CP Summer School: Hybrid algorithms, local search, and Eclipse

Mark Wallace

September 2005

Chapter 1

Introduction

1.1 Tribes

There are three research communities exploring combinatorial optimisation problems. Within each community there is strong debate and ideas are shared naturally. Between the communities, however, there is a lack of common background and limited cross-fertilisation.

We belong to one of those communities: the CP community.[1] The other two are Mathematical Programming (MP) and Stochastic Search and Metaheuristics (SSM). Currently SSM seems to be the largest of the three. It has become clear that such a separation hampers progress towards our common goal, and there should be one larger community - whose name is a point of contention - which should include us all.

Hybrid algorithms lie at the boundary between CP, MP and SSM. We will explore some of the techniques used in MP and SSM, and show how they can be used in conjunction with CP techniques to build better algorithms. We will not here be investigating the "frontiers of research" in these communities. However it is my belief that CP can contribute right now at these frontiers. Hybrid techniques are not peripheral to the research of any of these communities. They are the key to real progress in all three.

1.2 Overview

Firstly we explore the mapping of problems to algorithms, the requirement for problem decomposition, and the need for linking solvers and solver cooperation. Different ways of linking solvers will be discussed, and some of their benefits and applications.

Secondly we will investigate different kinds of search, coming from the different communities, and see how they can be used separately, and together.

[1] There are also, of course, many people in the CP community who are not exploring combinatorial optimisation problems.

The objective is to lower the barrier to exploiting hybrid techniques, encouraging research at the boundaries of CP, MP and SSM, and finally to help bring these communities together.

Chapter 2

Hybrid Constraint Solving

2.1 The Conceptual Model and the Design Model

To solve a problem we start with a problem-oriented *conceptual* model. The syntax of conceptual models is targeted to clarity, expressive power and ease of use for people interested in solving the problem.

The conceptual model is mapped down to a *design* model which is machine-oriented [Ger01]. The design model specifies the algorithm(s) which will be used to solve the problem at a level that can be interpreted by currently implemented programming platforms, like ECLiPSe.

Real problems are complex and, especially, they involve different kinds of constraints and variables. For example a "workforce scheduling" problem [AAH95] typically involves the following decision variables:

- For each task, one or more *employees* assigned to the task.
- For each task, a *start time*
- For each (group of) employee(s), a *route* that takes them from task to task.
- For each (group of) employee(s), *shift start, end, and break times*

This is in sharp contrast to typical CSP puzzles and benchmarks, such as graph colouring, where all the variables are of the same "kind" and sometimes even have the same initial domains.

The constraints and data in real problems are also diverse. The workforce scheduling problem includes:

- *Location and distance* constraints on and between tasks
- *Skills* data and constraints on and between employees and tasks
- *Time* constraints on tasks and employee shifts

Naturally there are many more constraints in real workforce scheduling problems on vehicles, road speeds, equipment, team composition and so on.

The consequence is that the algorithm(s) needed to solve real problems are typically *hybrid*. The skills constraints are best solved by a different sub-algorithm from the routing constraints, for example.

2.2 Mapping from the Conceptual to the Design Model

To map a problem description to an algorithm, it is necessary to decompose the whole problem into parts that can be efficiently solved. The challenge is to be able to glue the subproblem solutions together into a consistent solution to the whole problem. Moreover, for optimisation problems, it is not enough to find the optimal solution to each subproblem. Glueing these "local" optima together does not necessarily yield a "global" optimum.

For these reasons we need to ensure that the subproblem algorithms cooperate with each other so as to produce solutions that are both consistent with each other and, as near optimal as possible. The design of *hybrid algorithms* that meet these criteria is the topic of this section.

In principle we can map a conceptual model to a design model by

- Associating a behaviour, or a constraint solver, with each problem constraint
- Adding a search algorithm to make up for the incompleteness of the constraint solvers

In practice the design model produced by any such mapping is strongly influenced by the particular choice of conceptual model. The "wrong" conceptual model could make it very hard to produce an efficient algorithm to solve the problem.

For this reason we must map a given conceptual model to an efficient design model in two steps:

- Transform the conceptual model into another one that is more suitable for mapping
- Add constraint behaviour and search, to yield an efficient design model

The first step - transforming the conceptual model - is an art rather than a science. It involves four kinds of transformations:

- Decomposition - separating the constraints and variables into subproblems
- Transformation - rewriting the constraints and variables into a form more suitable for certain kinds of solving and search
- Tightening - the addition of new constraints whose behaviour will enhance efficient problem solving

- Linking - the addition of constraints and variables that will keep the separate subproblem solvers "in step" during problem solving

The decomposition is germane to our concerns. It is therefore worth discussing briefly here. Firstly, we note that the decomposition covers the original problem (of course), but it is *not* a partition: otherwise the subproblems would have no link whatsoever between them.

Therefore some subproblems share some variables. Each subproblem solver can then make changes to a shared variable, which can be used by the other solver. Sometimes constraints are shared by different subproblems. In this case the same constraint is handled multiple times by different solvers, possibly yielding different and complementary information within each solver. When the constraints in different subproblems are transformed in different ways, the result is that the same constraint may appear several times in several different forms in the transformed conceptual model. We shall see later a model with a resource constraint that is written three different ways for three different solvers.

Many of the "tricks of the trade" used in transforming the conceptual model have been covered in the earlier section on *Modelling*. We shall now move on to examine design models for a variety of hybrid algorithms.

Chapter 3

Constraint Solvers

In this section we discuss different constraint solvers, and constraint behaviours. We investigate what kinds of information can be passed between them, in different hybrid algorithms, and how their cooperative behaviour can be controlled.

The solvers we will cover are

- Finite domain (FD) constraint propagation and solving
- Global constraints and their behaviour
- Interval constraints, and bounds propagation
- Linear constraint solving
- Propositional clause (SAT) solving
- Set constraint solving
- One-way constraints (or "invariants")

Referring back to the three research communities, we can relate these solvers to the CP and MP communities. Accordingly this work lies on the border of CP and MP. The hybrids on the border of CP and SSM will be explored in the next section.

3.1 Constraints which Propagate Domain Information

3.1.1 Information Exported

Finite domains and global FD constraints have already been covered earlier. The relevant issues for hybrid algorithms are

- what information can be extracted from the solver

- under what conditions *all* the information has been extracted from the solver: i.e. when can we be sure that a state which appears to be feasible for the other subproblems and their solvers is also guaranteed to be consistent with the FD solver.

The answers are as follows:

- Information that can be extracted from the solver includes upper and lower bounds on the variables, domain size and if necessary precise information about which values are in the domain of each variable.
- The solver reports inconsistency whenever a domain becomes empty.
- The solver can also report which constraints are entailed by the current (visible) domain information. This is more usually extractible in the form of which constraints are still "active". Active constraints are ones which, to the FD solvers knowledge, are not yet entailed by the domains of the variables. Some FD solvers don't guarantee to detect this entailment until all the variables have been instantiated. Many FD solvers support *reified* constraints, which have a boolean variable to flag entailment or disentailment (inconsistency with the current variable domains).

The domain, inconsistency and entailment information are all logical consequences of the FD constraints and input variable domains. For this reason, no matter what other constraints in other subproblems are imposed on the variables, this information is still correct. In any solution to the whole problem, the values of the FD variables must belong to the propagated domains. Inconsistency of the subproblem, implies inconsistency of the whole problem. If the variable domains entail the subproblem constraints, then they are still entailed when the constraints from the rest of the problem are considered.

3.1.2 Global Constraints

Notice that global constraints are often themselves implemented by hybrid techniques, even though the information imported and exported is restricted to the above. A special case is the use of continuous variables in global constraints. The classic example is a global constraint for scheduling, where resource variables are FD, but the task start time variables could be continuous. As far as I know the hybrid discrete/continuous scheduling constraint is not yet available in any CP system.[1]

3.1.3 Interval Constraints

For interval constraint solvers only upper and lower bounds, and constraint entailment are accessible. The problem with continuous constraints is that they are not necessarily instantiated during search. Since continuous variables can

[1] CP scheduling will be covered in more detail later.

take infinitely many different values, search methods that try instantiating variables to all their different possible values don't necessarily terminate. Instead search methods for continuous values can only tighten the variable's bounds, until the remaining interval associated with the variable becomes "sufficiently" small.

Not all values within these small intervals are guaranteed to satisfy all the constraints. Indeed there are common cases where, actually, there are no feasible solutions, even though the final intervals appear prima facie compatible. One vivid example is Wilkinson's problem (quoted in [Van98]). It has two constraints: $Prod_{i=1}^{20}(X+i) + P \times X^{19} = 0$ and $X \in [-20.4..-9.4]$. When $P = 0$ the constraints have 11 solutions ($X = -10 \ldots X = -20$), but when P differs infinitesimally from 0 (viz. $P = 2^{-23}$), it has no solutions!

For these reasons "answers" returned by search routines which associate small intervals with continuous variables are typically conjoined with a set of undecided constraints, which must be satisfied in any solution.

3.2 Linear Constraints

3.2.1 Underlying Principles

A linear constraint solver can only handle a very restricted set of constraints. These are linear numeric constraints that can be expressed in the form $Expr \geq Number$ or $Expr \leq Number$. The expression on the left hand side is a sum of *linear terms*, which take the form *Coefficient* $\times$ *Variable*. The coefficients, and the number on the right hand side, are either integers or real numbers [Wil99].

Linear constraint solvers are designed not only to find feasible solutions, but also to optimise against a cost function in the form of another linear expression.

In the examples in this chapter we shall typically write the linear constraints in the form $Expr \geq Number$, and assume that the optimisation direction is *minimisation*.

Whilst much less expressive than CP constraints, they have a very important property: any set of linear constraints can be tested for *global* consistency in polynomial time. This means we can throw *all* the linear constraint of a problem into the linear solver and immediately determine whether they are consistent.

By adding just one more kind of constraint, an *integrality* constraint that requires a variable to take only integer values, we can now expressive any problem in the class *NP*. (Of course the consistency problem for mixed linear and integrality constraints - termed MIP - is NP-hard).

The primary information returned by the linear solver is consistency or inconsistency among the set of linear constraints. However for building cooperative solvers we will seek more than this.

Firstly the solver can also export an optimal solution. In general there may be many optimal solutions, but even from a single optimum we now have a known optimal value for the cost function. No matter how many other constraints there may be in other solvers, the optimal value cannot improve when

they are considered, it can only get worse. Thus the linear solver returns a bound on the cost function.

Linear constraints are special because if S_1 and S_2 are two solutions (two complete assignment that satisfy all the linear constraints), then any assignment that lies on the line between S_1 and S_2 is also feasible. For example if $X = 1$, $Y = 1$ is a solution, and so is $X = 4$ and $Y = 7$, then we can be sure that $X = 2$ and $Y = 3$ is a solution, and so is $X = 3$ and $Y = 5$. Moreover since the cost function is linear, the cost of any solution on the line between $S1$ and $S2$ has a cost between the cost of $S1$ and the cost of $S2$.

These properties have some important consequences. Supposing $Expr \geq Number$ is a constraint, and that at the optimal solution the value of $Expr$ is strictly greater than $Number$. Then the problem has the same optimal value even if this constraint is dropped (or "relaxed"). Otherwise you could draw a line between a new optimal solution and the old one, on which all points are feasible for the relaxed problem. Moreover the cost must decrease continuously towards the new optimum solution. Therefore at the point where this line crosses the line $Expr = Number$ (i.e. at the first point where the solution is also feasible for the original problem) the cost is less than at the old optimal solution, contradicting the optimality of the original solution.

In short, for a linear problem all the constraints which influence the optimal cost are *binding* at an optimal solution, in the sense that the expression on the left hand side is equal to the number on the right.

3.2.2 Shadow Prices and Dual Costs

If such a binding constraint was dropped, then the relaxed problem typically would have a new optimum value (with a solution that would have violated the constraint). Along the line between the old and new optimum, the cost function steadily improves. Instead of relaxing the constraint we can change the number on its right hand side so as to partially relax the constraint. As long as the constraint is still binding, at the new optimum the expression on the left hand side is equal to the new number on the right hand side. We can measure how much the optimum value of the cost function improves as we change the number on the right hand side of the constraint. This ratio is called the "shadow price" of the constraint.[2]

Indeed using the shadow price we can relax the constraint altogether, and effectively duplicate it in the optimisation function. If λ is the shadow price then dropping $Expr \geq Number$ and adding $\lambda \times (Number - Expr)$ to the cost function, we have a new problem with the same optimum solution as the old one.

There is another very interesting way to approach the very same thing. If we have a set of linear constraints of the form $Expr \geq Number$, we multiply each

[2]Technically the shadow price only takes into account those constraints which were binding at the current optimal solution. If at the new optimal solution for the relaxed problem another constraint became binding, then the shadow price would be an overestimate of the improvement in the optimum value.

constraint by a positive number and we add all the constraints together, i.e. we add all the multiplied left hand sides to create a new expression $SumExpr$ and we add all the multiplied right hand sides to get a new number $SumNumber$, then $SumExpr \geq SumNumber$ is again a linear constraint. Surprisingly everything you can infer from a set of linear constraints you can get by simply performing this one manipulation: forming a *linear combination* of the constraints.

In particular if you multiply all the constraints by their shadow prices at the optimum, and add them together the final right hand side is the optimum cost, and the left hand side is an expression whose value is guaranteed to be smaller than the expression defining the cost function. Indeed of all the linear combinations of the constraints whose left hand side is dominated by the cost function, the one formed using the shadow prices has the highest value on the right hand side.

We call the multipliers that we use in forming linear combinations of the constraints *dual values*. Fixing the dual values to maximize the right-hand-side expression *SumNumber* is a way of finding the optimal cost. At the optimal solution, the dual values are the shadow prices.

3.2.3 Simplex and Reduced Costs

In the section on Underlying Principles, above, we showed that at an optimal solution the set of tight constraints are the only ones that matter. In general if the problem has n decision variables, then it must have at least n linearly independent (i.e. different) constraints otherwise the cost is unbounded. Indeed there must be an optimal solution with n (linearly independent) tight constraints. Moreover n such constraints define a unique solution.

Consequently one way of finding an optimal solution is simply to keep choosing sets of n linearly independent constraints, making them tight (i.e. changing the inequality to an equality), and computing their solution until there are no more such sets. Then just record the best solution. In fact this can be made more efficient by always modifying the current set into a new set that yields a better solution.

We can rewrite an inequality constraint as an equation with an extra (positive) variable, called the "slack" variable. The n slack variables associated with the n tight constraints are set to zero. If there are m inequality constraints, the total number of variables (decision variables and slack variables) is $m + n$. The other m variables are constrained by m equations, and we say they are *basic* variables.

An optimal solution can be found by a hill climbing algorithm (the "Simplex" algorithm) which at each steps swaps one variable out of the basis and another in. The variables swapped into the basis is computed using a function called the *reduced cost*. For each current basis, a reduced cost can be computed for each non-basic variable, and any variable with a negative reduced cost has the potential to improve the solution. Another variable is chosen to be swapped out, and if the swap indeed yields a better solution, then the move is accepted.

When no non-basic variable has a negative reduced cost the current solution is optimal.

3.2.4 Information Exported from the Linear Constraint Solver

The information that can be extracted from a linear constraint solver is as follows:

- An optimal bound on the cost variable, a shadow price for each constraint, and a reduced cost for each problem variable. Additionally one solution with optimal cost can be extracted from the solver.
- The solver reports inconsistency whenever the linear constraints are inconsistent with each other

Upper and lower bounds for a variable X can be elicited from a linear solver by using X and $-X$ in turn as the cost function. However these bounds are computationally expensive to compute, and even for linear constraints, the FD solver typically computes bounds much more cheaply.

Unlike the FD solver, there is no problem of consistency of the constraints inside the solver, but the problem remains how to ensure these are consistent with the constraints in other solvers.

3.3 Propositional Clause Solvers and Set Solvers

Proposition clause solvers are usually called SAT solvers. SAT solvers are typically designed to find as quickly as possible an instantiation of the propositional (i.e. boolean) variables that satisfies all the clauses (i.e. constraints). Many SAT solvers, such as zChaff [ZMMM01], generate nogoods, expressed as clauses.

The challenge, for hybridising a SAT solver with other solvers is to interrupt the SAT solving process - which in general includes an exponential-time search procedure - before it has necessarily found a solution, to extract useful information from it, and to return it to the same state as when the SAT solver was invoked.

Information that can be extracted from a SAT solver is as follows:

- Nogood clauses
- Feasible solutions (in case the solver terminates with success before being interrupted)
- Inconsistency (in case the solver terminates with failure before being interrupted)

Set solvers typically only handle finite sets of integers (or atoms). Their behaviour is similar to FD solvers, except they propagate upper and lower bounds (i.e. the set of element that *must* belong to the set, and the set of elements that

could still belong to the set. The set cardinality is typically handled as an FD variable.

Information that can be extracted from a set solver is as follows:

- Upper and lower bounds
- Set cardinality
- The solver reports inconsistency whenever the upper bound ceases to be a superset of the lower bound

All the variables in a SAT solver are booleans. Set solvers also can be transformed into a representation in terms of boolean variables (associate a boolean variable with each value in the upper bound, setting those already in the lower bound to one, and setting the cardinality to the sum of the booleans).

3.4 One-way Constraints, or Invariants

Historically constraint programming is related to theorem proving, and constraint programs are often thought of as performing very fast logical inferences. Thus, for example, from the constraints $X = 2 + Y$ and $Y = 3$ the constraint program infers $X = 5$, which is a logical consequence.

Nevertheless there is information exported from constraint solvers which is very useful for guiding search but has no logical interpretation. One-way solvers can propagate and update such *heuristic* information very efficiently.

For many problems it is often useful to export a *tentative* value for a variable. This has no logical status, but is also very useful for guiding heuristics. This may be the value of the variable in a solution to a similar problem, or it may be the value of the variable in the optimal solution of a relaxed version of the problem. In a decomposition where the variable belongs to more than new subproblem, the tentative value may be its value in a solution of one subproblem.[3]

Now if variable Y has tentative value 3, and the constraint $X = 2 + Y$ is being handled by a one-way solver, then the tentative value of X will be set by the solver to 5. If the tentative value of Y is then changed, for some reason, to 8, then the tentative value of X will be updated to 10.

A one-way solver, then, computes the value of an expression, or function, and uses the result to update the (tentative) value of a variable. A one-way solver can also handle constraints by reifying them, and then updating the value of the boolean associated with the constraint. This enables the programmer to quickly detect which constraints are violated by a tentative assignment of values to variables.

If a variable becomes instantiated, the one-way solver treats this as its (new) tentative value, and propagates this to the tentative values of other variables as usual.

Information that can be extracted from a one-way solver is as follows:

[3]Indeed a variable may have several tentative values, but we shall only consider a single tentative value here.

- Tentative values for variables.
- Constraints violated by the tentative assignment

Chapter 4

Communicating Solvers

Constraint solvers can cooperate by sharing *lemmas* about the problem, or by sharing heuristic information. A lemma, from the CP viewpoint, is just a redundant constraint that can be used by the other solver to help focus and prune search. These lemmas can take many different forms: they may be *nogoods*, *cutting planes*, generated *rows*, *fixed variables*, tightened *domains*, propagated *constraints*, *cost bounds*, and even *reduced costs* [FLM04].

Some lemmas are logical consequences of the problem definition: these include global cuts and cost bounds. Other lemmas are valid only on the assumption that certain extra constraints have been imposed (during search). Validity in this case means that the problem definition conjoined with the specified extra constraints entail the lemma. A typical example is the information propagated during search by a bounds consistency algorithm. Naturally this information is only true for those parts of the search space lying below the search tree node where the information was inferred.

Heuristic information can also be global or local. Global heuristic information includes the number of constraints on a variable, an optimal solution to the linear relaxation problem at the "root" node of the search tree. Local heuristic information might include the size of the variables domains at a certain point during search, or the shadow prices of the linear constraints at the current node in the branch and bound search tree.

The key difference between lemmas and heuristic information lies in the consequences of errors.

If the heuristic information is out of date, or wrong, then the algorithm performance may be affected but the solution produced by the algorithm remains correct. However if lemmas are wrong, then the algorithm will generally yield some wrong answers.

4.1 Channeling Constraints

The same problem often needs to be modelled using different variables and constraints for efficient handling in different solvers. The n-queens problem for example can be modelled by n variables each with domain $1..n$, where each variable represents a queen in a different row, and each value represents the column in which that queen is placed. This is ideal for FD solvers, and disequations. It can also be modelling by n^2 zero-one variables, one for each square on the board. This is best for integer linear constraints where a constraint that two queens cannot share a line is encoded by constraining the sum of all the boolean variables along the line to be less than or equal to one.

All CSP problems can be transformed to SAT problems, using the same mapping of FD variables to booleans. Some researchers hope to achieve such high performance for SAT solvers that this transformation will prove the best way to solve all CSP problems [AdVD$^+$04]

SAT can perform very competitively on some typical puzzles and simple benchmarks. However the programmer has a great deal of control over FD search which is both a disadvantage (because SAT search is automatic) and an advantage because the programmer can tune the search to the problem at hand.

CSP problems can also be transformed to integer/linear problems, and some mathematical programmers believe that all such problems can be formulated (as a design model) in such a way that the integer/linear solver offers the fastest solution method. Whilst there is some truth in this claim, there are many problems and puzzles for which CP outperforms the best integer/linear models designed to date.

The point here is that all the different kinds of solvers - FD, interval, integer/linear, SAT, set - are sometimes more suitable and sometimes not so suitable as other solvers, *and to date we have discovered no way of inferring from the conceptual model which will be the best.*

Indeed it is our experience that, although for a specific problem instance one solver will be fastest, for most problem classes different solvers do better on different problem instances. Moreover in solving a single problem instances, there are different stages of the problem solving process when different solvers make the fastest progress.

Consequently the most robust algorithm, achieving overall good performance with the best worst-case performance, is a combination of all the solvers, where constraints are posted to *all* the solvers which can handle them. Good performance is achieved because the solvers communicate information with each other. To make this communication possible we require channeling constraints that enable information exported by one solver, expressed in terms of the variables handled by the solver, to be translated into information expressed in terms of a different set of variables that occur in another solver [CLW96].

The behaviour of a channeling constraint between an FD variable and its related boolean variables is not very mysterious. If variable V has domain $1..n$, and there are n boolean variables, B_1 which represents $V = 1$, B_2 which represents $V = 2$ etc., then we have the following propagation behaviour:

- If B_i is instantiated to 1 then propagate $V = i$
- If B_i is instantiated to 0 then propagate $V \neq i$
- If V is instantiated to i then propagate $B_i = 1$
- If the value i is removed from the domain of V, then propagate $B_i = 0$

To complete the channeling constraints we add the constraint $\sum_i B_i = 1$, which reflects that V takes one, and only one, value from its domain.[1]

Channeling constraints support communication between FD, interval, linear, SAT and set solvers. Note that they can also be useful when two design models of the same problem are mapped to the same solver. As a simple example the same n-queens problem can be modelled with a queen for every row, and a queen for every column. Let QR_i denote the position (i.e. column) of the queen in row i, and let QC_m denote the position (i.e. row) of the queen in column m. The channeling constraints are as follows:

- If QR_i takes the value m, then propagate $QC_m = i$.
- If QC_j takes the value n, then propagate $QR_n = j$
- If m is removed from the domain of QR_i, then propagate $QC_m \neq i$
- If n is removed from the domain of QC_j, then propagate $QR_n \neq j$

For standard models and FD implementations, the combined model, with channeling constraints, has better search behaviour than either of the original or dual model.

4.2 Propagation and Local Cuts

Let us now assume that we have one search routine which posts constraints on the variables at each node of the search tree. These constraints are then communicated to all the solvers through channeling constraints, and each solver then derives some information which it exports to the other solvers. The other solvers then use this information to derive further information which is in turn exported, and this continues until no more new information is produced by any of the solvers.

In this section we shall discuss what information is communicated between the solvers, and how it is used.

4.2.1 Information Communicated to the Linear Solver

When the domain of an FD variable changes, certain boolean variables become instantiated (either to 1 or to 0), and this values can be passed in to the linear solver.

[1] For mathematical programmers, the B_i comprise an SOS set of type one.

The importance of this communication is worth illustrating with a simple example. Consider the constraints $X \in 1..2$, $Y \in 1..2$ and $Z \in 1..2$, and *alldifferent([X,Y,Z])*. In the linear solver this information is represented using six boolean variables $X_1, X_2, Y_1, Y_2, Z_1, Z_2$ and five constraints $\sum_i X_i = 1$, $\sum_i Y_i = 1$, $\sum_i Z_i = 1$, $X_1 + Y_1 + Z_1 \leq 1$, $X_2 + Y_2 + Z_2 \leq 1$. This immediately fails in both the FD and the linear solvers. Suppose now we change the problem, and admit $Z = 3$. The linear solver determines that there is a solution (e.g. $X = 1, Y = 2, Z = 3$), but infers nothing about the values of X, Y or Z.[2] The FD solver immediately propagates the information that $Z = 3$, via the channeling constraint which adds $Z_3 = 1$ to the linear solver.

Sometimes FD variables are represented in the linear solver as continuous variables with the same bounds. In this case only bound updates on the FD variables are passed to the linear solver. An important case of this is branch and bound search, when one of the solvers has discovered a feasible solution with a certain cost. Whilst linear solvers can often find good cost lower bounds - i.e. optimal solutions to relaxed problems that are at least as good as any feasible solution - they often have trouble finding cost *upper* bounds - i.e. feasible but not necessarily optimal solutions to the real problem. The FD solution exports a cost upper bound to the linear solver. This can be used later to prune search when, after making some new choices, the linear cost lower bound becomes greater than this upper bound. This is the explanation why hybrid algorithms are so effective on the Progressive Party Problem [SBHW95, RWH99]. CP quite quickly finds a solution with cost 13, but cannot prove its optimality. Integer/linear programming easily determines the lower bound is 13, but cannot find a feasible solution. [3]

Not all constraints can immediately be posted to an individual solver. A class of constraints identified by Hooker [HO99] are constraint with the form $FD(\overline{X}) \rightarrow LP(\overline{Y})$. In this simplified form the $\overline{X}$ are discrete variables, and FD is a complex non-linear constraint on these variables. Finding an assignment of the variables that violates this constraint is assumed to be a hard problem for which search and FD propagation may be suitable. Once the variables in $\overline{X}$ have been instantiated in a way that satisfies FD, however, the linear constraint $LP(\overline{Y})$ is posted to the linear solver. If at any point the linear constraints become inconsistent, then the FD search fails the current node and alternative values are tried for the FD variables. Assuming the class of FD constraints is closed under negation, we can write -$FD(\overline{X})$ for the negation of the constraint $FD(\overline{X})$. This syntax can be used both to express standard FD constraints, by writing -$FD(\overline{X}) \rightarrow 1 < 0$, (i.e. -$FD(\overline{X}) \rightarrow false$), and standard LP constraints by writing $true \rightarrow LP(\overline{Y})$.

Constraints involving non-linear terms (in which two variables are multiplied together), can be initially posted to the interval solver, but as soon as enough variables have become instantiated to make all the terms linear the constraint can be sent to the linear solver as well. In effect the linear constraint

[2] The linear solver can be persuaded to infer more information only by trying to maximise and minimise each boolean in turn, as mentioned earlier.

[3] Though Kalvelagen subsequently modelled this problem successfully for a MIP solver.

is information exported from the interval solver and sent to the linear solver. Naturally this kind of example can be handled more cleverly by creating a version of the original nonlinear constraint where the non-linear terms are replaced by new variables, and posting it to the linear solver. When the constraint becomes linear, this should still be added to the linear solver as it is logically, and mathematically, stronger than the version with the new variables.

For the workforce scheduling problem we can use several hybrid design models linking finite domain and integer/linear constraints. We can, for example, use the FD solver to choose which employees perform which tasks.

We can link these decisions to the linear solver using Hooker's framework, by writing "*If* task i and task j are assigned to the same employee *then* task i must precede task j". We also need a time period of t_{ij} to travel from the location of task i to that of task j. Thus if task i precedes task j we can write $S_j \geq S_i + t_{ij}$, where S_i is the start time of task i, and S_j that of task j.

Now the linking constraint $FD(\overline{X}) \rightarrow LP(\overline{Y})$ has

- $FD(\overline{X}) = assign(task_i, Emp) \wedge assign(task_j, Emp)$, and
- $LP(\overline{Y}) = S_j \geq S_i + t_{ij}$

More realistically for the workforce scheduling problem we want to put a disjunction constraint, "task i must precede task j or task j must precede task i", on the right hand side of the implication. To encode the disjunction in the integer linear solver we can use a boolean variable B and a big number, say M. The whole disjunction is encoded as the following two constraints:

- $S_j + B \times M \geq S_i + t_{ij}$
- $S_i + (1 - B) \times M \geq S_j + t_{ij}$

If $B = 1$ the first constraint is always true, whatever values are assigned to S_i and S_j, so only the second constraint is enforced, that task j precedes task i. If $B = 0$ the second constraint is relaxed, and we have task i preceding task j.

Now the linking constraint $FD(\overline{X}) \rightarrow LP(\overline{Y})$ has

- $FD(\overline{X}) = assign(task_i, Emp) \wedge assign(task_j, Emp)$, and
- $LP(\overline{Y}) = S_j + B \times M \geq S_i + t_{ij} \wedge S_i + (1 - B) \times M \geq S_j + t_{ij}$.

Note that the integrality of the boolean variable B is enforced separately, assuming that eventually B will be instantiated to either 1 or 0 during search.

4.2.2 Information Exported from the Linear Solver

Cost Lower Bounds

For minimisation problems, we have seen that cost upper bounds passed to the linear solver from the FD solver can be very useful. Cost lower bounds returned from the linear solver to the FD solver are equally important.

Hoist scheduling is a cyclic scheduling problem where the aim is to minimise the time for one complete cycle [RW98]. FD can efficiently find feasible schedules of a given cycle, but requires considerable time to determine that if the cycle time is too short, there is no feasible schedule. Linear programming, on the other hand, efficiently returns a good lower bound on the cycle time, so by running the LP solver first an optimal solution can be found quite efficiently by the FD solver. Tighter hybridisation yields even better algorithm efficiency and scalability [RYS02].

Returning to the workforce scheduling example, another approach is to use the full power of integer/linear programming to solve TSPTW (Traveling Salesman Problem with Time Windows) subproblems and return the shortest route covering a set of tasks. In this case we have a linking constraint for each employee, emp_n. Whenever a new task is added to the current set assigned to emp_n the implication constraint sends the whole set to the integer/linear solver which returns an optimal cost (shortest route) for covering all the tasks.

This is an example of a global optimisation constraint (see [FLM04]). Global optimisation constraints are ones for which special solving techniques are available, and from which not only cost lower bounds can be extracted, but also other information.

Reduced Costs

One very useful type of information is reduced costs. As we have seen, a single FD variable can be represented in the linear solver as a set of boolean variables. When computing an optimal solution, the LP solver also computes reduced costs for all those booleans which are 0 at an optimum. The reduced costs provide an estimate of how much worse the cost would become if the boolean was set to 1. This estimate is conservative, in that it may underestimate the impact on cost.

If there is already a cost upper bound, and the reduced cost shows that setting a certain boolean to 1 would push the cost over its upper bound, then we can conclude that this boolean must be 0. Via the channeling constraints this removes the associated value from the domain of the associated FD variable. Reduced costs therefore enable us to extract FD domain reductions from the linear solver.

Reduced costs can also be used for exporting heuristic information. A useful variable choice heuristic termed *max regret* is to select the variable with the greatest difference between its "preferred" value, and all the other values in its domain. This difference is measured in terms of its estimated impact on the cost, which we can take as the minimum reduced cost of all the other booleans representing values in the domain of this variable.

Relaxed Solution

The most valuable heuristic information exported from the linear solver is the relaxed solution which it uses to compute the cost lower bound. This assignment of values to the variables is either feasible for the whole problem - in which case

it is an optimal solution to the whole problem - or it violates some constraints. This information can then focus the search on "fixing" the violated constraints. Most simply this can be achieved by instantiating one of the variables in the violated constraints to another value (perhaps the one with the smallest reduced cost). However a more general approach is to add new constraints that prevent the violation occurring again without removing any feasible solutions.

Fixing Violations by adding cuts To fix the violation we seek a logical combination of linear constraints which exclude the current infeasible assignment, but still admits all the assignments which are feasible for this constraint.

If this is a conjunction of constraints, then we have a *global cut* which can be added to the design model for the problem. An example of this is a subtour elimination constraint, which rules out assignments that are infeasible for the travelling salesman problem.

If it is a disjunction, then different alternative linear constraints can be posted on different branches of a search tree. When, for example, an integer variable is assigned a non-integer value, say 1.5, by the linear solver, then on one branch we post the new bound $X \leq 1$ and on the other branch we post $X \geq 2$.

The challenge is to design these constraints in such a way that the alternation of linear constraint solving and fixing violations is guaranteed, eventually, to terminate.

Fixing violations by imposing penalties There is another quite different way to handle non-linear constraints within a linear constraint store. Instead of posting a new constraint, modify the cost function so that the next optimal solution of the relaxed problem is more likely to satisfy the constraint. For this purpose we need a way of penalising assignments that violate the constraint, in such a way the penalty reduces as the constraint becomes closer to being satisfied, and becomes 0 (or negative) when it is satisfied.

With each new relaxed solution the penalty is modified again, until solutions are produced where the penalty function makes little, or no, positive or negative contribution, but the constraint is satisfied. In case the original constraint was linear, we can guarantee that the optimum cost for the best penalty function is indeed the optimum for the original problem. This approach of solving the original problem by relaxing some constraints and adding penalties to the cost function is called "Lagrangian relaxation".

4.2.3 Information Imported and Exported from the One-Way Solver

The one-way solver is an important tool in our problem solving workbench. The solver is used to detect constraint conflicts, and thus help focus search on hard subproblems. A simple example of this is for scheduling problems where the objective is to complete the schedule as early as possible. FD propagation

is used to tighten the bounds on task start times. After propagation, each start time variable is assigned a tentative value, which is the smallest value in its domain. The one-way solver then flags all violated constraints, and this information is used to guide the next search choice.

For many scheduling problems tree search proceeds by posting at each node an ordering constraint on a pair of tasks: i.e. that the start time of the second task is greater than the end time of the first task. We can reify this ordering constraint with a boolean that is set to 1 if and only if the ordering constraint holds between the tasks. We can use these booleans to infer the number of resources required, based on the number of tasks running at the same time.

Interestingly this constraint can be handled be handled by three solvers: the one-way solver, the FD solver and the linear solver. The linear solver relaxes the integrality of the boolean, and simply finds optimum start times. The FD solver propagates the current ordering constraints, setting booleans between other pairs of tasks, and ultimately failing if there are insufficient resources. The one-way solver propagates tentative start time values exported from the linear solver to the booleans. This information reveals resource bottlenecks, so at the next search node an ordering constraint can be imposed on two bottleneck tasks. The ordering constraint is imposed simply by setting a boolean, which constrains the FD, linear and one-way solvers. This approach was used in [EW00].

The one way solver propagates not only tentative values, but also other heuristic, or meta-information. It allows this information to be updated efficiently by updating summary information rather than recomputing from scratch. This efficiency is the key contribution of *invariants* in the Localizer system [VM00].

For example if Sum is the sum of a set of variables V_i, then whenever the tentative value of a variable V_k is updated, from say m to n, the one way solver can efficiently update Sum by changing its tentative value by an amount $n-m$.

Earlier we discussed reduced costs. The "max regret" heuristic can be supported by using the one-way solver to propagate the reduced costs for the booleans associated with all the different values in the domain of a variable. $MaxRegret$ for the variable is efficiently maintained as the difference between the lowest and the second lowest reduced cost.

4.3 Subproblems Handled with Independent Search Routines

4.3.1 Global Cuts and Nogoods

A loose form of hybrid algorithm is to solve two subproblems with their own separate search routines. Each solution to one subproblem is tested against the other by initialising the common variables with the solution to the first subproblem, and then trying to solve the constrained version of the second one. Each time the second subproblem solver fails to find a solution, this is reported back to the first subproblem solver as a nogood. The idea is that future solutions

to the first subproblem will never instantiate the common variables in the same way.

This simple idea can be enhanced by returning not just a nogood, but a more generic explanation for the failure in the second subproblem. If the second subproblem is linear, then the impossibility of finding a solution to the second subproblem is witnessed by a linear combination of the constraints $SumExpr \geq SumNumber$ where $SumExpr$ is positive or zero, but $SumNumber$ is negative.

The actual dual values which yield this inconsistency not only show the inconsistency of the given subproblem, but may witness the inconsistency of other assignments to the shared variables. By replacing the values of the shared variables with new variables, and combining the linear constraints in the second subproblem using the same dual values we get a new linear constraint called a *Benders Cut* which can be posted to the first subproblem as an additional constraint. Benders Cuts can be used not only to exclude inconsistent subproblems, but also subproblems which cannot participate in an optimal solution to the whole problem [HO03, EW01].

4.3.2 Constraining the Second Subproblem - Column Generation

Curiously there is a form of hybridisation where an optimal solution to the first problem can be used to generate "nogood" constraints on the second problem. This is possible when any solution to the first problem is created using a combination of solutions to the second problem.

This is the case, for example where a number of tasks have to be completed by a number of resources. Each resource can be used to perform a subset of the tasks: finding a set of tasks that can be performed by a single resource (a "line of work") is the subproblem. Each line of work has an associated cost. The master problem is to cover all the tasks by combining a number of lines of work at minimal cost. Given an initial set of lines of work, the master problem is to find the optimal combination. The shadow prices for this solution associate a price with each task. A new line of work can only improve the optimum if its cost is less than the sum of the shadow prices of its set of tasks. Such lines of work are then added to the master problem. They appear as columns in the internal matrix used to represent the master problem in the LP solver. This is the reason for the name *Column Generation*. This is the requirement that is added as a constraint on the subproblem.

This technique applies directly to the workforce scheduling problem. A preliminary solution is computed where each employee performs just one task, for which he/she is qualified. This solution is then improved by seeking sets of tasks that can feasibly be performed by an employee, at an overall cost which is less than the sum of the shadow prices associated with the tasks. The master problem then finds the best combination of employee lines of work to cover all the tasks. This optimal solution yields a modified set of shadow prices which are used to constraint the search for new improving lines of work. This continues

until there are no more lines of work that could improve the current optimal solution to the master problem.

Chapter 5

Applications

Perhaps the first hybrid application using the CP framework was an engineering application involving both continuous and discrete variables [VC88]. The advantages of a commercial linear programming system in conjunction with constraint programming was discovered by the Charme team, working on a transportation problem originally tackled using CP. Based on this experience we used a combination of Cplex and ECLiPSe on some problems for BA [THG+98]. This worked and we began to explore the benefits of the combination on some well known problems [RWT99].

Hybrid techniques proved particularly useful for hybrid scheduling applications [DDLMZ97, JG01, Hoo05, BR03, RW98, EW00]. Hybridisation techniques based on Lagrangian relaxation and column generation were explored, as a way of combining LP and CP [SZSF02, YMdS02, SF03].

Finally these techniques began to be applied in earnest for real applications at Parc Technologies in the area of networking [EW01, CF05, OB04] and elsewhere for sports scheduling [ENT02, Hen01].

Chapter 6

Hybrid Search

6.1 Context

Search is needed to construct solutions to the *awkward* part of any combinatorial problem. When the constraint solvers have done what inferences they can, and have reached a fixpoint from which no further information can be extracted, then the system resorts to search. Because of this context it is impossible to say a priori that one search algorithm is more appropriate for a certain problem than another. There is always a chance that the non-recommended search algorithm happens to chance on an optimal solution straightaway.

Moreover there are almost endless possibilities for search hybridisation. Indeed having read and forgotten many research papers on search hybrids, I am aware that I cannot hope to survey them all.

My objective here is simply to paint the big picture, or create a global map, so that different techniques can be located, and their relationships can be understood.

6.2 Overview

We distinguish two main search methods:

- Constructive search, which adds constraints incrementally to partial solutions
- Local search, which seeks to improve a (set of) complete assignments of values to the variables by making small "local" changes

6.2.1 Constructive Search

The simplest form of constructive search is *greedy* search, which constructs a solution incrementally by adding constraints, but never reconsiders any alternatives in the light of new information (such as a dead-end).

Another form of constructive search is *labelling*, where the constraint added at each step equates a variable to a specific value. Clearly labelling can also be "greedy".

The general form of constructive search is *tree search*. Under each node of the tree, except the *leaf* nodes, there are several branches, and a different constraint is added at each branch. Each branch ends at another node.

The tree is complete if the disjunction of the constraints added on the branches under a node is entailed by the subproblem at the node. The subproblem at a node is a conjunction of the original problem with all the constraints added on its ancestor branches.

Optimisation can be achieved using tree search. This could be achieved by finding all solutions and keeping the best. However the *branch and bound* method improves on this by adding, after each new solution is found, a constraint that requires future solutions to be better than the current best one. After adding a new constraint like this, search can continue on the original search tree - since the constrained search tree is a subtree - or it can restart with a new search tree.

A search tree can be explored completely, or incompletely. Indeed greedy search can be seen as a form of incomplete tree search. The are many ways of limiting the search effort, to yield an incomplete tree search. The simplest is to stop after a specified maximum time. Similarly the number of backtracks can be limited, or a limited amount of search *credit* can be allocated to the search algorithm, and the algorithm can share that credit - which represents search effort - in different ways among the subtrees. For example *beam* search allows only a limited number of branches to be explored at each level of the search tree.

Finally the branches in a search tree can be explored in different orders. So even for complete search we can distinguish depth-first search, breadth-first, best-first, and others.

6.2.2 Local Search

For local search we need to associate with each complete assignment a value which we will call its *price*. The simplest form of local search is *Monte Carlo* search, which just tries different complete assignments at random, and just keeps the ones with the highest price. *Hill climbing* introduces the concept of a *neighbour*. Each neighbour of a complete assignment A is nearly the same as A up to a local change. The hill climbing algorithm moves from an assignment A to one of its neighbours B if B has a higher price than A. Hill-climbing stops when the current assignment has no neighbours with a higher price. There are variants of hill climbing where "horizontal" moves are allowed, and variants where at each step the neighbour with the highest price is chosen.

The simplest kind of local change is to change the value of a single variable. (This can be seen as the local search equivalent of labelling.)

For constrained problems special local changes are introduced which maintain the constraint, such as the two- and three- swaps used for the travelling

salesman problem.

Sometimes complex changes are used which involve several sequential local changes, for example in Lin and Kernighan's TSP algorithm [LK73].

Ultimately, as for example in *variable neighbourhood search* the change may not be local at all [MH97].

The drawback of hill climbing is that it stops on reaching a "local optimum" which may be far worse than the global optimum. Many forms of local search have been introduced which are designed to escape from local optima.

Some of these can work with just one complete assignment, such as Simulated Annealing and Tabu search. Others work on a whole population of solutions, combining them in various ways to yield (hopefully) better solutions. Examples include genetic algorithms, ant colony optimisation and several other methods which are inspired by nature.

6.3 Forms of Search Hybridisation

The benefits of hybrid search are similar to the benefits of hybrid constraint solving:

- Derive more information about feasibility
- Derive more information about optimality
- Derive more heuristic information

In principle tree search is useful for providing information about feasibility, and local search for providing information about optimality. Local search is also an excellent source of heuristic information.

6.4 Hybrid Solvers and Search

In this section, however, we will start by exploring how hybrid constraint solvers feed information to a non-hybrid constructive and local search.

6.4.1 Constructive Search

Clearly all the active solvers perform inference at each node of the search tree, inferring new information from the extra constraint posted on the previous branch.

In addition to inferring logical information, the solvers can export heuristic information. The FD solver can export domain sizes and the linear solver can export reduced cost information to be used for a max regret variable choice heuristic.

Moreover the linear relaxed solution exported from the linear solver, may be propagated onto other variables by the one-way solver, and the new tentative

values can contribute to constraint violations of measurable size. This can serve as another variable choice heuristic.

To this point we have made little mention of *value* choice heuristics. One source of value choice heuristics is the previous solution, in a changed problem which results from modifying the previous one. Another source is a solution to a relaxed (typically polynomial) subproblem, such as the subproblem defined by the linear constraints only. Let us suppose there are different value choice heuristics exported to the solver. When the heuristics agree, this is a pretty powerful indicator that the heuristic value is the best one.

Limited Discrepancy Search [HG95] is an incomplete constructive search method where only a limited number of variables are allowed to take non-preferred values. An idea due to Caseau is only to count the preferences when we are confident about the heuristic [CLPR01]. Using multiple value choice heuristics, we only count discrepancies from the heuristic suggestion *when the different value choice heuristics agree*. This agreement can also be used as a value choice heuristic: label first those variables whose value choices we are confident of.

6.4.2 Local Search

One important mechanism for escaping from local optima, in a local search algorithm, is to increase the penalty of all constraints violated at this local optimum. This changes the optimisation function until the assignment is no longer a local optimum [VT99]. The global optimum can hopefully be found when the constraint penalties have the right penalties.

We have already encountered the requirement to find the right penalties in an LP framework: Lagrangian relaxation. The penalty optimisation is often performed by a local improvement technique termed *subgradient optimisation*.

We have essentially the same technique within a local search setting [SW98, CLS00]. It would be interesting to characterise the class of problems for which ideal constraint penalties exist, that ensure there are no local optima, since all local optima are "globally" optimal. Clearly LP problems have this property, but are there larger classes?

6.5 Loose Search Hybridisation

6.5.1 Constructive then Local Search

The simplest and most natural form of search hybridisation is to use a constructive search to create an initial solution, and then to use local search to improve it. Local search routines need an initial solution to start with, and the quality of the initial solution can have a very significant impact on the performance of the local search. Constructive search is the only way an initial solution can be constructed, of course! Typically a greedy search is used to construct the initial solution, and constraint propagation plays a very important role in maximising

the feasibility of the initial solution. However when the domain of a variable becomes empty instead of failing the greedy search method chooses for that variable a value in conflict and continues with the remaining variables.

For industrial applications where constructive search is used as the heart of the algorithm, because of its suitability for dealing with hard constraints, there is a risk that the final solution suffers from some obvious non-optimality. If any users of the system can see such "mistakes" in the solution constructed by the computer, there is a loss of confidence. To avoid this, for many industrial applications which are handled using constructive search, a local search is added at the end to fix the more obvious non-optimalities.

For example in a workforce scheduling problem, the final solution can be optimised by trying all ways of swapping a single task from one employee to another and accepting any swap that improves the solution.

6.5.2 Local Search then Constructive

This is a rarer combination. The idea is that the local search procedure reaches a "plain" - an area where further improvement is hard to achieve. Indeed statistical analysis of local improvement procedures show a rate of improvement which decreases until new, better, solutions are rarely found.

At this point a change to a complete search procedure is possible. Indeed, by learning which variable values have proven their utility during the local search procedure, the subsequent complete search can be restricted only to admit values with a higher utility, and can converge quickly on better solutions than can be found by local search [Li97].

More generally, constructive branch and bound algorithms typically spend more time searching for an optimal solution than on proving optimality. Often after finding an intermediate best solution, the search "goes down a hole" and takes a long time to find a better solution. After a better solution is found, then the added optimisation constraint enables a number of better and better solutions are found quite quickly, because a large subtree has been pruned by the constraint. Consequently, using local search to quickly elicit a tight optimisation constraint can be very useful for accelerating the constructive branch and bound search.

This combination is typically used where a proof of optimality is required.

6.6 Master-Slave Hybrids

A variety of master/slave search hybrids are applied to a didactic transportation problem in an interesting survey paper [FLL04].

6.6.1 Constructive search aided by local search

As a general principle, the inference performed at each node in a constructive search may be achieved using search itself. For example when solving problems

involving boolean variables, one inference technique is to try instantiating future boolean variables to 1 and then to 0, and for each alternative applying further inference to determine whether either alternative is infeasible. If so, the boolean variable is immediately instantiated to the other value. This kind of inference can exploit arbitrarily complex subsearches.

Local search can be used in the same way to extract heuristic information. One use is as a variable labelling heuristic in satisfiability problems. At each node in the constructive search tree, use local search to extend the current partial solution as far as possible towards a complete solution. The initial solution used by local search can be the local search solution from a previous node. The variable choice heuristic is then to choose a variable in conflict in this local search solution [WS02].

Another use as a value choice heuristic is to extend each value choice to an optimal solution using local search, and choose the value which yields he best local search optimum. Combining this with the above variable choice heuristic results in a different local search optimum being followed at each node in the constructive search tree.

A quite sophisticated example of this hybridisation form is the "local probing" algorithm of [KE02]. The master search algorithm is a constructive search where at each node a linear temporal constraint is added to force apart two tasks at a bottleneck. This is similar to the algorithm mention in Section 4.2.3 above [EW00]. However the slave algorithm is a local algorithm which performs simulated annealing to find a good solution to the temporal subproblem. The resource constraints are handled in the optimisation function of the subproblem. Moreover this is a three-level hybridisation, because the local move used by the simulated annealing algorithm is itself a constructive algorithm. This bottom level algorithm creates a move by first arbitrarily changing the value of one variable. It then finds a feasible solution as close as possible to the previous one, but keeping the new value for the variable. If their is a single feasible solution constructible from the initial value assignment, then the move operator is guaranteed to find it. This algorithm is therefore complete in the sense that it guarantees to find a consistent solution if there is one, but it sacrifices any proof of optimality.

6.6.2 Local search aided by constructive search

When neighbourhoods are large, or if the search for the best neighbour is non-trivial, then constructive search can be used effectively for finding the next move in a local search algorithm.

One method is to use constraint propagation to focus the search for the best neighbour [PG99]. A more specialised, but frequently occurring, problem decomposition is to have local search find values for certain key variables, and allow constructive search to fill in the remaining ones. Indeed the simplex algorithm is an example of this kind of hybrid. For workforce scheduling we might use local search to allocate tasks to employees and constructive search to create an optimal tour for each employee. In this hybrid form the constructive

search is, in effect, calculating the cost function for the local search algorithm.

An example of this kind of decomposition is the very efficient tabu search algorithm for job shop scheduling described in [NS96]. A move is simply the exchange of a couple of tasks on the current solution's critical path. This is extended to a complete solution using constructive search.

The "combine and conquer" algorithm [BB98] is an example of this hybrid where a genetic algorithm is used for local search. The genetic algorithm works not on complete assignments, but on combinations of subdomains, one for each variable. A crossover between two candidates is a mixing and matching of the subdomains. The quality of a candidate is determined by a constructive search which tries to find the best solution possible for the given candidate within a limited time.

6.7 Complex Hybrids of Local and Constructive Search

There has been an explosion of research in this area over the last decade. Papers have been published in the SSM research community, the SAT community, the management science community and others. Some interesting collections include *The Knowledge Engineering Review, Vol 16, No. 1* and the *CPAIOR* annual conference.

In this section we will review two main approaches to integrating constructive and local search:

- Interleaving construction and repair
- Local search in the space of partial solutions

6.7.1 Interleaving Construction and Repair

Earlier we discussed how to apply local search to optimise an initial solution produced by a constructive search. However during the construction of the initial solution, the search typically reaches a node where all alternative branches lead to inconsistent subnodes. Instead of completing the construction of a complete infeasible assignment, some authors have proposed repairing the partial solution using local search [JL00].

This is applicable in case labelling used for the constructive search: extending the approach to other forms of constructive search is an open research problem. The local search is initiated with the first constructed infeasible partial assignment, and stops when a feasible partial assignment has been found. Then constructive search is resumed until another infeasible node cannot be avoided. This interleaved search continues until a feasible solution is found.

I have applied this technique in a couple of industrial applications. Each time the neighbourhood explored by the local search has been designed in an application-specific way. An assignment involved in the conflict has been

changed, so as to remove the conflict, but this change often causes a further conflict, which has to be fixed in turn. This iterated fixing is similar to *ejection chains* used in vehicle routing problems [RR96].

Caseau noted that this interleaving of construction and local search yields faster optimisation and better solutions than is achieved by constructing a complete (infeasible) assignment first and then applying local search afterwards [CL99]. Again this observation is borne out by my experience in an industrial crew scheduling problem, where previous approaches using simulated annealing produced significantly worse solutions in an order of magnitude longer execution time.

6.7.2 Local Search with Consistent Partial Solutions

There is a long line of research driven by the problem of what to do when a constructive search reaches a node whose branches all lead to inconsistent subnodes.

Weak commitment search [Yok94] stops the constructive search, and records the current partial assignment as tentative values. Constructive search then restarts at the point of inconsistency, minimising conflicts with the tentative values, but where conflict cannot be avoided, assigning new values to them. Each time a dead-end is reached the procedure iterates. Theoretical completeness of the search procedure is achieved by recording *nogoods*. An incomplete variant is to forget the nogoods after a certain time, effectively turning them into a tabu list.

Another related approach - which was introduced at the same conference - is to commit to the first value when restarting, and then try to label variables, as before, in a way consistent with the tentative variables. In this way all the variables eventually become committed, and completeness is achieved by trying alternatives on backtracking [VS94].

More recently, however, researchers have been prepared to sacrifice completeness, but have kept the principle of maintaining consistent partial solutions. In this framework a local move is as follows

1. Extend the current partial solution consistently by instantiating another variable

2. If no consistent extension can be found, uninstantiate a variable in the current partial solution

This approach was used in [Pre02], and termed *decision repair* in [JL00].

This framework can be explored in many directions:

- What variable and value choice heuristics to use when extending consistent partial solutions

- What propagation and inference techniques to use when extending consistent partial solutions

- What explanations or nogoods to record when a partial solution cannot be extended
- What variable to uninstantiate when a partial solution cannot be extended

In principle this framework can be extended to allow inconsistent partial solutions as well. With this tremendous flexibility almost all forms of search can be represented in the framework, from standard labelling to hill climbing [PV04].

Chapter 7

Summary

Hybrid techniques are exciting and an endless source of interesting research possibilities. Moreover they are enabling us to take great strides in efficiency and scalability for solving complex industrial combinatorial optimisation problems.

Unpredictability is perhaps the greatest practical problem we face when solving large scale combinatorial optimisation problems. When we first tackled the hoist scheduling problem using a hybrid FD/linear solver combination, what really excited me was the robustness of the algorithm for different data sets.

The downside is that it is hard to develop the right hybrid algorithm for the problem at hand. The more tools we have the harder it is to choose the best combination [WS02].

I believe that after an explosion of different algorithms, frameworks such as that of [PV04] will emerge and we will have a clear oversight of the different ways of combining algorithms. Eventually, a very long time in the future, we may even be able to configure them automatically. That is the holy grail for combinatorial optimisation.

Bibliography

[AAH95] N. Azarmi and W. Abdul-Hameed. Workforce scheduling with constraint logic programming. *BT Technology Journal*, 13(1), 1995.

[AdVD+04] Carlos Ansótegui, Alvaro del Val, Iván Dotú, Cèsar Fernàndez, and Felip Manyà. Modeling choices in quasigroup completion: Sat vs. csp. In *AAAI*, pages 137–142, 2004.

[BB98] Nicolas Barnier and Pascal Brisset. Combine and conquer: Genetic algorithm and cp for optimization. In *CP '98: Proceedings of the 4th International Conference on Principles and Practice of Constraint Programming*, page 463, London, UK, 1998. Springer-Verlag.

[BR03] C. Beck and P. Refalo. A hybrid approach to scheduling with earliness and tardiness costs. *Annals of Operations Research*, 118:49–71, 2003.

[CF05] W. Cronholm and Ajili F. Hybrid branch-and-price for multicast network design. In *Proceedings of the 2nd International Network Optimization Conference (INOC 2005),*, pages 796–802, 2005.

[CL99] Y. Caseau and F. Laburthe. Heuristics for large constrained vehicle routing problems. *Journal of Heuristics*, 5(3), 1999.

[CLPR01] Yves Caseau, Francois Laburthe, Claude Le Pape, and Benoit Rottembourg. Combining local and global search in a constraint programming environment. *Knowl. Eng. Rev.*, 16(1):41–68, 2001.

[CLS00] Kenneth M. F. Choi, Jimmy H. M. Lee, and Peter J. Stuckey. A lagrangian reconstruction of genet. *Artif. Intell.*, 123(1-2):1–39, 2000.

[CLW96] B. M. W. Cheng, J. H. M. Lee, and J. C. K. Wu. Speeding up constraint propagation by redundant modeling. In *Principles and Practice of Constraint Programming*, pages 91–103, 1996.

[DDLMZ97] Ken Darby-Dowman, James Little, Gautam Mitra, and Marco Zaffalon. Constraint logic programming and integer programming approaches and their collaboration in solving an assignment scheduling problem. *Constraints*, 1(3):245–264, 1997.

[ENT02] K. Easton, G. Nemhauser, and M. Trick. Solving the travelling tournament problem: A combined integer programming and constraint programming approach. In *4th International Conference on the Practice and Theory of Automated Timetabling (PATAT)*, number 2740 in LNCS, 2002.

[EW00] H. El Sakkout and M. Wallace. Probe backtrack search for minimal perturbation in dynamic scheduling. *Constraints*, 5(4), 2000.

[EW01] A. Eremin and M. Wallace. Hybrid benders decomposition algorithms in constraint logic programming. In *Proc. Constraint Programming*, number 2239 in LNCS, pages 1–15. Springer, 2001.

[FLL04] F. Focacci, F. Laburthe, and A. Lodi. Local search and constraint programming. In *Constraint and Integer Programming Toward a Unified Methodology*, volume 27 of *Operations Research/Computer Science Interfaces Series*, chapter 9. Springer, 2004.

[FLM04] F. Focacci, A. Lodi, and M. Milano. Exploiting relaxations in CP. In *Constraint and Integer Programming Toward a Unified Methodology*, volume 27 of *Operations Research/Computer Science Interfaces Series*, chapter 5. Springer, 2004.

[Ger01] Carmen Gervet. Large scale combinatorial optimization: A methodological viewpoint. *DIMACS Series in Discrete Mathematics and Computers Science*, 57:151–175, 2001.

[Hen01] Martin Henz. Scheduling a major college basketball conference–revisited. *Oper. Res.*, 49(1):163–168, 2001.

[HG95] W. D. Harvey and M. L. Ginsberg. Limited discrepancy search. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 607–615, 1995.

[HO99] J. N. Hooker and M. A. Osorio. Mixed logical / linear programming. *Discrete Applied Mathematics*, 96-97:395–442, 1999.

[HO03] J. N. Hooker and G. Ottosson. Logic-based Benders decomposition. *Mathematical Programming*, 96:33–60, 2003.

[Hoo05] J. N. Hooker. A hybrid method for planning and scheduling. *Constraints*, 10(4), 2005.

[JG01] V. Jain and I. Grossmann. Algorithms for hybrid MILP/CP models for a class of optimization problems. *INFORMS Journal on Computing*, 13(4):258–276, 2001.

[JL00] Narendra Jussien and Olivier Lhomme. Local search with constraint propagation and conflict-based heuristics. In *Proceedings of the Seventh National Conference on Artificial Intelligence (AAAI 2000)*, pages 169–174, Austin, TX, USA, August 2000.

[KE02] Olli Kamarainen and Hani El Sakkout. Local probing applied to scheduling. In *Principles and Practice of Constraint Programming*, pages 155–171, 2002.

[Li97] Yinghao Li. *Directed Annealing Search In Constraint Satisfaction and Optimisation*. PhD thesis, IC-Parc, 1997.

[LK73] S. Lin and B. W. Kernighan. An effective heuristic algorithm for the traveling salesman problem. *Operations Research*, 21:498–516, 1973.

[MH97] N. Mladenović and P. Hansen. Variable neighborhood search. *Comps. in Opns. Res.*, 24:1097–1100, 1997.

[NS96] Eugeniusz Nowicki and Czeslaw Smutnicki. A fast taboo search algorithm for the job shop problem. *Manage. Sci.*, 42(6):797–813, 1996.

[OB04] W. Ouaja and Richards E. B. A hybrid multicommodity routing algorithm for traffic engineering. *Networks*, 43(3):125–140, 2004.

[PG99] Gilles Pesant and Michel Gendreau. A constraint programming framework for local search methods. *Journal of Heuristics*, 5(3):255–279, 1999.

[Pre02] S. Prestwich. Combining the scalability of local search with the pruning techniques of systematic search. *Annals of Operations Research*, 115, 2002.

[PV04] Cédric Pralet and Gérard Verfaillie. Travelling in the world of local searches in the space of partial assignments. In *CPAIOR*, pages 240–255, 2004.

[RR96] C. Rego and C. Roucairol. A parallel tabu search algorithm using ejection chains for vehicle routing. In *Meta-Heuristics: Theory and Applications*. Kluwer, 1996.

[RW98] R. Rodosek and M. G. Wallace. A generic model and hybrid algorithm for hoist scheduling problems. In *CP '98: Proceedings of the 4th International Conference on Principles and Practice of Constraint Programming*, pages 385–399, London, UK, 1998. Springer-Verlag.

[RWH99] R. Rodosek, M. G. Wallace, and M. Hajian. A new approach to integrating mixed integer programming with constraint logic programming. *Annals of Operations research*, 86:63–87, 1999.

[RWT99] R. Rodosek, M. G. Wallace, and Hajian M. T. A new approach to integrating mixed integer programming with constraint logic programming. *Annals of Operations Research*, 86:63–87, 1999.

[RYS02] D. Riera and N. Yorke-Smith. An improved hybrid model for the generic hoist scheduling problem. *Annals of Operations Research*, 115:173–191, 2002.

[SBHW95] Barbara Smith, Sally Brailsford, Peter Hubbard, and H. Paul Williams. The Progressive Party Problem: Integer Linear Programming and Constraint Programming Compared. In *CP95: Proceedings 1st International Conference on Principles and Practice of Constraint Programming)*, Marseilles, 1995.

[SF03] Meinolf Sellmann and Torsten Fahle. Constraint programming based lagrangian relaxation for the automatic recording problem. *Annals of Operations Research*, 118:17–33, 2003.

[SW98] Y. Shang and B. Wah. A discrete lagrangian-based global-search method for solving satisfiability problems. *Journal of Global Optimization*, 12(1):61–100, 1998.

[SZSF02] M. Sellmann, K. Zervoudakis, P. Stamatopoulos, and T. Fahle. Crew assignment via constraint programming: Integrating column generation and heuristic tree search. *Annals of Operations Research*, 115:207–226, 2002.

[THG+98] Hajian M. T., El-Sakkout H. H., Wallace M. G., Richards E. B., , and Lever J. M. Towards a closer integration of finite domain propagation and simplex-based algorithms. *Annals of Operations Research*, 81:421–431, 1998.

[Van98] P. Van Hentenryck. A gentle introduction to Numerica. *Artificial Intelligence*, 103:209–235, 1998.

[VC88] Pascal Van Hentenryck and Jean-Philippe Carillon. Generality versus specificity: An experience with ai and or techniques. In *AAAI*, pages 660–664, 1988.

[VM00] Pascal Van Hentenryck and Laurent Michel. Localizer: A modeling language for local search. *Constraints*, 5:41–82, 2000.

[VS94] Gerard Verfaillie and Thomas Schiex. Solution reuse in dynamic constraint satisfaction problems. In *AAAI '94: Proceedings of the twelfth national conference on Artificial intelligence (vol. 1)*, pages 307–312, 1994.

[VT99] C. Voudouris and E.P.K Tsang. Guided local search. *European Journal of Operational Research*, 113:469–499, 1999.

[Wil99] H. P. Williams. *Model Building in Mathematical Programming.* Wiley, 1999.

[WS02] M. G. Wallace and J. Schimpf. Finding the right hybrid algorithm - a combinatorial meta-problem. *Annals of Mathematics and Artificial Intelligence*, 34(4):259 – 269, 2002.

[YMdS02] T. H. Yunes, A. V. Moura, and C. C. de Souza. Hybrid column generation approaches for urban transit crew management problems. *Transportation Science*, 39(2):273–288, 2002.

[Yok94] Makoto Yokoo. Weak-commitment search for solving constraint satisfaction problems. In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI-94); Vol. 1*, pages 313–318, Seattle, WA, USA, 1994.

[ZMMM01] Lintao Zhang, Conor F. Madigan, Matthew H. Moskewicz, and Sharad Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *ICCAD '01: Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, pages 279–285, 2001.