Algorithm Selection and Scheduling

Serdar Kadioglu[1], Yuri Malitsky[1], Ashish Sabharwal[2], Horst Samulowitz[2], and Meinolf Sellmann[2]

[1] Brown University, Dept. of Computer Science, Providence, RI 02912, USA
{serdark,ynm}@cs.brown.edu
[2] IBM Watson Research Center, Yorktown Heights, NY 10598, USA
{ashish.sabharwal,samulowitz,meinolf}@us.ibm.com

Abstract. Algorithm portfolios aim to increase the robustness of our ability to solve problems efficiently. While recently proposed algorithm selection methods come ever closer to identifying the most appropriate solver given an input instance, they are bound to make wrong and, at times, costly decisions. Solver scheduling has been proposed to boost the performance of algorithm selection. Scheduling tries to allocate time slots to the given solvers in a portfolio so as to maximize, say, the number of solved instances within a given time limit. We show how to solve the corresponding optimization problem at a low computational cost using column generation, resulting in fast and high quality solutions. We integrate this approach with a recently introduced algorithm selector, which we also extend using other techniques. We propose various static as well as dynamic scheduling strategies, and demonstrate that in comparison to pure algorithm selection, our novel combination of scheduling and solver selection can significantly boost performance.

1 Introduction

The constraint reasoning community has a long tradition of introducing and refining ideas whose practical impact often goes far beyond the field's scope. One such contribution is that of robust solvers based on the idea of *algorithm portfolios* (cf. [28, 11, 20, 21, 41, 25]). Motivated by the observation that solvers have complementary strengths and therefore exhibit incomparable behavior on different problem instances, algorithm portfolios run multiple solvers in parallel or select one solver, based on the features of a given instance. Portfolio research has led to a wealth of different approaches and an amazing boost in solver performance in the past decade. One of the biggest success stories is that of SATzilla [40], which combines existing Boolean Satisfiability (SAT) solvers and has now dominated various categories of the SAT Competition for about half a decade [29]. Another example is CP-Hydra [25], a portfolio of Constraint Programming (CP) solvers which won the CSP 2008 Competition. Instead of choosing a single solver for an instance, Silverthorn and Miikkulainen [30] proposed a Dirichlet Compound Multinomial distribution to create a schedule of solvers to be run in sequence. Other approaches (e.g., [17]) dynamically switch between a portfolio of solvers based on the predicted completion time. Alternatively, ArgoSmart [24] and Hydra [38] focus on not only choosing the best solver for an instance, but also the best parametrization of that solver. For a further overview of the state-of-the-art in portfolio generation, see the thorough survey by Smith-Miles [31].

A recently proposed algorithm selector for SAT based on nearest-neighbor classification [23] serves as the foundation for our work here. First, we present two extensions to it involving distance-based weighting and cluster-guided adaptive neighborhood sizes, demonstrating moderate but consistent performance improvements. Then we develop a new hybrid portfolio that combines algorithm selection and algorithm scheduling, in static and dynamic ways. To this end we present a heuristic method for computing solver schedules efficiently, which O'Mahony et al. [25] identified as an open problem. This also enables us to quantify the impact of various scheduling strategies and to report those findings accordingly. Finally, we are able to show that a completely new way of solver scheduling consisting of a combination of static schedules and solver selection is able to achieve significantly better results than plain algorithm selection.

Using SAT as the testbed, we demonstrate through extensive numerical experiments that our approach is able to handle even highly diverse benchmarks, in particular a mix of random, crafted, and industrial instance categories, with a *single* portfolio. This is in contrast to, for example, SATzilla, which has historically excelled only in different versions that were specifically tuned for each category. Our approach also works well even when the training set is not fully representative of the test set that needs to be solved.

2 Nearest-Neighbor-Based Algorithm Selection

Malitsky et al. [23] recently proposed a simple yet highly effective algorithm selector for SAT based on nearest-neighbor classification. We review this approach here, before proposing two improvements to it in Section 3 and algorithm schedules in Section 4.

Nearest-neighbor classification (k-NN) is a classic machine learning approach. In essence, we base our decision for a new instance on prior experience with the k training instances most similar to it. As the similarity measure between instances, we simply use the Euclidean or L^2 distance on 48 core features of SAT instances that SATzilla is based on [40]. Each feature is (linearly) normalized to fit the interval $[0, 1]$ across all training instances. As the solver performance measure, we use the PAR10 score of the solver on these k instances. PAR10 score for a given timelimit T is a hybrid measure, defined as the average of the runtimes for solved instances and of $10 \times T$ for unsolved instances. It is thus a combined measure of number of instances solved and average solution time.

It is well-known in machine learning that 1-NN (i.e., $k = 1$) often does not generalize well to formerly unseen examples, as it tends to over-fit the training data. A very large value of k, on the other hand, defeats the purpose of considering local neighborhoods. To find the "right" value of k, we employ another classic

Algorithm 1: Algorithm Selection using Nearest-Neighbor Classification

```
1  k-NN-Algorithm-Selection Phase
   Input   : a problem instance $F$
   Params: nearest neighborhood size $k$, candidate solvers $\mathcal{S}$, training instances
           $\mathcal{F}_{\text{train}}$ along with feature vectors and solver runtimes
   Output: a solver from the set $\mathcal{S}$
2  begin
3      compute normalized features of $F$
4      $\mathcal{F} \leftarrow$ set of $k$ instances from $\mathcal{F}_{\text{train}}$ that are closest to $F$
5      return solver in $\mathcal{S}$ that has the best PAR10 score on $\mathcal{F}$
6  end
7  Training Phase
   Input   : candidate solvers $\mathcal{S}$, training instances $\mathcal{F}_{\text{train}}$, time limit $T_{\max}$
   Params: neighborhood range $[k_{\min}, k_{\max}]$, number of sub-samples $m$, split ratio
           $m_b/m_v$
   Output: best performing $k$, reduced $\mathcal{F}_{\text{train}}$ along with feature and runtimes
8  begin
9      run each solver $S \in \mathcal{S}$ for time $T_{\max}$ on each $F \in \mathcal{F}_{\text{train}}$; record runtimes
10     remove from $\mathcal{F}_{\text{train}}$ instances solved by no solver, or by all within 1 second
11     compute feature vectors for each $F \in \mathcal{F}_{\text{train}}$
12     for $k \in [k_{min}, k_{max}]$ do
13         score[$k$] $\leftarrow 0$
14         for $i \in [1..m]$ do
15             $(\mathcal{F}_{\text{base}}, \mathcal{F}_{\text{validation}}) \leftarrow$ a random $m_b/m_v$ split of $\mathcal{F}_{\text{train}}$
16             add to score[$k$] performance of $k$-NN portfolio on $\mathcal{F}_{\text{validation}}$ using
                   training instances $\mathcal{F}_{\text{base}}$ and solver selection based on PAR10
17     score[$k$] $\leftarrow$ score[$k$]$/m$;    $k_{\text{best}} \leftarrow \text{argmin}_k$score[$k$]
18     return ($k_{\text{best}}$, $\mathcal{F}_{\text{train}}$, feature vectors, runtimes)
19 end
```

strategy in machine learning, namely *random sub-sampling validation*. The idea is to repeat the following process several times: Randomly split the training data into a base set and a validation set, train on the base set, and assess how well the learned approach performs on the validation set. We use a 67/33 base-validation split and perform random sub-sampling 100 times. We then finally choose the k that yields the best PAR10 performance averaged across the 100 validation sets.

Algorithm 1 gives a more formal description of the entire algorithm, in terms of its usage as a portfolio solver (i.e., algorithm selection given a new instance, as described above) and the random sub-sampling based training phase performed to compute the best value for k to use. The training phase starts out by computing the runtimes of all solvers on all training instances. It then removes all instances that cannot be solved by any solver in the portfolio within the time limit, or are solved by every solver in the portfolio within marginal time (e.g., 1 second for reasonably challenging benchmarks); learning to distinguish

Table 1. Comparison of Baseline Solvers, Portfolios, and VBS Performances: PAR10, average runtime in seconds, and number of instances solved (timeout 1,200 seconds).

	Pure Solvers							Portfolios		VBS
	agw-sat0	agw-sat+	**gnov-elty+**	SAT-enstein	march	pico-sat	kcnfs	SAT-zilla	***k*-NN**	
PAR10	5940	6017	5874	5892	8072	10305	6846	3578	**3151**	2482
Avg Time	634	636	626	625	872	1078	783	452	**442**	341
# Solved	290	286	293	292	190	83	250	405	**427**	457
% Solved	50.9	50.2	51.4	51.2	33.3	14.6	43.9	71.1	**74.9**	80.2

between solvers based on data from such instances is pointless. Along with the estimated best k, the training phase passes along this reduced set of training instances, their runtimes for each solver, and their features to the main solver selection phase. We emphasize that the training phase does not learn any sophisticated model (e.g., a runtime prediction model); rather, it simply memorizes the training performances of all solvers and "learns" only the value of k.

Despite the simplicity of this approach – compared, for example, to the description of SATzilla [40] – it is highly efficient and outperforms SATzilla2009_R, the Gold Medal winning solver in the random category of SAT Competition 2009. In Table 1 we compare simple k-NN algorithm selection with SATzilla_R, using the 2,247 random category instances from SAT Competitions 2002-2007 as the training set and the 570 such instances from SAT Competition 2009 as the test set. Both portfolios are based on the following local search solvers: Ag2wsat0 [36], Ag2wsat+ [37], gnovelty+ [26], Kcnfs04 [8], March_dl04 [16], Picosat 8.46 [3], and SATenstein [19], all in the versions that are *identical* to the ones that were used when SATzilla09_R [39] entered the 2009 competition. To make the comparison as fair as possible, k-NN uses only the 48 core instance features that SATzilla is based on and is trained for Par10-score. For both training and testing, we use a time limit of 1,200 seconds. Table 1 shows that SATzilla boosts performance of individual solvers dramatically.[3] The pure k-NN approach pushes the performance level substantially further. It solves 22 more instances and closes about one third of the gap between SATzilla_R and the virtual best solver (VBS),[4] which solves 457 instances.

3 Improving Nearest-Neighbor-Based Solver Selection

We now discuss two mutually orthogonal techniques to further improve the performance of the algorithm selector outlined in Section 2.

[3] The exact runtimes in Table 1 are lower than the ones reported in [23] due to faster machines: dual Intel Xeon 5540 (2.53 GHz) quad-core Nehalem processors with 24 GB of DDR-3 memory. The relative drop in the performance of *kcnfs*, we believe, is also due to this hardware difference.

[4] VBS refers to the "oracle" that always selects the solver that is fastest on the given instance. Its performance is the best one can hope to achieve with algorithm selection.

Distance-Based Weighting. A natural extension of k-NN is to scale the scores of the k neighbors of an instance based on the Euclidean distance to it. Intuitively speaking, inspired by O'Mahony et al. [25], we assign larger weights to instances that are closer to the test instance assuming that closer instances more accurately reflect the properties of the instance at hand. Hence, in Lines 5 and 16 of Algorithm 1, when computing the PAR10 score for solver selection for an instance F, we scale a solver S's penalized runtime (i.e., actual runtime or $10 \times T_{\max}$) on a neighbor F' by $\left(1 - \frac{dist(F,F')}{totalDist}\right)$, where *totalDist* corresponds to the sum of all distances from F to instances in the neighborhood under consideration.

Clustering-Based Adaptive Neighborhood Size. Rather than learning a single value for k, we adapt the size of the neighborhood based on the properties of the given test instance. To this end, we partition the instance feature space by clustering the training instances using g-means clustering [13]. An instance is considered to belong to the cluster it is nearest to (breaking ties arbitrarily). Algorithm 1 can be easily adapted to learn one k for each cluster. Given a test instance, we first determine the cluster to which it belongs and then use the value of k learned for this cluster during training. We note that our clustering is used to select only the *size* of the neighborhood based on instance features, not to limit the neighborhood itself; neighboring instances from other clusters can still be used when determining the best solver based on PAR10 score.

3.1 Experimental Setup and Evaluation

We now describe the benchmark used for portfolio evaluation in the rest of this paper. Note that such a benchmark involves not only training and testing instances but also the base solvers used for building portfolios. The challenging benchmark setting we consider mixes incomplete and complete SAT solvers, as well as industrial, crafted, and random instances. After describing these, we will assess the impact of weighting, clustering, and their combination, on pure k-NN. Note that the reported runtimes include all overhead incurred by our portfolios.

Benchmark Solvers. We consider the following 21 state-of-the-art complete and incomplete SAT solvers: 1. Clasp[9], 2. CryptoMiniSat [32], 3. Glucose [1], 4. Lineling [5], 5. LySat i [12], 6. LySat c [12], 7. March-hi [14], 8. March-nn [15], 9. MiniSAT 2.2.0 [33], 10. MXC [6], 11. PrecoSAT [4], 12. Adaptg2wsat2009 [22], 13. Adaptg2wsat2009++ [22], 14. Gnovelty+2 [27], 15. Gnovelty+2-H [27], 16. HybridGM3 [2], 17. Kcnfs04SAT07 [8], 18. Picosat [3], 19. Saps [18], 20. TNM [35], and 21. SATenstein [19]. We in fact use six different parametrizations of SATenstein, resulting in a total of 26 base solvers. In addition, we preprocess all industrial and crafted instances with SatElite (version 1.0, with default option '+pre') and let the following solvers run on both original and preprocessed version of each instance:[5] 1. Clasp, 2. CryptoMiniSat, 3. Glucose, 4. Lineling, 5. LySat c, 6. LySat i, 7. March-hi, 8. March-nn, 9. MiniSat, 10. MXC, and 11. Precosat. Our portfolio is thus composed of 37 solvers.

[5] Preprocessing usually does not improve performance on random instances.

Table 2. Average Performance Comparison of Basic k-NN, Weighting, Clustering, and the combination of both using the k-NN Portfolio. Numbers in braces show in how many of the 10 training-test splits does incorporating weighting and clustering outperform basic k-NN (column 2).

	Basic k-NN	Weighting	Clustering	**Weight.+Clust.**
# Solved	1609	1611	1615	**1617** (9/10)
# Unsolved	114	112	108	**106** (9/10)
% Solved	93.5	93.6	93.8	**93.9** (9/10)
Avg Runtime	588	584	584	**577** (7/10)
PAR10 Score	3518	3459	3369	**3314** (8/10)

Benchmark Instances. We selected $5,464$ instances from all SAT Competitions and Races during 2002 and 2010 [29], whereby we discarded all instances that cannot be solved by any of the aforementioned solvers within the competition time limit of 5,000 seconds (i.e., the VBS can solve 100% of all instances).

Now, we need to partition these instances into disjoint sets of training and testing instances. In research papers, we often find that only one training-test split of the instances is considered. Moreover, commonly this split is computed at random, thereby increasing the likelihood that the training set is quite representative of the test set. We propose to adopt some best practices from machine learning and to consider *multiple splits* as well as *a more challenging partitioning* into training and test sets. Our objective for the latter is to generate splits where entire benchmark families are completely missing in the training set, while for other families some instances are present in both the training and in the test partition. To asses which instances are related, we use the the first three characters in the prefix of an instance name and assume that instances starting with the same three characters belong to the same benchmark family. We select, at random, about 5% of benchmark *families* and include them fully in the test partition; this typically resulted in roughly 15% of all instances being in the test partition. Next, we randomly add more instances to the test partition until it has about 30% of all instances, resulting in a 70-30 split. The 10 such partitions used in our experimentation are available online for future reference.[6]

Results. Table 2 summarizes the performance gain from using weighting, clustering, and the combination of the two. We show the average performance (across the 10 training-test splits mentioned above) in terms of number of instances solved/not solved, average runtime, and PAR10 score. Depending on the performance measure, the combination of weighting and clustering is able to improve performance of basic k-NN on anywhere from 7 to 9 out of the 10 splits (shown in braces in the rightmost column). The gain is modest but serves as a good incremental step for the rest of this paper.

For completeness, we remark that these modest gains also translate to the benchmark discussed in Table 1, where the combination of weighting and clus-

[6] http://www.cs.toronto.edu/~horst/CP2011-Training-Test-Splits.zip

tering solves 7 more instances than basic k-NN and 29 more than SATzilla_R. We will return to this benchmark towards the end of this paper.

4 Building Solver Schedules

To further increase the robustness of our approach we consider computing schedules that define a sequence of solvers to try, along with individual time limits, given an instance. The general idea was previously introduced by Streeter [34] and in CP-Hydra [25]. In fact, Streeter [34] uses the idea of scheduling to *generate* algorithm portfolios. While he suggested using schedules that can suspend solvers and let them continue later on in exactly the same state they were suspended in, we will focus on solver schedules without preemption, i.e., each solver will appear in the schedule at most once. This setting was also used in CP-Hydra, which computes a schedule of CP solvers based on k nearest neighbors.

We note that a solver schedule can never outperform the VBS. In fact, a schedule is no better than the VBS *with a reduced captime of the longest running solver in the schedule.* Therefore, trivial schedules that split the available time evenly between all solvers have inherently limited performance. The reason why we may be interested in solver schedules nevertheless is to hedge our bets: We often observe that instances that cannot be solved by one solver even in a very long time can in fact be solved by another very quickly. Consequently, by allocating a reasonably small amount of time to other solvers we can provide a safety net in case our solver selection happens to be unfortunate.

4.1 Static Schedules

The simplest approach is to compute a static schedule of solvers. For example, we could compute a schedule that solves the most training instances within the allowed time (cf. [25]). We propose to do slightly more, namely to compute a schedule that, first, solves most training instances and that, second, requires the lowest amount of time among all schedules that are able to solve the same amount of training instances. We can formulate this problem as an integer program (IP), more precisely as a resource constrained set covering problem (RCSCP), where the goal is to select a number of solver-runtime pairs that together "cover" (i.e., solve) as many training instances as possible:

Solver Scheduling IP:

$$\min \quad (C+1)\sum_i y_i + \sum_{S,t} t x_{S,t} \qquad (1)$$

$$s.t. \quad y_i + \sum_{(S,t)\ \mid\ i \in V_{S,t}} x_{S,t} \geq 1 \qquad \forall i \qquad (2)$$

$$\sum_{S,t} t x_{S,t} \leq C \qquad (3)$$

$$y_i, x_{S,t} \in \{0,1\} \qquad \forall i, S, t \qquad (4)$$

Binary variables $x_{S,t}$ correspond to sets of instances that can be solved by solver S within a time t. These sets have cost t and a resource consumption coefficient t. To make it possible that all training instances can be covered even when they remain unsolved, we introduce additional binary variables y_i. These correspond to the set that contains only item i, they have cost $C + 1$ and time resource consumption coefficient 0. The constraints (2) in this model enforce that we cover all training instances, the additional resource constraint (3) that we do not exceed the overall captime C. The objective is to minimize the total cost. Due to the high costs for variables y_i (which will be 1 if and only if instance i cannot be solved by the schedule), schedules that solve most instances are favored, and among those the fastest schedule is chosen (as the cost of $x_{S,t}$ is t).

4.2 A Column Generation Approach

The main problem with the above formulation is the sheer number of variables. For our most up-to-date benchmark with 37 solvers and more than 5,000 training instances, solving the above problem is impractical, even when we choose the timeouts t smartly such that from timeout $t1$ to the next timeout $t2$ at least one more instance can be solved by the respective solver ($V_{S,t1} \subsetneq V_{S,t2}$). In our experiments we found that the actual time to solve these IPs may at times still be tolerable, but the memory consumption was often prohibitively high.

We therefore propose to solve the above problem approximately, using column generation (aka Dantzig-Wolfe decomposition) – a well-known technique for handling linear programs (LPs) with a lot of variables [7, 10]. We discuss it briefly in the general setting. Consider the LP:

$$\min c^{\mathrm{T}} x \quad \text{s.t.} \;\; A\, x \geq b, x \geq 0 \tag{5}$$

In the presence of too many variables, it is often not practical to solve the large system (5) directly. The core observation underlying column generation is that only a few variables (i.e., “columns”) will be non-zero in any optimal LP solution (at most as many as there are constraints). Therefore, if we knew which variables are important, we could consider a much smaller system $A'\, x' = b$ where A' contains only a few columns of A. When we choose only some columns in the beginning, LP duality theory tells us which columns that we have left out so far are of interest for the optimization of the global LP. Namely, only columns with *negative reduced costs* (which are defined based on the optimal duals of the system $A'\, x' = b$) can help the objective to decrease further.

Column generation proceeds by considering, in turn, a *master problem* (the reduced system $A'\, x' = b$) and a *subproblem* where we select a new column to be added to the master based on its current optimal dual solution. This process is iterated until there is no more column with a negative reduced cost. At this point, we know that an optimal solution to (5) has been found – even though most columns have never been added to the master problem!

When using standard LP solvers to solve the master problem and obtain its optimal duals, all that is left is solving the subproblem. To develop a subproblem

Algorithm 2: Subproblem: Column Generation

```
begin
    minRedCosts ← $\infty$
    forall Solvers $S$ do
        $T \leftarrow 0$
        forall $i$ do
            $j \leftarrow \pi(i)$;   $T \leftarrow T + \lambda_j$;   $\hat{t} \leftarrow \mathrm{Time}(S, j)$
            redCosts ← $\hat{t}(1 - \mu) - T$
            if redCosts < minRedCosts then
                Solver ← $S$
                timeout ← $\hat{t}$
                minRedCosts ← redCost
    if minRedCosts < 0 then return $x_{\mathrm{Solver,timeout}}$
    else return None
end
```

generator, we need to understand how exactly the reduced costs are computed. Assume we have a dual value $\lambda_i \geq 0$ for each constraint in A'. Then, the reduced cost of a column $\alpha := (\alpha_1, \dots, \alpha_z)^{\mathrm{T}}$ is defined as $\bar{c}_\alpha = c_\alpha - \sum_i \lambda_i \alpha_i$, where c_α is the cost of column α.

Equipped with this knowledge, we can apply column generation to solve the continuous relaxation of the Solver Scheduling IP. To this end, we begin the process by adding, at the start, all columns corresponding to variables y to our reduced system A'. Next, we repeatedly generate and solve a subproblem whose goal is to suggest a solver-runtime pair that is likely to increase the objective value of the (continuous) master problem the most. Hence, each column we add regards an $x_{S,t}$ variable, specifically the one with minimal reduced cost.

To find such an $x_{S,t}$, first, for all solvers S, we compute a permutation π of the instances such that the time that S needs to solve instance $\pi_S(i)$ is less than or equal that the solver needs to solve instance $\pi_S(i+1)$ (for appropriate i). See Algorithm 2. Obviously, we only need to do this once for each solver and not each time we want to generate a new column. Now, let us denote with $\lambda_i \geq 0$ the optimal dual value for the restriction to cover instance i (2). Moreover, denote with $\mu \leq 0$ the dual value of the resource constraint (3) (since that constraint enforces a lower-or-equal restriction μ is guaranteed to be non-positive). Finally, for each solver S we iterate over i and compute the term $T \leftarrow \sum_{k \leq i} \lambda_{\pi_S(k)}$ (which in each iteration we can obviously derive from the previous value for T). Let $\hat{t}$ denote the time that solver S needs to solve instance $\pi(i)$. Then, the reduced costs of the column that corresponds to variable $x_{S,t}$ are $\hat{t} - \hat{t}\mu - T$. We choose the column with the most negative reduced costs and add it to the master problem. If there is no more column with negative reduced costs, we stop.

We would like to point out two things. First, note that what we have actually done is to pretend that all columns were present in the matrix and computed the reduced costs for all of them. This is not usually the case in column generation approaches where most columns are usually found to have larger reduced costs *implicitly* rather than explicitly. Second, note that the solution returned from this process will in general not be integer but contain fractional values. Therefore, the solution obtained cannot be interpreted as a solver schedule directly.

This situation can be overcome in two ways. The first is to start branching and to generate more columns – which may still be needed by the optimal integer solution even though they were superfluous for the optimal fractional solution. This process is known in the literature as branch-and-price.

What we propose, and that is in fact the reason why we solved the original problem by means of column generation in the first place, is to stick to the columns that were added during the column generation process and to solve the remaining system as an IP. Obviously, this is just a heuristic that may return sub-optimal schedules for the training set. However, we found that this process is very fast and nevertheless provides high quality solutions (see empirical results in Section 4.4). Even when the performance on the training set is at times slightly worse than optimal, the performance on the test set often turned out as good or sometimes even better than that of the optimal training schedule – a case where the optimal schedule overfits the training data.

The last aspect that we need to address is the case where the final schedule does not utilize the entire available time. Recall that we even deliberately minimize the time needed to solve as many instances as possible. Obviously, at runtime it would be a waste of resources not to utilize the entire time that is at our disposal. In this case, we scale each solver's time in the schedule equally so that the total time of the resulting schedule will be exactly the captime C.

4.3 Dynamic Schedules

O'Mahony et al. [25] found that static schedules work only moderately well. Therefore, they introduced the idea of computing *dynamic* schedules: At runtime, for a given instance, CP-Hydra considers the ten nearest neighbors (in case of ties, up to fifty) and computes a schedule that solves as many of these instances as possible in the given time limit. Accordingly, the constraints in the Solver Scheduling IP are limited to the few instances in the neighborhood, which allows CP-Hydra to use a brute-force approach to compute dynamic schedules at runtime. This is reported to work well thanks to the small neighborhood size and the fact that CP-Hydra only has three constituent solvers.

Our column generation approach, yielding potentially sub-optimal but usually high quality solutions, works fast enough to handle even 37 solvers and over 5,000 instances within seconds. This allows us to embed the idea of dynamic schedules in the previously developed nearest-neighbor approach which selects optimal neighborhood sizes by random subsampling validation – which requires us to solve hundreds of thousands of these IPs.

Both cluster-guided adaptive neighborhood size and weighting discussed earlier can be incorporated into solver schedules as well. For the latter, we suggest a slightly different approach than CP-Hydra. Specifically, when given an instance

Table 3. Average performance of semi-static schedules compared with no schedules and with static schedules based only on the available solvers. Numbers in braces show in how many of the 10 training-test splits does semi-static scheduling with weighting and clustering outperform the same approach without scheduling (column 2).

	No Sched.	Static Sched.	Semi-Static Schedules			
	Wtg+Clu	Wtg+Clu	Basic k-NN	Weighting	Clustering	**Wtg+Clu**
# Solved	1617	1572	1628	1635	1633	**1636** (7/10)
# Unsolved	106	151	95	88	90	**87** (7/10)
% solved	93.9	91.2	94.6	94.9	94.8	**95.0** (7/10)
Avg Runtime	577	562	448	451	**446**	449 (10/10)
PAR10 score	3314	4522	2896	2728	2789	**2716** (8/10)

F, we adapt the objective function in the Solver Scheduling IP by multiplying the costs for the variables y_i, which were originally $C + 1$, with $2 - \frac{\text{dist}(F,F_i)}{\text{totalDist}}$. This favors schedules that solve more training instances that are closer to F.

We thus obtain four variations of dynamic schedules. We also used a setting inspired by the CP-Hydra approach: size 10 neighborhood size and weighting scheme as in [25]. We refer to this approach as SAT-Hydra. In our experiments with dynamic schedules as well as SAT-Hydra, we found the gain over and above k-NN solver selection with weights and clustering (the rightmost column in Table 2) was marginal. SAT-Hydra and dynamic schedule without weights and clustering, for example, each solved only 4 more instances. Due to limited space, we omit detailed experimental numbers and instead move on to scheduling strategies that turned out to be more effective.

4.4 Semi-Static Solver Schedules

Observe that the four algorithm selection portfolios that we developed in Section 2 can themselves be considered solvers. We can add the portfolio itself to our set of constituent solvers and compute a “static” schedule for this augmented collection of solvers. We quote “static” here because the resulting schedule is of course still instance-specific. After all, the algorithm selector portfolio chooses one of the constituent solvers based on the test instance’s features. We refer to the result of this process as *semi-static solver schedules*.

Depending on which of our four portfolios from Section 2 we use, we obtain four semi-static schedules. We report their performance Table 3. We observe that semi-static scheduling improves the overall performance in anywhere from 7 to 10 of the 10 training-test splits considered, depending on the performance measure used (compare with column 2 in the table for the best results without scheduling). All semi-static schedules here solve at least 20 more instances within the time limit. Again, the combination of weighting and clustering achieves the best performance and it narrows the gap to VBS in percentage of instances solved to nearly 5%. For further comparison, in the column 3 we show the performance of a static schedule that was trained on the entire training set and is the same for all test instances. We can confirm the earlier finding [25] that static

Table 4. Comparison of Column Generation and the Solution to the Optimal IP.

Schedule by	# Solved	# Unsolved	% Solved	Avg Runtime (s)	PAR10 score
Optimal IP	1635.8	87.1	95.0	442.5	2708.4
Column Generation	1635.7	87.2	95.0	448.9	2716.2

solver schedules are indeed inferior to dynamic schedules, and find that they are considerably outperformed by semi-static solver schedules.

Quality of results generated by Column Generation. Table 4 illustrates the performance of our Column Generation approach. We show a comparison of the resulting performance achieved by the *optimal* schedule. In order to compute the optimal solution to the IP we used Cplex on a machine with sufficient memory and a 15 second resolution to fit the problem into the available memory. As we can observe the column generation is able to determine a high quality schedule that results in a performance that nearly matches the one of the (coarse-grained) optimal schedule according to displayed measures.

4.5 Fixed-Split Selection Schedules

Based on this success, we consider a parametrized way of computing solver schedules. As discussed earlier, the motivation for using solver schedules is to increase robustness and hedge against an unfortunate selection of a long-running solver. At the same time, the best achievable performance of a portfolio is that of the VBS *with a captime of the longest individual run.* In both dynamic and semi-static schedules, the runtime of the longest running solver(s) was determined by the column generation approach working solely on training instances. This procedure inherently runs the risk of overfitting the training set.

Consequently, we consider splitting the time between an algorithm selection portfolio and the constituent solvers based on a parameter. For example, we could allocate 90% of the available time for the solver selected by the portfolio. For the remaining 10% of the time, we run a static solver schedule. We refer to these schedules as *90/10-selection schedules.* Note that choosing a fixed amount of time for the schedule of constituent solvers is likely to be suboptimal for the training set but offers the possibility of improving test performance.

Table 5 captures the corresponding results. We observe that using this restricted application of scheduling is able to outperform our best approach so far (semi-static scheduling, shown again in the first column, which is outperformed consistently in 9 out of 10 training-test splits). We are able to solve nearly 1642 instances on average which is 6 more than we were able to solve before and the gap to the virtual best solver is narrowed down to 4.69%. Recall that we consider a highly diverse set of benchmark instances from the Random, Crafted, and Industrial categories. Moreover, we do not work with plain random splits, but splits where complete families of instances in the test set are not represented in the training set at all.

Table 5. Average performance comparison of basic k-NN, weighting, clustering, and the combination of both using the k-NN Portfolio with a 90/10 fixed-split static schedule. Numbers in braces show in how many of the 10 training-test splits does fixed-split scheduling with weighting and clustering outperform the same approach with semi-static scheduling (column 2).

	Semi-Static	Fixed-Split Schedules			
	Wtg+Clu	Basic k-NN	Weighting	Clustering	**Wtg+Clu**
# Solved	1636	1637	1641	1638	**1642** (9/10)
# Unsolved	87	86	82	85	**81** (9/10)
% solved	95.0	95.0	95.3	95.1	**95.3** (9/10)
Avg Runtime	449	455	446	452	**445** (9/10)
PAR10 score	2716	2683	2567	2652	**2551** (9/10)

Compared to the plain k-NN approach of Malitsky et al. [23] that we started with (column 2 of Table 2), the fixed-split selection schedules close roughly one third of the gap to the VBS. The performance gain, as measured by Welch's T-test, is significant in most of the training-test splits. For example, the p-value for the T-test of an instance being solved or not by the two approaches has a median value of 0.05. Similarly, the median p-value across the 10 splits for the penalized runtime is 0.04, indicating the improvements are statistically significant.

5 Summary and Discussion

We considered the problem of algorithm selection and scheduling so as to maximize performance when given a hard time limit within which we need to provide a solution. We considered two improvements for simple nearest-neighbor solver selection, weighting and adaptive neighborhood sizes based on clustering. Then, we developed a light-weight optimization algorithm to compute near-optimal schedules for a given set of training instances. This allowed us to provide an extensive comparison of pure algorithm selection, static solver schedules, dynamic solver schedules, and semi-static solver schedules which are essentially static schedules combined with an algorithm selector.

While quantifying the performance of the various scheduling strategies we found out that dynamic schedules are only able to achieve rather minor improvements and that semi-static schedules work the best among these options. Finally, we compared two alternatives: use the optimization component or use a fixed percentage of the allotted time when deciding how much time to allocate to the solver suggested by the algorithm selector. In either case, we used a static schedule for the remaining time. This latter parametrization allowed us to avoid overfitting the training data and overall resulted in the best performance.

We tested this approach on a highly diverse benchmark set with random, crafted, and industrial SAT instances where we even deliberately removed entire families of instances from the training set. 90/10 fixed-split selection schedules demonstrated a convincing performance and solved, on average, over 95% of the instances that the virtual best solver is able to solve.

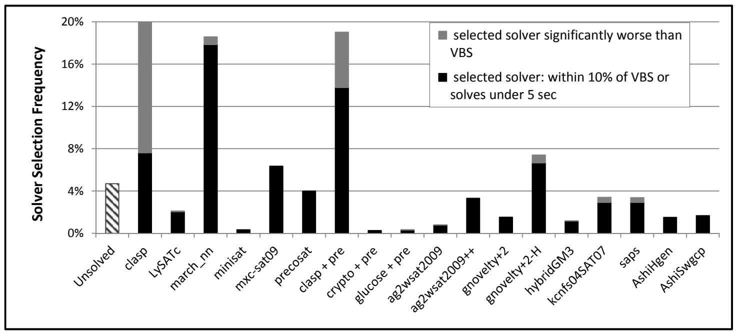

Fig. 1. Frequency of solver selection by 90-10 fixed-split schedule.

As an insight into the selection strategy of our fixed-split selection schedule, Figure 1 shows the fraction of test instances across all training-test splits on which any given solver was chosen and resulted in a successful run. The special bar labeled 'unsolved' shows how often the portfolio made a choice that resulted in failing to solve an instance (which here equals the gap to the VBS). Note that out of the 37 possible choices, our portfolio chose only 18 solvers in successful runs. Further, the black portion of the bars indicates how often was the selected solver nearly the best possible choice, defined as the solver taking within 10% of VBS time or solving the instance within 5 seconds. The predominant black regions, with the exception of Clasp, indicate that our portfolio often selected solvers with performance close to that of the VBS.

Table 6. Comparison of Major Portfolios for the SAT-Rand Benchmark (570 test instances, timeout 1,200 seconds). Values in braces denote p-value of Welch's T-test for the considered solver improving upon SATzilla_R as the baseline.

	SATzilla_R	SAT-Hydra	k-NN	**90-10**	VBS
# Solved	405	419	427 (0.071)	**435** (0.022)	457
# Unsolved	165	151	143 —	**135** —	113
% solved	71.5	73.5	74.9 —	**76.3** —	80.2
Avg Runtime	452	489	442 (0.367)	**400** (0.042)	341
PAR10 score	3578	3349	3151 (0.085)	**2958** (0.022)	2482

As a final remark, in Table 6, we close the loop and consider again the first benchmark set from Section 2 which compared portfolios for SAT Competition's random category instances, based on the same solvers as the gold-medal winning SATzilla_R. Overall, we go up from 405 (88.6% of VBS) for SATzilla_R to 435 (95.1% of VBS) instances solved for our fixed-split solver schedules. In other words, fixed-split selection schedule closes over 50% of the performance gap between SATzilla_R and the VBS. The p-values of Welch's T-test being below 0.05 (shown within braces) indicate that the performance achieved by our fixed-split selection schedule is statistically significantly better than SATzilla_R.

References

1. G. Audemard and L. Simon. GLUCOSE: a solver that predicts learnt clauses quality, *SAT Competition*, 7–8, 2009.
2. A. Balint, M. Henn, O. Gableske hybridGM. Solver description. *SAT Competition*, 2009.
3. A. Biere. Picosat version 846. Solver description. *SAT Competition*, 2007.
4. A. Biere. P{re,i}coSATSC09 *SAT Competition*, 41–43, 2009.
5. A. Biere. Lingeling, *SAT Race*, 2010.
6. D. R. Bregman. The SAT Solver MXC, Version 0.99, *SAT Competition*, 37–38, 2009.
7. G.B. Dantzig and P. Wolfe. The decomposition algorithm for linear programs. *Econometrica*, 29(4):767–778, 1961.
8. G. Dequen and O. Dubois. kcnfs. Solver description. *SAT Competition*, 2007.
9. M. Gebser, B. Kaufmann and T. Schaub. Solution Enumeration for Projected Boolean Search Problems, *CPAIOR*, 2009.
10. P.C. Gilmore and R.E. Gomory. A linear programming approach to the cutting stock problem. *Operations Research*, 9:849–859, 1961.
11. C.P. Gomes and B. Selman. Algorithm Portfolios. *Artificial Intelligence*, 126(1–2):43–62, 2001.
12. Y. Hamadi, S. Jabbour, and L. Sais. LySAT: solver description, *SAT Competition*, 23–24, 2009.
13. G. Hamerly and C. Elkan. Learning the K in K-Means. *NIPS*, 2003.
14. M. Heule and H. van Marren. march hi: solver description, *SAT Competition*, 27–28, 2009.
15. M. Heule and H. van Marren. march nn, *http://www.st.ewi.tudelft.nl/sat/download.php*.
16. M. Heule, J. Zwieten, M. Dufour, H. Maaren. March_eq: implementing additional reasoning into an efficient lookahead SAT solver. *Theory and Applications of Satisfiability Testing*, 3542:345–359, 2004.
17. B. Huberman, R. Lukose and T. Hogg. An Economics Approach to Hard Computational Problems. *Science*, 265:51–54, 2003.
18. F. Hutter, D.A.D. Tompkins and H.H. Hoos. Scaling and probabilistic smoothing: Efficient dynamic local search for SAT. *CP 02*, 233–248, 2002.
19. A.R. KhudaBukhsh, L. Xu, H.H. Hoos, K. Leyton-Brown. SATenstein: Automatically Building Local Search SAT Solvers From Components. *IJCAI*, 2009.
20. M.G. Lagoudakis, M.L. Littman. Learning to select branching rules in the DPLL procedure for satisfiability. *SAT*, 2001.
21. K. Leyton-Brown, E. Nudelman, G. Andrew, J. McFadden, Y. Shoham. A Portfolio Approach to Algorithm Selection. *IJCAI*, 1542–1543, 2003.
22. C.M. Li and W. Wei. Combining Adaptive Noise and Promising Decreasing Variables in Local Search for SAT. Solver description. *SAT Competition*, 2009.
23. Y. Malitsky and A. Sabharwal and H. Samulowitz and M. Sellmann. Non-Model-Based Algorithm Portfolios for SAT. *SAT*, to be published, 2011.
24. M. Nikolic, F. Maric, P. Janici. Instance Based Selection of Policies for SAT Solvers. *Theory and Applications of Satisfiability Testing*, 326–340, 2009.
25. E. O'Mahony, E. Hebrard, A. Holland, C. Nugent, B. O'Sullivan. Using Case-based Reasoning in an Algorithm Portfolio for Constraint Solving. *Irish Conference on Artificial Intelligence and Cognitive Science*, 2008.
26. D.N. Pham and C. Gretton. gnovelty+. Solver description. *SAT Competition*, 2007.
27. D.N. Pham and C. Gretton. gnovelty+ (v.2). Solver description. *SAT Competition*, 2009.
28. J. R. Rice The algorithm selection problem. *Advances in computers*, 65–118, 1976.
29. SAT Competition. *http://www.satcomptition.org*.
30. B. Silverthorn and R. Miikkulainen. Latent Class Models for Algorithm Portfolio Methods. *AAAI*, 2010.
31. K.A. Smith-Miles. Cross-disciplinary perspectives on meta-learning for algorithm selection. *ACM Comput. Surv.*, 41(1): 6:1–6:25, 2009.
32. M. Soos CryptoMiniSat 2.5.0. Solver description. *SAT Race*, 2010.
33. N. Sorensson and N. Een. MiniSAT 2.2.0, *http://minisat.se*, 2010.
34. M. Streeter and D. Golovin and S. F. Smith. Combining Multiple Heuristics Online. *AAAI*, 1197–1203, 2007.
35. W. Wei and C.M. Li. Switching Between Two Adaptive Noise Mechanisms in Local Search for SAT. Solver description. *SAT Competition*, 2009.
36. W. Wei, C.M. Li, H. Zhang. Combining adaptive noise and promising decreasing variables in local search for SAT. Solver description. *SAT Competition*, 2007.
37. W. Wei, C.M. Li, H. Zhang. Deterministic and random selection of variables in local search for SAT. Solver description. *SAT Competition*, 2007.
38. L. Xu, H. H. Hoos, K. Leyton-Brown. Hydra: Automatically Configuring Algorithms for Portfolio-Based Selection. *AAAI*, 2010.
39. L. Xu, F. Hutter, H.H. Hoos, K. Leyton-Brown. SATzilla2009: an Automatic Algorithm Portfolio for SAT. Solver description. *SAT Competition*, 2009.
40. L. Xu, F. Hutter, H.H. Hoos, K. Leyton-Brown. SATzilla: Portfolio-based Algorithm Selection for SAT. *JAIR*, 32(1):565–606, 2008.
41. L. Xu, F. Hutter, H.H. Hoos, K. Leyton-Brown. SATzilla-07: The Design and Analysis of an Algorithm Portfolio for SAT. *CP*, 712–727, 2007.