Combining Multiple Constraint Solvers: Results on the CPAI'06 Competition Data

Matthew Streeter[1] Daniel Golovin[1] Stephen F. Smith[2]

Computer Science Department[1] and
The Robotics Institute[2]
Carnegie Mellon University
Pittsburgh, PA 15213
{matts,dgolovin,sfs}@cs.cmu.edu

Abstract. In a recent paper [5], we presented an algorithm that constructs a schedule for interleaving the execution of two or more solvers, with the goal of obtaining improved average-case running time relative to the fastest individual solver. In this paper, we evaluate this algorithm experimentally using data from the CPAI'06 constraint solver competition.

1 Introduction

Many computational problems that arise in practice are NP-hard and thus are unlikely to admit algorithms with provably good worst-case performance. These problems must nevertheless be solved, and in many problem domains heuristics have been developed that perform much better in practice than a worst-case analysis would guarantee. Unfortunately, the behavior of a heuristic on a previously unseen problem instance can be difficult to predict in advance, and the running times of two different heuristics on the same instance can easily differ by orders of magnitude. For this reason, if a heuristic has been running unsuccessfully for some time it may be worthwhile to suspend the execution of that heuristic and start running a different heuristic instead.

Table 1. Behavior of two solvers on three instances from the CPAI'06 competition.

Instance	`BPrologCSPSolver70a` CPU (s)	`Abscon 109 ESAC` CPU (s)
allIntervalSeries/series-10	0.021	0.72
fisher/FISCHER1-1-fair	0.046	≥ 1800
pseudoSeries/aim/aim-100-1-6-1	≥ 1800	1.089

The potential reduction in average-case running time that can be achieved by interleaving the execution of multiple heuristics is illustrated in Table 1. Here, although both solvers take > 600 seconds on average, a schedule that simply ran the two solvers in parallel would take less than one second on average.

In this paper, we seek to improve the average-case performance of constraint solvers by interleaving the execution of multiple (currently available) constraint solvers according to a *task-switching schedule*. We construct task-switching schedules using a recently-developed algorithm [5] and evaluate their performance using data from the CPAI'06 competition.

1.1 Task-switching schedules

Let $\mathcal{H} = \{h_1, h_2, \ldots, h_k\}$ be a set of deterministic heuristics, and let $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$ be a set of instances of some decision problem. Heuristic h_j, when run on instance x_i, runs for $\tau_{i,j}$ time units before returning a (provably correct) "yes" or "no" answer. A *task-switching schedule* $S : \mathbb{Z}_+ \rightarrow \mathcal{H}$ specifies, for each integer $t \geq 0$, the heuristic $S(t)$ to run from time t to time $t+1$. For example, to execute the task-switching schedule depicted in Figure 1 we would run h_1 for two time units; then run h_2 for two time units, then run h_1 for four additional time units, and so on.

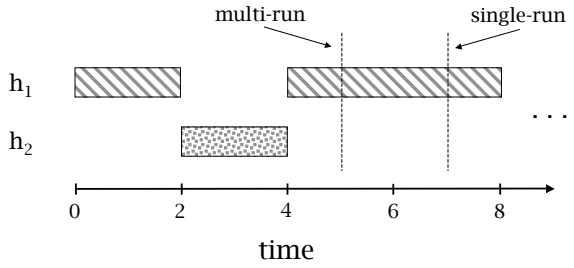

Fig. 1. A task-switching schedule.

A task-switching schedule may either be executed in *single-run mode* or in *multi-run mode*. The two modes differ in what happens to the heuristic (call it h) that is currently running when the task-switching schedule starts running a new heuristic h': in single-run mode, the current run of h is discarded, while in multi-run mode the execution state of h is saved and will be restored if h is run again. For any schedule S, let $c_i^s(S)$ denote the time S takes to solve x_i when S is executed in single-run mode and let $c_i^m(S)$ denote the time it takes when executed in multi-run mode.[1] For example, if heuristics h_1 and h_2 both require 3 time units to solve instance x_i (i.e., $\tau_{i,1} = \tau_{i,2} = 3$), the task-switching schedule S depicted in Figure 1 will require 5 time units to solve x if it is executed in multi-run mode but will require 7 time units if it is executed in single-run mode (i.e., $c_i^m(S) = 5$ and $c_i^s(S) = 7$).

We now consider the problem of computing a good task-switching schedule. That is, given as input the matrix τ, we would like to compute a schedule

[1] Formally, $c_i^m(S)$ is the smallest integer t such that for some heuristic h_j, $|\{t' < t : S(t') = h_j\}| = \tau_{i,j}$. Similarly, $c_i^s(S)$ is the smallest integer t such that, for some heuristic h_j, $S(t-\tau_{i,j}) = S(t-\tau_{i,j}+1) = S(t-\tau_{i,j}+2) = \ldots = S(t-1) = h_j$.

that minimizes $\sum_{i=1}^{n} c_i^s(S)$ (or $\sum_{i=1}^{n} c_i^m(S)$). Of course, we would not use the resulting task-switching schedule to solve instances in $\mathcal{X}$ (which we must already have solved in order to fill in the table τ). Rather, we would hope that a task-switching schedule that performs well on the instances in $\mathcal{X}$ would also perform well on similar problem instances, which we would be able to solve more quickly via the task-switching schedule.

Unfortunately, it is NP-hard to compute even an approximately optimal task-switching schedule. This follows from the fact that the problem of computing an optimal task-switching schedule generalizes *min-sum set cover*. Feige et al. [1] showed that it is NP-hard to approximate min-sum set cover within a factor of α for any $\alpha < 4$, and gave a greedy algorithm that achieves the optimal approximation ratio of 4. In [5], we showed how to generalize this greedy algorithm to obtain a 4-approximation to the optimal task-switching schedule. Our results are summarized in the following theorem.

Theorem 1. *Let* $C^* = \min_S \sum_{i=1}^{n} c_i^m(S)$. *There exists a poly-time greedy approximation algorithm that returns a schedule* S^m *such that* $\sum_{i=1}^{n} c_i^m(S^m) \leq 4C^*$. *A different greedy approximation algorithm returns a schedule* S^s *such that* $\sum_{i=1}^{n} c_i^s(S^s) \leq 4C^*$.

In [5] we also derived bounds on the number of training instances required in order to PAC-learn an optimal (or approximately optimal) schedule for instances drawn independently from a probability distribution. We also developed an online algorithm that receives a sequence $\langle x_1, x_2, \ldots, x_n \rangle$ of problem instances one at a time, and solves each instance (via a task-switching schedule) before moving on to the next.

1.2 Related work

Our work is closely related to previous work on *algorithm portfolios* [2, 3]. An algorithm portfolio consists of a set of heuristics that are run in parallel (or interleaved on a single processor) according to some schedule. The schedules considered in previous work simply run each heuristic in parallel at equal strength and assign each heuristic a fixed restart threshold. The term "algorithm portfolio" has also been used to describe algorithms such as SATzilla [6], which use machine learning to attempt predict which heuristic will solve a given instance the fastest and then run that heuristic exclusively.

Task-switching schedules were introduced in a recent paper by Sayag et al. [4], who gave an exact algorithm for computing an optimal task-switching schedule (as already mentioned, doing so is NP-hard, and the running time of their algorithm is exponential in the number of heuristics). For a more detailed discussion of related work, see [5].

2 Results

In this section, we use the greedy algorithms alluded to in Theorem 1 to construct task-switching schedules for interleaving solvers from the CPAI'06 competition.

To do so, we used the data available on the competition web site[2] to determine the running time of each constraint solver on each benchmark instance. If a solver did not return a solution within the half hour time limit, we artificially set its running time equal to half an hour. We used this data as input to our greedy approximation algorithms. Note that in performing these experiments, we did not actually run any of the constraint solvers.

One might worry that task-switching schedules computed in this way are highly tuned to the specific benchmark instances that were used in the competition. To address this concern, we evaluate our task-switching schedules using leave-one-out cross-validation.

The instances in the CPAI'06 competition were divided into five categories: 2-ARY-EXT, 2-ARY-INT, GLOBAL, N-ARY-EXT, and N-ARY-INT. We performed separate experiments on the instances in each category. We present the results for the category N-ARY-INT in detail, then summarize the results for the other four categories.

2.1 Results for category N-ARY-INT

The CPAI'06 competition included 925 instances in the N-ARY-INT category. Of the 14 solvers that were run on these instances, two produced incorrect answers for one or more instances and were excluded from the competition. 726 of the 925 instances were solved by at least one of the 12 remaining solvers within the half hour time limit. We use these 726 instances and these 12 solvers in our experiments.

Table 2 displays the number of instances solved within the half hour time limit as well as the average CPU time for each of the 12 solvers as well as four schedules: $Greedy^m$, $Greedy^s$, $Parallel^m$, and $Parallel^s$. $Greedy^m$ is the schedule S^m from Theorem 1 executed in multi-run mode, and similarly $Greedy^s$ is the schedule S^s from Theorem 1 executed in single-run mode. $Parallel^m$ is a schedule that runs all 12 heuristics in parallel, each at equal strength. $Parallel^s$ is a single-run version of $Parallel^m$ which first runs each heuristic for 1 second, then runs each heuristic for 2 seconds, then runs each heuristic for 4 seconds, and so on.

As shown in Table 2, the two greedy schedules outperform each of the 12 original solvers as well as the two parallel schedules, both in terms of average CPU time and in terms of the number of instances solved within the half hour time limit. Note that the results listed for the schedules executed in multi-run mode are optimistic in that they assume there is no overhead associated with keeping multiple runs in memory; however there is no such issue with the schedules executed in single-run mode. Also note that because we artificially set a solver's CPU time equal to the half hour time limit for instances it did not solve, the values for the average CPU time of the 12 heuristics are actually lower bounds, and using the (unknown) actual values could significantly improve our results. Figure 2 illustrates the task-switching schedule $Greedy^s$.

[2] `http://www.cril.univ-artois.fr/CPAI06/`

Table 2. Results for category N-ARY-INT (cross-validation results are parenthesized).

Solver	Num. solved	Avg. CPU (s)
$Greedy^m$	706 (701)	338 (407)
$Greedy^s$	631 (625)	395 (498)
$Parallel^m$	630	2460
$Parallel^s$	614	4896
`BPrologCSPSolver70a`	579	636
`Abscon 109 ESAC`	509	614
`Abscon 109 AC`	490	659
`sugar`	431	766
`CSPtoSAT+minisat`	395	888
`CSP4J - MAC`	370	963
`CSP4J - Combo`	364	998
`galac`	352	990
`galacJ`	331	1043
`Tramontane`	313	1075
`Mistral`	304	1103
`sat4jCSP`	228	1264

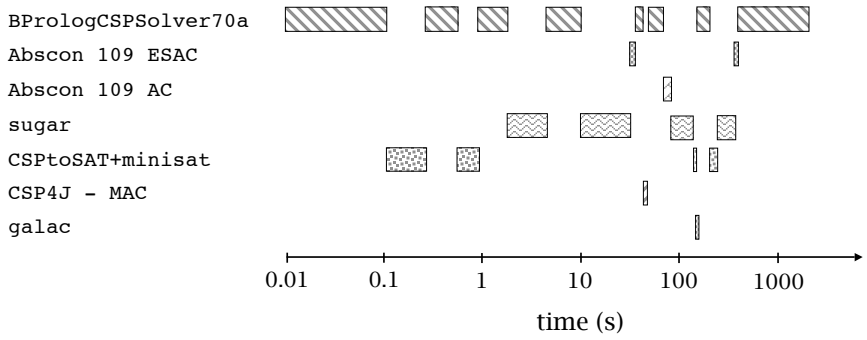

Fig. 2. The task-switching schedule $Greedy^s$.

To address the possibility of overfitting, we evaluated the task-switching schedules returned by the greedy algorithm using leave-one-out cross-validation.[3] The cross-validation results appear in parentheses in Table 2. The number of instances solved by $Greedy^s$ decreased by about 1% under cross-validation, while the average CPU time increased by about 26%. The results for $Greedy^m$ were similar.

[3] Leave-one-out cross-validation is performed as follows: for each instance, we remove that instance from the matrix τ and run the greedy algorithm on the remaining data to obtain a schedule to use in solving that instance.

2.2 Summary of results for all categories

We performed similar experiments on the instances in the four remaining categories: 2-ARY-EXT, 2-ARY-INT, GLOBAL, and N-ARY-EXT. In each experiment, we removed solvers that produced an incorrect answer on one or more instances, and we removed instances that none of the solvers could solve within the half hour time limit.

The results for all five instance categories are summarized in Table 3. In four out of five categories, the two greedy schedules outperform the corresponding parallel schedules and the best individual solver in terms of the number of instances solved within the time limit. The one exception to this trend is the GLOBAL category, which contained a small number of relatively easy instances. In this category, both the greedy schedules and the parallel schedules solve exactly the same number of instances as the best individual solver. In terms of average CPU time, the greedy schedules consistently outperform the corresponding parallel schedules, and usually (but not always) outperform the best individual solver.

Table 3. Summary of results (cross-validation results are parenthesized).

Category	Solver	Num. solved	Avg. CPU (s)
2-ARY-EXT	*Greedy*^{m}	1120 (1110)	107 (148)
	Greedy^s	1114 (1104)	150 (237)
	`VALCSP`	1093	126
	Parallel^m	1068	588
	Parallel^s	1042	1413
2-ARY-INT	*Greedy*^{m}	682 (674)	127 (167)
	Greedy^s	675 (667)	187 (262)
	Parallel^m	649	781
	Parallel^s	619	1894
	`buggy_2_5_s`	627	290
GLOBAL	*Greedy*^{m}	127 (127)	0.13 (1.14)
	Greedy^s	127 (127)	0.13 (2.78)
	`BPrologCSPSolver70a`	127	0.31
	Parallel^m	127	1.48
	Parallel^s	127	3.61
N-ARY-EXT	*Greedy*^{m}	298 (296)	298 (425)
	Greedy^s	292 (289)	352 (572)
	`Abscon 109 AC`	277	279
	Parallel^m	266	1522
	Parallel^s	252	3708
N-ARY-INT	*Greedy*^{m}	706 (701)	338 (407)
	Greedy^s	631 (625)	395 (498)
	Parallel^m	632	2109
	Parallel^s	614	4896
	`BPrologCSPSolver70a`	579	636

3 Discussion

In this paper we have investigated the potential for exploiting the complementary strengths of multiple constraint solvers through the use of task-switching schedules. As indicated in Table 3, our results include task-switching schedules that, if entered in the competition, would have run faster on average than any of the individual solvers and would have solved more instances within the half hour time limit. We hope that these results will encourage hybridization of existing constraint solvers.

A natural way to improve on the results presented here would be to use machine learning to take advantage of instance-specific features, as is done in SATzilla [6]. We plan to pursue this approach as future work.

References

1. Uriel Feige, László Lovász, and Prasad Tetali. Approximating min sum set cover. *Algorithmica*, 40(4):219–234, 2004.
2. Carla P. Gomes and Bart Selman. Algorithm portfolios. *Artificial Intelligence*, 126:43–62, 2001.
3. Bernardo A. Huberman, Rajan M. Lukose, and Tad Hogg. An economics approach to hard computational problems. *Science*, 275:51–54, 1997.
4. Tzur Sayag, Shai Fine, and Yishay Mansour. Combining multiple heuristics. In *Proceedings of the* 23^{rd} *International Symposium on Theoretical Aspects of Computer Science*, pages 242–253, 2006.
5. Matthew Streeter, Daniel Golovin, and Stephen F. Smith. Combining multiple heuristics online. In *Proceedings of the Twenty-Second Conference on Artificial Intelligence (AAAI-07)*, pages 1197–1203, 2007.
6. Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. SATzilla2007: a new & improved algorithm portfolio for SAT, 2007.