cycle through the constraints in this fashion until no further filtering is possible. This allows one to draw some inferences that do not follow from any single constraint, even though the constraints are processed individually.

A type of domain filtering known as *bounds propagation* is useful for the freight transport problem. The current domain of each variable x_i is represented as a set $\{L_i, L_i + 1, \ldots, U_i\}$ of consecutive integers, since the domains are reduced solely by tightening lower and upper bounds on the variables.

Bounds propagation for a knapsack covering constraint $ax \geq a_0$ is based on the observation that, for any i, the constraint implies

$$x_i \geq \frac{a_0 - \sum_{j \neq i} a_j x_j}{a_i} \geq \frac{a_0 - \sum_{j \neq i} a_j U_j}{a_i}$$

The second inequality is due to $x_j \leq U_j$. This allows one to update each lower bound L_i as follows:

$$L_i' = \max \left\{ L_i, \left\lceil \frac{a_0 - \sum_{j \neq i} a_j U_j}{a_i} \right\rceil \right\}$$

where $\lceil \delta \rceil$ is δ rounded up. In other words, one *propagates* the upper bounds U_j to obtain new lower bounds. If there is a knapsack packing constraint $bx \leq b_0$ in the problem, one can use it to propagate the revised lower bounds L'_j in analogous fashion and obtain new upper bounds:

$$U'_i = \min \left\{ U_i, \left\lfloor \frac{b_0 - \sum_{j \neq i} b_j L'_j}{b_i} \right\rfloor \right\}$$

where $\lfloor \delta \rfloor$ is δ rounded down.

At this point all of the upper and lower bounds have been updated. They can now be viewed as the original bounds L_i, U_i and propagated by cycling through $ax \geq a_0$ and $bx \leq b_0$ again. If desired, the process is repeated until a fixed point is obtained; that is, until the bounds are unchanged from the previous update.

Bounds propagation is easily applied to constraints (a) and (b) of the problem instance (2.3). Using the initial bounds $(L_i, U_i) = (0, 3)$, inequality (a) raises the lower bound on x_1 from 0 to

$$L'_1 = \max \left\{ 0, \left\lceil \frac{42 - 5U_2 - 4U_3 - 3U_4}{7} \right\rceil \right\} = 1 \tag{2.4}$$

yielding a revised domain $\{1, 2, 3\}$ for x_1. No other domain reductions are possible.

Inference: Valid Inequalities

In addition to deducing smaller domains, one can deduce *valid inequalities* or *cutting planes* from the knapsack constraints. The inferred inequalities are normally added to the original constraint set in order to produce a stronger continuous relaxation.[2]

A simple class of valid inequalities for $ax \geq a_0$ are *knapsack cuts*, which are obtained as follows. Again, let $\{L_i, L_i + 1, \ldots, U_i\}$ be the current domain of x_i. Let a *packing* for $ax \geq a_0$ be a set I of indices for which the corresponding terms $a_i x_i$ cannot by themselves satisfy $ax \geq a_0$, even with each x_i at its upper bound. That is,

$$\sum_{i \in I} a_i U_i < a_0$$

[2]Reducing a domain to $\{L_i, \ldots, U_i\}$ can be viewed as a special case of deducing inequalities, because it, in effect, generates the valid inequalities $L_i \leq x_i \leq U_i$. It is nonetheless useful to treat this special case separately, since reduced domains are normally maintained in a different data structure than valid inequalities.

Thus, the remaining terms $a_i x_i$ for $i \notin I$ must cover the gap:

$$\sum_{i \notin I} a_i x_i \geq a_0 - \sum_{i \in I} a_i U_i$$

This yields the valid knapsack cut

$$\sum_{i \notin I} x_i \geq \left\lceil \frac{a_0 - \sum_{i \in I} a_i U_i}{\max_{i \notin I}\{a_i\}} \right\rceil$$

Since there can be a large number of packings, one may want to consider only cuts that correspond to maximal packings, which are packings of which no proper superset is a packing. One can also neglect any packing that contains all indices but one, say i, since the corresponding cut imposes a lower bound on x_i that is identical to that obtained by domain filtering.

There are analogous knapsack cuts for the knapsack packing inequality $bx \leq b_0$, but they add nothing to the freight problem because $b = (1, \ldots, 1)$.

In the above problem instance, the constraint (a) in (2.4) has three maximal packings that omit more than one index. They correspond to knapsack cuts as follows:

$$\begin{aligned} I = \{1,2\}: &\quad x_3 + x_4 \geq 2 \\ I = \{1,3\}: &\quad x_2 + x_4 \geq 2 \\ I = \{1,4\}: &\quad x_2 + x_3 \geq 3 \end{aligned} \tag{2.5}$$

Relaxation: Linear Programming

A continuous relaxation of the problem instance (2.4) can be obtained by allowing each variable x_i to take any value in the continuous interval $[L_i, U_i]$, where $\{L_i, L_i + 1, \ldots, U_i\}$ is its current domain. One can also add the knapsack cuts (2.5) before relaxing the problem. This yields the relaxation below:

$$\text{Linear:} \begin{cases} \min\ 90x_1 + 60x_2 + 50x_3 + 40x_4 & \\ 7x_1 + 5x_2 + 4x_3 + 3x_4 \geq 42 & (a) \\ x_1 + x_2 + x_3 + x_4 \leq 8 & (b) \\ x_3 + x_4 \geq 2 & (c) \\ x_2 + x_4 \geq 2 & (d) \\ x_2 + x_3 \geq 3 & (e) \\ L_i \leq x_i \leq U_i, \quad i = 1, \ldots, 4 & \end{cases} \tag{2.6}$$

$$\text{Domains: } x_i \in \Re,\ i = 1, \ldots, 4$$

which can be solved by linear programming. The knapsack cuts (c)–(e) are those obtained when (L_i, U_i) are the original bounds $(0, 3)$. In general, stronger knapsack cuts can be obtained when the bounds are tightened during the branching search, but this is not done here.

Branching Search

A search tree for problem instance (2.3) appears in Figure 2.3. Each node of the tree below the root corresponds to a restriction of the orig-

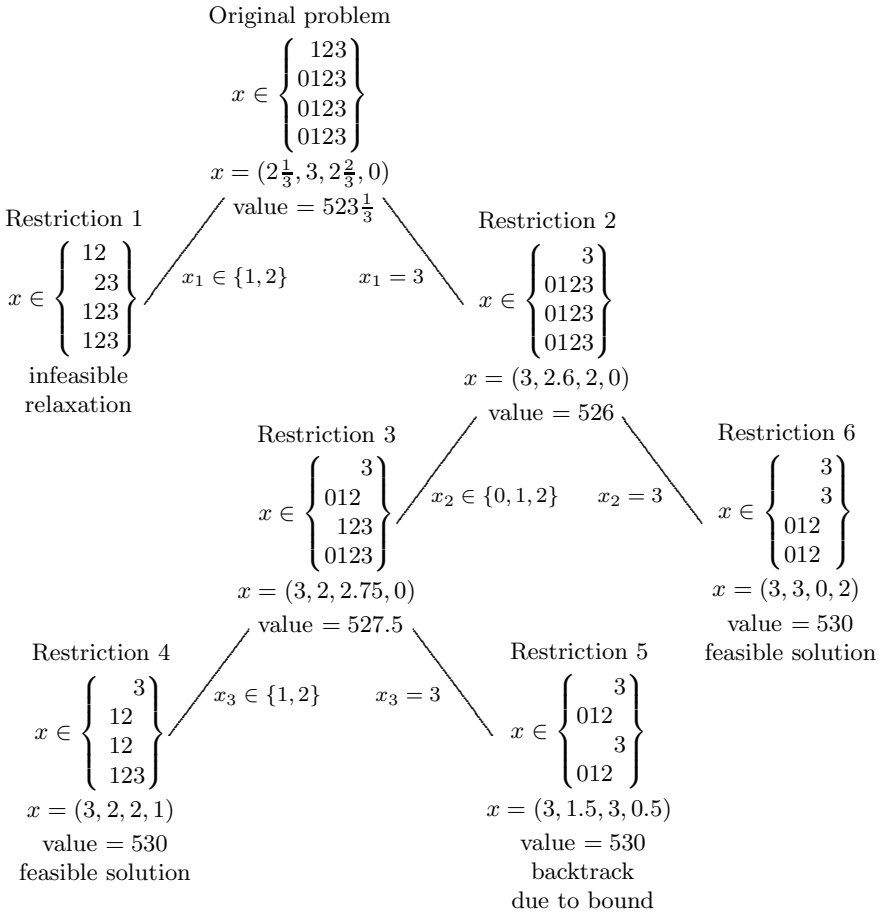

Figure 2.3. Branch-and-relax tree for the freight transfer problem. Each node of the tree shows, in braces, the filtered domains after domain reduction. The rows inside the braces correspond to $x_1, \ldots, x_4$. The solution of the continuous relaxation appears immediately below the domains.

inal problem. A restriction is processed by applying inference methods (bounds propagation and cut generation), then solving a continuous relaxation, and finally branching if necessary.

Bounds propagation applied to the original problem at the root node reduces the domain of x_1 to $\{1,2,3\}$, as described earlier. Figure 2.3 shows the resulting domains in braces. Next, three knapsack cuts (2.5) are generated, using the lower and upper bounds L_i, U_i that define the current domains. They are added to the constraint set in order to obtain the continuous relaxation (2.6). The optimal solution $x = (2\frac{1}{3}, 3, 2\frac{2}{3}, 0)$ of the relaxation has value $523\frac{1}{3}$.

This solution is infeasible in the original problem because x_1 and x_3 do not belong to their respective domains (they are nonintegral). It is therefore necessary to branch on one of the domain constraints $x_1 \in \{1,2,3\}$, $x_3 \in \{0,1,2,3\}$. Branching on the first splits the domain into $\{1,2\}$ and $\{3\}$, as the solution value $2\frac{1}{3}$ of x_1 lies between 2 and 3. This generates restrictions 1 and 2, which become the contents of set S in the branching algorithm of Figure 2.2.

The tree will be traversed in a depth-first manner. Moving first to restriction 1, where x_1's domain is $\{1,2\}$, bounds propagation yields the smaller domains shown in Figure 2.3. These new bounds yield no further knapsack cuts. Due to the tighter bounds L_i, U_i, the continuous relaxation (2.6) of the current restriction is infeasible, and no branching is necessary. This leaves restriction 2 in S.

Moving now to restriction 2, no further domain filtering is possible, and solution of the relaxation (2.6) yields $x = (3, 2.6, 2, 0)$. Branching on x_2 creates restrictions 3 and 6.

Continuing in a depth-first manner, restriction 3 is processed next. Inference yields the domains shown, and branching on x_3 produces restrictions 4 and 5. The continuous relaxation of restriction 4 has the integral solution $x = (3,2,2,1)$, which is feasible in the restriction. It becomes the *incumbent* solution (the best feasible solution so far), and no branching is necessary.

Restriction 5 is processed next. Here, the relaxation has a nonintegral solution, but its optimal value 530 is no better than the value of the incumbent solution. Since 530 is a lower bound on the optimal value of any further restriction of restriction 5, there is no need to branch. The tree is therefore "pruned" at restriction 5, and the search proceeds to restriction 6.

The continuous relaxation of this restriction has an integral solution, and there is no need to branch, thus completing the search. Since the solution is no better than the incumbent (in fact it is equally good), the incumbent solution $x = (3,2,2,1)$ is optimal.

In some cases, one can prove the infeasibility of a restriction by inference alone, perhaps by reducing a variable domain to the empty set. When this occurs, there is no need to solve a relaxation of the restriction.

2.2.4 Example: Production Planning

A very simple production planning problem illustrates how logical and continuous variables can interact. A manufacturing plant has three operating modes, each of which imposes different constraints. The objective is to decide in which mode to run the plant, and how much of each of two products to make, so as to maximize net income.

Formulating the Problem

Let x_A, x_B be the production levels of products A and B, respectively. In mode 0 the plant is shut down, and $x_A = x_B = 0$. In mode 1, it incurs a fixed cost of 35, and the production levels must satisfy the constraint $2x_A + x_B \leq 10$. Mode 2 incurs a fixed cost of 45 and the constraint $x_A + 2x_B \leq 10$ is imposed. The company earns a net income of 5 for each unit of product A manufactured, and 3 for each unit of product B.

A natural modeling approach is to let a boolean variable δ_k be true when the plant runs in mode k. Then, the model is immediate:

$$\begin{aligned}
&\text{Linear: } \max\ 5x_A + 3x_B - f \\
&\text{Logic: } \delta_0 \vee \delta_1 \vee \delta_2 \\
&\text{Conditional: } \begin{cases} \delta_0 \Rightarrow (x_A = x_B = f = 0) \\ \delta_1 \Rightarrow (2x_A + x_B \leq 10,\ f = 35) \\ \delta_2 \Rightarrow (x_A + 2x_B \leq 10,\ f = 45) \end{cases} \\
&\text{Domains: } x_A, x_B \geq 0,\ \ \delta_k \in \{\mathit{true}, \mathit{false}\},\ k = 0, 1, 2
\end{aligned} \tag{2.7}$$

where variable f represents the fixed cost. The logic constraint means that at least one of the three boolean variables must be true.

Relaxation

The model has an interesting relaxation because each production mode k enforces constraints that define a polyhedron in the space of the continuous variables x_A, x_B, f. So the projection of the feasible set onto this space is the union of the three polyhedra, illustrated in Figure 2.4. The best possible continuous relaxation of this set is its *convex hull*, which is itself a polyhedron and is also shown in the figure. The convex hull of a set is union of all line segments connecting any two points of the set.

There is a general procedure, given in Section 4.7.1, for writing a linear constraint set that describes the convex hull of a disjunction of

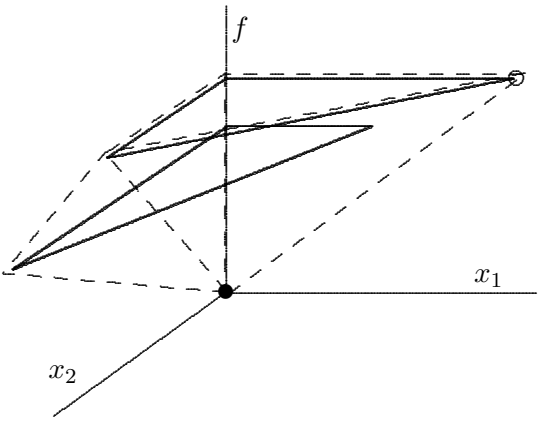

Figure 2.4. Feasible set (two triangular areas and black dot) of a production planning problem, projected onto the space of the continuous variables. The convex hull is the volume inside the dashed polyhedron. The open circle marks the optimal solution.

linear systems. In this case, the convex hull description is

$$\begin{aligned}
&2x_{A1} + x_{B1} \leq 10y_1 \\
&x_{A2} + 2x_{B2} \leq 10y_2 \\
&f \geq 35y_1 + 45y_2 \\
&x_A = x_{A1} + x_{A2}, \quad x_B = x_{B1} + x_{B2} \\
&y_0 + y_1 + y_2 = 1, \quad y_k \geq 0, \ k = 0, 1, 2 \\
&x_{Ak} \geq 0, \ x_{Bk} \geq 0, \quad k = 1, 2
\end{aligned} \tag{2.8}$$

where the variables y_k correspond to the boolean variables δ_k, and fixing $\delta_k = \mathit{true}$ is equivalent to setting $y_k = 1$. Since there are new variables x_{Ak}, x_{Bk}, y_k in the relaxation, one should, strictly speaking, say that it describes a set whose projection onto x_A, x_B, f is the convex hull of (2.7)'s feasible set. A lower bound on the optimal value can now be obtained by minimizing $5x_A + 3x_b - f$ subject to (2.8).

To take advantage of this relaxation, the solver must somehow recognize that the feasible set in continuous space is a union of polyhedra. This can be done by collecting the constraints of (2.7) into a single *linear disjunction*, resulting in the model:

Linear: max $5x_A + 3x_B - f$

Linear disjunction:

$$\begin{bmatrix} \delta_0 \\ x_A = x_B = 0 \\ f = 0 \end{bmatrix} \vee \begin{bmatrix} \delta_1 \\ 2x_A + x_B \leq 10 \\ f = 35 \end{bmatrix} \vee \begin{bmatrix} \delta_2 \\ x_A + 2x_B \leq 10 \\ f = 45 \end{bmatrix}$$

Domains: $x_A, x_B \geq 0, \quad \delta_k \in \{\mathit{true}, \mathit{false}\}, \ k = 0, 1, 2$

The linear disjunction has precisely the same meaning as the logic and conditional constraints of (2.7). Its presence tells the solver to formulate a relaxation for the union of polyhedra described by its three disjuncts. In general the user should be alert for metaconstraints that can exploit problem structure, although in this case an intelligent modeling system could automatically rewrite (2.7) as (2.8).

Branching Search

The problem can now be solved by branch and relax, where branching takes place on the boolean variables. In this case, the solution of the relaxation is $(y_1, y_2) = (0, 1)$, with $(x_A, x_B, f) = (10, 0, 35)$. Because this solution is feasible in (2.7) with $(\delta_0, \delta_1, \delta_2) = (\mathit{false}, \mathit{false}, \mathit{true})$, there is no need to branch. One should operate the plant in mode 2 and make product A only.

In this problem, there is no branching because (2.8) is a convex hull relaxation not only for the corresponding linear disjunction but for the entire problem. In more complex problems, an auxiliary variable y_k may take a fractional value, in which case one can branch by setting $\delta_k = \mathit{true}$ and $\delta_k = \mathit{false}$.

Inference

Inference plays a role in such problems when the logical conditions become more complicated. Suppose, for example, that Plant 1 can operate in 2 modes (indicated by boolean variables δ_{1k} for $k = 0, 1$) and Plant 2 in 3 modes (indicated by variables δ_{2k}). There is also a rule that if plant 1 operates in mode 1, then plant 2 cannot operate in mode 2. The model might be as follows (where $\neg$ means *not*).

$$\text{Linear: } \max\ cx$$

$$\text{Logic: } \begin{cases} \delta_{10} \vee \delta_{11} \\ \delta_{20} \vee \delta_{21} \vee \delta_{22} \\ \delta_{11} \rightarrow \neg\delta_{22} \end{cases}$$

$$\text{Conditional: } \delta_{ik} \Rightarrow A^{ik}x \geq b^{ik}, \quad \text{all } i, k$$

$$\text{Domains: } x \geq 0, \quad \delta_{ik} \in \{\mathit{true}, \mathit{false}\}$$

To extract as many linear disjunctions as possible, one can use the *resolution* algorithm (discussed in Section 3.5.2) to compute the *prime implications* of the logical formulas in the above model:

$$\begin{array}{ll} \delta_{10} \vee \delta_{11} & \delta_{10} \vee \neg\delta_{22} \\ \delta_{20} \vee \delta_{21} \vee \delta_{22} & \neg\delta_{11} \vee \delta_{20} \vee \delta_{21} \\ \neg\delta_{11} \vee \neg\delta_{22} & \delta_{10} \vee \delta_{20} \vee \delta_{21} \end{array}$$

These are the undominated disjunctions implied by the logical formulas. Three of the prime implications contain no negative terms, and they provide the basis for linear disjunctions. The model becomes:

$$\text{Linear: } \max cx$$

$$\text{Logic: } \begin{cases} \neg\delta_{11} \vee \neg\delta_{22} \\ \delta_{10} \vee \neg\delta_{22} \\ \neg\delta_{11} \vee \delta_{20} \vee \delta_{21} \end{cases}$$

Linear disjunction:

$$\bigvee_{k\in\{0,1\}} \begin{bmatrix} \delta_{1k} \\ A^{1k}x \geq b^{1k} \end{bmatrix}$$

$$\bigvee_{k\in\{0,1,2\}} \begin{bmatrix} \delta_{2k} \\ A^{2k}x \geq b^{2k} \end{bmatrix}$$

$$\begin{bmatrix} \delta_{10} \\ A^{10}x \geq b^{10} \end{bmatrix} \vee \begin{bmatrix} \delta_{20} \\ A^{20}x \geq b^{20} \end{bmatrix} \vee \begin{bmatrix} \delta_{21} \\ A^{21}x \geq b^{21} \end{bmatrix}$$

$$\text{Domains: } x \geq 0, \quad \delta_{ik} \in \{true, false\}$$

Again, an intelligent modeling system can carry out this process automatically.

Inference plays a further role as branching proceeds. When branching fixes a boolean variable to true or false, it may be possible to deduce that some other variables must be true or false, thus reducing their domains to a singleton. For example, if δ_{10} is fixed to false, then δ_{11} must be true and δ_{22} false. The resolution method draws all such inferences.

2.2.5 Example: Employee Scheduling

An employee scheduling problem is useful for introducing several of the modeling and inference techniques that have been developed in the constraint programming field. Since the objective is to find a feasible rather than an optimal solution, bounds do not play a role, and relaxation is relatively unimportant. Nonetheless, the integration of technologies is a key ingredient of the solution method, since the filtering algorithms rely on such optimization techniques as maximum flow algorithms and dynamic programming.

A certain hospital ward requires that a head nurse be on duty seven days a week, twenty-four hours a day. There are three eight-hour shifts, and on a given day each shift must be staffed by a different nurse. The schedule must be the same every week. Four nurses (denoted A, B, C and D) are available, all of whom must work at least five days a week.

Since there are 21 eight-hour periods a week, this implies that three nurses will work five days and one will work six. For continuity, no shift should be staffed by more than two different nurses during the week.

Two additional rules reduce the burden of adjusting to shift changes. No employee is asked to work different shifts on two consecutive days; there must be at least one day off in between. Also, an employee who works Shift 2 or 3 must do so at least two days in a row. Shift 1 is the daytime shift and requires less adjustment.

Formulating the Problem

There are two natural ways to think about an employee schedule. One, shown in Table 2.4, is to view it as assigning a nurse to each shift on each day. Another is to see it as assigning a shift (or day off) to each nurse on each day; this is illustrated by Table 2.5, in which Shift 0 represents a day off. Either viewpoint gives rise to a different problem formulation, neither of which is convenient for expressing all the constraints of the problem. Fortunately, there is no need to choose between them. One can use both formulations in the same model and connect them with *channeling constraints*. This not only accommodates all the constraints but makes propagation more effective.

To proceed with the first formulation, let variable w_{sd} be the nurse that is assigned to shift s on day d. Three of the scheduling requirements are readily expressed in this formulation, using metaconstraints

Table 2.4. Employee scheduling viewed as assigning workers to shifts.

	Sun	*Mon*	*Tue*	*Wed*	*Thu*	*Fri*	*Sat*
Shift 1	*A*	*B*	*A*	*A*	*A*	*A*	*A*
Shift 2	*C*	*C*	*C*	*B*	*B*	*B*	*B*
Shift 3	*D*	*D*	*D*	*D*	*C*	*C*	*D*

Table 2.5. Employee scheduling viewed as assigning shifts to workers.

	Sun	*Mon*	*Tue*	*Wed*	*Thu*	*Fri*	*Sat*
Worker A	1	0	1	1	1	1	1
Worker B	0	1	0	2	2	2	2
Worker C	2	2	2	0	3	3	0
Worker D	3	3	3	3	0	0	3

that are well known to constraint programmers. One is the *all-different* constraint, alldiff(x), where x denotes a list $x_1, \ldots, x_n$ of variables. It simply requires that $x_1, \ldots, x_n$ take different values. It can express the requirement that three different nurses be scheduled each day:

$$\text{alldiff}(w_{\cdot d}), \text{ all } d$$

The notation $w_{\cdot d}$ refers to (w_{1d}, w_{2d}, w_{3d}).

The *cardinality* constraint can be used to require that every nurse be assigned at least five days of work. The constraint in general is written

$$\text{cardinality}(x \mid v, \ell, u)$$

where $x = (x_1, \ldots, x_n)$ and v is an m-tuple $(v_1, \ldots, v_m)$ of values. $\ell = (\ell_1, \ldots, \ell_m)$ and $u = (u_1, \ldots, u_m)$ contain lower and upper bounds respectively. The vertical bar in the argument list indicates that everything before the bar is a variable and everything after is a parameter. The constraint requires, for each $i = 1, \ldots, m$, that at least ℓ_i and at most u_i of the variables in x take the value v_i. To require that each nurse work at least five and at most six days, one can write

$$\text{cardinality}(w_{\cdot\cdot} \mid (A, B, C, D), (5, 5, 5, 5), (6, 6, 6, 6))$$

where $w_{\cdot\cdot}$ refers to a tuple of all the variables w_{sd}.

The *nvalues* constraint is written

$$\text{nvalues}(x \mid \ell, u)$$

and requires that the variables $x = (x_1, \ldots, x_n)$ take at least ℓ and at most u different values. To require that at most two nurses work any given shift, one can write

$$\text{nvalues}(w_{s\cdot} \mid 1, 2), \text{ all } s$$

Both nvalues and cardinality generalize the alldiff constraint, but one should use alldiff when possible, since it invokes a more efficient filtering algorithm.

The remaining constraints are not easily expressed in the notation developed so far, because they relate to the pattern of shifts worked by a given nurse. For this reason, it is useful to move to the formulation suggested by Table 2.5. Let y_{id} be the shift assigned to nurse i on day d, where shift 0 denotes a day off. It is first necessary to ensure that all three shifts be assigned for each day. The alldiff constraint serves the purpose:

$$\text{alldiff}(y_{\cdot d}), \text{ all } d$$

This condition is implicit in the constraints already written, but it is generally good practice to write redundant constraints when one is aware of them, in order to strengthen propagation.

The *stretch* constraint was expressly developed to impose conditions on stretches or contiguous sequences of shift assignments in employee scheduling problems. Given a tuple $x = (x_1, \ldots, x_n)$ of variables, a stretch is a maximal sequence of consecutive variables that take the same value. Thus, if $x = (x_1, x_2, x_3) = (a, a, b)$, x contains a stretch of value a having length 2 and a stretch of b having length 1, but it does not contain a stretch of a having length 1. The stretch constraint is written

$$\text{stretch}(x \mid v, \ell, u, P)$$

where x, v, ℓ, and u are defined as in the cardinality constraint. P is a set of *patterns*, each of which is a pair (v, v') of distinct values. The stretch constraint requires, for each v_i in v, that every stretch of value v_i have length at least ℓ_i and at most u_i. It also requires that whenever a stretch of value v comes immediately before a stretch of value v', the pair (v, v') must occur in the pattern set P. The requirements concerning consecutive nursing shifts can now be written

$$\text{stretch-cycle}(y_{i\cdot} \mid (2,3),(2,2),(6,6),P), \text{ all } i$$

where P contains all patterns that include a day off:

$$P = \{(s,0),(0,s) \mid s = 1,2,3\}$$

A cyclic version of the constraint is necessary because every week must have the same schedule. The cyclic version treats the week as a cycle and allows a single stretch to extend across the weekend.

Finally, the two formulations must be forced to have the same solution. This is accomplished with *channeling constraints*, which in this case take the form

$$w_{y_{id}d} = i, \text{ all } i, d \tag{2.9}$$

and

$$y_{w_{sd}s} = s, \text{ all } s, d \tag{2.10}$$

Constraint (2.9) says that on any given day d, the nurse assigned to the shift to which nurse i is assigned must be nurse i, and similarly for

(2.10). The subscripts y_{id} in (2.9) and w_{sd} in (2.10) are *variable indices* or *variable subscripts.*

The model is now complete. It can be written with five metaconstraints, plus domains:

$$\begin{aligned}
&\text{Alldiff:} \left\{ \begin{array}{l} (w_{\cdot d}) \\ (y_{\cdot d}) \end{array} \right\}, \text{ all } d \\
&\text{Cardinality: } (w_{\cdot\cdot} \,|\, (A,B,C,D),(5,5,5,5),(6,6,6,6)) \\
&\text{Nvalues: } (w_{s\cdot} \,|\, 1,2), \text{ all } s \\
&\text{Stretch-cycle: } (y_{i\cdot} \,|\, (2,3),(2,2),(6,6),P), \text{ all } i \\
&\text{Linear:} \left\{ \begin{array}{l} w_{y_{id}d} = i, \text{ all } i \\ y_{w_{sd}d} = s, \text{ all } s \end{array} \right\}, \text{ all } d \\
&\text{Domains:} \left\{ \begin{array}{l} w_{sd} \in \{A,B,C,D\}, \; s = 1,2,3 \\ y_{id} \in \{0,1,2,3\}, \; i = A,B,C,D \end{array} \right\}, \text{ all } d
\end{aligned} \tag{2.11}$$

The linear constraints are so classified because they are linear aside from the variable indices, which will be eliminated as described below. One can appreciate the convenience of metaconstraints by attempting to formulate this problem with, say, a mixed integer linear programming model.

Inference: Domain Filtering

The alldiff constraint poses an interesting filtering problem that, fortunately, can be easily solved. Suppose, for example, that the current domains of assignment variables w_{s1} are: The alldiff constraint poses an interesting filtering problem that, fortunately, can be easily solved. Suppose, for example, that the current domains of assignment variables w_{s1} are:

$$w_{11} \in \{A,B\}, \; w_{21} \in \{A,B\}, w_{31} \in \{A,B,C,D\}$$

The days are numbered $1,\ldots,7$ so that the subscript 1 in w_{s1} refers to Sunday. Thus, only nurse A or B can be assigned to Shift 1 or 2 on Sunday, while any nurse can be assigned to shift 3. Since

$$\text{alldiff}(w_{11}, w_{21}, w_{31})$$

must be enforced, one can immediately deduce that neither A nor B can be assigned to Shift 3. Thus, the domain of w_{31} can be reduced to $\{C,D\}$. This type of reasoning can be generalized by viewing the solution of alldiff as a maximum flow problem on a network, for which there

are very fast algorithms. Optimality conditions for maximum flows allow one to identify values that can be removed from domains. These ideas are presented in Section 3.9. Straightforward generalizations of this flow model can provide filtering for the cardinality and nvalues constraints.

Filtering for the stretch constraint is more complicated, but nonetheless tractable. Suppose, for example, that the domains of y_{Ad} contain the values listed beneath each variable:

y_{A1}	y_{A2}	y_{A3}	y_{A4}	y_{A5}	y_{A6}	y_{A7}
0	0		0		0	0
1			1	1	1	1
2	2	2		2		2
3	3	3	3		3	3

(2.12)

This means, for instance, that nurse A must work either Shift 2 or Shift 3 on Tuesday. After some thought, one can deduce from the stretch constraint in (2.11) that several values can be removed, resulting in the following domains:

y_{A1}	y_{A2}	y_{A3}	y_{A4}	y_{A5}	y_{A6}	y_{A7}
0			0		0	0
				1	1	1
2	2	2				2
3	3	3				3

(2.13)

Note that two variables are fixed. A polynomial-time dynamic programming algorithm can remove all infeasible values for any given stretch constraint. It is described in Section 3.12.

Inference for Variable Indices

The variably indexed expression $w_{y_{id}d}$ in (2.11) can be processed with the help of an *element* constraint. Domain reduction algorithms for this constraint are well known in the constraint programming community, and elementary polyhedral theory provides a continuous relaxation for the constraint.

The type of element constraint required here has the form

$$\text{element}(y, x, z) \tag{2.14}$$

where y is an integer-valued variable, x is a tuple $(x_1, \ldots, x_m)$ of variables, and z is a variable. The constraint requires that z be equal to the y^{th} variable in the list $x_1, \ldots, x_m$. One can therefore deal with a variably indexed expression like x_y by replacing it with z and adding

the element constraint (2.14). In particular, the constraint $w_{y_{id}d} = i$ is parsed by replacing it with $z_{id} = i$ and generating the constraint

$$\text{element}\,(y_{id}, (w_{0d}, \ldots, w_{3d}), z_{id})$$

Similarly, $y_{w_{sd}d} = s$ is replaced with $\bar{z}_{sd} = s$ and

$$\text{element}\,(w_{sd}, (y_{Ad}, \ldots, y_{Dd}), \bar{z}_{sd})$$

Filtering for element is a simple matter. An example will illustrate this and show how propagation of channeling constraints between two models can reduce domains. Focusing on Sunday (Day 1), suppose the domains of w_{s1} and y_{i1} contain the elements listed beneath each variable:

w_{01}	w_{11}	w_{21}	w_{31}
A		A	A
B	B		B
C	C	C	
	D	D	D

y_{A1}	y_{B1}	y_{C1}	y_{D1}
0	0		0
1	1	1	
	2	2	2
3		3	3

Suppose that no further propagation is possible among the constraints involving only the variables w_{sd}, and similarly for the constraints involving only the variables y_{id}. Nonetheless, the channeling constraints can yield further domain reduction. For instance, since y_{A1} has domain $\{0, 1\}$, the constraint

$$\text{element}\,(y_{A1}, (w_{01}, \ldots, w_{31}), z_{A1})$$

implies that y_{A1} must select either w_{01} or w_{11} to be equated with z_{A1}. But $z_{A1} = A$, and only w_{01}'s domain contains A. Thus y_{A1} must select w_{01}, and y_{A1}'s domain can be reduced to $\{0\}$. Similarly, the constraints $\bar{z}_{01} = 0$ and

$$\text{element}\,(w_{01}, (y_{A1}, \ldots, y_{D1}), \bar{z}_{01})$$

remove C from the domain $\{A, B, C\}$ of w_{01}. Deductions of this kind reduce the domains to:

w_{01}	w_{11}	w_{21}	w_{31}
A			A
B	B		
	C	C	
		D	D

y_{A1}	y_{B1}	y_{C1}	y_{D1}
0	0		
	1	1	
		2	2
3			3

Search, Relaxation, and Symmetry

The nurse scheduling problem is solved through a combination of branching and propagation. Branching proceeds by splitting domains with more than one element. The search process is too long to reproduce here, but a feasible solution is exhibited in Tables 2.4 and 2.5.

The choice of which domain to split, and how to split it, is more an art than a science, but in feasibility problems a common rule is to use *first fail* branching: to branch on a variable that is more likely to lead quickly to infeasibility in the branch that is explored first. The rationale for this strategy is that it helps to avoid exploring large subtrees that contain no solution. The likelihood of causing infeasibility is gauged in various ways, perhaps by the size of the domain. Branching on a smaller domain may result in more propagation and therefore detection of infeasibility before fixing too many variables.

As remarked earlier, the domain store normally plays a different role than a relaxation because its constraints are an input to filtering. Yet, the domain store is nonetheless a relaxation and can play that role as well. One can "solve" the domain store, and if its solution happens to be feasible, the search for a feasible solution is over. The domain store can be solved simply by selecting a value from each domain, but in practice, one can use a heuristic method to construct and examine several promising solutions. This is sometimes called *probing* because it, in effect, probes deeper into the tree for feasible solutions. Such an incomplete search within a larger complete search sometimes proves very effective.

Symmetry is a important feature of the employee scheduling problem. The four employees, for example, are indistinguishable, and their roles can be interchanged in any solution to obtain another feasible solution. Similarly, Shifts 2 and 3 are subject to the same rules and can be interchanged in any solution. The days of the week can be rotated with no effect on feasibility. The presence of symmetry can prolong the search, because one may waste time enumerating several subtrees that are identical up to an interchange of variables or values. There are several means of *breaking* symmetry, the simplest of which is to fix some variables without loss of generality. For instance, employees A, B, and C can be assigned at the outset to work Shifts 1, 2 and 3, respectively, on Sunday.

Symmetry is not discussed further in this book, but it is an active research area that may pay dividends for accelerating search. One should bear in mind, however, that symmetry tends to occur more often in textbook problems than real-world problems, and more often in feasibility problems than optimization problems. In a real employee scheduling problem, for example, weekends are likely to have different rules than

weekdays, and employees are likely to have different privileges. If the objective is to minimize cost or maximize the satisfaction of preferences, employees will probably have different shift-dependent wage rates as well as different preferences.

2.2.6 Example: Continuous Global Optimization

Optimization problems need not contain discrete variables to be combinatorial in nature. A continuous optimization problem may have a large number of locally optimal solutions, which are solutions that are optimal in a neighborhood about them. Nonlinear programming solvers, highly developed as they are, are geared to finding only a local optimum. To identify a global optimum, there is often no alternative but to enumerate the local optima, explicitly or implicitly, in search of the best one.

The most popular and effective global solvers use a branch-and-relax approach that is closely analogous to the one presented for the freight transfer problem. The main difference is that the variable domains are continuous intervals rather than finite sets, the algorithm branches on a variable by splitting an interval into two or more intervals. This sort of branching divides continuous space into increasingly smaller boxes until a global solution can be isolated in a very small box. Constraint propagation and relaxation are also somewhat different than in discrete problems, because they employ techniques that are specialized to the nonlinear functions that typically occur in continuous problems.

Global optimization is best illustrated with a very simple example that is chosen to highlight some of the simpler techniques rather than to represent a practical application. The problem is:

$$\begin{aligned} &\text{Linear: } \begin{cases} \max x_1 + x_2 \\ 2x_1 + x_2 \leq 2 \end{cases} \\ &\text{Bilinear: } 4x_1x_2 = 1 \\ &\text{Domains: } x_1 \in [0,1],\ x_2 \in [0,2] \end{aligned} \tag{2.15}$$

The feasible set is illustrated in Figure 2.5. There are two locally optimal solutions,

$$\begin{aligned} (x_1, x_2) &= (\tfrac{1}{2} + \tfrac{1}{4}\sqrt{2}, 1 - \tfrac{1}{2}\sqrt{2}) \approx (0.853553, 0.292893) \\ (x_1, x_2) &= (\tfrac{1}{2} - \tfrac{1}{4}\sqrt{2}, 1 + \tfrac{1}{2}\sqrt{2}) \approx (0.146447, 1.707107) \end{aligned}$$

the latter of which is globally optimal.

Inference: Bounds Propagation

Bounds propagation can be applied to nonlinear constraints as well as linear inequalities. This is easiest when a constraint can be "solved" for

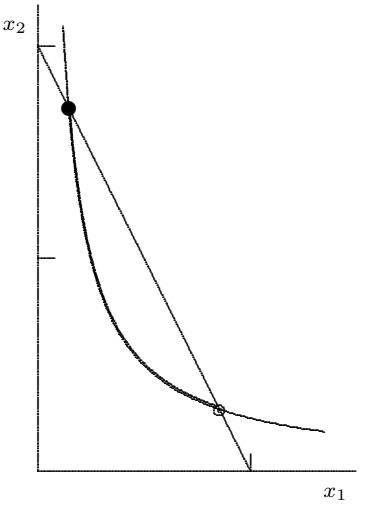

Figure 2.5. Illustration of a continuous global optimization problem. The heavy curve represents the feasible set. The dot marks the optimal solution, and the circle marks a second locally optimal solution.

a bound on each variable in terms of a monotone function of the other variables, as is possible for linear inequalities. To obtain a new bound for a given variable, one need only substitute the smallest or largest possible values for the other variables.

In the example, $4x_1x_2 = 1$ can be solved for $x_1 = 1/4x_2$. By substituting the upper bound of 2 for x_2, one can update the lower bound on x_1 to $\frac{1}{8}$, resulting in the domain $[0.125, 1]$. Similarly, the domain for x_2 can be reduced to $[0.25, 2]$.

More generally, suppose constraint C can be solved for a lower bound $g_1(x_2, \ldots, x_n)$ on x_1, where g_1 is monotone nonincreasing in each argument (as is commonly the case). That is, any solution $x = (x_1, \ldots, x_n)$ that is feasible for C satisfies $x_1 \geq g_1(x_2, \ldots, x_n)$. If each x_j has domain $[L_j, U_j]$, the lower bound L_1 can be updated to $g_1(U_2, \ldots, U_n)$, and analogously for an upper bound.

Each constraint of the problem can be used to reduce the domains obtained from the previous constraint. In the example, the linear constraint $2x_1+x_2 \leq 2$ allows further reduction of the domains $[0.125, 1]$ and $[0.25, 2]$ to $[0.125, 0.875]$ and $[0.25, 1.75]$, respectively. These domains can be still further reduced by cycling back through the two constraints repeatedly, asymptotically reaching a fixed point that is approximately $[0.146, 0.854]$ for x_1 and $[0.293, 1.707]$ for x_2. Global solvers typically truncate this process quite early, however, because the marginal gains of

further iterations are not worth the time investment. The computations to follow reflect the results of cycling through the constraints only once.

Relaxation: Factored Functions

Linear relaxations can often be created for nonlinear constraints by *factoring* the functions involved into more elementary functions for which linear relaxations are known. For instance, the constraint $x_1x_2/x_3 \leq 1$ can be written $y_1 \leq 1$ by setting $y_1 = y_2/x_3$ and $y_2 = x_1x_2$. This factors the function x_1x_2/x_3 into the elementary operations of multiplication and division, for which tight linear relaxations have been derived.

The constraint $4x_1x_2 = 1$ in the example (2.15) can be written $4y = 1$ by setting $y = x_1x_2$. The product $y = x_1x_2$ has the well-known relaxation

$$\begin{aligned} L_2x_1 + L_1x_2 - L_1L_2 \leq y \leq L_2x_1 + U_1x_2 - U_1L_2 \\ U_2x_1 + U_1x_2 - U_1U_2 \leq y \leq U_2x_1 + L_1x_2 - L_1U_2 \end{aligned} \tag{2.16}$$

where again $[L_i, U_i]$ is the current interval domain of x_i.

The example (2.15) therefore has the relaxation

$$\text{Linear:} \begin{cases} \max\ x_1 + x_2 \\ 4y = 1 \\ 2x_1 + x_2 \leq 2 \\ L_2x_1 + L_1x_2 - L_1L_2 \leq y \leq L_2x_1 + U_1x_2 - U_1L_2 \\ U_2x_1 + U_1x_2 - U_1U_2 \leq y \leq U_2x_1 + L_1x_2 - L_1U_2 \\ L_i \leq x_i \leq U_i, \quad i = 1, 2 \end{cases} \tag{2.17}$$

The constraint $4y = 1$ can be eliminated by substituting $1/4$ for y in the other constraints. Initially, $[L_1, U_1] = [0.125, 0.875]$ and $[L_2, U_2] = [0.25, 1.75]$, since these domains result from the bounds propagation described in the previous section.

Branching Search

At the root node of a branch-and-relax tree (Figure 2.6), the initial linear relaxation (2.17) is solved to obtain the solution $(x_1, x_2) = (\frac{1}{7}, \frac{41}{24}) \approx (0.143, 1.708)$. This solution is infeasible in the original problem (2.15), since $4x_1x_2 = 1$ is not satisfied. It is therefore necessary to branch by splitting the domain of a variable.

The choice of branching heuristic can be crucial to fast solution, but for purposes of illustration one can simply split the domain $[0.25, 1.75]$ of x_2 into two equal parts. Restriction 1 in Figure 2.6 corresponds to the lower branch, $x_2 \in [0.25, 1]$. The domains reduce as shown. The

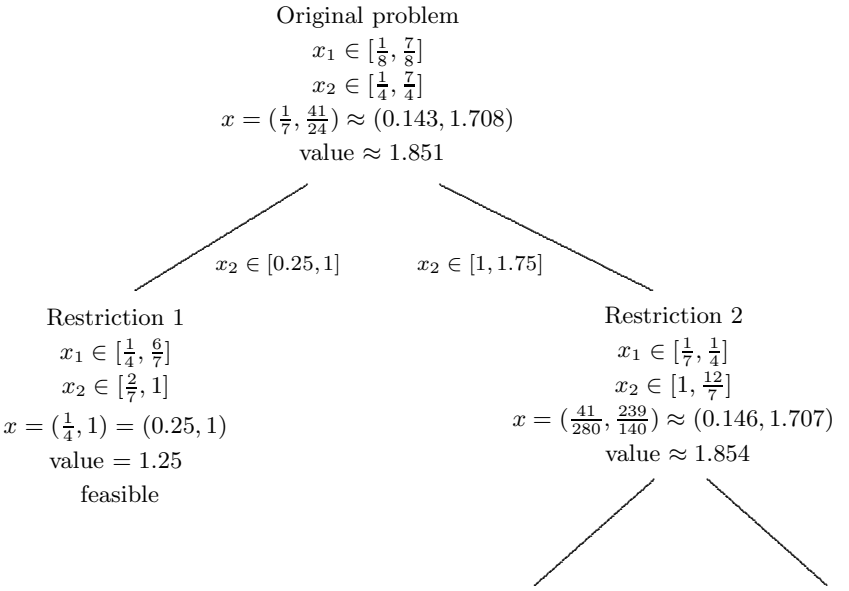

Figure 2.6. A portion of a branch-and-relax tree for a global optimization problem. The reduced domains and solution of the relaxation are shown at each node.

solution $(x_1, x_2) = (0.25, 1)$ of the relaxation is feasible and becomes the incumbent solution. The objective function value is 1.25, which provides a lower bound on the optimal value of the problem. No further branching on the left side is necessary. This, in effect, rules out the locally optimal point $(0.854, 0.293)$ as a global optimum, and it remains only to find the other locally optimal point to a desired degree of accuracy.

Moving to Restriction 2, the solution of the relaxation is $(x_1, x_2) = (\frac{41}{280}, \frac{239}{140}) \approx (0.146429, 1.707143)$. This is slightly infeasible, as $4x_1x_2$ is 0.999898 rather than 1. One could declare this solution to be feasible within tolerances and terminate the search. Alternatively, one could continue branching to find a strictly feasible solution that is closer to the optimum than the incumbent.

Before branching, however, it may be possible to reduce domains by using Lagrange multipliers and the lower bound 1.25 on the optimal value. The constraints $2x_1 + x_2 \leq 2$ and $1/4 \leq U_2x_1 + L_1x_2 - L_1U_2$, respectively, have Lagrange multipliers 1.1 and 0.7 in the solution of the relaxation (2.17) (the remaining constraints have vanishing multipliers since they are not satisfied as equalities). The multiplier 1.1 indicates that any reduction Δ in the right-hand side of $2x_1 + x_2 \leq 2$ reduces the optimal value of the relaxation, currently 1.854, by at least 1.1Δ. Thus any reduction Δ in the left-hand side of the constraint, currently equal

to 2, has the same effect. Since the optimal value should not be reduced below 1.25 (the current lower bound on the optimal value of the original problem), we should have $1.1\Delta \leq 1.854 - 1.25$, or

$$1.1[2 - (2x_1 + x_2)] \leq 1.854 - 1.25$$

This implies $2x_1 + x_2 \geq 1.451$, an inequality that must be satisfied by any optimal solution found in the subtree rooted at the current node. Propagation of this inequality reduces the current domain $[1, 1.714]$ of x_2 to $[1.166, 1.714]$. Similar treatment of the inequality $1/4 \leq U_2x_1 + L_1x_2 - L_1U_2$, which currently is $(12/7)x_1 + (1/7)x_2 \geq 97/196$, has no effect on the domains.

In general, if a constraint $ax \leq a_0$ has Lagrange multiplier λ in the optimal solution of the current relaxation, v is the optimal value of the relaxation, and L is a lower bound on the optimal value of the original problem, then the inequality

$$ax \geq a_0 - \frac{v - L}{\lambda}$$

can be propagated at the current node. This can be particularly useful when the constraint $ax \leq a_0$ is a bound on a single variable, because it provides a direct reduction of that variable's domain.

Unless proper care is exercised, global optimization methods can produce "feasible" solutions that are not actually feasible, due to truncation and other computational errors. The design of *safe* methods that are guaranteed to produce strictly feasible solutions has been the focus of much research in recent years.

2.2.7 Example: Product Configuration

A product configuration example shows how inference and relaxation combine in a slightly more complex problem. It also introduces indexed linear constraints, which occur when the coefficients of a linear constraint are indexed by variables.

The problem is to decide what type of power supply, disk drive, and memory chip to install in a laptop computer, and how many of each. The objective is to minimize total weight while meeting the computer's requirements—upper and lower bounds on memory, disk space, net power supplied, and weight. Only one type of each component may be installed. One power supply, at most three disk drives, and at most three memory chips may be used. Data for a small instance of the problem appear in Table 2.6.

Table 2.6. Data for a small instance of the product configuration problem.

Component i	*Type* k	*Net power generation* A_{i1k}	*Disk space* A_{i2k}	*Memory capacity* A_{i3k}	*Weight* A_{i4k}	*Max number used*
1. Power supply	A	70	0	0	200	1
	B	100	0	0	250	
	C	150	0	0	350	
2. Disk drive	A	−30	500	0	140	3
	B	−50	800	0	300	
3. Memory chip	A	−20	0	250	20	3
	B	−25	0	300	25	
	C	−30	0	400	30	
Lower bound L_j		0	700	850	0	
Upper bound U_j		∞	∞	∞	∞	
Unit cost c_j		0	0	0	1	

Formulating the Problem

To formulate the problem as an optimization model, let variable t_i be the type of component i that is chosen, and q_i the quantity. It is convenient to let variable r_j denote the total amount of attribute j that is produced, where $j = 1, 2, 3, 4$ correspond respectively to net power, disk space, memory, and weight. The computer requires a minimum L_j and maximum U_j of each attribute j.

In general, the objective in such a problem is to minimize the cost of supplying the attributes, where c_j is the unit cost of attribute j. In this case, only the weight is of concern, so that $(c_1, \ldots, c_4) = (0, 0, 0, 1)$.

The interesting aspect of the problem is modeling the connection between the amount r_j of attribute j supplied and the configuration chosen. The component characteristics can be arranged in a three-dimensional array A, in which A_{ijk} is the net amount of attribute j produced by type k of component i. A natural way to write r_j is to sum the contributions of each component

$$r_j = \sum_i A_{ijt_i} q_i$$

where A_{ijt_i} is the amount contributed by the selected type of component i. This leads immediately to the optimization model

$$\text{Linear:} \begin{cases} \min \sum_j c_j r_j & (a) \\ r_j = \sum_i A_{ijt_i} q_i, \ \text{ all } j & (b) \end{cases} \tag{2.18}$$

$$\text{Domains:} \begin{cases} t_1 \in \{\text{A,B,C}\}, t_2 \in \{\text{A,B}\}, t_3 \in \{\text{A,B,C}\} \\ r_j \in [L_j, U_j], \ \text{ all } j \\ q_1 \in \{1\}, \ \ q_2, q_3 \in \{1,2,3\} \end{cases}$$

The optimal solution of this model is $t = (t_1, t_2, t_3) = (\text{C,A,B})$ and $q = (1, 2, 3)$, which results in weight 705. This calls for installation of the largest power supply (type C), two of the smaller disk drives (type A), and three of the medium-sized memory chips (type B).

Constraints (a) of (2.18) are linear equations with some variable indices t_i. They are processed using a specially-structured element constraint. In general, a constraint like (a) contains inequalities of the form

$$\sum_{i \in I} A_{iy_i} x_i + \sum_{i \in I'} A_i x_i \geq b \tag{2.19}$$

where each A_{ik}, each A_i, and b are tuples of equal length. Two such inequalities are used to enforce an equation. (2.19) can be compiled by replacing it with the metaconstraint

$$\text{indexed linear} \begin{cases} \sum_{i \in I} z_i + \sum_{i \in I'} A_i x_i \geq b \\ \text{element}\left(y_i, x_i, z_i \,|\, (A_{i1}, \ldots, A_{im})\right), \ \text{all } i \in I \end{cases} \tag{2.20}$$

Each element constraint sets z_i equal to the y_i^{th} tuple in the list $A_{i1}x_i, \ldots, A_{im}x_i$. Filtering and relaxation techniques can be devised to exploit the special structure of this constraint, which can be called an *indexed linear element* constraint.

Following this pattern, model (2.18) becomes:

$$\begin{aligned}
&\text{Linear: } \min \sum_j c_j r_j \\
&\text{Indexed linear: } \begin{cases} r_j = \sum_i z_{ij}, \quad \text{all } j \\ \text{element}(t_i, q_i, z_{ij} \,|\, (A_{ij1}, \ldots, A_{ijm})) \ , \quad \text{all } i, j \end{cases} \\
&\text{Domains: } \begin{cases} t_1 \in \{\text{A,B,C}\}, t_2 \in \{\text{A,B}\}, t_3 \in \{\text{A,B,C}\} \\ r_j \in [L_j, U_j], \quad \text{all } j \\ q_1 \in \{1\}, \quad q_2, q_3 \in \{1, 2, 3\} \end{cases}
\end{aligned} \tag{2.21}$$

In practice, the solution software should generate this reformulation auto-matically from (2.18). The modeler would not have to be aware of the reformulation.

Inference: Propagation of the Element Constraint

Domain reduction for the linear constraints in (2.21) can be achieved by bounds propagation, as discussed earlier. It remains to propagate the indexed linear metaconstraint.

Domain filtering for the element constraint in an indexed linear metaconstraint can take advantage of its special form. The procedure is quite straightforward if tedious, and the details are presented in Sections 3.8.1 – 3.8.2. Additional filtering may be derived from the fact that the indexed linear constraint implies integer knapsack inequalities. If $[L_{rj}, U_{rj}]$ is the current domain of r_j, then

$$L_{rj} \leq \sum_i A_{ijt_i} q_i \leq U_{rj}$$

This in turn yields the integer knapsack inequalities

$$\begin{aligned}
L_{rj} &\leq \sum_i \max_{k \in D_{t_i}} \{A_{ijk}\} \, q_i \\
\sum_i \min_{k \in D_{t_i}} \{A_{ijk}\} \, q_i &\leq U_{rj}
\end{aligned}$$

which can be used to reduce the domains of the variables q_i.

Three of the attributes in the example problem (power, disk space, and memory), respectively, yield the following knapsack inequalities:

$$\begin{aligned} 0 &\leq \max\{70, 100, 150\}\, q_1 + \max\{-30, -50\}\, q_2 + \\ &\qquad\qquad \max\{-20, -25, -30\}\, q_3 \\ 700 &\leq \max\{0, 0, 0\}\, q_1 + \max\{500, 800\}\, q_2 + \max\{0, 0, 0\}\, q_3 \\ 850 &\leq \max\{0, 0, 0\}\, q_1 + \max\{0, 0\}\, q_2 + \max\{250, 300, 400\}\, q_3 \end{aligned}$$

which simplify to

$$0 \leq 150q_1 - 30q_2 - 20q_3, \quad 700 \leq 800q_2, \quad 850 \leq 400q_3 \tag{2.22}$$

Weight yields no useful inequality as its lower bound is zero. Propagation of these inequalities reduces the domain of q_3 to $\{3\}$ but does not affect the other domains. When all the constraints of (2.21) are propagated, however, the domains are reduced to the following:

$$\begin{array}{lll} q_1 \in \{1\} & q_2 \in \{1, 2\} & q_3 \in \{3\} \\ t_1 \in \{C\} & t_2 \in \{A, B\} & t_3 \in \{B, C\} \\ z_{11} \in [150, 150] & z_{21} \in [-75, -30] & z_{31} \in [-90, -75] \\ z_{12} \in [0, 0] & z_{22} \in [700, 1600] & z_{32} \in [0, 0] \\ z_{13} \in [0, 0] & z_{23} \in [0, 0] & z_{33} \in [900, 1200] \\ z_{14} \in [350, 350] & z_{24} \in [140, 600] & z_{34} \in [75, 90] \\ r_1 \in [0, 45] & r_2 \in [700, 1600] & r_3 \in [900, 1200] \\ r_4 \in [565, 1040] & & \end{array} \tag{2.23}$$

Relaxation: Convex Hull of the Element Constraint

Problem (2.21) is linear except for the indexed linear constraints and the integrality restriction on q_i. It can therefore be relaxed by dropping the integrality condition and finding a linear relaxation for the indexed linear constraints.

The key to relaxing an element constraint of the form

$$\text{element}\,(y, x, z \,|\, (A_1, \ldots, A_m))) \tag{2.24}$$

where $x \geq 0$ is to note that for any given domain of y, the constraint implies a disjunction of linear equations:

$$\bigvee_{k \in D_y} (z = A_k x) \tag{2.25}$$

This says that at least one of the equations $z = A_k x$ must hold. Relaxations for disjunctions of linear systems in general are well studied, and

(2.25) in particular has the relaxation

$$z = \sum_{k \in D_y} A_k x_k, \quad x = \sum_{k \in D_y} x_k$$

where $x_k \geq 0$ are new variables. This is, in fact, a convex hull relaxation, which is the tightest possible linear relaxation. A solution of the relaxation is feasible for (2.24) if exactly one of the x_ks (say x_{k^*}) is positive and therefore equal to x. In this case, z is equated with $A_{k^*}x$.

Based on this idea, the relaxation of (2.21) becomes:

$$\text{Linear:} \begin{cases} \min \sum_j c_j r_j & (a) \\ r_j = \sum_i \sum_{k \in D_{t_i}} A_{ijk} q_{ik}, \ \text{all } j & (b) \\ q_i = \sum_{k \in D_{t_i}} q_{ik}, \ \text{all } i & (c) \\ L_{r_j} \leq r_j \leq U_{r_j}, \ \text{all } j & (d) \\ L_{q_i} \leq q_i \leq U_{q_i}, \ \text{all } i & (e) \\ L_{r_j} \leq \sum_i \max_{k \in D_{t_i}} \{A_{ijk} q_i\}, \ \text{all } j & (f) \\ \sum_i \min_{k \in D_{t_i}} \{A_{ijk} q_i\} \leq U_{r_j}, \ \text{all } j & (g) \\ q_{ik} \geq 0, \ \text{all } i,k & (h) \end{cases} \qquad (2.26)$$

$$\text{Domains: } q_i, q_{ik}, r_j \in \Re, \ \text{all } i,j,k$$

The relaxation can be strengthened with cuts derived from the knapsack inequalities (f) and (g). For example, the first inequality in (2.22) yields the cuts $q_1 + (1 - q_3) \geq 1$ and $q_1 + (1 - q_2) \geq 1$, while the second and third, respectively, yield $q_2 \geq 1$ and $q_3 \geq 3$. In this particular example, knapsack cuts do not strengthen the relaxation.

The solution of (2.26), given the reduced domains (2.23), is $q_1, q_{13} = 1$, $q_2, q_{2A} = 2$, and $q_3, q_{3B} = 3$, with the other q_{ik}s equal to zero. Since the q_is are integer, and exactly one $q_{ij} > 0$ for each i, the solution is feasible and no branching is necessary. The nonzero q_{ij}s indicate that $(t_1, t_2, t_3) = (C, A, B)$. Since $(r_1, \ldots, r_4) = (15, 1000, 900, 705)$, the minimum weight is $r_4 = 705$.

Branching Search

A solution of the relaxation (2.26) is feasible for the original problem (2.21) unless it violates the integrality constraint for some q_i or fails

to satisfy the indexed linear constraint. In the former case, branching follows the usual pattern. If $\hat{q}_i$ is a noninteger value in the solution of the relaxation, the domain $\{L_{q_i}, \ldots, U_{q_i}\}$ of q_i is split into $\{L_{q_i}, \ldots, \lfloor \hat{q}_i \rfloor\}$ and $\{\lceil \hat{q}_i \rceil, \ldots, U_{q_i}\}$. At either branch, the smaller domains are propagated and the resulting relaxation solved.

The solution of the relaxation violates the indexed linear constraint for some i when at least two q_{ik}s are positive, say q_{ik_1} and q_{ik_2}. In this case, the search can branch by splitting the domain of t_i into two sets, one excluding k_1 and the other excluding k_2.

2.2.8 Branch and Price

Branch and price is a special case of branch and relax designed for problems with an integer programming formulation—that is, a formulation consisting of linear constraints and integer-valued variables. Despite this restriction, the method is of interest here because it has become one of the more popular settings for integrated problem solving. It has proved particularly successful for airline crew scheduling and other transport-related applications.

Branch and price can be attractive when the integer programming model has a huge number of variables. In such cases the variables are added to the model only as needed to solve the continuous relaxation. A variable that does not occur in the current model can be *priced* to determine whether its addition would result in a better solution. Variables are added until no further improvement is possible, at which point the continuous relaxation has been solved. Typically, only a small fraction of the total variable set is needed to reach an optimal solution. Once the continuous relaxation is solved, one can branch as in ordinary branch-and-relax methods and solve the continuous relaxations at the branches by adding further variables.

The pricing problem can be solved by any convenient method, and this is where other technologies may enter the picture. Constraint programming, in particular, can be useful for this purpose, because it can deal with the often complex constraints that must be observed when the variables are generated. Airline crew scheduling, for example, is normally constrained by a host of complicated work rules.

An integer programming problem has the form

$$\begin{aligned} &\min\ cx \\ &Ax \geq b,\ \ x \geq 0 \text{ and integral} \end{aligned} \tag{2.27}$$

where A is an $m \times n$ matrix. Its continuous relaxation is

$$\begin{aligned} &\min\ cx \\ &Ax \geq b,\ \ x \geq 0 \end{aligned} \tag{2.28}$$

Since the expression Ax can be written $\sum_j A_j x_j$, where A_j is the jth column of A, adding a variable x_j to the problem can be viewed as adding a column to the coefficient matrix. It is for this reason that branch-and-price methods are often called *column generation* methods.

The problem restriction at each node of the branch-and-relax tree includes some constraints due to branching and likewise has the form (2.27). The task is to solve its continuous relaxation (2.28). When A has a huge number of columns, (2.28) is initially formulated with only a small subset of the columns:

$$\begin{aligned} &\min\ cx \\ &\sum_{j \in J} A_j x_j \geq b \\ &x_j \geq 0,\ \text{all } j \in J \end{aligned} \tag{2.29}$$

This is the *restricted master problem*. It is a restriction of (2.27) because any feasible solution of it is a feasible solution of (2.27) with the remaining variables set to zero.

When (2.29) is solved by linear programming software, one typically obtains not only optimal values for each x_j but an optimal *dual multiplier* u_i for each inequality constraint $\sum_j A_{ij} x_j \geq b_j$. Dual multipliers are a special case of Lagrange multipliers and will be discussed in Section 3.3. For present purposes, they can be viewed as the marginal rate of increase in the optimal cost cx for each unit increase in the right-hand side b_i. It will be seen that adding a new variable x_j ($j \notin J$) to the problem and re-solving can improve the optimal solution only if

$$c_j - \sum_i u_i A_{ij} < 0 \tag{2.30}$$

or more succinctly, $c_j - uA_j < 0$, where u is the row vector $[u_1 \cdots u_m]$. Intuitively, this is because increasing the value of x_j (now zero since x_j is not in the problem) by one unit increases cost by c_j units but has the same effect as forcing each right-hand side b_i down by A_{ij} units, which reduces the cost by $\sum_j u_i A_{ij}$ units. The net effect on cost is the left-hand side of (2.30), which is known as the *reduced cost* of x_j. If the reduced cost is negative, then the cost is brought down and the solution improved by introducing x_j to the problem and giving it some positive value. The process of evaluating the reduced cost of x_j is known as *pricing* the variable.

The problem of finding a variable with negative reduced cost can be solved in any number of ways. For instance, one can use a heuristic algorithm that generates columns of A until it finds one with a negative

reduced cost. If no such column is found, then one must switch to an exact algorithm to make sure there is no improving column. If there is none, then the current solution of the restricted master problem (2.27) is optimal for (2.28). Another approach, explored in the next section, is to solve the pricing problem with a combination of optimization and constraint programming techniques.

2.2.9 Example: Airline Crew Scheduling

A simplified airline crew scheduling problem shows how constraint programming and mathematical programming ideas can be combined in a branch-and-price framework. It also illustrates set-valued variables, which have become important in constraint programming solvers.

The goal is to assign flight crew members to flights so as to minimize cost while covering all the flights and observing a number of work rules. Each flight j starts at time s_j, finishes at time f_j, and requires n_j crew members. A small example with six flights appears in Table 2.7. Whenever a crew member staffs two consecutive flights, the rest period between the flights must be at least $\Delta_{\min}$ and at most $\Delta_{\max}$. The total flight time assigned to a crew member must be between $T_{\min}$ and $T_{\max}$. There may be other restrictions on scheduling as well.

Integer Programming Formulation

The set of flights assigned to a crew member is known as a *roster*. The problem can in principle be formulated by generating all possible rosters and assigning one roster to each crew member in such a way as to cover every flight. The cost c_{ik} of assigning crew member i to roster k depends on a number of factors, such as seniority, the timing of flights and rest periods, and so forth.

Let δ_{jk} be 1 when flight j is part of roster k, and 0 otherwise. Also let 0-1 variable $x_{ik} = 1$ when crew member i is assigned roster k. Then, the

Table 2.7. Flight data for a small crew scheduling problem.

j	s_j	f_j
1	0	3
2	1	3
3	5	8
4	6	9
5	10	12
6	14	16

problem can be formulated with an integer programming model in which dual multipliers u_i, v_{ik} are associated with the constraints as shown:

$$\text{0-1 linear:} \begin{cases} \min \sum_{ik} c_{ik} x_{ik} & \\ \sum_{k} x_{ik} = 1, \ \text{ all } i & (u_i) \\ \sum_{i} \sum_{k} \delta_{jk} x_{ik} \geq n_j, \ \text{ all } j & (v_j) \end{cases} \tag{2.31}$$

$$\text{Domains: } x_{ik} \in \{0, 1\}, \text{ all } i, k$$

Suppose for the example of Table 2.7 that the minimum and maximum gap between flights are $(\Delta_{\min}, \Delta_{\max}) = (2, 3)$, and the minimum and maximum flight times are $(T_{\min}, T_{\max}) = (6, 10)$. Thus, Flight 6 cannot immediately follow Flight 1, since the gap is too large (11), and Flight 5 cannot immediately follow Flight 4, since the gap is too small (1). The constraints permit four possible rosters: $\{1, 3, 5\}$, $\{1, 4, 6\}$, $\{2, 3, 5\}$, and $\{2, 4, 6\}$. If there are two crew members, and each flight requires one crew member, the continuous relaxation of problem (2.31) becomes:

$$\begin{aligned} &\min \ z \\ &\begin{bmatrix} 10 & 12 & 7 & 13 & 9 & 11 & 6 & 12 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_{11} \\ x_{12} \\ x_{13} \\ x_{14} \\ x_{21} \\ x_{22} \\ x_{23} \\ x_{24} \end{bmatrix} \begin{matrix} = \\ = \\ = \\ \geq \\ \geq \\ \geq \\ \geq \\ \geq \\ \geq \end{matrix} \begin{bmatrix} z \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \\ &x_{ik} \geq 0, \text{ all } i, k \end{aligned} \tag{2.32}$$

The problem is written in matrix form to show the eight columns, which correspond to the four possible rosters for each of the two crew members. The top row of the matrix contains the costs c_{ik}.

Rather than solve the complete problem (2.32), the problem is first solved with a subset of the columns, perhaps the following:

$$
\begin{array}{l}
\min\ z \\
\begin{bmatrix} 10 & 13 & 9 & 12 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix}
\begin{bmatrix} x_{11} \\ x_{14} \\ x_{21} \\ x_{24} \end{bmatrix}
\begin{array}{c} = \\ = \\ = \\ \geq \\ \geq \\ \geq \\ \geq \\ \geq \\ \geq \end{array}
\begin{bmatrix} z \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}
\begin{array}{c} \\ (10) \\ (9) \\ (0) \\ (0) \\ (0) \\ (0) \\ (0) \\ (3) \end{array} \\
x_{ik} \geq 0, \text{ all } i, k
\end{array}
\tag{2.33}
$$

The optimal solution is $(x_{11}, x_{14}, x_{21}, x_{24}) = (0, 1, 1, 0)$. The corresponding dual multipliers $u = (10, 9)$ and $v = (0, 0, 0, 0, 0, 3)$ are shown on the right.

The Pricing Problem

The pricing problem is to identify a variable of (2.31) with a negative reduced cost. A column associated with x_{ik} contains a 1 in the row corresponding to u_i and the rows corresponding to v_j for each flight j in the roster. The reduced cost of x_{ik} is therefore

$$c_{ik} - u_i - \sum_{j \text{ in roster } k} v_j \tag{2.34}$$

One way to formulate the pricing problem is to model it with a directed graph. The graph contains a node j for each flight, plus a source node s and sink node t. The graph contains a directed arc (j, j') when flight j can immediately precede j' in a roster; that is, $\Delta_{\min} \leq s_{j'} - f_j \leq \Delta_{\max}$. There are also arcs (s, j) and (j, t) for every flight j. Every possible roster corresponds to a path from s to t, although not every path corresponds to a roster, because the total fight time may not be in the range $[T_{\min}, T_{\max}]$.

The graph corresponding to the example appears in Figure 2.7. Only some of the arcs incident to s and t are shown, since the others can be removed by elementary preprocessing of the graph. For example, arc $(s, 3)$ is removed because every path from 3 to t results in a total flight time less than the minimum of $T_{\min} = 6$. This can be ascertained by computing the longest path from 3 to t when the length of each arc (j, j') is set to the duration of flight j.

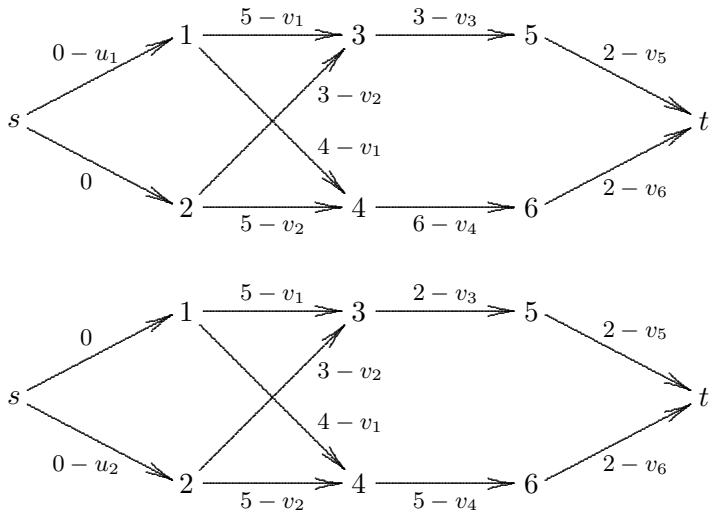

Figure 2.7. Path model for crew rostering. The top graph pertains to crew member 1, and the bottom graph to crew member 2. The arc lengths reflect costs offset by dual multipliers u_i, v_j.

The graph-based model assumes that the cost c_{ik} associated with x_{ik} can be equated with the length of the corresponding path, if the arc lengths are suitably defined. This is possible if the cost depends only on the cost $\alpha_{jj'}$ of staffing each flight j with crew member i and transferring him or her to the next flight j'. The length of each arc (j, j') is set equal to $\alpha_{jj'}$, the length of each (s, j) to the cost α_{sj} of transferring from home to flight j, and the length of (j, t) to the cost α_{jt} of operating flight j and returning home. The arc costs associated with each crew member in the example problem are shown in Figure 2.7 (ignore the terms u_i and v_j for the moment).

The reduced cost (2.34) of a variable x_{ik} can also be equated with the length of the corresponding path, if the arc lengths are offset by the dual multipliers. The cost term c_{ik} in (2.34) is the path length using the arc lengths just defined. The path length becomes the reduced cost if the length of each (j, j') is set to $\alpha_{jj'} - v_j$, the length of each (s, j) to $\alpha_{sj} - u_i$, and the length of each (j, t) to $\alpha_{jt} - v_j$. These adjustments are shown in Figure 2.7.

The pricing problem is now the problem of finding a path in the directed graph G with negative length, for which the total flight time lies in the interval $[T_{\min}, T_{\max}]$. Let X_i be the set of flights assigned to crew member i. Since each path is specified by some X_i, the problem

can be written as follows:

$$\begin{aligned} &\text{Path: } (X_i, z_i \mid G, c, s, t), \text{ all } i \\ &\text{Set sum: } T_{\min} \leq \sum_{j \in X_i} p_j \leq T_{\max}, \text{ all } i \\ &\text{Domains: } X_i \subset \{\text{flights}\}, \; z_i < 0, \text{ all } i \end{aligned} \tag{2.35}$$

where $p_j = f_j - s_j$ is the duration of flight j. The path metaconstraint requires that the nodes in X_i define a path from s to t in G with length z_i, where c contains the edge lengths c_{ij}. Since z_i's domain elements are negative, the constraint enforces a negative reduced cost. The set sum constraint simply requires that the total flight duration be within the prescribed bounds.

In the example of Figure 2.7, the arc lengths are defined by the dual multipliers $u = (10, 9)$ and $v = (0, 0, 0, 0, 0, 3)$. The shortest path in the graph for Crew Member 1 is s-1-4-6-t, which has length -1. This defines the column corresponding to variable x_{12} in (2.32). The shortest path for Crew Member 2 is s-2-3-5-t with length -2, which defines the column corresponding to x_{23}. So variables x_{12} and x_{23} have negative reduced costs, and their columns are added to the restricted master problem (2.33). The new solution is $(x_{11}, x_{12}, x_{14}, x_{21}, x_{23}, x_{24}) = (0, 1, 0, 0, 1, 0)$ with dual multipliers $u = (10, 5)$ and $v = (0, 1, 0, 0, 0, 2)$. When the arc lengths are updated accordingly, the shortest path for both crew members has length zero. This means there is no improving column, and the solution of the restricted master is optimal. Since this solution is integral, there is no need to branch. The optimal rostering assigns Flights 1, 4 and 6 to Crew Member 1, and the remaining flights to Crew Member 2.

Bounds Propagation for Set-Valued Variables

The pricing problem (2.35) can, in general, be solved by a combination of bounds propagation and branching. Bounds propagation must be reinterpreted for the variables X_i since they are set valued, but the idea is straightforward. The domain of each X_i is stored in the form of bounds $[L_{X_i}, U_{X_i}]$, where L_{X_i} is a set that X_i must contain, and U_{X_i} is a set that must contain X_i. Initially L_{X_i} is empty and U_{X_i} is the set of all flights, but branching and bounds propagation may tighten these bounds. For instance, if branching fixes $j \in X_j$, then $L_{X_i} = \{j\}$.

The path constraint in (2.35) tightens bounds for both z_i and X_i. It updates the lower bound L_{z_i} on z_i by finding a shortest path from s to t whose node set lies in the range $[L_{X_i}, U_{X_i}]$. If ℓ is the length of the shortest path, then L_{z_i} is updated to $\max\{L_{z_i}, \ell\}$. If $\ell \geq 0$, the domain

of z_i becomes empty because U_{z_i} is initially negative, and there is no variable with negative reduced cost (the constraint also updates U_{z_i} by finding a longest path, but this is not relevant here). Since G is acyclic, there are very fast algorithms for finding a shortest path, despite the possibility of negative arc lengths. These algorithms can be adapted to ensure that the node set lies in the range $[L_{X_i}, U_{X_i}]$ by temporarily modifying G. To make sure the path nodes belong to U_{X_i}, simply delete all nodes in G that are not in U_{X_i}. To make sure all nodes in L_{X_i} belong to the path, avoid routing around nodes in L_{X_i}. That is, whenever G contains an arc (j_1, j_3) and a path from j_1 to j_3 through a third node $j_2 \in L_{X_i}$, delete the arc (j_1, j_3) from G.

The modified graph G' can be used to tighten the bounds for X_i as well. If every path from s to t in G' contains node j, then add j to L_{X_i}. If no path contains j, then remove j from U_{X_i}. This can be accomplished by supposing that the arcs of G' carry flow in the direction of their orientation. It is assumed that a flow of 1 unit volume enters G' at node s and exits at node t. If the minimum flow through node j subject to these conditions is 1, then j contains all paths from s to t, and if the maximum is 0, it contains no paths. Minimum and maximum flow problems of this sort can be solved very quickly. Thus, if $L_{X_i} = \{3\}$ in the example, the minimum flow through node 5 in G' is 1 and the maximum through 4 is 0, which indicates that node 5 may be added to L_{X_i} and node 4 removed from U_{X_i}.

The set sum constraint can also be used for propagation in the obvious way. Let $[L_{z_i}, U_{z_i}]$ be the current domain for z_i. One can update L_{z_i} to

$$\max\left\{ L_{z_i}, \sum_{j \in L_{X_i}} p_j + \sum_{j \in U_{X_i} \setminus L_{X_i}} \min\{0, p_j\} \right\} \tag{2.36}$$

and analogously for U_{z_i}. Also, if

$$p_k + \sum_{j \in L_{X_i}} p_j > U_{z_i} \quad \text{or} \quad p_k + \sum_{j \in L_{X_i}} p_j < L_{z_i}$$

for $k \notin L_{X_i}$, then k can be removed from U_{X_i}. One can also write a sufficient condition for adding k to L_{X_i}.

Branching as well as propagation may be required to solve the pricing problem. A standard way to branch on a set-valued variable X_i is to consider the options $j \in X_i$ and $j \notin X_i$, where $j \in U_{X_i} \setminus L_{X_i}$. The former branch is implemented by adding j to L_{X_i}, and the latter by removing j from U_{X_i}.

The pricing problem discussed here contains only two metaconstraints. Realistic problems must deal with many complicated work rules. Spe-

cialized constraints and associated filtering methods have been developed for some of these.

2.2.10 Exercises

1 Write two integer linear inequalities (with initial domains specified) for which bounds propagation reduces domains more than minimizing and maximizing each variable subject to a continuous relaxation of the constraint set. Now write two integer linear inequalities for which the reverse is true.

2 Identify all the packings of $6x_1 + 4x_2 + 3x_3 + x_4 \geq 29$, with $x_i \in \{0, 1, 2, 3, 4\}$ for $i = 1, \ldots, 4$. Write the corresponding knapsack cuts.

3 Define a *cover* for an inequality $ax \leq a_0$ that is analogous to a packing for $ax \geq a_0$. What is the knapsack cut corresponding to a given cover?

4 Solve the problem of minimizing $3x_1 + 4x_2$ subject to $2x_1 + 3x_2 \geq 10$ and $x_1, x_2 \in \{0, 1, 2, 3\}$ using branch and relax without bounds propagation. Now, solve it using branch and infer without relaxation. Finally, solve it using branch and relax with propagation. Which results in the smallest search tree? (When using branch and infer, follow the common constraint programming practice of branching on the variable with the smallest domain.)

5 Write the knapsack cuts that correspond to the nonmaximal packings $I = \{1\}, \{2\}, \{3\}, \{4\}$ for constraint (2.3a). If a linear programming solver is available, verify that when these cuts are added to the relaxation (2.6), the resulting branch-and-relax tree has five rather than seven nodes.

6 Formulate the following problem with the help of conditional constraints. A lumber operation wishes to build temporary roads from Forests 1 and 2 to Sawmills A and B. Due to topography and environmental regulations, roads can be built only in certain combinations: (a) from 1 to A, 1 to B, and 2 to B; (b) from 1 to A, 1 to B, and 2 to A; (c) from 1 to A, 2 to A, and 2 to B. Each sawmill j requires d_j units of timber, the unit cost of shipping timber over a road from forest i to sawmill j is c_{ij}, and the fixed cost of a road from i to j is f_{ij}. Choose which roads to build, and how much timber to transport over each road, so as to minimize cost while meeting the sawmill demands. Now, formulate the problem without conditional constraints and with a disjunctive linear constraint.

7 Consider the problem

$$\begin{aligned}&\text{Linear: } \min\ cx\\ &\text{Logic: } \begin{cases} y_1 \vee y_2\\ y_1 \rightarrow y_3\end{cases}\\ &\text{Conditional: } y_k \Rightarrow \left(A^k x \geq b^k\right),\ k = 1, 2, 3\end{aligned}$$

where each y_k is a boolean variable. Write the problem without conditional constraints and using as many linear disjunctions as possible. Hint: the formula $y_1 \rightarrow y_3$ is equivalent to $\neg y_1 \vee y_3$.

8 Formulate the following problem using the appropriate global constraints. There are six security guards and four stations. Each station must be staffed by exactly one guard each night of the week. A guard must be on duty four or five nights a week. No station should be staffed by more than three different guards during the week. A guard must never staff the same station two or more nights in a row and must staff at least three different stations during the week. A guard must not staff Stations 1 and 2 on consecutive nights (in either order), and similarly for Stations 3 and 4. Every week will have the same schedule.

9 Write all feasible solutions of the stretch-cycle constraint in (2.11) to confirm that the domains in (2.12) can be reduced to (2.13) and cannot be reduced further.

10 Formulate this lot sizing problem using a combination of variable indices, linear constraints, and stretch constraints. In each period t, there is a demand d_{it} for product i, and this demand must be met from the stock of product i at the end of period t. At most one product can be manufactured in each period t, represented by variable y_t. If nothing is manufactured, $y_t = 0$, where product 0 is a dummy product. The quantity of product i manufactured in any period must be either q_i or zero. When product i is manufactured, its manufacture must continue no fewer than ℓ_i and no more than u_i periods in a row. The manufacture of product i in any period must be followed in the next period by the manufacture of one of the products in S_i (one may assume $0, i \in S_i$). The unit holding cost per period for product i is h_i, and the unit manufacturing cost is g_i. The setup cost of making a transition from product i in one period to product j in the next is c_{ij} (where possibly i and/or j is 0). Minimize total manufacturing, holding, and setup costs over an n-period horizon while

meeting demand. After formulating the problem, indicate how to replace the variable indices with element constraints. Hint: let variable x_{ij} represent the quantity of product i manufactured in period t, and s_{it} the stock at the end of the period. The element(y, x, z) constraint also has a form element$(y, z \mid a)$. where a is a tuple of constants. It sets z equal to a_y.

11 A difficulty with the model of the previous exercise is that the setup cost after an idle period is always the same, regardless of which (non-dummy) product was manufactured last. How can the model be modified to allow setup cost to depend on the last non-dummy product? Hint: define several dummy products.

12 Formulate the following problem using linear disjunctions, variable indices, and stretch-cycle constraints. The week is divided into n periods, and in each period t, d_t megawatts of electricity must be generated. Each power plant i generates at most q_i megawatts while operating. Once plant i is started, it must run for at least ℓ_i periods, and once shut down, it must remain idle for at least ℓ'_i periods. The cost of producing one megawatt of power for one period at plant i is g_i, and the cost of starting up the plant is c_i. Determine which plants to operate in each period to minimize cost while meeting demand. The schedule must follow a weekly cycle. Indicate how to replace the variable indices with element constraints. Hint: let $y_{it} = 1$ if plant i operates in period t, and let the setup cost incurred by plant i in period t be $c_{iy_{i,t-1}y_{it}}$, where $c_{i01} = c_i$ and $c_{i00} = c_{i01} = c_{i11} = 0$.

13 Derive a linear relaxation for $y = x^2$ from (2.16), and derive a linear relaxation for $y = a^x$ when $a > 0$. Now, write a factored relaxation for the problem

$$\begin{aligned} &\max\ a^{x_1} + b^{x_2} \\ &(x_1 + 2x_2)^2 \le 1,\ \ L_j \le x_j \le U_j,\ j = 1, 2 \end{aligned}$$

when $a, b > 0$.

14 Consider the problem:

$$\begin{aligned} &\min\ x_1^2 + x_2^2 \\ &3x_1 + x_2 \le 15 \\ &x_1, x_2 \ge 0 \text{ and integral} \end{aligned}$$

Suppose that the best known feasible solution for the problem is $(x_1, x_2) = (4, 3)$. The continuous relaxation has the optimal solution

$(x_1, x_2) = (4.5, 1.5)$ with a Lagrange multiplier of 3 corresponding to the first constraint. Derive an inequality that must be satisfied by any optimal solution. (Round down the right-hand side, since the coefficients and variables are integral).

15 What knapsack covering inequality can be inferred from the meta-constraint (2.20)?

16 A farmer wishes to apply fertilizer to each of several plots. The additional crop yield from plot i per unit of type k fertilizer applied is a_{ik}, and the runoff into streams is c_{ik}. There are storage facilities on the farm for at most k different kinds of fertilizer. Formulate the problem of identifying which fertilizer to apply to each plot, and how much, to maximize total additional yield subject to an upper limit U on the amount of runoff. Now, reformulate the problem using an indexed linear constraint. Hint: the nvalues constraint is useful here.

17 Recall the crew rostering problem illustrated by Figure 2.7. Suppose the current domain of the set-valued variable X_1 is the interval $[L_{X_1}, U_{X_1}]$ where $L_{X_1} = \{6\}$ and $U_{X_1} = \{1, 2, 3, 4, 6\}$. Use the min and max flow test to filter the domain.

18 If the third column is omitted from (2.33), the resulting dual multipliers are $(u_1, u_2) = (10, 9)$ and $(v_1, \ldots, v_6) = (0, 3, 0, 0, 0, 0)$. Use the graphs of Figure 2.7 to identify one or more columns with negative reduced cost.

19 Section 2.2.9 describes how to modify an acyclic graph G to obtain a graph G' such that finding a shortest path in G' finds a shortest path in G that includes a given subset of nodes. Show that this method breaks down when the graph is not acyclic.

20 Suppose the graph G in the previous exercise is acyclic, and its nodes are numbered so that $i < j$ whenever (i, j) is an arc. Design an algorithm based on this ordering to find the modified graph G'.

21 Write an expression analogous to (2.36) for updating U_{z_i}.

22 Consider the set sum constraint $\ell \leq \sum_{j \in X} p_j \leq u$, where X is a set-valued variable with domain $[L_X, U_X]$. Write two conditions that are jointly necessary and sufficient for the set sum constraint to be feasible. If the constraint is feasible and $k \notin L_X$, write two conditions, either of which is sufficient to add k to L_X.

23 In a two-dimensional cutting stock problem, customers have placed orders for q_i rectangular pieces of glass having size i (which has dimensions $a_i \times b_i$) for $i = 1, \ldots, m$. These orders must be filled by

cutting them from standard $a \times b$ sheets. There are many patterns according to which a standard sheet can be cut into one or more of the pieces on order. Let y_j be the number of standard sheets that are cut according to pattern j, and let A_{ij} be the number of pieces of size i the pattern yields from one sheet. Write an integer programming problem that minimizes the number of standard sheets cut while meeting customer demand. Indicate how to solve the problem by column generation by formulating a subproblem to find a pattern with a negative reduced cost. The subproblem should use the diffn global constraint (see Chapter 5). Hint: To allow for multiple pieces of the same size, define several distinct 0-1 variables δ_k for each size i, and let $\delta_k = 1$ if a piece of size i is cut. Then for each i, and k corresponding to size i, there is a term $u_i\delta_k$ in the objective function of the subproblem, where u is the tuple of dual multipliers, and a linear disjunction

$$\begin{bmatrix} \delta_k = 1 \\ \Delta x_k = (a_i, b_i) \end{bmatrix} \vee \begin{bmatrix} \delta_k = 0 \\ \Delta x_k = (0, 0) \end{bmatrix}$$

in the constraint set.

2.3 Constraint-Directed Search

An ever-present issue when searching over problem restrictions is the choice of which restrictions to consider and in what order. Branching search addresses this issue in a general way by letting problem difficulty guide the search. If a given restriction is too hard to solve, it is split into problems that are more highly restricted, and otherwise one moves on to the next restriction, thus determining the sequence of restrictions in a recursive fashion.

Another general approach is to create the next restriction on the basis of lessons learned from solving past restrictions. At the very least, one would like to avoid solving restrictions that are no better than past ones, in the sense that they cannot produce solutions any better those already found. This can be accomplished as follows. Whenever a restriction is solved, one can generate a constraint that excludes that restriction, and perhaps others that are no better. Such a constraint is a *nogood*. Then, when the next restriction is selected, it must satisfy the nogoods generated so far. The process continues until no restriction satisfies the nogoods, indicating an exhaustive search. At this point, the best candidate solution found in the process of solving restrictions is optimal in the original problem.

To put this more precisely, the search proceeds by creating a sequence of restrictions $P_1, P_2, \ldots$ of the original problem P. The optimal value v_k of each restriction P_k is computed and a nogood N_k derived. The nogood is designed to exclude P_k and perhaps other restrictions with optimal values no better than v_k. The next restriction P_{k+1} is selected so that it satisfies the set $\mathcal{N}_k$ of nogoods $N_1, \ldots, N_k$ generated so far (the initial nogood set $\mathcal{N}_0$ is empty). The algorithm continues until no restriction satisfies $\mathcal{N}_k$, at which point the optimal value of P is the minimum of $v_1, \ldots, v_k$. If the optimal value is infinite, P is infeasible.

The search is exhaustive because it excludes a restriction only when the restriction is infeasible, or some restriction already solved has a solution that is at least as good. It necessarily terminates if there are a finite number of possible restrictions, because each iteration excludes at least one restriction. If there are infinitely many restrictions, some care must be taken in designing the nogoods to ensure that the search is finite.

This formal process does not indicate in what sense a restriction may be viewed as satisfying or violating a constraint N_k. In practice the restrictions are parameterized by the variables $x = (x_1, \ldots, x_n)$, so that each $x \in D$ determines a restriction $P(x)$ with optimal value $v(x)$. The nogoods are written as constraints on the problem variables x rather than constraints on restrictions. The nogoods exclude restrictions by excluding values of x that give rise to those restrictions. Thus, in iteration k, a solution x^k of the current nogood set $\mathcal{N}_k$ is found and the next restriction is $P(x^k)$.

The most obvious way to define a restriction $P(x^k)$ is simply to fix x to x^k. In this case, a nogood is generated to exclude x^k (and perhaps other solutions that are no better than x^k). The next solution examined must satisfy all the nogoods so far generated, which means that it must differ from all the solutions so far enumerated. The search terminates when the nogoods rule out all possible solutions.

Typically, however, x^k defines a restriction by fixing only some of the variables in x to their values in x^k. That is, $P(x^k)$ is defined by adding to P the constraint $(x_{j_1}, \ldots, x_{j_p}) = (x^k_{j_1}, \ldots, x^k_{j_p})$, where p $(< n)$ and the index set $\{j_1, \ldots, j_p\}$ may vary from one iteration to the next. This occurs, for example, in constraint-directed branching and the various forms of Benders decomposition. The method by which nogoods are constructed is problem-dependent and ideally exploits the problem structure to obtain strong nogoods. A generic constraint-directed search algorithm appears in Figure 2.8.

There is normally a good deal of freedom in how to select a feasible solution x^k of $\mathcal{N}_k$, and a constraint-directed search is partly characterized

Let $v_{\text{UB}} = \infty$ and $\mathcal{N} = \emptyset$.
Associate a restriction $P(x)$ of P with each $x \in D$, and let $v(x)$
be the optimal value of $P(x)$.

While $\mathcal{N}$ is feasible repeat:
Select a feasible solution $x = s(\mathcal{N})$ of $\mathcal{N}$.
Compute $v(x)$ by solving $P(x)$ and let $v_{UB} = \min\{v(x), v_{UB}\}$.
Define a nogood N that excludes x and possibly other solutions x'
with $v(x') \geq v(x)$.
Add N to $\mathcal{N}$ and process $\mathcal{N}$.
The optimal value of P is v_{UB}.

Figure 2.8. Generic constraint-directed search algorithm for solving a minimization problem P with variable domain D, where s is the selection function. $\mathcal{N}$ contains the nogoods generated so far.

by its *selection function*; that is, by the way it selects a feasible solution for a given $\mathcal{N}_k$. Certain selection functions can make subsequent $\mathcal{N}_k$s easier to solve—a theme that is illustrated below. In some constraint-directed methods, such as dynamic backtracking methods, it may be necessary to *process* the nogood set $\mathcal{N}_k$ to make it easier to solve. In such cases, the selection function is designed to make processing easier. In partial-order dynamic backtracking, for example, a solution of $\mathcal{N}_k$ is selected to *conform* to previous solutions, so that $\mathcal{N}_k$ can be processed by a fast version of resolution (parallel resolution).

It may not be evident at this point how constraint-directed search reflects the search-infer-and-relax paradigm. There is obviously an enumeration of problem restrictions, but no mention has been made of relaxation. Section 2.3.5 will show, however, that each nogood set $\mathcal{N}_k$ is naturally viewed as a relaxation R_k of the problem. The search process, therefore, solves a series of relaxations whose solutions guide the choice of the next restriction. Unlike branching methods, constraint-directed search requires the solution of every restriction, regardless of the outcome of solving the previous relaxation.

Subsequent sections describe three instances of constraint-directed search: constraint-directed branching, partial-order dynamic backtracking, and logic-based Benders decomposition. Table 2.8 indicates briefly how these methods exemplify the search-infer-and-relax scheme.

2.3.1 Constraint-Directed Branching

Constraint-directed branching stems from the observation that branching on variables is a special case of constraint-directed search. The leaf nodes of the branching tree correspond to problem restrictions. The no-

Table 2.8. How some selected constraint-directed search methods fit into the search-infer-and-relax framework.

Solution method	*Restriction* P_k	*Relaxation* R_k	*Selection function* $s(R_k)$	*Inference*
DPL for SAT	Created by adding conflict clauses	Processed conflict clauses	Unit clause rule + greedy solution of R_k	Nogood generation + parallel resolution
Partial-order dynamic backtracking	Created by adding nogoods	Processed nogoods	Greedy, but consistent with partial order	Nogood generation + parallel resolution
Logic-based Benders decomposition	Subproblem defined by solution of master	Master problem (Benders cuts)	Optimal solution of master	Derivation of Benders cuts (nogoods)

goods help guide future branching and contain information about why the search backtracked at previous leaf nodes. Well-chosen nogoods can permit the search to prune large portions of the enumeration tree.

Search algorithms of this sort are widely used in artificial intelligence and constraint programming. Nogoods in the form of *conflict clauses* have played a particularly important role in fast algorithms for the propositional satisfiability problem—a key problem for combinatorial optimization.

Branching can be understood as constraint-directed search in the following way. To simplify discussion, suppose that a feasible solution, rather than an optimal solution, is sought. The initial nogood set $\mathcal{N}_0$ is empty. The branching process reaches the first leaf node by fixing some of the variables $(x_1, \ldots, x_p)$ to certain values $(v_1, \ldots, v_p)$. These assignments partially define a feasible solution x^0 for $\mathcal{N}_0$. The remaining variables $x_{p+1}, \ldots, x_n$ can be assigned values from their current domains, as desired. Thus, the branching mechanism helps to define the selection function due to its role in choosing a feasible solution for $\mathcal{N}_0$.

The restriction P_1 corresponding to the first leaf node fixes $(x_1, \ldots, x_p)$ to $(v_1, \ldots, v_p)$. Thus, P_1 contains the original constraints of P plus the branching constraints $x_j = v_j$ that bring the search to the leaf node.

If the solution $(x_1, \ldots, x_n) = (v_1, \ldots, v_n)$ is feasible in P, the search terminates at the first leaf node. If the search backtracks, it is because fixing $(x_1, \ldots, x_p)$ to $(v_1, \ldots, v_p)$ violates some constraint. In other words, P_1 is infeasible. Typically, only some of the fixed variables $x_1, \ldots, x_p$ are actually responsible for the violations, perhaps $x_{j_1}, \ldots, x_{j_r}$. A nogood

$$(x_{j_1}, \ldots, x_{j_r}) \neq (v_{j_1}, \ldots, v_{j_r})$$

is constructed to prevent assigning the same values to these critical variables again. From here out the search avoids taking branches that are inconsistent with the nogood.

Each subsequent leaf node corresponds to a problem restriction P_k and nogood set $\mathcal{N}_k$, where P_k contains the constraints of P and the branching constraints that define the leaf node. Since the search avoids branches that violate previous nogoods, the fixed variables at the leaf node partially define a feasible solution for $\mathcal{N}_k$. A nogood is generated as before. If the search tree is complete at this point, the accumulated nogoods are jointly unsatisfiable. This means that the next nogood set $\mathcal{N}_{k+1}$ will be infeasible and the search will terminate. Otherwise the search continues to the next leaf node.

As remarked earlier, an appropriate selection function can result in nogoods that make subsequent nogood sets easy to solve. In constraint-directed branching, the branching mechanism defines the selection function in such a way that future branching can easily avoid violating the nogoods. This idea is best explained by example, presented in the next section.

2.3.2 Example: Propositional Satisfiability

Propositional satisfiability is one of the fundamental problems of combinatorial optimization, partly because a wide range of combinatorial problems can be formulated in the language of propositional logic. The currently fastest algorithms for propositional satisfiability use a form of the Davis-Putnam-Loveland (DPL) algorithm with clause learning. This algorithm can be interpreted as constraint-directed branching, and a small example will illustrate the basic idea. The example also prepares the ground for the discussion of partial-order dynamic backtracking in the next section.

The example is artificial but is contrived to show how nogoods in the form of conflict clauses help solve a satisfiability problem. Let's suppose I must hire workers to complete a job and have workers 1, ..., 6 to choose from. Workers 3 and 4 are temporary workers. I am constrained

by the following conditions:

(a) I must hire at least one of the workers 1, 5 and 6.
(b) I cannot hire 6 unless I hire 1 or 5.
(c) I cannot hire 5 unless I hire 2 or 6.
(d) If I hire 5 and 6, I definitely must hire 2.
(e) If I hire 1 or 2, then I must hire at least one temporary worker.
(f) I can hire neither 1 nor 2 if I hire any temporary workers.

I wish to know whether it is possible to satisfy these conditions simultaneously.

Formulating the Problem

Let x_j have the value T (for *true*), if I hire worker j, and F (for *false*) otherwise. Thus, x_j can be viewed as a proposition that asserts that I am hiring worker j. Conditions (a)–(f) can be written in logical form by using some standard notation ($\vee$ for *or*, $\wedge$ for *and*, $\neg$ for *not*, and $\rightarrow$ for *implies*).

$$\begin{array}{ll} x_1 \vee x_5 \vee x_6 & (a) \\ x_6 \rightarrow (x_1 \vee x_5) & (b) \\ x_5 \rightarrow (x_2 \vee x_6) & (c) \\ (x_5 \wedge x_6) \rightarrow x_2 & (d) \\ (x_1 \vee x_2) \rightarrow (x_3 \vee x_4) & (e) \\ (x_3 \vee x_4) \rightarrow (\neg x_1 \wedge \neg x_2) & (f) \end{array} \tag{2.37}$$

The conjunction of these formulas is a proposition Q that must be true if I am to meet the conditions.

A logical proposition is *satisfiable* if some assignment of truth values to its variables makes it true. To check the satisfiability of Q, it is convenient to write Q in *conjunctive normal form* (CNF), which is to say as a conjunction of logical clauses. A *clause* is a disjunction of variables or their negations, such as $\neg x_1 \vee x_2$. The disjuncts $\neg x_1$, x_2 are known as *literals*. An implication $x_1 \rightarrow x_2$ can be written $\neg x_1 \vee x_2$ because it is interpreted as a material conditional—that is, it states that either x_2 is true or x_1 is false.

Thus, formula (b) in (2.37) can be written $x_1 \vee x_5 \vee \neg x_6$, and similarly for (c). Formula (d) is equivalent to $\neg(x_5 \wedge x_6) \vee x_2$, which can be written $\neg x_5 \vee \neg x_6 \vee x_2$. Formula (e) is equivalent to the conjunction of two conditionals, $x_1 \rightarrow (x_3 \vee x_4)$ and $x_2 \rightarrow (x_3 \vee x_4)$. Formula (f) is

equivalent to four conditionals:

$$\begin{aligned} x_3 &\rightarrow \neg x_1 \\ x_3 &\rightarrow \neg x_2 \\ x_4 &\rightarrow \neg x_1 \\ x_4 &\rightarrow \neg x_2 \end{aligned}$$

Proposition Q can be put in CNF by writing it as the conjunction of the clauses obtained from the propositions in (2.37).

The satisfiability problem for Q now can be stated

$$\text{Logic:} \left\{ \begin{array}{llllllll} x_1 & & & & \vee & x_5 \vee & x_6 & (a) \\ x_1 & & & & \vee & x_5 \vee \neg x_6 & & (b) \\ & x_2 & & & \vee \neg x_5 \vee & & x_6 & (c) \\ & x_2 & & & \vee \neg x_5 \vee & \neg x_6 & & (d) \\ \neg x_1 & \vee & x_3 \vee & x_4 & & & & (e1) \\ & \neg x_2 \vee & x_3 \vee & x_4 & & & & (e2) \\ \neg x_1 & \vee \neg x_3 & & & & & & (f1) \\ \neg x_1 & & \vee \neg x_4 & & & & & (f2) \\ & \neg x_2 \vee \neg x_3 & & & & & & (f3) \\ & \neg x_2 & \vee \neg x_4 & & & & & (f4) \end{array} \right. \qquad (2.38)$$

Domains: $x_j \in \{T, F\},\ j = 1, \ldots, 6$

where *logic* indicates that the constraint is a conjunction of the clauses listed.

Inference: Unit Clause Rule

A simple form of propagation for logical clauses is the *unit clause rule*. The rule says that when all but one of the literals in a clause have been determined to be false, the remaining literal must be true. In other words, the clause has been reduced to a *unit clause* (a clause containing one literal) by fixing all other literals to false. For instance, if $(x_1, x_2) = (T, F)$, then the third literal in $\neg x_1 \vee x_2 \vee \neg x_3$ must be true, which fixes $x_3 = F$. Thus, the domain of x_3 is reduced to $\{F\}$.

The unit clause rule is applied repeatedly until the domains can be reduced no further. This process need not detect all variables that can be fixed, as in the case of clauses $x_1 \vee x_2$, $x_1 \vee \neg x_2$. Variable x_1 must be true, but the unit clause rule fixes no variables.

Search

Figure 2.9 depicts a branching tree for problem (2.38). The tree will first be described as a branching search and subsequently reinterpreted as a constraint-directed search.

The branching order is $x_1, \ldots, x_6$, and the left branch (F) is taken first. The unit clause rule is applied at each node. The resulting search scheme is essentially the DPL method with conflict clauses.

At Node 5, the unit clause rule uses clauses (a) and (b) to remove both T and F from x_6's domain, resulting in an empty domain and infeasibility. Since the branching assignments $x_1 = F$ and $x_5 = F$ are enough to result in this empty domain, either x_1 or x_5 must be true in any feasible solution. So, a nogood $(x_1, x_5) \neq (F, F)$ is created. It can be written as the *conflict clause* $x_1 \vee x_5$.

Due to the empty domain at Node 5, the search backtracks to Node 4 and takes the branch to Node 6, where conflict clause $x_2 \vee \neg x_5$ is generated. At this point, the search backtracks again to Node 4 and notes that the entire subtree of Node 4 is infeasible. Since setting $(x_1, x_2) = (F, F)$ is enough to create this infeasibility, a second conflict clause $x_1 \vee x_2$ is associated with Node 6. The search must now backtrack all the way to

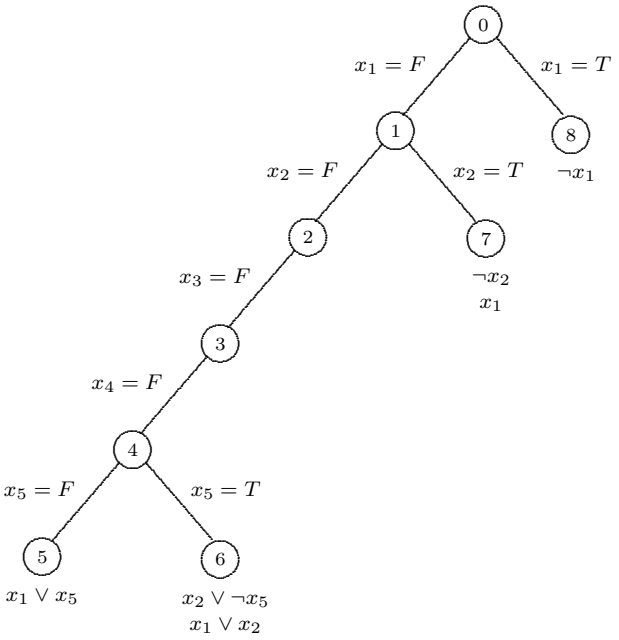

Figure 2.9. Branching tree for a propositional satisfiability problem. Conflict clauses are shown below the nodes with which they are associated.

Node 1 in order to find a right branch that is consistent with this clause. As a result, a large portion of the search tree is bypassed. This sort of move is sometimes called *backjumping*.

At Node 7, the unit clause rule uses (f3), (f4) and (e2) to derive a contradiction. The conflict clause $\neg x_2$ is created, and the search backtracks to Node 1, where a second conflict clause x_1 is created. Finally, at Node 8, the unit clause rules uses (f1), (f2) and (e1) to derive a contradiction. This creates the conflict clause $\neg x_1$, and the search terminates without finding a feasible solution. It must be concluded that conditions (a)-(f) cannot be simultaneously satisfied.

When the search is interpreted as a constraint-directed search, the leaf nodes correspond to nogood sets $\mathcal{N}_k$ as shown in Table 2.9. $\mathcal{N}_k$ is solved by assigning variables the values they receive in the branching tree along the path from the root to the leaf node. (Assignments to any remaining variables can be made arbitrarily.) Thus, the branching pattern defines the selection function. The problem restriction P_k consists of the original clauses and the branching constraints at the leaf node. The nogoods are the conflict clauses generated at each leaf node in the branching search.

The initial nogood set $\mathcal{N}_0$ is empty. It is solved by setting $x_1, \ldots, x_5$ to F, as in the branching tree, and x_6 to either value. P_1 contains the original clauses and the constraint $(x_1, \ldots, x_5) = (F, F, F, F, F)$. P_1 is infeasible because these assignments already falsify a clause. The conflict clause $x_1 \vee x_5$ mentioned earlier is generated as a nogood. A solution for $\mathcal{N}_1$ is similarly obtained, and the nogood $x_2 \vee \neg x_5$ generated.

The third nogood $x_1 \vee x_2$ is generated in the branching scheme by backtracking to Node 4 and noting that the subtree at that node is infeasible. In constraint-directed search, however, $x_1 \vee x_2$ is generated

Table 2.9. Interpretation of the Davis-Putnam-Loveland procedure with conflict clauses as constraint-directed branching. Each problem restriction P_k consists of the original problem and all previously generated nogoods. The symbol $*$ indicates a value that can be either T or F.

k	*Node*	*Relaxation* R_k	*Solution* x^k *of* R_k	*Nogoods generated*
0	5		$(F, F, F, F, F, *)$	$x_1 \vee x_5$
1	6	$x_1 \vee x_5$	$(F, F, F, F, T, *)$	$x_2 \vee \neg x_5$
2	7	$x_1 \vee x_2$	$(F, T, *, *, *, *)$	$\neg x_2$
3	8	x_1	$(T, *, *, *, *, *)$	$\neg x_1$
4	–	$\emptyset$	R_4 is infeasible	

by *processing* the nogoods; that is, by inferring it from the nogoods already generated. It is not hard to see that $x_1 \vee x_5$ and $x_2 \vee \neg x_5$ imply $x_1 \vee x_2$. Formally, this can be obtained by *resolving* $x_1 \vee x_5$ and $x_2 \vee \neg x_5$. In general, two clauses can be resolved when exactly one variable (in this case, x_5) switches sign between them. The resolvent is obtained by removing the variable that switches sign and retaining all the other literals. Once the nogood $x_1 \vee x_2$ is obtained by resolution, the two nogoods $x_1 \vee x_5$ and $x_2 \vee \neg x_5$ become redundant, in a sense that will be clarified in the next section, and are dropped.

In similar fashion, the nogood set $\mathcal{N}_3$ is obtained by resolving $x_1 \vee x_2$ and $\neg x_2$ to obtain x_1, and dropping $x_1 \vee x_2$. Nogood set $\mathcal{N}_4$ consists of the resolvent of x_1 and $\neg x_1$, which is the necessarily false empty clause $\emptyset$. Since $\mathcal{N}_4$ is infeasible, the search terminates at this point without having found a solution.

A key property of constraint-directed branching is that the selection function and nogood processing work together to make the nogood sets easy to solve. The selection function simply mimics the branching process. It assigns each variable its current value in the branching tree. When the nogoods are processed as described above, this assignment always solves $\mathcal{N}_k$ (unless it is infeasible). This idea is generalized in partial-order dynamic backtracking, to be discussed next.

2.3.3 Partial-Order Dynamic Backtracking

A slight change in the selection function and nogood processing converts constraint-directed branching to a more general search procedure: partial-order dynamic backtracking.

In the chronological backtracking algorithm of the previous section, the order in which variables are instantiated and uninstantiated is fixed by the branching order. In partial-order dynamic backtracking, the order of instantiation can change dynamically in the course of the algorithm. In fact, the instantiation order is determined only in part by a constantly changing partial-order on the variables, and in part by choices made dynamically by the user. The algorithm, therefore, allows more freedom in the search than branching does, without sacrificing completeness.

When a nogood (conflict clause) is generated, the user selects one of the variables in the clause to be *last*. The remaining variables in the clause are *penultimate*. The last variables in the current nogood set define a partial order in which each penultimate variable in a nogood precedes the last variable in that nogood. The last variable in a new nogood must be selected so as to be consistent with the partial order defined by the existing nogoods. In constraint-directed branching, the last variable is always the one on which the search last branched.

When conflict clauses are resolved, the eliminated variable must be the last variable in both clauses (*parallel* resolution). Note that the resolvents generated in constraint-directed branching are always parallel resolvents. A conflict clause is redundant and can be eliminated when one of its penultimate literals is last in another clause.

Since there is no fully defined branching order, the selection function cannot mimic the branching order as in constraint-directed branching. Rather, the solution of the current relaxation must *conform to* the nogoods in the current nogood set. This means that whenever a variable occurs penultimately in a nogood, it must be assigned a value opposite to the sign in which it occurs. Thus, a penultimate variable is set to false if it occurs positively, and to true if it is negated. This criterion is well defined, because it can be shown that a variable will have the same sign in all its penultimate occurrences. If a variable does not occur penultimately in any nogood, it can be set to any value that violates none of the nogoods.

The nogood set can always be solved in this fashion without backtracking, unless of course the nogoods are infeasible, in which case the search is over. Note that the selection function used in constraint-directed branching observes this same rule of conformity.

A partial-order dynamic backtracking algorithm for feasibility problems is stated in Figure 2.10. In this algorithm, $\leq_{\mathcal{N}}$ denotes the partial order that is defined by the penultimate and last variables in each clause.

Let $\mathcal{N} = \emptyset$.
For a given nogood N let x_N be the last variable in N.
For a given nogood set $\mathcal{N}$ let the partial order $\leq_{\mathcal{N}}$ be the transitive closure of the relations $x_j \leq_{\mathcal{N}} x_N$ for all penultimate variables x_j in N and every $N \in \mathcal{N}$.

While $\mathcal{N}$ is feasible repeat:
  Select a feasible solution x of $\mathcal{N}$ that conforms to the nogoods in $\mathcal{N}$.
  If x is feasible in P then stop.
  Else
    Define a nogood N that excludes x and possibly other solutions that are infeasible in P.
    Select a variable x_N in N to be last in N, so that $x_N \not\leq_{\mathcal{N}} x_j$ for all x_j that are penultimate in N.
    Add N to $\mathcal{N}$ and process $\mathcal{N}$ with parallel resolution.
P is infeasible.

Figure 2.10. Partial-order dynamic backtracking algorithm for a feasibility problem P with boolean variables.

2.3.4 Example: Propositional Satisfiability

The satisfiability instance (3.47) solved earlier is convenient for illustrating partial-order dynamic backtracking. Table 2.10 summarizes the procedure. Initially, the conformity principle imposes no restriction since the nogood set is empty. For purposes of this illustration, the nogood sets are solved by setting variables unaffected by the conformity principle to false, if possible, or to true, if necessary, to avoid violating a nogood. The unit clause rule is applied after each setting.

For $k = 0$ in Table 2.10, the variables are assigned values in the order in which they are indexed, but any order would be acceptable. When x_5 is reached, a clause in the original constraint set is violated, and the nogood $x_1 \vee x_5$ is generated. The variable x_1 is arbitrarily selected as last, as is indicated by writing the nogood as $x_5 \vee x_1$.

At this point, x_5 occurs positively as a penultimate variable, and it must therefore be set to false in the solution of $\mathcal{N}_1$. Variable x_1 is arbitrarily assigned next, and it must be assigned true to avoid violating the nogood $x_5 \vee x_1$. At this point a clause in the original constraint set is already violated. The restriction P_2, which contains the original clauses and $(x_1, x_5) = (T, F)$, is infeasible, and the nogood $x_5 \vee \neg x_1$ is generated (one could generate the stronger nogood $\neg x_1$, since it alone creates an infeasibility). Variable x_1 must be selected as last, since it occurs after x_5 in the partial order defined by the one existing nogood. Now the two nogoods $x_5 \vee x_1$ and $x_5 \vee \neg x_1$ can be parallel-resolved, resulting in the new nogood x_5, whose only variable is necessarily chosen as last. The other two nogoods are now redundant and are dropped, since the last literal of the clause x_5 occurs penultimately in both.

Table 2.10. Partial-order dynamic backtracking solution of a propositional satisfiability problem. The symbol $*$ indicates a value that can be either T or F. The "last" variable in each nogood is written last.

k	*Nogood set* $\mathcal{N}_k$	*Solution* x^k *of* $\mathcal{N}_k$	*Nogoods generated*
0		$(F, F, F, F, F, *)$	$x_5 \vee x_1$
1	$x_5 \vee x_1$	$(T, *, *, *, F, *)$	$x_5 \vee \neg x_1$
2	x_5	$(*, F, *, *, T, *)$	$\neg x_5 \vee x_2$
3	$\begin{cases} x_5 \\ \neg x_5 \vee x_2 \end{cases}$	$(*, T, *, *, T, *)$	$\neg x_2$
4	$\emptyset$	$\mathcal{N}_4$ is infeasible	

Nogood set $\mathcal{N}_2$ can be solved without regard to conformity, since no variables occur penultimately in it. Variable x_5 must be set to true, and variable x_2 is arbitrarily set to false next. This already violates a constraint and yields the nogood $\neg x_5 \vee x_2$, in which x_2 is arbitrarily chosen to be last. The current nogoods x_5 and $\neg x_5 \vee x_2$ have the resolvent x_2, but they do not have a parallel resolvent since x_5 does not occur last in both clauses. Both clauses are therefore retained in $\mathcal{N}_3$. When nogood x_2 is generated by the solution of $\mathcal{N}_3$, two steps of parallel resolution yield the empty clause, and the search terminates without finding a feasible solution.

2.3.5 Relaxation in Constraint-Directed Search

Constraint-directed search has so far been characterized as an enumeration of problem restrictions, but relaxations play a central role as well. In fact the nogood set $\mathcal{N}_k$ can be reinterpreted as a problem relaxation, and nogoods R_k if the nogoods are written to impose bounds on the optimal value, rather than excluding unwanted solutions. The bounds, in effect, state that the excluded solutions cannot yield an optimal value better than the current solution. So, to achieve a better value, one must choose a solution that is allowed by the nogoods.

Recall that in step k of the search, a feasible solution x^k of the nogood set $\mathcal{N}_k$ is found, and the next restriction $P_{k+1} = P(x^k)$ is solved for its optimal value $v(x^k)$. Suppose that the resulting nogood N_{k+1} excludes the solutions in set T, so that $v(x) \geq v(x^k)$ for all $x \in T$. If v represents the optimal value of the original problem, the nogood can be written as a *nogood bound*

$$v \geq B_{k+1}(x)$$

where

$$B_{k+1}(x) = \begin{cases} v(x^k) & \text{if } x \in T \\ -\infty & \text{otherwise} \end{cases}$$

The nogood bound says that any $x \in T$ must result in a restriction $P(x)$ with optimal value at least $v(x^k)$. Thus, in order to do better than $v(x^k)$, one must avoid solutions in T.

A pair (v, x) satisfies the nogood bounds accumulated so far if and only if it is feasible in the following optimization problem R_k:

$$\begin{aligned} &\min\ v \\ &v \geq B_i(x), \quad i = 1, \ldots, k \\ &x \in D \end{aligned} \tag{2.39}$$

It is straightforward to show that (2.39) is a relaxation of P (and therefore each P_i) in the sense defined earlier. In fact, $B_i(x)$ need not go to

$-\infty$ for $x \notin T$. It is enough that $B_i(x)$ provide a valid lower bound on the optimal value of $P(x)$.

THEOREM 2.1 *Let $v(x)$ be the optimal value of restriction $P(x)$ for all $x \in D$. Then (2.39) is a relaxation of (2.1) if $B_i(x) \leq v(x)$ for all $x \in D$ and $i = 1, \ldots, k$.*

In some constraint-directed methods, such as Benders decomposition, the bound $B_i(x)$ remains fairly tight when x is in the vicinity of solutions in T.

Rather than find a solution x^k that satisfies the nogood set $\mathcal{N}_k$, the search algorithm now finds a solution (v_k, x^k) of R_k for which v_k is less than the value of the best candidate solution found so far. That is,

$$v_k < \min\left\{v(x^1), \ldots, v(x^{k-1})\right\}$$

It is therefore not necessary to solve R_k to optimality—only to find a sufficiently good solution. If no such solution exists, the search terminates. A generic algorithm appears in Figure 2.11.

Solving R_k to optimality can be useful, however, because its optimal value provides a lower bound on the optimal value of the original problem. The best candidate solution provides an upper bound, which means that one can halt the algorithm at any point and bracket the optimal value between two bounds. The search terminates when the two bounds converge.

2.3.6 Logic-Based Benders Decomposition

Benders decomposition is a constraint-directed search in which all the problem restrictions are defined by fixing the same subset of variables.

Let $v_{\text{UB}} = \infty$. Initially the relaxation R minimizes v subject to $x \in D$.
Associate a restriction $P(x)$ of P with each $x \in D$, and let $v(x)$ be the optimal value of $P(x)$.

While R has a feasible solution $(v, \bar{x})$ with $v < v_{\text{UB}}$ repeat:
  Compute $v(\bar{x})$ by solving $P(\bar{x})$.
  Let $v_{UB} = \min\{v(\bar{x}), v_{UB}\}$.
  Add to R a nogood bound $v \geq B(x)$ for which $B(\bar{x}) = v(\bar{x})$ and $B(x) \leq v(x)$ for all $x \in D$.
The optimal value of P is v_{UB}.

Figure 2.11. Generic constraint-directed search algorithm for solving a minimization problem P with variable domain D.

That is, when a solution x^k of the current relaxation is found, the next restriction is defined by fixing the variables $(x_1, \ldots, x_p)$ to the solution values $(x_1^k, \ldots, x_p^k)$ in each iteration, where the same variables $x_1, \ldots, x_p$ are fixed in each iteration. They might be called the *search variables*, because the search procedure in effect enumerates values of these variables. In a Benders context, the nogood bounds are known as *Benders cuts*, the relaxation R_k as the *master problem*, and the restriction $P(x^k)$ as the *subproblem* or *slave problem*.

The primary rationale for Benders decomposition is that the problem may have special structure that allows it to simplify considerably when certain variables are fixed to any value. The choice of search variables is therefore crucial to the success of the method, as is the ability to formulate strong Benders cuts.

Classical Benders decomposition is defined for the case in which the subproblem is a continuous linear or nonlinear programming problem. The Benders cuts are obtained from Lagrange multipliers associated with the constraints of the subproblem. The root idea of the Benders method, however, can be generalized to great advantage. The Benders cuts can be obtained from a logical analysis of the subproblem, resulting in a *logic-based* Benders method. This, in principle, allows the subproblem to take any form, but a separate analysis must be conducted for each class of subproblems.

To prepare a problem for solution by a Benders method, the variables are decomposed into a vector x of search variables and a vector y of subproblem variables. The problem P can now be written

$$\begin{aligned} &\min\ f(x,y) \\ &\mathcal{S}(x,y) \\ &x \in D_x,\ y \in D_y \end{aligned}$$

where $\mathcal{S}(x,y)$ is a constraint set that contains variables x, y.

In step k of the algorithm, an optimal solution x^k is computed for relaxation R_k. The problem restriction (subproblem) $P(x^k)$ is obtained by fixing x to x^k in P. $P(x^k)$ is therefore

$$\begin{aligned} &\min\ f(x^k,y) \\ &\mathcal{S}(x^k,y) \\ &y \in D_y \end{aligned}$$

where $\mathcal{S}(x^k,y)$ is the constraint set that remains when x is fixed to x^k in $\mathcal{S}(x,y)$. The resulting Benders cut $v \geq B_{k+1}(x)$ involves only the variables x_j, since only they are relevant to defining $P(x^k)$. The cut is added to R_k to obtain the next master problem.

Thus the master problem R_k at step k is

$$\begin{aligned} &\min\ v \\ &v \geq B_i(x), \quad i = 1, \ldots, k \\ &x \in D_x \end{aligned}$$

The algorithm appears in Figure 2.12. In practice, the first relaxation R_0 may be augmented with precomputed nogoods for a "warm start." R_0 may also contain constraints from problem P that involve only the search variables, as well as other constraints that involve only x and are valid for P.

2.3.7 Example: Machine Scheduling

A simple machine scheduling problem illustrates logic-based Benders decomposition. It is a small representative of an important class of planning and scheduling problems that frequently occur in manufacturing and supply chain management. The goal in these problems is to assign tasks to facilities and then schedule the tasks. The facilities might be factories, machines in a factory, transport modes, delivery vehicles, or computer processors.

In the problem instance at hand, five jobs are to be allocated to two machines, named A and B, and scheduled on them. Each job j has a release time r_j and a deadline d_j. The time required to process job j on machine i is p_{ij}. The specific problem data appear in Table 2.11. Note that machine A is faster than machine B. The objective is to minimize makespan; that is, to minimize the finish time of the last job to finish.

Let $v_{\text{LB}} = -\infty$ and $v_{\text{UB}} = \infty$.
Initially the relaxation R minimizes v subject to $x \in D_x$
  and possibly other valid constraints involving x.
Let $\bar{x}$ be a feasible solution of R.

While $v_{\text{LB}} < v_{\text{UB}}$ repeat:
  Let $v(\bar{x})$ be the minimum value of $f(\bar{x}, y)$ subject to $\mathcal{S}(\bar{x}, y)$ and $y \in D_y$.
  Let $v_{UB} = \min\{v(\bar{x}), v_{UB}\}$.
  Add to R a Benders cut $v \geq B(x)$.
  Compute the optimal value v_{LB} of R.
The optimal value of P is v_{UB}.

Figure 2.12. Generic logic-based Benders algorithm for minimizing $f(x, y)$ subject to $\mathcal{S}(x, y)$ and $(x, y) \in D_x \times D_y$.

Table 2.11. Data for a machine scheduling problem.

Job j	*Release time* r_j	*Dead-line* d_j	*Processing time* p_{Aj}	p_{Bj}
1	0	10	1	5
2	0	10	3	6
3	2	7	3	7
4	2	10	4	6
5	4	7	2	5

Formulating the Problem

It is convenient to use a metaconstraint *disjunctive* to represent the scheduling portion of the problem. The constraint may, in general, be written

$$\text{disjunctive}(s \mid p)$$

where $s = (s_1, \ldots, s_n)$ are the start times of the jobs to be scheduled, and $p = (p_1, \ldots, p_n)$ are the processing times. The constraint is satisfied when the jobs do not overlap. That is,

$$s_j + p_j \leq s_k \text{ or } s_k + p_k \leq s_j, \text{ all jobs } j, k \text{ with } j \neq k$$

The name *disjunctive* derives from the fact that the task of scheduling jobs sequentially is commonly known as disjunctive scheduling, as opposed to cumulative scheduling, in which several jobs can run simultaneously subject to resource constraints.

Only two types of decision variables are needed to formulate the problem—the start time s_j already mentioned, and the machine x_j to which job j is assigned.

If there are n jobs and m machines, the formulation is

$$\text{Linear: } \begin{cases} \min\ M \\ M \geq s_j + p_{x_j j}, \text{ all } j \\ s_j + p_{x_j j} \leq d_j, \text{ all } j \end{cases}$$

$$\text{Disjunctive: } ((s_j \mid x_j = i) \mid (p_{ij} \mid x_j = i)), \text{ all } i$$

$$\text{Domains: } s_j \in [r_j, \infty),\ x_j \in \{1, \ldots, m\}, \text{ all } j$$

In the objective function, M represents the makespan. The linear constraints define the makespan and enforce the deadlines. The release times are observed in the domain constraints. A disjunctive scheduling

constraint is imposed for each machine. In the disjunctive constraints, the notation $(s_j \mid x_j = i)$ denotes the tuple of start times s_j for jobs assigned to machine i, and similarly for the processing times.

A natural decomposition for this problem distinguishes the assignment portion from the scheduling portion. One can search over various assignments of jobs to machines and, for each, try to find a feasible schedule for the jobs assigned to each machine. The assignment variables x_j are therefore the search variables, and each subproblem P_k is a scheduling problem that decouples into separate scheduling problems for the individual machines.

Relaxation: The Master Problem

The master problem (relaxation) R_k minimizes makespan v subject to the Benders cuts generated so far. It can be solved by whatever method is most suitable for its structure. One option is to solve it as an MILP problem, since it is naturally expressed in this form. For this purpose, the variables x_i can be replaced with 0-1 variables x_{ij}, where $x_{ij} = 1$ when job i is assigned to machine j.

The master problem can be strengthened by adding valid constraints. One can observe, for example, that the jobs assigned to a machine must fit within the earliest release time and latest deadline of those jobs. In fact, this is true of any subset of the jobs assigned to a given machine. To formulate this condition, let $J(t_1, t_2)$ be the set of jobs whose time windows lie in the interval $[t_1, t_2]$. So $J(t_1, t_2) = \{j \mid [r_j, d_j] \subset [t_1, t_2]\}$. The total processing times of the jobs in $J_i(t_1, t_2)$ that are assigned to a given machine i must not exceed $t_2 - t_1$:

$$\sum_{j \in J(t_1, t_2)} p_{ij} x_{ij} \leq t_2 - t_1 \tag{2.40}$$

It suffices to consider release times for t_1 and deadlines for t_2. In the problem instance at hand,

$$\begin{array}{ll} J(0,7) = \{3,5\} & J(2,10) = \{3,4,5\} \\ J(0,10) = \{1,2,3,4,5\} & J(4,7) = \{5\} \\ J(2,7) = \{3,5\} & J(4,10) = \{5\} \end{array}$$

Some of these sets give rise to vacuous or redundant inequalities (2.40). For instance, the inequality for $J(0,7)$ is $p_{i3}x_{i3} + p_{i5}x_{i5} \leq 7$, which is $3x_{A3} + 2x_{A5} \leq 7$ for $i = A$ and $7x_{B3} + 5x_{B5} \leq 7$ for $i = B$. The former is obviously redundant since $x_{ij} \in \{0, 1\}$. The latter is dominated by another inequality for Machine B (namely, $7x_{B3} + 5x_{B5} \leq 5$). The

nonredundant inequalities for Machine A are

$$\begin{aligned} J(0,10): &\quad \sum_{j\in\{1,2,3,4,5\}} p_{Aj}x_{Aj} \le 10 \\ J(2,10): &\quad \sum_{j\in\{3,4,5\}} p_{Aj}x_{Aj} \le 8 \end{aligned} \tag{2.41}$$

and those for Machine B are

$$\begin{aligned} J(0,10): &\quad \sum_{j\in\{1,2,3,4,5\}} p_{Bj}x_{Bj} \le 10 \\ J(2,7): &\quad \sum_{j\in\{3,5\}} p_{Bj}x_{Bj} \le 5 \\ J(2,10): &\quad \sum_{j\in\{3,4,5\}} p_{Bj}x_{Bj} \le 8 \\ J(4,7) &\quad \sum_{j\in\{5\}} p_{Bj}x_{Bj} \le 3 \end{aligned} \tag{2.42}$$

Section 4.16.3 shows how to identify nonredundant inequalities of this sort in a systematic way.

Further inequalities can be added to the master problem to constrain the makespan v:

$$\begin{aligned} v &\ge \sum_{j\in\{1,2,3,4,5\}} p_{ij}x_{ij}, \quad i = A, B \\ v &\ge 2 + \sum_{j\in\{3,4,5\}} p_{ij}x_{ij}, \quad i = A, B \\ v &\ge 4 + \sum_{j\in\{5\}} p_{ij}x_{ij}, \quad i = A, B \end{aligned} \tag{2.43}$$

The three sets of inequalities correspond to the release times 0, 2, and 4. The first includes the jobs in $J(0,\infty)$, the second the jobs in $J(2,\infty)$, and the third the jobs in $J(4,\infty)$.

The master problem is now

$$\begin{aligned} &\min\ v \\ &\text{inequalities (2.41)–(2.43)} \\ &\text{Benders cuts} \\ &x_{ij} \in \{0,1\}, \text{ all } i,j \end{aligned}$$

An optimal solution of the initial master problem (without Benders cuts) sets $x_{A1} = x_{A2} = x_{A3} = x_{A5} = x_{B4} = 1$, with all other $x_{ij} = 0$. This

assigns Jobs 1, 2, 3 and 5 to the faster Machine A and Job 4 to Machine B. The objective function value is 9. It will be seen shortly, however, that 9 is not a feasible makespan.

Inference: Benders Cuts

The inference stage consists of inferring Benders cuts from the subproblem that results when the master problem variables are fixed to their current values.

The subproblem separates into an independent scheduling problem on each machine. Thus, if $\bar{x}$ is the solution of the previous master problem, the subproblem on each machine i is

$$\begin{aligned}
&\min\ M_i \\
&s_j + p_{ij} \le d_j, \ \text{ all } j \text{ with } \bar{x}_{ij} = 1 \\
&\text{disjunctive}((s_j \mid \bar{x}_{ij} = 1) \mid (p_{ij} \mid \bar{x}_{ij} = 1)) \\
&s_j \in [r_j, \infty), \text{ all } j
\end{aligned}$$

If M_i^* is the optimal makespan on machine i for each i, then the optimal makespan overall is $\max_i\{M_i^*\}$.

The subproblem does not separate in this way if there are precedence constraints between jobs, because the time at which a job can be scheduled may depend on the times at which jobs are scheduled on other machines. Yet even when there are precedence constraints, separability can be preserved if they involve only jobs that must be scheduled on the same machine. Thus, if jobs j and k must be scheduled on the same machine, and j must precede k, one can add constraint $x_j = x_k$ to the master problem and the constraint $s_j + d_{ij} \le s_k$ to the scheduling problem on every machine.

As noted above, the initial solution of the master problem assigns Job 4 to Machine B and the other jobs to Machine A. The minimum makespan schedule on Machine B simply starts Job 4 at its release time, resulting in a makespan of 8. Thus, whenever Job 4 is assigned to Machine B (i.e., whenever $x_{4B} = 1$), the makespan will be at least 8. This gives rise to the Benders cut

$$v \ge 8x_{B4}$$

Any solution that achieves a makespan better than 8 must avoid assigning Job 4 to Machine B.

The minimum makespan schedule for Jobs 1, 2, 3 and 5 on Machine A appears in Figure 2.13. The resulting makespan is 10, which produces the Benders cut

$$v \ge 10(x_{A1} + x_{A2} + x_{A3} + x_{A5} - 3) \tag{2.44}$$

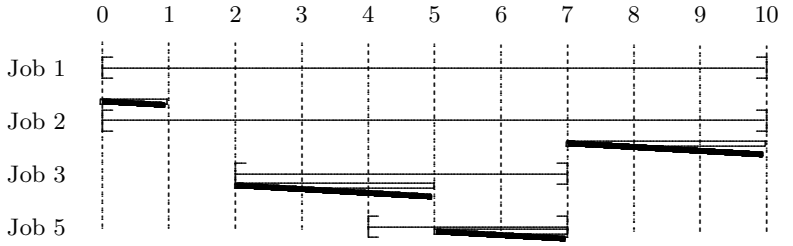

Figure 2.13. A minimum makespan schedule on Machine A for Jobs 1, 2, 3 and 5 of Table 2.11. The horizontal lines represent time windows.

Thus, any solution that obtains a makespan better than 10 must avoid assigning at least one of these jobs to machine A.

The master problem is now

$$\begin{aligned}
&\min\ v\\
&\text{inequalities (2.41)–(2.43)}\\
&v \geq 10\\
&v \geq 10(x_{A1} + x_{A2} + x_{A3} + x_{A5} - 3)\\
&x_{ij} \in \{0,1\},\ \text{ all } i,j
\end{aligned}$$

Solution of this problem results in the same machine assignments as before, but the optimal value is now 10. Since this is a lower bound on the minimum makespan, and a feasible solution with makespan 10 was found by solving the subproblem, the algorithm terminates. The schedule found by solving the subproblem is optimal.

In practice, the success of a Benders method often rests on finding strong Benders cuts that rule out as many infeasible solutions as possible. One way to deduce stronger cuts is to examine more closely the reasoning process that proves infeasibility in the subproblem. In the present case, the initial infeasibility on Machine A can be proved using a well-known *edge-finding* technique, and it provides the basis for a stronger cut.

Edge finding is a procedure for identifying jobs that must precede, or that must follow, a set J of other jobs. The basic idea is that if there is too little time to complete both job k and the jobs in J when k starts no earlier than the jobs in J start, then k must finish before the jobs in J start. To state this more precisely, let E_j and L_j be the earliest and latest possible start times for job j, which is to say that the current domain of start time t_j is the interval $[E_j, L_j - p_{ij}]$. If

$$\max_{j \in J \cup \{k\}} \{L_j\} - \min_{j \in J} \{E_j\} < \sum_{j \in J \cup \{k\}} p_{ij} \tag{2.45}$$

then job k must finish before any job in J starts. A similar rule can be used to establish that job k cannot start until all the jobs in J have finished. This process is called edge finding because, by establishing precedence relations between the jobs, it adds edges to a directed graph that represents these relations.

Edge finding may permit one to reduce some of the domains. Initially, the domain of t_j is the interval $[r_j, d_j - p_{ij}]$, but as edge finding proceeds the current domain $[E_j, L_j - p_{ij}]$ may become a smaller interval. If a domain is eventually reduced to the empty set, then there is no feasible schedule.

In particular, if edge finding determines that job k must precede the jobs in J, then for any subset J' of J, k must finish by $L'_k = \max_{j \in J'}\{L_j\} - \sum_{j \in J'} p_{ij}$. This means the latest finish time L_k can be tightened to L'_k if this number is smaller than E_j. Similar reasoning can be used when job k must follow the jobs in J.

The key to forming a strong Benders cut is to keep track of which jobs are actually involved in the domain reduction operations that lead to an empty domain. These jobs alone are sufficient to create infeasibility. A Benders cut that involves fewer jobs is a stronger cut.

To keep track of the relevant jobs, one can associate with each latest finish time L_j the set J_j^L of jobs that helped to tighten that bound, and a similar set J_j^E for an earliest start time. Initially, $J_j^L = J_j^E = \{j\}$. If edge finding determines that job k precedes the jobs in J and tightens L_k as a result, then the jobs in J_j^L are added to J_k^L. If E_k is tightened, the jobs in J_j^E are added to J_k^E. When the edge finding process is finished, and the domain of some t_j is found to be empty, one can conclude that the jobs in $J_j^L \cup J_j^E$ are sufficient to create the infeasibility.

In the example, edge finding deduces that Jobs 2, 3, and 5 are sufficient to create infeasibility on Machine A, whether or not Job 1 is assigned to the machine. The reasoning can be traced as follows. Since a feasible schedule with makespan 10 exists (Figure 2.13), one can show that 10 is the optimal makespan by proving the infeasibility of a makespan of 9. Imposing a maximum makespan of 9 reduces the deadlines of Jobs 1 and 2 to 9. The initial time windows for jobs 2, 3, and 5 are therefore

$$[E_2, L_2] = [0, 9] \qquad [E_3, L_3] = [2, 7] \qquad [E_5, L_5] = [4, 7]$$

and the domains of t_2, t_3 and t_5 are

$$[E_2, L_2 - p_{A2}] = [0, 6] \qquad [E_3, L_3 - p_{A3}] = [2, 4] \qquad [E_5, L_5 - p_{A5}] = [4, 5]$$

Edge finding deduces that Job 2 must precede Jobs 3 and 5, as can be seen from (2.45):

$$\max\{L_2, L_3, L_5\} - \min\{E_3, E_5\} < p_{A2} + p_{A3} + p_{A5}$$

Figure 2.14 illustrates the situation. At this point the latest finish time L_2 for job 2 can be tightened to $\max\{L_3, L_5\} - (p_{13} + p_{15}) = 2$. Since Jobs 3 and 5 brought this about, they are added to J_2^L, which becomes $\{2, 3, 5\}$. But now the domain of t_2 is the interval $[0, -1]$, which is the empty set. This proves infeasibility, and the jobs involved in the proof are those in $J_2^L \cup J_2^E = \{2, 3, 5\}$. Because Job 1 was not involved in the proof, one can deduce a stronger Benders cut than (2.44):

$$v \geq 10(x_{A2} + x_{A3} + x_{A5} - 2) \tag{2.46}$$

The stronger Benders cut would not have accelerated the solution of this particular problem, but strong cuts can be very helpful in general.

In many cases, edge finding must be combined with other procedures, such as branching, to prove infeasibility. But the above ideas can be extended to such cases. For instance, if edge finding is combined with branching, there is a proof of infeasibility at every leaf node ℓ of the search tree. Along the path from node ℓ to the root there is a series of domain reductions that contribute to this proof, and one can note the set J_ℓ of jobs that contribute to these reductions. Only the jobs in $\bigcup_\ell J_\ell$ need be included in the Benders cut, where the union is taken over all leaf nodes.

One difficulty with a cut of the form (2.46) is that it imposes no bound on v when a proper subset of Jobs 2, 3, and 5 is assigned to Machine A. One should be able to say something about the resulting minimum makespan if only Jobs 2 and 3 are assigned to this machine, for example. In fact, one can often derive a cut that remains useful in such cases. Section 3.13.3 shows how to do this when all the release times are equal.

Actually it is not necessary to re-solve the master problem each time a Benders cut is generated. If the master is solved by branching, one can suspend the branching as soon as a feasible solution is discovered

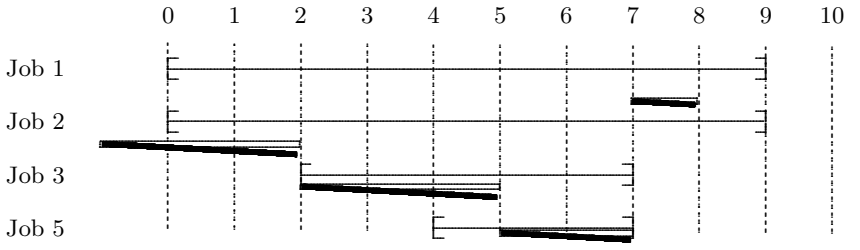

Figure 2.14. Edge-finding proof of infeasibility of scheduling Jobs 1, 2, 3 and 5 on Machine A to achieve a makespan of 9.

and use that solution to define a subproblem. The resulting Benders cut is added to the master problem, and the search continues until another feasible solution is discovered. The search terminates when the search tree is exhaustive and its optimal solution has the same value as the best subproblem solution. This process has been called *branch and check*.

2.3.8 Exercises

1 A group of medications are commonly used to treat a form of cancer, but they can be taken only in certain combinations. A patient who takes Medications 1 and 2 must take Medication 5 as well. Medication 1 can be taken if and only 5 is not taken. At least one of Medications 3, 4, and 5 must be taken. If 5 is taken, then 3 or 4 must be taken. If 4 is taken, then 3 or 5 must be taken. Medication 3 must be taken if both 4 and 5 are taken. Medication 3 cannot be taken without 4, and 5 cannot be taken without 1. Let x_j be true when medication j is taken, and write these conditions in propositional form. Convert them to CNF without adding variables.

2 Find a feasible solution of the CNF expression in Exercise 1 using a DPL algorithm with clause learning. Branch on variables in the order $x_1, \ldots, x_5$, and take the false branch first.

3 Interpret the branching search of Exercise 2 as constraint-directed search by writing a table similar to Table 2.9.

4 Find a feasible solution of the problem in Exercises 1, 2, and 3 by partial-order dynamic backtracking. Experiment with various choices of the last literal in a nogood, and with various heuristics for solving the problem restriction.

5 Find an optimal solution of Exercise 4 using partial-order dynamic backtracking, where the objective is to minimize the number of medications taken. Solve the current nogood set by setting a variable to false whenever possible. When a feasible solution is found, generate a nogood that rules it out, and continue the search. Thus, if the solution $x = (T, F, T, T, F)$ is found, generate the nogood $\neg x_1 \vee x_2 \vee \neg x_3 \vee \neg x_4 \vee x_5$. Continue until the search is exhaustive, and the optimal solution is the best feasible solution found.

6 Prove Theorem 2.1.

7 Interpret the solution of the nogood set in Exercise 5 as the solution of a relaxation. For each step k, write the resulting nogood in the form of an inequality $v \geq B_k(x)$. In the simplest scheme, $B_k(x)$ is

either 0 or ∞ when P_k is infeasible, depending on x. If a feasible solution is found for P_k, $B_k(x)$ is either zero or a finite value (i.e., the number of true variables in the feasible solution).

8 Write (2.40) for each release time t_1 and each deadline t_2 ($t_1 < t_2$) in the problem of Table 2.11. Verify that (2.41) are the nonredundant inequalities.

9 Write the symmetric form of the edge-finding rule (2.45) that is sufficient to establish that job k must start after all the jobs in J finish. Indicate how E_k can be updated if k must follow J.

10 Suppose that in the problem of Table 2.11, Jobs 2, 3, and 4 are assigned to Machine A. Use the edge-finding rules to find jobs that must precede, or follow, subsets of jobs, and update the bounds accordingly. Note that when bounds have been updated, it may be possible to find additional edges. For example, initially one cannot deduce that Job 3 must follow 2 (so that E_3 is not updated), even though one can deduce that Job 2 must precede $\{3, 4\}$ (which updates L_1). However, after L_1 is updated, one can deduce that 3 follows 2 and update E_3. In this case, edge finding identifies all possible bound updates, but this is not true in general.

11 What is the minimum makespan on Machine A in the Exercise 10? What jobs play a role in deriving the minimum? Trace the algorithm that computes J_j^L and J_j^E to verify this.

12 Write a Benders cut that corresponds to the minimum makespan solution of Exercise 11.

13 Exhibit a disjunctive scheduling problem in which edge finding fails to discover all precedence relations.

14 A number of projects must be carried out in a shop, and each project j must start and finish within a time window $[r_j, d_j]$. Once started, the project must run p_j days without interruption. Only one project can be underway at any one time. Every month, the shop is shut down briefly to clean and maintain the equipment, and no project can be in process during this period. The goal is to find a feasible schedule. Formulate this problem and indicate how to solve it with logic-based Benders decomposition. Hint: let each month's schedule be a subproblem. Note that logic-based Benders can provide a scheme for optimizing a master schedule (here, assignment of jobs to months) and daily schedules simultaneously. The monthly resource constraints

take the form of a relaxation of the subproblem within the master problem.

15 A vehicle routing problem requires that a fleet of vehicles make deliveries to customers within specified time windows. Each customer j must take delivery between e_j and d_j, and vehicle i requires time p_{ijk} to travel from customer j to customer k. The objective is to minimize the number of vehicles. Describe a Benders-based solution method in which the master problem assigns customers to vehicles and the subproblem routes each vehicle. The subproblem for each vehicle is a traveling salesman problem with time windows. The subproblem can be written with variable indices and the circuit constraint (see Chapter 5), although it would be desirable to design a metaconstraint specifically for traveling salesman problems with time windows.

2.4 Local Search

Local search methods solve a problem by solving it repeatedly over small subsets of the solution space, each of which is a *neighborhood* of the previous solution. The neighborhood consists of solutions obtained by making small changes in the previous solution, perhaps by changing the value of one variable or swapping the values of two variables.

The motivation for local search is that a neighborhood is more easily searched than the entire solution space. By moving from neighborhood to neighborhood, the search may happen upon a good solution. Well-designed local search methods can, in fact, deliver remarkably good solutions within a reasonable time, although tuning them to work efficiently is more an art than a science. Local search has become indispensable for attacking many practical problems that are too large to solve by exact methods.

In general, the neighborhoods examined during the search cover only a small portion of the solution space. Even the neighborhoods themselves may not be examined exhaustively. Local search therefore provides no guarantee that the solution is optimal or even lies within any given distance from the optimum.

Local search fits naturally into the solution scheme presented here. Because each neighborhood is the feasible set of a problem restriction, local search in effect solves a sequence of problem restrictions. Inference and relaxation can also play a role. In fact, many local search strategies can be viewed as analogs of branching or constraint-directed search, and these analogies suggest how techniques from exhaustive search can

be transferred to heuristic methods. The role of relaxation in branching, for example, can be mirrored in such branching-related local search methods as greedy randomized adaptive search procedures (GRASPs). Inference is already a part of local search methods related to constraint-directed search, such as tabu search, where the tabu list can be viewed as consisting of nogoods. The analogy can be exploited further, because ideas from such techniques as partial-order dynamic backtracking can be imported into tabu search, resulting in a more sophisticated heuristic method.

2.4.1 Some Popular Metaheuristics

Such popular local search schemes or *metaheuristics* as simulated annealing, tabu search, and GRASP algorithms are easily seen to be searches over problem restrictions (Table 2.12). Simulated annealing randomly chooses a solution x' in the neighborhood of the current solution x. If x' is better than x, then x' is *accepted* and becomes the current solution, whereupon the process repeats. If x' is no better than x, x' is nonetheless accepted with a certain probability p. If x' is not accepted, another solution x' is chosen randomly from the neighborhood of x, and the process repeats. The algorithm mimics a cooling process in which molecules seek a minimum energy configuration. The probability p decreases with the *temperature* as the process continues. The search may be terminated at will, and it may be rerun with several different starting points. Clearly the neighborhoods are not examined exhaustively in this

Table 2.12. How some selected heuristic methods fit into the search-infer-and-relax framework.

Solution method	*Restriction* P_k	*Relaxation* R_k	*Selection function* $s(R_k)$	*Inference*
Simulated annealing	Neighborhood of current solution	P_k	Random solution in neighborhood	None
Tabu search	Neighborhood minus tabu list	P_k	Best solution in neighborhood	Addition of nogoods to tabu list
GRASP	Neighborhood of partial solution	Problem specific	Random or greedy selection of solution in neighborhood	None

method. Each restriction is "solved" simply by selecting a solution, or at most a few solutions, randomly from the current neighborhood.

Tabu search differs in that it exhaustively searches each neighborhood. The best solution x' in the neighborhood of the current solution x becomes the current solution. To reduce the probability of cycling repeatedly through the same solutions, a *tabu list* of the last few solutions is maintained. Solutions on the tabu list are excluded from the neighborhood of x (the tabu list can also contain the types of alterations or *moves* performed on the last few solutions to obtain the next solution, rather than the solutions themselves). The items on the tabu list can be viewed as nogoods that rule out solutions or moves that have recently been examined. Tabu search is therefore an inexhaustive form of constraint-directed search.

Each iteration of a GRASP has two phases, the first of which constructs a solution in a greedy fashion, and the second of which uses this solution as a starting point for a local search. The greedy algorithm of the first phase assigns values to one variable at a time until all variables are fixed. The possible values that might be assigned to each variable x_k are ranked according to an easily computable criterion. The algorithm is adaptive in the sense that this ranking depends on what values were assigned to $x_1, \ldots, x_{k-1}$. One of the highly ranked values is then randomly selected as the value of x_k. This random component allows different iterations of the GRASP to construct different starting solutions.

The local search phase can be seen as a search over problem restrictions for reasons already discussed. The greedy phase is likewise a search over problem restrictions in a sense that is reminiscent of a branching search. Recall that a branching search typically branches on a problem P by assigning some variable its possible values. This creates a series of restrictions $P_1, \ldots, P_m$ whose feasible sets partition the feasible set of P. The search may then create restrictions of each P_i by branching on a second variable, and so on recursively.

The greedy algorithm is analogous, except that it generates only one restriction of P rather than an exhaustive list of restrictions $P_1, \ldots, P_m$. Specifically, it creates a restriction P_1 by setting x_1 to a value that is highly ranked. It then restricts P_1 by setting x_2 to a highly ranked value (given the value of x_1), and so forth, until all variables are assigned values.

2.4.2 Local Search Conceived as Branching

Simulated annealing and GRASPs can be seen as special cases of a generic local search procedure that is analogous to branching but does

not explore all possible branches. This interpretation of local search also incorporates relaxation in a natural way.

The generic local search algorithm of Figure 2.15 keeps "branching" until it arrives at a problem that is easy enough to solve, at which point it solves the problem (by searching a neighborhood) and backtracks. When branching on a given problem restriction P, however, the algorithm creates only one branch. The search may backtrack to P later and generate additional branches. The branches eventually created at P differ in two ways, however, from those in a normal branching search: (a) they need not be exhaustive, which is to say the union of their feasible sets need not be the feasible set F of P, and (b) their feasible sets need not partition F.

Local search and GRASPs are special cases of this generic algorithm in which each restriction P is specified by setting one or more variables. If all the variables $x = (x_1, \ldots, x_n)$ are set to values $v = (v_1, \ldots, v_n)$, P's feasible set is a neighborhood of v. P is easily solved by searching the neighborhood. If only some of the variables $(x_1, \ldots, x_k)$ are set to $(v_1, \ldots, v_k)$, P is regarded as too hard to solve.

A pure local search algorithm, such as simulated annealing, branches on the original problem P_0 by setting all the variables at once to $v = (v_1, \ldots, v_n)$. The resulting restriction P is solved by searching a neighborhood of v. Supposing P's solution is v', the search backtracks to P_0 and branches again by setting $x = v'$. Thus, in pure local search, the search tree is never more than one level deep. The algorithm stops generating branches whenever the user terminates the search, generally long before the search is exhaustive.

Let $v_{\text{UB}} = \infty$ and $S = \{P_0\}$.
While S is nonempty repeat:
    Select a restriction $P \in S$ and remove P from S.
    If P is too hard to solve then
        Add a restriction of P to S.
    Else
        Let v be the value of P's solution and let $v_{\text{UB}} = \min\{v, v_{\text{UB}}\}$.
        Remove P from S.
The best solution found for P_0 has value v_{UB}.

Figure 2.15. Generic algorithm for local search conceived as branching. The algorithm solves a minimization problem P_0. Set S contains the problem restrictions generated so far. v_{UB} is the value of the incumbent solution. Note that the algorithm is almost identical to the generic branching algorithm of Figure 2.1.

In simulated annealing, P is "solved" by randomly selecting one or more elements of the neighborhood until one of them, say v', is accepted. The search backtracks to P_0 and branches by setting $x = v'$.

In a GRASP-like algorithm, the branching choices differ in the constructive and local search phases. In the constructive phase, the search branches by setting variables one at a time. At the original problem P_0, it branches by setting one variable, say x_1, to a value v_1 chosen in a randomized greedy fashion. It then branches again by setting x_2, and so forth. The resulting restrictions P are regarded as too hard to solve until all the variables x are set to some value v. When this occurs, a solution v' of P is found by searching a neighborhood of v, and the algorithm moves into the local search phase. It backtracks directly to P_0 and branches by setting $x = v'$ in one step. Local search continues as long as desired, whereupon the search returns to the constructive phase.

It was noted earlier that branching need not create a partition, and this is true in particular of a GRASP scheme. Figure 2.16, for instance, illustrates a small GRASP search in which the initial constructive phase assigns variables x_1, x_2, and x_3 the values A, B, and C, respectively, thus arriving at Restriction 3. At this point, the algorithm moves into the local search phase. It searches a neighborhood of $x = (A, B, C)$ by considering all interchanges of two components of x and selects $x = (B, A, C)$. It backtracks to the root and immediately generates a branch (Restric-

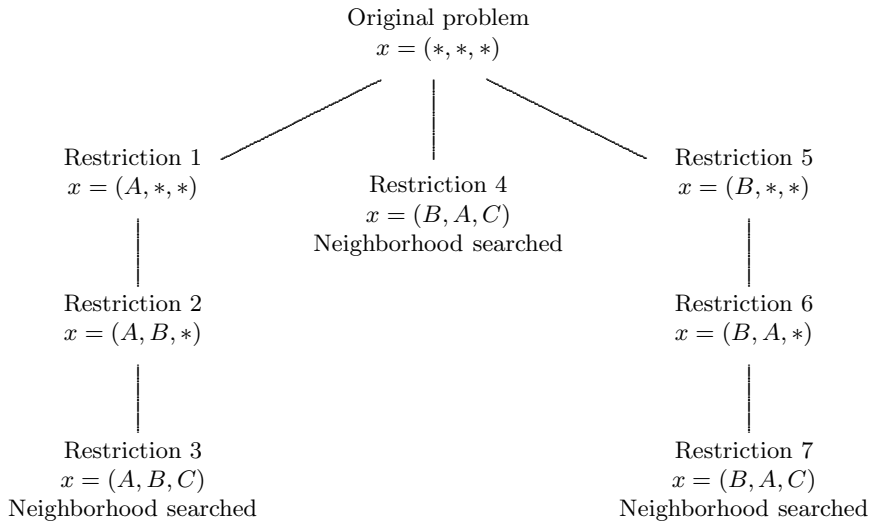

Figure 2.16. Branching tree for a GRASP search. $x = (A, *, *)$ indicates that x_1 is set to A, but x_2 and x_3 are not set.

tion 4), at which the feasible set is a neighborhood of $x = (B, A, C)$. After searching this neighborhood, the local search is terminated and a new constructive phase assigns B, A, and C, respectively, to x_1, x_2, and x_3, thus arriving at Restriction 7. The neighborhood here is the same as for Restriction 4. Thus the branches at the root node do not create a partition: $x = (B, A, C)$ is consistent with two of the branches.

2.4.3 Relaxation

Conceiving local search as part of a quasi-branching scheme has the advantage of revealing an analogy with branch-and-relax algorithms, and thereby suggesting how relaxation can be used to accelerate the search.

The idea can be illustrated in the example of Figure 2.16. Suppose that an objective function $f(x)$ is to be minimized. Thus the solution of restriction 3 has value $f(B, A, C)$. Suppose that $x = (B, A, C)$ is still the incumbent solution when restriction 5 is encountered. If a *relaxation* of restriction 5 is solved and its value is no less than $f(B, A, C)$, there is no need to branch further at restriction 5. Restrictions 6 and 7 are pruned from the tree. Figure 2.17 contains a generic local search algorithm with relaxation.

In ordinary branch-and-relax algorithms, pruning the tree at some node ensures that no problem below the node will be solved. This is not true of local search. For example, Restriction 7 in Figure 2.16 is solved despite the pruning because it is identical to Restriction 4. In general, a restriction might be reached via several paths in the tree, and pruning one path may leave other access routes open. Nonetheless, pruning by relaxation reduces the size of the search tree that would otherwise be traversed. This is illustrated by the example in Section 2.4.4.

Let $v_{\text{UB}} = \infty$ and $S = \{P_0\}$.
While S is nonempty repeat:
    Select a restriction $P \in S$ and remove P from S.
    If P is too hard to solve then
        Let v_{R} be the optimal value of a relaxation of P.
        If $v_{\text{R}} < v_{\text{UB}}$ then
            Add a restriction of P to S.
    Else
        Let v be the value of P's solution and $v_{\text{UB}} = \min\{v, v_{\text{UB}}\}$.
The best solution found for P_0 has value v_{UB}.

Figure 2.17. Generic local-search-and-relax algorithm for solving a minimization problem P_0. The notation is the same as in Figure 2.15.

2.4.4 Constraint-Directed Local Search

Nothing in the generic local search algorithm of Figure 2.15 or 2.17 prevents enumeration of the same solution several times. Repetition can be reduced or eliminated by maintaining a list of nogoods (i.e., a tabu list) and rejecting any solution or partial solution that violates one of the nogoods. The list could be of finite length, as in tabu search, or it could remember all nogoods generated. In the latter case, the search would eventually become complete.

In exhaustive constraint-directed search, the algorithm terminates when the nogood set becomes infeasible. An inexhaustive version can be obtained by generating less than a complete set of nogoods. For instance, older nogoods can be dropped as in tabu search. This means that the nogood set may never become infeasible, and some other termination condition must be used. A generic constraint-directed local search algorithm appears in Figure 2.18.

In particular, partial-order dynamic backtracking can be converted to an inexhaustive search method by dropping older nogoods from the relaxation. In fact, since the nogood set remains small, it may be practical to process the nogoods more intensely than with the parallel resolution method described in Section 2.3.3. This allows more freedom in the solution of the current nogood set, because the solution may be allowed to *conform* to previous solutions in a weaker sense. By carrying over such ideas from constraint-directed search to heuristic methods, one can obtain an entire family of generalizations and extensions of tabu search.

Let $v_{\text{UB}} = \infty$, and let R be a relaxation of P.
While R is feasible repeat as desired:
  Select a feasible solution $x = s(R)$ of R.
  If x is feasible in P then
    Let $v_{UB} = \min\{v_{UB}, f(x)\}$.
    Define a nogood N that excludes x and possibly other solutions x' with $f(x') \geq f(x)$.
  Else
    Define a nogood N that excludes x and possibly other solutions that are infeasible in P.
  Remove nogoods from R as desired.
  Add N to R and process R.
The optimal value of P is v_{UB}.

Figure 2.18. Generic constraint-directed local search algorithm for solving a minimization problem P with objective function $f(x)$, where s is the selection function. R is the relaxation of the current problem restriction.

A Benders method can also be converted to a heuristic method by dropping older Benders cuts from the master problem, or perhaps generating cuts that are too weak to ensure termination. Such techniques may be used by practitioners when a Benders algorithm bogs down.

2.4.5 Example: Single-Vehicle Routing

The idea of a local-search-and-relax algorithm can be illustrated with a single-vehicle routing problem with time windows, also known as a traveling salesman problem with time windows. A vehicle must deliver packages to several customers and then return to its home base. Each package must be delivered within a certain time window. The truck may arrive early, but it must wait until the beginning of the time window before it can drop off the package and proceed to the next stop. The problem is to decide in what order to visit the customers so as to return home as soon as possible, while observing the time windows.

The data for a small problem appear in Table 2.13. The home base is at location A, and the four customers are located at B, C, D and E. The travel times are symmetric, and so the time from A to B and from B to A is 5, for instance. The time windows indicate the earliest and latest time at which the package may be dropped off. The vehicle leaves home base (location A) at time zero and returns when all packages have been delivered.

Exhaustive enumeration of the twenty-four possible routings would reveal six feasible ones: ACBDEA, ACDBEA, ACDEBA, ACEDBA, ADCBEA, and ADCEBA. The last one is optimal and requires thirty-four time units to complete.

Local Search

A simple heuristic algorithm adds one customer at a time to the route in a greedy fashion, by adding the customer that can be served the earliest. The search creates a branch whenever a customer is added. When all customers have been served, or when it is no longer possible to observe

Table 2.13. Data for a small single-vehicle routing problem with time windows.

	Travel time to:			
Origin	B	C	D	E
A	5	6	3	7
B		8	5	4
C			7	6
D				5

Customer	*Time window*
B	[20,35]
C	[15,25]
D	[10,30]
E	[25,35]

time windows, the search jumps to a random node N in the current search tree. It deletes from the tree all successors of N to keep memory requirements under control. It creates a branch at N by adding a random customer. At subsequent branches, customers are added according to the greedy criterion. The process can start over repeatedly as desired by returning to the root node.

This algorithm can be viewed as a generalized GRASP. It is a GRASP in the sense that it alternates between a greedy phase and a local search phase. The greedy phase constructs a solution as in an ordinary GRASP. The local search phase, however, does not necessarily select the next solution from a neighborhood of the current solution, as in a conventional GRASP. Rather, it randomly jumps to a previously enumerated partial solution and randomly instantiates one more variable. If the random jump is restricted to a jump to the immediate successor of the current leaf node, then the random instantiation is equivalent to randomly selecting a solution in a neighborhood of the current solution, where the neighborhood consists of solutions that differ in one variable. Thus, when the random jump is restricted in this way, a generalized GRASP becomes a conventional GRASP.

Figure 2.19 illustrates a possible search. Starting from the home base (Node 0), the earliest possible delivery is to Customer D at time 10. The travel time to D is only 3, but D's time window starts at 10. The search therefore branches to Node 1. Departing Customer D at time 10, the earliest possible delivery is to Customer C at time 17, and so forth. The greedy procedure is fortunate enough to obtain a feasible solution at Node 3 without backtracking. The search jumps randomly to Node 1, whereupon Nodes 2 and 3 are deleted. A randomly chosen customer, E, is added to the route, and the greedy criterion adds Customer B at node 5. This violates the time windows, and the search randomly jumps to Node 0, where it randomly adds Customer B. Nodes 1–5 are deleted, and the greedy process obtains another infeasible routing at Node 8. The search is arbitrarily terminated at this point.

This can be viewed as a local search algorithm in the sense that the greedy procedure searches a neighborhood in the space of problem restrictiontions. The neighborhood consists of all restrictions that can be formed from the current restriction by adding a customer to the end of the route. A deleted node can reappear due to subsequent branching.

Local Search with Relaxation

A relaxation mechanism can help the search avoid unproductive areas of the search tree. One way to relax the problem is to replace the travel times for unscheduled trip segments with lower bounds on the travel

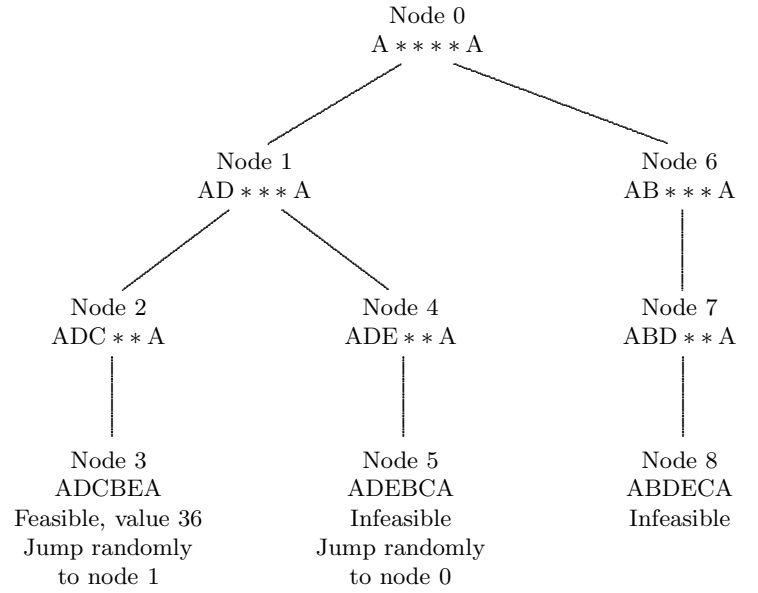

Figure 2.19. Local-search tree for a single-vehicle routing problem with time windows. The notation AD ∗ ∗ ∗ A *indicates a partial routing that runs from A to D, through 3 unspecified stops, and back to A.*

times. A segment is the portion of the trip between two customers, i and j, that are adjacent on the route. If customer j has not been scheduled, then the preceding customer i and the segment travel time are unknown. Yet a simple lower bound on this time is the travel time to j from the nearest customer that could precede j.

To make this more precise, let t_{ij} be the travel time between customers i and j, and let variable x_i be the ith customer visited (where x_0 is fixed to be the home base, Customer 0). Suppose a partial route consisting of the first k customers has been formed, so that $x_0, \ldots, x_k$ have been assigned distinct values. For $j \notin \{x_0, \ldots, x_k\}$, the travel time to customer j from the customer that precedes it in the route will be at least

$$L_j = \min_{i \notin \{j, x_0, \ldots, x_{k-1}\}} \{t_{ij}\}$$

and the travel time from the last customer served to the base will be at least

$$L_0 = \min_{j \notin \{x_0, \ldots, x_k\}} \{t_{j0}\}$$

Then, if T is the earliest time the vehicle can depart customer k,

$$T + L_0 + \sum_{j \notin \{x_0, \ldots, x_k\}} L_j$$

is a lower bound on the duration of any completion of the partial route. If this value is greater than or equal to the value of the incumbent solution, there is no need to branch further.

The search algorithm can be amended so that whenever the tree can be pruned at some node by bounding, that node is deleted from the tree. The search then proceeds exactly as it does when it constructs a feasible route or encounters infeasibility: it jumps to a randomly chosen node that remains in the tree and branches by adding a random customer to the end of the route. A node deleted by bounding may reappear due to subsequent branching, whereupon it will again be deleted. Unlike a conventional GRASP, this particular algorithm will never find an alternate route to solutions below a node that is pruned by bounding.

It is illustrative to rerun the search of Figure 2.19 with bounding, and the result appears in Figure 2.20. In the partial route ADE at Node 4, the vehicle cannot depart E before time 25. Since B and C are unscheduled, the lower bound on the duration of the completed route is

$$25 + \min\{t_{CB}, t_{EB}\} + \min\{t_{BC}, t_{EC}\} + \min\{t_{BA}, t_{CA}\} = 40$$

Since this is larger than the incumbent value of 36, Node 4 is deleted. The search randomly jumps to Node 0 and randomly adds customer B at Node 5. Here the relaxation value is 38, which again allows the node to be pruned.

Constraint-Directed Search

The vehicle routing problem can solved in a manner similar to partial-order dynamic backtracking, as illustrated in Table 2.14. However, since the size of the nogood set will be limited, it is practical to process the nogood set more thoroughly to avoid backtracking while solving it. This, in turn, allows one to solve the relaxation without any restrictions other than the nogoods themselves.

Initially there are no nogoods, and a greedy algorithm selects the first solution ADCBEA by moving from each customer to the next customer that can be served most quickly. The greedy solution is feasible, and the nogood ADCB is generated to rule out this particular nogood. The meaning of the nogood ADCB is that no solution beginning ADCB can be considered. In Iteration 1, the greedy algorithm is constrained by the nogood ADCB and selects ADCEBA, which generates nogood ADCE.

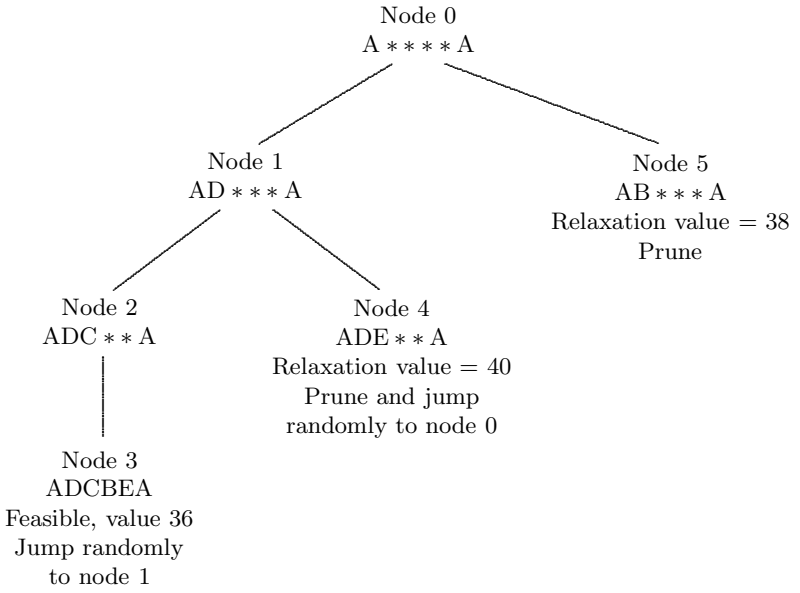

Figure 2.20. Local-search-and-relax tree for a single-vehicle routing problem with time windows. The notation is the same as in Figure 2.19.

Table 2.14. Solution of a single-vehicle problem with time windows by incomplete constraint-directed search.

k	*Relaxation* R_k	*Solution of* R_k	*Value*	*New nogoods*
0		ADCBEA	36	ADCB
1	ADCB	ADCEBA	34	ADCE
2	ADC	ADBECA	infeas.	EC
3	ABDE, ABEC ADBE, ADC ADEC, AEC	ADBCEA	infeas.	BC
4	ABC, ABEC AD, AEC, AEDB	ACDBEA	38	ACDB
5	ABC, ACDB AD, AEDB	ACDEBA	36	ACDE
⋮				

These two nogoods obtained so far exclude all solutions beginning ADC, and so the nogood ADC comprises the relaxation in Iteration 2. In effect ADCB and ADCE are resolved to yield ADC.

The greedy solution subject to ADC is the infeasible solution ADBECA. Some analysis reveals that the cause of the infeasibility is the subsequence EC, which is therefore generated as a nogood. To avoid backtracking in the solution of the next relaxation R_3, all excluded subsequences beginning with A must be spelled out: AEC, ABEC, ADEC, ABDE, ADBE. These are added to R_3, which has solution ADBCEA, which is again infeasible. Because subsequence BC is the cause of the infeasibility, all excluded subsequences beginning with A are added to the nogood set, and all possible resolutions performed to obtain the relaxation shown for Iteration 4. The resulting feasible solution generates nogood ACDB.

At this point, some of the older nogoods are dropped before adding ACDB to keep the nogood list short. Since the nogoods in R_1 and R_2 are no longer present in R_4, the nogoods in R_3 that are still present are dropped, leaving ABC, AD, and AEDB. Now, the new nogood ACDB is added to obtain R_5. The process continues in this fashion until one wishes to terminate it. As it happens, the algorithm discovers the optimal solution ADCEBA in Iteration 1.

2.4.6 Exercises

1 Consider a knapsack packing problem in which the objective is to maximize cx subject to $ax \leq 26$ and each $x_j \in \{0, 1\}$, where the data appear in Table 2.15. Generate part of a local-search-and-relax-tree similar to that of Figure 2.20. At each step of the greedy phase, fix to 1 the variable x_i that has not already been fixed and that has the largest ratio c_i/a_i. A leaf node is reached when no more variables can be fixed to 1. Thus every leaf node will correspond to a feasible solution. After evaluating a leaf node, backtrack to a random node in the current tree, and randomly select the next variable to instantiate before resuming the greedy approach. As a relaxation, maximize cx subject to $ax \leq 26$, $x_j \in [0, 1]$, and the currently fixed values (this is trivial to solve). For instance, after finding the first feasible solution (which has value 53), one might randomly backtrack to the

Table 2.15. Data for a small knapsack packing problem.

i	1	2	3	4	5
c_i	24	14	15	9	14
a_i	11	7	8	5	9
c_i/a_i	2.182	2.000	1.875	1.800	1.556

root node, which causes the other nodes created so far to be deleted. If one randomly selects $x_5 = 1$, the optimal value of the relaxation is 50. This is worse than the incumbent value 53, thus allowing the tree to be pruned. The search backtracks to the root node (the only other node in the tree) and randomly selects another variable to instantiate to 1.

2 Solve the problem of Exercise 1 with constraint-directed search. The heuristic for solving the current problem restriction is to fix the variables in the order $x_1, \ldots, x_5$, fixing each x_i to 1 if this, in combination with the variables already fixed, does not violate a nogood. If the solution is infeasible, the nogood is

$$x_1^{1-v_1} \vee \cdots \vee x_j^{1-v_j} \tag{2.47}$$

where v_i is the value to which x_i is fixed, $x_i^1 = x_i$, $x_i^0 = \neg x_i$, and j is the smallest index such that $\sum_{i=1}^{j} a_i v_i > 26$. If the solution is feasible, the nogood is (2.47) with $j = 5$. Since the order of instantiation is constant, parallel resolution of the nogoods is adequate. Solve the problem by a complete search, which implements a depth-first search with conflict clauses. The first few steps of the search appear in Table 2.16. If the older nogoods are dropped as the search proceeds, the result is an incomplete constraint-directed search. Then, instantiate the variables in any order, but apply full resolution to the nogoods, dropping the older nogoods as the search proceeds. One could also use an incomplete form of partial-order dynamic backtracking. All of these incomplete searches might be described as sophisticated forms of tabu search.

Table 2.16. First few iterations of constraint-directed search for a knapsack packing problem.

k	*Relaxation* R_k	*Solution* x *of* R_k	*Value*	*New nogoods*
0		$(1,1,1,1,1)$	infeas.	$\neg x_1 \vee \neg x_2 \vee \neg x_3 \vee \neg x_4$
1	$\neg x_1 \vee \neg x_2 \vee \neg x_3 \vee \neg x_4$	$(1,1,1,0,1)$	infeas.	$\neg x_1 \vee \neg x_2 \vee \neg x_3 \vee \neg x_5$
2	$\begin{cases} \neg x_1 \vee \neg x_2 \vee \neg x_3 \vee \neg x_4 \\ \neg x_1 \vee \neg x_2 \vee \neg x_3 \vee \neg x_5 \end{cases}$	$(1,1,1,0,0)$	53	$\neg x_1 \vee \neg x_2 \vee \neg x_3 \vee x_4 \vee x_5$
3	$\neg x_1 \vee \neg x_2 \vee \neg x_3$	$(1,1,0,1,1)$	infeas.	$\neg x_1 \vee \neg x_2 \vee x_3 \vee \neg x_4 \vee \neg x_5$
$\vdots$				

3 A *genetic algorithm* mimics evolution by natural selection. It begins with a set of solutions (i.e., a population) and allows some pairs of solutions, perhaps the best ones, to mate. A crossover operation produces an offspring that inherits some characteristics of the parent solutions. At this point, the less desirable solutions are eliminated from the population so that only the fittest survive. The process repeats for several generations, and the best solution in the resulting population is selected. Indicate how this algorithm can be viewed as examining a sequence of problem restrictions. In what way does generation of offspring produce a relaxation of the current restriction? If the "solution" of the relaxation is the selection of some good solutions, how does solution of the relaxation guide the selection of the next restriction? Why is relaxation bounding, however, unhelpful in this algorithm? Hint: relaxation bounding is helpful when it obviates the necessity of solving the current restriction. Think about how the current relaxation is obtained.

4 *Ant colony optimization* can be applied to the traveling salesman problem on n cities as follows. Initially all the ants of the colony are in City 1. In each iteration, each ant crawls from its current location i to city j with probability proportional to u_{ij}/d_{ij}, where u_{ij} is the density of accumulated pheromone deposit on the trail from i to j, and d_{ij} is the distance from i to j. Each ant deposits pheromone at a constant rate while crawling, and a certain fraction of the pheromone evaporates between each iteration and the next. Each ant remembers where it has been and does not visit the same city twice until all cities have been visited. After returning to City 1, the ants forget everything and start over again. When the process terminates, the shortest tour found by an ant is selected. Show how this algorithm can be understood as enumerating problem restrictions. How can relaxation bounding be introduced into the algorithm?

5 *Particle swarm optimization* can be applied to global optimization, as follows. The goal is to search a space of many dimensions for the best solution. A swarm of particles are initially distributed randomly through the space. Certain particles have two-way communication with certain others. In each iteration, each particle moves randomly to another position, but with higher probability of moving closer to a communicating particle that occupies a good solution. After many iterations, the best solution found is selected. How can this process be viewed as enumerating a sequence of problem restrictions? Why is there no role for relaxation bounding here?

2.5 Bibliographic Notes

Section 2.1. Various elements of the search-infer-and-relax framework presented here were proposed in [12, 57, 182, 184, 185, 188, 198, 200]. An extension to dynamic backtracking and heuristic methods is given in [192, 193].

Section 2.2. Cuts for general integer knapsack constraints are discussed in [75, 234, 268, 269]. A comprehensive treatment can be found in [14], which strengthens the inequalities discussed here. Nearly all development of knapsack cuts, however, has been concerned with 0-1 knapsack constraints, beginning with [15, 166, 262, 335].

Conditional modeling is advocated in [198]. The idea of disjunctive modeling goes back at least to [146, 159] and is developed in [16, 17, 33, 60, 161, 198, 222, 277, 322] and elsewhere.

The employee timetable model is similar to one presented in [285]. An overview of employee timetabling appears in [238].

Continuous global optimization is extensively surveyed in [252], and the integrated approach taken here is similar to that described in [312] and implemented in the solver BARON. Factorization of functions for purposes of relaxation was introduced by [236].

The product configuration model is based on [315]. The generic element constraint was introduced by [174]. The form of the constraint used here appears in [315].

Column generation methods have been used for decades. A unifying treatment of branch and price for mixed integer programming can be found in [27]. Branch-and-price with CP-based column generation originated with [209, 343], and the area is surveyed in [115]. The airline crew rostering example described here is based on [120]. CP-based branch-and-price methods are surveyed in [115, 290].

Section 2.3. Constraint-directed search is discussed in connection with dynamic backtracking in [144, 145, 235], which also point out the connection between nogood-based search and branching. The ideas are further developed in [185].

The Davis-Putnam-Loveland (DPL) method for the propositional satisfiability problem was originally a resolution method proposed by Davis and Putnam [102]. Loveland [227] replaced the resolution step with branching. The fastest satisfiability algorithms, such as CHAFF [242], combine DPL with clause learning [32].

Partial-order dynamic backtracking was introduced in [235] and generalized in [56]. It is unified with other forms of dynamic backtracking and further generalized in [185], which also proves the completeness and polynomial complexity of parallel resolution for partial-order dynamic backtracking.

Classical Benders decomposition is due to [43] and was generalized to nonlinear programming in [142]. Logic-based Benders decomposition was introduced in [201] and developed in [185, 199]. Its application to planning and scheduling, with a CP subproblem, was proposed in [185], first implemented in [203], and extended in [168]. The machine scheduling example described here is adapted from [189]. Edge finding originates in [69, 70]. Branch and check is proposed in [185] and successfully implemented in [314].

Section 2.4. The integrated approach to heuristic methods presented here follows [193]. Tabu search is due to [150, 167]. GRASP originated with [307]. The idea of using relaxations in local search appears in [271].

http://www.springer.com/978-0-387-38272-2

Integrated Methods for Optimization
Hooker, J.N.
2007, XIV, 486 p. 72 illus., Hardcover
ISBN: 978-0-387-38272-2