Using Case-based Reasoning in an Algorithm Portfolio for Constraint Solving*

Eoin O'Mahony, Emmanuel Hebrard,
Alan Holland, Conor Nugent, and Barry O'Sullivan

Cork Constraint Computation Centre
Department of Computer Science, University College Cork, Ireland
{e.omahony|e.hebrard|a.holland|c.nugent|b.osullivan}@4c.ucc.ie

Abstract. It has been shown in areas such as satisfiability testing and integer linear programming that a carefully chosen combination of solvers can outperform the best individual solver for a given set of problems. This selection process is usually performed using a machine learning technique based on feature data extracted from constraint satisfaction problems. In this paper we present CPHYDRA, an algorithm portfolio for constraint satisfaction that uses case-based reasoning to determine how to solve an unseen problem instance by exploiting a case base of problem solving experience. We demonstrate the superiority of our portfolio over each of its constituent solvers using challenging benchmark problem instances from the most recent CSP Solver Competition.

1 Introduction

It is recognised within the field of constraint programming that different solvers are better at solving different problem instances, even within the same problem class [3]. It has been shown in other areas, such as satisfiability testing [13] and integer linear programming [5], that the best on-average solver can be out-performed by carefully exploiting a portfolio of possibly poorer on-average solvers. Selecting from a portfolio usually relies on a machine learning technique based on feature data extracted from constraint satisfaction problems.

Several related pieces of work have been reported in the literature. The SATZILLA[1] system builds runtime prediction models using linear regression techniques based on structural features computed from instances of the Boolean satisfiability problem. Given an unseen instance of the satisfiability problem, SATZILLA selects the solver from its portfolio that it predicts to have the fastest running time on the instance. In the International SAT Competition 2007, SATZILLA won two of the categories, and came second and third in two others. The AQME system is a portfolio approach to solving quantified Boolean formulae, i.e. SAT instances with some universally quantified variables [9]. AQME is built on the Weka data-mining library[2]. Three versions of AQME have

* This work was supported by Science Foundation Ireland (Grant No. 05/IN/I886).

[1] http://www.cs.ubc.ca/labs/beta/Projects/SATzilla/

[2] http://www.cs.waikato.ac.nz/ml/weka/

competed in the International Competitive Quantified Boolean Formula Evaluation[3]: a version using decision trees to select which solver to use, a version using logistic regression, and another using 1-nearest neighbour. Like SATZILLA, AQME selects one solver to run for a given unseen formula. Our approach contrasts with both of these in that we select a set of solvers to run on the given instance rather than a single solver. Streeter et al. [12] build upon the work of Sayag et al. [11], by using optimisation techniques to produce a schedule of solvers that should be tried in a specific order, for specific amounts of time, in order to maximise the probability of solving the given instance. This work is similar to ours, except we use a much more "knowledge-light" approach by relying on case-based reasoning (CBR) to advise on the composition of our schedule of solvers. A related work to our own is by Gebruers et al. [2], who use case-based reasoning to select solution strategies for constraint satisfaction. Our approach is quite different since we do not tune a particular solver, but make a more high-level decision about which solvers to run.

The motivation for this research is two-fold. Firstly, constraint programming systems are often quite difficult for non-expert users to apply in practice. We address this concern by developing a system that uses artificial intelligence techniques to automatically select an appropriate solver for a given unseen problem instance. Secondly, we aim to show that given the current state of CSP Solver development, one could win the International CSP Solver Competition by not implementing any new solvers, but by using machine learning to select between a small set of common solvers that have already been developed. Our approach uses case-based reasoning to inform the selection process. We build a case base of problem solving experience by solving a variety of typical problem instances with each solver in our algorithm portfolio. We employ case retrieval methods in a number of increasingly sophisticated ways, giving better performance in each case. We demonstrate the superiority of our portfolio over each of its constituent solvers using challenging benchmark problem instances from the most recent CSP Solver Competition. We show that CBR can be used to control an algorithm portfolio for satisfying CSPs from a wide variety of problem settings with performance comparable with the best possible choice for each instance.

2 Preliminaries

Constraint Satisfaction Problem. A *constraint satisfaction problem* (CSP) is defined by a finite set of variables, each associated with a domain of possible values that the variable can be assigned, and a set of constraints that define the set of allowed assignments of values to the variables [7]. The *arity* of a constraint is the number of variables it constrains. Given a CSP the computational task is normally to find an assignment to the variables that satisfies the constraints, which we refer to as a *solution*. The precise manner in which the constraints in a CSP are defined varies, but typically one can identify three general forms: an *extensional* constraint is defined by a table of allowed/disallowed assignments to the variables it constrains; an *intentional* constraint is defined in terms of an expression that defines the relationship that must hold amongst

[3] `http://www.qbflib.org/index_eval.php`

the assignments to the variables it constrains; a *global* constraint is non-binary and is often associated a dedicated filtering algorithm. Typical examples of global constraints are: ALLDIFFERENT which requires that the variables it constrains take different values; ATMOSTNVALUE which requires that the variables it constrains take at most a specified number of different values.

The International CSP Solver Competition. The motivation for the work reported here was to build an algorithm portfolio to participate in the 2008 International CSP Solver Competition[4]. There are *five categories of benchmark problem* in the competition: 2-ARY-EXT instances only involving binary (and unary) extensional constraints; N-ARY-EXT instances involving extensional constraints. At least one constraint has an arity strictly greater than two; 2-ARY-INT instances involving binary (and unary) intentional constraints. Binary (and unary) extensional constraints can also be involved but no global constraint are used. At least one constraint is specified intentionally; N-ARY-INT instances involving intentional constraints. Extensional constraints can also be involved but no global constraint are used. At least one constraint is specified intentionally and at least one constraint has an arity strictly greater than two; GLB instances involving any kind of constraints, including global constraints.

Each entry is run on a cluster of Linux-based computers. On each instance each entry is given a time limit within which to solve the instance, typically 30 minutes; solving means either finding a solution, or proving that none exists. The entry that solves the most instances is declared the winner, with ties being broken by considering the minimum total solution time.

3 Portfolios of Solvers

It often occurs that there is a low correlation between the performance of different search techniques. This may, fortuitously, offer the prospect of combining various search mechanisms so that we form a portfolio of algorithms amongst which computing resources are shared so that the combined effectiveness of multiple solvers outperforms the single most effective solver. There is a balance between risk (the variability in search performance) and reward (the expected search performance) that must be achieved. In previous work, Huberman *et al.* [4] developed a theory of algorithm portfolio design that employed an economics-based approach in an effort to balance risk and reward. Theirs was a general method for combining existing programs in a *static* portfolio so that the combinations were unequivocally superior to any of the individual algorithms. They employed Modern Portfolio Theory, as described by Markowitz [8], to model the *efficient frontier*. An efficient portfolio is one that has the highest possible reward for a given level of risk, or the lowest risk for a given reward.

3.1 Case-Based Reasoning

Case-Based Reasoning is a machine learning methodology that adopts a lazy learning approach and contains no explicit model of the problem domain. Instead a set of past

[4] Competition web-site: `http://cpai.ucc.ie/08/`

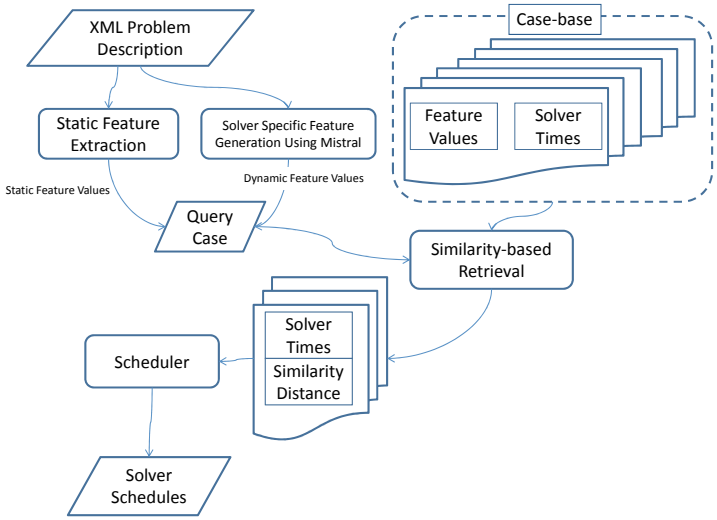

Fig. 1. Overview of CPHYDRA.

examples called *cases* are retained. Each case is made up of a description of a past example or experience and its respective solution. The full set of past experiences encapsulated in individual cases is called the case base.

In CBR problems are solved "by using or adapting solutions to old problems" [10]. When a new problem is presented, the case base is searched, similar past examples are found and these are used to solve the presented problem. The need to detect and model general patterns over the entire problem space is avoided. This approach to problem solving in CBR has a number of advantages. In particular, CBR has proven to be successful in solving *weak-theory* problems, in which little insight into the problem exists and the problem domain may be complex. This characteristic makes CBR a good candidate for the tasks of solver selection and portfolio generation in CSPs. While many different solving techniques exist, it is a difficult task even for domain experts to predict which techniques, or combination thereof, will be effective on a given problem instance.

Given its apparent suitability, it is not surprising that CBR has previously been considered for the task of strategy selection in Constraint Programming. CBR has been outlined as part of a framework for capturing expert knowledge in terms of CP problem modelling [6]. CBR has also successfully outperformed other CP strategy selection techniques when tested on two different CSPs [2]. Within CPHYDRA, the CBR methodology is used to inform a portfolio approach to problem solving.

3.2 CPHYDRA

The most fundamental element of any CBR system is the case base. In CPHYDRA, cases contain the feature values describing a particular CSP problem instance. Each problem instance in the CSP Solver Competition is described in XML and we can use

this to help generate the feature values for a particular problem instance. We do this in two distinctly different ways. Firstly, we extract a set of *syntactic* features such as maximum constraint arity, average and maximum domain size, number of variables and of constraints, domain continuity, ratio of different types of constraints (extensional, intentional or global) and ratio of subclasses of constraints within these main categories. Then we complement these static features with *solver specific* features by running a constraint solver, `Mistral`, for a limited amount of time (typically two seconds) and recording its modeling choices and search statistics, such as number of variables whose domains are represented as a range/boolean/bitset, number of extra variables created to represent constraint reification, number of nodes explored, number of constraint propagation calls and average constraint weighting.

We used 36 features in total to describe the problem instances. Of these features, 14 were generated using `Mistral`. It is also important to note that quantified features, such as maximum domain size, were log-scaled whereas the ratios were all percentages. As well as the set of feature-values, each case contains a list of how long each solver being used by CPHYDRA took to solve that problem instance. This formed the 'experience' element of each case.

The CBR methodology can be broken down into four distinct phases: Retrieval, Reuse, Revision and Retention. These phases are often referred to as the 'Four REs'-cycle [1]. The first two are of particular importance to CPHYDRA, as can be seen in Figure 1. However, the final two phases are relevant too. We will now discuss each of these four phases relates to CPHYDRA in turn.

Retrieval. To begin with, a query case is first produced. The XML presentation of the problem instance is used to generate both the static and the dynamic feature-values as described previously. Using the query case the case base is searched and the most similar cases to the present problem description are retrieved. A simple k-nearest neighbour (K-NN) algorithm is used for this task; we set $k = 10$. In situations where similarity ties occur all cases with similarity equal to the k^{th}-ranked case are also returned. Since all features are real valued the Euclidean similarity measure is used.

Reuse. The objective of CPHYDRA is not simply to supply a prediction but a schedule describing how long each solver should run. This makes the Reuse phase of CPHYDRA more complex than many other CBR systems where the objective is a classification. As can be seen in Figure 1 the retrieval process returns the set of solver times for each of the k most similar problem instances found in the case base along with their similarities to the Query Case. This information is then used to generate a solver schedule. This process will be explained in detail in Section 3.3.

Revision. During this phase the proposed solution is evaluated and validated. This process involves running each of the solvers for the proportion of time allocated by the scheduler and determining if the problem is successfully solved by one of the solvers within its allocated time slot. If at least one solver solves the problem instance within its time slot then the schedule is deemed a success. In competition conditions there is no opportunity to revise a solution in light of its performance or to update the case base. However, in non-competition conditions this would be desirable.

Retention. Normally once a satisfactory solution has been determined the problem description and solution are added to the case base. However, in CPHYDRA the 'solution' or experience attached to each case, the solver times, are only indirectly used to produce a solution. In order to create a complete case that could be retained each solver would have to run until it solved the problem instance or timed-out. This phase is not possible in competition conditions but could form part of a continuous online learning system.

As we have already stated, the most involved action in this cycle is the generation of a schedule given the solver performances on similar past examples. We will now discuss this process in greater detail.

3.3 Solver Scheduling

Typically, in previous CSP Solver Competitions runtime distributions for each solver display a very fast rate of decay. That is, the number of problems solved for a given amount of CPU-time decreases rapidly. Moreover, when analysing the competition results, it turns out that no solver is completely dominated. For example, sixteen of the twenty-two solvers are the fastest on at least one instance, and nine of them are the fastest on at least one hundred instances. The conjunction of these two conditions entails that partitioning CPU-time between solvers is inherently a good strategy. We show, however, that one can improve a naive partitioning by taking into account the information given by the case base reasoner. We define a method for computing good CPU-time partitions, i.e., *solver schedules*.

Our goal is to compute a solver schedule, that is a function $f : \mathcal{S} \mapsto \mathbb{R}$ mapping an amount of CPU-time to each element of a set $\mathcal{S}$ of solvers. Given a query instance, the case base reasoner returns a set $\mathcal{C}$ of similar cases. Informally, we compute the schedule so that the number of cases in $\mathcal{C}$ that would be solved using this schedule is maximised. More formally, consider a set $\mathcal{C}$ of similar cases. For a given solver $s \in \mathcal{S}$ and a time point $t \in [0..1800]$ we define $C(s,t)$ as the subset of $\mathcal{C}$ solved by s if given at least time t. The schedule f can be computed using the following constraint program:

$$\text{maximise } |\bigcup_{s\in\mathcal{S}} C(s, f(s))| \tag{1}$$

$$\text{subject to } \sum_{s\in\mathcal{S}} f(s) \leq 1800 \tag{2}$$

Notice that this problem is NP-hard as a generalisation of the knapsack problem. However, because the number of solvers is small, solving this problem to optimality is easy in practice. We refined the objective function (1) by weighting the elements of the sets (cases) according to their similarity to query case. Let $d(c)$ be the distance of case $c \in \mathcal{C}$ to the query case, we can modify the objective function in the following way:

$$\text{maximise } \sum_{c\in\bigcup_{s\in\mathcal{S}} C(s,f(s))} \frac{1}{d(c)+1} \tag{3}$$

We solved this problem using a very simple complete search procedure. The low number of solvers in CPHYDRA (5) and number of similar cases (10 to 50) makes the problem tractable. However, for a large number of solvers a more sophisticated approach would be necessary.

Often, the constraint program above can be trivially solved by allocating to each solver s an amount of CPU-time t such that $|C(s,t)|$ is maximised. For instance consider the situation where we have five solvers, and none of them can solve any more instances after $t = 100$. In this case the objective function is trivially maximised, without violating the constraints, by allocating 100 seconds to each solver. In other words, in this type of situation the schedule, as defined previously, is useless. We, therefore, distinguish these cases and apply a simple alternative procedure. We first disregard solvers that are dominated by some others. That is, let t be a time point such that no case can be solved by any solver in less than t seconds. We say that s_1 is dominated iff (1) $\exists s_2 \in \mathcal{S}$ such that $C(s_1,t) \subset C(s_2,t)$, or (2) there exists $s_2 \in \mathcal{S}$ such that $C(s_1,t) = C(s_2,t)$ and there exists t' such that $C(s_2,t') = C(s_2,t)$ and $C(s_1,t') \subset C(s_2,t')$, or (3) $C(s_1,t) = C(s_2,t)$ for all t and s_2 comes before s_1 for some arbitrary static order (tie breaking). Since we are focusing on maximising the probability of solving the query instance within the time limit, the order does not matter. However, it is important to notice that if one wants to minimise the expected solving time, the chosen order can be significant. Once dominated solvers are eliminated, the remaining CPU-time is distributed amongst non-dominated solvers proportionally to the amount of CPU-time already assigned. This corresponds to the maximally risk-aggressive algorithm portfolio that seeks to maximise the probability of finding a solution. However, this does not necessarily minimise the expected time for finding a solution.

4 Experimental Results

Experimental Aims. Given that our aim is to produce a portfolio of solvers for the CSP Competition, to test different strategies we use the percentage of problems solved by each strategy as a metric, where the total number of problems is the number of problems that were solved by at least one solver within the corresponding time cutoff. Notice that in Figure 2 the solver `Buggy` has a higher percentage of solved problems after five minutes than when it is able to use the full thirty minutes, i.e., it solved more instances in five minutes relative to other solvers, but not in absolute value.

Experimental Data and Methodology. The following tests ran on data from the 2006 CSP Solver Competition. We used this data as it is complete and has a diverse range of solvers. The complete case base therefore contains 3013 cases, one for each instance solved by at least one entrant of the competition within a 30 minutes cutoff. All the results were obtained by running a randomised ten-fold cross validation ten times and averaging the results. We compared the five best solvers of the previous competition (`Buggy`, `Bprolog`, `Sugar`, `Mistral` and `Abscon`) against CPHYDRA with the same solvers in the portfolio, and using the three following strategies:

Split schedule: A schedule giving each solver an equal portion of the total time.
Static schedule: A schedule generated as in Section 3.3 using the entire case base.
Dynamic schedule: A schedule generated as in Section 3.3 using the k nearest neighbours of the target case ($k = 10$).

Experimental Results. It is clear from Figure 2 that even a naive schedule such as dividing the time evenly between the solvers performs very well compared to the best

individual solver with an 8-10% greater success rate. In this graph, we plot the percentage of instances solved, within either 30 minutes, 5 minutes of 1 minutes, for each individual solver and each of the time splitting strategies described above. Notice that the percentages are over the number of instances that were solved by at least one solver in the respective duration. However, the best scheduling approach only marginally outperforms the others (See Table 1). The performance of the split schedule degrades as the available time is decreased. There is a 1% difference between it and a dynamic schedule when 30 minutes is available whereas there is a 3% gap when the time available is decreased to 5 minutes and a 10% gap when there is only 1 minute available.

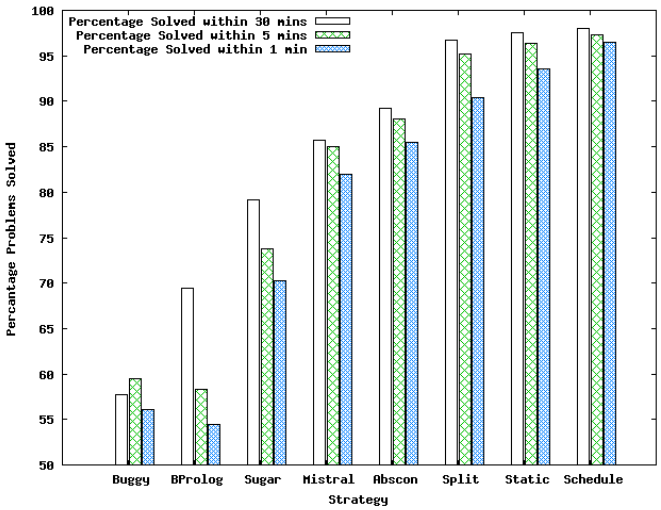

Fig. 2. A graph comparing the percentage of problems solved by different strategies when there are different amounts of time available. Running each solver in the portfolio exclusively, splitting the time evenly between the solvers, and two different schedules.

This is illustrated in Figure 3. The static and dynamic schedules begin to outperform a split schedule rapidly as the time available decreases. The static and dynamic schedules are roughly equivalent in terms of problems solved when the time allowed is more than 5 minutes. We believe the reason for this is that most of the problems in the CSP Solver Competition are solved in very short time period. Thus, given generous time constraints, any distributed schedule should do well. The efficiency of these approaches becomes apparent when the time available decreases and algorithm selection becomes more critical. The dynamic schedule clearly outperforms the static schedule when the time allowed is shorter.

From Table 1 it is clear that the relative performance gap increases as the time available decreases. There is a clearly discernable trend as the time available approaches 30 seconds. The static and dynamic schedules go from being within 0.2% of each other at 30 minutes and at 15 minutes to being 1% apart at 5 minutes, 1.8% apart at 2 minutes

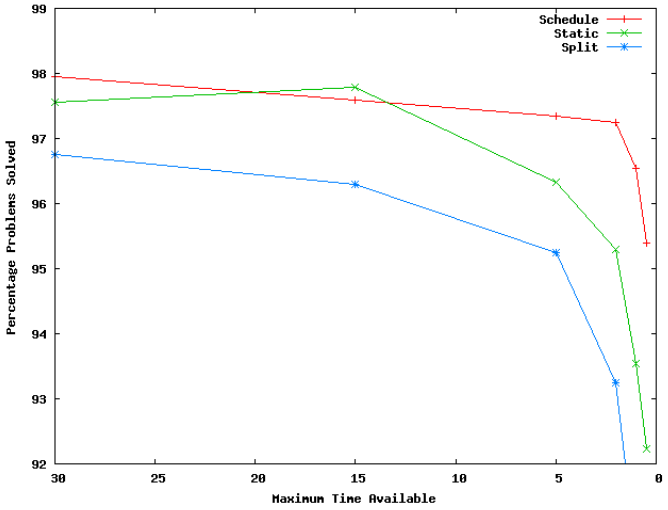

Fig. 3. Percentage (avg) of solved instances for each approach for different time-limits (minutes).

to 3% apart at 1 minute. This is evidence that the case-based reasoning system is having a beneficial effect and the schedules generated on local neighbours are more robust than schedules generated from all known data.

Table 1. Table detailing the performance of each approach.

	Split	**Static**				**Dynamic**			
Time (Mins)	Solved	Best	Worst	Avg	Dev	Best	Worst	Avg	Dev
30	96.76	97.63	97.42	97.56	0.075	98.16	97.13	**97.86**	0.28
15	96.30	97.83	97.76	**97.79**	0.02	97.76	97.39	97.59	0.12
5	95.24	96.39	96.18	96.33	0.07	97.50	97.22	**97.34**	0.08
2	93.25	95.60	95.31	95.49	0.1	97.95	97.02	**97.25**	0.31
1	90.45	93.77	93.26	93.54	0.14	96.79	96.32	**96.54**	0.15
0.5	85.38	92.24	92.12	92.22	0.04	95.63	95.02	**95.39**	0.18

Preformance at the 2008 CSP Solver Competition. At the time of writing the final version of this paper, preliminary results from the 2008 CSP Solver Competition were available, showing that we achieved our goal to obtain better performance than each of the constituent solvers of the portfolio. Our competition entry of CPHYDRA comprised three solvers: `Abscon`, `Choco` and `Mistral`. Unfortunately the results of other entrants are not available yet. The results are summarized in Table 2. For each solver, we give the percentage of instances solved by each solver, as well as the average CPU-time spent on solved instances. CPHYDRA dominates its constituent solvers in every category for the percentage of instances solved (the criterion used to rank solvers during the competition). More surprisingly, it is also competitive in average CPU-time. Therefore, CPHYDRA would win a competition against its constituent solvers.

Table 2. Results summary for CPHYDRA and its constituent solvers at the 2008 CSP Solver Competition.

Category (#instances)	CPHYDRA		Abscon		Choco		Mistral	
	Solved	CPU-time	Solved	CPU-time	Solved	CPU-time	Solved	CPU-time
Binary Extensional (622)	**92%**	**62.44 s**	88%	93.26 s	89%	95.78 s	89%	70.21 s
Binary Intentional (634)	**94%**	71.37 s	81%	**43.40 s**	82%	55.89 s	82%	58.23 s
Global (501)	**84%**	80.83 s	37%	170.62 s	69%	69.69 s	80%	**56.59 s**
N-ary Extensional (607)	**97%**	**78.20 s**	90%	80.09 s	73%	189.10 s	94%	83.67 s
N-ary Intentional (660)	**86%**	54.70 s	74%	53.21 s	78%	49.70 s	80%	**32.02 s**

5 Conclusion

We introduced CPHYDRA, a portfolio of constraint solvers exploiting a case base of problem solving experience. We detailed the novelties of our approach with respect to related work. In particular, CPHYDRA combines machine learning (CBR) with the idea of partitioning CPU-time between components of the portfolio in order to maximise the expected number of solved problem instances within a fixed time limit. Finally, we assessed the effectiveness of our portfolio over each of its constituent solvers using data from the most recent CSP Solver Competition showing the dominance of CPHYDRA.

References

1. Agnar Aamodt and Enric Plaza. Case-based reasoning: Foundational issues, methodological variations, and system approaches. *AI Commun.*, 7(1):39–59, 1994.
2. Cormac Gebruers, Brahim Hnich, Derek Bridge, and Eugene Freuder. Using cbr to select solution strategies in constraint programming. In *Proc. of ICCBR*, pages 222–236, 2005.
3. Carla P. Gomes and Bart Selman. Algorithm portfolios. *Artif. Intell.*, 126(1-2):43–62, 2001.
4. Bernardo E. Huberman, Rajan M. Lukose, and Tadd Hogg. An Economics Approach to Hard Computational Problems. *Science*, 275:51–54, 1997.
5. Kevin Leyton-Brown, Eugene Nudelman, and Yoav Shoham. Learning the empirical hardness of optimization problems: The case of combinatorial auctions. In *CP*, pages 556–572, 2002.
6. James Little, Cormac Gebruers, Derek Bridge, and Eugene Freuder. Capturing constraint programming experience: A case-based approach. In A. M. Frisch, editor, *Workshop on Reformulating Constraint Satisfaction Problems*, 2002.
7. Alan K. Mackworth. Consistency in networks of relations. *Artif. Intell.*, 8(1):99–118, 1977.
8. Harry M. Markowitz. Portfolio selection. *Journal of Finance*, 7(1):77–91, 1952.
9. Luca Pulina and Armando Tacchella. A multi-engine solver for quantified boolean formulas. In *CP*, pages 574–589, 2007.
10. C. Riesbeck and R. Schank. *Inside Case-Based Reasoning*. Erlbaum, 1989.
11. Tzur Sayag, Shai Fine, and Yishay Mansour. Combining multiple heuristics. In *STACS*, pages 242–253, 2006.
12. Matthew J. Streeter, Daniel Golovin, and Stephen F. Smith. Combining multiple heuristics online. In *AAAI*, pages 1197–1203. AAAI Press, 2007.
13. Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. : The design and analysis of an algorithm portfolio for sat. In *CP*, pages 712–727, 2007.