# Learning Algorithm Portfolios for Parallel Execution

Xi Yun[1] and Susan L. Epstein[1, 2]

[1] Department of Computer Science, The Graduate School of The City University of New York, New York, NY 10016, USA
[2] Department of Computer Science, Hunter College of The City University of New York, New York, NY 10065, USA
xyun@gc.cuny.edu, susan.epstein@hunter.cuny.edu

**Abstract.** Portfolio-based solvers are both effective and robust, but their promise for parallel execution with constraint satisfaction solvers has received relatively little attention. This paper proposes an approach that constructs algorithm portfolios intended for parallel execution based on a combination of case-based reasoning, a greedy algorithm, and three heuristics. Empirical results show that this method is efficient, and can significantly improve performance with only a few additional processors. On problems from solver competitions, the resultant algorithm portfolios perform nearly as well as an oracle.

**Keywords:** constraint satisfaction, algorithm portfolio, parallel processing, machine learning.

## 1 Introduction

Given a set of solvers and a set of constraint satisfaction problems (*CSPs*), no one solver may consistently outperform all the others on every problem (e.g., [1-5]). Informally, an algorithm portfolio is a set of algorithms that run according to some schedule on a set of problems. The thesis of this work is that learning and parallelism can improve the efficiency and effectiveness of algorithm portfolios, so that they outperform each of their constituents. This paper explores offline learning to construct such portfolios for CSPs. Given the performance of several algorithms on a training set, we seek an algorithm portfolio that executes on multiple processors to solve the most problems within some time limit. The principal result reported here is that, given several additional processors, our method can construct algorithm portfolios whose performance is competitive with that of an *oracle*, a solver that always chooses the best available algorithm for each problem.

For parallel execution, a portfolio could simply schedule the same CSP on many processors, each of which would execute a different solver on it, and then race until some algorithm found a solution. Given the number of plausible solver configurations, this approach is not realistic. It is, however, possible to learn to schedule a set of solvers on a set of processors. Our approach combines case-based reasoning (*CBR*), a greedy algorithm, and a set of heuristics. Although CBR [6] and greedy algorithms [7] have been applied to construct portfolios for CSPs before, this work is, to the best of our knowledge, the first to combine them in a single framework. Given a CSP, our

method uses CBR to identify a small set of similar training problems, and then greedily generates an effective portfolio without the complete search necessary to find an optimal one. In addition, we introduce three heuristics that transform algorithm portfolios intended for a single processor into ones intended for parallel execution. Extensive experiments show that portfolios produced by our method would solve more problems, not only when they are designed for one processor, but also consistently improve performance when they are designed for as many as 16 processors.

The next two sections provide background on CSPs and algorithm portfolios. Section 4 formulates algorithm portfolio construction as a machine learning task and reviews related work. Section 5 discusses a general framework that combines CBR with a greedy algorithm to construct algorithm portfolios, and Section 6 generalizes that framework to parallel algorithms. Subsequent sections detail and discuss the experimental design and results, and offer some conclusions.

## 2 Constraint Satisfaction Problems

A CSP here is a triple $<X, D, C>$, where $X$ is a set of variables, $D$ is a set of finite domains associated with those variables, and $C$ is a set of constraints that those variables must satisfy. A constraint defined on two variables is *binary*, and one defined on $n > 2$ variables is *n-ary*. An *extensional* constraint explicitly represents a set of tuples; an *intensional* constraint implicitly describes tuples with a predicate.

An *instantiation* of a CSP assigns values to its variables from their respective domains. A *consistent* instantiation violates no constraint. An instantiation of all the variables is a *complete* instantiation, and a complete and consistent instantiation is a *solution*. A CSP is *solvable* if it has at least one solution; otherwise it is *unsolvable*.

Many constraint solvers search for a solution to a CSP with *systematic backtracking*, which assigns values to variables one at a time and checks consistency after each assignment. After an assignment, any inconsistent value for an as-yet-unassigned variable is temporarily removed from that variable's domain. A *wipeout* occurs when a domain becomes empty. At that point, search backtracks to an earlier variable with an alternative value, restores removed values along the way, and assigns another value to the earlier variable. Search returns a solution when one is found, or halts when the domain of the variable at the root of the search tree becomes empty.

A CSP solver is typically a complex combination of fundamental search algorithms, along with a set of techniques, heuristics, and policies to realize and support them. To improve overall search performance, *preprocessing* techniques manipulate the problem before a full search, *variable-ordering* heuristics choose the next variable to be assigned a value, and *value-ordering* heuristics choose a value for it. Once a heuristic orders the possible variables or values, *randomization* chooses one at random, usually from a small set of the top-ranked candidates [8]. A *restart* policy is a sequence of termination conditions that trigger the re-initiation of the search. Combined with randomization, a restart policy may improve search performance. Although the many ways to assemble a solver's components and then set their parameters yield a broad spectrum of search performance, they also provide fertile raw material for effective algorithm portfolio construction.

## 3 Algorithm Portfolios

An algorithm portfolio for CSP solution was originally defined as a method that combined different algorithms to improve search performance while it lowered *search risk*, the standard deviation of a performance metric (e.g., expected CPU time or number of backtracks to solve a problem) [9, 10]. In other words, an algorithm portfolio searched for a Pareto frontier in the two-dimensional space defined by a given performance metric and its standard deviation. Later, an algorithm portfolio was generalized to denote a combination of different algorithms intended to outperform the search performance of any of its constituent algorithms [3, 6, 11-14]. Here we extend that formulation, so that an algorithm portfolio schedules its constituent algorithms to run concurrently on a set of processors.

Let an *algorithm* be any CSP solver, as described in the previous section. Given a set $A = \{a_1, a_2, \ldots, a_m\}$ of $m$ algorithms, a set $P = \{x_1, x_2, \ldots, x_n\}$ of $n$ problems, and a set of $B$ consecutive time intervals $T = \{t_1, t_2, \ldots, t_B\}$, a *simple schedule* $S_k$ for a problem on a single processor specifies which algorithm addresses the problem in each time interval, that is, $S_k$: $T \rightarrow A$. (At most one algorithm executes in any time interval in a simple schedule.) A *schedule* for $K$ processors is a set of $K$ simple schedules, one for each processor. (Here, a schedule addresses only one problem at a time.) An *algorithm portfolio* is then a quintuple $<P, A, K, S, B>$ where $S$ is a set of schedules that deploy algorithms $A$ on $K$ processors to solve problems from $P$ within $B$. Note that our definition includes both *simple* ($K = 1$) and *parallel* ($K > 1$) algorithm portfolios. Without loss of generality, we also simplify $T$ to $\{1, 2, \ldots, B\}$. Of course, neither a simple nor a parallel schedule can outperform an oracle's perfect algorithm selection.

Clearly, on one processor at most $B$ time can be allotted to any algorithm on any problem. Thus the performance of $A$ on $P$ can be represented as an $n \times m$ *performance matrix* $\tau$. If the entry $\tau_{ij} \in \{1, 2, \ldots, B\}$ then $a_j$ solves $x_i$ in time $\tau_{ij}$; otherwise $x_i$ goes unsolved by $a_j$ in time $B$. A *deterministic* algorithm consistently produces the same output given the same problem and time cutoff; that is, for a deterministic algorithm each $\tau_{ij}$ is fixed. In contrast, the output of a randomized algorithm may change from one run to the next (i.e., $\tau_{ij}$ is a random number).

Given a problem, a *sequential* algorithm portfolio executes algorithms on it in a specific order, but does not preserve any intermediate search data for an algorithm when the portfolio leaves it. Thus, a sequential portfolio must restart on the problem if it later reapplies a previous algorithm to it. In contrast, a *switching* algorithm portfolio interleaves algorithms, and preserves intermediate search data, so that search can continue from a previous state when it returns to an earlier algorithm. *Algorithm selection* is an algorithm portfolio that schedules only one algorithm [13, 15].

The schedule for a *static algorithm portfolio* is constructed in advance, and goes unchanged during search. In contrast, a *dynamic algorithm portfolio* can profit from feedback as it executes, and adjust its schedule accordingly. For example, the dynamic algorithm portfolios in [2, 16] iteratively share a (possibly varying-length) time slice among all available algorithms, but modify the algorithms' relative priorities based on their progress. Adjustments for a dynamic portfolio can be triggered by unsatisfactory performance during execution [17, 18]. Most of the work referenced thus far is for simple schedules, which interleave algorithms on a single processor.

There are other ways to exploit parallel processing beyond the scope of this paper. These include *search space splitting* to partition the search space of a CSP into subspaces and uses different processors to explore difference subspaces [19], and *structural decomposition* to separate a CSP into simpler, smaller-size subproblems based on the structure of its constraint hypergraph [20, 21]. Moreover, a parallel SAT solver can share clauses learnt on different processors, where each processor executes a manually pre-determined algorithm [22].

The current algorithm portfolio performance metric is runtime, which may be used to optimize different objective functions. For example, a portfolio may be required to minimize its expected runtime on a problem generated at random from some problem distribution. (Alternatives are introduced in [14].) Recent CSP solver competitions evaluated solvers on how many problems they solved under a fixed, per-problem time limit, and broke ties on average solution time across solved problems [23, 24]. We compare algorithm portfolio construction methods (henceforward, *constructors*) with the same standard. (In contrast, SAT solver competitions have compared solvers with a complex scoring function that includes the performance of all competitors [25].)

As formulated here, the differences between two solvers may be simply in their choice of even a single technique, heuristic, or policy that sustains performance diversity. Thus an algorithm portfolio can be thought of as a mixture of experts [26], including variable-ordering and value-ordering heuristics, restart policies, and nogood learning methods. In particular, even if only one heuristic is available, the portfolio could consist of the heuristic and its opposite, or the heuristic and random selection.

## 4 Learning an Effective Algorithm Portfolio

Algorithm portfolio constructors that learn are classified as online or offline based on the way they use their training problems. An *offline* constructor observes the performance of algorithms on a set of training problems and then builds a portfolio of those algorithms to optimize its performance on an entire testing set [3, 6, 13]. An *online* constructor solves one problem at a time, and the knowledge it relies on for that problem comes only from the problems that preceded it [2, 7, 16]. This paper focuses on offline algorithm constructors.

Our case-based approach to algorithm portfolio construction relies on feature extraction. Figure 1 represents offline algorithm portfolio construction with feature ex-

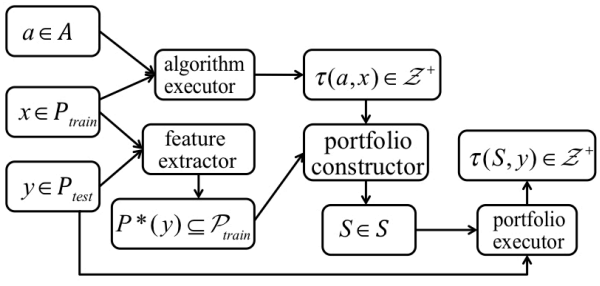


**Fig.1.** Algorithm portfolio construction as offline learning.

traction as a machine-learning task. Given a set $P_{train}$ of training problems, a set $P_{test}$ of testing problems, and a performance matrix $\tau(a,x)$ that stores the time required by each algorithm $a \in A$ to solve each problem $x \in P_{train}$, the constructor's task is to find a schedule $S$ with optimal performance that uses $A$ to solve $P_{test}$. Here, all entries in $\tau$ are discrete, fixed positive integers, that is, all algorithms are assumed to be deterministic. $P^*(y)$ is a set of CSPs similar to testing problem $y$. (Portfolios of randomized algorithms are discussed in [3, 27].)

The two portfolio constructors most relevant here are CPHYDRA [6] and GASS [7]. Let $P(a_j, S)$ be the problems in $P$ solved by $a_j$ under schedule $S$. CPHYDRA defines the optimal schedule as one that maximizes the number of problems solved within $B$:

$$\operatorname*{argmax}_S \left| \bigcup_j P(a_j, S) \right| \text{ such that length}(S) \leq B$$

Because it uses relatively few algorithms in competition, CPHYDRA can address optimality with exhaustive search, in time $O(2^m)$ where $m$ is the number of algorithms. CPHYDRA had two entries in the 2008 competition, both with $m = 3$: CPHYDRA_k_10 used 10 similar training examples (i.e., $|P^*(y)| = 10$), and CPHYDRA_k_40 used 40. Among 24 competitiors, both versions finished in the top two solvers, except in the category for global constraints. CPHYDRA also weights training problems by their Euclidean distance from the testing problem. Its approach was later exploited and tailored for SAT problems [28] as well.

GASS' greedy algorithm bases its optimal schedule on $c_i(S)$, the expected time to solve $x_i$ under schedule $S$. Its optimal schedule minimizes the overall runtime (equivalent to the average runtime under fixed $n$) to solve all problems in $P_{train}$:

$$\operatorname*{argmax}_S \sum_{i=1}^{n} c_i(S)$$

At each step, GASS greedily maximizes the number of problems solved per unit of time, and counts only problems solved for the first time during the current time step. In time $O(nm \log n \cdot \min\{n, Bm\})$, GASS returns an approximate schedule that is at most four times worse (a *4-approximation*) than the optimal switching schedule. The computation of any better approximation is NP-hard [7].

## 5 WG, a New Constructor for Switching Algorithm Portfolios

Our Weighted Greedy (*WG*) algorithm is a new constructor for switching algorithm portfolios that exploits the perspectives of both GASS and CPHYDRA. For a single processor, CPHYDRA uses CBR to select a small set of similar training problems for each testing problem. It then does a complete search, exponential in the number of algorithms $m$, to find an optimal schedule for the new problem. In contrast, the impact of $m$ on GASS is at worst quadratic; GASS' greedy approach is heavily dependent on the number of training problems $n$ instead. WG exploits the fact that some problems are far more similar to a given testing problem than others, so that a properly selected subset of problems can estimate the runtime of the testing problem more precisely.

*Input*: training set $P = \{x_1, x_2, \ldots, x_n\}$, algorithms $A = \{a_1, a_2, \ldots, a_m\}$, time limit $B$, testing problem $y$, weight function $w$: $\Re^d \rightarrow \Re^d$, neighbor set ratio $r$
*Output*: schedule $S$ for a non-parallel switching algorithm portfolio

For $i = 1$ to $n$, compute Euclidean distance between $x_i$ and $y$
$P^* \leftarrow \{100r\%$ of problems in $P$ closest to $y\}$
For each $x_i$ in $P^*$, compute weight $w_i = w(x_i)$
Initialize time step $z \leftarrow 1$, overall time $T \leftarrow 0$, and time spent $t_j \leftarrow 0$ for algorithm $a_j$
While $P^* \neq \emptyset$ and $T < B$
    Select $a_j$ with execution time $\Delta_z$ to maximize $N_j^z(t_j + \Delta_z)/\Delta_z$
    Remove from $P^*$ problems solved by $a_j$ during step $z$
    Schedule $a_j$ with execution time $\Delta_z$ in $S$
    Update times: $t_j \leftarrow t_j + \Delta_z$, $T \leftarrow T + \Delta_z$, and $z \leftarrow z + 1$
Return $S$

**Fig. 2.** High-level pseudocode for WG, a weighted greedy constructor for one processor.

On one processor, to schedule within time limit $B$ algorithms from $A$ for a problem $y$ given prior experience on a set of problems $P$, WG combines GASS and CPHYDRA into a single framework for switching scheduling. (See Figure 2.) WG is similar to GASS, except that it represents problems by numeric feature vectors, and restricts its attention to similar problems (i.e., reasons based only on similar cases). WG initially selects a *neighbor set* $P^*$ that is the $100r\%$ of the most similar training problems (i.e., have feature vectors closest in Euclidean distance to that of $y$), $0 < r \leq 1$. The influence of these problems in the selection of an algorithm may be uniform, or be weighted in proportion to their distance $d_i$ from $y$. The weight functions investigated here are shown in Table 1, where $d_{min}$ denotes the smallest distance from a neighbor set problem to $y$, and $d_{max}$ denotes the largest distance.

During each new interval $\Delta_z$, WG counts (from the performance matrix $\tau$) and weights how many training problems in the current neighbor set $P^*$ it could solve within time $t + \Delta_z$ if it assigned problem $x_i$ to algorithm $a_j$ during that interval:

$$N_j^z(t) = \sum_{x_i \in P^*} w_i \zeta_{ij}(t) \text{ where } \zeta_{ij}(t) = \begin{cases} 1 & \text{if } \tau_{ij} \leq t \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

WG then greedily maximizes (1) per unit of time expended, that is, it calculates

$$\operatorname{argmax}_{a_j, \Delta_z} \frac{N_j^z(t + \Delta_z)}{\Delta_z}$$

and removes those now-solved similar problems from $P^*$. The time complexity of WG is $O(rnm \log rn \cdot \min\{rn, Bm\})$ because it considers every algorithm $a_j$ with every interval length $\Delta_z$.

**Table 1.** Three weight functions that measure problem similarity, where $d_i$ denotes the Euclidean distance of problem $y$ from the $i$th neighbor set problem $x_i$. Here, $\varepsilon = 0.001$.

| Reciprocal weighting | Normalized weighting | Normalized-fixed weighting |
|---|---|---|
| $w_i = \frac{1}{1 + d_i}$ | $w_i = 1 - \frac{(n-1)(d_i - d_{\min})}{n(d_{\max} - d_{\min})}$ | $w_i = 1 - \frac{(1-\varepsilon)(d_i - d_{\min})}{d_{\max} - d_{\min}}$ |

## 6 Creation of Portfolios for Parallel Processing

An intuitive way to parallelize WG for $K$ identical processors $\pi_1, \pi_2, \ldots, \pi_K$ is to partition the similar training problems $P^*$ into $K$ subsets $P^1, P^2, \ldots, P^K$ at random, and then use WG to construct a schedule for processor $\pi_k$ based its corresponding subset $P^k$. We call this *RPWG* (randomized parallel WG). With uniform weights $w_i = 1$, RPWG is a naïve parallel version of GASS. (To reduce the impact of randomness, RPWG could construct such a partition $v$ times, although to conserve time $v = 1$ here.) Thus the overall complexity of RPWG is $O(vrnm \log(rn/K) \cdot \min\{rn/K, Bm\})$. Similarly, RP-CPHYDRA, the naïve parallel version of CPHYDRA, randomly partitions the similar training problems into $K$ subsets and then uses CPHYDRA on each subset to construct a schedule for each processor. Section 7 investigates both these naïve parallel constructors as baselines. (Other recent work relevant to parallel algorithm portfolios includes online learning [2, 16] and methods that split problems [29, 30].)

Effectively, the construction of a parallel algorithm portfolio to solve as many training problems as possible on $K$ processors is an integer-programming (*IP*) problem. The goal is to find the schedule $S$ that specifies the time allotments to all algorithms on all processors, such that no problem can receive more than $B$ time from all the processors together, and the total number of problems solved is a maximum. The expression $(1 - \zeta_{ij}(t_{kj}))$ is 1 if problem $x_i$ is unsolved by algorithm $a_j$ after time $t_{kj}$ allocated to it on $\pi_k$, and 0 otherwise. The product of $(1 - \zeta_{ij}(t_{kj}))$ over all $j$ and $k$ is 1 if problem $x_i$ is not solved by any algorithm on any processor in schedule $S_k$, and 0 otherwise. Thus the best schedule is

$$\arg\max_{S=\{S_1,\ldots,S_K\}} \sum_{i=1}^{n} [1 - \prod_{k=1}^{K} \prod_{j=1}^{m} (1 - \zeta_{ij}(t_{kj}))] \text{ such that } \sum_{j=1}^{m} t_{kj} \le B \text{ and } t_{kj} \ge 0 \quad (2)$$

Intuitively, when two schedules solve the same number of training problems, we would prefer the one that consumes less total time. Thus (2) becomes:

$$\arg\min_{S=\{S_1,\ldots,S_K\}} \left\{ (KB+1) \sum_{i=1}^{n} \prod_{k=1}^{K} \prod_{j=1}^{m} (1 - \zeta_{ij}(t_{kj})) + \sum_{k=1}^{K} \sum_{j=1}^{m} t_{kj} \right\} \text{ such that } \sum_{j=1}^{m} t_{kj} \le B \text{ and } t_{kj} \ge 0 \quad (3)$$

Expression (3) seeks to minimize the cost of schedule $S$, as measured by a penalty for unsolved problems (counted in the first sum) and the resources $t_{kj}$ allocated to all processors. Each unsolved problem incurs cost $KB + 1$, which is greater than all available time on all processors. This guarantees that any benefit introduced by reduction in overall runtime will be overshadowed by the penalty for solving one less problem. The optimization in (3) is NP-hard; others have proposed the use of column generation to solve a simpler IP problem for algorithm scheduling for non-parallel algorithm portfolios [28]. Instead here we adopt heuristics to generalize WG for this IP problem.

We argue that the optimal solution to (3) can occur only when there exists at most one processor $k$ for each algorithm $a_j$ such that $t_{kj} > 0$. For example, consider a schedule that allocates time $t_{1j}$ and $t_{2j}$ $(0 < t_{1j} < t_{2j})$ to the same algorithm on processors 1 and 2, respectively. These times are resources only, and are not directed to any particular problem or algorithm. Any problem solved by some algorithm on processor 1 in $t_{1j}$ can be solved by the same algorithm on processor 2 in $t_{2j}$. Removing the algorithm from processor 1 does not increase the number of unsolved training problems because

*Input*: training set $P = \{x_1, x_2, \ldots, x_n\}$, algorithms $A = \{a_1, a_2, \ldots, a_m\}$, time limit $B$, testing problem $y$, weight function $w$: $\Re^d \rightarrow \Re^d$, neighbor set ratio $r$, processors $\{\pi_1, \pi_2, \ldots, \pi_K\}$
*Output*: schedule $S = \{S_1, S_2, \ldots, S_K\}$ for a parallel switching algorithm portfolio

```
1  For i = 1 to n, compute Euclidean distance between x_i and y
2  P* ← {100r% of problems in P closest to y}
3  Compute weight w_i for each x_i in P* with w
4  Initialize time step z ← 1, overall time T^u ← 0 on processor π_u,
   time t_uj ← 0 for a_j on π_u
5  While P* ≠ Ø and T^u < B for at least one u
6    Select a_j on π_u with time Δ_z to maximize N_j^z(t_j + Δ_z)/ Δ_z      ** Retain **
7    Remove from P* problems solved by a_j during step z
8    Schedule a_j with execution time Δ_z on π_u
9    Update times: t_uj ← t_uj + Δ_z, T^u ← T^u + Δ_z, and z ← z + 1
10 For each π_u where T^u < B
11   If T^u = 0                                                           ** Spread **
12   then assign a_j to π_u for B, where a_j solves the most problems in P and a_j ∉ S
13        update times: t_uj ← B, T^u ← B, and z ← z + 1
14   else π_u executes the first algorithm placed on π_u until B          ** Return **
15        update times: t_uj ← t_uj + (B – T^u), T^u ← B, and z ← z + 1
16 Return S
```

**Fig. 3.** High-level pseudocode for RSR-WG, a weighted greedy algorithm that constructs a parallel switching schedule with heuristics Retain, Spread, and Return.

the same problems will be solved on processor 2, but it does reduce the total runtime, and produces a better schedule.

Inspired by this argument, Figure 3 introduces *RSR-WG* for parallel algorithm portfolios, where RSR stands for three heuristics: Retain, Spread, and Return. Like WG, RSR-WG selects an initial set of similar training problems and tries to schedule greedily, but with modifications from our three heuristics. *Retain* (line 6) places algorithm $a_j$ on processor $\pi_u$ if that placement will maximize equation (1) per unit of expended time and $\pi_u$ still has time available ($T^u < B$). Among such processors, Retain prefers one that has already hosted $y$ before ($t_{uj} \neq 0$), and otherwise selects one that has thus far been used the least (i.e., has minimum $T^u$). If a parallel schedule $S$ solves all training problems without making full use of all the processors, *Spread* (line 11) places the algorithm $a_j$ that solves the most problems in $P$ but does not appear in $S$ on a processor that was idle throughout $S$ (if one exists), breaking ties at random. (The rationale here is that $a_j$ may be generally effective but not outstanding on $y$.) Finally, if a processor is not fully used in $S$ (i.e., $T^u < B$), *Return* (line 14) places the first algorithm it executed on that processor until the time limit. Obviously, RSR-WG achieves the performance of an oracle when $K = m$, but it is also effective when $K$ is relatively small compared to $m$, as demonstrated in the next section.

## 7 Experimental Design and Results

We compared the performance of parallel algorithm portfolios from three constructors to that of four non-parallel solvers on problems from the Third International CSP solver competition (*CPAI'08*). To extract the 36 features values (e.g., number of variables, maximum domain size) used by CPHYDRA and RSR-WG, we ran the CSP solver Mistral 1.550 ([31]). For feature extraction we allotted 1 second on an 8 GB Mac Pro with a 2.93 GHz Quad-Core Intel Xeon processor.

CPAI'08 included 3307 problems in 5 categories. Some solvers could not address problems in every category; we merged the 2-ARY-INT and *N*-ARY-INT ($N > 2$) categories because the same solvers addressed both. Because our experiments count solved problems (those where a solver finds a solution or proves that none exists), we excluded any problem that was not solved by any solver within the CPAI'08 time limit of 1800 seconds. If CPHYDRA does not extract features quickly enough, it simply splits its schedule evenly among its three algorithms. Rather than test portfolios' luck with an algorithm this way (and penalize a portfolio with more algorithms at its disposal), we chose to exclude such problems. Table 2 summarizes the remaining 2865 problems in 4 categories.

Stratified partitioning was used in all runs, to maintain the proportions of problems from different categories in each subset. Table 3 reports the performance, in number of problems solved within 1800 seconds each, of an oracle and three non-parallel algorithm portfolio constructors as baselines: CPHYDRA_k_10, CPHYDRA_k_40, and GASS. The data for GASS was obtained by 10-fold cross-validation with stratified partitioning on the 2865 problems.

All portfolio construction experiments ran under 10-fold cross-validation on a Dell PowerEdge 1850 cluster with one head node and 86 compute nodes, each with four Intel 2.80 GHz Woodcrest dual-core processors. RSR-WG results reported here are for portfolio construction (i.e., scheduling) time plus runtime. The runtimes of RPWG and RP-CPHYDRA did not include portfolio construction time, which gave them a slight advantage. In extensive testing, uniform weighting and the three weight functions in Table 1 produced slightly different performance improvements in RSR-WG, but no one statistically significantly outperformed the others consistently. Thus this paper reports only on the normalized-fixed weight function.

In CPAI'08, CPHYDRA chose 10 or 40 similar problems from which to learn, so here RP-CPHYDRA selects $10*K$ neighbors, randomly distributes them to $K$ processors, and executes a complete search for the optimal schedule on each processor. RP-CPHYDRA's portfolio construction time was limited to 180 seconds. If it did not produce the optimal schedule in that time, the best schedule found so far was used. To

**Table 2.** Competition problems by category. Experiment problems were those for which at least one solver found a solution or showed that none existed, and also had features extractable within one second. Solvable problems had at least one solution.

| Applicable solvers | Category | Competition problems | Experiment problems | Experiment solvable problems |
|---|---|---|---|---|
| 17 | GLOBAL | 556 | 493 | 256 |
| 22 | *k*-ARY-INT ($k \geq 2$) | 1412 | 1303 | 739 |
| 23 | 2-ARY-EXT | 635 | 620 | 301 |
| 24 | N-ARY-EXT ($N>2$) | 704 | 449 | 156 |

**Table 3.** Benchmark results for the 3rd International CSP solver competition.

| Solver | Oracle | GASS | CPHYDRA_k_10 | CPHYDRA_k_40 |
|---|---|---|---|---|
| Number solved | 2865 | 2773 | 2577 | 2573 |
| % solved | 100% | 96.79% | 89.95% | 89.81% |

reduce search time, any algorithm dominated by another algorithm (i.e., always outperformed by it on all 2865 problems) was also eliminated from RP-CPHYDRA's consideration. RP-CPHYDRA also scaled all schedules (as discussed in Section 8) to exploit the full time limit $B$.

Table 4 compares the performance of parallel portfolios from three constructors: RP-CPHYDRA (the parallel version of CPHYDRA), RPWG (the naïve parallel version of GASS), and RSR-WG. It lists the total number of problems (out of 2865) solved by each constructor's portfolios, and flags experiments where RSR-WG portfolios were statistically significantly better ($p < 0.005$) than those of a naïve parallel constructor.

For RSR-WG we simulated all 24 solvers from the original competition [23]. For RSR-WG only, we tested as many as $K = 16$ processors. Both $K = 8$ and $K = 16$ produced near-oracle performance; indeed, 2 out of 10 runs for $K = 16$ were perfect. Execution of RSR-WG on $K = 16$ processors is a reasonable approach for modern computers, where it would produce portfolios able to solve only one fewer problem than an oracle. (Execution of RSR-WG on one computer with multiple cores could degrade performance, for example, due to overhead introduced by memory sharing.)

One important question is the number of training problems to use for CBR, as measured by the *neighbor set ratio* (# problems / # training problems). For $K$ from 1 to 16 we tested neighbor set ratios of 0.005 to 0.16, which yield neighbor sets that range in size from 14 to 458, respectively. Table 5 reports on how many problems (out of 2865) RSR-WG solved, and shows how the neighbor set ratio impacts performance under different numbers of processors $K$. Boldface entries in Tables 4 and 5 indicate the best performance for each $K$. Clearly RSR-WG efficiently generates effective algorithm portfolios, and does best with small neighbor set ratios for $K > 1$. On $K = 1$, RSR-WG outperforms GASS, CPHYDRA_k_10, and CPHYDRA_k_40.

**Table 4.** Performance of 3 parallel portfolio constructors on 2865 problems, with best value for $K$ processors in boldface. * means RSR-WG outperformed RPWG; † means RSR-WG outperformed RP-CPHYDRA.

| $K$ | RP-CPHYDRA | Neighbor set ratio | | | | | |
|---|---|---|---|---|---|---|---|
| | | 0.005 | | 0.01 | | 0.02 | |
| | | RPWG | RSR-WG | RPWG | RSR-WG | RPWG | RSR-WG |
| 1 | 2779 | 2771 | 2773 | 2778 | 2779 | **2787** | 2786† |
| 2 | 2807 | 2801 | **2826*** | 2799 | 2821* | 2802 | 2823*† |
| 3 | 2817 | 2808 | **2841***† | 2810 | 2836*† | 2808 | 2839*† |
| 4 | 2827 | 2810 | **2850***† | 2812 | 2847*† | 2811 | 2847*† |
| 5 | 2830 | 2817 | **2855***† | 2819 | 2851*† | 2816 | 2852*† |
| 6 | 2831 | 2821 | **2857***† | 2818 | 2855*† | 2819 | 2856*† |
| 7 | 2834 | 2823 | **2858***† | 2823 | **2858***† | 2824 | 2857*† |
| 8 | 2834 | 2825 | 2859*† | 2825 | **2860***† | 2825 | 2858*† |

**Table 5.** Mean and standard deviation for the number of problems solved by RSR-WG out of 2865, with normalized-fixed weight function over 10 runs with $K$ processors. Best value for $K$ processors is in boldface

| $K$ | Neighbor set ratio | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0.005 | | 0.01 | | 0.02 | | 0.04 | | 0.08 | | 0.16 | |
| 1 | 2773 | 3.65 | 2779 | 3.20 | 2786 | 2.30 | **2789** | 3.17 | 2788 | 3.09 | **2789** | 2.51 |
| 2 | **2826** | 3.51 | 2821 | 2.49 | 2823 | 3.16 | 2816 | 2.97 | 2810 | 2.99 | 2809 | 2.87 |
| 3 | **2841** | 2.12 | 2836 | 1.93 | 2839 | 2.56 | 2832 | 2.07 | 2827 | 2.27 | 2819 | 2.07 |
| 4 | **2850** | 2.15 | 2847 | 1.57 | 2847 | 2.63 | 2843 | 2.06 | 2838 | 2.22 | 2832 | 2.50 |
| 5 | **2855** | 1.37 | 2851 | 2.35 | 2852 | 0.88 | 2850 | 1.78 | 2845 | 2.72 | 2843 | 3.26 |
| 6 | **2857** | 0.95 | 2855 | 1.07 | 2856 | 1.26 | 2853 | 1.64 | 2851 | 1.03 | 2850 | 1.07 |
| 7 | **2858** | 0.79 | **2858** | 0.57 | 2857 | 0.82 | 2855 | 1.83 | 2854 | 2.35 | 2854 | 1.14 |
| 8 | 2859 | 1.18 | **2860** | 1.34 | 2858 | 1.06 | 2858 | 1.18 | 2856 | 0.74 | 2855 | 1.43 |
| 16 | **2864** | 0.42 | **2864** | 0.00 | **2864** | 0.00 | 2863 | 0.00 | 2861 | 0.42 | 2861 | 0.47 |

Finally, Figure 4 compares the runtimes of an oracle solver and RSR-WG in one run with neighbor set ratio 0.005 and weight function normalized-fixed. (Again, RSR-WG's time includes both portfolio construction and search.) As in [23], each plus sign represents one of the 2865 problems. Those at the far right correspond to problems that went unsolved by RSR-WG in 1800 seconds. Those on the diagonal correspond to problems that were solved by RSR-WG as quickly as an oracle would have solved them. Clearly, more processors reduced the number of unsolved problems (from 90 to 6 in this particular run) and solved more problems as quickly as an oracle.

## 8 Discussion

As indicated above, the 1800-second runtime per problem for RSR-WG in these experiments includes the time to extract features, construct the schedule, and to execute it. RSR-WG adopts a greedy approach that dramatically reduces its scheduling time but still generates effective portfolios. For example, over 10 runs the average scheduling time of RSR-WG for $K = 8$ processors ranged from 14.56 to 14.96 seconds ($\sigma$ in [6.05, 6.35]) with normalized-fixed weights and a neighbor set ratio of 0.16. For $K = 1$ processor under the same conditions, average scheduling time ranged from 14.30 to 14.80 seconds ($\sigma$ in [5.75, 6.15]). These are small but statistically significant differences. In contrast, RP-CPHYDRA sometimes failed to compute an optimal schedule within 180 seconds. When $K = 1$, CPHYDRA failed to compute an optimal schedule 4.81% of the time. When $K > 1$, CPHYDRA must construct a schedule for each processor, on training sets that may be considerably more diverse. This can increase the search effort; indeed, for $K = 8$, CPHYDRA failed to compute an optimal schedule 14.39% of the time. As for GASS, because it learns on all the training problems, it required more than 5 days of execution time for its single entry in Table 2.

Instead of Spread, one might *scale* $S$ to extend it to the entire time limit $B$, that is, allocate $B$ to algorithms proportionally to their runtimes in $S$. CPHYDRA adopted scaling, and so did RP-CPHYDRA in our experiments. Scaling, however, would be unwise in RSR-WG because the earliest designated algorithms might be both most promising

and quick, in which case they would only be allotted relatively short time intervals $\Delta_z$. Whether or not scaling is appropriate, we believe, is probably determined by the problem set. The Return heuristic succeeds, we suspect, because as $K$ approaches $m$ it is better to allot larger time intervals to an algorithm on a single processor.

We temper the results on $K = 16$ with the observation that it is very nearly a race, when the problems in the neighbor set are sufficiently descriptive to eliminate the poorest performers on $y$. We prefer to consider the near-optimal performance for $K =$

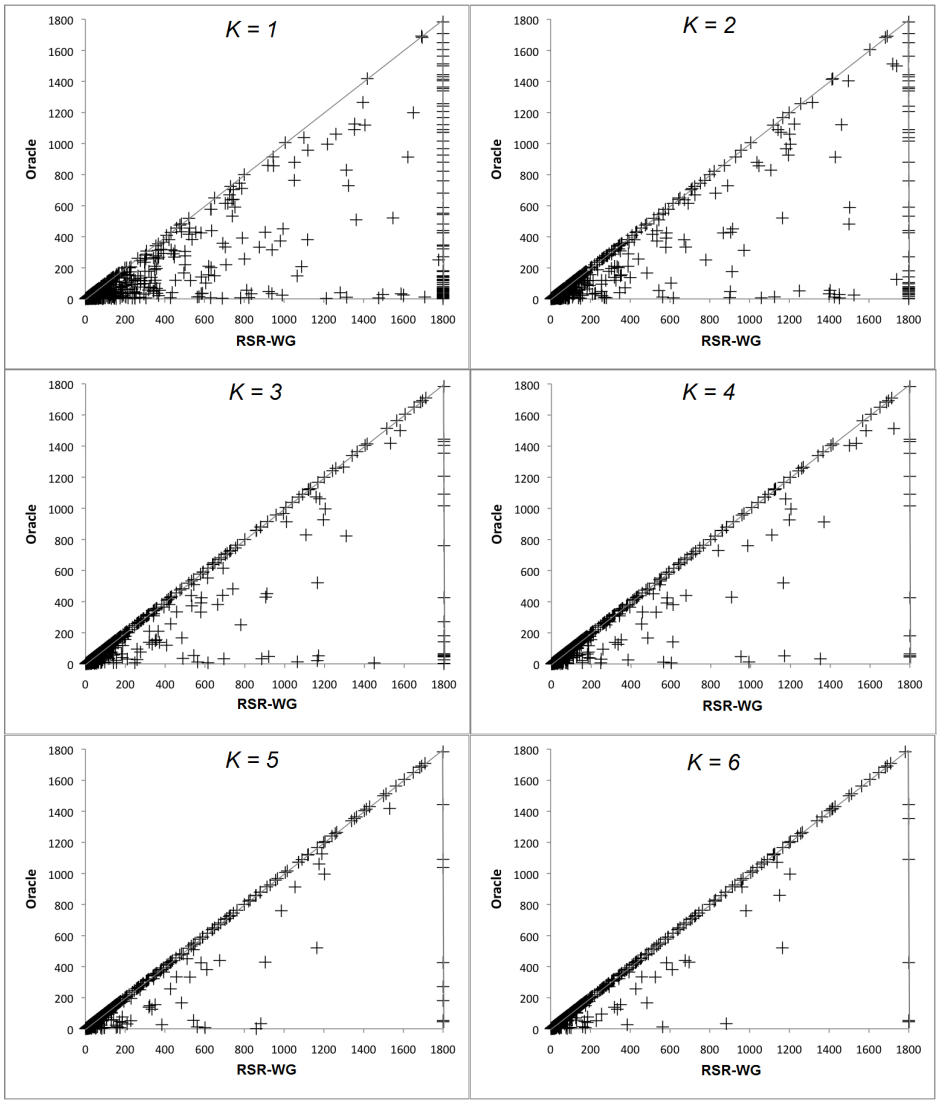


**Fig. 4.** Comparison of (ideal) oracle runtime (*y-axis*) to RSR-WG's time (*x-axis*) for 1 run with weight function normalized-fixed and neighbor set ratio 0.005. Each + denotes a result on one of the 2865 problems. Number of processors $K$ ranges from 1 to 6.

8, and even $K = 4$, and to remember that RSR-WG was charged for scheduling time, while its competitor constructors were not.

Coarser granularity (indicated by a smaller $B$, which allocates longer intervals) impacts the scheduling efficiency of RSR-WG, but the effectiveness of the resultant portfolio depends on the performance matrix entries for the neighbors of the testing problem. A smaller $B$ does not necessarily reduce the effectiveness of the resultant algorithm portfolio; if that were the case, a switching (or scheduling) portfolio would always be superior to algorithm selection. In addition to Table 5, where $B = 1800$, we tested RSR-WG with $B = 20$, 10, 5, 4, 3, 2, and 1. (This is equivalent to time allocations that, instead of 1 second on a processor, are 90, 180, 360, 450, 600, 900, or 1800 seconds. Note that $B = 1$ is equivalent to racing one algorithm on each processor to address a problem.) In these granularity experiments, for $K = 1$ the number of solved problems peaked at $B = 10$. For $1 < K \leq 8$, no coarser granularity ever showed a significant improvement; indeed, performance degraded slightly as $B$ decreased. Both improvement on $K = 1$ and failure to improve when $K > 1$ were consistent across all neighbor set ratios reported here, with peaks at either $B = 5$ or $B = 10$ when $K = 1$.

The success of RSR-WG algorithm portfolios relies heavily on the diversity of the performance of its constituent algorithms and the relevance of the extracted features. Typically, algorithm portfolio constructors select their algorithms and features based upon domain-specific knowledge. The reader may, for example, wonder how RSR-WG would perform if it relied on the three solvers CPHYDRA used in CPAI'08. The difficulty here is that CPHYDRA included solvers from the 2006 competition, solvers that did not enter CPAI'08, and whose performance was therefore unavailable on the 2008 problems. Although algorithm choice based on domain knowledge and feature selection can further enhance a portfolio's performance, it could also make it vulnerable to overfitting. When the number of features is larger, feature selection can be of considerable benefit to an algorithm portfolio constructor [13, 14], and we intend to explore it in future work.

Current work is proceeding in several directions. In practice, many algorithms may perform differently on the same problem in different runs, but still exhibit a certain level of consistency [3]. Indeed, in (sequential) CSP solver competitions, solvers typically fix their parameter values and introduce relatively little randomness to achieve stable performance. In that case, with coarse granularity (e.g., $B = 10$), a solver's performance is nearly deterministic. Greater randomness, however, could change solvers' performance dramatically, and thereby potentially benefit parallel constraint solving. A generalization of RSR-WG is in process to handle such behavior. On the other hand, automatic parameter tuning could introduce much diversity, and should fare well in algorithm portfolios [32]. Specifically, one may view different configurations of an algorithm as different algorithms, and thereby combine parameter tuning and an algorithm portfolio in the same framework. We are pursuing this avenue as well.

The performance of any algorithm portfolio is, of course, bounded by that of an oracle. The combination of algorithms as black boxes eliminates any opportunity to improve an individual algorithm. In contrast, parallelism can be achieved by a variety of problem decomposition methods (e.g., search space splitting), as discussed in Section 3. Although the results of recent SAT solver competitions suggest that a well-designed algorithm portfolio outperforms decomposition methods on a small number

of processors [22], decomposition methods have shown their potential on many more processors (e.g., 64 cores or more in [19]). We will explore this in future work.

## 9 Conclusions

This paper presents WG, a constructor for non-parallel algorithm portfolios based on case-based reasoning and a greedy algorithm. It formulates parallel algorithm portfolio construction as an integer-programming problem, and generalizes WG to RSR-WG, a constructor for parallel algorithm portfolios based on a property of the optimal solution to the inherent integer-programming problem. To address a set of problems one at a time, RSR-WG creates portfolios of deterministic algorithms offline. Experiments show that the parallel algorithm portfolios produced by RSR-WG are statistically significantly better than those produced by naïve parallel versions of popular portfolio constructors. Moreover, with only a few additional processors, RSR-WG portfolios are competitive with an oracle solver on a single processor.

**Acknowledgements.** This research was supported in part by the National Science Foundation under grants IIS-0811437, CNS-0958379 and CNS-0855217, and the City University of New York High Performance Computing Center.

## References

1. Gebruers, C., Hnich, B., Bridge, D. Freuder, E.: Using CBR to Select Solution Strategies in Constraint Programming. In: Third International Conference on Case-based Reasoning, pp. 222-236. (2005)
2. Gagliolo, M. Schmidhuber, J.: Learning Dynamic Algorithm Portfolios. Annals of Mathematics and Artificial Intelligence. 47(3), 295-328 (2006)
3. Silverthorn, B. Miikkulainen, R.: Latent Class Models for Algorithm Portfolio Methods. In: Twenty-Fourth AAAI Conference on Artificial Intelligence, pp. 167-172. (2010)
4. Stern, D., Herbrich, R., Graepel, T., Samulowitz, H., Pulina, L. Tacchella, A.: Collaborative Expert Portfolio Management. In: Twenty-Fourth AAAI Conference on Artificial Intelligence, pp. 179-184. (2010)
5. Xu, L., Hutter, F., Hoos, H.H. Leyton-Brown, K.: The Design and Analysis of an Algorithm Portfolio for SAT. In: 13th International Conference on Principles and Practice of Constraint Programming, LNCS 4741, pp. 712-727. Springer. (2007)
6. O'Mahony, E., Hebrard, E., Holland, A., Nugent, C. O'Sullivan, B.: Using Case-Based Reasoning in an Algorithm Portfolio for Constraint Solving. In: Nineteenth Irish Conference on Artificial Intelligence and Cognitive Science. (2008)
7. Streeter, M., Golovin, D. Smith, S.F.: Combing Multiple Heuristics Online. In: the Twentysecond National Conference on Artificial Intelligence, pp. 1197-1203. (2007)
8. Gomes, C., Selman, B. Crato, N.: Heavy-Tail Distributions in Combinatorial Search. In: Third International Conference on Principles and Practice of Constraint Programming, LNCS 1330, pp. 121-135. Springer. (1997)
9. Huberman, B., Lukose, R. Hogg, T.: An Economics Approach to Hard Computational Problems. Science. 256, 51-54 (1997)

10. Gomes, C. Selman, B.: Algorithm Portfolio Design: Theory vs. Practice. In: Thirteenth Conference On Uncertainty in Artificial Intelligence, pp. 190-197. Morgan Kaufmann. (1997)
11. Guerri, A. Milano, M.: Learning Techniques for Automatic Algorithm Portfolio Selection. In: Sixteenth European Conference on Artificial Intelligence, pp. 475-479. (2004)
12. Xu, L., Hoos, H.H. Leyton-Brown, K.: Hydra: Automatically Configuring Algorithms for Portfolio-Based Selection. In: Twenty-Fourth AAAI Conference on Artificial Intelligence, pp. 179-184. (2010)
13. Xu, L., Hutter, F., Hoos, H.H. Leyton-Brown, K.: SATzilla: Portfolio-Based Algorithm Selection for SAT. Journal of Artificial Intelligence Research. 32, 565-606 (2008)
14. Horvitz, E., Ruan, Y., Gomes, C.P., Kautz, H.A., Selman, B. Chickering, D.M.: A Bayesian Approach to Tackling Hard Computational Problems. In: Seventeenth Conference in Uncertainty in Artificial Intelligence, pp. 235-244. Morgan Kaufmann Publishers Inc., 720234 (2001)
15. Rice, J.R.: The Algorithm Selection Algorithm. Advances in Computers. 15, 65-118 (1976)
16. Gagliolo, M. Schmidhuber, J.: Towards Distributed Algorithm Portoflios. In: International Symposium on Distributed Computing and Artificial Intelligence, pp. 634-643. (2008)
17. Carchrae, T. Beck, J.C.: Low-Knowledge Algorithm Control. In: Nineteenth National Conference on Artificial Intelligence, Sixteenth Conference on Innovative Applications of Artificial Intelligence, pp. 49-54. AAAI Press / The MIT Press, 1597158 (2004)
18. Carchrae, T. Beck, J.C.: Applying Machine Learning to Low-Knowledge Control of Optimization Algorithms. Computational Intelligence. 21(4), 372-387 (2005)
19. Bordeaux, L., Hamadi, Y. Samulowitz, H.: Experiments with Massively Parallel Constraint Solving. In: Twenty-First International Joint Conference on Artificial Intelligence, pp. 443-448. Morgan Kaufmann Publishers Inc., 1661516 (2009)
20. Singer, D. Monnet, A.: Jack-SAT: A New Parallel Scheme to Solve the Satisfiability Problem (SAT) Based on Join-and-Check. In: Seventh International Conference on Parallel Processing and Applied Mathematics (PPAM), pp. 249-258. Springer-Verlag. (2007)
21. Li, W. van Beek, P.: Guiding Real-World SAT Solving with Dynamic Hypergraph Separator Decomposition. In: Sixteenth IEEE International Conference on Tools with Artificial Intelligence, pp. 542-548. (2004)
22. Hamadi, Y. Sais, L.: ManySAT: A Parallel SAT Solver. Journal on Satisfiability, Boolean Modeling and Computation. 6, 245-262 (2009)
23. CPAI'08, http://www.cril.univ-artois.fr/CPAI08/
24. Fourth International CSP Solver Competition: http://www.cril.univ-artois.fr/CSC09/
25. The SAT 2007 Competition: satcompetition.org/2007/rules07.html.
26. Dietterich, T.G.: Ensemble Methods in Machine Learning. In: the First International Workshop on Multiple Classifier Systems, pp. 1-15. (2000)
27. Streeter, M., Golovin, D. Smith, S.F.: Restart Schedules for Ensembles of Problem Instances. In: the Twentysecond National Conference on Artificial Intelligence, pp. 1204-1210. (2007)
28. Kadioglu, S., Malitsky, Y., Sabharwal, A., Samulowitz, H. Sellmann, M.: Algorithm Selection and Scheduling. In: 17th International Conference on Principles and Practice of Constraint Programming, LNCS 6876, pp. 454-469. (2011)
29. Segre, A.M., Forman, S., Resta, G. Wildenberg, A.: Nagging: A Scalable Fault-Tolerant Paradigm for Distributed Search. Artificial Intelligence. 140, 71-106 (2002)
30. Vander-Swalmen, P., Dequen, G. Krajecki, M.: A Collaborative Approach for Multi-Threaded SAT Solving. International Journal of Parallel Programming. 37, 324-342 (2009)
31. Mistral. 4c.ucc.ie/~ehebrard/Software.html.
32. Hutter, F., Hoos, H.H., Leyton-Brown, K. Stützle, T.: Paramils: An Automatic Algorithm Configuration Framework. Journal of Artificial Intelligence Research. 36, 267–306 (2009)