Search in AI - Escaping from the CSP Straightjacket

Mark Wallace [1]

Abstract. We investigate some useful strategies for solving a variety of hard search problems. In the process we identify weaknesses in the standard CSP formalisation of such problems.

1 Search in Artificial Intelligence

A basic challenge in Artificial Intelligence is to understand how people solve problems. In 1968 Herbert Simon wrote [14]:

> Problem solving is often described as a search through a vast maze of possibilities...Successful problem solving involves searching that maze selectively and reducing it to manageable proportions

Simon suggested that people solve problems by following search strategies. Ultimately, the study of human problem solving reduces to the study of search strategies and how they are developed and applied:

> Once the strategy is selected, the course of search depends only on the structure of the problem, not on any characteristics of the problem solver.

Since 1968 there has been a great deal of investigation into search strategies. The reason has been that many tough problems in AI - and other disciplines - can only be solved by search, and without a good search strategy these problems could not be solved at all (in any useful time span). What have emerged are a variety of techniques that can be combined into search strategies tuned to specific problem classes. One set of techniques, for example, are those associated with *finite domains*, including *forward checking* and *fail first*.

Simon used as an example the crypt-arithmetic problem `DONALD + GERALD = ROBERT`. He described in detail the set of choices which might be pursued by a thinking person in solving the problem. It is truly remarkable to discover how closely his description fits the behaviour of a constraint program solving this problem using the very techniques mentioned above: *forward checking* and *fail first*.

In the following we will explore search strategies for solving hard problems. The challenge is no longer simply to capture the "intelligent" behaviour people use in problem solving. Now researchers are developing and applying more and more sophisticated strategies, so that we can program computers to produce answers to problems that were previously beyond our reach.

[1] ICL and IC-Parc, Imperial College, London SW7 2AZ, UK, email: mgw@icparc.ic.ac.uk

2 Formalising Search - Some Shaky Foundations

2.1 CSP

Before defining and comparing search strategies we need a standard definition of a search problem and a search strategy. One definition for a class of search problems has become standard in the AI community, under the name *CSP* (for Constraint Satisfaction Problem).

> A CSP $< \mathcal{V}, \mathcal{D}, \mathcal{C} >$ is defined as a set of variables $\mathcal{V}$, a set of domains $\mathcal{D}$ and a set of constraints $\mathcal{C}$. Each domain D in $\mathcal{D}$ is a set of values, and each variable is associated with one domain. Each constraint C in $\mathcal{C}$ is a set of tuples, and is associated with an ordered subset of $\mathcal{V}$, which is called its *scope*. Each constraint has a fixed width n. All its tuples are n-ary, and its scope has cardinality n.
> An assignment of one value from its domain to each variable in a CSP is termed a *solution* if the tuple of values assigned to the scope of each constraint belongs to its set of tuples.

CSPs have sufficient expressive power to formalise a very wide class of search problems: arguably all the interesting ones.

A CSP can be solved by searching through all possible assignments until a solution is found. However this strategy (termed "the British Museum Algorithm" by Hoare [1]), is not guaranteed to terminate unless all the variables' domains are finite, and even then it is unlikely to terminate in any reasonable time even for toy problems.

Two very general search strategies are

- **constructive search**, where values are assigned to variables one-at-a-time, until a complete assignment has been built
- **local search**, where search moves from complete assignment to complete assignment by changing the values of a few variables

These strategies can then be refined by adding further techniques. For example *forward checking* is a refinement of constructive search that, after each assignment of a value to a variable, uses the constraints to reduce the domains of the remaining unassigned variables. *Fail first* is another enhancement to constructive search. The idea is to make the next assignment to the remaining variable which has the smallest domain.

Because of these virtues, I had originally envisaged presenting some of the recent advances in modelling and solving hard search problems using the CSP formalism. However the deficiencies of the CSP formalism proved frustrating enough to dissuade me.

2.2 The Drawbacks of CSP

The drawbacks of the CSP formalisation $< \mathcal{V}, \mathcal{D}, \mathcal{C} >$ lie in its set of variables $\mathcal{V}$, its set of domains $\mathcal{D}$, its set of constraints $\mathcal{C}$ and its definition of a *solution*.

2.2.1 A Fixed Set of Variables

Many search problems are most naturally modelled in terms of a set of variables whose size is, at least initially, unknown. The most well-known class of such problems are configuration problems, where the number and range of subsequent choices will depend on earlier ones. (For example in a computer conguration problem, the choice of a powerful processor may - still - entail air conditioning, which gives rise to a number of choices about the size, power, location and configuration of the air conditioning system.)

In fact introducing search problems, Herbert Simon uses the metaphor of a "vast maze". A maze is a search problem involving a sequence of choices where, like configuration problems, later choices depend on earlier ones.

To model a maze in terms of a CSP it is necessary to establish an upper bound on the number of decisions that might be needed in order to navigate to the centre of the maze. One such upper bound is the number of places in the maze where the paths branch. If the CSP includes a variable for each such branching point, then any solution to the maze will include some assignment to the variables which correspond to branching points that do not lie on the path to the centre. In this way the maze can be modelled using a fixed set of variables, but somewhat redundantly.

There is a class of problems, however, where the restriction to a fixed set of variables appears to conflict with the model supporting the best search strategy.

In many industries the raw materials come in fixed sizes, and customer orders must be satisfied from the raw materials with a minimum of wastage. One of the earliest applications of constraint logic programming was of this kind [2]: the customer orders were for a certain number of pieces of wood of different sizes, and the raw material came in the form of large wooden boards from which the pieces were to be cut.

For real industrial problems the number of ways of cutting the original raw material can be very large, so that precomputing all possible cuttings is impractical.

This problem is most naturally formulated using sets. The solution is a multi-set of boards, cut in possibly different ways, such that the sum of all the pieces cut from these boards meets the customer's requirements.

Such problems are often solved using a technique which interleaves the "slave problem" of computing different ways of cutting the raw material with the "master problem" of how many of each of these to use for satisfying the customers' orders. In this case solutions to the slave problem become variables in the master problem. As the algorithm progresses, the number of variables in the master problem grows and there is no reasonable tight bound on the number of variables needed.

Consequently, (while it is possible to formulate this problem as a CSP) it is not possible to formulate it as a CSP in a way that supports the best search strategy.

2.2.2 A Set of Domains

The second component of the formalisation $< \mathcal{V}, \mathcal{D}, \mathcal{C} >$ of a CSP is the set of domains.

The set of domains is redundant: they are simply unary constraints. In principle, then, they are not needed in the CSP formalisation.

The reason they are there is because in the CSP framework domains play a key role in the search strategies. As a result the CSP formalism is quite intimately linked with a particular armoury of search strategies, including the examples - *forward checking* and *fail first* - introduced above.

The drawback is that search strategies which use information other than domains are not as easy to express in a CSP framework. A standard technique for solving search problems in Operations Research is to formalise the problem in terms of two kinds of constraints, linear constraints and integrality constraints. (Problems formulated in this way are termed MIP or *mixed integer problems*). At each node in the search tree, the linear constraints are solved. If the linear constraints have no solution, then neither does the original problem.

This approach can be enhanced to handle problem which involve other kinds of numeric constraints. Each non-linear constraint is approximated as closely as possible by a set of linear constraints, and then the same technique is applied, solving the "relaxed" problem, comprising solely the linear constraints.

This argument might suggest adding the linear constraints as another component in the specification of a CSP. However the same argument can be applied to several other kinds of information about any search problem.

Another piece of information about a variable is the number of constraints within whose scope it lies. This information proves particularly useful when constraints are automatically simplified or eliminated when they become redundant during search.

More radically, heuristic information is often very important input for a search strategy. For example if a similar problem has been solved before, then the previous solution is a useful heuristic to guide the search for a solution to the new problem. The previous solution should not be kept as a complete assignment, because then it could not be used during constructive search. Instead, the information can be held as a *tentative* assignment to the variables [16].

The kinds of information which can support different search strategies is unlimited. Obviously it is not possible to capture all these kinds of information in the formalisation of search problems. **There seems no reason to single out variable domains, from all the other kinds of information, and promote them to a special status.**

2.2.3 A Set of Constraints

The third component of the formalisation $< \mathcal{V}, \mathcal{D}, \mathcal{C} >$ of a CSP is the set of constraints. Mathematically every constraint denotes a set of tuples, so the formalisation is correct. **However, the formalisation of constraints in CSP admits no structure. It fails to capture the semantics of the different classes of constraints.**

In the previous section we identified one special class of constraints: linear numeric constraints. van Hentenryck et.al. [15] introduced a particularly efficient propagation algorithm for some further classes of constraints. Jeavons et.al. [8] presented a criteria on the constraints of a CSP that could predict how hard it would be to solve.

Research strategies that apply different techniques to different classes of constraints within a problem can only be applied after classifying the constraints appropriately. Unfortunately the formalisation of a constraint as a set of tuples obliterates its structure. An inequality constraint $X + 3 \geq Y$ is possible to classify when presented *intentionally* as a mathematical formula, but this is hard to classify when presented as an infinite set of pairs of numbers.

Furthermore the restriction that constraints have a fixed width makes it impossible to handle "global" constraints. A global constraint is a constraint on a list of variables whose propagation behaviour is tailored to the constraint. The algorithm supporting this behaviour is independent of the number of variables in the list. In other words the behaviour is independent of the cardinality of the

scope of the constraint.

Consider the pigeon-hole problem. This can be formalised, like all CSPs, as a satisfiability problem whose constraints all have width three. However this formalisation proves extremely hard to solve (indeed it cannot be solved by resolution in anything less than exponential time [6].) However using the *alldistinct* global constraint, and its specific propagation algorithm, failure is detected in a single propagation step.

According to the CSP formalisation, each time the *alldistinct* constraint has a scope of different cardinality, it is a different constraint. Consequently each different pigeonhole problem requires a different formalisation, and there is no natural mapping from pigeonhole problems represented as CSPs to the one *alldistinct* propagation strategy.

2.2.4 Solutions

A solution to a CSP is an assignment to all its variables which satisfy the constraints. This definition is mathematically correct. The decision problem, which asks whether a given CSP has any solutions, relies on the above definition of a solution. Formally, to solve a CSP is to answer the decision problem. We should not go further and identify solving a CSP with searching for such a solution.

The CSP formalisation places too much emphasis on search techniques that work by assigning values to variables. There are many problems for which an intensional description of the set of solutions is much more useful than a list of assignments (especially in case the list is infinite)!

For mixed integer problems, a very useful representation of the set of solutions is the tightest conjunction of linear constraints which are satisfied by all the integer solutions. For scheduling problems, the temporal ordering of tasks is more important than their precise starting times. In both cases the answer to the decision problem is easily extracted from intensional representation of the solutions.

2.3 Optimisation

The CSP formalism does not include an optimisation function. However it can be easily enhanced to include optimisation, yielding CSOP. Given an optimal value, the "proof of optimality" is simply another CSP, so prima facie the difference between CSP and CSOP is not very important.

However the introduction of an optimisation function strongly influences some aspects of CSP and their solution techniques. **CSOPs are much better addressed by local search algorithms than CSPs naturally are.**

Of course CSPs have been increasingly addressed by local search techniques since Minton's initial work on the n-queens problem [9]. However turning a CSP into a CSOP by adding a penalty for each violated constraint ([4]), yields a cost function which may have a very "flat bottom" if the CSP has many solutions.

In fact the shape of the optimisation function is a very important influence on how hard it is to solve. Let us say a CSOP has a *fine granularity* if the number of assignments having any given optimisation function value is small. If many assignments have the same value, the CSOP has a *coarse granularity*. It is easy to see that CSOPs with a coarse granularity are in general easier to solve than ones with a fine granularity. The special case where very few assignments have the optimal value makes a CSOP relatively hard to solve.

The concept of a phase transition, which helps predict which uniform, random CSPs are likely to be hard to solve, is conspicuously less useful for CSOPs. In general all fine granularity CSOPs are hard.

The third factor that makes CSOPs rather different from CSPs is the speed with which the search converges on better and better solutions. As we shall see below, the choice of which value to assign first to a variable is not critical in a CSP. If the problem is unsatisfiable, the value order makes no difference to the total search space. **By contrast, in CSOPs a good value ordering can dramatically speed up the convergence of the search process.**

3 Search Strategies

3.1 Extending Constructive Search

3.1.1 Constructive Search

In our house we lose things every day. Having no idea where they are we search everywhere till (on a good day) we find them. For "blind" search, where there is no information about the problem available, the only possible strategy is Hoare's British Museum algorithm.

However the interesting problems are those where the constraints can be formalised, quite compactly. Indeed for the class of (NP-hard) problems we are interested in, the number of complete assignments grows exponentially with the size of the problem statement.

In this case it is possible to enhance constructive search by checking each constraint as soon as all the variables in its scope have been assigned a value (or "instantiated").

A further enhancement is to reason on constraints that are not yet (completely) instantiated. Often this reasoning takes the form of logical deduction, and it produces new (or "tighter") constraints on those variables which have not yet been instantiated. We term this *constraint propagation*. Some examples are forward checking, arc- and path-consistency [10, 3, 7].

Besides constraint propagation, there are other techniques that can be used to enhance the performance of constructive search. The algorithm must select, at each search step

- which variable to assign next, and
- which value for that variable to try first.

Heuristics governing the choice of variable and value can have a significant effect on search performance.

In principle it is best to assign next the variable whose removal from the problem renders the remaining subproblem easiest to solve. If the problem has only binary constraints, and forward checking is applied at each search step, then each constraint with the chosen variable in its scope can simply be dropped from the subproblem. In this case the variable choice can be automated using a measure κ of problem constrainedness [5]: always choose the variable whose removal yields the least constrained subproblem.

3.1.2 Beyond Constructive Search

Industrial applications have forced us to generalise the underlying constructive search in order to achieve good constraint propagation. This is best illustrated using scheduling problems as an example.

The scheduling problem is to assign a starting time to each of a given set of tasks, so as to satisfy (simple temporal) constraints on their starting and ending times, and to ensure the number of tasks in progress at any time stays below a given limit.

Constructive search can be applied to this problem by assigning a value one-at-a-time to each task starting time. This is in general a poor strategy for such problems since the assigning of a certain time

t to a task may differ insignificantly from the assigning of $t + 1$ to the same task.

It is much more significant to assign a relationship - before, after or overlapping - to a pair of tasks. Once this relationship has been fixed for each pair of tasks, then assigning start times to the tasks can be achieved without any need to search through alternative assignments.

For this purpose a scheduling problem can be modelled by introducing a variable for each pair of tasks, whose value represents their relationship. Constraints enforce the consistency between the "relationship" variables and the start time variables. Further constraints are also needed to maintain the relationship between pairs of relationship variables. For example if *task1 before task2* and *task2 before task3* then *task1 before task3*.

When this approach is used for non-toy problems, with say 100 tasks, the number of "relationship" variables needed is 10,000. The number of constraints between the relationship variables is $100 \times 100 \times 100$, which is a million. Unfortunately few computing platforms support such large numbers of constraints, and applying constraint propagation techniques to problems of this size is (currently) impractical.

It is possible to apply a generalisation of constructive search to scheduling problems which is much more scalable. The principle is that instead of assigning a value to a variable at each step, the search routine adds a constraint on a pair of task start or end times. The added constraint, or pair of constraints, enforces one of the relationships - before, after, or overlaps - between two tasks, but achieves it without introducing extra variables. When an ordering has been imposed on all the task start and end times, the problem can then be solved without search as above.

The generalisation of constructive search is to allow constraints to be posted at each search step other than just variable assignments (which are themselves just constraints of the form *Variable = value*).

The standard search technique used by Operations Researchers for solving MIP problems is such a generalisation of constructive search. In this case the constraints added are typically new bounds on the variables. These have the form $Variable \geq integer$ or $Variable \leq integer$. However current linear solvers allow any (set of) linear constraints to be added at a search node. In the hybrid search strategy of [12], for dynamic scheduling problems, the constraints added to the linear solver during search are orderings on task start and end times.

In this context the notion of constrainedness cannot serve as a guide for choosing which constraint to add at a search step. We need to distinguish between constraint classes, and even between individual constraints.

The best illustration of the choice of which constraints to add during search is in an algorithm for solving propositional satisfiability problems [11].

The search strategy analyses the remaining unsatisfied clauses and looks for patterns. Depending on the best pattern, it either chooses a pair of variables X and Y and adds the constraint $X = Y$ or it chooses a single variable and adds $X = true$. (The alternative decisions are respectively $X \neq Y$ and $X = false$.) This approach has been enhanced to achieve worst case performance $O(1.472^n)$.

3.2 Beyond Local Search

Local search moves from complete assignment to complete assignment by changing the values of a few variables. For many applications, in fact, a local move simply switches the value of a single variable (eg [9]).

In this section I want to make the point that a good local search algorithm may involve much more complex moves than simply switching a few variable values. In effect the problem may need radical reshaping so that local moves converge quickly towards a solution.

Local search, as introduced above, deals only with complete assignments. Naturally most such assignments are not solutions, and violate some of the problem constraints. In industrial applications, however, the distinction between satisfying a constraint and not satisfying it is not always so sharp. Some constraints should be satisfied in a solution, but do not have to be. These constraints are called *soft* constraints. Solutions which violate soft constraints incur a penalty.

These penalties help steer local search towards good solutions, that have low penalties. A move from one assignment to another, with a smaller penalty, is generally assumed to be a move "in the right direction". On the other hand a move to another assignment with a higher penalty may be a move in the wrong direction, and may be rejected.

In fact problems with soft constraints are optimisation problems. The penalties associated with soft constraints are components of the cost function. In a sense they are not constraints at all, but merely conditions used in computing the cost function.

For the purposes of local search, even hard constraints are often treated as soft constraints, which can be violated by complete assignments during the search process, though they incur a high penalty. On completion of the local search process, the system then checks these constraints explicitly and continues the search in case any of them are still violated.

For local search to work, there must be some correlation between the cost associated with assignments that are just a few moves away from each other. If there were no such correlation, then local search would be no better than the British Museum algorithm.

A local search strategy that only moved to assignments with smaller and smaller costs (confusingly termed hill climbing) would become trapped in local optima. These are non-optimal assignments whose immediate neighbours (assignments reachable in one move) all have higher costs.

However there are some important problem classes for which hill climbing does guarantee to find the global optimum. One such class is the class of problems with linear equations and a linear optimisation function. All the variables are continuous, so they can take infinitely many different values. Since problems in this class have infinite search spaces, local search would not seem to be a very suitable search strategy.

The traditional technique of solving linear equations, is to take the equations, one at a time, make one variable the subject of the equation, and then eliminate it from the remaining equations by substitution. Eventually either an inconsistency is found (an equation with no solution), or all the equations have been used and some (or all) of the variables have been eliminated. If there are n (linearly independent) equations, and m variables, with $m \geq n$, then there will be $m - n$ variables left.

Under certain assumptions about the equations, the optimal solution can be found by choosing the right set of variables to eliminate (termed the *basis*) and assigning the rest to 0. Moreover, having chosen one basis, if it does not yield the optimal solution, then by switching one variable into the basis and another one out, an equal or better basis can be found.

We can define a move as a switch of one variable in the basis for another, and setting the remaining (non-basic) variables to 0. In this search space, hill climbing does guarantee to yield the optimal

solution.[2]

3.3 Combining constructive and local search

Constructive search can exploit the constraints very well to avoid exploring irrelevant parts of the search space. However constructive search, and even generalised constructive search, has a weakness in addressing optimisation problems. The optimisation function typically involves many variables and, until most of them are instantiated, little can be deduced about the value of this function by constraint propagation.

Local search, by contrast, focusses on optimisation but cannot easily be tuned to handle hard constraints.

By combining the two forms of search in a single algorithm the advantages of each can be exploited, leading to a new kind of search. We present two forms of this algorithm

3.3.1 Combined Search for Solving a CSP

For CSPs, there is no optimisation function in the original problem, so, to guide the local search function, each violated constraint incurs a penalty. The general search strategy is as follows:

1. Generate a complete assignment and optimise it using local search.
2. If the complete assignment is a solution, then stop.
3. Record the value assigned to each variable, abandon the assignment itself, then attach each value to its variable again as a *tentative* value.
4. Find the constraints violated by the assigned and tentative values of the instantiated, respectively uninstantiated, variables in their scope.
5. Choose an unassigned variable in the scope of a violated constraint, and assign a value to it. (Note that this is a choice point: all the values in the variable's domain must be tried to cover the whole search space).
6. Apply local search to the remaining (uninstantiated) problem variables, starting from their previous tentative values. Return to (2) above.

An instance of this algorithm, incorporating constraint propagation within the constructive search component, was applied successfully to some propositional satisfiability problems [13].

3.3.2 Combined Search for Solving Optimisation Problems

For optimisation problems, we introduce a class of constraints $\mathcal{C}_{simp}$ which are respected by the local search algorithm. We call them "simple" constraints. The assignments generated by the local search algorithm always satisfy the simple constraints.

The following algorithm is a generalisation of MIP branch and bound. In MIP branch and bound $\mathcal{C}_{simp}$ is the class of linear constraints, and the local search algorithm a linear solver.

1. Generate an initial complete assignment and optimise it using local search. (This assignment satisfies all the simple problem constraints, and is an optimal solution satisfying these constraints.)
2. If the complete assignment is a solution, then stop.
3. Record the value assigned to each variable, abandon the assignment itself, then attach each value to its variable again as a *tentative* value.
4. For some problem constraint that is violated by the tentative assignments, choose a simple constraint which precludes this violation, and add it to the problem. (Note that this is a choice point: the set of alternative such simple constraints must all be tried to cover the whole search space).
5. Restore the tentative assignments, apply local search and optimise in the light of the newly added simple constraints. Return to (2) above.

The linear solver satisfies a number of conditions which ensure that MIP is sound, complete and terminating. These conditions are as follows:

1. There is a class of simple constraints $\mathcal{C}_{simp}$ which is satisfied by every assignment produced by local search
2. Each problem constraint can be represented by logical combinations of simple constraints.[3]
3. The search is finite. [4]
4. The local search produces the optimal solution for the subproblem comprising only the simple constraints.

Another instance of this generic algorithm was presented at ECAI 1999 [12]. In this case the application was scheduling, $\mathcal{C}_{simp}$ was the class of simple temporal constraints, and the remaining problem constraints were resource utilisation constraints.

4 Applications

Whilst CSPs have been studied for many years in the AI community, many of the applications of constraint technology have addressed problems which cannot be naturally formulated in this way.

Many practical applications involve both continuous variables (representing, for example, time, quantity and distance) and discrete variables (representing resources, tasks, customers and, more generally, logical disjunction).

Constraint technology, and in particular hybrid algorithms of the kind presented in the last section, have underpinned a breakthrough in solving large scale industrial optimisation problems of this kind.

This success has attracted the notice of Operations Researchers and the constraint programming community is growing rapidly as a result of this success.

The research issues are very hard. We tackle difficult problems that have never been solved before, we develop solutions that benefit schools, hospitals, and other organisations, and ultimately we seek to understand the nature of search.

This exciting area brings AI and OR researchers together to tackle problems neither could solve alone. We very much hope you will join us!

5 Appendix

Constraint Logic Programming (CLP) offers a very nice formalism for expressing hard search problems. The following examples show

[2] Whilst we have outlined the Simplex algorithm here, the interior point method is also an instance of local search.

[3] Assume that, for every violation of a problem constraint by a complete assignment returned by local search, there is a finite set of (alternative) simple constraints which it also violates. Each assignment which satisfies the problem constraint also satisfies at least one constraint in this set.

[4] There is no infinite sequence of assignments and simple constraints, such that the i^{th} assignment satisfies the first i constraints but violates all the constraints after the i^{th}.

how each of the features hard to express in the CSP formalisation are expressed using CLP.

5.1 The basic CSP

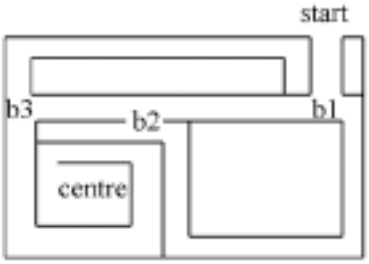
The following CSP has four variables W, X, Y, Z. X has domain $1...10$, and W, Y and Z have the same domain $\{a, b, c, d\}$. The binary constraint q has three tuples and scope X, Y. The ternary constraint p has only two tuples, and scope W, Y, Z

```
% Constraint tuples
q(1,a).    q(2,b).     q(3,c).
p(a,b,c).  p(a,c,c).

% Problem definition
problem(W,X,Y,Z) :-
% Variables and domains
        X:: 1..10, [W,Y,Z]::[a,b,c,d],
% Constraint scopes
        q(X,Y),
        p(Y,Z,W).
```

5.2 An Unknown Number of Variables

The maze problem is most naturally modelled using a variable list, whose length is initially unknown. We solve the following maze:



This maze is searched by the following program:

```
%Constraint tuples
link(start,b1).
% There are two links from b1 to b2
link(b1,start). link(b1,b2).  link(b1,b2).
link(b2,b1).  link(b2,b1).  link(b2,b3).
link(b3,centre). link(b3,dead_end).
link(centre,b3).
%Program
maze(Path) :-
        fromto(start,Here,There,centre),
        fromto([],ThisPath,[Here|ThisPath],Path)
        do    link(Here,There),
              not member(There,ThisPath).
```

5.3 A Constraint of Unknown Width

The pigeonhole example can be modelled very generally.

```
pigeon(Pigeons,Holes) :-
        Pigeons::Holes,
        alldistinct(Pigeons),
        labeling(Pigeons).
```

A particular instance of this class can be solved by invoking

```
?- pigeon([P1,P2,P3,P4],[h1,h2,h3]).
```

5.4 Search by Posting Constraints

This problem has continuous domains, and could not be solved by assigning start times.

```
% Constraint tuples
no_overlap(S1,D1,S2,D2) :- S1 *>= S2+D2.
no_overlap(S1,D1,S2,D2) :- S2 *>= S1+D1.

% Problem definition
schedule(S1,S2,S3) :-
        S1::0.0..10.0, S2::3.0..8.0, S3::0.0..7.0,
        no_overlap(S1,5,S2,5),
        no_overlap(S2,5,S3,5),
        no_overlap(S1,5,S3,5),
        sequence([S1,S2,S3]).

sequence(List) :-
        foreach(S1,List),
        foreach(S2,List)
        do (S1==S2 -> true ; seq(S1,S2)).

seq(S1,S2) :- S1*>=S2.
seq(S1,S2) :- S2*>=S1.
```

ACKNOWLEDGEMENTS

The existence of this paper is thanks to all my colleagues at IC-Parc, and especially Joachim, Kish and Warwick and their brilliant work on the ECLiPSe constraint logic programming platform.

REFERENCES

[1] C.A.R.Hoare, 'An overview of some formal methods for program design', in *Essays in Computing Science*, ed., C. B. Jones, 371 – 388, Prentice Hall, (1989).

[2] M. Dincbas, H. Simonis, and P. Van Hentenryck, 'Solving a Cutting-Stock Problem in Constraint Logic Programming ', in *Fifth International Conference on Logic Programming*, Seattle, WA, (August 1988).

[3] E.C. Freuder, 'Synthesizing constraint expressions', *Communications of the ACM*, **21**(11), 958–966, (November 1978).

[4] Eugene C. Freuder and Richard J. Wallace, 'Partial constraint satisfaction', *Artificial Intelligence*, **58**, 21–70, (1992).

[5] Ian P. Gent, Ewan MacIntyre, Patrick Prosser, and Toby Walsh, 'The constrainedness of search', in *Proceedings of the Thirteenth National Conference on Artificial Intelligence and the Eighth Innovative Applications of Artificial Intelligence Conference*, pp. 246–252, Menlo Park, (August 4–8 1996). AAAI Press / MIT Press.

[6] A. Haken, 'The intractability of resolution', *Theoretical Computer Science*, **39**(2-3), 297–308, (August 1985).

[7] R.M. Haralick and G.L. Elliot, 'Increasing tree search efficiency for constraint satisfaction problems', *Artificial Intelligence*, **14**, 263–314, (October 1980).

[8] Peter Jeavons, 'Constructing constraints', in *Principles and Practice of Constraint Programming - CP98*, Pisa, Italy, (October 1998).

[9] S. Minton, M. D. Johnston, A. B. Philips, and P. Laird, 'Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems', *Artificial Intelligence*, **58**, (1992).

[10] U. Montanari, 'Networks of constraints : fundamental properties and applications to picture processing', *Information Science*, **7**(2), 95–132, (1974).

[11] R. Rodosek, 'A new approach on solving 3-Satisfiability', *Lecture Notes in Computer Science*, **1138**, 197–??, (1996).

[12] H. El Sakkout, T. Richards, and M. Wallace, 'Minimal perturbance in dynamic scheduling', in *Proceedings of the 13th European Conference on Artificial Intelligence (ECAI-98)*, ed., Henri Prade, pp. 504–508, Chichester, (August 23–28 1998). John Wiley & Sons.

[13] J. Schimpf and M. Wallace, ‘Finding the right algorithm - a combinatorial meta-problem’, *Electronic Notes in Discrete Mathematics*, **4**, 80–92, (1999).
[14] Herbert Simon, *The Sciences of the Artificial*, MIT Press, 1969.
[15] P. Van Hentenryck, Y. Deville, and C.-M. Teng, ‘A generic arc-consistency algorithm and its specialisations’, *Artificial Intelligence*, **57**, (1992).
[16] M. Wallace, S. Novello, and J. Schimpf, ‘Eclipse - a platform for constraint programming’, *ICL Systems Journal*, **12**(1), 159–200, (1997).