Toward Unification of Exact and Heuristic Optimization Methods

J. N. Hooker

Carnegie Mellon University
jh38@andrew.cmu.edu

Revised December 2012

Abstract

This paper argues that many exact and heuristic methods have common structure that permits some degree of unification. It interprets solution algorithms as primal-dual methods in which the primal component searches over problem restrictions, and the dual component obtains bounds on the optimal value. In particular, the concept of an inference dual provides the basis for constraint-directed search, which is a feature of many exact and heuristic methods. The motivations for unification are (a) to encourage the exchange of algorithmic techniques between exact and heuristic methods, and (b) to design solution methods that transition gracefully from exact to heuristic modes as problem instances scale up.

1 Introduction

Exact and heuristic methods for optimization are often regarded as belonging to entirely different categories. Exact methods generally take the form of branching and other forms of exhaustive search, while heuristic methods have a very different character, perhaps local search or an imitation of a natural process, such as annealing or biological evolution.

This bifurcation of methods has several drawbacks. It discourages the cross-fertilization of algorithmic ideas developed for exact and heuristic methods. It requires switching to a different solution algorithm when problem instances become too large to solve exactly. Ideally, a single solution method would transition gracefully from exhaustive to inexhaustive search as the problem scales up. Finally, it draws attention away from the underlying unity of exact and heuristic methods.

This paper makes a case that there is, in fact, a fair amount of common structure in exact and heuristic methods. Most algorithms of either type are special cases of a primal-dual solution strategy, although particular algorithms may lack some elements of the strategy. The primal component of the strategy

enumerates problem restrictions in search of a good solution, while the dual component seeks a proof that bounds the optimal value, with the eventual aim of proving the optimality of a solution found by primal search. The dual can be an inference dual or relaxation dual.

Recognizing this commonality has at least three benefits. One is that the structural similarity of exact and heuristic methods may suggest algorithms that can be run in either an exact or heuristic mode. A second benefit is cross-fertilization of algorithmic ideas. When heuristic methods are viewed as having a primal-dual structure, it may become evident how to adapt the inference and relaxation techniques of exact methods to a heuristic method. In particular, the dual perspective may yield a lower bound on the optimal value, which is normally unavailable in heuristic methods. Conversely, when the primal-dual structure of exact methods is recognized, one may see how to adapt the local search ideas of heuristic methods to exhaustive search. For example, tree search with strong branching, a well-known approach to mixed integer programming, can be interpreted as a local search algorithm for solving a dual problem. Local search techniques can then be adapted to obtain alternative branching strategies.

A third benefit is more effective exploitation of problem structure in heuristic methods. A primal-dual method simultaneously solves the primal problem and an inference dual, or the primal problem and a relaxation dual. But it is primarily through inference methods (e.g., constraint propagation and filtering) and relaxation methods (e.g., cutting planes) that exact methods take advantage of problem structure. By viewing a heuristic method as possessing these same elements, similar strategies for exploiting structure may become evident.

Duality is proposed as a unifying principle for exact optimization methods in [28, 31]. In the present paper, this framework is extended to encompass heuristic methods. The structural similarity between some exact and heuristic methods is pointed out in [27], in particular between branching and GRASP, as well as between Benders decomposition and tabu search. These ideas are incorporated here.

The paper begins below with introduction to inference and relaxation duality, followed by a description of primal-dual algorithmic structure. It then identifies this structure in a sampling of exact algorithms, including the simplex method, branch-and-bound-methods, satisfiability solvers, and Benders decomposition. Following this it finds the same structure in such heuristic methods as local search, tabu search, evolutionary algorithms, ant colony optimization, and particle swarm optimization. Along the way, attempts are made to indicate how algorithmic ideas can be exchanged between exact and heuristic methods, and where possible, how a single algorithm can be designed to transition from one to the other.

2 Fundamentals

An optimization problem may be written

$$\min_x \{f(x) \mid x \in S\} \tag{1}$$

where f is the objective function and S the feasible set. Generally, x is a tuple $(x_1, \ldots, x_n)$ of variables. The *epigraph* of (1) is $E = \{(v, x) \mid v \geq f(x),\ x \in S\}$. A *relaxation* of (1) is a problem

$$\min_x \{f'(x) \mid x \in S'\} \tag{2}$$

whose epigraph contains E. If the epigraph of (2) is a subset of E, then (2) is a *restriction* of (1).

2.1 Inference and Relaxation Duals

An *inference dual* [26] seeks the tightest bound on the objective function that can be deduced from the constraint set, using a specified method of logical deduction. So an inference dual of (1) is

$$\max_{v,P} \left\{ v \;\middle|\; (x \in S) \overset{P}{\Rightarrow} (f(x) \geq v),\ P \in \mathcal{P} \right\} \tag{3}$$

where $\mathcal{P}$ is a specified proof family. The notation $A \overset{P}{\Rightarrow} B$ means that proof P deduces B from A. The inference dual clearly satisfies *weak duality*, meaning that $f(x) \geq v$ for any feasible x in the primal and any feasible (v, P) in the dual. *Strong duality* holds when the primal and dual have the same optimal value. By convention, the optimal value of a minimization problem is ∞ when it is infeasible and $-\infty$ when it is unbounded, and vice-versa for a maximization problem.

As an example, suppose (1) is a linear programming (LP) problem

$$\min_x \{cx \mid Ax \geq b,\ x \geq 0\} \tag{4}$$

The inference dual is

$$\max_P \left\{ v \;\middle|\; (Ax \geq b,\ x \geq 0) \overset{P}{\Rightarrow} (cx \geq v),\ P \in \mathcal{P} \right\} \tag{5}$$

The classical LP dual is obtained when $\mathcal{P}$ is defined as follows. Let a *surrogate* of $Ax \geq b$ be a nonnegative linear combination $uAx \geq ub$. Then u can be regarded as encoding a proof of $cx \geq v$ when $uAx \geq ub$ dominates $cx \geq v$; that is, when $uA \leq c$ and $ub \geq v$, or no $x \geq 0$ satisfies $uAx \leq ub$. If $\mathcal{P}$ consists of all $u \geq 0$, the inference dual (5) becomes

$$\max_{u,v} \{v \mid uA \leq c,\ ub \geq v,\ u \geq 0\}$$

for a feasible LP. This is equivalent to $\max_u \{ub \mid uA \le c,\ u \ge 0\}$, which is the classical LP dual. Inference LP duality is strong, while classical LP duality is strong under a constraint qualification (i.e., the primal or classical dual is feasible).

It is shown in [29] that the Lagrangean dual, surrogate dual, and subadditive dual are inference duals for appropriate definitions of $\mathcal{P}$. Additional inference duals are defined below in the context of algorithms in which they play a central role.

A *relaxation dual* is defined over a family of parameterized relaxations. It seeks the relaxation whose optimal value provides the tightest possible lower bound on the optimal value of the original problem. So a relaxation dual of (1) can be written

$$\max_u \{\, \theta(u) \mid\ u \in U\} \tag{6}$$

where

$$\theta(u) = \min_x \{ f'(x,u)\ \mid x \in S'(u)\,\} \tag{7}$$

Problem (7) is a relaxation of (1) parameterized by $u \in U$, which means that for any $u \in U$, $S'(u) \supset S$ and $f'(x,u) \le f(x)$ for all $x \in S$. The relaxation dual satisfies weak duality because $\theta(u)$ is the minimum value of some relaxation of (1) and is therefore a lower bound on $f(x)$ for any $x \in S$.

The classical LP dual is a relaxation dual as well as an inference dual for a feasible LP. We let the parameterized relaxation minimize cx over a surrogate of $Ax \ge b$. The resulting dual is $\max\{\theta(u) \mid u \ge 0\}$ where

$$\theta(u) = \min\{cx \mid uAx \ge ub,\ x \ge 0\}$$

The Farkas Lemma implies that the relaxation dual is equivalent to the classical LP dual if the LP is feasible. The Lagrangean, surrogate, and subadditive duals are likewise relaxation duals.

2.2 The Algorithmic Framework

A general algorithm for combinatorial optimization attacks the primal problem by enumerating restrictions of it. Branching methods, for example, create restrictions by branching on variables. Each node of the search tree corresponds to a restriction of the original problem. The motivation for creating restrictions is that they may be easier to solve than the original problem. If the enumeration is exhaustive, the smallest optimal value of a restriction is the optimal value of the original problem.

Heuristic methods often enumerate restrictions by local search; that is, by examining neighborhoods defined around a sequence of solutions. The neighborhood is the feasible set of a restriction and should be small enough to search easily. The restriction is solved by selecting an attractive solution in the neighborhood to serve as the center of the next neighborhood. Local search is, of course, rarely exhaustive.

A *primal-dual* algorithm seeks lower bounds on the optimal value while enumerating restrictions. It obtains bounds by (partially) solving an inference dual or a relaxation dual. A lower bound can reduce enumeration, because one may wish to terminate the algorithm after finding a restriction whose optimal value is close to the lower bound.

A key issue is how restrictive the restrictions should be. Tighter restrictions fix more variables and have smaller feasible sets. They may be easier to solve, but there are many to enumerate, and most have poor solutions. Looser restrictions have larger feasible sets and therefore better optimal values, but they may be hard to solve. One general strategy is to tighten the restrictions until they become easy to solve. This occurs in branching methods, which branch until the problems at leaf nodes can be solved. The opposite strategy is to enlarge the feasible set until the solution quality improves, but not so much that the restrictions become intractable, as is commonly done in large neighborhood search methods. In some cases, the fixed variables are selected in advance, as in Benders decomposition.

An important feature of primal-dual methods is the use of inference and relaxation to guide the search over restrictions, as well as to obtain bounds. The inference dual provides the basis for *constraint-directed search*, a very broad class of search methods that include branching, Benders decomposition, dynamic backtracking, and tabu search. Suppose that an algorithm enumerates restrictions by fixing a tuple y^k of variables in x to $\bar{y}^k$ in each iteration k. So $x = (y^k, z^k)$, where z^k is the tuple of variables in x that are not fixed. This defines a subproblem of (1):

$$\min_{z^k} \left\{ f(\bar{y}^k, z^k) \;\middle|\; (\bar{y}^k, z^k) \in S \right\} \tag{8}$$

The inference dual of (8) is

$$\max_{v,P} \left\{ v \;\middle|\; ((\bar{y}^k, z^k) \in S) \overset{P}{\Rightarrow} (f(\bar{y}^k, z^k) \geq v),\ P \in \mathcal{P} \right\} \tag{9}$$

If (v^*, P^*) is an optimal solution of (9), then P^* deduces the lower bound v^* when y^k is fixed to $\bar{y}^k$. This same proof P^* may deduce a useful bound $LB_k(y^k)$ for general y^k, where in particular $LB_k(\bar{y}^k) = v^*$. This means that if a future iteration fixes the variables in y^k, the resulting optimal value (after solving the subproblem) cannot be better than the $LB_k(y^k)$. The next partial solution y^{k+1} cannot result in a value better than

$$\max_{\ell=1,\ldots,k} \left\{ LB_\ell(y^\ell) \mid y^\ell \subset y^{k+1} \right\} \tag{10}$$

where $y^\ell \subset y^{k+1}$ means that all the variables in tuple y^ℓ occur in tuple y^{k+1}. There is no need to consider a value of y^{k+1} for which the bound in (10) is no better than the best value found so far. This is the principle behind Benders decomposition, for example.

An important special case occurs when the subproblem (8) is a feasibility problem with no objective function. In this case, the solution P^* of the inference

dual is a proof of infeasibility. This same proof may deduce infeasibility for values of y^k other than $\bar{y}^k$. The bound $LB_k(y^k)$ therefore becomes a *nogood constraint* $N_k(y^k)$, which is violated by any value of y^k for which P^* proves infeasibility in the subproblem. In particular, $y^k = \bar{y}^k$ violates $N_k(y^k)$. The next partial solution y^{k+1} cannot result in a feasible subproblem unless it satisfies the constraints in

$$\left\{ N_\ell(y^\ell) \mid y^\ell \subset y^{k+1},\ \ell = 1, \dots, k \right\}$$

This is the principle behind dynamic backtracking, clause learning in satisfiability algorithms, and tabu search. It is convenient to refer to a bound $LB_k(y^k)$ as a *nogood bound*.

Relaxation, as well as inference, can also guide the search over problem restrictions. Typically, a relaxation (or relaxation dual) of the current restriction is solved when the restriction itself is too hard to solve. This is done at the nodes of a branch-and-bound tree, for example. If the solution of the relaxation is feasible for the restriction, then the restriction itself is solved. Otherwise, the solution may provide a clue as to how the restriction should be tightened to obtain an easier problem. For example, if an LP relaxation of an integer programming problem is solved, the search can branch on variables that take a fractional value in the solution of the relaxation.

3 Exact Methods

3.1 Simplex Method

The simplex method for an LP problem (4) enumerates problem restrictions while solving the LP dual of each restriction to obtain a bound. It can be interpreted as a local search as well as constraint-directed search.

In each iteration of the simplex method, the variables x are partitioned into basic and nonbasic variables, $x = (x_B, x_N)$, with a corresponding partition $A = [B\ N]$ of the coefficient matrix. A restriction is created by fixing $x_{N'} = 0$, where $x_{N'}$ consists of all the nonbasic variables except a variable x_j that will enter the basis. It minimizes $cx = c_B x_B + c_N x_N = c_B x_B + c_j x_j$ subject to the requirement that x lie on an edge of the feasible polyhedron:

$$\min_{x_B, x_j} \left\{ c_B x_B + c_j x_j \;\middle|\; x_B + B^{-1} A_j x_j = B^{-1} b,\ x_B, x_j \geq 0 \right\} \tag{11}$$

where A_j is column j of A. The feasible set of (11) can be regarded as a neighborhood of the current solution $(x_B, x_N) = (B^{-1}b, 0)$, and the simplex method can therefore be seen as a local search method (Fig 1). The restriction (11) is solved by moving to the opposite end of the edge, so that

$$x_j = \min_i \left\{ \frac{(B^{-1}b)_i}{(B^{-1}A_j)_i} \;\middle|\; (B^{-1}A_j)_i > 0 \right\} = \frac{(B^{-1}b)_{i'}}{(B^{-1}A_j)_{i'}}$$

where i' is the minimizing value of i.

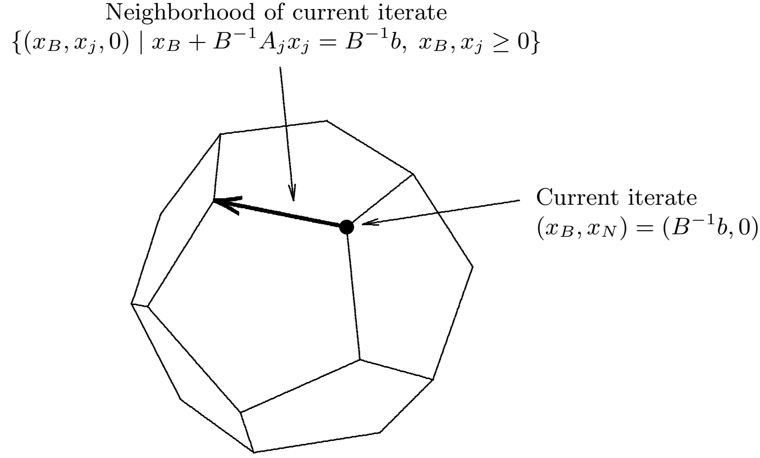

Figure 1: Simplex method as local search.

The dual analysis is somewhat different from the textbook version but allows the simplex method to be interpreted as constraint-directed search. We first write the subproblem (11) as

$$\begin{aligned}\min_{x_B, x_j} \{&c_B x_B + c_j x_j + c_{N'} \bar{x}_{N'} \mid \\ &x_B + B^{-1}A_j x_j + B^{-1}N' \bar{x}_{N'} = B^{-1}b;\ x_B, x_j \geq 0\}\end{aligned} \tag{12}$$

by inserting the nonbasic variables $x_{N'}$ and fixing them to a constant $\bar{x}_{N'}$ that is currently zero. The LP dual of (12) is

$$\max_u \left\{ uB^{-1}b + (c_{N'} - uB^{-1}N')\bar{x}_{N'} \;\middle|\; u \leq c_B,\ uB^{-1}A_j \leq c_j \right\} \tag{13}$$

Because the first constraint is tight for $i \neq i'$, the dual solution u satisfies $u_i = c_i$ for $i \neq i'$ and $u_{i'} \leq c_{i'}$. Due to weak duality, the dual solution yields a valid nogood bound

$$LB(x_{N'}) = uB^{-1}b + (c_{N'} - uB^{-1}N')x_{N'}$$

even when $x_{N'}$ is not fixed to zero. So when we define the next neighborhood by selecting a variable x_k to release from zero, we cannot improve on the current solution value $c_B B^{-1}b$ unless

$$c_k - uB^{-1}A_k = c_k - \sum_{i \neq i'} c_i (B^{-1}A_k)_i - u_{i'}(B^{-1}A_k)_{i'} < 0$$

Because $u_{i'} \leq c_{i'}$, it suffices that $c_k - c_B B^{-1}A_k < 0$; that is, the reduced cost is negative. We therefore select an x_k with negative reduced cost. It is in this manner that the inference dual of the restricted problem guides the simplex method.

3.2 Branch and Bound as Constraint-Directed Search

Like the simplex method, branch-and-bound methods for discrete optimization are local search methods that use constraint-directed search. Unlike the simplex method, the local search mechanism solves a relaxation dual. Both interpretations can lead to significant improvements in the classical branch-and-bound algorithm.

We first develop the constraint-directed search perspective. To simplify discussion, suppose for the moment that we seek only a feasible solution. At each node of the search tree we branch on a variable x_j by creating a child node for each possible value of x_j. A node on level k is reached by setting $(x_1, \ldots, x_k)$ to some set of values $(v_1, \ldots, v_k)$, which defines a restriction of the original problem. The tree search can be interpreted as enumerating problem restrictions that correspond to the leaf nodes.

To view the search as constraint-directed, suppose we find that a leaf node subproblem is infeasible, perhaps by trying to solve its LP relaxation. At this point, we backtrack. An alternate interpretation of backtracking is that we create a nogood constraint $(x_1, \ldots, x_k) \neq (v_1, \ldots, v_k)$. We then generate the next node of the tree by assigning values to variables $x_1, x_2, \ldots$ that satisfy all the nogood constraints created so far, continuing until the subproblem is solved or proved infeasible. The nogood constraints ensure that we do not examine the same leaf node twice. If the subproblem is solved, we are done. Otherwise, we continue generating leaf nodes in this fashion until no assignment satisfies all of the nogood constraints, at which point the search is exhaustive. Any search tree can be built in this way.

So far, the inference dual does not play a role in the creation of nogood constraints. We may be able strengthen the nogoods, however, by examining the proof of infeasibility that solves the inference dual of the subproblem or its relaxation. We may find, for example, that only some of the variable settings appear as premises of the proof, perhaps $(x_{j_1}, \ldots, x_{j_p}) = (v_{j_1}, \ldots, v_{j_p})$. Then we can create a strengthened nogood constraint $(x_{j_1}, \ldots, x_{j_p}) \neq (v_{j_1}, \ldots, v_{j_p})$. This may allow us to skip an entire section of the branching tree. This is known as *backjumping*, a special case of *dynamic backtracking* [18, 19, 38]. It is the basis for clause learning in satisfiability algorithms, discussed in Section 3.4 below.

Constraint-directed search can be specialized to integer programming [26], where it is sometimes referred to as *conflict analysis*. Suppose we wish to solve the problem

$$\min_{x \in \mathbb{Z}^n} \{cx \,| Ax \geq b,\ x \geq 0\} \tag{14}$$

The LP relaxation of the subproblem at a given leaf node is

$$\min_{x} \{cx \mid Ax \geq b,\ p \leq x \leq q\} \tag{15}$$

where p and q are lower and upper bounds imposed by branching (Fig. 2). If $p_j = q_j$, the variable x_j has been fixed. We associate dual variables u, σ, τ with the three constraints in (15). If the subproblem relaxation (15) is infeasible, its LP dual is solved by a tuple (u, σ, τ) that proves infeasibility because

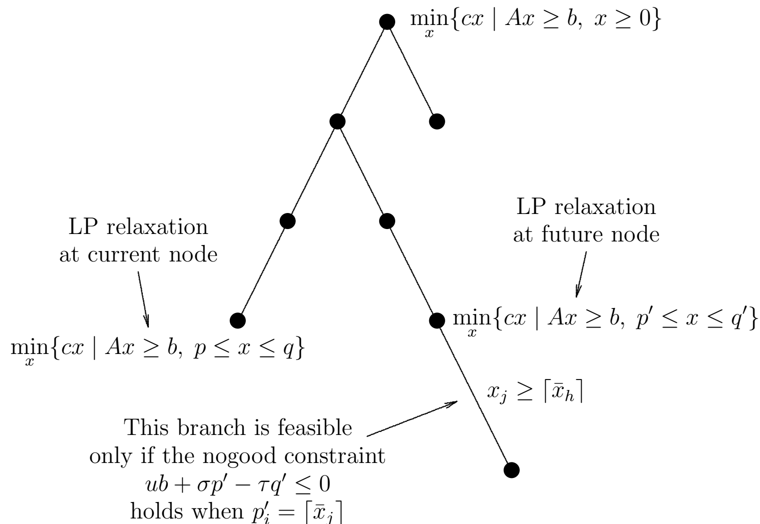

Figure 2: Conflict analysis in a branch-and-bound search.

$uA + \sigma - \tau \leq 0$ and $ub + \sigma p - \tau q > 0$. This can guide future branching. If p, q are the branching bounds at a subsequent node of the tree, the subproblem at that node can be feasible only if

$$ub + \sigma p - \tau q \leq 0 \tag{16}$$

This can be treated as a nogood constraint. Suppose we wish to branch on a variable x_j with a fractional value $\hat{x}_j$ in the solution of (15). A left branch defined by $x_j \leq \lfloor \hat{x}_j \rfloor$ is feasible only if setting $q_j = \lfloor \hat{x}_j \rfloor$ satisfies (16), which is to say

$$\lfloor \hat{x}_j \rfloor \geq \sigma p + ub - \sum_{i \neq j} \tau_i q_i \tag{17}$$

So the left branch can be eliminated if this inequality is violated, and analogously for the right branch. This test can be performed repeatedly at a given node by substituting into (17) the dual solution (u, σ, τ) obtained at each infeasible leaf node encountered so far. The idea can be extended to feasible leaf nodes and strengthened in various ways [31].

This type of bounding enables backjumping in an integer programming search tree. Although backjumping an old idea [16, 48], constraint-directed branching has only recently been implemented in an integer programming solver [1, 44].

3.3 Branch and Bound as Dual Local Search

The generation of a branch-and-bound tree can be regarded as a local search algorithm for solving a relaxation dual. This opens the possibility of applying

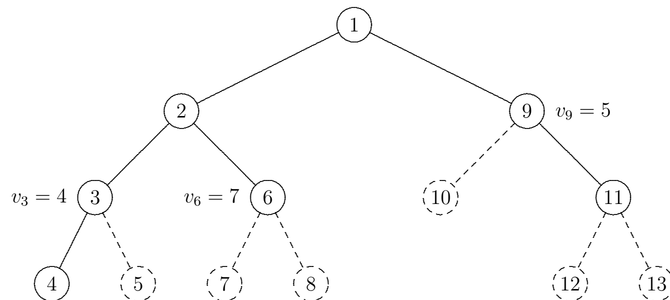

Figure 3: Calculation of bound in incomplete branch-and-bound tree (dashed portion is unexplored).

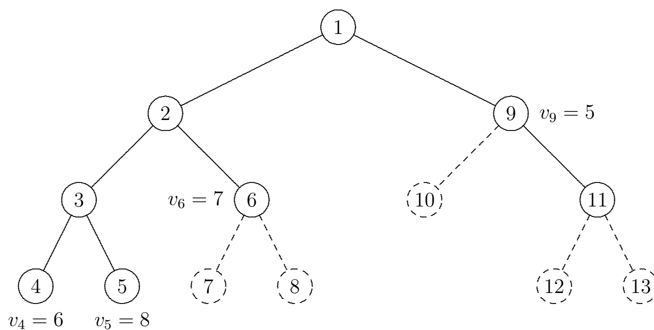

Figure 4: Recalculation of bound after adding node 5.

local search ideas to define branching strategies. It also suggests how a branch-and-bound method can gracefully transition to a heuristic method.

Any complete or incomplete search tree T yields a lower bound $\theta(T)$ on the optimal value of the original problem. The bound $\theta(T)$ is the minimum of v_i^* over all leaf nodes and incomplete nodes i, where v_i^* is the optimal value of the relaxation at node i. An incomplete node is a nonleaf node at which at least one child is missing from T. The minimum can be taken only over leaf nodes and incomplete nodes that do not descend from another incomplete node. For example, Fig. 3 shows a search tree in which only the solid portion has been explored. It proves the bound $\min\{v_3^*, v_6^*, v_9^*\} = 4$.

The tree variable T can be viewed as parameterizing a family of relaxations with optimal value $\theta(T)$. The problem of finding the best possible bound is the relaxation dual $\max_T\{\theta(T)\}$, where T ranges over all complete and incomplete search trees. There is no duality gap if the relaxation dual is solved to optimality, because $\theta(T)$ is the optimal value of the original problem for any complete tree T.

Branching trees are built by adding one child node at a time. This in effect defines a neighborhood of each incomplete tree T that consists of trees obtained

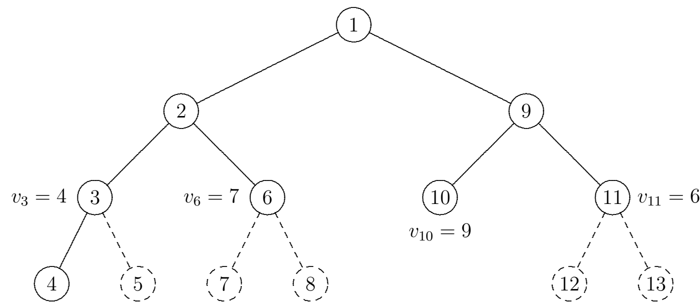

Figure 5: Recalculation of bound after adding node 10.

by adding one child node to some node of T. The choice of which node to add is therefore a choice of which element of the neighborhood to select. Because adding a node never results in a weaker bound, local search in this context always results in monotone nondecreasing bounds. In a discrete optimization problem, a local search converges finitely to the best possible bound. For example, adding any of the unexplored nodes in Fig. 3 creates a neighboring tree.

One well-known strategy for node selection is strong branching [8], which adds to T the node at which the relaxation provides the largest bound. (The relaxation value may be estimated by using pseudocosts [7, 17].) This is in effect a strategy for selecting a tree in the neighborhood of T. There are, however, other ways to design a local search, which result in different and perhaps novel node selection rules. For example, one can use uphill search by selecting the neighboring tree T' that maximizes $\theta(T')$. This corresponds to none of the traditional node selection rules, including strong branching, depth-first search, breadth-first search, iterative deepening, and limited discrepancy search.

Uphill search can be used in Fig. 3. Adding node 5 proves the bound $\max\{v_4^*, v_6^*, v_{10}^*, v_{11}^*\} = 5$ as shown in Fig. 4, which is an improvement. Adding node 10 replaces $v_9^* = 5$ with two tigher bounds as in Fig. 5, but the overall minimum of 4 does not improve. Enumeration of all neighboring trees reveals that the tree of Fig. 4 yields the greatest improvement in the bound.

One could also use *simulated annealing* [36] in this context. This is a local search algorithm in which a tree T' is randomly selected from the neighborhood N of T. If $\theta(T') < \theta(T)$, tree T' is accepted, and the process repeats in a neighborhood N' of T'. If $\theta(T') \geq \theta(T)$, then T' is nonetheless accepted with probability p, and the process repeats in neighborhood N'. If T' is rejected, another tree T' is randomly selected from N. The probability p may decrease according to a "cooling schedule" that mimics an annealing process toward a minimum-energy state in materials. A simulated annealing approach relieves the computational burden of evaluating $\theta(T')$ for all neighboring trees T'; that is, computing the relaxation value at all nodes that could be added to T. It may examine a larger number of trees, however, before a suitable lower bound is obtained.

A tree-building strategy designed to solve the relaxation dual is not

necessarily a good strategy for finding feasible solutions. For example, an uphill search tends to add shallow nodes to the tree, which are not likely to yield feasible solutions. However, this is a fundamental difficulty of using the same tree search to find feasible solutions and to prove optimality; that is, using the same tree search to solve the primal and the relaxation dual. One escape from this dilemma is to use a primal heuristic that finds feasible solutions in parallel with a branching tree that finds bounds. Primal heuristics have in fact become a standard feature of branch-and-bound solvers.

The use of a primal heuristic allows one to design a local search method that is best suited for the relaxation dual. By running the two heuristics in parallel, the optimal value can be bracketed by increasingly tight upper bounds from the primal heuristic and lower bounds from the dual heuristic. The dual heuristic can also contribute upper bounds when the solution of the relaxation at a leaf node is integral. An exact solution is obtained when the tightest upper and lower bounds coincide, which in the worst case will occur when the dual heuristic runs to completion. One could also design a primal algorithm that becomes exhaustive search if it runs long enough, such as iterative deepening or limited discrepancy search. The branch-and-bound mechanism becomes a heuristic method simply by terminating the primal and dual algorithms when acceptably tight bounds are obtained.

3.4 Clause Learning for Satisfiability Solvers

Clause learning is essential to the success of state-of-the-art satisfiability solvers [4, 22, 39]. It is a straightforward case of constraint-directed search that relies on the inference dual.

The satisfiability problem seeks a feasible solution for a set of logical clauses. A clause is a disjunction of literals, each of which has the form x_j or $\neg x_j$, where x_j is a boolean variable and $\neg$ is negation. So a clause has the form

$$\bigvee_{j \in P} x_j \vee \bigvee_{j \in N} \neg x_j$$

Most state-of-the-art solvers use some variant of the Davis-Putnam-Loveland-Logemann (DPLL) algorithm [11, 12], which is a simple branching scheme. The algorithm branches at each nonleaf node by setting some variable x_j to true and to false, which adds the unit clause x_j or $\neg x_j$ to the clause set at the child nodes. To simplify exposition, we suppose that the subproblem at each node contains all the original clauses and all unit clauses added by branching.

Before branching at each node, *unit resolution* is applied to the clause set at that node. Unit resolution is an inference method that operates by removing unit clauses. Thus if the clause set contains a unit clause consisting of the literal x_j, it concludes that x_j must be true, deletes all clauses that contain x_j (because they are satisfied), and deletes the literal $\neg x_j$ wherever it occurs (because it cannot be true). A unit clause $\neg x_j$ is treated analogously. The process continues until no unit clauses remain. If unit resolution generates the

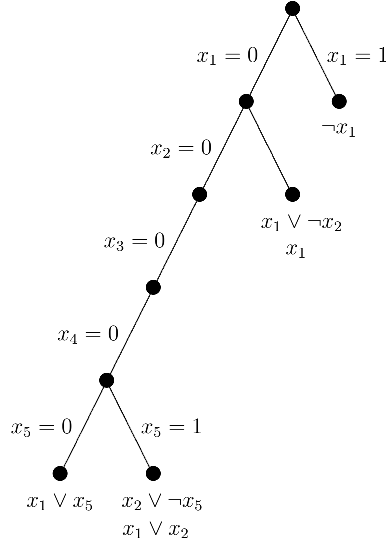

Figure 6: DPLL search tree with backjumping.

empty clause, the clause set is unsatisfiable, and the search backtracks. If no clauses remain after unit resolution, the clause set is satisfied, and the search terminates. If neither occurs, the search branches on a variable that remains in the clause set.

If the subproblem at a node is unsatisfiable, the unit resolution proof of the empty clause solves an inference dual—namely, the inference dual in which the inference method is unit resolution. We can formulate a nogood constraint by identifying the branching-induced unit clauses that serve as premises in the proof. For example, Fig. 6 shows a branching tree for the satisfiability problem below:

$$\begin{array}{lll} x_1 \vee x_5 \vee x_6 & x_2 \vee \neg x_5 \vee \neg x_6 & \neg x_1 \vee \neg x_3 \\ x_1 \vee x_5 \vee \neg x_6 & \neg x_1 \vee x_3 \vee x_4 & \neg x_1 \vee \neg x_4 \\ x_2 \vee \neg x_5 \vee x_6 & \neg x_2 \vee x_3 \vee x_4 & \neg x_2 \vee \neg x_3 \\ & & \neg x_2 \vee \neg x_4 \end{array}$$

Fixing $(x_1, \ldots, x_5) = (0, 0, 0, 0, 0)$ yields a unit resolution proof of infeasibility in which $x_1 = 0$ and $x_5 = 0$ serve as premises. We therefore generate the nogood constraint $x_1 \vee x_5$, which in this context is known as a *conflict clause*. Backtracking, we fix $(x_1, \ldots, x_5) = (0, 0, 0, 0, 1)$ and obtain the conflict clause $x_2 \vee \neg x_5$, which combines with the first conflict clause to obtain the resolvent $x_1 \vee x_2$. We must now backjump to level 2 of the tree to satisfy this conflict clause, which accelerates the search. Setting $(x_1, x_2) = (0, 1)$ already creates infeasibility and the conflict clause $x_1 \vee \neg x_2$, which resolves with $x_1 \vee x_2$ to yield x_1. Only one more solution $x_1 = 1$ is possible, which creates infeasibility and the conflict clause $\neg x_1$. This resolves with x_1 to obtain the empty clause,

which means that no more solutions satisfy the conflict clauses, and the search is complete. In practice, conflict clauses are rapidly identified by finding certain types of cuts in an *implication graph* [22], but this process corresponds to constructing a unit resolution proof of infeasibility.

Note that next leaf node can always be generated by assigning values to variables so as to satisfy the conflict clauses so far generated, until unit resolution detects infeasibility or satisfaction. In fact, the branching framework can be dropped altogether, and the algorithm can proceed purely as constraint-directed search. However, because the conflict clauses are clauses, they can simply be added to the constraint set, which allows unit resolution to detect infeasibility sooner. Some solvers that operate along these lines may nonetheless abandon the search tree framework and proceed in a mode similar to pure constraint-directed search.

A DPLL algorithm with clause learning is a special case of dynamic backtracking. A slight generalization of this process results in partial-order dynamic backtracking [9, 38], which is also a form of constraint-directed search. The main point here is that DPLL with clause learning has basically the same algorithmic structure as the simplex method, branch-and-bound with conflict analysis, and—as we see next—Benders decomposition.

3.5 Benders Decomposition

Benders decomposition [6] is constraint-directed search in which the inference dual is the LP dual, and the nogood constraints are known as *Benders cuts.* It is distinguished by the fact that the subproblem is an LP and always contains the same variables, These variables are carefully chosen to define a subproblem that is easy to solve.

Once Benders decomposition is seen as a special case of constraint-directed search, one can substitute different kinds of inference duals for the LP dual and obtain Benders cuts for a wide variety of subproblems, not just LPs. The resulting method might be called *logic-based* Benders decomposition [26, 32, 33], because Benders cuts are based on the logical form of the optimality proof that solves the dual. The method has been applied to a wide variety of problems, often with substantial improvement over the previous state of the art (surveyed in [31]). We will see in Section 4.3 that tabu search has a similar structure, and this may allow importation of Benders techniques into a local search framework.

Logic-based Benders decomposition begins with a partition of the variables $x = (y, z)$, where z is the tuple of variables that appear in the subproblem. The problem therefore has the form

$$\min_{y,z} \{ f(y,z) \mid (y,z) \in S \} \tag{18}$$

Like branching, Benders decomposition conducts a search over problem restrictions, but it defines a restriction by always fixing the same variables y. If y is fixed to $\bar{y}$, we have the subproblem

$$\min_{z} \{ f(\bar{y},z) \mid (\bar{y},z) \in S \} \tag{19}$$

The inference dual is

$$\max_{v,P} \left\{ v \;\middle|\; ((\bar{y}, z) \in S) \overset{P}{\Rightarrow} (f(\bar{y}, z) \geq v),\ P \in \mathcal{P} \right\} \tag{20}$$

If (v_k, P_k) is an optimal solution of (20) in iteration k, the proof P_k may deduce a bound $LB_k(y)$ for general y. We therefore create the nogood bound (Benders cut) $v \geq LB_k(y)$ and add it to the pool of nogood bounds from previous iterations. To select the next value of y, we minimize v subject to all the Benders cuts. That is, we solve the *master problem*

$$\min_{y,v} \{ v \mid v \geq LB_\ell(y),\ \ell = 1, \ldots, k \} \tag{21}$$

The solution $\bar{y}$ of the master problem defines the next subproblem. The algorithm terminates when the optimal value of the master problem equals $\min\{v_1, \ldots, v_k\}$.

The classical Benders method is applied to a problem of the form

$$\min_{y,z} \{ f(y) + cz \mid g(y) + Az \geq b,\ y \in S,\ z \geq 0 \} \tag{22}$$

which has the subproblem

$$\min_{z} \{ f(\bar{y}) + cz \mid Az \geq b - g(\bar{y}), z \geq 0 \} \tag{23}$$

The LP dual of the subproblem is

$$\max_{u} \{ f(\bar{y}) + u(b - g(\bar{y})) \mid uA \leq c,\ u \geq 0 \}$$

Assuming the subproblem (23) is feasible, the dual solution u^k in iteration k proves a nogood bound on v for any y:

$$v \geq \mathrm{LB}_k(y) = f(y) + u^k(b - g(y))$$

This is the classical Benders cut. When (23) is infeasible, the dual solution u^k proves infeasibility because $u^k A \leq 0$ and $u^k(b - g(\bar{y}) > 0$. So the next solution y can be feasible only if

$$u^k(b - g(y)) \leq 0$$

which becomes the Benders cut.

Algorithms having this same structure can solve a wide variety of problems. To use an example from [30], suppose that jobs $1, \ldots, n$ can be processed in any of shops $1, \ldots, m$. We wish to assign jobs to shops, and schedule the jobs in each shop, so as to minimize makespan. Job j has processing time p_{ij} in shop i, consumes resources at rate c_{ij}, and has deadline d_j. Shop i has available resources C_i. Suppose, for example, if jobs 1, 2 and 3 are assigned to a shop 1 with maximum resources $C_1 = 4$, where the processing time and resource consumption of each job are illustrated in Fig. 7. The time windows are $[E_j, L_j]$ are given in the figure. The minimum makespan schedule for this

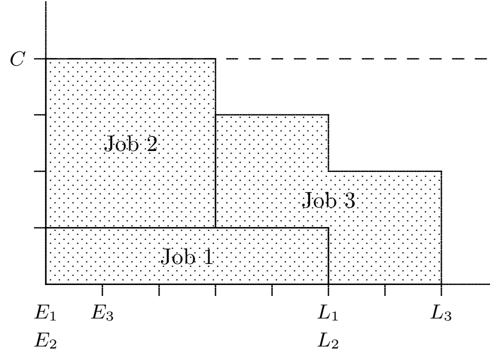

Figure 7: Cumulative scheduling subproblem for logic-based Benders decomposition.

shop is shown in the figure, with makespan 7. Similar schedules are computed for jobs assigned to other shops.

To formlate the problem in general, let binary variable $y_{ij} = 1$ if job j is assigned to shop i, and let z_j be the start time of job j. The problem is

$$\min_{M,y,z} \left\{ M \;\middle|\; \begin{array}{l} \left\{ \begin{array}{c} M \geq z_j + \sum_i p_{ij} y_{ij} \\ 0 \leq z_j \leq d_j - \sum_i p_{ij} y_{ij} \end{array} \right\}, \; j = 1, \ldots, n \\ \text{cumulative}(z^{yi}, p^{yi}, c^{yi}, C_i), \; i = 1, \ldots, m \end{array} \right\}$$

The first constraint defines the makespan M, and the second constraint enforces the deadlines. The cumulative constraint requires that jobs assigned to shop i be scheduled so that the total rate of resource consumption does not exceed C_i at any time. The notation z^{yi} denotes the tuple of start times z_j for all jobs j that y assigns to shop i, and similarly for p^{yi} and c^{yi}.

The subproblem is not an LP, but logic-based Benders cuts can nonetheless be derived from the inference dual. For a fixed assignment $\bar{y}$, the subproblem schedules the jobs to minimize makespan M_i in each shop i:

$$\min_{M_i,z} \left\{ M_i \;\middle|\; \begin{array}{c} \left\{ \begin{array}{c} M_i \geq z_j + p_{ij} \\ 0 \leq z_j \leq d_j - p_{ij} \end{array} \right\}, \text{ all } j \text{ with } \bar{y}_{ij} = 1 \\ \text{cumulative}(z^{\bar{y}i}, p^{\bar{y}i}, c^{\bar{y}i}, C_i) \end{array} \right\}, \; i = 1, \ldots, m$$

In each iteration k, each subproblem i can be solved by, say, constraint programming to obtain the minimum makespan M_{ik} for that shop. The proof of optimality solves the inference dual and provides the basis for several kinds of Benders cuts. At a minimum, one can observe the set J_{ik} of jobs that play a role in the proof and write the logic-based Benders cut

$$M \geq LB_{ik}(y) = M_{ik} \sum_{j \in J_{ik}} (1 - y_{ij}) \tag{24}$$

for each i. If all jobs assigned to a shop have the same release time, the cut can be strengthened to

$$M \geq M_{ik} - \left(\sum_{j \in J_{ik}} p_{ij}(1 - y_{ij}) + \max_{j \in J_{ik}} \{d_j\} - \min_{j \in J_{ik}} \{d_j\} \right)$$

The master problem is

$$\min_{M,y} \{M \mid M \geq LB_{i\ell}(y),\ \ell = 1, \ldots, k,\ i = 1, \ldots, m; \}$$

In the example of Fig. 7, the Benders cut for shop 1 is

$$M \geq 7 \sum_{j \in J_{1k}} (1 - y_{1j})$$

where $J_{1k} = \{1, 2, 3\}$. Analysis of the optimality proof (inference dual) for shop 1 reveals that only jobs 2 and 3 play a role in the proof. We can therefore strengthen the cut by letting $J_{1k} = \{2, 3\}$.

This approach can reduce solution time by several orders of magnitude relative to mixed integer and constraint programming solvers [30, 49]. It can be converted to a heuristic method simply by terminating it when the optimal value of the master problem is close to the best subproblem makespan obtained so far. Another option is to drop the oldest Benders cuts so that the number of cuts never exceeds a limit, much as is done in tabu search. The master problem therefore plays the role of a tabu list, as discussed below in Section 4.3.

4 Heuristic Methods

4.1 Local Search

Local search enumerates problem restrictions by examining neighborhoods of trial solutions. The neighborhood can be viewed as the feasible set of a restriction, which is solved by evaluating some or all of the solutions in the neighborhood. The next trail solution s elected on this basis of this evaluation, and a new neighborhood is defined for it.

Neighborhoods are typically defined by making small changes in the current solution, perhaps by swapping elements of the solution. More radical alterations are allowed in *very large neighborhood search* [2, 41], perhaps multiple swaps. In *variable-depth neighborhood search*, the size of the neighborhood is adjusted to that the subproblem is tractable but is general enough to have a good solution.

When neighborhoods are seen as problem restrictions, one can apply methods used in exact methods for creating problem restrictions. For example, one can fix some of the variables, and let the neighborhood represent the subproblem over the remaining variables. The size of the neighborhood can be adjusted by increasing the number of fixed variables until the subproblem becomes soluble, as is done in branching.

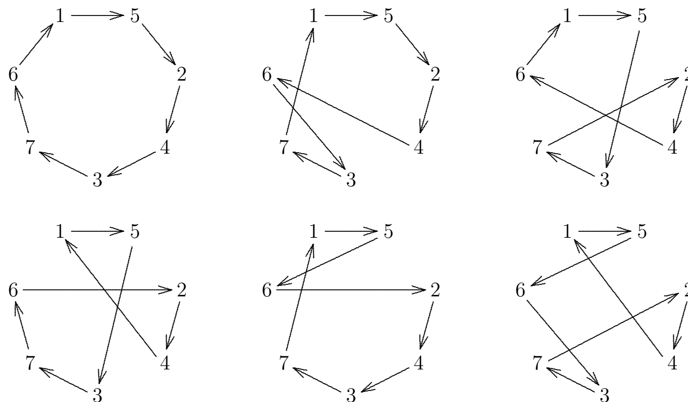

Figure 8: The tour $x = (5, 4, 7, 3, 2, 1, 6)$ (upper left) and 5 neighbors when (x_1, x_2, x_3) is fixed to $(5, 4, 7)$.

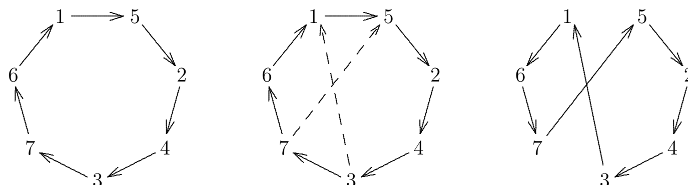

Figure 9: Obtaining one 2-opt neighbor of the tour on the left. There are 13 other neighbors.

The idea can be illustrated with the traveling salesman problem (TSP). Let $x_i \in \{1, \ldots, n\}$ be the city that immediately follows city i, and let c_{ij} be the distance from city i to city j. The TSP can be written

$$\min_x \left\{ \sum_i c_{ix_i} \;\middle|\; \text{circuit}(x_1, \ldots, x_n) \right\}$$

where the circuit constraint requires that $x_1, \ldots, x_n$ describe a Hamiltonian cycle. We can define a neighborhood of the current solution $\bar{x}$ by fixing some of the variables $(x_1, \ldots, x_k)$ to $(\bar{x}_1, \ldots, \bar{x}_k)$ and solving for a minimum cost tour over $(x_{k+1}, \ldots, x_n)$. A neighborhood of this sort is formed by reconnecting subchains. Suppose, for example, that $(\bar{x}_1, \ldots, \bar{x}_7) = (5, 4, 7, 3, 2, 1, 6)$. This corresponds to the tour $1, 5, 2, 4, 3, 7, 6, 1$, as illustrated in the upper left of Fig. 8. If we fix only $(\bar{x}_1, \bar{x}_2, \bar{x}_3) = (5, 4, 7)$, the neighborhood consists of the 5 ways to reconnect the subchains $1 \rightarrow 5$, $2 \rightarrow 4$, $3 \rightarrow 7$, and 6.

This is quite different from the neighborhoods used in existing heuristics [21, 41]. Consider, for example, the well-known *2-opt* neighborhood, and suppose one starts with the solution $\bar{x} = (2, 3, \ldots, n, 1)$ in an undirected graph, which corresponds to the tour $1, 2, \ldots, n, 1$. A 2-opt neighborhood of $\bar{x}$ consists

of the tours obtained by replacing any two edges $(i, i+1)$ and $(j, j+1)$ with (i, j) and $(i+1, j+1)$, respectively (an example appears in Fig. 9). This corresponds to replacing $x = \bar{x}$ with

$$(x_1, x_2, \ldots, x_{i-1}, x_i, x_{i+1}, x_{i+2}, \ldots, x_{j-1}, x_j, x_{j+1}, \ldots, x_{n-1}, x_n) = \\ (2, 3, \ldots, i, j, j+1, i+1, \ldots, j-2, j-1, j+2, \ldots, n, 1)$$

This is very different from fixing a subset of variables and allowing the rest to vary. The well-known Lin-Kernighan heuristic [37] essentially defines an r-opt neighborhood, where r is adjusted dynamically, but this again is very different from leaving a subset of variables unfixed, and similarly for other heuristics (surveyed in [41]).

Defining neighborhoods by partial variable assignments suggests a simple strategy for converting a local search heuristic to an exact algorithm. One can enumerate fixed values by creating a search tree, and carry the search to completion to obtain an exact solution.

The analogy with branching also suggests the use of relaxation bounding in heuristic search, an idea proposed in [42]. One can solve a relaxation of the subproblem, and if the optimal value of the relaxation is no less than the best solution value found so far, the neighborhood can be rejected and replaced by another.

There is a good deal of research on the integration of heuristic methods and exact methods from constraint programming, surveyed in [10, 45]. Recent work includes [5, 13, 23, 35, 40].

4.2 GRASP

We saw in the previous section that local search based on partial variable assignments can be viewed as an incomplete branching search. The same is true of a *greedy randomized adaptive search procedure* (GRASP) [47], if it is generalized somewhat. This again allows for a relaxation bounding mechanism.

Each iteration of a conventional GRASP has a constructive phase followed by a local search phase. The constructive phase fixes the variables one at a time, based on a greedy choices that are perturbed by a random element. After a complete solution is obtained by fixing all variables, a local search begins with this solution. The process is then repeated as desired.

The algorithm can be viewed as a randomized tree search that forgets much of where it has already searched. The constructive phase is a probe to the bottom of the search tree. Each step of the local search phase is a special case of a random backtrack to a higher node in the portion of the tree currently in memory. A pure local search might backtrack only one or two levels, depending on which variables are altered to obtain a neighboring solution. Once the search backtracks to a higher node, all successors of that node are erased to reduce memory requirements, although the value of the best feasible solution obtained so far is stored for bounding purposes. If a relaxation is available, one an use the same sort of bounding mechanism as in exhaustive branch-and-bound search.

The result is that each iteration is an incomplete branch-and-bound method in which many nodes of the search tree evaporate soon after they are created.

A vehicle routing example from [29] illustrates the idea. A vehicle must begin at location A and deliver packages to four locations before returning to A. It may arrive as early as desired but must deliver the packages within the time windows shown in Table 1. Travel times t_{ij} are also shown. The objective is to return home as soon as possible. Figure 10 shows an incomplete randomized search tree. The initial greedy probe selects each location with the earliest possible delivery, resulting in the tour ADCBEA with value 36. The search randomly backtracks to node 2, whereupon nodes 3 and 4 are deleted. The next customer E is randomly selected, and the next tour is continued in greedy fashion, this time without finding a feasible tour. The final random backtrack is to node 1.

Bounding can be introduced by defining a relaxation at each node. A simple relaxation might observe that if a partial tour $A, x_2, \ldots, x_k$ has been defined, the travel time to an unvisited customer j from the previous customer will be at least $L_j = \min_i\{t_{ij}\}$, where i ranges over all customers not in $\{j, x_1, \ldots, x_{k-1}\}$. The travel time home from the last customer will be at least $L_1 = \min_j\{t_{j1}\}$, where j ranges over all customers not in $\{x_1, \ldots, x_k\}$. Then if T is the earliest time the vehicle can depart customer k, the total time is at least

$$T + L_0 + \sum_{j \notin \{x_1,\ldots,x_k\}} L_j$$

This relaxation yields a bound of 40 at node 5, which allows the search to backtrack to node 2 before completing the greedy probe, as shown in Fig. 11. A bound of 38 at node 6 in Fig. 11 terminates the third probe as well, thus accelerating the search as in a branch-and-bound tree.

4.3 Tabu Search

Tabu search [20, 24] is a constraint-directed form of local search. Viewing the method in this light suggests several ideas for modifying the search and connecting it with exact constraint-directed algorithms, such as Benders decomposition.

Standard tabu search evaluates all solutions in a neighborhood of the current solution $\bar{x}$ to find the best solution x' that does not appear on the *tabu list*. A new neighborhood is defined about x', $\bar{x}$ is added to the tabu list, and the oldest

Table 1: Data for a small 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]

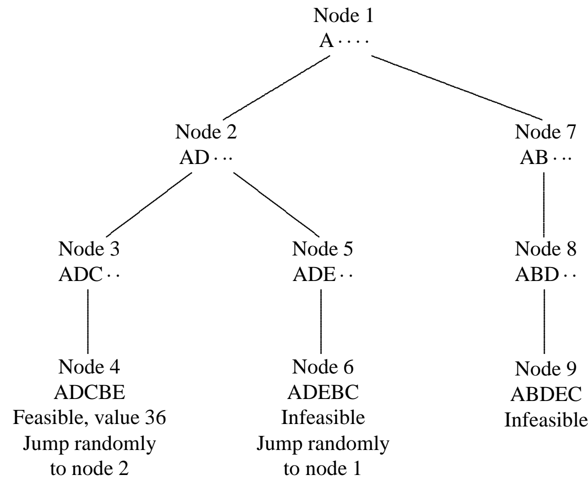

Figure 10: Incomplete search tree created by a generalized GRASP.

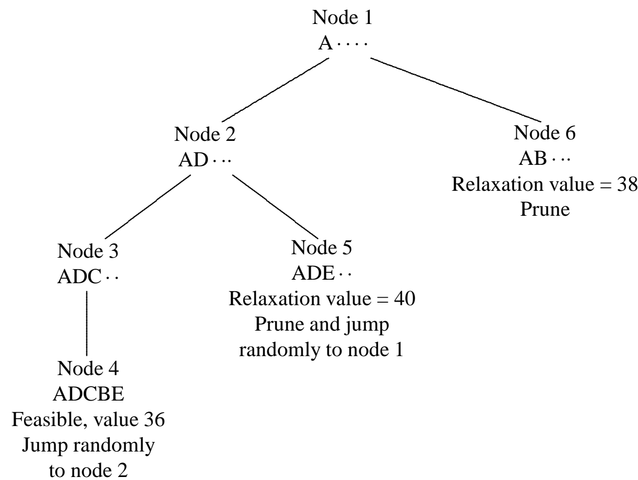

Figure 11: Incomplete search tree with relaxation bounding.

element of the tabu list is removed. The function of the tabu list is to prevent the search from cycling through the same solutions. It is inspired by short-term memory in human searches, which prevents one from looking in a spot that was recently checked. When all goes well, a seeker with short-term memory keeps migrating to different parts of the search space, where there is little chance of drifting back to previous locations.

A long tabu list may exclude all solutions in the neighborhood, while a shorter one may fail to prevent cycling. This dilemma is often addressed by adding “medium-term” and “long-term” memory. A common medium-term

Table 2: Intercity distances for a traveling salesman problem.

	2	3	4	5	6
1	7	5	∞	4	4
2		8	7	∞	6
3			3	8	∞
4				4	7
5					5

device is to remember which elements of the solution were recently removed, so that they will not be re-inserted in the near future. For example, an edge that is removed from a TSP tour remains on the tabu list for several iterations, during which it will not be re-inserted. This in effect excludes solutions that are similar to a solution examined in the recent past and more effectively prevents cycling. Long-term memory might use a *diversification* strategy, which keeps track of solution values obtained over a long stretch of the search. If there has been no significant improvement, the search randomly jumps to a different part of the search space.

Tabu search is clearly an instance of constraint-directed search in which the tabu list contains nogood constraints. The search is inexhaustive because constraints are dropped from the list. Medium-term memory plays a role similar to nogood constraints obtained from the inference dual, which tend to exclude solutions similar to solutions that have been tried before (although the constraints imposed by medium-term memory may exclude optimal solutions if they remain on the list long enough). This suggests that, rather than design the tabu list in an ad hoc manner, one can use an inference dual of the subproblem to obtain tabu items.

Suppose, for example, that the neighborhood in iteration k is obtained by fixing a tuple x^k of some of the variables to $\bar{x}^k$, so that the subproblem is (8). An optimal solution of the subproblem is the best solution in the neighborhood. A nogood cut $v \geq LB_k(x^k)$ is obtained from the solution of the inference dual (9). Now the nogood constraint $LB_k(x^k) < LB_k(\bar{x}^k)$ can be added to the tabu list. This excludes not only solutions x in which $x^k = \bar{x}^k$, but additional solutions that are similar in the sense that they cannot have a better value when the subproblem is solved.

This mechanism also provides a principled way to define the next neighborhood. We can fix x^{k+1} to a value $\bar{x}^{k+1}$ that satisfies all the nogood constraints in the current tabu list, perhaps while minimizing a function $f_{k+1}(x^{k+1})$ that approximates the objective function value $f(x)$. Due to the nogood constraints in the tabu list, this neighborhood will differ from those created recently. When the same variables x^k are fixed in each iteration, a tabu search of this kind becomes a heuristic form of Benders decomposition. Selecting the next solution, subject to the tabu list, corresponds solving the master problem. Even when x^k changes, the search becomes an exact method when all the nogoods added to the tabu list are retained.

As an example, consider a traveling salesman problem on 6 cities with symmetric distances as shown in Table 2 (an infinite distance indicates that the edge is missing). The objective is to find a hamiltonian cycle of minimum length. Let x_i be the city that follows city i in the cycle. We will define the first neighborhood by fixing (x_1, x_2), the second neighborhood by fixing (x_2, x_3), and so forth cyclically. Table 3 displays an exhaustive search. In the first line of the table, a neighborhood of size 3 is defined by fixing $(x_1, x_2) = (2, 3)$, and the best solution in the neighborhood is shown. We use *enumerative nogoods* rather than bounds by keeping track of which solutions have been examined. Thus we place $(x_1, x_2) \neq (2, 3)$ on the tabu list. We define the next neighborhood by fixing (x_2, x_3) at their current values. Only one solution in the neighborhood of 3 satisfies the tabu list, indicated by the notation 3(1). As the search proceeds, all nogoods are retained on the tabu list.

If the tabu list excludes the entire neighborhood, we enlarge the neighborhood by fixing fewer variables. This occurs in iterations 4, 6, 10 and 20. In the final iteration, we must expand the neighborhood to the entire search space by fixing no variables and enumerate all solutions that satisfy the tabu list. As it happens, there are no such solutions, and the search is complete, having found two optimal tours of length 29.

Table 4 displays an inexhaustive tabu search based on the same nogoods. The search is converted to a heuristic algorithm simply by dropping all but the two most recent nogoods on the tabu list. It proceeds identically with the exhaustive search through iteration 7. In iteration 8, the current tabu list allows a better solution (value 30), and from there out the search diverges from the exhaustive search. The search continues as long as desired.

An inexhaustive tabu search can be converted to an exhaustive one simply by retaining all enumerative nogoods on the tabu list, but one must take care that the nogoods are valid (do not exclude feasible solutions). When medium-term memory is exercised by excluding components of a solution, the resulting nogoods are frequently invalid. This can be remedied by generating a valid nogood, as well, in each iteration. The valid nogoods are retained, and the invalid ones dropped after a time as in traditional tabu search. The generation of invalid nogoods can then be discontinued after a desired number of iterations, at which point the search is run to completion with only valid nogoods.

4.4 Evolutionary Algorithms

Evolutionary algorithms represent one of several "social" metaheuristics in which an evolving population of solutions share information with each other. Other metaheuristics in this category include ant colony optimization and particle swarm optimization, discussed below. At each stage of evolution, the current population can be viewed as the feasible set of a problem restriction. These algorithms therefore search over problem restrictions, as does local search. Unlike local search, however, the solution population generally does not form a neighborhood in any interesting sense.

The best known evolutionary algorithm is a *genetic algorithm* [3, 25, 43],

Table 3: Exhaustive tabu search.

Iter.	Neighborhood	Size	Solution	Value	Tabu List
1.	$(x_1, x_2) = (2, 3)$	3(3)	$x = (2, 3, 4, 5, 6, 1)$	31	$(x_1, x_2) \neq (2, 3)$
2.	$(x_2, x_3) = (3, 4)$	3(1)	$x = (6, 3, 4, 5, 1, 2)$	29	$(x_1, x_2) \neq (2, 3)$ $(x_2, x_3) \neq (3, 4)$
3.	$(x_3, x_4) = (4, 5)$	3(1)	$x = (6, 3, 4, 5, 1, 2)$	30	$(x_1, x_2) \neq (2, 3)$ $(x_2, x_3) \neq (3, 4)$ $(x_3, x_4) \neq (4, 5)$
4.	$x_4 = 5$	8(4)	$x = (3, 6, 2, 5, 1, 4)$	34	$(x_1, x_2) \neq (2, 3)$ $(x_2, x_3) \neq (3, 4)$ $x_4 \neq 5$
5.	$(x_5, x_6) = (1, 4)$	3(2)	$x = (2, 6, 5, 3, 1, 4)$	33	$(x_1, x_2) \neq (2, 3)$ $(x_2, x_3) \neq (3, 4)$ $x_4 \neq 5$ $(x_5, x_6) \neq (1, 4)$
6.	$x_6 = 4$	8(3)	$x = (5, 1, 2, 3, 6, 4)$	34	$(x_1, x_2) \neq (2, 3)$ $(x_2, x_3) \neq (3, 4)$ $x_4 \neq 5,\ x_6 \neq 4$
7.	$(x_1, x_2) = (5, 1)$	2(1)	$x = (5, 1, 4, 6, 3, 2)$	35	$(x_1, x_2) \neq (2, 3), (5, 1)$ $(x_2, x_3) \neq (3, 4)$ $x_4 \neq 5,\ x_6 \neq 4$ $(x_1, x_2) \neq (5, 1)$
8.	$(x_2, x_3) = (1, 4)$	3(1)	$x = (6, 1, 4, 2, 3, 5)$	34	$(x_1, x_2) \neq (2, 3), (5, 1)$ $(x_2, x_3) \neq (3, 4), (1, 4)$ $x_4 \neq 5,\ x_6 \neq 4$
9.	$(x_3, x_4) = (4, 2)$	3(2)	$x = (3, 6, 4, 2, 1, 5)$	30	$(x_1, x_2) \neq (2, 3), (5, 1)$ $(x_2, x_3) \neq (3, 4), (1, 4)$ $x_4 \neq 5,\ x_6 \neq 4$ $(x_3, x_4) \neq (4, 2)$
10.	$x_4 = 2$	8(2)	$x = (6, 3, 1, 2, 4, 5)$	33	$(x_1, x_2) \neq (2, 3), (5, 1)$ $(x_2, x_3) \neq (3, 4), (1, 4)$ $x_4 \neq 5, 2,\ x_6 \neq 4$
$\vdots$					
20.	$x_2 = 4$	8(1)	$x = (3, 4, 2, 6, 1, 5)$	36	$(x_1, x_2) \neq (2, 3), (5, 1)$ $(x_2, x_3) \neq (3, 4), (1, 4)$ $x_4 \neq 5, 2,\ x_6 \neq 4$ $x_3 \neq 5,\ x_5 \neq 4,\ x_2 \neq 4$ $(x_1, x_6) \neq (2, 5), (5, 1)$
21.	$\emptyset$	0			

which is based on mating and survival of the fittest. Each member of the population has a "genotype" that encodes a possible solution of the problem to be solved. In each iteration, some or all members of the solution population mate to produce an offspring whose genotype is a mixture of genes from the parents. The new genotype is obtained by a *crossover* operation that may simply splice together subsequences from the parental genotypes. Random changes

Table 4: Inexhaustive tabu search.

Iter.	Neighborhood	Size	Solution	Value	Tabu List
1.	$(x_1, x_2) = (2, 3)$	3(3)	$x = (2, 3, 4, 5, 6, 1)$	31	$(x_1, x_2) \neq (2, 3)$
2.	$(x_2, x_3) = (3, 4)$	3(1)	$x = (6, 3, 4, 5, 1, 2)$	29	$(x_1, x_2) \neq (2, 3)$ $(x_2, x_3) \neq (3, 4)$
3.	$(x_3, x_4) = (4, 5)$	3(1)	$x = (6, 3, 4, 5, 1, 2)$	30	$(x_2, x_3) \neq (3, 4)$ $(x_3, x_4) \neq (4, 5)$
4.	$x_4 = 5$	8(4)	$x = (3, 6, 2, 5, 1, 4)$	34	$(x_2, x_3) \neq (3, 4)$ $x_4 \neq 5$
5.	$(x_5, x_6) = (1, 4)$	3(2)	$x = (2, 6, 5, 3, 1, 4)$	33	$x_4 \neq 5$ $(x_5, x_6) \neq (1, 4)$
6.	$x_6 = 4$	8(3)	$x = (5, 1, 2, 3, 6, 4)$	34	$x_4 \neq 5,\ x_6 \neq 4$
7.	$(x_1, x_2) = (5, 1)$	2(1)	$x = (5, 1, 4, 6, 3, 2)$	35	$(x_1, x_2) \neq (5, 1)$ $x_6 \neq 4$
8.	$(x_2, x_3) = (1, 4)$	3(2)	$x = (3, 1, 4, 5, 6, 2)$	30	$(x_1, x_2) \neq (5, 1)$ $(x_2, x_3) \neq (1, 4)$
9.	$(x_3, x_4) = (4, 5)$	3(2)	$x = (6, 4, 3, 5, 1, 2)$	29	$(x_2, x_3) \neq (1, 4)$ $(x_3, x_4) \neq (4, 5)$
10.	$(x_4, x_5) = (5, 1)$	2(1)	$x = (3, 6, 2, 5, 1, 4)$	34	$(x_3, x_4) \neq (4, 5)$ $(x_4, x_5) \neq (5, 1)$
11.	$(x_5, x_6) = (1, 4)$	3(2)	$x = (2, 6, 5, 3, 1, 4)$	35	$(x_4, x_5) \neq (5, 1)$ $(x_5, x_6) \neq (1, 4)$
⋮					

or "mutations" may be introduced to diversify the gene pool. Some offspring are lucky enough to inherit superior traits from both parents and therefore improve the quality of the population. The worst solutions are removed from the population, representing survival of the fittest, and the next iteration begins. After a number of generations, the population is examined to find the best solution.

This is a case in which solving a relaxation of the current restriction guides the search, as in branch-and-bound search, although the mechanism is very different. In a genetic algorithm, enlarging the population to include offspring in effect relaxes the current restriction. The relaxation is solved by evaluating the members of the population to identify the fittest. The next restriction consists of those solutions identified. There is no obvious role for relaxation bounding in this model, however.

The inference dual of the problem restriction is uninteresting, because it is solved simply by examining all the solutions in the population and selecting the fittest. However, rather than defining an inference dual based on *deduction*, one might use *induction*. That is, one could examine the current population for characteristics that tend to predict good solution values, perhaps using regression analysis. This might result in a function $LB_k(x)$ that predicts an objective function value based on whether x has certain characteristics. Then a nogood bound would have the form, "v is *probably* bounded below by $LB_k(x)$."

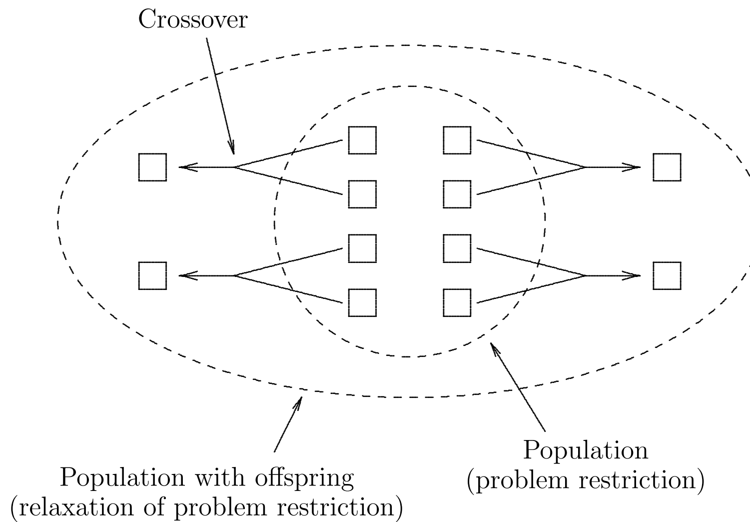

Figure 12: Genetic algorithm interpreted as a search over relaxations of problem restrictions. The "solution" of the relaxation (identification of the fittest solutions) determines the next restriction (population).

This could guide the choice of parents to mate in future generations, or even the design of the crossover operation.

An alternate interpretation of genetic algorithms sees them as local search algorithms in which the current population P is regarded as a single solution. The value of P depends somehow on the fitness of the gene pool, perhaps the average value of the genotypes in P. A neighborhood of P is defined by adding offspring to obtain P', and neighboring populations are subsets of P' having a specified size. This neighborhood is the feasible set of a problem restriction, which is solved by selecting the fittest population in the neighborhood. At termination, the best solution in the current population P is selected.

Relaxation bounding can play a role in this model, as a device for sizing neighborhoods. If v^* is the best population fitness obtained so far, offspring can be created until P's neighborhood contains a population with fitness better than v^*. In effect, neighborhoods that lack this property are pruned. So we enlarge the feasible set until the subproblem has an acceptable solution, as in variable-depth neighborhood search, rather than restricting the subproblem until it becomes soluble, as in branching search.

4.5 Ant Colony Optimization

Ant colony optimization [14, 15] mimics the seeking behavior of ants, who communicate with other ants by depositing pheromones as they explore for food. The idea is best explained by example, such as the TSP.

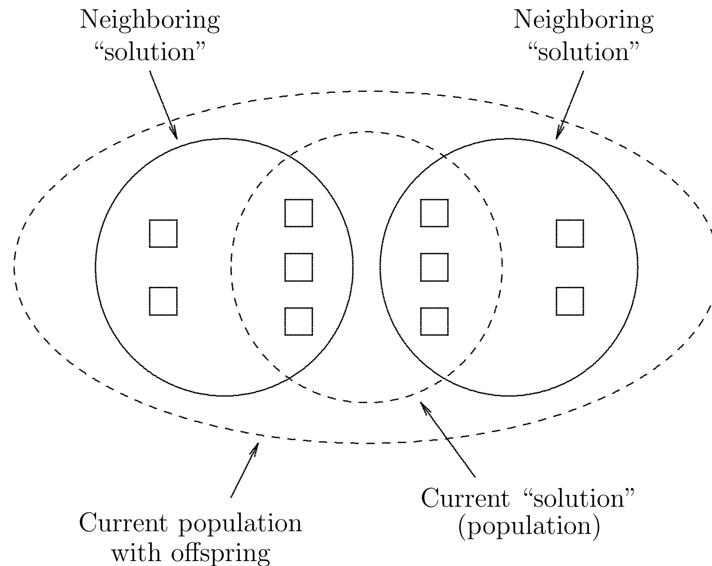

Figure 13: Genetic algorithm interpreted as local search. The current "solution" is the population, and each neighboring "solution" is a subset of the population and offspring.

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 p_{ij}/d_{ij}, where p_{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 (Fig. 14). 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.

In each iteration, each ant carries out a randomized greedy heuristic in which the probabilities are defined by pheromone deposits and distances. Thus the ant search is basically a GRASP with a constructive phase but no local search phase. Each variable assignment in the constructive phase creates a problem restriction, and we therefore have a search over problem restrictions as in branching search. Relaxation bounding can be introduced as described in the Section 4.1 above. If at any point in the greedy algorithm the relaxation value is worse than an incumbent solution, the tree is pruned at that point. The ant immediately returns to the anthill and begins another tour.

Ant colony optimization can be converted to an exact algorithm, in much the same way as the constructive phase of a GRASP. The ants collectively create a search tree in which the branching decisions are influenced by pheromones. We

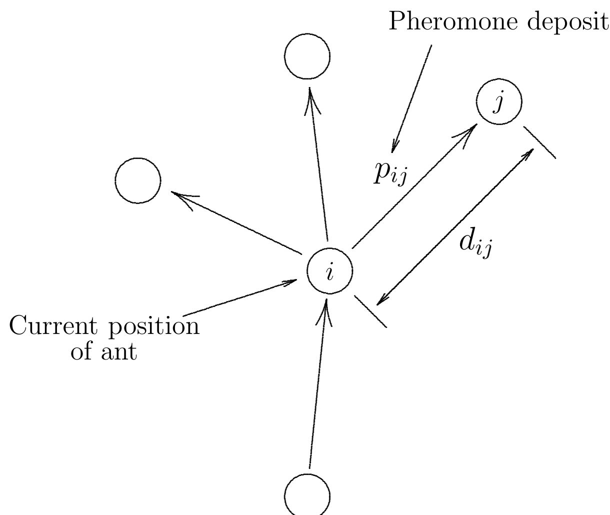

Figure 14: Ant colony optimization for the TSP.

can simply keep track of this tree and prevent an ant from entering a subtree that has already been explored. Depending on the problem structure, we may be able to introduce backjumping with the constraint-directed search mechanism described in Section 2.2.

4.6 Particle Swarm Optimization

Particle swarm optimization [34, 46] is normally presented as a method for continuous global optimization, although it can be applied to discrete problems as well.

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.

The current particle positions represent the feasible set of a problem restriction, and the method therefore searches over problem restrictions. The restriction is solved in the sense that all of the positions are evaluated, although no particular solution is selected until the algorithm terminates.

The particle swarm algorithm can be viewed as constraint-directed search in which the inference dual uses inductive inference, as suggested for genetic algorithms in Section 4.4. The inference dual is again solved simply by evaluating all solutions of the subproblem. An examination of the solution values suggests that solutions in certain regions tend to be better, and particles therefore gravitate toward these regions in the next iteration. Thus, information

that is inductively inferred from the optimality proof guides the choice of future restrictions. One could carry this further by analyzing the solution set more closely, perhaps to identify specific characteristics that tend to predict better solution values. Particles could then move in directions that tend to favor these characteristics.

Constraint-directed search based on inductive inference is obviously related to machine learning and learning-based algorithms in general, and there may be some value in investigating how these fit into the framework suggested here.

5 Conclusion

This paper presented a framework in which many exact and heuristic methods can be seen as having common structure that permits some degree of unification, as summarized in Tables 5 and 6. It interprets solution algorithms as primal-dual methods in which the primal component searches over problem restrictions, and the dual component obtains bounds on the optimal value. In particular, the concept of an inference dual provides the basis for constraint-directed search, an algorithmic scheme shared by many exact and heuristic methods. The motivations behind unification are (a) to encourage the exchange of algorithmic techniques between exact and heuristic methods, and (b) to design solution methods that transition gracefully from exact to heuristic modes as problem instances scale up.

References

[1] T. Achterberg. Conflict analysis in mixed integer programming. *Discrete Optimization*, 4:4–20, 2007.

[2] R. K. Ahuja, O. Ergun, J. B. Orlin, and A. P. Punnen. A survey of very large scale neighborhood search techniques. *Discrete Applied Mathematics*, 123:75–102, 2002.

[3] N. A. Barricelli. Numerical testing of evolution theories. Part II: Preliminary tests of performance, symbiogenesis and terrestrial life. *Acta Biotheoretica*, 16:99–126, 1963.

[4] P. Beame, H. Kautz, and A. Sabharwal. Understanding the power of clause learning. In *International Joint Conference on Artificial Intelligence (IJCAI 2003)*, 2003.

[5] P. Benchimol, J.-C. Régin, L.-M. Rousseau, M. Rueher, and W.-J. van Hoeve. Improving the Held and Karp approach with constraint programming. In A. Lodi, M. Milano, and P. Toth, editors, *Proceedings of the International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*

Table 5: Summary of exact methods.

Method	*Restrictions Enumerated*	*Relaxation Dual Providing Bounds*	*Nogoods from Inference Dual*	*To Convert to a Heuristic Method*
DPLL (for SAT)	Leaf nodes of search tree		Conflict clauses	Drop some conflict conflict clauses
Simplex (for LP) (for LP)	Edges of polyhedron	Surrogate dual	Reduced costs	Terminate prematurely
Branch & bound (for IP)	Leaf nodes of search tree	LP bounds	Conflict analysis	Forget tree and use nogood constraints
Branch & bound as local search	Incomplete search trees	Uphill search over incomplete trees		Terminate prematurely
Benders decomposition	Subproblems	Master problem	Logic-based Benders cuts	Drop some Benders cuts

Table 6: Summary of heuristic methods.

Method	*Restrictions Enumerated*	*Relaxation Dual Providing Bounds*	*Nogoods from Inference Dual*	*To Convert to an Exact Method*
Local search (e.g., for TSP)	Neighborhoods (e.g., 2-opt nbhds)	Adjustment of neighborhood size		Search over partial solutions
GRASP (e.g., for TSPTW)	Nodes of incomplete search tree	As in branch & bound		Don't forget nodes, complete the tree
Tabu search (e.g., for TSPTW)	Neighborhoods		Items on tabu list	Search over partial solutions; dynamic backtracking
Genetic algorithm	Populations		Crossover guidance from inductive inference dual	
Genetic algorithm as local search	Subsets of population plus offspring	Control of neighborhood size	Logic-based	
Ant colony optimization	Same as GRASP	As in GRASP		As in GRASP
Particle swarm optimization	Sets of swarm lcoations		Relocation guidance from inductive inference dual	

(CPAIOR 2010), volume 6140 of *Lecture Notes in Computer Science*, pages 40–44, New York, 2010. Springer.

[6] J. F. Benders. Partitioning procedures for solving mixed-variables programming problems. *Numerische Mathematik*, 4:238–252, 1962.

[7] M. Benichou, J. M. Gautier, P. Girodet, G. Hentges, R. Ribiere, and O. Vincent. Experiments in mixed-integer linear programming. *Mathematical Programming*, 1:76–94, 1971.

[8] R. E. Bixby, W. Cook, A. Cox, and E. K. Lee. Parallel mixed integer programming. Technical report CRPC-TR95554, Center for Research on Parallel Computation, 1995.

[9] C. Bliek. Generalizing dynamic and partial order backtracking. In *National Conference on Artificial Intelligence (AAAI 1998)*, pages 319–325, Madison, WI, 1998.

[10] C. Blum, J. Puchinger, G. Raidl, and A. Roli. Hybrid metaheuristics. In P. van Hentenryck and M. Milano, editors, *Hybrid Optimization: The Ten Years of CPAIOR*, pages 305–336. Springer, New York, 2011.

[11] M. Davis, G. Logemann, and H. Putnam. A machine program for theorem proving. *Communications of the ACM*, 5:394–397, 1962.

[12] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215, 1960.

[13] K. Dhyani, S. Gualandi, and P. Cremonesi. A constraint programming approach to the service consolidation problem. In A. Lodi, M. Milano, and P. Toth, editors, *Proceedings of the International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR 2010)*, volume 6140 of *Lecture Notes in Computer Science*, pages 97–101, New York, 2010. Springer.

[14] M. Dorigo. Optimization, learning and natural algorithms. PhD thesis, Politecnico di Milano, 1992.

[15] M. Dorigo and L.M. Gambardella. Ant colony system: A cooperative learning approach to the traveling salesman problem. *IEEE Transactions on Evolutionary Computation*, 1:53–66, 1997.

[16] J. Gaschnig. Experimental studies of backtrack vs. waltz-type vs. new algorithms for satisficing-assignment problems. In *Proceedings, 2nd National Conference of the Canadian Society for Computational Studies of Intelligence*, pages 19–21, 1978.

[17] J. M. Gautier and R. Ribiere. Experiments in mixed-integer linear programming using pseudo-costs. *Mathematical Programming*, 12:26–47, 1977.

[18] M. L. Ginsberg. Dynamic backtracking. *Journal of Artificial Intelligence Research*, 1:25–46, 1993.

[19] M. L. Ginsberg and D. A. McAllester. GSAT and dynamic backtracking. In *Principles and Practice of Constraint Programming (CP 1994)*, volume 874 of *Lecture Notes in Computer Science*, pages 216–225, New York, 1994. Springer.

[20] F. Glover. Tabu search: Part I. *ORSA Journal on Computing*, 1:190–206, 1989.

[21] B. L. Golden and W. R. Stewart. Empirical analysis of heuristics. In E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys, editors, *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*, pages 207–249. Wiley, New York, 1985.

[22] C. P. Gomes, H. Kautz, A. Sabharwal, and B. Selman. Satisfiability solver. In F. van Harmelen, V. Lifschitz, and B. Porter, editors, *Handbook of Knowledge Representation*, pages 89–134. Elsevier, Amsterdam, 2008.

[23] N. El Hachemi, M. Gendreau, and L.-M. Rousseau. Hybrid LS/CP approach to solve the weekly log-truck scheduling problem. In W.-J. van Hoeve and J. N. Hooker, editors, *Proceedings of the International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR 2009)*, volume 5547 of *Lecture Notes in Computer Science*, pages 319–320, New York, 2009. Springer.

[24] P. Hansen. The steepest ascent mildest descent heuristic for combinatorial programming. In presentation at *Congress on Numerical Methods in Combinatorial Optimization*, Capri, 1986.

[25] J. H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, 1975.

[26] J. N. Hooker. *Logic-Based Methods for Optimization: Combining Optimization and Constraint Satisfaction*. John Wiley, New York, 2000.

[27] J. N. Hooker. Unifying local and exhaustive search. In L. Villaseñor and A. I. Martinez, editors, *Avances in la Ciencia de la Computación (ENC 2005)*, pages 237–243, Puebla, Mexico, 2005.

[28] J. N. Hooker. Duality in optimization and constraint satisfaction. In J. C. Beck and B. M. Smith, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR 2006)*, volume 3990 of *Lecture Notes in Computer Science*, pages 3–15, New York, 2006. Springer.

[29] J. N. Hooker. *Integrated Methods for Optimization*. Springer, New York, 2007.

[30] J. N. Hooker. Planning and scheduling by logic-based Benders decomposition. *Operations Research*, 55:588–602, 2007.

[31] J. N. Hooker. *Integrated Methods for Optimization, 2nd ed.* Springer, New York, 2012.

[32] J. N. Hooker and G. Ottosson. Logic-based Benders decomposition. *Mathematical Programming*, 96:33–60, 2003.

[33] J. N. Hooker and H. Yan. Logic circuit verification by Benders decomposition. In V. Saraswat and P. Van Hentenryck, editors, *Principles and Practice of Constraint Programming: The Newport Papers*, pages 267–288, Cambridge, MA, 1995. MIT Press.

[34] J. Kennedy and R. Eberhart. Particle swarm optimization. In *Proceedings of IEEE International Conference on Neural Networks*, pages 1942–1948, 1995.

[35] M. Khichane, P. Albert, and C. Solnon. Strong combination of ant colony optimization with constraint programming optimization. In A. Lodi, M. Milano, and P. Toth, editors, *Proceedings of the International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR 2010)*, volume 6140 of *Lecture Notes in Computer Science*, pages 232–246, New York, 2010. Springer.

[36] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[37] S. Lin and B. Kernighan. An effective heuristic algorithm for the traveling salesman problem. *Operations Research*, 21:498–516, 1973.

[38] D. A. McAllester. Partial order backtracking. Manuscript, AI Laboratory, MIT, Cambridge, MA, 1993.

[39] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference (DAC 2001)*, pages 530–535, 2001.

[40] Q. D. Pham, Y. Deville, and P. Van Hentenryck. Constraint-based local search for constrained optimum paths problems. In A. Lodi, M. Milano, and P. Toth, editors, *Proceedings of the International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR 2010)*, volume 6140 of *Lecture Notes in Computer Science*, pages 267–281, New York, 2010. Springer.

[41] D. Pisinger and S. Ropke. Large neighborhood search. In M. Gendreau and J.-Y. Potvin, editors, *Handbook of Metaheuristics*, pages 399–420. Springer, New York, 2010.

[42] S. Prestwich. Exploiting relaxation in local search. In *First International Workshop on Local Search Techniques in Constraint Satisfaction (LSCS 2004)*, Toronto, 2004.

[43] I. Rechenberg. *Evolutionsstrategie*. Holzmann-Froboog, Stuttgart, 1973.

[44] T. Sandholm and R. Shields. Nogood learning for mixed integer programming. In *Workshop on Hybrid Methods and Branching Rules in Combinatorial Optimization*, 2006.

[45] P. Shaw. Constraint programming and local search hybrids. In P. van Hentenryck and M. Milano, editors, *Hybrid Optimization: The Ten Years of CPAIOR*, pages 271–304. Springer, New York, 2011.

[46] Y. Shi and R. Eberhart. A modified particle swarm optimizer. In *Proceedings of IEEE International Conference on Neural Networks*, pages 69–73, 1998.

[47] J. P. M. Silva and K. A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48:506–521, 1999.

[48] R. M. Stallman and G. J. Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Journal of Artificial Intelligence*, 9:135–196, 1977.

[49] T. H. Yunes, I. Aron, and J. N. Hooker. An integrated solver for optimization problems. *Operations Research*, 58:342–356, 2010.