Non-Model-Based Search Guidance for Set Partitioning Problems

Serdar Kadioglu, Yuri Malitsky
Brown University, Providence, RI 02912
{*serdark,ynm*}@*cs.brown.edu*

Meinolf Sellmann
IBM Research, Yorktown Heights, NY 10598
meinolf@us.ibm.com

Abstract

We present a dynamic branching scheme for set partitioning problems. The idea is to trace features of the underlying MIP model and to base search decisions on the features of the current subproblem to be solved. We show how such a system can be trained efficiently by introducing minimal learning bias that traditional model-based machine learning approaches rely on. Experiments on a highly heterogeneous collection of set partitioning instances show significant gains over dynamic search guidance in Cplex as well as instance-specifically tuned pure search heuristics.

Search is an integral part of solution approaches for NP-hard combinatorial optimization and decision problems. Once the ability to reason deterministically is exhausted, state-of-the-art solvers try out different alternatives which may lead to an improved (in case of optimization) or feasible (in case of satisfaction) solution. This consideration of alternatives may take place highly opportunistically as in local search approaches, or systematically as in backtracking-based methods.

Efficiency could be much improved if we could effectively favor alternatives that lead to optimal or feasible solutions and a search space partition that allows short proofs of optimality or infeasibility. After all, the existence of an "oracle" is what distinguishes a non-deterministic from a deterministic Turing machine. This of course means that assuming $P \mathrel{!=} NP$, perfect choices are impossible to guarantee. The important insight is to realize that this is a worst-case statement. In practice, we may still hope to be able to make very good choices on average.

The view outlined above has motivated research on exploiting statistical methods to guide the search. The idea of using survey propagation in SAT (A. Braunstein 2005) has led to a remarkable performance improvement of systematic solvers for random SAT instances. In stochastic offline programming (Malitsky and Sellmann 2009), biased randomized search decisions are based on an offline training of the solver. This approach later led to the idea of instance-specific algorithm configuration (ISAC (S. Kadioglu 2010)). Here, offline training is used to associate certain features of the problem instance with specific parameter settings for the solver, whereby the latter may include the choice of branching heuristic to be used. In (Samulowitz and Memisevic 2007) branching heuristics for quantified Boolean formulae (QBF) were selected based on the features of the current subproblem which led to more robust performance and solutions to formerly unsolved instances.

In this paper, we combine the idea of instance-specific algorithm configuration with the idea of a dynamic branching scheme that bases branching decisions on the features of the current subproblem to be solved. Like the ISAC configuration system, our approach is not based on graphical or any other machine learning models. Instead, we cluster training instances according to their features and determine an assignment of branching heuristics to clusters that results in the best performance when the branching heuristic is dynamically chosen based on the current subproblem's nearest cluster. We test our approach on the MIP-solver Cplex that we use to tackle set partitioning problems. Our experiments show that this approach can effectively boost search performance even when trained on a rather small set of instances.

Related Work

In short, we follow-up on the idea of choosing a branching heuristic dynamically based on certain features of the current subproblem. This idea, to adapt the search to the instance to be solved, is by no means new. Algorithm configuration methods like stochastic offline programming (Malitsky and Sellmann 2009) or ISAC (S. Kadioglu 2010) determine the search heuristic based on the root-node features of the problem instance. The local search SAT solver SATenstein (A. R. KhudaBukhsh 2009) parametrizes its search so that it can be tuned for particular problem instance distributions.

The dynamic search engine (S. L. Epstein 2002) goes further than the previously mentioned approaches in that it adapts the search heuristics based on the current state of the search. In (Leventhal and Sellmann 2008), value selection heuristics for Knapsack were studied and it was found that accuracy of search guidance may depend heavily on the effect that decisions higher in the search tree have on the distribution of subproblems that are encountered deeper in the tree. The latter obviously creates a serious chicken-and-egg problem for statistical learning approaches: **the distri-**

bution of instances that require search guidance affects the choice of heuristic but the latter then affects the distribution of subproblems that are encountered deeper in the tree. In (Samulowitz and Memisevic 2007) a method for adaptive search guidance for QBF solvers was based on logistic regression. The issue of subproblem distributions was addressed by adding subproblems to the training set that were encountered during previous runs.

Inspired by the success of the approach in (Samulowitz and Memisevic 2007), we aim to boost the Cplex MIP solver to faster solve set partitioning problems. To this end, we modify the branching heuristics, basing it on the features of the current subproblem to be solved. The objective of this study is to find out whether such a system can be effectively trained to improve the performance of a generalized solver for a specific application. We will need to specify features that characterize instances to a set partitioning problem. Then, we need to find a methodology that allows us to learn for which subproblem (as characterized by their features) we wish to invoke which branching heuristic.

In (Samulowitz and Memisevic 2007), a logistic regression model was used to predict the heuristic that results in the lowest runtime with the highest probability. Empirical hardness models like this are quite common for this kind of task and have been used successfully for solver portfolio generation (Gagliolo and Schmidhuber 2006; Gomes and Selman 2001; L. Xu 2008) in the past. The problem with such models is that the learning models place a rather restrictive bias on the functions that can be learned. Consequently, predictions of which solver or which heuristic performs best for which instance can easily result in choices that underperform by one or two orders of magnitude in actual runtime.

Recently, training methods have been proposed that impose only a very limited bias. Before we explain our approach in detail, let us first review the idea of instance-specific algorithm configuration that is not model-based.

Instance-specific Algorithm Configuration

The instance-specific algorithm configuration (ISAC) approach (S. Kadioglu 2010) works as follows. In the learning phase, we are provided with the parametrized solver A, a list of training instances T, and their corresponding feature vectors F. First, we normalize the features in the set and memorize the scaling and translation values for each feature.

Then, we use an algorithm to cluster the training instances based on the normalized feature vectors (e.g., g-means (Hamerly and Elkan 2003)). The final result of the clustering is a number of k clusters S_i, and a list of cluster centers C_i. For each cluster of instances S_i we compute favorable parameters P_i via some instance-oblivious tuning algorithm (such as GGA (C. Ansotegui-Gil 2009)).

When running algorithm A on an input instance x, we first compute the features of the input and normalize them using the previously stored scaling and translation values for each feature. Then, we determine the cluster with the nearest center to the normalized feature vector. Finally, we run algorithm A on x using the parameters learnt for this cluster.

Learning Dynamic Search Heuristics

ISAC is interesting as it provides a low-bias learning approach for instance-specific choices of an algorithm. An aspect that is missing, however, is a dynamic choice of heuristics during search.

We can adapt the approach by modifying the systematic solver that we use to tackle the combinatorial problem in question. We propose the following approach:

1. First, we cluster our training instances based on the normalized feature vectors like in ISAC.
2. We parametrize the solver by leaving open the association of branching heuristic to cluster.
3. At runtime, whenever the solver reaches a new search node (or at selected nodes), we compute the features of the current subproblem to be solved.
4. We compute the nearest cluster and use the corresponding heuristic to determine the next branching constraint.

Now, the problem has been reduced to finding a good assignment of heuristics to clusters. At this point we have stated the problem in such a way that we can use a standard instance-oblivious algorithm configuration system to find such an assignment. We will use GGA (C. Ansotegui-Gil 2009) for this purpose.

Note how our approach circumvents the chicken-and-egg problem mentioned in the beginning that results from the tight correlation of the distribution of subproblems encountered during search and the way how we select branching constraints. Namely, by associating heuristics and clusters *simultaneously* we implicitly take into account that changes in our branching strategy result in different subproblem distributions, and that the best branching decision at a search node depends heavily on the way how we will select branching constraints further down in the tree.

That being said, the clusters themselves should reflect not only the root-node problems but also the subproblems that may be encountered during search. To this end, we can add subproblems encountered during the runs of individual branching heuristics on the training instances to the clusters. This changes the shape of the clusters and may also create new ones. However, note that we do *not* use these subproblems to learn a good assignment of heuristics to clusters which will be purely based on the original training instances. This assures that we do not base the assignment of heuristics to clusters on subproblems that will not be encountered.

Boosting Branching in Cplex for SPP

We will now apply the methodology established above to improve branching in the state-of-the-art MIP solver Cplex when solving set partitioning problems. Set-partitioning is one of *the* most prominent combinatorial optimization problems, with applications from crew-scheduling to combinatorial auctions to coalition structure generation.

Definition 1 *Given* items $1 \ldots n$ *and a collection of* m *sets of these items, which we call* bags*, and a cost associated with each bag, the* set partitioning problem (SPP) *consists in* *finding a set of bags such that the union of all bags contains all items, the bags are pairwise intersection-free, and the cost of the selection is minimized.*

We express this problem as an IP where x_i is a binary variable deciding whether bag i should be included, c_i is the cost of including the bag, and S_j represents the bags that item j appears in:

$$\begin{aligned} \min \sum_{i=1}^{m} c_i x_i & \\ s.t. \sum_{i \in S_j} x_i = 1 & \qquad 1 \leq j \leq n \\ x_i \in \{0,1\} & \qquad 1 \leq i \leq m \end{aligned}$$

To apply our methodology, we need to define instance features for set partitioning problems, and we need to devise various branching heuristics that our solver can choose from.

Set Partitioning Features

In order to characterize SPP instances, we first compute the following vectors:

- the normalized cost vector $c' \in [1, 100]^m$,
- the vector of bag densities $(|S_i|/n)_{i=1\ldots m}$,
- the vector of item costs $(\sum_{i,j \in S_i} c'_i)_{j=1\ldots n}$,
- the vector of item coverings $(|\{i \mid j \in S_i\}|/m)_{j=1\ldots n}$,
- the vector of costs over density $(c'_i/|S_i|)_{i=1\ldots m}$,
- the vector of costs over square density $(c'_i/|S_i|^2)_{i=1\ldots m}$,
- the vector of costs over $k \log k$-density $(c'_i/(|S_i| \log |S_i|))_{i=1\ldots m}$, and
- the vector of root-costs over square density $(\sqrt{c'_i}/|S_i|^2)_{i=1\ldots m}$.

As features we then compute the averages, median, standard deviations, and the entropies of all these statistics for all vectors. In addition to the eight vectors above we add one more feature that represents the number of sets divided by the number of items, therefore ending up with 33 features. One of the benefits of this feature set is that it is invariant under column and row permutations of the problem matrix.

Branching Heuristics

We also need to provide a portfolio of different branching selection heuristics. We consider the following, all of which we implement using Cplex's built-in branching methods.

Most-Fractional Rounding (Fractional Rounding): One of the simplest MIP branching techniques is to select the variable that has a relaxed LP solution whose fractional part is closest to 0.5 and to round it first.

Most-Fractional Up (Fractional Up): For binary IPs like SPP it has often been noted that rounding the branching variable up first is beneficial. The common understanding is that forcing a variable to 1 will force many other binaries to 0 and thus increases the integrality of many variables when diving in this direction. This in turn may lead to shallower subtrees and integer feasible solutions faster. With this motivation in mind we introduce this branching heuristic which selects the variable with the most fractional LP value and enforce its ceiling as lower bound first.

Best Pseudo-Cost Lower Estimate First (Pseudo Best): One of the most substantial contributions to search in mixed integer programming was the discovery that the running average of the per unit cost-deprivation of prior branching decisions on a variable provides a very good estimate of the per unit deprivation that we are likely to encounter when imposing a new branching constraint on a variable. In practice it was found that these pseudo-costs (M. Benichou 1971) are surprisingly accurate and they are widely used in state-of-the-art MIP solvers.

For this branching heuristic we select the variable that has the lowest pseudo-costs and branch in the direction that is estimated to hurt the objective least first.

Best Pseudo-Cost Up (Pseudo Up): With the same motivation as for most-fractional branching we choose the variable with the lowest pseudo-costs, but this time we always round the variable up first.

Lowest Fractional Set Down (Non-Unary 0): We introduce a new non-unary branching heuristic for SPP that is based on a construction heuristic for capacitated network design problems (Holmberg and Yuan 2000).

Inspired by this primal heuristic we propose the following. We select the variables in the order of lowest fractional LP value, up until the total sum of all fractional values is closest to 0.5. For example, say the variables with the lowest fractional values are X_7 with a current LP value of 0.05, X_1 with a value of 0.1, X_9 with 0.15, and X_3 with 0.9. Then we would select X_7, X_1, X_9 as their sum is 0.3. Had we included X_3 the sum would have been 1.2 which has an absolute difference from 0.5 of 0.7 whereas the sum without X_3 is only 0.2 away. On the other hand, had X_3 had a fractional value of 0.3 we would have included it as the sum is now 0.6 which is only 0.1 away from the desired value of 0.5.

Now, we split the search space by requiring that all variables equal 0, or that their sum is at least 1. We branch in the zero direction first. In the example above, the branching constraints added are $X_7, X_1, X_9 = 0$ first and on backtrack $X_7 + X_1 + X_9 \geq 1$. Note that both constraints are violated by the current LP relaxation.

Highest Fractional Set Up (Non-Unary 1): We use a modification of the previous approach, but this time we focus on the highest variables first. We select, in order, the variables with the highest fractional LP values. For each we consider the "missing fraction" which is 1 minus the current LP value. Again we add these missing fractions until we achieve a sum that is closest to 0.5. In this case, we set all variables to 1 first and on backtrack enforce that their sum is lower or equal the number of variables in the set minus 1.

For example, assume X_4 has a current LP value of 0.95, X_5 0.9, X_2 0.85, and X_3 0.1. Then, we branch by enforcing $X_4, X_5, X_2 = 1$ first. Upon backtracking we add the constraint $X_4 + X_5 + X_2 <= 2$.

Numerical Results

Implementation

We embedded the above heuristics in the state-of-the-art MIP solver Cplex Version 12.1. Note that we only modify the branching strategy by implementing a branch callback function. When comparing with default Cplex we use an empty branch callback to ensure the comparability of the approaches.[1] The empty branch callback causes Cplex to compute the branching constraint using its internal default heuristic. We do not change any other Cplex behavior, the system uses all the standard features like pre-solving, cutting planes, etc. Also, the search strategy, i.e., what open node to consider next, is left to Cplex. Note however that, when Cplex dives in the tree, it considers the first node returned by the branch callback first so that, e.g., it does make a difference whether we round a variable up or down first.

Trace: In our new approach, which we refer to as *Trace*, the branch callback works as follows. First, we compute the features of the current subproblem. We do this by adapting the new upper and lower bounds on variables as given by Cplex in an internal data structure which incrementally adjusts the 33 feature values. We do this for two reasons. First, it is difficult to efficiently get access to the internal pre-solved problem from Cplex. Secondly, due to cuts added and our non-unary branching constraints the resulting MIP will in general no longer be a pure set partitioning problem for which the features were defined. By using Cplex' bounds on the variables at least we benefit from any inferences that Cplex may have conducted which may allow it to set variables to one of their bounds even if these variables were not branched on in the path from the root to the current node.

We determine how to branch by using the normalized features to find the nearest cluster center and use the heuristic associated with that cluster. To learn a good assignment of heuristics to clusters offline we employed GGA. On our training set with 300 instances learning took 28 CPU days.

We will compare Trace with the Cplex default as well as each of the pure heuristics (with *pure* we mean that we choose one heuristic and stick to it throughout the search), each of which we use to solve all our instances. This comparison by itself is what is commonly seen in operations research papers when it comes to determining the quality of branching heuristics.

Online Best Cluster Approach (OBCA): We add two more approaches to our comparison. The first is the online best cluster approach (OBCA). OBCA determines offline which pure heuristic works best for each cluster. During the search, it determines which cluster is nearest to the current subproblem and uses the associated heuristic for branching. The difference to Trace is that the latter uses GGA to assign heuristics to clusters. Note that Trace may very well assign a heuristic to a cluster that is not the best when used throughout the entire search.

Naturally, OBCA's assignment of heuristics to clusters is highly biased by the instances in those clusters. Consequently, we consider a version of OBCA where we add subproblems to the set of instances which are encountered when solving the original training instances using the pure heuristics. This way, we might hope to give better search guidance for the subproblems that we will encounter during search. We refer to this version as OBCA+

[1]Note that Cplex switches off certain heuristics as soon as branch callbacks, even empty ones, are being used so that the entire search behavior is different.

ISAC

The second approach that we add to our comparison is ISAC. We consider the choice of a pure branching heuristic as a Cplex parameter and use ISAC to determine for which instance it should employ which pure search heuristic. Again, we consider a version where we only use our training instances to determine the clusters, as well as an ISAC+ where we also add the subproblems encountered during search of pure heuristic to the training set.

Benchmark Instances

An important drawback of the research presented here (and algorithm tuning and solver portfolios in general) is that we need enough training instances to allow the system to learn effectively. This is a hard pre-condition that must be met before the work presented here can be applied. Any benchmark set that consists of only a few dozen instances is not enough to allow any meaningful learning. In fact, we would argue that benchmarks with a low number of instances (say, less than 100) can hardly be used to draw any conclusions that would generalize. However, the fact is that a lot of research is still based on such benchmark sets, and often it is even the case that approaches are designed and tested on the very same set of instances. From the research on algorithm tuning we understand in the meantime that results obtained in this way cannot be assumed to generalize to other instances.

To access a meaningful number of both training and test instances we developed an SPP instance generator. Unlike most generators in the literature, we designed the generator in such a way that it produces a highly heterogeneous set of benchmark instances. The generator first picks uniformly at random a number of bags between 200 and 2000, as well as a number of items between 20 and 50. It then flips three fair coins. The first coin determines whether the costs for each bag are chosen uniformly at random between 1 and 1,000 or whether this random number also gets multiplied by the number of items in each respective bag (making bags with more items more costly in general than bags with low numbers of items). The second coin determines whether, for the instance under construction, all sets will contain the same number of items or whether the number of items for each bag is determined individually by choosing a density uniformly at random between 10% and 30% of the total number of items. The last coin determines how we fill each bag. We either pick a subset of the desired size uniformly at random out of all such subsets, or we cluster the items that get added to bags by choosing a normal distribution around some item index and adding items in the proximity of that target item with higher probability than other items. Finally, to ensure feasibility, for each item the generator adds one bag which contains only that item at a high cost of 10^5.

The generated SPP instances were evaluated with default Cplex. To ensure instances of meaningful hardness, we kept

Training Set 300 instances	Default	Fractional (UP)	Oracle	ISAC	ISAC+	OBCA	OBCA+	Trace
Average (σ)	57.1 (63.2)	44.6 (53.2)	**38.0** (45.1)	41 (48.6)	42 (48.2)	41.1 (47.7)	50.9 (61.9))	38.4 (45.5)
Median	31.9	23.2	18.5	21.1	23.1	20.6	25.7	**17.6**
Min	0.04	0.04	0.02	0.02	0.32	0.04	0.06	0.04
Max	298	299	153	245	256	244	300	241
PAR 10 (σ)	57.1 (63.2)	44.6 (53.2)	**38.0** (45.1)	41 (48.6)	42 (48.2)	41.1 (47.7)	68.9 (247)	38.4 (45.4)
Shifted Geo Mean	43.2	36.2	**32.9**	34.2	35.2	34.7	39.1	33.2
Average # nodes (σ)	46K (61K)	30K (51K)	28K (49K)	30K (51K)	30K (51K)	24K (51K)	30K (51K)	**23K** (51K)
Nodes per second	**806**	673	736	732	715	584	590	599
Solved	300	300	300	300	300	300	298	300
Unsolved	0	0	0	0	0	0	2	0
% Solved	100	100	100	100	100	100	99.3	100

Table 1: Training Results. All times are CPU times in seconds. Timeout was 300 seconds.

the first 500 instances for which Cplex needed at least 1,000 search nodes to solve the instance, but took at most five minutes to solve. We split these 500 instances into 300 training and 200 testing instances. Note that this is a very modest training set. A learning approach like the one presented here will generally benefit a lot from larger training sets with at least 1,000 training instances, especially when the instances exhibit such a great diversity. We limited ourselves to a lower number of training instances for several reasons. First, it makes learning more challenging. Secondly, it is more realistic that only a limited number of training instances would be available (although again, we must assume there are more than a few dozen). Finally, for ISAC+ and OBCA+ we wanted to add subproblems encountered while solving the original 300 training instances using the pure heuristics. Doing so resulted in 14 clusters with a total of over 1,900 instances which were used for training by ISAC+ and OBCA+ as well as to determine the clusters for Trace (recall however that for the latter we do not use the subproblem instances for training by GGA).

Results

We present our experimental results in Table 1 and 2. All experiments were run on Dell PowerEdge M610s, with 16 Xeon 2.4 CPUs and 24Gb of memory.

In the tables, apart from the usual average CPU time, standard deviation, median time, and number of instances solved (timeout was 300 seconds), we also give the Par10 score (a weighted average where unsolved instances are scored with 10 times the timeout) and the shifted geometric mean. The latter is the geometric mean of the runtimes plus 10 seconds, and is used for benchmark sets where runtimes of individual instances can differ greatly. This causes the long running instances to dominate the average runtime comparison. The runtimes are shifted by ten seconds to prevent instances that are solved in extremely short amounts of time to greatly influence the mean – after all, in practice we rarely care whether an instance is solved in 10 or 100 milliseconds.

Considering the pure heuristics first, in all measures we observe that on both training and test set the somewhat simplistic most fractional up and most fractional rounding heuristics fare best and even outperform Cplex' default heuristic. In the table we only present the performance of the best pure heuristic, *Fractional (UP)*.

In the column "Oracle" the tables give the performance of the fastest pure heuristic for each individual instance. This is a natural limit for the ISAC approach that, at the very best, could choose the fastest pure heuristic for each instance.

As we can see, there is significant room for improvement. Unsurprisingly, ISAC is able to realize some of this potential on the training set, yet it does not generalize too well. On the test set it would have been better to just use the best heuristic that we had found on the training set. Adding subproblems to the training set in ISAC+ does not help either.

Considering OBCA, we find that changing the branching heuristic during search is clearly beneficial. OBCA can reduce the runtime by over 15% compared to the Cplex default. Surprisingly, the performance of OBCA+ is much worse. Recall that we added some subproblems to the training set to allow OBCA+ to get a more realistic view into the problems where it will need to make a branching decision. Clearly, this did not work at all. If anything, we misled OBCA by making it consider subproblems it is unlikely to see when the branching heuristic is changed during search.

To avoid exactly this problem, we had first invented Trace. Recall that Trace only considers subproblem instances for clustering purposes, but bases its assignment of heuristics to clusters *solely* on the performance when running one of the original training instances *while* changing the branching heuristics in accordance to the heuristic/cluster assignment during search. As the experimental results show, Trace significantly reduces the runtime by over 20% when compared with the Cplex default. Note that the new branching heuristic is not directly embedded into the system and therefore cannot exploit branching heuristics like strong-branching which requires a tight integration with the solver. In light of this, an improvement by 20% over one of the most efficient Set Partitioning solvers is very encouraging and a proof of concept that dynamic branching strategies can be learned effectively, even on a relatively small heterogenous set of instances.

At the same time, Trace works very robustly. Its standard deviations in runtime are lower that those of any pure branching heuristic, and the spread of runtimes (min to max) is also greatly reduced. None of the instances are barely solved within the allowed timeout which cannot be said for any pure heuristic.

On the other hand, we see that changing heuristics during search imposes a noticeable cost – the number of nodes

Test Set 200 instances	Default	Fractional (UP)	Oracle	ISAC	ISAC+	OBCA	OBCA+	Trace
Average (σ)	46.4 (52.7)	42.2 (51.8)	**35.0** (40.7)	42.4 (50.7)	42.3 (50.4)	38.7 (45.3)	52.9 (61.7)	35.3 (40.3)
Median	24.2	20.4	**18.0**	21.1	21	18.8	28.9	18.9
Min	0.06	0.04	0.04	0.04	0.04	0.04	0.06	0.04
Max	254	267	201	267	267	227	300	183
PAR 10 (σ)	46.4 (52.7)	42.2 (51.8)	**35.0** (40.7)	42.4 (50.7)	42.3 (50.4)	38.7 (45.3)	66.4 (216)	35.3 (40.3)
Shifted Geo. Mean	38.1	35.2	**32.0**	35.6	35.6	33.9	41	32.6
Average # nodes (σ)	44K (69K)	30K (64K)	29K (63K)	30K (63K)	30K (63K)	24K (63K)	30K (63K)	**22K** (64K)
Nodes per second	**949**	711	828	708	710	620	567	623
Solved	200	200	200	200	200	200	199	200
Unsolved	0	0	0	0	0	0	1	0
% Solved	100	100	100	100	100	100	99.5	100

Table 2: Test Set Results. All times are CPU times in seconds. Timeout was 300 seconds.

per second is clearly less due to the costly re-computations of the subproblem features. This is outweighed by a very significant reduction in choice points, though: Trace consistently visits only about 50% of the number of nodes of the Cplex default.

Conclusion

We introduced the idea to use an offline algorithm tuning tool to learn an assignment of branching heuristics to training instance clusters which is used dynamically during search to determine a preferable branching heuristic for each subproblem encountered during search. This approach, named Trace, was evaluated on a set of highly diverse set partitioning instances. We found that the approach clearly outperforms the Cplex default and also the best pure branching heuristic considered here. While not limited by it, it comes very close to choosing the performance of an oracle that magically tells us which pure branching heuristic to use for each individual instance.

We conclude that mixing branching heuristics can be very beneficial, yet care must be taken when learning when to choose which heuristics, as early branching decisions determine the distribution of instances that must be dealt with deeper in the tree. We solved this problem by using the offline algorithm tuning tool GGA to determine a favorable synchronous assignment of heuristics to clusters so that instances can be solved most efficiently.

Our approach requires a reasonable amount of training instances, as well as a number of branching heuristics, and of course meaningful features that can characterize subproblems during search. For practitioners, who actually need their problems to be solved repeatedly, access to a good number of training instances is less of a problem as it poses for academics. For us, the main obstacle of applying Trace to other problems is therefore the definition of good problem features. We are currently working on features for general MIP problems which we hope can alleviate this issue for a wide variety of problems.

References

A. Braunstein, M. Mezard, R. Z. 2005. Survey propagation: An algorithm for satisfiability. *Random Structures and Algorithms* 27:201–226.

A. R. KhudaBukhsh, L. Xu, H. H. H.-K. L.-B. 2009. Satenstein: Automatically building local search sat solvers from components. *Proc. of the 21th Int. Joint Conference on Artificial Intelligence (IJCAI)* 517–524.

C. Ansotegui-Gil, M. Sellmann, K. T. 2009. A gender-based genetic algorithm for the automatic configuration of solvers. *Proc. of the 15th Int. Conference on the Principles and Practice of Constraint Programming (CP)* 142–157.

Gagliolo, M., and Schmidhuber, J. 2006. Learning dynamic algorithm portfolios. *Annals of Mathematics and Artificial Intelligence* 47(304):295–328.

Gomes, C., and Selman, B. 2001. Algorithm portfolios. *Artificial Intelligence Journal* 126(1-2):43–62.

Hamerly, G., and Elkan, C. 2003. Learning the k in k-means. *Neural Information Processing Systems, MIT Press, Cambridge*.

Holmberg, K., and Yuan, D. 2000. Lagrangean heuristic based branch-and-bound approach for the capacitated network design problem. *Operations Research* 48:461–481.

L. Xu, F. Hutter, H. H. K. L.-B. 2008. Satzilla: Portfolio-based algorithm selection for sat. *JAIR* 32(1):565–606.

Leventhal, D., and Sellmann, M. 2008. The accuracy of search heuristics: An empirical study on knapsack problems. *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR)* 142–157.

M. Benichou, J.M. Gauthier, P. G. G. H.-G. R.-O. V. 1971. Experiments in mixed-integer programming. *Math. Programming 1* 76–94.

Malitsky, Y., and Sellmann, M. 2009. Stochastic offline programming. *Proc. of the 21th Int. Conference on Tools with Artificial Intelligence(IJAIT)*.

S. Kadioglu, Y. Malitsky, M. S. K. T. 2010. Sac - instance specific algorithm configuration. *Proc. of the 19th European Conference on Artificial Intelligence (ECAI)*.

S. L. Epstein, E. C. Freuder, R. J. W. A. M. B. S. 2002. The adaptive constraint engine. *Proc. of the 8th Int. Conference on the Principles and Practice of Constraint Programming (CP)* 525–542.

Samulowitz, H., and Memisevic, R. 2007. Learning to solve qbf. *Proc. of the 22nd National Conference on Artificial Intelligence (AAAI)*.