A Search-Infer-and-Relax Framework for Integrating Solution Methods

J.N. Hooker

Tepper School of Business,
Carnegie Mellon University,
Pittsburgh, USA
john@hooker.tepper.cmu.edu

Abstract. We present an algorithmic framework for integrating solution methods that is based on search, inference, and relaxation and their interactions. We show that the following are special cases: branch and cut, CP domain splitting with propagation, popular global optimization methods, DPL methods for SAT with conflict clauses, Benders decomposition and other nogood-based methods, partial-order dynamic backtracking, various local search metaheuristics, and GRASPs (greedy randomized adaptive search procedures). The framework allows elements of different solution methods to be combined at will, resulting in a variety of integrated methods. These include continuous relaxations for global constraints, the linking of integer and constraint programming via Benders decomposition, constraint propagation in global optimization, relaxation bounds in local search and GRASPs, and many others.

1 Introduction

The constraint programming and optimization communities have developed a wide variety of effective methods for solving combinatorial optimization problems. Yet they are described in different literatures using different terminology and implemented in a growing collection of different solvers. Recent advances in hybrid methods show how to integrate algorithmic ideas from several sources, but hybrid methods themselves are multiplying, since there are so many ways to hybridize. Practical application would be much more effective if a single solver could bring a wide variety of methods under one roof, not only to allow the user to select the best one, but to allow the integration of techniques from different methods.

We suggest that the goal of integration should be addressed at a fundamental and conceptual level rather than postponing it to the software design stage. The growing repertory of combinatorial optimization methods should be interpreted as special cases of a single solution method that can be adjusted to exploit the structure of a given problem. This overarching method would then dictate the architecture of a general-purpose solver.

One approach, some elements of which are proposed in [4, 10, 11, 12, 13, 15, 17], is to view solution methods as instances of a *search-infer-and-relax algo-*

rithm. The search phase enumerates restrictions of the problem, perhaps by branching, neighborhood search, or creation of subproblems. Inference may take the form of cutting planes, filtering, or nogood generation. Relaxation provides bounds on the optimal value that can reduce the search space.

We show in this paper that a wide variety of solution methods have this structure, including branch and cut, standard CP methods, popular global optimization methods, DPL methods for the propositional satisfiability problem, generalizations of Benders decomposition and other varieties of nogood-based search, partial-order dynamic backtracking and related methods, local search metaheuristics, and GRASPs (greedy randomized adaptive search procedures).

However, it is one thing to observe in a general way that solution algorithms tend to have a search-infer-and-relax structure, and another thing to demonstrate it in precise algorithmic terms. While such methods as branch and cut or standard CP methods readily fit into this framework, it is less obvious how to treat some of the other methods. The main contribution of this paper, relative to previous work, is to extend the range of solution methods that can be viewed as having common structure, while trying to make their commonality more precise.

In particular, we extend the analysis to "heuristic" methods, such as local search and GRASPs. Although one can distinguish exact from inexact methods, this distinction need not imply a fundamental distinction of the algorithmic approach. We view them as special cases of the same search strategy, adjusted in some cases to be exhaustive and in other cases to be inexhaustive.

Some aspects of the integration scheme described here are implemented in the solution and modeling system SIMPL [1], which combines integer and constraint programming but has not yet been extended to other methods.

2 The Basic Ideas

– *Search* is an enumeration of problem restrictions, each of which is obtained by adding constraints to the problem. The motivation for examining problem restrictions is that they may be easier to solve than the original. In branching search, for example, the problem restrictions correspond to nodes of the search tree. In Benders decomposition and its generalizations, the restrictions are subproblems. In local search, each neighborhood is the feasible set of a problem restriction.
– *Inference* derives valid constraints that were only implicit in the constraint set. They can rule out infeasible or suboptimal restrictions that would otherwise be solved. Popular forms of inference include the identification of valid inequalities in integer programming, the generation of nogoods (such as Benders cuts and conflict clauses), and domain filtering in constraint programming.
– *Relaxation*, like restriction, is motivated by the desire to solve a problem that is easier than the original. Solution of a relaxation may provide an optimal solution of the original problem, but more often it provides a bound on the optimal value. Popular forms of relaxation include the constraint

store in constraint programming, continuous relaxations of 0-1 inequalities or global constraints, and the master problem in Benders decomposition and its generalizations.

The interaction of these elements is key to problem solving.

- *Search and inference.* Inference reduces the number of restrictions that must be enumerated in the search. For instance, domain filtering reduces branching by eliminating values on which one must branch. Conversely, restricting the problem can make inference more effective. Branching on variables, for example, reduces domains and triggers further domain reduction through propagation.
- *Search and relaxation.* Relaxation provides valuable information for directing the search. For instance: the solution of a continuous relaxation suggests how to branch (perhaps on a variable with a fractional value); the solution of a master problem can define the next subproblem (in Benders-like methods); and the result of a neighborhood search can provide the center of the next neighborhood to be searched. Conversely, problem restriction during search can yield a tighter relaxation, perhaps one whose optimal solution is feasible in the original problem. Relaxation and restriction also interact in a bounding mechanism that is used by branch-and-relax methods but has much wider application. If the relaxation of a restriction has an optimal value that is no better than that of the best solution found so far, then the restriction need not be solved.
- *Inference and relaxation.* The solution of a relaxation can help identify useful inferences, such as separating cuts in integer programming. Conversely, inference can generate constraints that strengthen the relaxation, as cutting planes strengthen a continuous relaxation.

Inference and relaxation are most effective when they exploit problem structure. For instance, specialized cutting planes or domain filtering methods can be developed for constraints or subsets of constraints that have special characteristics. Arguably the success of combinatorial optimization relies on the identification of structure, and the problem formulation should indicate to the solver where the structure lies.

3 Overview of the Solution Method

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

$$\begin{aligned} &\min\ f(x) \\ &S(x) \\ &x \in D \end{aligned}$$

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 subject to a set $S(x)$ of constraints, each of which is either satisfied or violated by any given $x \in D$. Generally x is a vector $(x_1, \ldots, x_n)$ and D 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 P. A *feasible* solution is one that satisfies all the constraints in $S(x)$, and the *feasible set* of P 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 and is said to have optimal value ∞.

3.1 Search

Search is carried out by solving a series of problem restrictions $P_1, P_2, \ldots, P_m$ of P and picking the best *candidate* solution. The search is *complete* if the feasible set of P is equal to the union of the feasible sets of $P_1, \ldots, P_m$. In incomplete search the restrictions may not be solved to optimality.

The most basic kind of search simply enumerates elements of the domain D and selects the best feasible solution. This is 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 local search.

3.2 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 P_k. Constraints that can be inferred from P alone are added to P_k and all subsequent restrictions.

Inference procedures are typically applied to individual constraints or small highly-structured groups of constraints rather than the entire problem. As a result, implications of the entire constraint set may be missed.

One can partially address this problem by propagating constraints through a constraint store S. When inferences are drawn from constraint C, they are actually drawn from $\{C\} \cup S$. Processing each constraint enlarges S and thereby strengthens the implications that can be derived from the next constraint. Propagation of this sort is practical only if the constraint store contains elementary constraints that all of the inference algorithms can accommodate. Constraint programming solvers typically store in-domain constraints, and mixed integer solvers store linear inequalities.

3.3 Relaxation

Relaxation is often used when the subproblems P_k are themselves hard to solve. A relaxation R_k of each P_k is 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 variable to take any real value.

The optimal value v of the relaxation R_k is a lower bound on the optimal value of P_k. If v 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.

```
Let $v_{\mathrm{UB}} = \infty$ and $S = \{P_0\}$. Perform **Branch**.
The optimal value of $P_0$ is $v_{\mathrm{UB}}$.

Procedure **Branch**.
  If $S$ is nonempty then
    Select a problem restriction $P \in S$ and remove $P$ from $S$.
    If $P$ is too hard to solve then
      Add restrictions $P_1, \ldots, P_m$ of $P$ to $S$ and perform Branch.
    Else
      Let $v$ be the optimal value of $P$ and let $v_{\mathrm{UB}} = \min\{v, v_{\mathrm{UB}}\}$.
```

Fig. 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*

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.

4 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. If a restriction P_k is too hard to solve, it attacks P_k in a similar manner by branching on P_k. The most popular branching mechanism is to branch on a variable x_j. The domain of x_j is partitioned into two or more disjoint subsets, and restrictions are created by successively restricting x_j to each of these subsets.

Branching continues until no restriction so far created is left unsolved. If the procedure is to terminate, problems must 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. Figure 1 displays a generic branching algorithm.

4.1 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 domain 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 more rapidly become singletons. *Constraint programming solvers* are typically built on a branch-and-infer framework.

```
Let $v_{\rm UB} = \infty$ and $S = \{P_0\}$. Perform **Branch**.
The optimal value of $P$ is $v_{\rm UB}$.

Procedure **Branch**.
  If $S$ is nonempty then
    Select a problem restriction $P \in S$ and remove $P$ from $S$.
    Repeat as desired:
      Add inferred constraints to $P$.
      Let $v_{\rm R}$ be the optimal value of a relaxation $R$ of $P$.
    If $v_{\rm R} < v_{\rm UB}$ then
      If $R$'s optimal solution is feasible for $P$ then let $v_{\rm UB} = \min\{v_{\rm R}, v_{\rm UB}\}$.
      Else add restrictions $P_1, \ldots, P_m$ of $P$ to $S$ and perform **Branch**.
```

Fig. 2. *Generic branching algorithm, with inference and relaxation, for solving a minimization problem P_0. The 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*

4.2 Branch and Relax

Relaxation can also 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 is feasible in P_k, it is optimal for P_k and becomes a candidate solution. Otherwise the algorithm branches on P_k. To ensure termination, the branching mechanism must be designed so that R_k's solution will in fact be feasible for P_k if one descends deeply enough into the search tree.

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.

The addition of inferred constraints to P_k can result in a tighter bound when one solves its relaxation R_k. This is the idea behind *branch-and-cut* methods, which add cutting planes to the constraint set at some or all of the nodes. Conversely, the solution of R_k can provide guidance for generating further constraints, as for instance when *separating cuts* are used. A generic branching algorithm with inference and relaxation appears in Fig. 2.

It is straightforward to combine elements of constraint programming and integer programming in this framework. Domain filtering can be applied to integer inequalities as well as global constraints at each node of the search tree, and tight relaxations can be devised for global constraints as well as specially-structured inequalities.

4.3 Continuous Global Optimization

A continuous optimization problem may have a large number of locally optimal solutions and can therefore be viewed as a combinatorial problem. The most popular and effective global solvers use a branch-and-relax approach that combines relaxation with constraint propagation [21, 22, 24]. Since the variable domains are continuous intervals, the solver 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.

Two types of propagation are commonly used: bounds propagation, and propagation based on Lagrange multipliers. Bounds propagation is similar to that used in constraint programming solvers. Lagrange multipliers obtained by solving a linear relaxation of the problem provide a second type of propagation. If a constraint $ax \leq \alpha$ has Lagrange multiplier λ, 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

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

can be deduced and propagated. Reduced-cost-based variable fixing is a special case.

Linear relaxations can often be created for nonlinear constraints by "factoring" the functions involved into more elementary functions for which linear relaxations are known [24].

5 Constraint-Directed Search

A 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. This suggests defining the current restriction by adding a constraint that excludes previous solutions, as well as some additional solutions that one can determine in advance would be no better. Such a constraint is often called a *nogood*.

Restrictions defined in this manner may be hard to solve, but one can solve a more tractable relaxation of each restriction rather than the restriction itself. The only requirement is that the relaxation contain the nogoods generated so far. The nogoods should therefore be chosen in such a way that they do not make the relaxation hard to solve.

More precisely, the search proceeds by creating a sequence of restrictions $P_0, P_1, \ldots, P_m$ of P, where $P_0 = P$ and each P_k is formed by adding a nogood N_{k-1} to P_{k-1}. It solves a corresponding series of relaxations $R_0, R_1, \ldots, R_m$ of P to obtain solutions $x^0, x^1, \ldots, x^m$. Each relaxation R_k contains the nogoods $N_0, \ldots, N_{k-1}$ in addition to the constraints in R_0.

Step k of the search begins by obtaining a solution x^k of R_k. If x^k is infeasible in P, a nogood N_k is designed to exclude x^k and possibly some other solutions

```
Let $v_{\mathrm{UB}} = \infty$, and let $R$ be a relaxation of $P$.
Perform **Search**.
The optimal value of $P$ is $v_{\mathrm{UB}}$.

Procedure **Search**.
  If $R$ is feasible then
    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 $P$.
    Add $N$ to $R$ and perform **Search**.
```

Fig. 3. *Generic constraint-directed 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*

that are infeasible for similar reasons. If x^k is feasible in P, it may or may not be optimal, and it is recorded as a candidate for an optimal solution. A nogood N_k is designed to exclude x^k and perhaps other solutions whose objective function values are no better than that of x^k. The search continues until R_k becomes infeasible, indicating that the solution space has been exhausted. An generic algorithm appears in Fig. 3.

The search is exhaustive because the infeasibility of the final relaxation R_m implies the infeasibility of P_m. Thus any feasible solution x of P that is not enumerated in the search is infeasible in P_m. This is possible only if x has been excluded by a nogood, which means x is no better than some solution already found.

If the domains are finite, the search will terminate. Each relaxation excludes a solution that was not excluded by a previous relaxation, and there are finitely many solutions. If there are infinite domains, more care must be exercised to ensure a finite search and an optimal solution.

Interestingly, there is no need to solve the relaxations R_k to optimality. It is enough to find a feasible solution, if one exists. This is because no solution is excluded in the course of the search unless it is infeasible, or an equally good or better solution has been found.

There is normally a good deal of freedom in how to select a feasible solution x^k of R_k, and a constraint-directed search is partly characterized by its *selection function*; that is, by the way it selects a feasible solution $s(R_k)$ for a given R_k. Certain selection functions can make subsequent R_k's easier to solve, a theme that is explored further below.

We briefly examine three mechanisms for generating nogoods: constraint-directed branching, partial-order dynamic backtracking, and logic-based Benders decomposition.

5.1 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, which are defined in part by nogoods that exclude previous leaf nodes. The nogoods take the form of "conflict clauses" that contain information about why the search backtracked at previous leaf nodes. Well-chosen conflict clauses 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. Conflict clauses have played a particularly important role in fast algorithms for the propositional satisfiability problem (SAT), such as Chaff [20].

Branching can be understood as constraint-directed search in the following way. We branch on variables in a fixed order $x_1, \ldots x_n$. The original problem P corresponds to the first leaf node of the branching tree, and its relaxation R contains only the domain constraints of P. The branching process reaches the first leaf node by fixing $(x_1, \ldots, x_n)$ to certain values $(v_1, \ldots, v_n)$, thereby creating P_1. If the search backtracks due to infeasibility, typically only some of the variables are actually responsible for the infeasibility, let us say the variables $\{x_j \mid j \in J\}$. A nogood or *conflict clause* N is constructed to avoid this partial assignment in the future:

$$\bigvee_{j \in J} (x_j \neq v_j) \tag{1}$$

If a feasible solution with value z is found at the leaf node, then a subset of variables $\{x_j \mid j \in J\}$ is identified such that $f(x) \geq z$ whenever $x_j = v_j$ for $j \in J$. A nogood N of the form (1) is created.

Each of the subsequent leaf nodes corresponds to a relaxation R_k, which contains the nogoods generated so far. A feasible solution $s(R_k)$ of R_k is now selected to define the next solution to be enumerated. A key property of constraint-based branching is that the selection function s is easy to compute. The solution $s(R_k)$ sequentially assigns $x_1, x_2, \ldots$ the values to which they are fixed at the current leaf node, until such an assignment violates a nogood in R_k. At this point the unassigned variables are sequentially assigned any value that, together with the assignments already made, violates none of the nogoods in R_k. Constraint-based search does not actually construct a search tree, but the values to which x_j is fixed at the current leaf node are encoded in the nogoods: if one or more nogoods in R_k contain the disjunct $x_j \neq v_j$, then x_j is currently fixed to v_j (all disjuncts containing x_j will exclude the same value v_j).

It is shown in [12] that this procedure finds a feasible solution of R_k without backtracking, if one exists, provided the nogoods are processed by *parallel resolution* before computing $s(R_k)$. Consider a set $S = \{C_i \mid i \in I\}$ where each C_i has the form

$$\bigvee_{j \in J_i} (x_j \neq v_j) \vee (x_p \neq v_{pi})$$

and where p is larger than all the indices in $J = \bigcup_{i \in I} J_i$. If $\{v_{pi} \mid i \in I\}$ is equal to the domain of x_p, then S has the parallel resolvent

$$\bigvee_{j \in J} (x_j \neq v_j)$$

Thus parallel resolution always resolves on the *last* variable x_p in the clauses resolved. (In constraint-directed branching, J_i is the same for all $i \in I$, but this need not be true in general to derive a parallel resolvent.) Parallel resolution is applied to R_k by deriving a parallel resolvent from a subset of nogoods in R_k, deleting from R_k all nogoods dominated by the resolvent, adding the resolvent to R_k, and repeating the process until no parallel resolvent can be derived. In the context of constraint-based branching, parallel resolution requires linear time and space.

The Davis-Putnam-Loveland (DPL) algorithm for SAT is a special case of constraint-directed branching in which the *unit clause rule* is applied during the computation of $s(R_k)$. The SAT problem is to determine whether a set of logical clauses is satisfiable, where each clause is a disjunction of literals (x_j or $\neg x_j$). The unit clause rule requires that whenever x_j (or $\neg x_j$) occurs as a unit clause (a clause with a single literal), x_j is fixed to true (respectively, false) and the literal $\neg x_j$ (respectively, x_j) is eliminated from every clause in which it occurs. The procedure is repeated until no further variables can be fixed. During the computation of $s(R_k)$, the unit clause rule is applied after each x_j is assigned a value, and subsequent assignments must be consistent with any values fixed by the rule.

An infeasible assignment $(x_1, \ldots, x_n) = (v_1, \ldots, v_n)$ for the SAT problem is one that violates one or more clauses. A conflict clause (1) is obtained by identifying a subset of variables $\{x_j \mid j \in J\}$ for which the assignments $x_j = v_j$ for $j \in J$ violate at least one clause. Thus if setting $(x_1, x_2) = (\text{true}, \text{false})$ violates a clause, the conflict clause is $\neg x_1 \vee x_2$.

5.2 Partial-Order Dynamic Backtracking

Partial-order dynamic backtracking (PODB) is a generalization of branching with conflict clauses [3, 8, 9]. In a conventional branching search, one backtracks from a given node by de-assigning the assigned variables in a certain order. In PODB, this complete ordering of the assigned variables is replaced by a partial ordering. Thus the search cannot be conceived as a tree search, but it remains exhaustive while allowing a greater degree of freedom in how solutions are enumerated.

PODB can be viewed as constraint-based search in which the selection function $s(R_k)$ is computed in a slightly different way than in constraint-based branching. In constraint based branching, there is a complete ordering on the variables, and $s(R_k)$ is computed by assigning values to variables in this order. In PODB, this ordering is replaced by a partial ordering.

The partial ordering is defined as follows. Initially, no variable precedes another in the ordering. At any later point in the algorithm, the partial ordering is defined by the fact that some variable x_j in each nogood N of R_k has been designated as *last* in N. Every other variable in N is *penultimate* and precedes x_j in the partial ordering. The ordering is updated whenever a new nogood is created. Any variable x_j in the new nogood can be chosen as last, provided the choice is consistent with the current partial ordering. Thus if x_k is a penultimate variable in the nogood, x_j must not precede x_k in the current partial ordering.

The nogoods are processed by parallel resolution exactly as in constraint-based branching, which as shown in [], again consumes linear time and space. The selection function $s(R_k)$ is computed as follows. It first assigns values to variables x_j that are penultimate in some nogood. As before, it assigns x_j the value v_j if the disjunct $x_j \neq v_j$ occurs in a nogood (all penultimate disjuncts containing x_j exclude the same value v_j). The remaining variables are assigned values as in constraint-based branching, but in any desired order.

5.3 Logic-Based Benders Decomposition

Benders decomposition [2, 7] is a constraint-directed search that enumerates possible assignments to a *subset* of the variables, which might be called the *search variables*. Each possible assignment defines a *subproblem* of finding the optimal values of the remaining variables, given the values of the search variables. Solution of the subproblem produces a nogood that excludes the search variable assignment just tried, perhaps along with other assignments that can be no better. Since the subproblems are restrictions of the original problem, a Benders algorithm can be viewed as enumerating problem restrictions.

Benders is applied to a problem P of the form

$$\begin{aligned} &\min\ f(x) \\ &C(x,y) \\ &x \in D_x,\ y \in D_y \end{aligned} \tag{2}$$

where x contains the search variables and y the subproblem variables. $C(x,y)$ is a constraint set that contains variables x, y. To simplify exposition we assume that the objective function depends only on x. The more general case is analyzed in [12, 16].

In the constraint-directed search algorithm, each problem restriction (subproblem) P_k is obtained from P by fixing x to the solution x^{k-1} of the previous relaxation R_{k-1}. P_k is therefore the feasibility problem

$$\begin{aligned} &C(x^{k-1},y) \\ &y \in D_y \end{aligned} \tag{3}$$

where $C(x^{k-1},y)$ is the constraint set that remains when x is fixed to x^{k-1} in $C(x,y)$.

Unlike many constraint-directed methods, a Benders method obtains nogoods by solving the restriction P_k. If P_k has a feasible solution y^k, then $(x,y) = (x^{k-1},y^k)$ is optimal in P, and the search terminates. Otherwise a nogood or

Benders cut $N_k(x)$ is generated. $N_k(x)$ must exclude x^{k-1}, perhaps along with other values of x that are infeasible for similar reasons.

In classical Benders decomposition, the subproblem is a continuous linear or nonlinear programming problem, and $N_k(x)$ obtained from Lagrange multipliers associated with the constraints of the subproblem. In a more general setting, $N_k(x)$ is based on an analysis of how infeasibility of the subproblem is proved when $x = x^{k-1}$. The same proof may be valid when x takes other values, and these are precisely the values that violate $N_k(x)$. The result is a "logic-based" form of Benders.

The Benders cut $N_k(x)$ is added to the previous relaxation R_{k-1} to obtain the current relaxation or *master problem* R_k:

$$\begin{aligned} &\min f(x) \\ &N_i(x), \quad i = 0, \ldots, k-1 \\ &x \in D_x \end{aligned} \tag{4}$$

If the master problem is infeasible, then P is infeasible, and the search terminates. Otherwise we select any optimal solution $s(R_k)$ of R_k and denote it x^k, and the algorithm proceeds to the next step. A Benders method therefore requires solution of both P_k and R_k, the former to obtain nogoods, and the latter to obtain P_{k+1}.

Logic-based Benders decomposition can be combined with constraint programming in various ways [5, 6, 12, 14, 16, 18, 19, 26]. One type of integration is to solve the subproblem by constraint programming (since it is naturally suited to generate Benders cuts) and the master problem R_k by another. Planning and scheduling problems, for example, have been solved by applying integer programming to R_k (task allocation) and constraint programming to P_k (scheduling) [12, 14, 19, 26]. This approach has produced some of the largest computational speedups available from integrated methods, outperforming conventional solvers by several orders of magnitude.

6 Heuristic Methods

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. Since each neighborhood is the feasible set of a problem restriction, local search can be viewed as a search-infer-and-relax method.

In fact, it is useful to conceive local search as belonging to a family of *local-search-and-relax* algorithms that resemble branch-and-relax algorithms but are inexhaustive (Fig. 4). A number of other heuristic methods, such as GRASPs, belong to the same family. The analogy with branch and relax suggests how inference and relaxation may be incorporated into heuristic methods.

The generic local-search-and-relax algorithm of Fig. 4 "branches" on a problem restriction P_k by creating a further restriction P_{k+1}. For the time being, only one branch is created. Branching continues in this fashion until a restriction is

```
Let $v_{\mathrm{UB}} = \infty$ and $S = \{P_0\}$.
Perform **LocalSearch**.
The best solution found for $P_0$ has value $v_{\mathrm{UB}}$.

Procedure **LocalSearch**.
  If $S$ is nonempty then
    Select a restriction $P =$ from $S$.
    If $P$ is too hard to solve then
      Let $v_{\mathrm{R}}$ be the optimal value of a relaxation of $P$.
      If $v_{\mathrm{R}} < v_{\mathrm{UB}}$ then
        Add a restriction of $P$ to $S$.
        Perform **LocalSearch**.
      Else remove $P$ from $S$.
    Else
      Let $v$ be the value of $P$'s solution and $v_{\mathrm{UB}} = \min\{v, v_{\mathrm{UB}}\}$.
      Remove $P$ from $S$.
```

Fig. 4. *Generic local-search-and-relax algorithm for solving a minimization problem* P_0

created that is easy enough to solve, whereupon the algorithm returns to a previous restriction (perhaps chosen randomly) and resumes branching. There is also a bounding mechanism that is parallel to that of branch-and-relax algorithms.

Local search and GRASPs are special cases of this generic algorithm in which each restriction P_k 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_k's feasible set is a neighborhood of v. P_k 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_k is regarded as too hard to solve.

A pure local search algorithm, such as simulated annealing or tabu search, 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_1 is solved by searching a neighborhood of v. Supposing P_1'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.

In simulated annealing, P_k is "solved" by randomly selecting one or more elements of the neighborhood until one of them, say v', is accepted. A solution v' is accepted with probability 1 if it is better than the currently best solution, and with probability p is it is no better. The probability p may drop (reflecting a lower "temperature") as the search proceeds.

In tabu search, P_k is solved by a complete search of the neighborhood, whereupon the best solution becomes v'. In this case the neighborhood of v' excludes solutions currently on the tabu list.

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 [25]. 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 $P_1, P_2, \ldots$ 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.

A GRASP provides the opportunity to use the bounding mechanism of the local-search-and-relax algorithm, a possibility already pointed out by Prestwich [23]. If a relaxation of P_k has an optimal value that is no better than that of the incumbent solution, then there is nothing to be gained by branching on P_k.

References

1. Aron, I., J. N. Hooker, and T. H. Yunes, SIMPL: A system for integrating optimization techniques, in J.-C. Régin and M. Rueher, eds., *Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR 2004)*, *Lecture Notes in Computer Science* **3011** (2004) 21–36.
2. Benders, J. F., Partitioning procedures for solving mixed-variables programming problems, *Numerische Mathematik* **4** (1962) 238–252.
3. Bliek, C. T., Generalizing partial order and dynamic backtracking, *Proceedings of AAAI* (AAAI Press, 1998) 319–325.
4. Bockmayr, A., and T. Kasper, Branch and infer: A unifying framework for integer and finite domain constraint programming, *INFORMS Journal on Computing* **10** (1998) 287–300.
5. Cambazard, H., P.-E. Hladik, A.-M. Déplanche, N. Jussien, and Y. Trinquet, Decomposition and learning for a hard real time task allocation algorithm, in M. Wallace, ed., *Principles and Practice of Constraint Programming (CP 2004)*, *Lecture Notes in Computer Science* **3258**, Springer (2004).
6. Eremin, A., and M. Wallace, Hybrid Benders decomposition algorithms in constraint logic programming, in T. Walsh, ed., *Principles and Practice of Constraint Programming (CP 2001)*, *Lecture Notes in Computer Science* **2239**, Springer (2001).
7. Geoffrion, A. M., Generalized Benders decomposition, *Journal of Optimization Theory and Applications* **10** (1972) 237–260.
8. Ginsberg, M. L., Dynamic backtracking, *Journal of Artificial Intelli- gence Research* **1** (1993) 25–46.
9. Ginsberg, M. L., and D. A. McAllester, GSAT and dynamic backtrack- ing, *Second Workshop on Principles and Practice of Constraint Pro- gramming (CP1994)* (1994) 216–225.
10. Hooker, J. N., Logic-based methods for optimization, in A. Borning, ed., *Principles and Practice of Constraint Programming*, *Lecture Notes in Computer Science* **874** (1994) 336–349.
11. Hooker, J. N., Constraint satisfaction methods for generating valid cuts, in D. L. Woodruff, ed., *Advances in Computational and Stochasic Optimization, Logic Programming and Heuristic Search*, Kluwer (Dordrecht, 1997) 1–30.
12. Hooker, J. N., *Logic-Based Methods for Optimization: Combining Optimization and Constraint Satisfaction*, John Wiley & Sons (New York, 2000).

13. Hooker, J. N, A framework for integrating solution methods, in H. K. Bhargava and Mong Ye, eds., *Computational Modeling and Problem Solving in the Networked World* (Proceedings of ICS2003), Kluwer (2003) 3–30.
14. Hooker, J. N., A hybrid method for planning and scheduling, in M. Wallace, ed., *Principles and Practice of Constraint Programming (CP 2004)*, *Lecture Notes in Computer Science* **3258**, Springer (2004).
15. Hooker, J. N., and M. A. Osorio, Mixed logical/linear programming, *Discrete Applied Mathematics* **96-97** (1999) 395–442.
16. Hooker, J. N., and G. Ottosson, Logic-based Benders decomposition, *Mathematical Programming* **96** (2003) 33–60.
17. Hooker, J. N., G. Ottosson, E. Thorsteinsson, and Hak-Jin Kim, A scheme for unifying optimization and constraint satisfaction methods, *Knowledge Engineering Review* **15** (2000) 11–30.
18. Hooker, J. N., and Hong Yan, Logic circuit verification by Benders decomposition, in V. Saraswat and P. Van Hentenryck, eds., *Principles and Practice of Constraint Programming: The Newport Papers (CP95)*, MIT Press (Cambridge, MA, 1995) 267–288.
19. Jain, V., and I. E. Grossmann, Algorithms for hybrid MILP/CP models for a class of optimization problems, *INFORMS Journal on Computing* **13** (2001) 258–276.
20. Moskewicz, M. W., C. F. Madigan, Ying Zhao, Lintao Zhang, and S. Malik, Chaff: Engineering an efficient SAT solver, *Proceedings of the 38th Design Automation Conference (DAC'01)* (2001) 530–535.
21. Neumaier, A., Complete search in continuous global optimization and constraint satisfaction, in A. Iserles, ed., *Acta Numerica 2004* (vol. 13), Cambridge University Press (2004).
22. Pinter, J. D., *Applied Global Optimization: Using Integrated Modeling and Solver Environments*, CRC Press, forthcoming.
23. Prestwich, S., Exploiting relaxation in local search, *First International Workshop on Local Search Techniques in Constraint Satisfaction*, 2004.
24. Sahinidis, N. V., and M. Tawarmalani, *Convexification and Global Optimization in Continuous and Mixed-Integer Nonlinear Programming*, Kluwer Academic Publishers (2003).
25. Silva, J. P. M., and K. A. Sakallah, GRASP–A search algorithm for propositional satisfiability, *IEEE Transactions on Computers* **48** (1999) 506–521.
26. Thorsteinsson, E. S., Branch-and-Check: A hybrid framework integrating mixed integer programming and constraint logic programming, T. Walsh, ed., *Principles and Practice of Constraint Programming (CP 2001)*, *Lecture Notes in Computer Science* **2239**, Springer (2001) 16–30.