Using CBR to Select Solution Strategies in Constraint Programming*

Cormac Gebruers[1], Brahim Hnich[1], Derek Bridge[2], and Eugene Freuder[1]

[1] Cork Constraint Computation Centre, University College Cork, Cork, Ireland
{c.gebruers,b.hnich,e.freuder}@4c.ucc.ie
[2] Department of Computer Science, University College Cork, Cork, Ireland
d.bridge@cs.ucc.ie

Abstract. Constraint programming is a powerful paradigm that offers many different strategies for solving problems. Choosing a good strategy is difficult; choosing a poor strategy wastes resources and may result in a problem going unsolved. We show how Case-Based Reasoning can be used to select good strategies. We design experiments which demonstrate that, on two problems with quite different characteristics, CBR can outperform four other strategy selection techniques.

1 Introduction

Organisations, from factories to universities, must daily solve hard combinatorial problems. *Constraint programs*, which reason with declaratively-stated hard and soft constraints, are one of the most expressive, flexible and efficient weapons in the arsenal of techniques for automatically solving these hard combinatorial problems. They have been successfully employed in many real-life application areas such as production planning, staff scheduling, resource allocation, circuit design, option trading, and DNA sequencing [21].

Despite the broad applicability of constraint programs, constraint programming is a skill currently confined to a small number of highly-experienced experts. For each problem instance, a constraint programmer must choose an appropriate *solution strategy* (see Sect. 2). A poor choice of solution strategy wastes computational resources and often prevents many or all problem instances from being solved in reasonable time. The difficulty of choosing a good solution strategy is compounded by the growing number of strategies. Our understanding of when it is appropriate to use a strategy has not kept pace. Improvements in the quality of decision-making would have considerable economic impact.

In this paper, we use decision technologies to support the choice of solution strategy. That is, for a given problem, such as the Social Golfer Problem (defined later), we try to predict good solution strategies for instances of that problem which differ in size, constraint density, and so on. In Sect. 2, we introduce constraint programming and define what we mean by a solution strategy. Sect. 3

* This material is based upon work supported by Science Foundation Ireland under Grant No. 00/PI.1/C075.

X	D	C
x_1	$\{1, 2, 3\}$	$x_1 \leq x_2$
x_2	$\{2, 3, 4\}$	

Fig. 1. A simple CSP

shows how CBR and decision trees can be used to select solution strategies; we also define three benchmark approaches. Sect. 4 explains our experimental methodology. Experiments are reported in Sect. 5.

2 Constraint Programming

A solution strategy S comprises a model M, an algorithm A, a variable ordering heuristic V_{var} and a value ordering heuristic V_{val}: $S =_{\text{def}} \langle M, A, V_{var}, V_{val} \rangle$ [1]. We will look at each component in turn.

2.1 Models

The task of modelling is to take a problem and express it as a Constraint Satisfaction Problem (CSP). We define a CSP to be a triple $\langle X, D, C \rangle$. X is a finite set of *variables*. D is a function that associates each $x \in X$ with its *domain*, this being the finite, non-empty set of values that x can assume. C is a finite set of *constraints* which restrict the values that the variables can simultaneously assume. A simple example is given in Fig. 1. A solution to a CSP is an assignment of values to variables such that every variable has exactly one value from its domain assigned to it and all the constraints are satisfied. For example, $\{x_1 = 1, x_2 = 3\}$ is one solution to the CSP in Fig. 1.

There are often multiple ways of taking an informally-expressed problem and expressing it as a CSP. We refer to each formulation as a model. To exemplify, we consider an example problem known as the Social Golfers Problem:

> *"The coordinator of a local golf club has come to you with the following problem. In her club, there are 32 social golfers, each of whom play golf once a week, and always in groups of 4. She would like you to come up with a schedule of play for these golfers, to last as many weeks as possible, such that no golfer plays in the same group as any other golfer on more than one occasion."* Problem 10 in [6]

The problem is generalised with its instances being described by four parameters $\langle w, m, n, t \rangle$: the task is to schedule $t = m \times n$ golfers into m groups each of n golfers over w weeks so that no golfer plays any other golfer more than once. The Social Golfer Problem has elements in common with many real-world scheduling problems. A factory needing a daily schedule might need to solve different instances of such a problem (i.e. with different parameter values) every 24 hours. Savings in solution time could be of considerable value.

We will briefly describe three possible models for the Social Golfers Problem. In the *set-based* model [15], there are $m \times w$ variables. Each variable MW_i^j represents the ith group of players in the jth week. Hence, the values these variables take on are sets, each containing n player identifiers.

In the *integer total-golfer* model [18], there are $t \times w$ variables. Each variable TW_i^j represents the ith golfer in the jth week. Hence, the values these variables take on are integers between 1 and m, identifying which of the m groups golfer i belongs to in week j. In fact, this model is not used in practice, because no efficient way has been found of expressing the constraint that groups from different weeks share at most one golfer.

However, even the integer total-golfer model does have a use, because there is a practical model which combines the set-based model with the (incomplete) integer total-golfer model [2]. Combining models is a commonplace and productive technique in constraint programming. The combined model contains both the $m \times w$ set-valued variables and the $t \times w$ integer-valued variables. Additional constraints, known as *channelling constraints*, ensure that, when a golfer is assigned to a group in the integer total-golfer model, the set-based model makes that golfer a member of the appropriate set-valued variable, and vice-versa.

2.2 Algorithms

Solving a CSP involves *search*. Each variable is assigned a value in turn and the legality of that assignment is tested. If any constraints involving that variable have been violated, the value is retracted and another tried in its place. Suppose, for the CSP in Fig. 1, that $x_1 = 3$; if we now try to assign $x_2 = 2$, we violate the constraint $x_1 \leq x_2$. Hence, we would backtrack: we would retract $x_2 = 2$ and try, e.g., $x_2 = 3$ instead.

When, during search, a legal assignment has been made, certain values in the domains of *other* variables may no longer be able to participate in the solution under construction. These unsupported values can be removed from further consideration, a process known as *propagation*. In Fig. 1, suppose we assign $x_1 = 3$; then 2 can be removed from the domain of x_2: it cannot participate in any solution where $x_1 = 3$ as it would violate the constraint $x_1 \leq x_2$. The constraint programming community has devised numerous algorithms that give different trade-offs between the relative degrees of search and propagation

2.3 Variable and Value Ordering Heuristics

The efficiency of search and propagation may be influenced significantly by the order in which variables are instantiated and the order in which values are chosen. The constraint programming literature lists numerous heuristics for guiding these choices; see, e.g., [20].

2.4 Strategy Selection

We have presented above the four components of a solution strategy, $S =_{\text{def}} \langle M, A, V_{var}, V_{val} \rangle$. Defining solution strategies in this modular way is convenient

but may be misleading. It is not meant to imply that each component of a strategy can be chosen independently of the other components. Some components may be incompatible with others; and good performance from a strategy will require that the chosen components perform as a cohesive unit. Hence, we do not treat strategy selection as four independent decisions, nor four cascaded decisions. Instead, we treat each solution strategy, of which there are many, as if it were an atomic entity. Strategy selection is then a single decision: choosing, if possible, the best strategy from this large set of atomic strategies.

2.5 Related Work

CBR has previously been used to support software development tasks. There is work, for example, on design reuse and code reuse, of which [7] and [8] are representative. More in the spirit of the work we report in this paper, however, is the use of CBR to choose data structures for storing matrices in scientific problem-solving environments [22].

While many synergies between constraint technology and CBR have been reported (with a review in [19]), the only work we know of in which CBR is used to make constraint programming decisions is our own. In [12], we use CBR to choose models for logic puzzles; in [5], we use CBR to choose between integer linear programming and constraint programs for bid evaluation problems.

There is related work that does not use CBR. For example, Borret and Tsang develop a framework for systematic model selection [3], building on Nadel's theoretical work [14]. Minton dynamically constructs constraint programs by performing an incomplete search of the space of possible programs [13]. The contrast between his work and ours is that we seek to re-use existing strategies, rather than construct new programs.

Rather different again is the system reported in [4], which executes multiple strategies, gathers information *at runtime* about their relative performance and decides which strategies to continue with. The focus in that system is domains where optimisation is the primary objective, rather than constraint satisfaction.

Finally, we note that, outside of constraint programming, machine learning has been used in algorithm selection tasks, e.g. sort algorithm selection [9].

3 Strategy Selection Techniques

We describe here how we have applied CBR and decision trees to strategy selection. We also describe three benchmark approaches.

3.1 Case-Based Reasoning

Each case represents one problem instance. The 'description' part of a case is a feature vector that characterises the instance (see below for a discussion of the features). The 'solution' part of a case identifies the solution strategies that have performed well on this instance. This needs a little more explanation.

Our decision to treat a solution strategy, although it is made up of four components, as an atomic entity means that the 'solution' part of a case needs only contain solution strategy labels. Thus, our task has become one of case-based classification. In fact, as we will explain in detail in Sect. 4.2, it may be appropriate to regard more than one solution strategy as appropriate for a particular instance. Hence, the 'solution' part of a case is a *set* of solution strategy labels. In summary, each case $\langle \mathbf{x}, S \rangle$ comprises feature vector $\mathbf{x} = \langle v_1, \ldots, v_i \rangle$ and a set S of strategy identifiers.

With strategy selection reduced to a classification task, simple CBR techniques suffice. We retrieve the k-nearest neighbours (we use $k = 3$) and we use majority voting to to choose a strategy.

It remains to discuss the features we use. For three reasons, we have chosen to use *surface features* in our work to-date:

- The over-riding reason is that, as a matter of good methodology, we need to discover just how predictive surface features are before turning to other approaches.
- A lesser reason is that, anecdotally, surface features (if anything) would appear to be what human programmers use when selecting an initial strategy.
- Finally, surface features are cheap-to-extract and cheap-to-compare in the similarity measure. By contrast, the main alternative is to compare the constraint graphs of problem instances. For reasons of computational complexity, this is to be avoided, if possible.

The features we use might also be described as *static features*: they can be obtained *prior* to execution; an example is the constraint density of the problem instance. We are not using *dynamic features*, that are only obtainable during execution, e.g. the number of backtracks at a certain point.

Finally, it has turned out that all our features are numeric, and so we compute similarity as the inverse of Euclidean distance with range normalisation [23].

However, prior to using case bases for strategy selection, we use the *Wrapper method* to select a predictive subset of the features [11] and these are the ones used in the CBR.

3.2 Decision Trees

The decision trees we use are induced by C4.5 [17] from the same problem instances that make up the case bases in our CBR approach. The tests that label the interior nodes of the trees are drawn from the same features as used in the CBR systems. Leaves are labelled by solution strategies. We use C4.5 with all its default settings, also allowing it to prune the trees to avoid over-fitting.

3.3 Benchmark Approaches

We have used three benchmark approaches for strategy selection:

Random: A strategy is selected randomly, with equal probability, from among the candidates.

Weighted Random: A strategy is selected randomly, but the probability that a candidate is selected is proportional to how often that strategy is a winning strategy in the dataset.

Use Best: In this approach, the same strategy is selected every time: the one that is a winner most often in the dataset.

4 Experimental Methodology

4.1 Candidate Strategies

As Sect. 2 shows, for any given problem instance, there is a vast number of possible strategies, combining different models, algorithms and heuristics. In practice, human programmers entertain very few strategies. Similarly, in our experiments it is not feasible to choose among all possible strategies. Instead, we use around ten candidate strategies. Lest we be accused of thereby making the prediction task too easy, we use candidates that informal experimentation shows to be competitive on the different problem instances and which give, as much as possible, a uniform distribution of winners because this maximises the difficulty of strategy selection.

4.2 Winning Strategies

We have to define what it means for a candidate strategy to be a winner on a problem instance. Surprisingly, it is not easy to obtain a consensus within the constraint programming community on this.

To exemplify this, suppose the execution times of two strategies s_1 and s_2 on a problem instance are 1000ms and 990ms respectively. While s_2 is the winner, some might argue that s_2 exhibits no material advantage: the difference is 10ms, which is only 1% of the faster execution time. Similarly, if s_3 takes 505000ms and s_4 takes 500000ms, s_4 is the winner; but in percentage terms the difference between them is also 1%, the same as that between s_2 and s_1. In some domains, where time is critical, any advantage may be worth having; in other domains, performance within, e.g., an order-of-magnitude of the fastest strategy may be regarded as acceptable. In the latter case, if a strategy selection technique were to pick any of the high-performing strategies, it could be regarded as having made a correct choice.

Our resolution to this lack of consensus is to use different definitions of *winner*: we parameterise the definition of winning strategy and plot results for different parameter values. We define a winning strategy using a window of execution time. The best execution time recorded for an instance constitutes the window's lower bound. The upper bound is determined by a multiplication factor. We denote different winning strategy definitions by their multiplication factor, e.g. $\times 1.0$, $\times 10.0$, etc. If a strategy's execution times falls within the window, it is considered one of the joint winners.

4.3 Dataset Generation

For each problem, we generate a dataset of problem instances. We need to label each instance with its set of winning strategies. So we solve each instance with each of the candidate strategies in turn and record the execution times. Since some strategies perform unreasonably poorly on certain instances, execution is done subject to a timeout of 6000ms. We do not admit into a dataset instances where all strategies time out and instances where all strategies are joint winners. These instances are of no use in prediction experiments.

4.4 Evaluation

The dataset is randomly partitioned into a training set and a test set, where the training set is 60% of the instances. For each instance in the test set, we use CBR, decision trees and the benchmark approaches to predict a solution strategy for that instance. We determine, in each case, whether the prediction is one of the winning strategies. Results are subject to 10-fold cross-validation.

We report the following results:

Prediction Rate: This is the number of times a strategy selection technique predicts a winning strategy — the higher the better.

Total Execution Time: This is the total execution time of the predicted strategies over all test instances — the lower the better.
Where the strategy selection technique predicts a strategy that was one of the timeout strategies, we add only the timeout value (6000ms) into the total. This understates the true total execution time, which we would have obtained had we not subjected strategy execution to a timeout. Strategy selection techniques that incorrectly pick strategies that timeout are, therefore, not being penalised as much as they could on these graphs.

Note that prediction rate on its own would be a misleading metric — there would be little utility to a technique that picked the best strategy for 90% of the instances if these were ones where solving time was short but which failed to pick the best strategy for the remaining 10% of instances if these were ones where the solving time exceeded the total for the other 90%. This motivates the use of total execution time as an additional metric which gives a good indication of whether the technique is making the right choices when it matters, i.e. when incorrect choices of strategy lead to very long solving times. For comparison, we also plot the minimum possible total execution time, i.e. the sum of the execution times of the best strategy for each instance.

5 Experiments

For each problem, we describe the features we use, the candidate strategies, the distribution of those strategies in the dataset and we plot the prediction rate and the total execution time.

Feature	Type	Min.	Max.	*Predictive?*		
				CBR	Full DT	Pruned DT
w	*integer*	1	13	✓	depth 1	×
m	*integer*	2	7	✓	×	×
n	*integer*	2	10	×	×	×
t	*integer*	4	70	×	depth 0	×
m/w	*real*	$\frac{2}{13}$	7	✓	depth 1	×
n/w	*real*	$\frac{2}{13}$	10	✓	depth 1	×
t/w	*real*	$\frac{4}{13}$	70	×	depth 1 or 2	×
n/m	*real*	$\frac{2}{7}$	5	✓	×	×

Table 1. Social Golfer Problem Features (w : number of weeks; m : number of groups; n : number of golfers per group; t : total number of golfers)

5.1 The Social Golfer Problem

Features The features we use are summarised in Table 1. Note how we define some features as ratios of others. One might argue that the feature, e.g., m/w is unnecessary when we already have the features m and w. However, unless we use a non-linear similarity measure [16], similarity on features m and w will not necessarily be the same as similarity on feature m/w. By explicitly including features such as m/w, we avoid the need for a non-linear similarity measure.

The final three columns of Table 1 attempt to show which of the features are selected by the Wrapper method for use in CBR and at what depth in the full decision trees induced by C4.5 the different features appear. It has to be kept in mind that this is only rough summary information: different outcomes are possible on the different folds of the cross-validation. The full decision tree has a depth of only 2; the reason that the final column, for the pruned tree, contains no information is that the full tree is pruned to a tree containing just one node, labelled by *use* s_2, i.e. use strategy 2. In fact, this is not a good decision tree for this dataset: in approximately 40% of the instances s_2 is outperformed.

Candidate Strategies Twelve strategies, summarised in Table 2, are used in our Social Golfer experiments; each is the winner on certain instances. In two strategies, we use the set-based model. In the rest, we use the combined model. Using this combined model, Bessiere et al. investigate different ways of expressing the partitioning and disjointness constraints [2]. They design ways of expressing them 'globally', which we denote fcp_g and fcd_g respectively, and ways of decomposing them into more primitive forms, which we denote by fcp_d and fcp_g respectively. So, in fact, there is not a single model here; there are four, depending on which combination of constraints is used: $\langle fcp_g, fcd_g \rangle$, $\langle fcp_g, fcd_d \rangle$, $\langle fcp_d, fcd_g \rangle$ or $\langle fcp_d, fcd_d \rangle$. Experiments reported in [2] reveal that fcp_g and fcp_d perform identically for the Social Golfers Problem, so we can arbitrarily adopt fcp_g. But this still leaves us with two models, $\langle fcp_g, fcd_g \rangle$ and $\langle fcp_g, fcd_d \rangle$.

ID	*Strategy*			
	Model	Algorithm	Var. Heuristic	Val. Heuristic
s_1	*set model*	*dfs, IlcExtended*	*group set*	*lex (set)*
s_2	*set model*	*dfs, IlcExtended*	*week set*	*lex (set)*
s_3	$fcp_g fcd_g$	*dfs, IlcExtended*	*group set*	*IloChooseMinSizeInt, lex (set)*
s_4	$fcp_g fcd_g$	*dfs, IlcExtended*	*week set*	*IloChooseMinSizeInt, lex (set)*
s_5	$fcp_g fcd_g$	*dfs, IlcExtended*	*static golfer*	*IloChooseMinSizeInt, lex (set)*
s_6	$fcp_g fcd_g$	*dfs, IlcExtended*	*static week*	*IloChooseMinSizeInt, lex (set)*
s_7	$fcp_g fcd_g$	*dfs, IlcExtended*	*min domain*	*IloChooseMinSizeInt, lex (set)*
s_8	$fcp_g fcd_d$	*dfs, IlcExtended*	*group set*	*IloChooseMinSizeInt, lex (set)*
s_9	$fcp_g fcd_d$	*dfs, IlcExtended*	*week set*	*IloChooseMinSizeInt, lex (set)*
s_{10}	$fcp_g fcd_d$	*dfs, IlcExtended*	*static golfer*	*IloChooseMinSizeInt, lex (set)*
s_{11}	$fcp_g fcd_d$	*dfs, IlcExtended*	*static week*	*IloChooseMinSizeInt, lex (set)*
s_{12}	$fcp_g fcd_d$	*dfs, IlcExtended*	*min domain*	*IloChooseMinSizeInt, lex (set)*

Table 2. Social Golfer and Extra Golfer Strategies (Strategies s_1 and s_2 are used only for the Social Golfer Problem)

For algorithms and heuristics, we follow [2], which gives us a good number of competitive strategies. In particular, we use ILOG Solver's Depth-First Search algorithm (*dfs*) with the propagation level parameter for global constraints set to *IlcExtended*, which maximises the propagation [10]. We have used five variable ordering heuristics but just one value ordering heuristic, *IloChooseMinSizeInt, lex (set)* [2]. Space limitations preclude a description of their details.

Dataset Characteristics Our Social Golfer dataset contains 367 instances prior to filtering. (The exact number of instances after filtering depends on the parameterisation of the winning window.) Fig. 2 shows, for different winner parameterisations, the number of instances where there are ties for first place; we show in how many instances there is a clear winner, in how many two strategies tie, in how many three strategies tie, and so on. More ties and higher cardinality ties make prediction easier. Fig. 3 shows, for different winner parameterisations, the percentage of instances for which each of the twelve strategies is one of the winners. The sum of the percentages exceeds 100% because an instance can have more than one winning strategy.

Results Figs. 4 and 5 show the prediction rate and the total execution time for CBR, unpruned decision trees (which gave better results than pruned ones) and the benchmarks, again for different winner parameterisations. The results are discussed in Sect. 5.3.

5.2 The Extra Golfer Problem

The Extra Golfers Problem is a generalisation of the Social Golfers Problem. It introduces x additional golfers (in our experiments $x \in [1..4]$), i.e. $t = m \times n + x$.

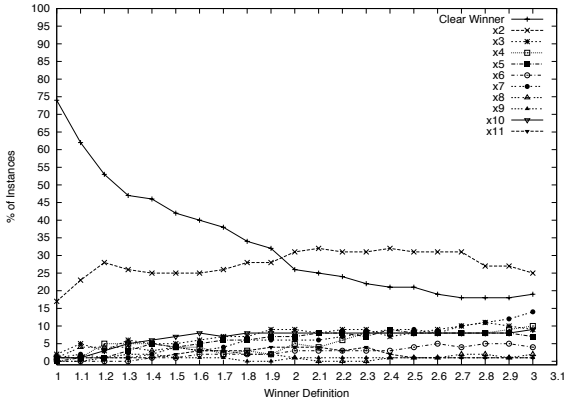

Fig. 2. Ties, Social Golfer Dataset

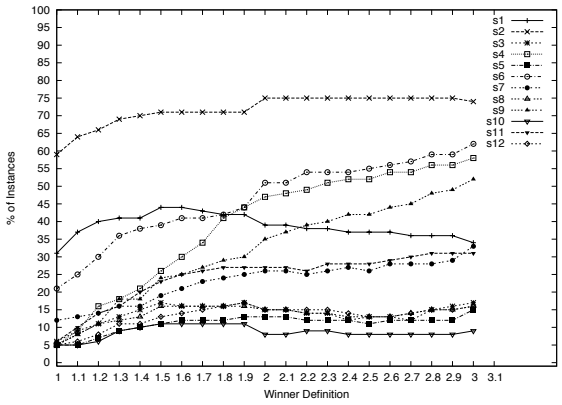

Fig. 3. Strategy Distribution, Social Golfer Dataset

Thus there is an excess of golfers and some golfers rest each week, i.e. the set of golfers is no longer *partitioned* into groups each week, as there will be some golfers left over. This may not seem like a very different problem. But, in fact, this small change to the problem brings large differences in terms of winning strategies (compare Figs. 2 and 3 with Figs 6 and 7), and therefore it is an interesting second problem for us.

Features We summarise the features in Table 3. Compared with the Social Golfer Problem, there are some additional features, and the 'predictiveness' of the features (summarised in the final three columns) is different. In this dataset, there is no dominant strategy (unlike s_2 in the Social Golfers dataset), and so all the decision trees are more complex than they were for the Social Golfers.

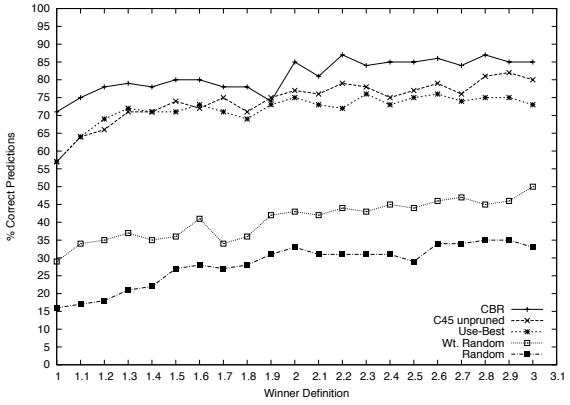

Fig. 4. Prediction Rates, Social Golfer Dataset

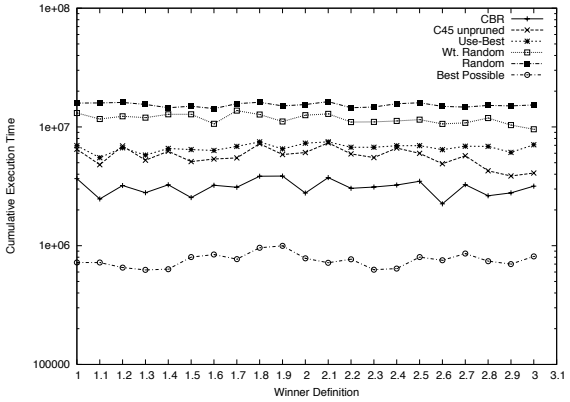

Fig. 5. Total Execution Time, Social Golfer Dataset

Candidate Strategies Ten of the same strategies that were used for the Social Golfer Problem (Table 2) can be used for the Extra Golfers Problem. The two which use the pure set-based model, s_1 and s_2, are inapplicable because the set-based model assumes that the golfers are partitioned.

Dataset Characteristics Our Extra Golfers dataset contains 440 instances prior to filtering. The number of ties and the distribution of the strategies are shown in Figs. 6 and 7. As we mentioned above, these graphs show the Extra Golfer datasets to be quite different from those for the Social Golfer Problem.

Results Results are shown in Figs. 8 and 9, and are discussed in Sect. 5.3.

Feature	Type	Min.	Max.	*Predictive?*		
				CBR	Full DT	Pruned DT
w	*integer*	1	13	✓	depth 3	×
m	*integer*	2	7	×	depth 3	×
n	*integer*	2	10	✓	depth 1 or 3	depth 1
t	*integer*	5	74	×	depth 2 or 3	×
x	*integer*	1	4	×	depth 2 or 3	×
m/w	*real*	$\frac{2}{13}$	7	✓	depth 3	×
n/w	*real*	$\frac{2}{13}$	10	✓	depth 0 or 3	depth 0
t/w	*real*	$\frac{5}{13}$	74	✓	depth 2	depth 2
x/w	*real*	$\frac{1}{13}$	4	×	depth 3 or 4	×
n/m	*real*	$\frac{2}{7}$	5	✓	depth 1 or 2 or 3	depth 2 or 3
t/m	*real*	$\frac{5}{7}$	37	✓	×	×
x/m	*real*	$\frac{1}{7}$	2	✓	depth 3	×
t/n	*real*	$\frac{5}{10}$	37	✓	depth 2 or 3	depth 2
x/n	*real*	$\frac{1}{10}$	2	×	depth 3	×
x/t	*real*	$\frac{1}{74}$	$\frac{4}{5}$	×	depth 3 or 4	depth 3 or 4

Table 3. Extra Golfer Problem Features (w : number of weeks; m : number of groups; n : number of golfers per group; x : extra golfers; t : total number of golfers)

5.3 Discussion of Results

As we would expect, the graphs for prediction rate (Figs. 4 and 8) exhibit better performance as the winning strategy definition is relaxed: as the number of joint winners grows, it becomes easier to predict a wining strategy. Of the techniques, for the Social Golfer dataset, CBR outperforms the next best techniques (use-best and decision trees) by about 10%, achieving a prediction rate of between 70 and 80%. For the Extra Golfer dataset, while CBR still has the best prediction rate, use-best and decision trees are not far behind.

The graphs for total execution time (Figs. 5 and 9) give an indication of the quality of a technique, regardless of whether a winning strategy is predicted or not. The Social Golfer dataset is the tougher of the two, because the differences in execution times render the costs of making a wrong decision greater. Here, CBR significantly outperforms the other strategies — note the logarithmic scale. This is largely because it predicts far fewer strategies that time out. The Extra Golfers dataset again brings use-best, decision trees and CBR closer in terms of performance with CBR doing slightly better.

6 Conclusions and Future Work

In this paper, we have demonstrated that CBR outperforms four other strategy selection techniques on two problems with quite different characteristics. We

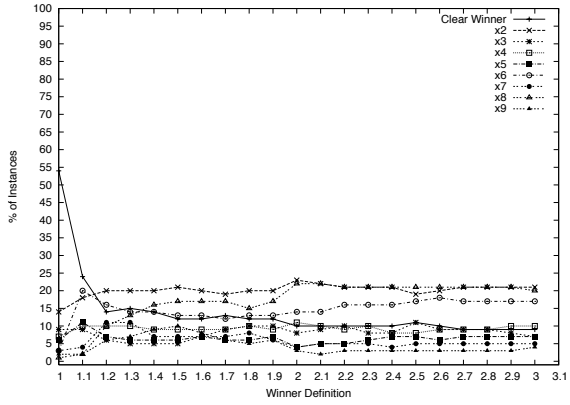

Fig. 6. Ties, Extra Golfer Dataset

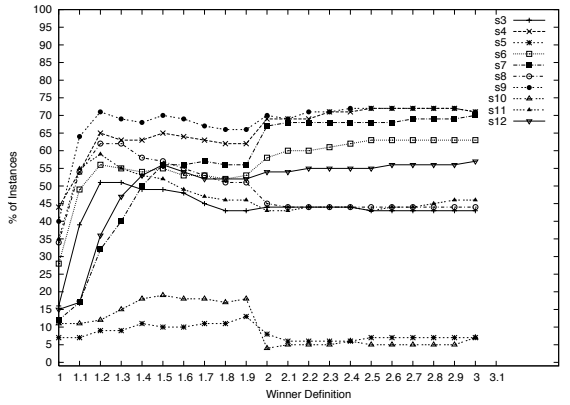

Fig. 7. Strategy Distribution, Extra Golfer Dataset

have shown empirically that CBR achieves higher prediction rates than the other techniques, and predicts fewer strategies that fail to find a solution in reasonable time. By using CBR to select solution strategies, we have demonstrated that significant amounts of computation time can be saved; such savings can have considerable economic impact.

We have shown that it is possible to achieve good results using just surface features. We have added clarity to strategy selection methodology by introducing a parameterised definition of winning strategy and determining the impact of different parameterisations.

Future work will involve other datasets for other constraint programming problems; more considered selection of case base size and contents (including consideration of case-base editing); scaling the system to facilitate a broader

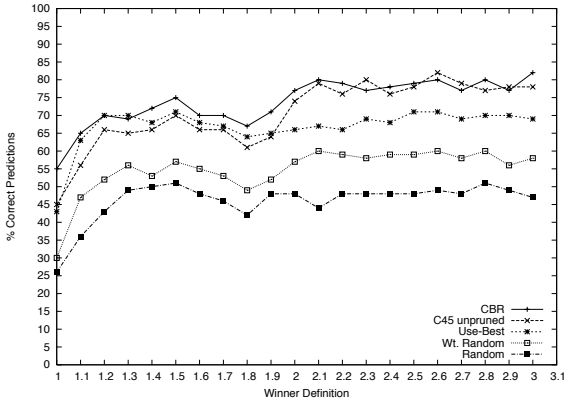

Fig. 8. Prediction Rates, Extra Golfer Dataset

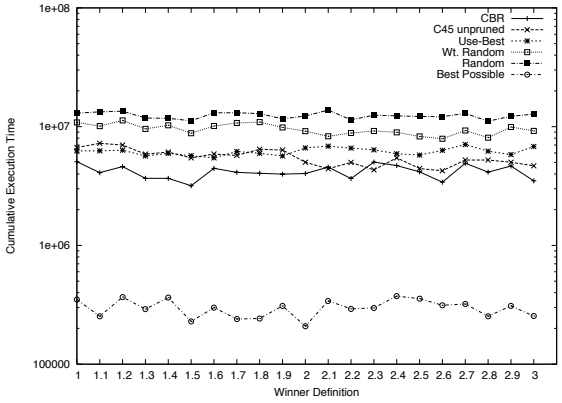

Fig. 9. Total Execution Time, Extra Golfer Dataset

selection of candidate strategies; deploying a wider range of strategy selection teqhniques (e.g. statistical methods); and further analysis of how dataset characteristics impact strategy selection.

References

1. Beacham, A., Chen, X., Sillito J. and Van Beek, P.: Constraint Programming Lessons Learned from Crossword Puzzles. In *Procs. of 14th Canadian Conference on Artificial Intelligence*, pp.78–87, 2001
2. Bessiere, C., Hebrard, E., Hnich, B. and Walsh, T.: Disjoint, Partition and Intersection Constraints for Set and Multiset Variables. In *The Principles and Practice of Constraint Programming, Procs. of CP-2004*, pp.138–152, 2004

3. Borret, J.E. and Tsang, E.P.K.: A Context for Constraint Satisfaction Problem Formulation Selection. *Constraints*, vol.6(4), pp.299–327, 2001
4. Carchrae, T. and Beck, J.C.: Low-Knowledge Algorithm Control. In *Procs. of the 19th AAAI*, pp.49–54, 2004
5. Gebruers, C., Guerri, A., Hnich, B. and Milano, M.: Making Choices using Structure at the Instance Level within a Case Based Reasoning Framework. In *Integration of AI and OR Technologies in Constraint Programming for Combinatorial Optimization Problems*, Springer Verlag, pp.380–386, 2004
6. Gent, I., Walsh, T. and Selman, B.: *CSPLib: A Problem Library for Constraints*. http://4c.ucc.ie/~tw/csplib/ (Last accessed 02/02/2005)
7. Gomes, P.: *A Case-Based Approach to Software Design*, PhD Dissertation, Universidade de Coimbra, Portugal, 2003.
8. Grabert, M. and Bridge, D.: Case-Based Reuse of Software Examplets. *Journal of Universal Computer Science*, vol.9(7), pp.627-640, 2003
9. Guo, H.: *Algorithm Selection for Sorting and Probabilistic Inference: A Machine Learning-Based Approach*. PhD Dissertation, Dept. of Computing and information Sciences, Kansas State University, 2003
10. *ILOG Solver*. http://www.ilog.com/products/solver/ (Last accessed 02/02/2005)
11. Kohavi, R. and John, G.: Wrappers for Feature Subset Selection. *Artificial Intelligence*, vol.97(1–2), pp.273–324, 1997
12. Little, J., Gebruers, C., Bridge, D. and Freuder, E.: Capturing Constraint Programming Experience: A Case-Based Approach. In *International Workshop on Reformulating Constraint Satisfaction Problems*, Workshop Programme of the 8th International Conference on Principles and Practice of Constraint Programming, 2002
13. Minton, S.: Automatically Configuring Constraint Satisfaction Programs: A Case Study. *Constraints*, vol.1(1), pp.7–43, 1996
14. Nadel, B.: Representation Selection for Constraint Satisfaction: A Case Study Using n-Queens. *IEEE Expert*, vol.5(3), pp.16–23, 1990
15. Novello, S.: *An ECLiPSe Program for the Social Golfer Problem*. http://www.icparc.ic.ac.uk/eclipse/examples/golf.ecl.txt (Last accessed 02/02/2005)
16. Pang, R., Yang, Q. and Li, L.: Case Retrieval using Nonlinear Feature-Space Transformation. In *Procs. of the 7th European Conference on Case-Based Reasoning*, pp.361–374, 2004
17. Quinlan, J.R.: *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993
18. Smith, B.: *Reducing Symmetry in a Combinatorial Design Problem*. Technical Report 2001.01, University of Leeds School of Computing Research Report Series, 2001
19. Sqalli M., Purvis, L. and Freuder, E.: Survey of Applications Integrating Constraint Satisfaction and Case-Based Reasoning. In *Procs. of the 1st International Conference and Exhibition on The Practical Application of Constraint Technologies and Logic Programming*, 1999
20. Tsang, E.: *Foundations of Constraint Satisfaction*. Academic Press, 1993
21. Wallace, M.G.: Practical Applications of Constraint Programming. *Constraints*, vol.1(1–2), pp.139–168, 1996
22. Wilson, D. C., Leake, D. B. and Bramley, R: Case-Based Recommender Components for Scientific Problem-Solving Environments. In *Procs. of the 16th International Association for Mathematics and Computers in Simulation World Congress*, CD-ROM, Session 105, Paper 2, 2000
23. Wilson, R. and Martinez, T.: Improved Heterogeneous Distance Functions. *Journal of Artificial Intelligence Research*, vol.6, pp.1–34, 1997