8 A CONSTRAINT PROGRAMMING TOOLKIT FOR LOCAL SEARCH

Paul Shaw[1], Vincent Furnon[2] and Bruno De Backer[2]

[1]ILOG S.A.
Les Taissounieres HB2, 1681, route des Dolines
06560 Valbonne, FRANCE
`pshaw@ilog.fr`

[2]ILOG S.A.
9, rue de Verdun, BP 85
94253 Gentilly Cedex, FRANCE
`{vfurnon,bdebacker}@ilog.fr`

Abstract: Constraint programming is both a modeling and problem solving technology applicable to a wide range of optimization problems. Its inherent flexibility means that it is one of the leading technologies used in practical applications – aided largely by the existence of *constraint programming systems.*

ILOG Solver is a commercial constraint programming system taking the form of a C++ component library thatusers can tightly integrate withtheir own applications. In this chapter we concentrate on one aspect of ILOG Solver – the local search facilities added in version 5.0. The chapter is tutorial in nature, with local search features being elaborated by examples.

8.1 INTRODUCTION

Constraint programming (Rossi (2000), Tsang (1993), van Hentenryck (1989)) is a technology which is becoming of growing importance in the field of optimization, its main advantage over other techniques being that one can directly describe and solve general models using it. The practical success of constraint programming in recent years has been greatly aided by both academics and, most importantly, commercial software vendors who have produced *constraintprogramming systems.* These systems form a basis for the production of optimization tools, which are becoming increasingly industrially important. Examples of academic systems are Claire (Caseau and Laburthe (1995)), OZ (Henz et al. (1993), Henz et al. (1995)), clp(fd) (Condognet and Diaz (1996)), and CHR (Frühwirth (1998)). The most prolific commercial libraries are ILOG Solver (ILOG (2000b)), ECLiPSe (IC-PARC (2001)) and CHIP (Dincbas et al. (1988), Simonis et al. (1995)). In this chapter, we describe the evolution of one of these constraint programming systems, ILOG Solver, with respect to the local search support which has recently been introduced.

Our main objective in the addition of local search support to Solver was to provide concepts and objects which were as natural as possible for the local search practitioner, or indeed beginner. Therefore, the concepts should follow closely what already exists in the literature: see, e.g., Aarts and Lenstra (1997). For example, because the concepts of solutions and neighborhood structures are fundamental, these should be important objects in ILOG Solver's local search. Second, we wished to provide a level of local search support which would be flexible and extensible. Thus, we follow the already proven approach already taken in Solver, which is to provide support for common, but not all conceivable, demands. To cope with less common or esoteric requests, Solver is *extensible*; users are at liberty to extend the important parts of the constraint programming system. This means that typical users are not burdened with learning a huge system, and yet such a system can still meet a wide variety of needs. In the same spirit, Solver's local search support provides objects for constructing the most common types of local search such as hill climbing, simulated annealing, and tabu search. More esoteric methods, which commonly have specific application in any case, can be written by users to exactly suit their needs.

This chapter is set out as follows. Section 8.2 gives an overview of constraint programming for those unfamiliar with the technology. Section 8.3 describes Solver's local search mechanisms in detail, discussing the main components, and how they interact. Examples are given of the use of the main components, and code of a complete local search example is presented. Finally, we discuss the advantages and disadvantages of embedding a local search toolkit in a constraint programming system. Section 8.4 then presents a complete real-world example of facility location, solved by three different search methods. Of particular interest is a local/complete search hybrid. Section 8.5 demonstrates how the local search features of Solver can be extended and an example is given of how to code a new neighborhood. Section 8.6 gives a more complete demonstration of how Solver can be extended on all fronts, resulting in a constraint programming system for vehicle routing, ILOG Dispatcher. Dispatcher's main concepts are introduced through an example. We concentrate on the extension of Solver's generic local search facilities, which have been targeted towards vehicle routing. Section 8.7 describes related work both in the constraint programming domain, and in respect of systems for local search. We conclude in Section 8.8.

8.2 CONSTRAINT PROGRAMMING PRELIMINARIES

This chapter concentrates on the local search features of ILOG Solver. As such, we introduce other notions of constraint programming only when relevant. However, we gain a certain economy if basic ideas are presented together. This section then – which readers familiar with constraint programming can skip – gives an overview of constraint programming and its basic notions.

Fundamentally, a constraint programming problem is defined by a set of unknowns or variables, and a set of constraints. Each variable has a finite set of possible values (a domain), and each constraint involves a subset of the variables. Each such constraint indicates which partial assignments, involving only the variables incident on the constraint, will violate (or, alternatively, satisfy) the constraint. The problem is then one of finding an assignment (single element of the domain) for each variable such that no constraints are violated. This problem is $\mathcal{NP}$-complete (Garey and Johnson (1979)). The optimization variant of this problem is one where we must find the minimal value for one of the variables such that no constraints are violated.

8.2.1 Tree Search

In constraint programming systems, solutions to problems are generally found using *tree search* and, in the simplest case, the tree search mechanism known as depth-first search. The advantage of depth-first search is that it uses memory economically while being complete: it guarantees to find a solution if one exists, or reports that there is no solution if the problem is not solvable. Tree search in the worst case examines all combinations of values for all variables, but this worst case is rarely attained. Most commonly, significant parts of the search tree are pruned by the action of constraints.

We demonstrate how depth-first search operates by means of a small example. Consider a problem with three variables a, b and c which have domains $\{0, 2\}$, $\{0, 1\}$ and $\{0, 1\}$, respectively. We shall also consider three constraints: $a \neq b$, $a \geq c$ and $b \leq c$. By inspection, the solutions to this problem are: $a = 2, b = 0, c = 0$, $a = 2, b = 0, c = 1$ and $a = 2, b = 1, c = 1$. The other five possible combinations of values for a, b and c violate at least one constraint and so are not admissible. We will perform a depth-first search, choosing to instantiate variables in lexicographical order: first a, then b, then c. We shall choose possible values for variables in ascending order. The specification of the order that variables and values are chosen, and more generally the description of the growth of the search tree, is called a constraint programming *goal.* The resulting search tree for this goal is depicted in Figure 8.1. Nodes in the tree represent states of the assignment of variables. When a node is crossed, i.e. $\otimes$, this state is one in which at least one constraint is violated. Arcs are transitions between states and are labelled with the assignment that takes place at the state transition. Finally, non-crossed leaves in the tree are *solution states,* where all variables are assigned and no constraint is violated.

The depth-first search proceeds as follows. To begin, we take variable a and assign it its smallest value, 0, followed by an assignment of variable b to 0, which is its smallest value. Although we do not have a complete solution yet, we can check the validity of the constraint $a \neq b$ as all the variables involved in the constraint are

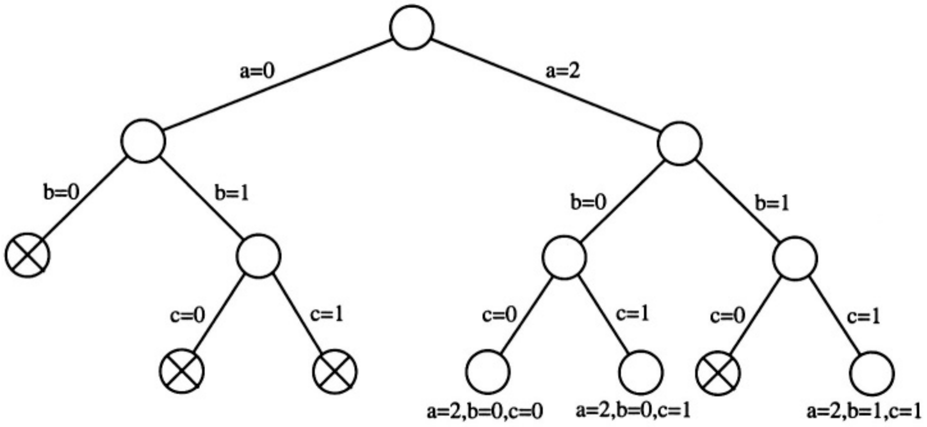

Figure 8.1 Operation of Depth-First Search

instantiated. That is, no assignments we make after this point can change the value – violated or satisfied – of the constraint. In this case we see that the constraint $a \neq b$ is violated: we call this state a *failure* (marked by $\otimes$). One of the assignments $a = 0$ or $b = 0$ needs to be changed: we undo assignments in a "depth-first" manner by undoing the most recent assignment first. We move the search back to the state before assigning b, a movement known as *backtracking,* and try the next value for b which is 1. This time, the constraint $a \neq b$ is not violated, and so we continue, by instantiating c. In fact, we find that neither value for c is satisfactory as 0 violates constraint $b \leq c$, whereas 1 violates constraint $a \geq c$. We have now proved that there exist no solutions for which $a = 0$ as we have explored all combinations of values for b and c with $a = 0$. We therefore backtrack to the top of the tree and assign $a = 2$. Consequently, we find that instantiating both b and c to 0 results in no constraints being violated: a solution. If we were interested in any solution to the problem we could simply stop. However, we can also continue to find more solutions via backtracking. When we backtrack we can change the assignment of c to 1 to find a second solution. Assigning $b = 1$ followed by $c = 0$ results in a failure (violates $b \leq c$), but assigning $c = 1$ on backtrack produces the final solution.

8.2.2 Constraint Propagation

The technique of constraint checking described above is known as backward checking as constraints are checked retrospectively after all variables involved in the constraint have been assigned. This is an improvement over waiting until a leaf node is reached before checking the validity of all constraints (known as *generate and test*). Nevertheless, it is normally too weak a technique to solve all but the smallest of problems. Constraint programming systems normally use a more powerful approach to pruning known as *constraint propagation.* Constraint propagation is a much more active technique than backward checking. Instead of checking the validity of constraints, future possibilities are filtered out using an appropriate filtering algorithm. This nearly al-

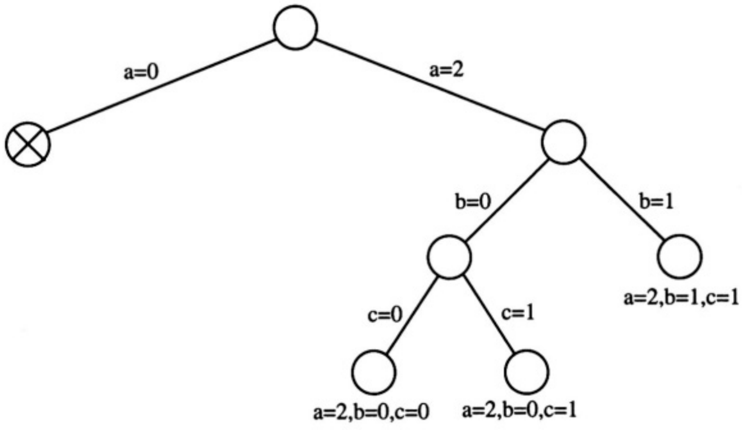

Figure 8.2 Operation of Depth-First Search with Propagation

ways results in less failures occurring – that is, dead-ends in the search tree are noticed higher up, removing the need to explicitly explore the sub-tree.

In backward checking, each variable was either assigned to a single value, or had not yet been assigned. Constraint propagation, on the other hand, maintains the *current domains* of each variable. The current domain begins equal to the initial domain, and is filtered as search progresses down the tree. If a value can be shown not to be possible for a variable (as it would violate at least one constraint), it is removed from the current domain. Such removed values are no longer considered as branching possibilities, and so the search tree is more effectively pruned than it would be for a backward checking algorithm. The algorithms that prove the illegality of domain values and filter them are typically polynomial in complexity; general methods (Bessiere and Regin (1997), Mackworth (1977)) and algorithms specific to the constraint type exist (Beldiceanu and Contjean (1994), Régin (1994), Régin (1996)).

Filtering of domain values is performed *independently* for each constraint; the only manner in which the constraints communicate is through changes to the current domains of variables. Roughly speaking, the constraint propagation algorithm operates by asking each constraint in turn to perform any filtering it can by examining the current domains of the variables that it constrains. Each constraint performs its filtering in a round-robin fashion until either a variable has an empty domain (no possible values) or there is a complete pass where no domains are changed by the constraints. In the first case a *failure* and a backtrack occur. In the second, a fixed point has been reached: if all variables are assigned a value, we have a solution, otherwise branching is required to find a solution or come up with a proof of insolvability. The fixed point reached by constraint propagation does not depend on the order in which constraints filter domain values. The same fixed point results regardless of the ordering.

Figure 8.2 shows how depth-first search with propagation works on our simple problem. As can be seen it is more efficient than the backwards checking version in terms of the number of failures: only one failure now results; all other paths lead to solutions. The search proceeds as follows. At the top of the tree, before any branching, constraint propagation is carried out. In this case, however, no filtering can be done by any of the constraints individually. A branch is then created, and we assign $a = 0$. The constraint $a \neq b$ deduces that $b \neq 0$ and thus $b = 1$. The constraint $a \geq c$ deduces that $c \neq 1$ and thus $c = 0$. The constraint $b \leq c$ deduces that $c \neq 0$ (which causes a failure as $c = 0$) and that $b \neq 1$ (which also causes a failure as $b = 1$). Note that only one failure will occur depending on which deduction is made first, resulting in either b or c having an empty domain. When the failure occurs, the solver backtracks to the root node, and restores the current domains of the variables to that which they were directly before branching. The right branch is then taken by assigning $a = 2$. As at the root node, none of the constraints can reduce domains, and so we branch again, assigning $b = 0$. Again, no filtering occurs, and a final branch to the first solution is made by assigning $c = 0$. Backtracking and assigning $c = 1$ gives the second solution in like manner. We then backtrack to before the assignment of b and take the right branch by assigning $b = 1$. In this case, filtering is performed by the constraint $b \leq c$ which removes the value 0 from the domain of c. This is the only filtering that can be performed at this node, and it results in a solution. Branching on c was not required as it had already been given a unique value by propagation. In this manner the final solution is found.

8.2.3 A Note on Optimization

Up until now, we have described how to solve a decision problem: find an assignment that satisfies all constraints. However, often in real-life situations, we are presented with optimization problems. Typically, these problems are solved as a series of decision problems. When a solution of value v is found, a constraint is added to the problem stating that any solution with value v or greater is illegal. This process is repeated until no better solutions are found, the last one found being the optimal. Each time a new better solution is found, there is no need to restart the search, instead, we can continue to search the tree, taking into account the new bounding constraint. The bounding constraint thus becomes progressively tighter as the search-tree is traversed.

8.2.4 Powerful Modeling and Searching

In the preceding pages, for pedagogical purposes, we have given examples only of simple constraints with simple propagation rules. By contrast, when we are faced with industrial optimization problems, models are more complex, and consequently constraint programming systems provide rich constraints to deal with this complexity. Aside from their richness in modeling, these constraints often have powerful and efficient propagation algorithms, resulting in intensive filtering of domains; see, e.g., Régin (1994), Régin (1996), Regin and Puget (1997). Examples of the more complex types of constraints available in constraint programming libraries are:

- $y = a[x]$: y is constrained to be equal to the xth element of a where a is either an array of constants or variables.

- *all-diff(a)*: all variables in array a take on different values.

- *c=card(a, v)*: variable c is constrained to be the number of occurrences of value v in the array of variables a.
- *min-dist(a, d)*: all pairs of variables in array a must take on values which differ by at least d.

In open constraint programming systems, users can write their own constraints, by giving the appropriate filtering rules to the solving engine. This is often indispensable as the system can never have every constraint built-in which would result in the most natural model.

Constraint programming systems invariably offer the ability to write one's own search goals: that is, describe how variables should be given values in the search tree. This can be very important as the order of variable instantiation can play a large part in how much propagation results. For example, in our three-variable example, no failures result if we instantiate the variables in the order b, a, c instead of a, b, c. As another example, in the more practical setting of routing, it is invariably better to build up a route in one long chain, rather than start on many chains which will eventually join up. Such heuristic orders can often improve performance by more than an order of magnitude. The openness and flexibility of constraint programming systems enables users to write more efficient optimization software than would be possible using a black-box system.

8.3 THE LOCAL SEARCH TOOLKIT

In recent years, many successful optimization applications have been based on constraint programming; see, e.g., Wallace (1996). ILOG Solver is the fastest available constraint programming system (Fernandez and Hill (2000)), which owes its success to the basic techniques of tree-based search and powerful propagation. However, for some problems, even these powerful methods can result in propagation that is too weak for the search to produce high quality solutions within a reasonable time frame. For such problems and others, it is often found that local search can be an effective technique. For this reason, local search extensions was added to ILOG Solver 5.0. We describe these extensions in this section.

8.3.1 Philosophy

ILOG Solver is an open system for constraint programming which is presented as a C++ library. It is open in the sense that users have access to primitives of the library such as constraints and the search process. Importantly, this access is at the programming language (C++) level and not at a parameter-changing level, allowing wide ranging changes or specializations of the library.

The local search extensions to ILOG Solver fit with this philosophy, providing a flexible and open local search system which meshes with the traditional tree-search approach. The tenets of the local search extensions are the following:

- **An object-based view of local search concepts.** ILOG Solver has an object-based view of variables, constraints and goals. Its local search extensions take the same view resulting in objects for the main local search concepts, such as solutions, neighborhoods, and meta-heuristics. The benefits of the object view are numerous, especially when concepts such as neighborhoods are presented as abstract classes which are specialized to give desired behavior for a particular case. An object-based model also allows operations to be carried out on these standard local search concepts; for example, two neighborhoods can be combined to create a third.

- **Control given to the user between moves.** In certain local search systems, for example OpenTS (IBM Open Source Software (2001)), the local search itself is performed as a black box. The number of iterations or stopping condition is given as a parameter to the local search, and the search runs until that condition is met. This is inherently less flexible than a system that gives control back to the user after each movement. In the latter, the user is at liberty to do whatever he wants; for example, he could restart the search from a previously recorded point, change the size of a tabu list, switch to a different neighborhood etc. Without such control between moves, such changes need to be anticipated by the system designer, and done through *call-backs* (pieces of code which are attached to the engine and called after each move). The call-back method, however, is often limited. For instance, if the system designer did not envisage that one could switch neighborhood during search, and did not provide a method for doing this from the call-back, then it is simply impossible.

- **Tree-based search used to find move.** The traditional tree-based search methods of finding solutions to a problem are used to find an acceptable neighbor to move to at each step in the local search process. That is, instead of solving the whole problem using tree-based search, we solve the subproblem of finding an acceptable neighbor given the current solution and a specification of the neighborhood. When a local search is performed, a series of complete searches (one per move made) are launched, each one resulting in the new neighbor to move to from the current solution. This structure is quite appealing because of the fact that the search for a neighbor is an ordinary constraint programming search goal. Thus, it can be mixed with other constraint programming goals to produce methods that perform local and complete search together. An example of how this can be done is presented in Section 8.4. The idea of exploring a neighborhood using constraint programming was first introduced by Pesant and Gendreau (1996) and Pesant and Gendreau (1999).

- **Open local search objects.** Concepts in the local search are not only represented as objects: in the case of neighborhoods and meta-heuristics, these objects are extensible. That is, the user is at liberty to define his own neighborhood and meta-heuristics for a specific problem or problem domain. This means that the toolkit can address problems which were either not thought of or were too rarely occurring for the appropriate support to be put into a general tool. Extensions to local search objects are discussed in detail in Section 8.5.

ILOG Solver's fundamental local search objects are *solutions, neighborhoods,* and *meta-heuristics,* whose roles will now be described.

8.3.2 Solutions

Solution objects are simple containers that hold variables in the ILOG Solver model, together with their values corresponding to a solution. Often the value of the objective variable is also stored in the solution so that the quality of the solution can be easily accessed without reference to a solving engine to compute it from the values of the decision variables. Solutions are objects which perform several functions within ILOG Solver.

The first function has little to do with local search: solution objects can be useful for accessing more than one solution to a problem, which can be inconvenient using a single solver which can only be in a single state at a time. Decision and non-decision variables can be present in the solution, meaning that the values of all variables can be read off from the solution, including those whose values are a consequence of the values of the decision variables. In a local search context, a solution object is often used to preserve the best solution seen during a run.

Second, solution objects are used to represent the *current solution* to a problem in a local search context. Here, the solution contains only *decision* variables of the problem, as these are the ones which will be changed by the neighborhoods. Moves will be explored from this current solution and eventually one accepted. The action of accepting a move changes the current solution.

Third, and less obviously, solution objects are used to specify neighborhoods. Each neighbor in the neighborhood is described by a *solution delta.* The solution delta is simply a solution object which contains a subset of the decision variables. This subset is exactly the set of variables which change their values at this neighbor, together with their new values. Solution deltas will be elaborated on in the next section on neighborhoods.

An example of an ILOG Solver code which uses a solution object is shown below:

```
01. IloEnv env;
02. IloModel mdl(env);
03. IloIntVar x(env, 0, 2), y(env, 0, 1);
04. mdl.add(x < y);
05. IloSolution soln(env);
06. soln.add(x);
07. soln.add(y);
08. IloSolver solver(mdl);
09. solver.solve();
10. soln.store(solver);
11. solver.end();
12. cout << "X = " << soln.getValue(x) << endl;
13. cout << "Y = " << soln.getValue(y) << endl;
```

ILOG Solver is based on ILOG Concert Technology, a modeling layer enabling different solving engines to solve the same model. All Concert objects begin with the prefix `Ilo`.

Line 1 of the code creates the Concert *environment* which is an object used for memory allocation and bookkeeping. It is invariably passed when modeling objects are created. Line 2 then creates a Concert model. A model is essentially a bag of modeling objects such as variables and constraints. Line 3 creates two variables which can take only integer values: `x` between 0 and 2, and `y` only the values 0 and 1. Line 4 adds a constraint to the model indicating that x must be strictly less than y. When a constraint is added to the model, any variables mentioned do not need to be added; the solving engine finds them automatically. Lines 5–7 create a solution and add the two variables to it. This means that when the solution is stored, the values of variables `x` and `y` will be preserved within the solution. Line 8 creates the solving engine from the model; this invokes a procedure known as *extraction* which takes the concert model and translates it into structures suited to the solving engine.

Line 9 solves the problem using a *default goal.* (It is also possible to pass a goal to `solver.solve` and we shall see this later.) Solver will deliver the first solution it finds which satisfies the constraints of the problem. As it happens, there is a unique solution (`x=0, y=1`). After the solve, solver is in the *solution state*: we have direct access to variable values directly from the solver, for instance via: `solver.getValue(x)`. However, when we use the solver to solve another problem, the state of the solver will change, and the present solution will be lost unless we store it in a solution object.

Line 10 stores the solution by transferring the values of the variables as set by the solving engine to the solution object. Solution objects are invariant to changes of state in the solver and are thus preserved when the solver changes state or is destroyed. Line 11 destroys the solver. The solution is still preserved in `soln` as lines 12 and 13 demonstrate by printing the stored values of `x` and `y`.

The program produces the following output:

```
X = 0
Y = 1
```

8.3.3 Neighborhoods

A neighborhood in ILOG Solver is the structure which determines the range of solutions one can move to from the current solution. The neighborhood needs not guarantee that each neighboring solution respects the constraints of the problem nor that it meets some acceptance criterion (for example improving the quality of the solution); these are the jobs of the constraint solving engine and the meta-heuristic (see next section), respectively. The requirements of a neighborhood are quite simple and are summarized as follows:

- **The neighborhood receives the current solution.** This solution is the one from which all neighbors are specified. The neighborhood must keep this solution as a reference from which to define neighbors. The current solution is supplied to the neighborhood once for each move.

- **The neighborhood can determine its size s.** Normally, this is determined when the neighborhood receives the current solution.

- **The neighborhood can deliver neighbor i, where $0 \leq i < s$.** Neighborhoods have a random, not sequential, access interface. That is, a neighborhood must be capable of delivering the *i*th neighbor at any time. The reason for this is it vastly increases the flexibility of neighborhoods, as will be demonstrated later. A neighbor is defined by delivering a *solution delta*: the fragment of the solution which is changed by the neighbor, together with the new values for the changed part.

For example, suppose we have a neighborhoodwhich changes the value of one of a set of 0-1 variables, a so-called "flip" neighborhood. If there are n variables, then there will be n neighbors in the flip neighborhood.Neighbor i is defined by delivering a solution object containing only variable i. Its value in the solution object should be the value different from that in the current solution.`soln.setValue(var, value)` sets the value of variable`var` to be `value` in solution`soln`.

In addition:

- **The neighborhood is notified that neighbor j was chosen.** This is not a requirement of the neighborhood, but a service to it. When a move is made the neighborhood is told the index of the neighbor which was accepted, and the constrained variables are in a state which represents the new accepted solution. Normallynotificationdoes nothing unless the neighborhood desires it. However, receiving the notification that a particular neighbor was taken can be useful in adaptive neighborhoods where some state is incrementally maintained.

We now present a code segment showing the use of neighborhoods. In order to demonstrate how a neighborhood operates, we have to have some mechanism for "driving" the neighborhood. That is, for sending it the currentsolution, and retrieving the deltas. The pre-defined Solver goal `IloScanNHood` does just this, and is used in the following code.

```
IloEnv env;
IloModel mdl(env);
IloIntVarArray vars(env, 3, 1, 3);
mdl.add (vars);
mdl.add(a[0] < a[2]) ;
IloSolution soln(env);
soln.add(vars);
soln.setValue (vars[0], 1);
soln.setValue (vars [ 1 ] , 2);
soln.setValue (vars[2], 3);
IloNHood nhood = IloSwap(env, vars);
IloGoal scan = IloScanNHood(env, nhood, soln);
IloSolver solver(mdl);
soln.startNewSearch(scan);
while (solver.next())
```

```
16. cout << solver.getValue(a[0])
         << solver.getValue(a[1])
         << solver.getValue(a[2]);
```

In this example, we create a model with three variables in an array, each of which can take on the integer values 1, 2 or 3. The variables are added to the model as well as a constraint stating that the first variable in the array must be strictly less than the last variable. From lines 6–10, we then create a solution object and populate it with all variables, the values of these variables in the solution being set to 1, 2 and 3, respectively. We can see that this is a true solution as it does not violate the only constraint `a[0] < a[2]`. Line 11 creates a pre-defined Solver neighborhood which interchanges the values of any two variables in the array. Line 12 creates the Solver search goal which will exercise this neighborhood, taking each neighbor in turn from the neighborhood and instantiating the variables in `vars` according to the deltas delivered. Line 13 creates the solver from the model. Using `solver.solve(scan)` would produce only a single solution to the neighborhood exploration problem. That is, only the first neighbor would be produced. However, to demonstrate how the neighborhood is scanned, we will produce all solutions to the neighborhood scanning problem. This is achieved by a different set of member functions. `solver.startNewSearch(scan)` primes the solving engine with the goal `scan`. Thereafter, each call to `solver.next()` produces a new solution until a false value is returned, at which point all solutions have been found.

The program produces the following output:

```
213
132
```

Note that the "swap" neighborhood should produce exactly $n(n-1)/2$ neighbors, given an array of size n. In this case three neighbors should be produced. In fact, the constraint `vars[0] < a[2]` means that swapping the values of the first and last elements of the array is forbidden, and such a move is automatically rejected by the constraint solver when an attempt is made to instantiate the variables with the values 321.

Powerful operators and functions are also available to manipulate neighborhoods. The most useful of these is the concatenation operator. Writing `nhood1 + nhood2` produces a neighborhood whose neighbors are made up of those in `nhood1` and `nhood2`, the new neighborhood producing those in `nhood1` first.

Another very useful operator is one which randomizes a neighborhood. Writing `IloRandomize(env, nhood, randGen)` randomly jumbles the neighbors of `nhood` at each move, using random numbers drawn from the generator `randGen`, which is an instance of the Concert class `IloRandom`. Drawing neighbors in a random manner from a neighborhood can be particularly useful in avoiding stagnation and looping in meta-heuristics such as tabu search. The fact that neighborhoods can be randomly accessed means that such a randomization can be efficiently implemented. The newly created neighborhood merely contains a mapping table between the indices of its neighbors and those of `nhood`. Such a mapping is randomly re-created at each move.

To give another example, the neighborhood `IloContinue(env, nhood)` is a neighborhood which behaves as `nhood` except that when a move is made, neighbors for the next move are cyclicly indexed from the index of the neighbor that was taken. So, for instance, for a neighborhood which flips the values of n 0-1 variables, if variable i (indexed from 0) was the last variable flipped, at the next move, flips will be explored from variable $i+1$ though to variable i, cycling back to variable 0 after variable $n-1$. This type of neighborhood can be useful as it is often more profitable to explore new neighbors than re-explore old ones. To indicate the uses of neighborhood notification, `IloContinue` is a neighborhood which makes use of notification to change its behavior (to offset the neighbors by the appropriate index) for the next move.

8.3.4 Meta-Heuristics

What is termed a *meta-heuristic* in ILOG Solver can have a meaning which is somewhat different from the commonly used term. The term “meta-heuristic” in the literature is quite a general concept which normally means a method which guides the application of local improvement methods. “Guides” in this context can apply to quite a general set of behaviors including forbidding certain types of moves, encouraging others, restarting the search from some point, perturbing the search in some “intelligent” manner, changing neighborhoods, etc.

In ILOG Solver, a *meta-heuristic object* is an object which performs a much more precise role: it filters neighbors and nothing more. That is, a meta-heuristic can say that certain neighbors are forbidden. This is not as restrictive as one might imagine. As control is given to the user between moves, other types of meta-heuristic behavior can be implemented at that point, including restarting from a previous good solution, switching neighborhoods, changing the tabu tenure in a tabu search method etc. Such is the benefit of having control between moves: it alleviates the problem of trying to “black box” such greatly varying behaviors in the meta-heuristic object itself.

Meta-heuristics nearly always have *state*: for example the tabu list for a tabu search mechanism or the current temperature for a simulated annealing meta-heuristic. It is this state that allows the meta-heuristic to change its filtering from move to move.

Note that Solver’s meta-heuristics do not decide which neighbor to choose from among the neighborhood. This decision is left to another object known as a *search selector* which will be introduced in Section 8.3.5.

In order to fulfill the above filtering role, a meta-heuristic must perform the following tasks:

- **The meta-heuristic receives the current solution.** Almost always, the meta-heuristic will make use of the cost of the current solution which it can glean via `soln.getObjectiveValue()`. The current solution is supplied to the meta-heuristic once for each move.

  For meta-heuristics objects, often the most natural way to filter certain moves is to add a constraint to the constraint solver forbidding them. This is usually possible if the forbidden moves have some sort of logical structure to them. For instance, although usually not considered a meta-heuristic, a greedy improvement search is implemented in Solver by the meta-heuristic class `IloImprove`. When an object of this type receives the current solution, it takes the objective from the current solution and adds a constraint to the solver to state that any new solution (i.e. neighbor) found must have a cost which is lower (when minimizing). Then, all neighbors not strictly decreasing the cost will automatically be rejected by the constraint solver.

- **The meta-heuristic rejects or allows a given neighbor.** When the neighbor filtering cannot conveniently be expressed as a constraint, the meta-heuristic can reject the neighbor in an alternative manner. The meta-heuristic has the neighbor presented to it with the decision variables instantiated as they were for the previous code segment demonstrating `IloScanNHood`. In this case, the meta-heuristic has all information regarding the neighbor available, including the values of non-decision variables and the cost of the neighbor. If the filtering was difficult to express as a constraint, then an algorithm can be executed at this point to determine if the neighbor should be filtered or not.

  Note that this "test after instantiation" method of filtering neighbors is not as efficient as using a constraint as all decision variables must be instantiated before the test is executed. When the neighbors to be filtered are expressed as a constraint, then they may be filtered before all variables are instantiated.

In addition:

- **The meta-heuristic is notified when a move is made.** Like the notification of neighborhoods, this is not a requirement of the meta-heuristic, but a service to it. When a move is made the meta-heuristic is told the solution delta of the neighbor which was accepted, and the constrained variables are in a state which represents the new accepted solution. When a *neighborhood* is notified, the norm is that nothing happens, as neighborhood objects do not normally have state (unless they are adaptive as previously discussed). However, as it is normal for meta-heuristics to have state, typically the meta-heuristic changes its state when notified. As an example, in a simulated annealing meta-heuristic, the temperature would be decreased. Or in a tabu search meta-heuristic, some attributes would be dropped from the tabu list (the oldest ones), and others would be added according to the move which had just been made.

  If no neighbors could be legally taken, either because all were illegal or were forbidden by the meta-heuristic, then the meta-heuristic is informed of this fact. The meta-heuristic can then take action at this point to reduce its filtering such that when subsequent moves are attempted, not all will be filtered and progress can be made once more. For example, in the tabu search meta-heuristic supplied with ILOG Solver, when no moves can be taken, the oldest attributes are dropped from the tabu list, without adding any new attributes. This exactly mimics the method suggested in Glover and Laguna (1997) for managing the situation where all legal moves are tabu.

Meta-heuristics, as neighborhoods, can be composed. The expression `mh1 + mh2` results in a meta-heuristic which filters a neighbor if either `mh1` or `mh2` would have

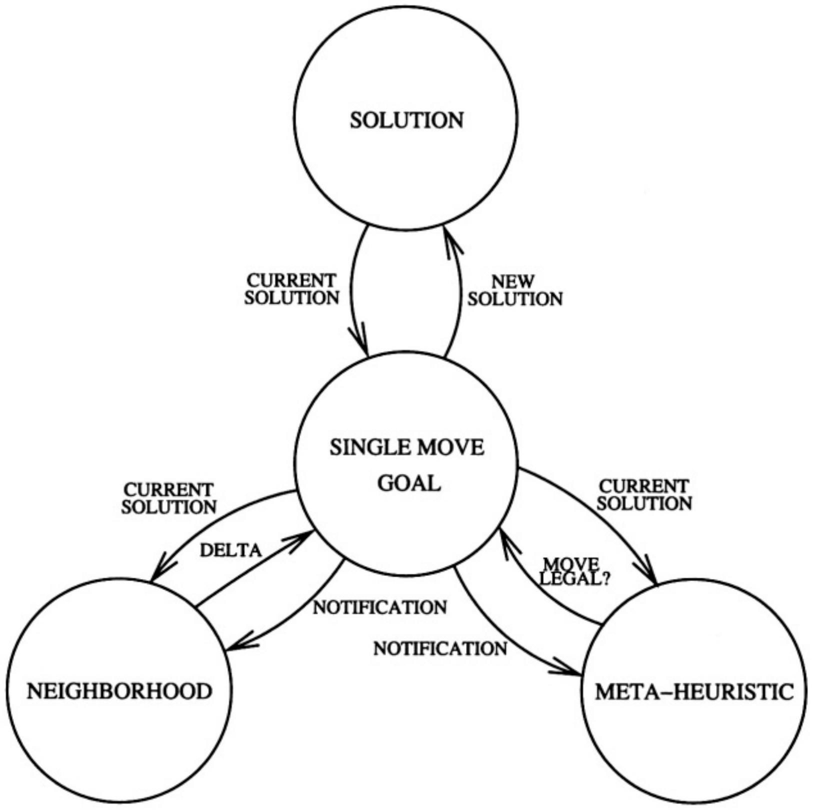

Figure 8.3 Interaction of Local Search Objects

filtered the neighbor alone. Such composition can be useful for creating composite meta-heuristics such as guided tabu search (de Backer et al. (2000)).

The goal `IloScanNHood` was used to explore the neighbors of a neighborhood. In order to combine neighborhoods and meta-heuristics, we use the Solver goal named `IloSingleMove` which combines a neighborhood and a meta-heuristic in order to make one local move in the search space. The operation of this goal is shown in Figure 8.3. We depict the main actors which are a solution, a neighborhood, a meta-heuristic, and the goal itself.

First of all, the current values of decision variables are fetched from the solution after which they are sent to both the neighborhood and the meta-heuristic. The size s of the neighborhood is then fetched (not shown in the figure). The goal then begins asking the neighborhood for solution deltas corresponding to neighbors for indices 0 through $s-1$. For each neighbor, the goal calculates the new values for all constrained variables according to the current solution and the solution delta. The variables are then instantiated with these values and the meta-heuristic is asked if the neighbor should be filtered. (Note that if the meta-heuristic specified its filtering via a constraint, and the neighbor is one which is to be filtered by the meta-heuristic, then the goal will never get to the point where it has to ask the raeta-heuristic to check the solution, as the violated constraints would cause a failure before this point.) If the neighbor should be filtered the goal causes a *failure*; otherwise, the goal succeeds in creating a leaf node (legal neighbor).

- In the case where the first legal neighbor of the neighborhood should be taken, the goal move stops at that leaf node: the remainder of the neighbors are not examined.
- In the case where, for instance, the neighbor improving the quality of the solution by the greatest amount is to be chosen, the goal continues examining neighbors but logs the best leaf node it has visited. When the whole search tree corresponding to the neighborhood has been examined, Solver jumps back to the best leaf node. The mechanism by which such selections are made is described in the next section.

In either of the above cases, when the preferred neighbor has been reached, both the neighborhood and meta-heuristic are notified of this. Then the new current assignment of decision variables is stored back in the solution. The performance of multiple moves is effected by repeating the entire process.

A detailed description of the implementation of the underlying mechanisms behind neighborhood exploration goals is given in Shaw et al. (2000).

8.3.5 Selection of a Neighbor

Above, we touched upon different methods of *selecting* a neighbor. The job of selecting which legal neighbor to move to is not performed by any local search object we have yet described. In ILOG Solver, this selection is performed by an object known as a *search selector.* Search selectors are not specific to local search but are objects which can be applied to any search goal.

For instance, consider a standard combinatorial problem with multiple solutions and a goal `g` which searches for those solutions. To find all such solutions, we could use `solver.startNewSearch(g)` and `solver.next()`. Imagine that we wanted to find the solution which minimized the value of some variable `v`. This could be done by using an instance of `IloSolution` to store the solution after each successful call to `solver.next()` when the new solution produced has a value of `v` smaller than the value of `v` in the solution. After the search tree has been fully explored the best solution is preserved.

The above mechanism is quite cumbersome; Solver provides a simpler way to search for the solution we are interested in. A *search selector* transforms a goal into one where only the solutions of interest form leaf nodes. For instance, the line

```
g = IloSelectSearch(g, IloMinimizeVar(env, v));
```

transforms the goal `g` such that it will produce only one leaf node: that which minimizes the value of `v`. This exactly mimics the process described in Section 8.2.3. Solver has other search selectors, notably `IloFirstSolution` which keeps only the first solution produced by the original goal. Finally, the users are at liberty to write their own search selectors to perform the selection they require. For example, “take the first solution of cost less than *c,* but if no such solutions exist, deliver the solution with lowest cost”.

One can see that as goals such as `IloScanNHood` are standard Solver goals, one can apply search selectors to them,too. In the context of local search, a selector to choose the best solution chooses the best neighbor.

The implementation of Solver’s search selectors is described in ILOG (2000b).

8.3.6 Simple Local Search Example

We now present a complete local search example. We have chosen the well-known *one-max* problem often used for demonstrative purposes. In a one-max problem we are given a set of n 0-1 variables and we are to find a solution with the maximum number of variables set to 1. Of course, thisproblem is easy, but the local search example which solves it demonstrates all the basic features we are interested in.

To begin, we create an environment, a model, and an integer determining the size of the problem.

```
01. IloEnv env;
02. IloModel mdl(env);
03. IloInt n = 5;
```

We now fill in the details of the model. We first create the set of 0-1 variables `vars` in line 4. Then, a variable which will represent the sum of the number of ones is created. It is constrained to be equal to the sum of the array `vars` by adding a constraint to the model.

```
04. IloIntVarArray vars(env, n, 0, 1);
05. IloIntVar ones(env, 0, n);
06. mdl.add(ones == IloSum(vars));
```

We next create a solution object to use in the local search optimization. We add the decision variables to the solution and an objective, which is to maximize the number of ones.

```
07. IloSolution soln(env);
08. soln.add(vars);
09. soln.add(IloMaximize(env, ones));
```

The model and solution specification is now complete. A solver is thencreated, and an initial solution generated using the pre-defined Solver goal `IloGenerate.` This goal instantiates variables in the order they appear in the array and chooses low values before higher ones. Thus, the initial solution that results is one where all variables have value zero. The initial solution is stored in the solution object via `soln.store`.

```
10. IloSolver solver(mdl);
11. solver.solve(IloGenerate(env, vars));
12. soln.store(solver);
13. DisplaySolution("Initial Solution: ", solver, vars);
```

`DisplaySolution` is a function which prints a message followed by the values of the decision variables. The values of the decision variables are retrieved from the solver using `solver.getValue`. The function is defined as follows:

```
void DisplaySolution(const char *message,
                     IloSolver solver,
                     IloIntVarArray vars) {
  cout << message;
  for (IloInt i = 0; i < vars.getSize(); i++)
    cout << solver.getValue(vars[i]);
  cout << endl;
}
```

After the initial solution has been generated and stored in the solution, we can progress to the local search. Line 14 creates the neighborhood that we will use: `IloFlip`. This neighborhood can change the value of one of the 0-1 variables in the array handed to it, and is thus capable of moving to the optimal solution which consists purely of ones. Line 15 creates the meta-heuristic, `IloImprove`, which filters any move which does not improve the quality of the solution. In our case, the objective variable, `ones`, must increase at each move. Finally, in line 16, we create the goal which makes one local search move. To create this goal we pass the environment, solution, neighborhood, and meta-heuristic objects.

```
14. IloNHood nhood = IloFlip(env, vars);
15. IloMetaHeuristic greedy = IloImprove(env);
16. IloGoal move = IloSingleMove(env, soln, nhood, greedy);
```

All necessary objects are now prepared, and we enter the loop which improves the solution. The `move` goal is executed repeatedly until it fails, with such a failure indicating that no moves which improve the solution could be found. This means that we have reached a local maximum. In this case, the local maximum coincides with the global maximum.

```
17. while (solver.solve(move))
18.   DisplaySolution("", solver, vars);
```

After the optimization loop, the final solution found is held in the solution object. However, if we wish this solution to be accessible from the solver, the constrained variables need to be instantiated with the values in the solution. This is carried out by the goal`IloRestoreSolution`.

```
19. solver.solve(IloRestoreSolution(env, soln));
20. DisplaySolution("Final Solution: ", solver, vars);
```

The whole program produces the following output. As expected, the number of ones increases monotonically. The moves generated by the "flip" neighborhood which change a 1 to a 0 are filtered out as they decrease the objective.

```
Initial Solution: 00000
10000
11000
11100
11110
11111
Final Solution: 11111
```

8.3.7 Discussion

We have presented a toolkit for local search built upon, and now part of, ILOG Solver. The toolkit is based on certain basic design principles: an object model of local search, user control between moves, neighborhood exploration using goals (facilitating meshing between local search and constraint programming), and an open interface.

Each entity is accorded its own specific task, and there is a clean separation between such tasks. *Solution* objects store the values of decision variables and the value of the objective. *Neighborhood* objects are responsible for defining the neighbors of the current solution. *Meta-heuristic* objects filter the neighbors of the current solution. *Search selector* objects choose one of the possible neighbors to move to. Finally, *Local search goals* bring all of these objects together to make moves in neighborhood space.

The adoption of an object model results in a flexible language with which to express local search concepts. For instance, the fact that neighbors should be drawn from a neighborhood in a random fashion is made part of the neighborhood itself, through the use of the neighborhood operator `IloRandomize`. `IloContinue` provides another way of exploring the neighborhood. Neighborhood modifiers are inherently more natural than different possibilities for exploration built into the neighborhood exploration goal (cf. neighborhood iterators in Fink et al. (1999b)).

Likewise, solution objects are useful as they can be cloned, copied, arrays of them kept, sorted etc. The user is not limited to the traditional "current solution" and "best solution". One possibility is to keep a library of elite solution fragments which can then be used to produce new solutions (see, e.g., Rochat and Taillard (1995)). Finally, meta-heuristic objects provide a simple protocol for directing the search via neighborhood filtering.

Operators over meta-heuristics provide methods to combine them: for instance, addition of meta-heuristics form a meta-heuristic which filters a neighbor if any meta-heuristic in the sum would filter the neighbor. Such simple combinations can allow one to produce new meta-heuristics: for instance, guided tabu search can be created from a combination of tabu and guided local search (de Backer et al. (2000)).

The advantages of including local search support in a constraint programming library have not yet been emphasized, and it is perhaps a fitting time to do so. The obvious alternative is to produce a "stand alone" local search toolkit: several academic toolkits are already available; see, e.g., Di Gaspero and Schaerf (2001), Fink et al. (1999b), Michel and van Hentenryck (1997), Michel and van Hentenryck (2000), IBM Open Source Software (2001). The advantages of instead using a constraint programming library as the base are two-fold. The first advantage is the flexibility of modeling provided by constraint programming libraries. Such modeling flexibility is unrivaled by any other technology. The problem variables and constraints can be described in a concise readable form and tools like ILOG Concert Technology (ILOG (2000a)) make it easy to manipulate and decompose models. Secondly, constraint programming comes equipped with another solving method distinct from local search: that of tree-based search. Such methods tend to work well when constraint propagation for a particular problem is strong, while local search methods often fill the gap when propagation is more limited. Thus, between the two a greater solution coverage for real-world problem solving is achieved. A software engineering advantage is that one can run either a complete or local search on the same model, a benefit advocated in Pesant and Gendreau (1999).

Allied to the advantage of the availability of depth-first search mechanisms is the need for an *initial solution* for local search problems. When one is dealing with unconstrained optimization, generating a first solution is not problematic: a random assignment suffices. For problems with constraints, the process of producing a legal first solution is not so clear. Commonly, a specific first solution method is used, but this may then break down if one or two additional side-constraints are added (see, for example, Kilby et al. (2000)). An alternative is to soften the constraint and value its violation. This is often useful, but in many cases results in a loss of search direction, with the local search spending large portions of its time trying to re-establish legality. (By contrast, when using constraint programming constraints can be imposed as hard constraints, or softened and added to the cost function, as desired.) Constraint programming offers a largely constraint-independent method of generating first solutions: tree-search with constraint propagation. Even if for the type of problem considered, constraint propagation is too weak to find optimal or very good solutions, it is often sufficient for finding legal first solutions. These solutions can subsequently be improved by local search.

The benefits of constraint programming's tree-based search approach also aids local search in another important way: we can use complete and local search together. There are various ways in which this can be done. One might simply want to find a good solution using local search which then provides an upper bound on the cost function for a proof of optimality by complete search. More sophisticated is to run local and complete search in parallel. We can communicate the upper bound from the local to the complete search, and communicate solutions found in the other direction. Local search can make use of these as elite solutions; for instance, they may make good points for strategic restart. Finally, we can couple both search methods together more tightly. One method of doing this is via a method known as *shuffling* (Adams et al. (1988), Applegate and Cook (1991)), referent domain optimization (Glover and Laguna (1997)), or large neighborhood search (Shaw (1998)). In this method each local move performs a relaxation of parts of the current solution, followed by a re-optimization of respective parts via complete search. This technique is very natural using a constraint programming library, and results achieved can be highly competitive; see, e.g., Rousseau et al. (2000), Shaw (1998). A demonstration of a quite different hybrid search is given in Section 8.4.5.

An obvious disadvantage of the integration of local search mechanisms into a constraint programming library is that such libraries are typically non-monotonic and perform constraint checking by propagation. Both of these attributes result in local searches which have significant computational overhead compared to approaches which do not have to deal with backtracking and propagation. We have described in Shaw et al. (2000) how to minimize these overheads via specific tree-search mechanisms. It is our belief that for complex real-world problems, the flexibility of the constraint programming library – with its ability to mix search methods – will more than compensate for such inefficiencies.

8.4 INDUSTRIAL EXAMPLE: FACILITY LOCATION

Facility location problems are of significant practical importance in transportation logistics. They concern the allocation of customers to facilities in a cost-effective manner. The facilities perform some service to the customers, be it supply of goods, pickup of goods, maintenance of equipment at the customer site etc. It is often the case that the "customers" and the "facilities" form different parts of the same company.

The particular problem we address here is posed as follows. Given a set of customers to be served, and a set of possible locations where facilities can be constructed, decide which sites to use for the construction of a facility, and which customers to serve from each facility. Each customer has a demand which is the amount of service required, and must be served from only one facility. Each facility has a capacity, dependent upon the construction site, which must be no less than the sum of the demands of the customers served by it. The construction of a facility entails a certain site-dependent cost, while the cost of serving a particular customer is dependent upon which facility serves it. Thus the optimization is a balance between reducing the opening costs and service costs. The former is best done by opening few facilities, whereas the latter is best done by opening many, such that distances from customer to facility are reduced.

Although the basic problem can be enriched with various types of additional constraints (which makes solving it using constraint programming all the more interesting), for simplicity we treat only the basic problem described above.

8.4.1 Problem Description

We are given a set of customers $\mathcal{C}$ and a set of possible facility locations $\mathcal{F}$. Each customer $c \in \mathcal{C}$ demands D_c of service. Each potential facility $f \in \mathcal{F}$ has capacity

C_f. The cost of constructing facility f is given by $cost(f)$ and the cost of serving customer c from facility f is $cost(c, f)$.

We represent a solution by a variable x_c for each customer c, indicating which facility serves that customer. Given such a representation, the constraint that each customer must be served from exactly one facility need not be modeled explicitly. The model can be expressed mathematically as:

Minimize:

$$\sum_{c \in C} cost(c, x_c) + \sum_{\{f \mid \exists c.x_c = f\}} cost(f)$$

Subject to:

$$\forall f \in \mathcal{F} \quad \sum_{\{c \mid x_c = f\}} D_c \leq C_f$$

An ILOG Solver program which implements this abstract model is now given. We assume that the costs of building each facility are present in an array `bc`, while the costs of serving customers from facilities are held in a two dimensional array `sc`. The capacities of the facilities are held in the array `capacities`, while the demands of the customers are held in the array `demands`. With regards to problem size, `nbFac` holds the possible number of facility locations, while `nbCusts` indicates the number of customers to be served.

First of all the model, the basic decision variables, and variables which determine the load on each facility are created. The array `x` models the assignment of customers to facilities, and so each variable in this table is created with a domain from 0 to one less than the number of facilities. The `load` array will later contain variables representing the load on each facility. We assume the environment `env` has already been created.

```
IloModel mdl(env);
IloIntVarArray x(env, nbCusts, 0, nbFac - 1);
IloIntVarArray load(env, nbFac);
```

Now, we create the variables and constraints which maintain and constrain the load on each facility. This is done via a set of intermediate 0-1 variables `here`, which delivers an encoding of the set of customers assigned to a facility. The load on the facility is then formed by the scalar product of this array and the array of demands. Notice that for constraints to take effect, they must be added to the model of the problem `mdl`. Finally, we add a constraint stating that the sum of the loads on the facilities must equal the sum of the demands. Although this constraint is not strictly required, in a constraint programming context, it can help to prune out some possibilities earlier in the search.

```
IloInt i, j;
for (i = 0; i < nbFac; i++) {
  load[i] = IloIntVar(env, 0, capacities[i]);
  IloIntVarArray here(env, nbCusts, 0, 1);
  for (j = 0; j < nbCusts; j++)
    mdl.add(here[j] == (x[j] == i));
10.  mdl.add(load[i] == IloScalProd(here, demands));
   }
11.mdl.add(IloSum(load) == IloSum(demands));
```

We next add the constraints which maintain the cost of serving each customer from their chosen facility. We use the Solver constraint `IloTableConstraint`, which indexes an array of constants using a constrained variable as the index. On line 12, the array `cc` holds the costs of serving all customers. The variables in `cc` are declared as variables which can have discrete integer values. For instance, `cc[0]` can only take on values which are the costs of serving customer 0 from each facility. In general, the domain of variable `cc[j]` is thus defined to be `sc[j]`, the possible costs of serving customer j (line14). Thetableconstraint then constrains `cc[j]` to be equal to `sc[j][x[j]]` (line 15).

```
12. IloNumVarArray cc(env, nbCusts);
13. for (j = 0; j < nbCusts; j++) {
14.  cc[j] = IloIntVar(env, sc[j]);
15.  mdl.add(IloTableConstraint(env, cc[j], sc[j], x[j]));
   }
```

Having formed the cost of serving each customer from its chosen facility, we must also form the cost of building the facilities. The 0-1 array `open` indicates which facilities are open: namely those which have a non-zero load. Finally, the total cost of the problem is formed from the sum of the costs of serving the customers and the costs of building the open facilities. The latter term is formed by a scalar productbetween the `open` array and the build costs `bc`.

```
16.  IloIntVarArray open(env, nbFac, 0, 1);
17.  for (i = 0; i < nbFac; i++)
18.    mdl.add(open[i] == (load[i] != 0));
19.  IloIntVar cost(env, 0, IloIntMax);
20.  mdl.add(cost == IloSum(cc) + IloScalProd(open, bc));
```

8.4.2 Solution Methods

We present three alternative approaches to solving the above model using ILOG Solver. The first is to use a standard tree search to find the optimal solution. The second is to use a pure local search method where customers are moved from facility to facility. In this second method we use a short term tabu memory. The third is a mixture of local search and tree-based search methods. We describe each of these in more detail, giving example code of how each is implemented.

Before we begin describing the three methods in more detail, we first introduce some common objects we will need for all methods via the following code:

```
IloObjective obj = IloMinimize(env, cost);
IloSolution best(env);
best.add(obj);
best.add(x);
IloGoal assign = AssignServers(env, x, open, sc, bc);
```

Three objects are created:

- The objective of the problem: minimize the total cost.
- A solution to hold the best solution found to the problem. This solution keeps a note of the facility assigned to each customer and the objective value (via the `add` method).
- A tree-search goal to instantiate the `x` variables. We do not give the code for this goal but describe it.

  The goal makes branching decisions of the form "either customer c is served by facility f, or c is not served by f". c is chosen before f to be the customer with the smallest number of facilities that can serve it. f is then chosen to be the least costly facility where the cost of the facility is taken to be `sc[c][f]`, plus `bc[f]` if no customers are as yet served by f. Ties in the choice of c and f are broken by choosing the lowest indexed customer or facility.

8.4.3 Tree-Based Search

Most of the work in describing a tree-based search is the definition of the appropriate search goal, whose behavior we have just described. Once this is done, search is quite simple. Here, we add the objective to the model to indicate that we are looking for a solution with minimum cost. Then an instance of the class `IloSolver` is created to solve the model, and is primed with goal `assign`. The solutions to the problem are then generated using the `next()` function of the solver. As we are minimizing cost, each new solution produced by `next()` is guaranteed to have better cost than the previous one, such that the last one produced will be the optimal. The solution `best` is stored each time a solution is found, and so the optimal will be preserved therein.

```
mdl.add(obj);
IloSolver solver(mdl);
solver.startNewSearch(assign);
while (solver.next())
   best.store(solver);
```

8.4.4 Pure Local Search

A local search method which can be used on this problem is one where the facility assigned to a customer is changed. One can view this as moving a customer to another facility. The neighborhood is then the movement of each customer to all facilities other than its current one. Such a neighborhood is of size $O(|\mathcal{C}||\mathcal{F}|)$. In order to reduce the probability of becoming trapped in local optima, a short-term tabu memory is used which prohibits a customer moving back to a facility it was moved from for a certain number of iterations.

The first step in any local search, however, is to build a first solution. We achieve this via the following code (line numbers continue from 26 as this code is an alternative code to the tree-based search):

```
26. IloSolver solver(mdl);
27. if (!solver.solve(assign)) {
28.   cout << "No solution" << endl;
29.   env.end();
30.   return 0;
    }
31.   best.store(solver);
```

Unlike the tree-based search, we do not add the objective to the model. To do so would mean that we would try to find an optimal solution on line 27. Here, we are interested in any solution satisfying the constraints. If no initial solution could be found, we exit the program.

Next, we move onto the local search mechanism itself. First, we create the neighborhood. Solver's neighborhood `IloChangeValue` has neighbors in which each variable of a specified set can change its value to one within a specified range. When applied to the variables `x` which decide which facility serves each customer, this neighborhood corresponds to the movement of a customer from one facility to another. We also randomize this neighborhood. The reasons for this will be explained below.

```
32. IloNHood nh = IloChangeValue(env, x, 0, nbFac - 1);
33. IloRandom rnd(env);
34. nh = IloRandomize(env, nh, rnd);
```

We use Solver's built-in tabu search meta-heuristic which is based upon a short term tabu memory of assignments in the current solution. Any assignment which is undone cannot be re-done within the specified tabu period. The tabu period can be adjusted dynamically by the user using a simple interface. The meta-heuristic also includes an aspiration criterion which means that the tabu status of a move is overridden if it would result in a move which is better than any seen so far. Here, we create a tabu search with tenure 15. This means that if a facility f was assigned to a customer, and this assignment is changed, the f cannot be reassigned to the customer for 15 iterations.

```
35. IloMetaHeuristic mh = IloTabuSearch(env, 15);
```

Finally, we construct the goal which performs a local search move. For this, we need a solution object to hold the current solution. This is done on line 36 by cloning the best solution object `best`. We also specify that we would like to move to the minimum cost neighbor (which is non-tabu) via Solver's `IloMinimizeVar` search selector (line 37). This selector specifies that the goal willgenerate a maximum of one leaf node, and that node will be the one minimizing a specified variable; in this case the total cost. If more than one neighbor has the same cost, the first one encountered will be chosen. It is for this reason that the neighborhood was randomized earlier: it removes any structure or implied order in the neighborhood such that if there is such a tie-break, a neighbor minimizing the cost will be chosen randomly. On line 38, the `IloSingleMove` goal takes the solution, neighborhood, meta-heuristic and best-neighbor selector and produces a goal which will make the move. We attach to this another goal which stores the new solution in `best` if it is better than `best`, thus automatically keeping the best solution found up to date.

```
36. IloSolution soln(best.makeClone(env));
37. IloSearchSelector selMin = IloMinimizeVar(env, cost);
38. IloGoal move = IloSingleMove(env, soln, nh, mh, selMin)
                 && IloStoreBestSolution(env, best);
```

The local search loop can then be entered. In this example, up to thirty moves are made, with the cost of the current solution and best solution displayed at each stage. If at any stage a move could not be taken, this could mean that all moves are tabu. In this case, we make a call to the `complete()` method of the meta-heuristic, which is the notification performed when no move could be made. (This was previously discussed in Section 8.3.4.) `complete` returns true if the meta-heuristic would like to terminate the search (for instance, if it cannot reduce its filtering strength), or false if it wishes to continue. When `complete` is called on Solver's tabu search mechanism, the tabu status of the oldest element in the tabu list is revoked, potentially allowing a move to be made at the next step. A false value is always returned unless the tabu list was empty on entry to `complete`, in which case, true is returned.

```
39. for(i = 1; i <= 30; i++){
40.  if (solver.solve(move)) {
41.    cout  << "Cost = " << solver.getValue(cost)
             << " (Best = " << best.getValue(obj)
             << ") " << endl;
     }
42.   else if (mh.complete()) break;
   }
```

Finally, the best solution is presented for inspection.

```
43. cout << "Final Solution" << endl;
44. for (i = 0; i < nbCust; i++) {
45.   cout << best.getValue(x[i]) " ";
46. cout << endl ;
```

8.4.5 Hybrid Local/Tree-Based search

Unfortunately, each of the above methods has trouble finding high quality solutions, even when more effort is expended in tuning meta-heuristics or trying different possibilities for the search goals.

When one analyzes the local search procedure, one finds that it can be relatively difficult for a local search based on moves of customers from facility to facility to make bulk transfers of customers from one facility to another (or from two larger ones to three smaller ones, for example.) This is because the benefits of such a bulk transfers lie relatively far away in terms of number of moves, while there are great cost disadvantages on the way there, notably because of the fixed cost of the facilities. On the local search trajectory for such a transfer, more facilities are open than necessary, and this increases the cost. These reasons make the landscape difficult to negotiate for a search procedure which performs simple movements of customers.

There are standard techniques for diversifying the search (for instance, see a range of techniques in Glover and Laguna (1997)), such as the use of frequency memory, searching outside the feasible region, penalties and inducements, strategic oscillation, etc. However, the breadth of available techniques makes even such an investigation costly. We instead examine a different approach for improving the local search – we examine a hybrid method.

Using a constraint programming library, we can exploit the inherent structure in the problem. Although an assignment of facility to each customer is enough to determine a solution, we can look upon the problem as being two-tiered. There is first the problem of determining which facilities should be open, and then there is the secondary problem of determining which open facility should be assigned to which customer. If both of these problems are teased apart somewhat, a two-level search can be effected. The idea is to perform a "coarse" local search over a finer search. We perform a local search over which facilities are open using a move operator which opens or closes a facility. After each such move, the "finer details" are filled in – these being which facility each customer is served from. This could be performed by another local search process, but to illustrate a mixture here, we use a traditional tree search process to assign facilities. (One could also use a mixed integer programming technique to find the optimal assignment, although this may become less possible if the problem is enriched with side-constraints.) This tree search at each step results in large rearrangements of assignments of facilities to customers (bulk transfers) as facilities are opened and closed, largely circumventing the problems described earlier.

For the local search which moved customers from one facility to another we used a short-term tabu memory which disallowed moving a customer back to a facility it had come from for a certain number of iterations. In the hybrid method, we again use a short-term tabu memory, but this time we say that when a facility is opened or closed, it must remain in that state for a certain number of iterations.

We first begin by generating a first solution exactly as for the pure local search method: lines 26–31 in Section 8.4.4, and storing the first solution in the solution `best`. We next must create a new solution with a different structure. `best` contains the assignments of all facilities to customers and as such defines a complete solution. For the hybrid method, we perform local search over a partial solution: the "openness" of facilities, and so we need a solution object to represent this structure. The following code creates a solution containing the appropriate variables and the objective, and stores it.

```
IloSolution soln(env);
soln.add(obj);
soln.add(open);
soln.store(solver);
```

We then create a neighborhood which opens or closes a facility. For this, we use the neighborhood `IloFlip` which changes the value of a 0–1 variable. The use of `IloRandomize` ensures that the flips are explored in a random order, the reasons for this being the same as those described in Section 8.4.4.

```
IloNHood nh =
        IloRandomize(env, IloFlip(env, open), rnd);
```

As previously described, a short-term tabu memory is used. This is defined below. The parameter 3 ensures thatwhen a facility is opened or closed, it mustremain in that state for three iterations.

```
IloMetaHeuristic mh = IloTabuSearch(env, 3);
```

Finally, the goal to make a move is constructed. We choose the lowest cost non-tabu move using the `IloMinimizeVar` search selector which ensures that we open/close the facility which leads to a reassignment of facilities to customers at minimum cost. At each move we also store the new solution in`best` if it is the best seen so far.

```
IloSearchSelector selMin = IloMinimizeVar(env, cost);
IloGoal move =
        IloSingleMove(env, soln, nh, mh, selMin, assign)
     && IloStoreBestSolution(env, best);
```

Figure 8.4 shows graphically how ILOG Solver searches for the move (which facility to open or close) which minimizes the overall cost. The search-tree follows a two level structure. The top part corresponds to the local search action of opening or closing a facility, and works on the `open` variables, whereas the bottom part of the tree deals with assigning open facilities to customers. Although these are drawn differently in the figure, both are standard Solver tree-based searches which marry together in the standard way; the triangles merely represent larger search trees which cannot be depicted in detail. At the top level of the tree $|\mathcal{F}|$ open/close operations are explored – one for each facility. In this example we assume there are six facilities. Imagine that the current state of facilities is CCOOCO (C for closed and O for open), and that the previous three moves closed facility 2, and opened facilities 3 and 6 (counting from the left). It is thus tabu to open facility two, or close either of facilities 3 and 6 at this iteration. Such moves will immediately be rejected and will be dead-ends in the search tree. The$\otimes$ symbolswith TABU alongside indicate these illegal moves. Some "closure" moves may also not be possible if closing a facility would mean that the sum of demands of the customers exceeded the sums of the capacities of the open facilities. $\otimes$ symbols with CAP alongside indicate such failed closures for facilities 3 and 4 in our example. For all the remaining open/close moves, the additional `assign` goal will be

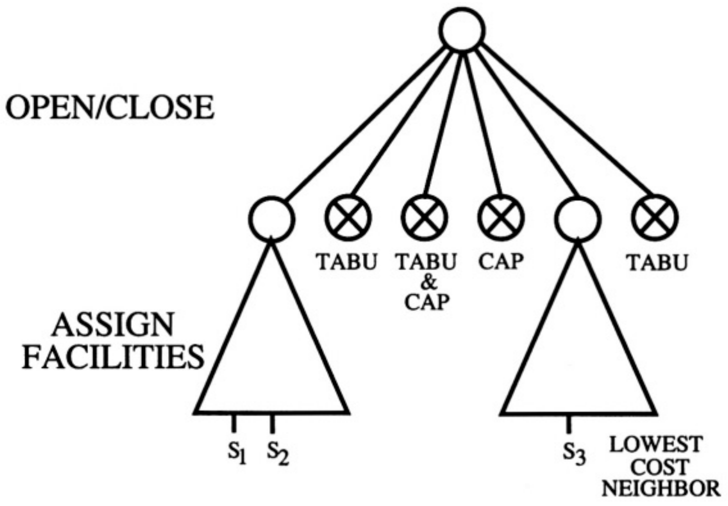

Figure 8.4 Finding the Lowest Cost Move

executed which assigns all customers to the open warehouses. The execution of these search trees is indicated by the triangles in the lower part of the figure. A number of complete assignments meeting all constraints may be found in each of these sub-trees. Each time one is found, the `IloMinimizeVar` search selector ensures that a constraint is added automatically by Solver stating that any solutions found in the entire tree after that point must have a cost which is strictly lower than the one just found. When the entire search tree has been traversed, `IloMinimizeVar` makes the search jump back to the best solution found. In this case it is S_3 indicating that the best legal non-tabu move to make is one which opens facility 5.

8.4.6 *Experiments Using the Three Methods*

We investigated the behavior of the three methods described in Sections 8.4.3, 8.4.4 and 8.4.5. We tested on all the capacitated facility location problems in the OR-Library (Beasley (1988), Beasley (1990)) of up to 50 customers and 50 facilities. The test code for each of the three methods corresponds almost exactly to the code presented. The main difference is the use of limited discrepancy search (LDS) (Harvey and Ginsberg (1995)) over depth-first search for the tree-based method and the hybrid method. LDS was chosen over depth-first search as the results are significantly better for the tree-based search. A small but non-negligible improvement in the hybrid search also results. ILOG Solver's search mechanism makes this improvement trivial to program. It is sufficient to add the line

```
assign = IloApply(env, assign, IloLDSEvaluator(env));
```

between the declaration of the `assign` goal and its use.

We set a time limit of two minutes on our experiments. For the pure and hybrid local search approaches, the initial solution is the first one generated by the complete

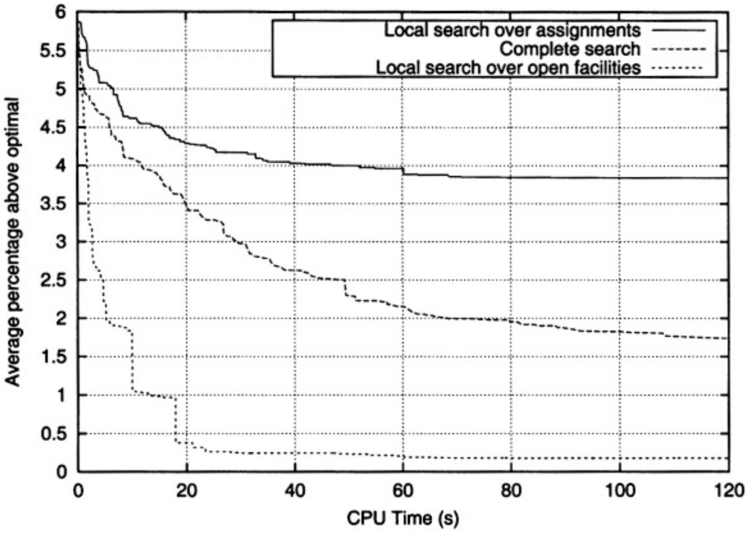

Figure 8.5 Comparison of Different Search Methods on Facility Location Problems

search technique, and so all methods begin at the same solution at the start of search. Figure 8.5 shows run time against the solution quality averaged over each of the location problems. (The average was formed as the geometric mean of ratios of cost to optimal cost. This average was then represented as a percentage by subtracting 1 and multiplying by 100.)

We immediately see the benefit of using the hybrid approach. It produces solutions around 0.2% above optimal after around one minute, whereas the other two methods do not attain the 2% mark in that time. It is clear that the tabu search over the assignment of facilities to customers quickly stagnates, despite our efforts to improve this state of affairs by using a varying tabu tenure. Complete search using limited discrepancy search works better, but its improvement tails off at near 1.5% from optimal. What we believe is important in this result is that the hybrid method required minimal tuning. For instance, it was still significantly better than the other two methods regardless of whether we used limited discrepancy search or depth-first search to perform the final assignments. We spent significantly more time attempting to improve the performance of the other two methods. Also significant is that such a hybrid search method would probably never have come about without the ability to simply integrate local and complete search using a constraint programming toolkit.

8.4.7 Note on Hybrid Approaches

The facility location problem as we have described here is also well suited to solution by mixed integer programming approaches. Such a method is easily implemented using ILOG tools (see, for example, van Hentenryck (1999)). We have concentrated on a constraint programming/local search approach here for simplicity. The reader should be aware however, that constraint programming, local search and linear programming can all be used to create a more complex and powerful hybrid.

8.5 EXTENDING THE TOOLKIT

ILOG Solver's local search mechanisms can be extended in two ways. Users are free to write their own neighborhoods or their own meta-heuristics. This considerably widens the scope of applicability of the local search mechanism as users are not limited to the neighborhoods or meta-heuristics supplied with the library. In this section, we demonstrate how new neighborhoods can be written; the mechanism for writing a new meta-heuristic is similar in style.

We have already seen that the neighborhood object is a class called `IloNHood`. In fact, `IloNHood` is a *handle class*. Handle classes are used throughout Solver and are classes which simply contain a pointer to a so-called *implementation class* which contains the object data. The use of handles means that pointers can generally be avoided, resulting in simpler code. The implementation class associated with `IloNHood` is `IloNHoodI` and it is with this second class that a new neighborhood is created: the relevant behavior is defined by sub-classing the `IloNHoodI` class.

In this section, we demonstrate how one could write the `IloFlip` neighborhood. For this example, the essentials of the `IloNHoodI` class are the following:

```
class IloNHoodI {
public:
  IloNHoodI(IloEnv env);
  virtual void start(IloSolver s, IloSolution currSoln);
  virtual IloInt getSize(IloSolver s) = 0;
  virtual IloSolution define(IloSolver s, IloInt idx) = 0;
};
```

The constructor of the class takes an environment. Any subclass of `IloNHoodI` must call this constructor, `start` is called before the neighborhood exploration begins. The current solution is passed as a parameter. For `IloFlip`, we simply need to keep the current solution. `getSize` is called after start and must deliver the number of neighbors in the neighborhood. For `IloFlip`, the number of neighbors is constant and equal to the number of variables, `define` is called numerous times with different indices in order to generate *solution deltas* representing the neighbors. For `IloFlip`, each delta will be a solution object containing one variable; the one which changes its value. In the above three methods, the active solver is passed as the first parameter.

The sub-class of `IloNHoodI`, `IloFlipI`, is shown below. The class has only two local variables (lines 2 and 3): the array `_vars` holds the array of variables to flip and `_currSoln` holds the current solution. The constructor (line 4) builds the subclass and initializes the array `_vars`. The `start` method (line 5) simply keeps the current solution in_currSoln. getSize (line 7) returns the size of the neighborhood, which is equal to the number of variables.

```
01. class IloFlipI : public IloNHoodI {
    private:
02.   IloNumVarArray _vars;
03.   IloSolution _currSoln;
    public:
04.  IloFlipI(IloEnv env, IloNumVarArray vars) :
         IloNHoodI(env), _vars(vars) { }
05.  void start(IloSolver, IloSolution currSoln) {
06.    _currSoln = currSoln;
     }
07.  IloInt getSize(IloSolver) { return _vars.getSize(); }
08.  IloSolution define(IloSolver, IloInt index);
    };
```

The define method is the only non-trivial method of IloFlipI. This method must return a solution delta with the variable being flipped set to its new value. Line 10 creates the delta, and line 11 adds a variable to it. For IloFlip, neighbor index i corresponds exactly to variable index i, and so the correct variable can be fetched using the neighbor index. Line 12 retrieves the value of this variable in the current solution, and line 13 sets the value in the delta to be "flipped". Finally, line 14 returns the solution delta.

```
09. IloSolution IloFlipI::define(IloSolver s, Ilolnt idx) {
10.   IloSolution delta(s.getEnv());
11.   delta.add(_vars[idx]);
12.   IloInt oldVal = _currSoln.getValue(_vars[idx]);
13.   delta.setValue(_vars[idx], 1 - oldVal);
14.   return delta;
    }
```

This concludes the definition of the neighborhood. Once created, user-defined neighborhoods behave exactly as Solver's built-in neighborhoods. For instance, they can be added, made the subject of randomization, etc. The library extensions become, for the user, part of the library itself.

8.6 SPECIALIZING THE TOOLKIT: ILOG DISPATCHER

Vehicle routing is one of the fields where local search is most widely used. This is mainly due to the fact that the problems encountered are usually large and difficult, and that most of the time, a "good solution" (in opposition with the best solution) is enough.

This section concentrates on how the Local Search Toolkit has been specialized to solve vehicle routing problems within the framework of ILOG Dispatcher. ILOG Dispatcher is a commercial product based on ILOG Solver. It is a C++ component library thatusers can tightly integrate with their own applications. It both extends Solver's modelinglayer, by providing various high-level classes related to vehicle routing (such as vehicles, visits, nodes, or dimensions), and Solver's search engine by offering first solution heuristics, specialized local search neighborhoods and meta-heuristics.

8.6.1 Basic Concepts

In the same way that we have variables and constraints in ILOG Solver, we have modeling classes in ILOG Dispatcher that help us model generic vehicle routing problems. Such problems suppose that there are a certain number of orders fromcustomers located at nodes which have to be performed by a fleet of vehicles. In Dispatcher, the action of performing an order, whether it is a delivery or pickup of goods or service (maintenance, for example), is called a visit. The basic variables of the problem are the ones describing the tour of each vehicle. For each visit, there are three variables, respectively representing:

- the visit immediately after the current visit, which we refer to as the next variable,
- the visit immediately before the current visit, which we refer to as the prev variable,
- the vehicle performing the current visit, which we refer to as the vehicle variable.

Here is an example of code which creates node, visit, and vehicle objects and which builds a route using constraints:

```
01. IloEnv env;
02. IloModel mdl(env);
03. IloNode node(env);
04. IloVisit visit(node);
05. mdl.add(visit);
06. IloNode depot(env);
07. IloVisit first(depot);
08. IloVisit last(depot);
09. IloVehicle vehicle(first, last);
10. mdl.add(vehicle);
11. mdl.add(visit.getPrevVar() == first);
12. mdl.add(visit.getNextVar() == last);
13. mdl.add(visit.getVehicleVar() == vehicle);
```

Lines 3 and 4 create a node and a visit`visit` located at that node. Line 4 adds the visit to the model. Line 6 creates a depot node, where a vehicle (line 9) starts and ends its tour (represented by visits at lines 7 and 8). Line 10 adds the vehicle to the model. Lines 11, 12 and 13 add constraints to the model specifying that the vehicle leaves the depot to perform `visit` and comes back to the depot. Note that lines 11 and 12 are equivalent to writing:

```
mdl.add(first.getNextVar() == visit);
mdl.add(last.getPrevVar() == visit);
```

This is due to the fact that next and prev variables are implicitly linked by the following constraint:

```
visit.getPrevVar().getNextVar() == visit;
```
and
```
visit.getNextVar().getPrevVar() == visit;
```

Line 13 is not mandatory because another constraint states that for two visits `v1` and`v2`:

```
if v1.getNextVar() == v2
then v1.getVehicleVar() == v2.getVehicleVar()
```

and that first and last visits of a vehicle have their vehicle variable bound to this vehicle.

The most innovative class in Dispatcher is probably dimensions. Dimensionsrepresent quantitieswhich are accumulated along routes. There are two types of dimensions: extrinsic and intrinsic. Extrinsic dimensions, instances of `IloDimension2`, represent quantities which depend on two visits to be evaluated. These dimensions usuallyrepresent the duration or the distance travelled on a route. Intrinsic dimensions, instances of `IloDimension1`, represent quantities which only depend on one visit. This can be quantities of good picked up or delivered at a visit expressed in weight, volume or number of pallets, for instance. Both classes are subclasses of `IloDimension` which share common features. For each visit there exist two dimension variables per dimension:

a) the cumul variable representing the amount of dimension accumulated when arriving at the visit; it can be accessed using:

```
IloVisit::getCumulVar(IloDimension),
```

b) the transit variable representing the amount of dimension added to the cumul variable whengoingfrom one visit to another; it can be accessed using:

```
IloVisit::getTransitVar(IloDimension).
```

These variables are linked together by the following constraint:

```
if v1.getNextVar() == v2
then v2.getCumulVar(dim)
  == v1.getCumulVar(dim) + v1.getTransitVar(dim)
```

Extrinsic dimensions add three extra dimension variables:

a) the delay variable representing the amount of dimension needed to perform a visit; it can represent service time, for example,

b) the travel variable representing the amount of dimension needed to travel from one visit to another; this usually represents distance,

c) the wait variable representing the slack on dimension between two visits that can be used to perform breaks.

These variables are constrained by the following:

```
v.getTransitVar(dim) == v.getDelayVar(dim)
                      + v.getTravelVar(dim)
                      + v.getWaitVar(dim)
```

and

```
if v1.getNextVar() == v2 and v1.getVehicleVar() == w
then v1.getTravelVar(dim) == dim.getDistance(v1, v2, w)
```

where `IloDimension2::getDistance()` returns the distance between visit `v1` and visit `v2` using vehicle `w` for dimension `dim`.

8.6.2 Specialized Neighborhoods and Meta-Heuristics

The recommended approach for solving routing problems using ILOG Dispatcher is the standard two phase approach: find a first solution and then improve it using local search. Therefore, Dispatcher extends Solver's local search classes by providing neighborhoods and meta-heuristic specific to vehicle routing.

8.6.2.1 Neighborhoods. The neighborhoods predefined in Dispatcher are the usual move operators used in vehicle routing. Here are the five most generic ones (see Figure 8.6):

a) 2-opt which inverts sub-parts of a tour.

b) Or-opt which moves sub-parts with a maximal length of three visits inside a tour.

c) Relocate which moves a visit from a tour to another.

d) Exchange which exchanges the positions of two visits from two different tours.

e) Cross which exchanges the end of a tour with the end of another tour.

In Dispatcher, Relocate and Exchange are generalized to moving pairs of visits linked by same-vehicle constraints. These constraints ensure that two visits are performed by the same vehicle. They are used to model pickup and delivery problems, for instance.

Dispatcher also provides more specific move operators which are used when performing some visits is not mandatory. Such cases can arise when there is more than one possible destination for a delivery but only one visit must be performed or when the problem is over-constrained and not all visits can be performed. The three move operators provided are:

a) MakePerformed which makes an unperformed visit performed by placing it after a performed visit.

b) MakeUnperformed which makes a performed visit unperformed.

c) SwapPerform which makes a performed visit unperformed and replaces it by an unperformed visit.

8.6.2.2 Meta-Heuristics. ILOG Dispatcher provides two specialized meta-heuristics: Guided local search and a specific tabu search.

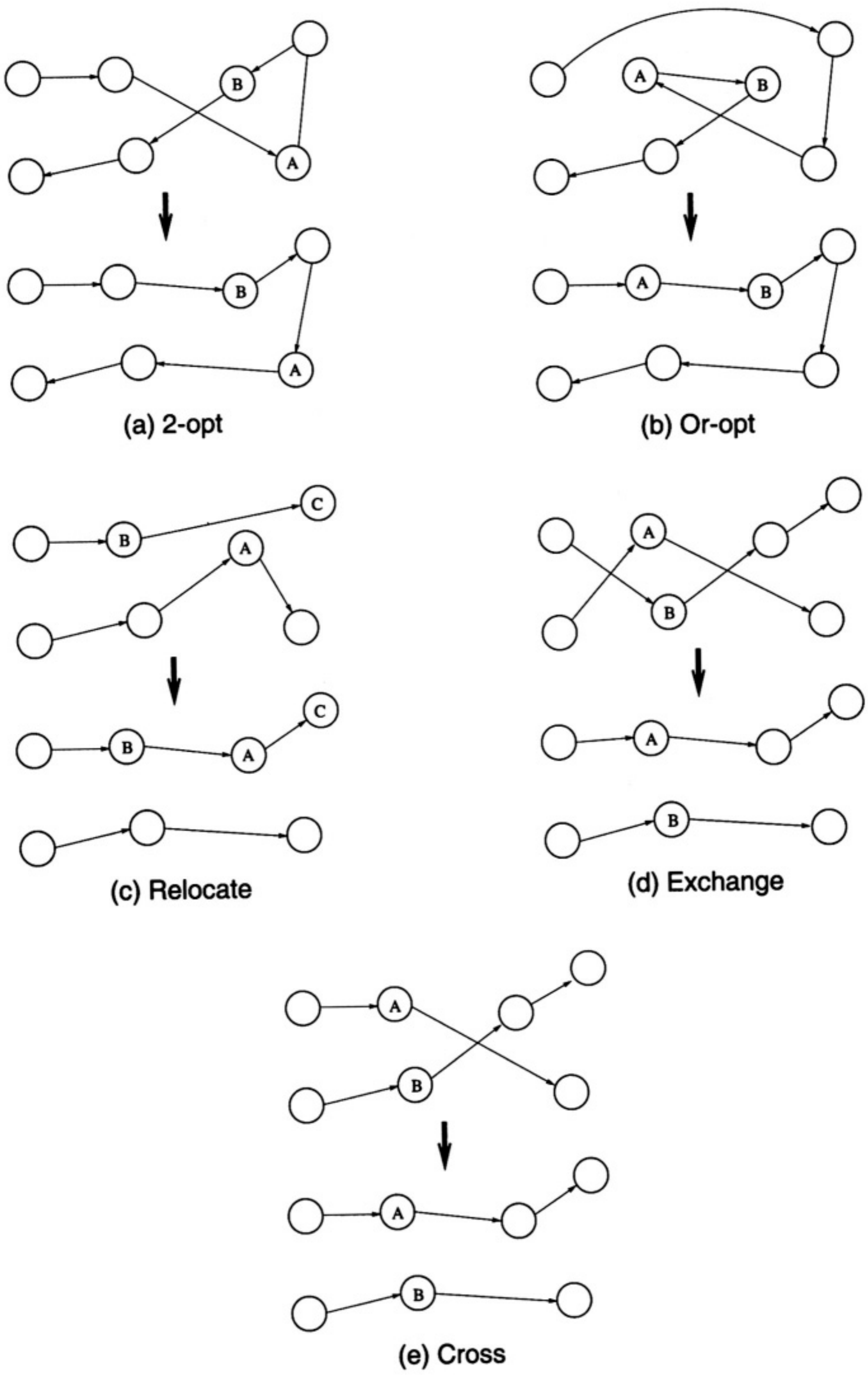

Figure 8.6 Move Operators

Guided local search has shown to work very well on vehicle routing problems. At each iteration of this search, a greedy search is performed to a local minimum taking into account a penalized cost function. The penalized cost function is created by adding a penalty term to the true cost function. This term is the sum of all penalties for possible arcs in the routing problem. Initially the penalty for each arc is equal to zero. Each time a local minimum is reached, the most costly and less penalized arc is penalized and a new greedy search is started.

The tabu search procedure used in Dispatcher is rather simple. Each time a move is performed, some arcs (pairs of connected visits) are added to the solution and some are removed. The added arcs are kept in a `keep` list and the removed ones in a `forbid` list. Arcs remain in a list for a certain number of moves known as their tenure. When a new move is considered, the number of arcs it adds which are in the `forbid` list is counted and as well as the number of arcs it removes which are in the `keep` list. If this number is above a certain tabu number, specific to each move operator, then the move is rejected: the move is tabu.

8.6.3 Example: Solving a Vehicle Routing Problem

The problem we address here is the popular capacitated vehicle routing problem with time windows. Given a set of deliveries of goods to customers which have to be performed within a time window, and a set of vehicles which can carry up to a certain quantity of goods (capacity), find the routes of minimum length. Each customer is positioned according to its coordinates and the distance between two customers is the Euclidean distance. This problem can be enriched with various types of side constraints, such as precedence constraints, same vehicle constraints, lateness and earliness costs, but for the sake of simplicity we treat only the problem described above.

8.6.3.1 Problem Description. We assume that all vehicles are identical and have a capacity of `capacity` and that their depot opens at `openTime` and closes at `closeTime` and is located at coordinates `(depotX, depotY)`. We also assume that the demands of the visits are held in an array `quantity`, that their coordinates are held in the arrays `x` and `y`, that time spent at each visit to unload (drop time) is held in the array `dropTime` and that their time windows are held in the arrays `minTime` and `maxTime`.

First an environment (line 1), a model (line 2) and the dimensions of the problem are created and added to the model (lines 3 to 8). There is one intrinsic dimension `weight`, which represents the weight carried by the vehicles, and two extrinsic dimensions, `time` and `length`, respectively, representing the time and the length of the routes. The extrinsic dimensions use the distance object `IloEuclidean` to compute distances.

```
01. IloEnv env;
02. IloModel mdl(env);
03. IloDimension1  weight(env,  "Weight");
04. mdl.add(weight);
05. IloDimension2  time(env,  IloEuclidean,  "Time");
06. mdl.add(time);
07. IloDimension2  length(env,  IloEuclidean,  "Length");
08. mdl.add(length);
```

Now we create the vehicles and add them to the model. Each vehicle starts and ends at visits located at a depot node `depot`. The cumul variables of these start and end visits are constrained to take into account the opening and closing hours of the depot (lines 12 and 14). The capacity of the vehicles is set to `capacity` (line 17) and the cost of the route is set to be equal to its length (line 16).

```
09. IloNode depot(env, depotX, depotY);
10. for (IloInt j = 0; j < nbOfTrucks + 10; j++) {
11.   IloVisit  start(depot,  "Depot");
12.   mdl.add(start.getCumulVar(time)  >=  openTime);
13.   IloVisit end(depot,  "Depot");
14.   mdl.add(end.getCumulVar(time)  <=  closeTime);
15.   IloVehicle  vehicle(start,  end);
16.   vehicle.setCost(length,  1.0);
17.   vehicle.setCapacity(weight,  capacity);
18.   mdl.add(vehicle);
19. }
```

After the vehicles, the visits are created and added to the model. Each of them is located at a node `customer` and the dimension variables are constrained to take into account the drop time (by constraining the delay variable for`time`, line 23), the quantity of goods the customer demands (by constraining the transit variable for `weight`, line 25) and its time window (by constraining the cumul variable for `time`, line 27).

```
20. for (IloInt i = 0; i < nbOfVisits; i++) {
21.   IloNode  customer(env,  x[i],  y[i]);
22.   IloVisit  visit(customer);
23.   mdl.add(visit.getDelayVar(time)  ==  dropTime[i]);
24.   IloNumVar  transit  =  visit.getTransitVar(weight);
25.   mdl.add(visit.transit == quantity[i]);
26.   IloNumVar cumul = visit.getCumulVar(time);
27.   mdl.add(minTime[i] <= cumul <= maxTime[i]);
28.   mdl.add(visit);
29. }
```

These lines finish stating the model and we can move on to solving the problem.

8.6.3.2 Finding a Solution. We present the standard two phase approach, first finding a solution using a first solution heuristic, then improving it with local search. We propose to decompose the local search phase into two sub-phases: a first phase of first accept greedy search, and a second phase based on tabu search.

The following lines show how to create a first solution using the savings heuristic based on Clarke and Wright (1964).

```
30. IloSolver solver(mdl);
31. IloDispatcher dispatcher(solver);
32. IloRoutingSolution solution(mdl);
33. IloNumVar cost = dispatcher.getCostVar();
34. IloGoal instCost =
35.         IloDichotomize(env, cost, IloFalse);
36. IloGoal goal = IloSavingsGenerate(env) && instCost;
37. if (!solver.solve(goal)) {
38.   cout<< "No solution" << endl;
39.   env.end();
40.   return 0;
41. }
42. solution.store(solver);
```

We create a solver, which extracts the model (line 30), and a dispatcher (line 31), which is an object whose purpose is to let the user access the state and values of Dispatcher objects during the search. A specific type of solution, containing the visits of the model, is created in line 32. This solution is nothing more than a standard Solver solution with a routing-oriented programming interface. In line 36 we create the Solver goal which uses the savings heuristic to find a solution. The goal created at lines 34 and 35 is used to instantiate the cost variable. The search starts in line 37 and if a solution is found it is stored (line 42). Now the local search phase can begin.

We start by a simple greedy search.

```
43. IloGoal improve = IloSingleMove(env,
44.                                 solution,
45.                                 IloTwoOpt(env)
46.                                 + IloOrOpt(env)
47.                                 + IloRelocate(env)
48.                                 + IloExchange(env)
49.                                 + IloCross(env),
50.                                 IloImprove(env),
51.                                 instCost);
52. while (solver.solve(improve)) {
53.   cout<< "Cost = "
54.        << dispatcher.getTotalCost() << endl;
55. }
```

The `IloSingleMove` goal is used once again, taking as parameter the Dispatcher move operators described in the preceding section (lines 45 to 49). The meta-heuristic `IloImprove` is used meaning a downhill greedy search is going to be performed.

`IloSolver::solve()` is called (line 52) and the current value of the objective is displayed (lines 53, 54) until no improvingmove can be found and that a local minimum is reached.

To climb out of the local minimum, we continue with a tabu search phase very similar to the one described in Section 8.4.4.

```
56. cout<< "Starting tabu" << endl;
57. IloRoutingSolution best = solution.makeClone(env);
58. IloDispatcherTabuSearch dts(env, 12);
59. IloSearchSelector sel = IloMinimizeVar(env, cost);
60. IloGoal tabuMove = IloSingleMove(env,
61.                                  solution,
62.                                  IloTwoOpt(env)
63.                                  + IloOrOpt(env)
64.                                  + IloRelocate(env)
65.                                  + IloExchange(env),
66.                                  dts,
67.                                  sel,
68.                                  instCost);
69. tabuMove = tabuMove
70.         && IloStoreBestSolution(env, best);
71. for (IloInt i = 0; i < 150; i++) {
72.   if (i == 70) dts.setTenure(20);
73.   if (i == 85) dts.setTenure(25);
74.   if (i == 105) dts.setTenure(12);
75.   if (solver.solve(tabuMove))
76.      cout<< "Cost = "
77.           << dispatcher.getTotalCost() << endl;
78.   else
79.      if (dts.complete()) break;
80. }
```

The current solution is cloned (line 57) in order to keep track of the best solution during the search (using the goal at line70). We use the Dispatcher-specific version of tabu search, described above, to move out of local mimima (line 58). The move operators considered are the same as during the greedy search phase, except for`IloCross` which can lead to too many symmetric neighbors (lines 62 to 65). Note that it is possible to modify the tenure during the search, as shown in lines 72 to 74. The resulting effect is an intensification or a diversification of the search, depending if the tenure is decreased or increased. The tabu search is run for 150 iterations (line 71) or until no non-tabu moves are left (line 79).

Finally, the best solution is restored and presented for inspection.

```
solver.solve(IloRestoreSolution(env, best)
                  && instCost);
cout<< "Final Solution" << endl;
cout<< "Cost = "
       << dispatcher.getTotalCost() << endl;
cout<< dispatcher << endl;
```

8.7 RELATED WORK

The work presented in this article can be related to several different areas.

Numerous constraint programming systems have been developed since the mid eighties, starting with CHIP (Dincbas et al. (1988)). These were based on the Prolog programming language and, therefore, implemented only depth-first search. The first implementations mixing local search and constraint programming can be traced back to the mid nineties, with works such as Pesant and Gendreau (1996), and the GreenTrip project (de Backer and Furnon (1999)) with an application to vehicle routing.

Our work was highly influenced by that of Pesant and Gendreau (1996) and later Pesant and Gendreau (1999). However, our approach differs from theirs in that they use *variables and constraints* to represent the neighborhood. *Interface constraints* then connect these new variables with the original model variables. In this way, propagation can proceed between the original model variables and the new "neighborhood" variables. These interface constraints depend on the current solution to the problem and are added anew to the constraint solver each time a local move is to be made. Thereafter, the neighborhood exploration proceeds by generating all possible assignments to the neighborhood variables using a standard tree search. The interface constraints interpret the assignments to the neighborhood variables and perform the required propagation into the model variables. One difference with our approach lies in the fact that some variables and constraints have to be added, thus increasing the memory consumption. The most notable difference, however, is the lower algorithmic complexity guarantee of our approach (Shaw et al. (2000)).

Caseau and Laburthe (1998) gives a description of Salsa, which is essentially a set of enhanced search structures that enable to mix local and complete search within a constraint programming framework. Our approach shares this feature by extending the notion of goals.

LOCALIZER (Michel and van Hentenryck (2000)) consists in the combination of a modeling language for combinatorial problems and a local search solver for these problems. It retains some nice features of constraint programming, for example the ability to describe a problem through constraints. The solution process is somewhat different, since propagation over the domains of variables is not used. When the value of a decision variable is changed, data structures known as invariants update the values of all dependent variables, including the objective. Such maintenance of only complete assignments makes it difficult to combine complete and local search methods.

Although not toolkits or frameworks as such, work has also been performed on solving constraint satisfaction problems using local search techniques. Probably the most well-known is the GSAT/WalkSat family of algorithms (Selman et al. (1994)) which is applied to propositional satisfiability problems. More recently, more complex strategies have been applied (McAllester et al. (1997)). Also notable is the seminal work of Minton et al. (1992) on min-conflicts methods. An interesting approach has also be advocated by Prestwich (2000) who performs local search not on the decision variables themselves, but on the decisions made by a constructive algorithm. The approach thus fits well into a constraint programming framework.

EasyLocal++ (Di Gaspero and Schaerf (2001)), HOTFRAME (Fink et al. (1999b)) and OpenTS (IBM Open Source Software (2001)) represent other attempts to provide reusable components for local search. EasyLocal++ and HOTFRAME are a set of C++ classes for implementing various meta-heuristics, while OpenTS is written in Java and provides the basic elements for implementing a tabu search. The main difference between these approaches and ours is that some coding is needed to implement the enforcement of the constraints representing the problem at hand, since there is no way to model the problem using constraints.

Finally, Mautor and Michelon (1997) present a combination of local and complete search which can be easily implemented using our approach.

8.8 CONCLUSION

We have described the local search facilities of the ILOG Solver constraint programming library. This local search toolkit is flexible and open, giving users the ability to experiment with different types of search strategies, neighborhoods, and mixes of complete and local search. The toolkit is also extensible, providing the basis for creating new neighborhoods and meta-heuristics. The toolkit was described both in terms of its primary goals, oriented towards usability, and in terms of its orthogonal object model; the latter description being made explicit by the inclusion of ILOG Solver code samples. Of particular interest is the way in which neighborhoods and meta-heuristics can be composed. We have also described the benefits that traditional constraint programming can bring to local search, and thus why it is advantageous to embed local search support in a constraint programming library.

A real-life example of facility location has been described with code demonstrating the ease by which complete, local, and hybrid search methods can be applied. Changing search methods results in only a small change to the ILOG Solver program, which means that alternative strategies can be investigated quickly. Importantly, this can be done without changing the model as the search and model are separate entities; an uncommon occurrence for local search algorithms. We have shown that a hybrid method – which uses local search to decide which facilities to open and tree search to assign facilities to customers – can be an effective technique.

The ease with which the toolkit can be extended was highlighted by an example which describes a neighborhood. Neighborhoods are defined explicitly by stating, for each neighbor, what parts of the solution change and how. This form means that very general and even unstructured neighborhoods can be created, for example, which use candidate lists and learning (Glover and Laguna (1997)). Such flexibility is not so readily available when neighborhoods are implicitly defined.

Finally, ILOG Dispatcher was described. Dispatcher is a *vertical* library built upon ILOG Solver and dedicated to solving vehicle routing problems. The verticalization theme is carried consistently through four different aspects of Dispatcher: modeling objects for routing, constraints for routing, tree-search algorithms and heuristics for routing, and local search neighborhoods and meta-heuristics for routing. Each one of these verticalizations is made possible by the openness of ILOG Solver in all of these aspects. The result is a product that is both cleanly designed and easy to use. The coded ILOG Dispatcher example clearly demonstrates that industrial problems can be modeled and solved in an intuitive manner.