# Capturing Constraint Programming Experience: A Case-Based Approach

James Little, Cormac Gebruers, Derek Bridge & Eugene Freuder

Cork Constraint Computation Centre[⋆]
University College, Cork
Ireland
{j.little,c.gebruers,d.bridge,e.freuder}@4c.ucc.ie

**Abstract.** The spread of constraint technology is inhibited by the lack of experienced constraint programmers. We present here a proposal for a tool that uses Case-Based Reasoning to store, retrieve and reuse constraint programming experience. We report our initial design ideas, and we sketch how the tool will operate in the domain of logic puzzles.

## 1 Introduction

Despite the broad applicability of constraint technology, constraint programming (CP) is currently restricted to highly-skilled modellers and programmers. Its wider uptake is hampered by a shortage of expert constraint programmers, a shortage caused in part by CP's learning curve and the wide range of skills required to build applications.

Our goal is to make CP technology more accessible by accelerating the accumulation of CP experience and making it more readily available to constraint programmers. We intend to build tools to support CP, and in this paper we propose one such tool. The tool will use Case-Based Reasoning (CBR) to store, retrieve and reuse constraint programming experience.

Section 2 looks at some of the problems faced by constraint programmers. Section 3 describes CBR and its application to programming tasks. Section 4 briefly describes our problem domain. In Section 5, we explain how our proposed system will operate.

## 2 Constraint Programming

The constraint programmer's main methodology for finding good models and solution strategies is empirical experimentation, akin to that done within academic research. The programmer may be assisted here by referring to papers that describe techniques that have been found to improve program quality and papers that describe case studies in specific domains, e.g. [4], [19], [15], [26], [11].

⋆ The Cork Constraint Computation Centre is supported by Science Foundation Ireland.

The ideas in these papers are gaining maturity and we might hope that these ideas and experiences will be collected together, much as has happened in the related but more mature field of Mathematical Programming [33].

An interesting, alternative way of spreading constraint technology is by shifting part of the CP burden from the programmer to the machine. One approach is to automatically generate constraint programs from abstract specifications. OPL Studio, for example, provides a high level modelling language for describing constraint models and can automatically utilise C++ library programs for execution [31]. The OPL language still for the most part requires the programmer to identify variables and values in the problem. However, for resource scheduling applications the modelling level is higher: the specification is in terms of concepts such as activities and resources.

In a similar vein, Constraint Lingo is a high-level specification language for so-called tabular constraint satisfaction problems (which include logic puzzles and certain classes of problems in graph theory such as graph colouring, finding independent sets and finding Hamiltonian cycles) [7]. A variety of compilers exists to translate Constraint Lingo specifications into constraint programs.

Ideas from research are being incorporated into the CGRASS system for the automatic transformation of naïve models to ones that may require less effort to solve [9]. CGRASS transformations simplify constraints. They have quite strong preconditions to limit transformation applicability to situations where benefits are likely to accrue.

Work that goes under the title of Constraint Acquisition is also of relevance. Constraint acquisition systems are typically interactive, seeking input from the user about her preferences, often asking the user to critique concrete solutions. The systems often incorporate machine learning techniques [22], [25], [23].

In summary, decisions in CP are not clear-cut, little advice is available, there is only very limited tool-support, and so it is left to the programmer to make the decisions on the basis of experience. Given that the decision-making is experience based, an approach based on Case-Based Reasoning might be appropriate.

## 3 Case-Based Reasoning

Case-Based Reasoning (CBR) is a problem-solving strategy based on reusing experience gained in previous problem-solving episodes [16]. It can be effective in domains in which problem types recur and similar problems have similar solutions [17]. In CBR, a new problem is not generally solved by reasoning 'from scratch'. Instead, it is solved by retrieving from a memory the solutions to similar previously-solved problems, and transferring and adapting these solutions to the new problem. The memory is referred to as a case base, and each problem-solving episode in the memory is known as a case.

Cases typically comprise at least two parts: a problem description and a description of a solution to that problem. Since we are describing both CBR and CP in this paper, the word 'solution' is ambiguous. It can refer either to the second part of a case or to the variable instantiations that solve a constraint satisfaction problem (CSP). In our work, these are not the same. To head off confusion, we have decided to abandon conventional CBR terminology. We will refer to the second part of a case as its ‘resolution’.

The CBR problem-solving cycle typically comprises four processes [1]:

**Retrieve:** A new problem is presented to the case base. The most similar case or cases are retrieved.

**Reuse:** The resolutions from the retrieved cases are reused to solve the new problem. This can involve adapting or combining the resolutions from the retrieved cases.

**Revise:** The new resolution formed in the previous process is tested, e.g., by applying it in the problem domain, by simulation or by subjecting it to expert scrutiny. It may be revised if it fails to adequately solve the new problem.

**Retain:** The new problem with its revised resolution are added to the case base as a new case. This means that CBR incorporates a form of learning: by retention of new cases, the problem-solver’s experience grows, which can increase its coverage or problem-solving accuracy.

All or some of the processes in the CBR cycle can be automated; see [18] for techniques and applications. As [18] makes clear, a variety of technologies has been used for each process. For example, the retrieval process might use simple $k$-nearest-neighbour techniques, *kd*-trees, case retrieval nets or any of a number of other technologies. The resolution adaptation carried out in the reuse process might be driven by formulae, rules or constraints. CBR systems have many successful commercial applications including diagnostic support to help-desks in call centres and product search in on-line shopping.

CBR and CP have been integrated in various ways before [28]. CP techniques have been particularly used to implement resolution adaptation, e.g. [24], [20]. An alternative view on much the same kind of system is that it is using exemplars from the case base to seed the search for a solution to a CSP, in the expectation that starting from an exemplar and adapting it might be more efficient than searching from scratch, e.g. [13]. A different integration of CP and CBR is where the CBR is used to deal with incompleteness and incorrectness in the constraint model. Here, the CBR system is invoked when the solution to the CSP fails to uniquely solve the value of a decision variable [27].

None of these previous ways of combining CP and CBR technology is quite what we have in mind in our work. Our goal is to support CP itself, rather than the process of finding solution(s) to particular CSPs. (Of course, these are not wholly separate since, if a system finds a solution to a CSP that violates user expectations, this might be one impetus for further constraint modelling.)

While CBR has not been previously used to support CP, it has been used within other forms of programming and software engineering in general. The rationale for using CBR in this way is the strong similarity between the CBR cycle and the kinds of processes involved in software reuse [29]. CBR can support software component reuse, especially of library classes in object-oriented programs [6], [30] and of code written in restricted programming languages [12], [14]. It has also been used at corporate level to support organisation learning in

A Round of Golf

When the Sunny Hills Country Club golf course isn't in use by club members, of course, it's open to the club's employees. Recently, Jack and three other workers at the golf course got together on their day off to play a round of eighteen holes of golf. Afterward, all four, including Mr. Green, went to the clubhouse to total their scorecards. Each man works at a different job (one is a short-order cook), and each shot a different score in the game. No one scored below 70 or above 85 strokes. From the clues below, can you discover each man's full name, job and golf score?

1. Bill, who is not the maintenance man, plays golf often and had the lowest score of the foursome.
2. Mr. Clubb, who isn't Paul, hit several balls into the woods and scored ten strokes more than the pro-shop clerk.
3. In some order, Frank and the caddy scored four and seven more strokes than Mr. Sands.
4. Mr. Carter thought his score of 78 was one of his better games, even though Frank's score was lower.
5. None of the four scored exactly 81 strokes.

**Fig. 1.** *A Round of Golf* Logic Puzzle

software development projects [2] where it can give a concrete realisation of the idea of an Experience Factory [3]. But we know of no work that uses CBR to support reuse in any form of declarative programming (although [10] describes the use of CBR for writing formal specifications).

## 4 Logic Puzzles

The system that we are building will help its user to write constraint programs. Initially the programs will be ones that solve Logic Puzzles. An example of a simple puzzle, called *A Round of Golf*[1], is given in Figure 1. Most people involved in CP are familiar with these puzzles, the best known example being variants of Lewis Carroll's Zebra Puzzle. They are a recreational pastime for many people and a good initial test domain for CP technology. Web sites are devoted to publishing such puzzles[2].

At one level, these puzzles are all highly similar. In [7], the puzzles are treated as instances of the class of tabular constraint satisfaction problems (tCSPs). In these problems, the solution has the structure of a table. The problem specifies attributes, with associated domains (data types), typically finite, that name the columns of the table. The problem also specifies the number of rows in the table

[1] By E.K.Rodehorst, Puzzle 9, Dell Favorite Logic Problems, Summer 2000. Available at `brownbuffalo.sourceforge.net`

[2] `crpuzzles.com/logic/`
`www.geocities.com/Heartland/Plains/4484/logic.htm`
`www.allstarpuzzles.com/logic/`

and it specifies constraints on the entries in the table. This similarity between puzzles is what licenses the development of high-level specification languages and compilers for translating from the specifications to constraint programs [8].

At another level, what is striking about these puzzles is how diverse they are. They differ from the Zebra Puzzle and from each other in many ways. They differ in the number of attributes, the cardinalities of the domains, whether an attribute's value must be different in each row of the table, and whether all domain values are used. Some have unique solutions; others (perhaps unintentionally) do not[3]. The types of clues are diverse: from the simple *"The Englishman lives in the red house"* (Zebra Puzzle) to *"The Bats' game drew as many fans as the Atoms' game and the second-worst-attended game together"* (Football Frenzy Puzzle[4]). The clues use a wide range of constructs: logical connectives (conjunction, disjunction), subset relationships, spatial relations (above, right of), ordering relations (immediately before), algebraic operators (sum), implied relationships, and both general and domain-specific associations.

While there are fairly standard ways of taking a puzzle and producing a constraint program (and these form the basis of the compilers mentioned above), because of the puzzle diversity, no single method will consistently produce the most efficient program across the range of puzzles. In [7], six different translations are applied to each of about 80 puzzles plus several other tCSPs. No translation consistently gave the most efficient constraint program.

At a more informal, anecdotal level, we have found much the same result. We have manually written constraint programs to solve many of these puzzles. These programs find solutions very quickly, mainly because of the small numbers of variables and possible values. But we have also managed to find some programs that are more efficient than others. Some of the differences between the programs are quite small and well-known (e.g. reordering variables, values and constraints, etc.). But others were more significant. We even found one puzzle[5] in which our best results were obtained by using a scheduling-type constraint model and using the search strategy of the ECLiPSe `edge_finder` library.

Such diversity makes this a more challenging domain for a system with the goal of assisting constraint programmers than one might initially suppose. The differing efficiencies of the different programs also show that the choice of a good translation is a matter of experimentation and experience. There is a huge space of options and scant 'theory' about the circumstances in which an option will reliably improve efficiency. It is for these reasons that we propose the use of CBR.

## 5 CBR-CP System

We propose the CBR-CP system to partially automate the process of writing programs to efficiently solve logic puzzles using CP experience contained in a case

[3] Clementine's Cleaners, by R.L.Whipkey, `crpuzzles.com/logic/logic0137.html`

[4] Dell Pencil Puzzles Vacation Special, Special Edition Fall 2001

[5] Pinocchio's Nose, by R.L.Whipkey, `crpuzzles.com/logic/logic0138.html`

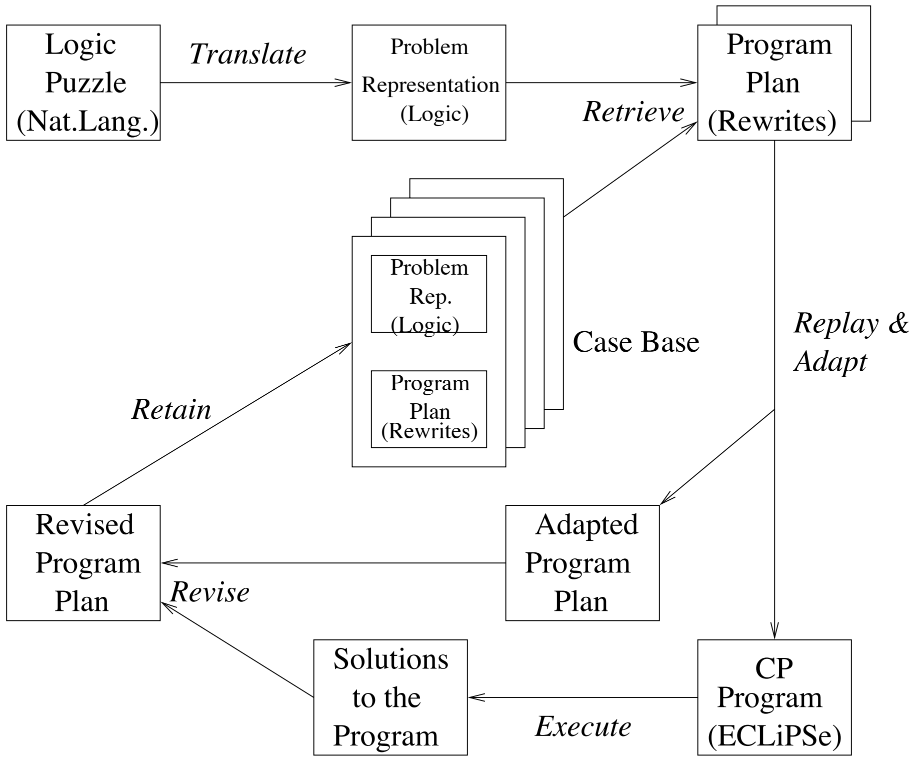

**Fig. 2.** Overview of the CBR-CP System

base. The process comprises a number of stages starting from the presentation of a new logic puzzle to the derivation of a program that solves the puzzle and the subsequent update of the case base. The stages are shown in Figure 2. Our preliminary design ideas for each of the stages are described below and are illustrated using the puzzle shown in Figure 1.

## 5.1 Translation from Natural Language to Problem Representation

The first stage of the process is to translate the puzzle from natural language to an unambiguous formal representation. The amount of interpretation required in this stage is well beyond the capabilities of current natural language understanding systems, and so we translate the puzzles manually.

There are many alternative ways of representing these puzzles unambiguously. We have chosen a representation that comprises two parts; viewing the puzzle as a tCSP, the first part defines the table schema; the second part constrains the table contents by translating the puzzle clues into constraints. (Adoption of Constraint Lingo [7], [8] would give us a higher-level way of writing the

```
% schema(TableName, Columns, NumOfRows)
schema(player, [firstname, surname, occ, score], 4),
% attribute(Name, Values)
attribute(firstname,[jack, bill, paul, frank]),
attribute(surname, [green, clubb, sands, carter]),
attribute(occ, [maint, clerk, caddy, cook]),
attribute(score, [70..85]),
% attribute(Name, AllDifferent?, AllUsed?, Numeric?, Cardinality)
attribute(firstname, yes, yes, no, 4),
attribute(surname, yes, yes, no, 4),
attribute(occ, yes, yes, no, 4),
attribute(score, yes, no, yes, 16),

% Clue 1
player(bill, _, BillOcc, _),
BillOcc #\= maint,
% etc!
```

**Fig. 3.** *A Round of Golf* Problem Representation

constraints. This is something we will consider in the future. There is a concern that the Constraint Lingo notation is not yet expressive enough.)

The problem representation of *A Round of Golf* is given in Figure 3. The `schema/3` predicate defines the table by specifying its name, the names of its columns in order, and the number of rows in the table. The `attribute/2` predicate is used to identify the puzzle's attributes (column names) and their domains. The `attribute/5` predicate gives further information about attributes, some of which can be deduced from the `attribute/2` and `schema/3` information, but we show it explicitly here. The reason for splitting attribute information using `attribute/2` and `attribute/5` is that the latter plays a role in case retrieval (below), whereas the former does not.

`AllDifferent?` is `yes` if each row must use a different value. `AllUsed?` is `yes` if the cardinality of the domain is less than or equal to that of the table. `Numeric?` is `yes` if the values are numeric. `Cardinality` is the size of the domain. We may yet want to record further information about attributes (e.g. summary information about the attributes' constraints).

The second part of the problem representation expresses the clues in the puzzle. Firstly, row 'patterns' are specified, containing constants and variables. Then relations between the variables and other constants are written. For example, the following

```
player(bill, _, BillOcc, _),
BillOcc #\= maint,
```

states that in any row of the table where the first name is `bill`, then the `occ` attribute does not take the value `maint`, i.e. Bill is not the maintenance man.

The Great Flying Contest

Problem Representation

```
schema(flight, [town, dist, boy, girl], 5),
attribute(town, [springs, pisgah, linda, holly, corners]),
attribute(dist, [6..20]),
attribute(boy, [daniel, jeff, tony, brent, peter]),
attribute(girl, [carrie, alexis, susan, leah, mia]),
attribute(town, yes, yes, no, 5),
attribute(dist, yes, no, yes, 15),
attribute(boy, yes, yes, no, 5),
attribute(girl, yes, yes, no, 5),

flight(_, AlexisDist, _, alexis),
flight(_, DanielDist, daniel, _),
AlexisDist #= DanielDist + 6,
% etc!
```

Program Plan

*A set of rewrites for converting the problem representation into a constraint program.*

**Fig. 4.** *The Great Flying Contest* Case

A higher-level representation of the clues might be possible (e.g. using the Constraint Lingo vocabulary), but our approach has an attractive consistency.

## 5.2 The Case Base

An (incomplete) example of a case is given as Figure 4. This case pertains to a puzzle called *The Great Flying Contest*[6]. We will not show the original natural language puzzle. Suffice it to say that it involves pairs of boys and girls whose model planes reach towns at various distances from their launch point. We have chosen this puzzle because it is similar to *A Round of Golf*. If we think of *A Round of Golf* as the puzzle for which we seek a constraint program, then *The Great Flying Contest* might be the case (or one of the cases) that we retrieve from the case base with a view to reusing the programming experience it contains.

As can be seen from Figure 4, in our system, the problem component of a case is a formal problem representation, i.e. the kind of representation that we were describing in the previous section and that we exemplified in Figure 3. The resolution component of a case is a set of rewrites which, applied to the problem representation will yield a constraint program. We will refer to this set of rewrites as a Program Plan. We will discuss these rewrites after we have discussed the kind of constraint program that we want to write for *The Great Flying Contest*.

[6] By R.L.Whipkey, `crpuzzles.com/logic/logic0132.html`

```
:- lib(fd).

start(Vars) :-
    TOWN = [Xsprings, Xpisgah, Xlinda, Xholly, Xcorners],
    BOY  = [Xdaniel, Xjeff, Xtony, Xbrent, Xpeter],
    GIRL = [Xcarrie, Xalexis, Xsusan, Xleah, Xmia],
    Vars = [Xsprings, Xpisgah, Xlinda, Xholly, Xcorners,
            Xdaniel, Xjeff, Xtony, Xbrent, Xpeter,
            Xcarrie, Xalexis, Xsusan, Xleah, Xmia],
    TOWN :: 6..20,
    BOY  :: 6..20,
    GIRL :: 6..20,

    alldistinct(TOWN),
    alldistinct(BOY),
    alldistinct(GIRL),

    Xalexis #= Xdaniel + 6,
    % etc!

    labeling(Vars).
```

**Fig. 5.** One constraint program for *The Great Flying Contest*

One of the features that makes *A Round of Golf* and *The Great Flying Contest* more similar to each other than they are to, say, the standard Zebra Puzzle is the existence of an attribute for which `AllDifferent?` is `yes` but `AllUsed?` is `no`. For *A Round of Golf*, this attribute is the players' scores; for *The Great Flying Contest*, it is the flight distances. (For both attributes, `Numeric?` is also `yes`. This increases similarity, but it may be incidental to predictions about the style of constraint program that best suits puzzles like these.) The possibility that such attributes might feature in puzzles is rarely appreciated by those whose exposure is confined to the Zebra Puzzle. It is one element of diversity, and it influences the efficiency of the different constraint programs that can be written to solve the puzzles.

There are many styles of constraint program that will solve *The Great Flying Contest*. We will discuss three of them. (In this discussion, we are ignoring fine details such as variable, value and constraint ordering, etc.) Consider first the two programs in Figures 5 and 6.

Figure 5 is the style of program that might conventionally be used to solve logic puzzles. One attribute of the tCSP is chosen to supply the values of the CSP (in this case, the distances flown); the rest supply the variables.

Such a program is easily written for puzzles such as the Zebra Puzzle. But by abbreviating the program in Figure 5, there is something that we have not yet revealed. For puzzles such as *The Great Flying Contest* (and *A Round of Golf*), in which there is an attribute for which `AllUsed?` is `no`, there is a need

```
:- lib(fd).
start(Schema):-
   Schema = [
      flight(A_town, A_dist, A_boy, A_girl),
      flight(B_town, B_dist, B_boy, B_girl),
      % etc!
   ],
   [A_boy, B_boy, C_boy, D_boy, E_boy] :: [daniel, jeff, tony, brent, peter],
   [A_girl, B_girl, C_girl, D_girl, E_girl] :: [carrie, alexis, susan, leah, mia],
   % etc!
   alldistinct([A_town, B_town, C_town, D_town, E_town]),
   alldistinct([A_boy, B_boy, C_boy, D_boy, E_boy]),
   % etc!
   CoOccurrences = [
      flight(_, AlexisDist, _, alexis),
      flight(_, DanielDist, daniel, _),
      % etc!
   ],
   AlexisDist #= DanielDist + 6,
   % etc!
   % Apply constraints to schema.
   constrain(CoOccurrences, Schema).

constrain([],_).
constrain([H|T], List) :-
  member(H, List),
  constrain(T, List).
```

**Fig. 6.** Another constraint program for *The Great Flying Contest*

to place some quite complicated constraints. Specifically, these constraints must capture the relationship between the five flights and fifteen possible distances. We have to ensure that whichever of these fifteen flight distances is chosen for, e.g., variable Xsprings, then the same flight distance will be chosen for the boy and girl whose plane flew to Springs. In general, we have to express the constraint that if one variable gets a value, then some other variables must receive the same value (even though more values than needed are available). Since we are not sure which these other variables are, the constraint involves an amount of disjunction. Because of the disjunction, this style of program, which is an efficient one (even possibly the best style) for the Zebra Puzzle, is less efficient (and was, in fact, the least efficient of the programs we tried) for *The Great Flying Contest*.

Figure 6 shows a different style of program.[7] The program sets up a list, called Schema, which has a position for each entry in the tCSP, and it sets up a second structure, called CoOccurrences, which is used to constrain Schema: each

[7] This is the dominant style of program to be found at brownbuffalo.sourceforge.net/

tuple in `CoOccurrences` must be a member (although not necessarily a distinct member) of `Schema`. Now, the requirement that the distance selected from the fifteen values and associated with a town must also be associated with the boy and girl whose plane flew to that town no longer needs to be expressed using a disjunctive constraint: it is captured through the list `Schema`.

This style of program is not the most efficient style for the Zebra Puzzle. But, to our surprise, because it avoids the disjunctive constraints, it is the more efficient of the two styles for *The Great Flying Contest*.

Furthermore, for *The Great Flying Contest*, we found that a hybrid of the two styles of program was even more efficient than either style alone. The hybrid, which we do not show, expresses most of the constraints in the style of Figure 5. But it does not include the disjunctive constraints mentioned above. Instead, it uses `Schema` and `CoOccurrences` lists for this part of the problem. Admittedly, the hybrid is not *much* more efficient than the program in Figure 6 (about a dozen fewer backtracks), but these are small CSPs.

This is the kind of experience that we want to capture in a case base. So, in the *Great Flying Contest* case, we want to associate the formal representation of the puzzle with a program plan for producing the hybrid constraint program. On the other hand, a case for the Zebra Puzzle would include a program plan that produces a program in the style of Figure 5.

Figure 4 does not show the details of the case's program plan. The program plan translates the problem representation into the hybrid constraint program that we have been discussing. The set of rewrites that constitute this compiler is too large and detailed to show in the figure. Readers can get an idea of the kind of mappings it has to perform by comparing the source (i.e. the problem representation in Figure 4) with the target (i.e. a hybrid of the programs in Figures 5 and 6).

There is a difference between our rewrites and the transformations found in other systems. In the CGRASS system, for example, the transformation operators are universally quantified. Any constraint which matches an operator's preconditions can be transformed in the way specified by the operator's add- and delete-lists. (There is no guarantee that the resulting program will run more quickly, although the preconditions are sometimes stronger than necessary in order to try to restrict operator applicability to circumstances when an improvement in efficiency is likely. In a sense, we are trying to characterise these circumstances in a case-based way.) By contrast, the rewrites in our system, while they may recur across different cases, will be case-specific. Different cases will have different sets of rewrites. They simply record the actions that an expert programmer carried out on this problem representation to produce a constraint program.

### 5.3 Retrieval from the Case Base

Rarely will the problem representation of the new puzzle exactly match any of the problem representations in the case base. Instead, we will try to retrieve the cases that have the most similar problem representations.

We intend using a similarity measure that is based on graph edit operations [5]. This requires us to view each representation as a graph and to define a number of primitive graph edit operations, each with an associated cost. Then, for each case in the case base, we search for the minimal cost sequence of edits that will transform the new problem representation to one that is equal to the case problem representation. The cases with lowest minimal costs are the ones that are the most similar to the new problem.

Of course, finding the minimal cost edit sequence is, in general, NP-complete. We have to ensure practical efficiency. Our intention is to sort both the new problem representation and the case problem representations and use an essentially greedy algorithm. It is quite common to further improve retrieval in Case-Based Reasoning by doing it in two steps [16]: a fetch using indexing features followed by a more detailed selection done by matching case details.

In the first step, relevant cases are fetched by matching features of the new problem with indexes into the case base. The features that are used for indexing should be predictive of matches using the similarity measure. We believe that a summary of the `attribute/5` information will be suitable. Thus the indexes we are contemplating are all Boolean-valued: whether there are any attributes in which values in the table will not be required to be all different, whether there are any attributes in which not all values are used, and whether there are any numeric attributes or not.

Having fetched a set of candidate cases using the indexes, we will carry out the more detailed structural match (using greedy graph edit distance) on just these candidates. The candidate(s) with the highest matches, will be passed to the next stage of the process.

## 5.4 Adaptation of the Selected Cases

The cases selected in the previous stage of the process contain program plans, but if these plans are to be reused, they must be adapted to fit the new problem. There are many ways of adapting cases [16]. In substitutional adaptation and transformational adaptation, for example, rules, formulae or constraints are used to make small changes to case resolutions, predicated on differences between the new problem and the retrieved case. This simple form of adaptation is not suitable for our purposes.

Of more interest is a form of adaptation known variously as derivational adaptation or generative adaptation [16], [18]. This form of adaptation is suitable when the case resolution takes the form of a trace of the reasoning that was used to solve the problem represented in the case. For example, in a case-based planning system, the resolution would be the decisions that the planner took. This, of course, is analogous to our own design: our case resolutions take the form of program plans, i.e. sets of rewrites.

The idea is to replay the decisions contained in the retrieved program plan [32]. The conditions of each action in the trace are tested in the new problem and, if they are true, the action is executed again.

A fall-back is needed to deal with any decisions that cannot be made through case replay. One possibility is to consult further similar cases [32]. We can also fall back on generic ways of compiling tCSPs into constraint programs. (We are working on the idea of defining an ordering relation on cases in the case base. The program plans in children cases will inherit their parents' program plans, and so need include only incremental rewrites that are specific to that child. If what is inherited produces an appropriate style of constraint program, then the incremental rewrites might simply cover such things as how to order variables, values and constraints, insertion of redundant constraints, etc.)

### 5.5 Execution, Revision and Retention

The constraint program can now be executed and the solution(s) displayed to the user. The user can judge the solution(s) that the program finds. If she feels that the constraint program fails to solve the puzzle, then she can manually revise the program. If the program successfully solves the puzzle, then it is a candidate for insertion into the case base.

## 6 Conclusions

We have sketched a simple case-based system that could support the constraint programmer. We are using logic puzzles as a relatively simple test domain. Once the prototype is implemented and tested, there are many avenues for further exploration. We mention three of the more important ones.

We have been describing a largely autonomous system. The user supplies a translation of the logic puzzle and then has no role to play until a constraint program is produced. We will aim to make subsequent versions of the system more interactive and, in particular, to accommodate mixed-initiative interaction.

We are also interested in the ideas contained in [21] where a case-based planner is used to plan laboratory experiments. We think it would be interesting to build a case-based CP system that planned and executed experiments, e.g. to compare the efficiency of candidate programs. We would hope to record negative as well as positive outcomes (i.e. experiments that worsened efficiency, as well as ones that improved it), to reuse both kinds of experience on unseen problems.

We intend also to test the work in more realistic domains in which we have CP expertise, most notably scheduling.

## References

1. Aamodt, A. & E. Plaza: Case-Based Reasoning: Foundational Issues, Methodological Variations, and System Approaches, *AI Communications*, vol.7(1), pp.39–59, 1994.
2. Althoff, K.-D., A. Birk, C.G. von Wangenheim & C. Tautz: CBR for Experimental Software Engineering, in [18], pp.235-254, 1998.

3. Basili, V.R., G. Caldiera & H.D. Rombach: Experience Factory, in J.J. Marciniak, *Encyclopedia of Software Engineering*, vol.1, pp.469-476, Wiley, 1994.
4. Borrett, J.E. & E.P.K. Tsang: A Context for Constraint Satisfaction Problem Formulation Selection, *Constraints*, vol.6(4), pp.299–327, 2001.
5. Bunke, H. & B.T. Messmer: Similarity Measures for Structured Representations, in S.Wess, K.-D.Althoff & M.Richter (eds.), *Procs. of the First European Workshop on Case-Based Reasoning*, LNAI 837, pp.106–118, Springer, 1993.
6. Fernández-Chamizo, C., P.A. González-Calero, M. Gómez-Albarrán, L. Hernández-Yáñez: Supporting Object reuse Through Case-Based Reasoning, in I.Smith & B.Faltings (eds.), *Procs. of the Third European Workshop on Case-Based Reasoning*, LNAI 1168, pp.135–149, Springer, 1996.
7. Finkel, R., V.W. Marek & M. Truszczyński: Tabular Constraint-Satisfaction Problems and Answer-Set Programming, *Procs. of the AAAI Symposium on Answer Set Programming*, 2001.
8. Finkel, R., V.W. Marek & M. Truszczyński: *Constraint Lingo: Towards High-Level Answer-Set Programming*, Unpublished manuscript.
9. Frisch, A.M., I. Miguel & T. Walsh: CGRASS: A System for Transforming Constraint Satisfaction Problems, in K.Apt, F.Fages, E.C.Freuder, B.O'Sullivan, F.Rossi & T.Walsh (eds.), *Workshop on Constraint Solving and Constraint Logic Programming*, University College Cork, pp.23–36, 2002.
10. Funk, P.J. & D. Robertson: Case-Based Support for the Design of Dynamic System Requirements, in J.-P.Haton, M.Keane & M.Manago (eds.), *Procs. of the Second European Workshop on Case-Based Reasoning*, LNAI 984, pp.211–225, Springer, 1994.
11. Gent, I.P., R.W. Irving, D.F. Manlove, P. Prosser & B.M. Smith: A Constraint Programming Approach to the Stable Marriage Problem, in T.Walsh (ed.), *Principles and Practice of Constraint Programming, Procs. of CP 2001*, LNCS 2239, pp.225–239, Springer, 2001.
12. Gomes, P. & C. Bento: Automatic Conversion of VHDL Programs into Cases, in S.Schmitt & I.Vollrath (eds.), *Procs. of the Workshop Programme for the Third International Conference on Case-Based Reasoning*, pp.II-27–II-36, 1999.
13. Huang, Y. & R. Miles: A Case Based Method for Solving Relatively Stable Dynamic Constraint Satisfaction Problems, in M.Veloso & A.Aamodt (eds.), *Procs. of the First International Conference on Case-Based Reasoning*, LNAI 1010, pp.481–490, Springer, 1995.
14. Job, D., V. Shankararaman & J. Miller: Hybrid AI Techniques for Automated Software Reuse, in S.Schmitt & I.Vollrath (eds.), *Procs. of the Workshop Programme for the Third International Conference on Case-Based Reasoning*, pp.IV-3–IV-12, 1999.
15. Kilby, P., P. Prosser & P. Shaw: A Comparison of Traditional and Constraint-Based Heuristic Methods on Vehicle Routing Problems with Side Constraints, *Constraints*, vol.5(4), pp.389–414, 2000.
16. Kolodner, J.: *Case-Based Reasoning*, Morgan Kaufmann, 1993.
17. Leake, D.B.: CBR in Context: The Present and Future, in D.B.Leake (ed.), *Case-Based Reasoning — Experiences, Lessons & Future Directions*, pp.3-30, AAAI Press, 1996.
18. Lenz, M., B. Bartsch-Spörl, H.-D. Burkhard & S. Wess (eds.): *Case-Based Reasoning Technology*, LNAI 1400, Springer, 1998.
19. Nadel, B.A.: Representation Selection for Constraint Satisfaction: A Case Study Using $n$-Queens, *IEEE Expert*, vol.5, pp.16–23, 1990.

20. Neagu, N. & B. Faltings: Exploiting Interchangeabilities for Case Adaptation, in D.W.Aha & I.Watson (eds), *Procs. of the Fourth International Conference on Case-Based Reasoning*, LNAI 2080, pp.422–436, Springer, 2001.
21. Oehlmann, R.: Investigative Actions: Case-Based Planning of Laboratory Experiments, in I.D.Watson(ed.), *Procs. of the Second United Kingdom Workshop on Case-Based Reasoning*, pp.106–119, 1996.
22. O'Sullivan, B., E.C. Freuder & S. O'Connell: Interactive Constraint Acquisition, in *Procs. of the Workshop in User-Interaction in Constraint Satisfaction, Workshop Proceedings at ICLP'01, (International Conferences on Constraint programming and Logic Programming)*, 2001.
23. Padmanabhuni, S., J.-H. You & A. Ghose: A Framework for Learning Constraints, in *Procs. of Workshop on Induction of Complex Representations, Workshop Programme at PRICAI-96 (Fourth Pacific Rim International Conference on Artificial Intelligence)*, 1996.
24. Purvis, L. & P. Pu: Adaptation Using Constraint Satisfaction Techniques, in M.Veloso & A.Aamodt (eds.), *Procs. of the First International Conference on Case Based Reasoning*, LNAI 101, pp.289–300, Springer, 1995.
25. Rossi, F. & A. Sperduti: Acquiring both Global and Local Preferences in Interactive Constraint Solving via Machine Learning Techniques, in *Procs. of the Workshop in User-Interaction in Constraint Satisfaction, Workshop Proceedings at ICLP'01, (International Conferences on Constraint programming and Logic Programming*, 2001.
26. Smith, B.M.: Dual Models of Permutation Problems, in T.Walsh (ed.), *Principles and Practice of Constraint Programming, Procs. of CP 2001*, LNCS 2239, pp.615–619, Springer, 2001.
27. Sqalli, M.H. & E.C. Freuder: CBR Support for CSP Modeling of InterOperability Testing, in *Procs. of the AAAI-98 Workshop on Case-Based Reasoning Integrations*, pp.155–160, 1998.
28. Sqalli, M.H., L. Purvis & E.C. Freuder: Survey of Applications Integrating Constraint Satisfaction and Case-Based Reasoning, in *Procs. of the First International Conference and Exhibition on The Practical Application of Constraint Technologies and Logic Programming*, 1999.
29. Tautz, C. & K.-D. Althoff: Using Case-Based Reasoning for Reusing Software Knowledge, in D.B.Leake & E. Plaza (eds.), *Procs. of the Second International Conference on Case-Based Reasoning*, LNAI 1266, pp.156–165, Springer, 1997.
30. Tessem, B., A. Whitehurst & C.L. Powell: Retrieval of Java Classes for Case-Based Reuse, in B.Smyth & P.Cunningham (eds.), *Procs. of the Fourth European Workshop on Case-Based Reasoning*, LNAI 1488, pp.148–159, Springer, 1998.
31. van Hentenryck, P.: *The OPL Optimisation Programming Language*, MIT Press, 1999.
32. Veloso, M.M.: Flexible Strategy Learning Using Analogical Replay of Problem Solving Episodes, in D.B.Leake (ed.), *Case-Based Reasoning — Experiences, Lessons & Future Directions*, pp.137–150, MIT Press, 1996.
33. Williams, H.P.: *Model Building in Mathematical Programming*, Wiley, 1999.