International Journal on Artificial Intelligence Tools

Stochastic Offline Programming*

Yuri Malitsky
Department of Computer Science, Brown University,
P.O. Box 1910, Providence, RI 02912, U.S.A
ynm@cs.brown.edu

Meinolf Sellmann
Department of Computer Science, Brown University,
P.O. Box 1910, Providence, RI 02912, U.S.A
sello@cs.brown.edu

Received (Day Month Year)
Revised (Day Month Year)
Accepted (Day Month Year)

We propose a framework which we call stochastic off-line programming (SOP). The idea is to embed the development of combinatorial algorithms in an off-line learning environment which helps the developer choose heuristic advisors that guide the search for satisfying or optimal solutions. In particular, we consider the case where the developer has several heuristic advisors available. Rather than selecting a single heuristic, we propose that one of the heuristics is chosen randomly whenever the heuristic guidance is sought. The task of the SOP is to learn favorable instance-specific distributions of the heuristic advisors in order to boost the average-case performance of the resulting combinatorial algorithm. Applying this methodology to a typical optimization problem, we show that substantial improvements can in fact be achieved when we perform learning in an instances specific manner.

Keywords: Parameter tuning; off-line learning; combinatorial optimization.

1. Introduction

Solving hard combinatorial problems efficiently requires search. One of the most difficult tasks when designing combinatorial solvers is to make good heuristic decisions which guide the search. While related communities have only recently started to exploit the potential of automatic tuning (see, e.g., the performance tuning tool in Cplex 11), the constraints community has a long history of research in this direction and has proposed some pioneering ideas for the automatic configuration of algorithms (see, e.g. Ref. 16). This paper follows this tradition by proposing a framework for automatically selecting and combining heuristic advisors. This is achieved

*This work was supported by the National Science Foundation through the Career: Cornflower Project (award number 0644113).

by extrapolating experience gathered off-line by experimenting with a given benchmark of representative problem instances.

Several seminal papers have advocated the idea of exploiting statistics and machine learning technology to increase the efficiency of combinatorial algorithms. One such approach protocols during the solution process of a constraint satisfaction problem which variable assignments cause a lot of filtering and bases future branching decisions on this data.[19] This technique, called *impact-based search*, is one of the most successful in constraint programming and has become part of the Ilog CP Solver. SatZ,[14] a very successful systematic solver for SAT, uses the propagation impact to determine the next branching variable. Other solvers compute or estimate the solution density that results for a subproblem after a potential variable assignment.[22] For incomplete solvers, *reactive tabu search* adapts parameters of a tabu search algorithm online.[3] Another algorithm was proposed that learns online which starting points for simple local search heuristics (such as hill-climbing) lead to good solutions.[5] With respect to offline learning, systems have been developed that automatically tune algorithm parameters for a given set of benchmark instances.[1,12] Moreover, so-called *algorithm portfolios* have also been introduced. It was proposed to run in parallel (or interleaved on a single processor) multiple stochastic solvers that tackle the same problem.[8] This approach was shown to work much more robustly than a single solver, an insight that has led to the technique of randomization with restarts which is commonly used in all state-of-the-art complete SAT solvers. An alternate but similar idea considers a set of algorithms for a given problem and bases the decision of which algorithm to employ on certain features of the given problem instance, whereby informative features and their correlation with the goodness of the algorithms in the portfolio (such as shortage of running time) are learned offline.[17]

All these approaches have two things in common. First, it is possible to construct *worst-case* scenarios where any statistical inference method fails completely. For instance, consider impact-based search for solving SAT. Take any two SAT formulae α, β, both over variables $x_1, \ldots, x_n$. Let us introduce a new variable x_0 and add $\vee x_0$ to all clauses in α and $\vee \bar{x_0}$ to all clauses in β. The SAT problem we want to solve is the conjunction of all modified clauses in α and β. Say we branch on variable x_0 and set it to false first. Impact-based search gathers statistics in the resulting left subtree to guide the search in the right subtree. However, after setting x_0 to false for the left subtree, the resulting problem is to find a satisfying assignment for α. In the right subtree, we set x_0 to true, and the task is to find a satisfying assignment for β. Since α and β were chosen independently from one another, it is not reasonable to assume that the statistics gathered when solving α are in any way meaningful for the solution of β. And obviously, α and β can be chosen in such a way that the statistics gathered when solving α are completely misleading when solving β.

The second aspect that the statistical approaches have in common is that they have all led to very impressive improvements in practice – despite the above worst-

case argument. That is to say, there is substantial *practical evidence* that exploiting (online or offline) statistical knowledge can boost the *average-case* performance of combinatorial solvers. In some sense one may argue that the very fact that statistical inference does not work in the worst-case is what *makes* it statistical inference. That is because, if we could draw any hard conclusions, we would revert to deterministic inference and filter variable domains or derive new redundant constraints. However, statistical inference only kicks in when our ability to reason about the given problem deterministically is exhausted.

In this paper, based on the above mentioned practical evidence, we introduce a framework which exploits offline learning as part of the programming process.

2. Algorithm Families and Instance-Specific Tuning

Our approach is motivated and heavily builds upon the previous work on algorithm portfolios[8,17] as well as the previous work on automatic parameter tuning and automatic algorithm configuration.[1,12] Particularly, we introduce *stochastic offline programming* which intertwines the development of combinatorial algorithms with *automated, instance-specific* selection of an algorithm from an entire family of algorithms.

Our vision is that the algorithm development must not be limited to one fixed algorithm. Instead, the developer ought to have the freedom to propose an entire family of algorithms. Then, provided with a method which associates any given benchmark problem instance with a vector of meaningful feature values, machine learning technology learns which algorithm in the family is best suited for a given instance. The final procedure solving the combinatorial problem therefore computes the features of the given instance and then picks which algorithm to employ accordingly.

We call this very general framework stochastic offline programming (SOP) since it makes automated offline learning part of the programming process. The adjective 'stochastic' is used because the learning is based on the assumption that the given training instances and their associated feature vectors are indicative of the instances that the resulting algorithm will be used on later. That is, we assume that the given benchmark instances are sampled from the same distribution as the real instances, and thus provide us with information on the kind of instances we ought to expect in the future. Note that our goal to boost the average-case performance requires the specification of an instance distribution, and in SOP we assume that this distribution is given indirectly by a sample of training instances and their associated feature vectors.

SOP is similar to and yet differs in some essential ways from existing approaches. Algorithm portfolios like SATzilla[21] for example, suffer from having to evaluate the performance of each algorithm before settling on the best one for a specific instance. This exhaustive evaluation limits the approach to work with only a few algorithms, relying on the *accidental variance* in algorithm performance on different input in-

stances. In contrast, SOP is meant to consider an entire family of algorithms (usually infinitely many) with the developer actively *providing* variance through instance-specific algorithm performance. The idea to consider an entire family of algorithms resembles the situation of tuning algorithm parameters, as done by algorithms like ParamILS.[12] The main difference of our framework is that algorithm parameters are not chosen universally and instead, the choice of parameters depends on the specific features of the instance that needs to be solved.

3. SOP for Homogeneous Benchmarks

We consider the following scenario: Given a combinatorial problem, the developer has devised an algorithm which, at some point, needs to make heuristic decisions. For example, a branch-and-bound algorithm where a branching assignment needs to be chosen at every choice point in the search tree. Or, the developer may have devised a constructive greedy algorithm which, at every step, uses a heuristic criterion to assign the value of a new variable.

Now, rather than choosing one heuristic out of the several available heuristics, we propose that the developer leaves the choice of the heuristic *at every step* to chance. That is, when a new choice must be made, one of the heuristics is chosen according to a predefined *distribution of heuristics (DoH)*. The objective of stochastic offline programming is then to automatically devise an algorithm which chooses the DoH that is best suited for a given problem instance. To this end, SOP is provided with the combinatorial algorithm, an algorithm which associates an input instance with a vector of instance-characterizing features, as well as a set of training instances.

3.1. *Selecting a DoH for Homogeneous Instances*

We first consider the case when the benchmark instances are homogeneous in the sense that the feature vectors associated with the instances show no (or only very little) variance. Out of the given algorithm family, we are then to select one algorithm which works well for a set of these homogeneous instances. This setting differs from algorithm portfolios in that we are dealing with an infinite number of potential DoHs, which renders infeasible approaches which try to learn the performance for each algorithm in the portfolio. On the other hand, the homogeneous benchmark setting of SOP also differs from and is strictly simpler than standard parameter tuning, since we know that the parameters are associated with the probability of selecting each heuristic. We are therefore able to exploit this information by expecting that small changes in the DoH will likely result in small changes in algorithm performance.

We propose Algorithm 1 to compute a good DoH for a set of homogeneous benchmark instances. The procedure presented is provided with an algorithm family 'A' for a combinatorial problem as well as a set 'S' of training instances which are considered to have similar features. Upon termination, the procedure returns a DoH for the given algorithm and benchmark set.

```
1: SOP-Homogeneous (Algorithm A, BenchmarkSet S)
2: distr ← RandDistr()
3: λ_l ← (√5−1)/(√5+1), λ_r ← 2/(√5+1)
4: while termination criterion not met do
5:    (a, b) ← ChooseRandPair(), m ← distr_a + distr_b
6:    X ← λ_l, Y ← λ_r
7:    L ← 0, R ← 1, length ← 1
8:    p_X ← Σ_{i∈S} Perf(A,distr[a=m X,b=m (1 − X)], i)
9:    p_Y ← Σ_{i∈S} Perf(A,distr[a=m Y,b=m (1 − Y)], i)
10:   while length > ε do
11:      if p_X < p_Y then
12:         p_Y ← p_X
13:         R ← Y, length ← R − L
14:         Y ← X, X ← L + λ_l length
15:         p_X ← Σ_{i∈S} Perf(A,distr[a=m X,b=m (1 − X)], i)
16:      else
17:         p_X ← p_Y
18:         L ← X, length ← R − L
19:         X ← Y, Y ← L + λ_r length
20:         p_Y ← Σ_{i∈S} Perf(A,distr[a=m Y,b=m (1 − Y)], i)
21:      end if
22:   end while
23:   distr ← distr[a=m X,b=m (1 − X)]
24: end while
25: return distr
```

Algorithm 1: SOP for Homogeneous Benchmarks

The problem of computing this favorable DoH 'distr' can be stated as a continuous optimization problem: Minimize$_{\text{distr}} \sum_{i \in \text{S}}$ Perf(A,distr,i) such that 'distr' is a probability distribution over the advisors used by 'A.' To solve this problem, we employ a local search procedure. We initialize 'distr' randomly. In each iteration, we randomly pick two heuristic advisors a, b and redistribute their joint probability mass 'm' among themselves while keeping the probabilities of all other advisors the same.

We heuristically expect that the one-dimensional problem which optimizes which percentage of 'm' is assigned to advisor a (the remaining percentage is then already determined to go to advisor b) is convex. We search for the best percentage using a method for minimizing one-dimensional convex functions over closed intervals which is based on the golden section (see Figure 1): We consider two points $X < Y$ within the interval $[0, 1]$ and measure their performance 'p_X' and 'p_Y.' The performance at X is assessed by running the algorithm 'A' on the given benchmark with distribution 'distr [a=m X,b=m $(1 - X)$]', which denotes the distribution resulting from 'distr'

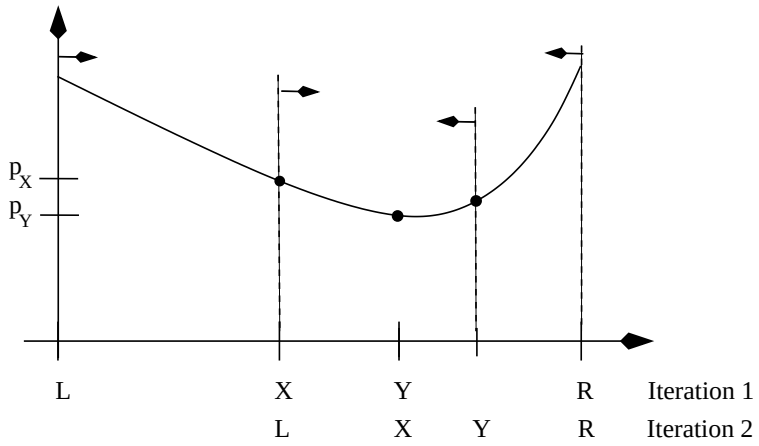

Fig. 1. Minimizing a One-Dimensional Convex Function by Golden Section.

when assigning probability mass 'Xm' to advisor 'a' and probability mass '$(1-X)$m' to advisor 'b'. Now, if the function is indeed convex, if $p_X < p_Y$ ($p_X \geq p_Y$), then we know that the minimum of this one-dimensional function lies in the interval $[0, Y]$ ($[X, 1]$). We continue splitting the remaining interval (which shrinks geometrically fast) until the interval size 'length' falls below a given threshold 'ϵ.' By choosing points X and Y based on the golden section, in each iteration we only need to evaluate one new point rather than two. Moreover, the points considered at each iteration are reasonably far apart from each other to make a comparison meaningful which is important for us as our function evaluation may be noisy (due to the randomness of the algorithm invoked) and points very close to each other are likely to produce very similar results.

3.2. *A Greedy Heuristic for Set Covering*

To make the discussion less abstract let us consider a specific example of an optimization problem and algorithm whose average-case performance we hope to boost by computing a favorable distribution of heuristic advisors. Given *items* $1 \dots n$ and a set of sets of these items, which we call *bags*, and a cost associated with each bag, the *set covering problem (SCP)* consists in finding a set of bags such that the union of all bags contains all items and the cost of the selection is minimized.

Note: As with any other tuning algorithm, our benchmark is **not** the set covering problem but the specific complete or incomplete algorithms for its optimization. Just like (Ref. 12) where the benchmarks were solvers for SAT, and not SAT itself, here we show how the performance of two concrete algorithms for set cover can be boosted.

Greedy Algorithm: A simple greedy algorithm for the SCP is to select bags one by one until we cover all items. Several heuristics have been proposed in the literature on how the next bag ought to be selected. For instance, we can select the bag

- that costs the least ($\min c$),
- that covers the most new items ($\max k$),
- that minimizes the ratio of costs over the number of newly covered items ($\min c/k$),
- that minimizes the ratio of costs over newly covered items times the logarithm of newly covered items ($\min \frac{c}{k \log k}$),
- that minimizes the ratio of costs over the square of newly covered items ($\min \frac{c}{k^2}$),
- and the bag that minimizes the ratio of square root of costs over the square of newly covered items ($\min \frac{\sqrt{c}}{k^2}$).

Greedy algorithms like this are frequently used to find high quality solutions within complete solution approaches. The traditional way of designing algorithms is to try all greedy variants on some training problems and to select the one which yields the best results on average. But an alternate an innovative idea was presented by Balas and Carrera.[2] Rather than choosing just one greedy variant or choosing a random variant each time the greedy algorithm is employed, it was suggested to choose randomly the heuristic for selecting the next bag *within* the greedy algorithm. It was reported that significantly improved solutions are found when this randomized greedy algorithm is run 30 times every time the complete algorithm calls the primal heuristic. This approach is outlined in Algorithm 2. Note that, if

```
1: SCP-Greedy (S_1, ..., S_m, c_1, ..., c_m)
2: bestValue ← ∞
3: for i = 1 ... 200 do
4:    solution ← ∅, cost ← 0
5:    while ⋃_{i∈solution} S_i ≠ {1, ..., n} do
6:       r ← PickAdvisor()
7:       j ← SelectBag_r(solution, S_1, ..., S_m, c_1, ..., c_m)
8:       solution ← solution ∪ {j}
9:       cost ← cost + c_j
10:   end while
11:   if bestValue > cost then
12:      bestValue ← cost
13:      bestSolution ← solution
14:   end if
15: end for
16: return bestSolution
```

Algorithm 2: Randomized Greedy Set Covering Algorithm

we are to run the greedy construction 30 times anyway, we could also run each of the pure heuristics of which there are only 6. The significance of the finding by Balas and Carrera[2] is that hybridizing the heuristics by choosing one of them uniformly at random at each step of the greedy construction and repeating this randomized construction 30 times yields results which are *better* than running the best pure heuristic. We test Algorithm 1 on this randomized greedy algorithm. Rather than choosing a selection heuristic uniformly at random in Lines 6-7, we intend to learn which distributions of selection heuristics are most promising.

In our experiments we found that the 30 iterations proposed by Balas and Carrera often resulted in high variance in the quality of the outcomes. This variance then led to very noisy data, complicating the effectiveness of Algorithm 1 in finding the global optima. Therefore in all our experiments we instead repeat the randomized construction 200 times.

Benchmark Sets: To experiment with our SOP framework, we require training and test sets for the SCP. We consider three different randomly generated benchmark sets whereby each set has instances with 100 items and 10,000 bags each.

- **Set 1:** Each bag contains exactly 4% of the items which are chosen uniformly without replacement. The cost of each bag is chosen uniformly at random between 1 and 1,000.
- **Sets 2:** For each item, for each bag we flip a coin and with a probability of 8% we insert the item into that bag. The cost of each bag is chosen uniformly at random between 1 and 1,000.
- **Sets 3:** For each bag, we repeatedly sample an item from the set of items for insertion into the bag. We assume the items are numbered and that the sample is taken from a Gaussian distribution which has its mean at a random (yet fixed) item. The standard deviation of this distribution is 50. The process of sampling and inserting items is repeated until the bag contains exactly 4% of the items. Again, the cost of each bag is chosen uniformly at random between 1 and 1,000.

For each of the three classes above, there are really two benchmark sets, one for training and one for test purposes, and each containing 100 SCP instances.

Experimental Results: In each iteration of Algorithm 1 we assume that the one-dimensional subproblem that is solved by the golden section is convex. In Figure 2 we plot the average solution quality (over 5 runs of Algorithm 1 for each DoH) that is achieved when mixing heuristics advisors "$\min c$" and "$\min c/k^2$" on an instance in Set 2. The error bars give the standard deviation over 5 runs at each DoH. With this graph we confirm our assumptions that a probabilistic mixture of two heuristics is in fact better than either of the heuristics by themselves. We also show that search space is in fact convex, validating our use of Algorithm 1 to find the best combination of heuristics. We observe curves like the one shown here for all instances and for all combinations of heuristic advisors that we looked at. Note that

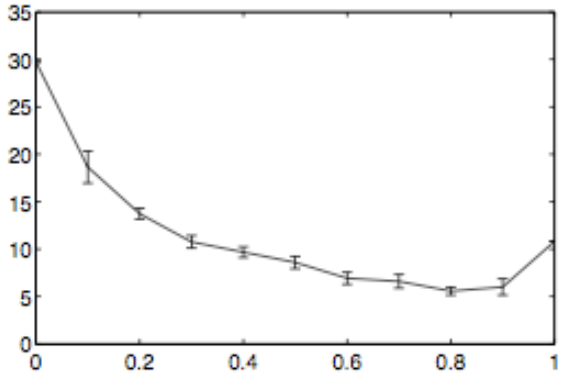

Fig. 2. Optimality Gap when Mixing Advisors "min c" and "min c/k^2."

Algorithm 1 would also work when the convexity assumption was in fact false, but then it may only provide a locally optimal solution. However, for our application we may expect that Algorithm 1 provides near-optimal DoHs.

In Table 1 we compare the solution quality achieved by different DoHs in terms of percent of optimality gap closed. To this end, we pre-computed optimal SCP solutions for all training and test instances in Set 1, Set 2, and Set 3. The initial gap is defined by the solution quality that is achieved when using the single advisor which gives, on average, the best solutions for the training instances. In column "Gap" we observe that the best pure greedy approach leaves an optimality gap between 7% and 9%.

We compare the following choices for the advisor selection in Line 6 of Algorithm 2. "All" performs a round-robin through all advisors. That is, in the ith construction of a greedy solution it always returns the same advisor, and moves on to the next advisor in iteration $i+1$. Note that, for this way of choosing advisors, it is of course not necessary to perform 200 iterations, since there are only 6 different advisors. Clearly, we expect a much better performance from mixing advisors and running the randomized algorithm 200 times to make up for the additional time spent. "Uniform" denotes the algorithm proposed by Balas and Carrera[2] where,

Table 1. Percent of Optimality Gap Closed over the Best Single-Advisor Heuristic. The standard deviations are provided inside the parentheses.

Benchmark		Gap	All	Uniform	SOP	Oracle
Set 1	Train	7.0%	22.8 (2.7)	20.4 (2.3)	41.0 (2.5)	41.1 (1.7)
	Test	7.7%	26.3 (2.6)	24.4 (2.0)	40.5 (1.9)	40.8 (2.3)
Set 2	Train	8.9%	32.6 (3.5)	44.3 (3.2)	57.4 (2.7)	58.8 (2.3)
	Test	8.5%	29.4 (3.5)	46.9 (3.0)	56.0 (2.7)	58.0 (2.9)
Set 3	Train	7.1%	20.8 (3.2)	23.0 (2.4)	39.1 (2.2)	40.0 (2.1)
	Test	8.3%	26.8 (3.2)	25.3 (2.4)	38.5 (2.1)	42.3 (2.0)

in each iteration of the greedy construction, we choose a different advisor by picking one uniformly at random. "SOP" denotes the greedy algorithm which uses the DoH which SOP found for the given training set. Finally, "Oracle" denotes the performance when we use the best DoH for each individual instance which we precomputed and use as an upper bound on the solution quality that can be achieved by the simple greedy heuristic that we are trying to tune.

For Set 1 we observe that running the greedy heuristic 6 times rather than just once while using a different advisor each time and returning the best cover in the end closes around 22% of the optimality gap that the best pure greedy construction left open. Surprisingly, for this benchmark set we do not find it worthwhile to invest the time to construct 200 covers while mixing advisors uniformly at random in the construction of each cover. In fact, the solution quality even slightly declines compared to "All." The situation is similar for benchmark Set 3. Only for Set 2 we find that mixing advisors and spending 194 extra iterations results in significantly better covers.

Looking at column "Oracle," we find that by spending an extra 199 iterations we can expect to close at most around 40% of the optimality gap for Sets 1 and 3, and around 60% for Set 2. Comparing with SOP, we see that the latter comes surprisingly close to realizing this potential! Moreover, when comparing the training and test performances that SOP achieves, we see that extrapolating the off-line experience gained on the training set is feasible and results in only slightly decreased performances on the test sets.

4. SOP for Heterogeneous Benchmarks

We now consider the general case where we are given a set of benchmark instances which show significant diversity with respect to their associated feature vectors. In this scenario we intend to learn a function which associates any given feature vector not represented in the training set with a DoH which we can expect to work well for instances with respective features.

4.1. *Selecting a DoH for Heterogenous Instances*

For this task we tried two different algorithms. The first followed the standard machine learning approach of multinomial logistic regression.[13] The idea aimed to learn a weight for each feature/advisor pair. Then, with the help of those weights, compute a distribution of heuristics (DoH) by taking, for each advisor, the weighted sum of the features of the instance times the respective feature/advisor weights and using this as the argument to the exponential function. This way, each advisor is associated with a weight, and the distribution is then computed through normalizing the weights.

This use of regression to predict the DoH is similar to the work proposed by Hutter *et.al.*[10,11] In their work, Hutter *et.al.* aim to find the parameter setting for a solver that will minimize its expected run time on a specific instance. Offline, the

```
1: SOP-Heterogeneous-Cluster(Algorithm A, BenchmarkSet S, Int k)
2: Init(featureMetric)
3: repeat
4:    [C_1, ..., C_k] ← Cluster(S,k,featureMetric)
5:    for i=1 to k do
6:       f_i ← SOP-Homogeneous(A,C_i)
7:       c_i ← Center(C_i)
8:    end for
9:    for all 1 ≤ i ≠ j ≤ k do
10:      for all instances a ∈ C_i do
11:         d_{a,c_j} ← max{ Perf(A, f_j, a) - Perf(A, f_i, a), 0}
12:      end for
13:   end for
14: until Adjust(featureMetric,d)=false
15: return ([f_1, ..., f_k], [c_1, ..., c_k],featureMetric)
```

Algorithm 3: Cluster-Based SOP for Heterogeneous Benchmarks

algorithm first learns a function $g(f, p)$, that predicts the runtime given a feature vector f that describes the problem instance and a parameter setting p. Therefore for any new instance, f is fixed and the minimum of g is assumed to correspond to the parameter settings that will yield the shortest expected run time. However, while this approach has shown encouraging results, it relies heavily on the assumption that the found parameters will work well together. This is simply not the case for our problem setting. For example, if we equate the parameter setting with our DoH, it is not clear what should be done if a high value is returned for two of the heuristics. This could mean that alternating between the two heuristics will lead to improved performance. On the other hand, it is equally likely that the solver should use only one of the heuristics exclusively. Furthermore, to train accurate prediction models, a large and diverse training set is required. Due to these complications, we move away from regression and propose a second algorithm for the heterogenous case.

Our second algorithm is based on the idea of grouping the benchmark instances in clusters and pre-computing a promising DoH for each cluster. Treating each such cluster as a set of homogeneous instances we can then employ Algorithm 1 which works on a much lower-dimensional space. Though, in order to cluster the instances, we require a distance metric in the feature space. Naturally, the distance between two feature vectors ought to reflect how well a DoH works on instances with respective features. In particular, we want to separate (and thus introduce a large distance between) feature vectors v_1, v_2 where a promising DoH for instances with features v_1 works badly for instances with features v_2, and vice versa.

The idea is realized in Algorithm 3. First we initialize the metric in the feature space arbitrarily (for example using the Euclidean norm). Then, in each iteration we cluster the benchmark set into k groups of instances, whereby we assume that

k is a parameter of the algorithm. To cluster the instances we use Lloyd's k-means clustering algorithm.[15] For each cluster, we compute the optimal DoH using Algorithm 1 as well as the center of gravity. We now assess what the distance between individual instances and the centers of gravity of the different clusters ought to be. To this end, for each pair of clusters $i \neq j$, we compute the difference between the performance on all instances in cluster i which is achieved by the best DoH for that cluster and the DoH of the other cluster.

The distance between an instance a in cluster C_i and the centers of gravity of cluster C_j is then the maximum of this regret and 0. Using these desired distances, we adjust the feature metric and iterate until the feature metric does not change anymore. Then, we return the best DoHs as well as the centers of gravity for each cluster, and the final feature metric. Equipped with this tuple, any formerly unseen instance with associated feature vector can easily be assigned to one of the clusters and the appropriate DoH can then be used for its optimization.

This procedure for finding the best clusters is straight-forward. The only issue is the parameter k which determines the number of clusters and thus the final number of different algorithms which will be used to optimize different instances. We need to strike a good balance here between our wish to provide many different algorithms so that we can provide a tailor-made algorithm for new instances and the need to devise robust algorithms which work well on a variety of instances with the same or similar features. In later sections we will show how to automate the choice of k, but for now the choice of k must be determined heuristically by the user.

4.2. *Tuning a Greedy Heuristic for Heterogenous Benchmarks*

We consider again the greedy heuristic for the SCP and the different advisors from Section 3.2. To make the choice of DoH instance-specific, we require features which characterize an instance.

Features: We gather the following data for SCP instances:

- 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 compute the maxima, minima, averages, standard deviations, and the logarithms of all these statistics for all vectors. To assess the performance achieved by a particular DoH (see the calls to function 'Perf' in Algorithm 3), we run the modified Algorithm 2 again five times and take the average of the solutions returned.

Benchmark Sets: To experiment with our SOP framework, we also require heterogenous training and test sets for the SCP. We consider two more benchmark sets, whereby again each set consists of a training set and a test set of SCP instances.

- **Set 4:** The training set consists in 90 instances, 30 from each of the sets Set 1, Set 2, and Set 3. The test set is the union of all test sets of Set 1-3.
- **Set 5:** For each instance, we first choose at random whether the constraint matrix is filled by row or by column and whether the respective row or column density is constant or variable. Then, we randomly select the density or desired mean density at 4% or 8%, and flip a coin to decide whether row or column densities that are set to one are chosen uniformly at random or with a Gaussian bias. Costs are uniformly chosen from [1,1000]. A training set with 200 instances and a test set with 100 instances are generated this way. Additional test sets are benchmark classes SCP 4, SCP 5, and SCP 6 from the OR library.[18]

Experimental Results: As in Section 3.2, we run various algorithms on our benchmarks, whereby the offline learning algorithms are run on the training set and evaluated on the training as well as the test sets. We compute again the optimality gap of the pure greedy heuristic that works best for the training set. The percentage of this gap that is closed by the contenders is given in Table 2. Again, we observe that using a uniform distribution is often not much better and sometimes even worse than using all of the advisors in a pure round-robin fashion. Our homogenous SOP approach already provides significantly improved DoHs, even though the effect on these heterogenous sets of SCP instances is not as dramatic as seen before on the homogenous benchmarks Set 1-3. In particular, the comparison with the 'Oracle' data where we computed the best DoH for each individual instance shows that there is still room for improvement.

On Set 4 we observe that the heterogenous SOP-clustering approach comes close to realizing the total potential that running the simple greedy algorithm 200 times with a mix of our six different advisors holds in store for us. On the test set that was generated by the same generator as the training set, the results are outstanding. On Set 5, which is much more diverse, we observe that instance-specific parameter

Table 2. Percent of Optimality Gap Closed over the Best Single-Advisor Heuristic. The standard deviations are provided inside the parentheses.

Benchmark		Gap	All	Uniform	SOP-homo	SOP-hetero		Oracle
						Regress	Cluster	
Set 4	Train	7.3%	26.0 (3.0)	30.6 (2.7)	39.6 (2.2)	40.5 (2.4)	46.0 (2.4)	49.2 (2.1)
	Test	8.2%	27.6 (3.1)	30.6 (2.8)	40.5 (2.5)	40.3 (2.5)	45.4 (2.5)	46.0 (2.5)
Set 5	Train	7.2%	22.3 (4.8)	31.9 (3.9)	39.0 (3.3)	32.8 (3.6)	47.7 (2.4)	64.1 (2.5)
	Test	7.6%	30.9 (4.2)	35.0 (4.1)	43.4 (3.6)	38.1 (3.7)	50.3 (3.7)	63.9 (2.9)
	SCP 4	12.4%	25.6 (2.9)	25.0 (2.8)	29.5 (3.3)	41.8 (2.3)	41.1 (2.3)	49.9 (2.2)
	SCP 5	11.8%	34.2 (3.0)	22.5 (1.8)	33.7 (2.5)	35.5 (1.4)	44.9 (2.3)	47.1 (2.4)
	SCP 6	12.4%	20.0 (6.0)	29.6 (3.9)	38.3 (5.9)	34.1 (4.3)	42.7 (3.7)	48.5 (3.5)

tuning significantly outperforms all other algorithms, although it clearly does not achieve the performance of a perfect oracle. What is remarkable is that the quality of the clustering approach does not decline much for the test sets SCP 4-6 which are drawn from the OR library! Moreover, note that both heterogenous SOP approaches keep their performance much more stable and closer to the training set over all four test sets in this benchmark than homogenous SOP.

5. SOP for Tree Search

Now we consider a complete tree search approach. For optimization problems like set covering, systematic approaches are commonly based on branch-and-bound. The latter draws its strength from the upper and lower bounds which are used for pruning purposes. For the lower bound, we use a linear relaxation which is computed by SoPlex 1.4.[20] There exist techniques to strengthen the lower bound, especially by adding valid inequalities. These techniques are outside the scope of this paper. Here, we focus on strengthening the upper bound which is commonly computed by a primal heuristic like the one presented in Algorithm 2. Thus with the help of SOP, we intend to improve the upper bound so that our branch-and-bound approach provides high quality solutions very quickly.

In principle, the situation is similar to that in the previous sections. There are some fundamental differences, though. First, we have seen before that the simple greedy heuristic, even when run multiple times and with high-quality DoHs, leaves an optimality gap of 3%-4%, depending on the benchmark set that is tackled. Embedded in a branch-and-bound approach, we expect the upper bound to converge to (near) optimality with as little search as possible. The second difference is that the primal heuristic is now called for every choice point. The distribution of problem instances encountered at the various choice points is likely to differ a lot from that represented by the input benchmark set.

Consequently, we set up the following algorithm that will be optimized by SOP. At every choice point, we compute an upper bound with the help of Algorithm 2, whereby we resort to the use an iteration limit of 30 as originally proposed by Balas and Carrera.[2] We use the smaller number of iterations instead of 200 in Line 3 since the primal heuristic is being called at every choice point for only slightly varying problems. Also at every choice point, we compute an LP-based lower bound and backtrack if the optimal relaxation is integer or when the global upper bound is already lower or equal to the local lower bound. If that is not the case, we branch and continue search in depth-first manner. For these experiments, the branching decisions are based on a single fixed branching heuristic. For SOP training purposes, we stop this tree-search after 50 nodes and note the final solution quality.

In Table 3 we show the optimality gap left open when using the best single advisor for the training set. We observe that this gap is now much smaller due to the search that is performed by our target algorithm. We also note that the best pure primal heuristic for the training sets of Sets 2 and 3 work comparably well on

Table 3. Percent of Optimality Gap Closed by the Tree-Search Algorithm over the Best Single-Advisor Heuristic for Homogeneous Benchmarks. The standard deviations are provided inside the parentheses.

Benchmark		Gap	All	Uniform	SOP (cluster)	Offline
Set 1	Train	1.05%	-7.39 (1.5)	-0.91 (1.6)	7.02 (1.4)	4.9 (1.4)
	Test	1.61%	16.9 (1.7)	14.4 (1.7)	24.53 (1.5)	20.1 (1.6)
Set 2	Train	1.95%	10.44 (2.2)	15.7 (2.2)	33.14 (1.8)	26.2 (1.9)
	Test	2.07%	17.05 (2.1)	32.3 (1.8)	37.23 (1.8)	36.3 (1.8)
Set 3	Train	1.12%	2.8 (1.5)	10.5 (1.3)	13.2 (1.3)	14.4 (1.4)
	Test	1.17%	1.8 (1.7)	12.1 (1.6)	14.0 (1.5)	10.4 (1.6)

Table 4. Percent of Optimality Gap Closed over the Best Single-Advisor Heuristic using tree search on Heterogeneous Benchmarks. The standard deviations are provided inside the parentheses.

Benchmark		Gap	All	Uniform	SOP (cluster)
Set 4	Train	1.3%	3.7 (1.6)	3.1 (1.7)	21.4 (1.5)
	Test	1.5%	13.7 (1.7)	20.0 (1.7)	25.1 (1.7)
Set 5	Train	2.7%	8.5 (3.0)	19.1 (2.8)	25.3 (1.5)
	Test	3.8%	9.7 (3.9)	18.2 (3.6)	24.8 (3.4)

the corresponding test sets. For Set 1, however, we find that the best pure advisor overtunes: It leaves a rather small optimality gap on the training set but loses a lot of quality (more than 60%) when used on the generalized test set. Note that this effect influences all other approaches for which we measure performance percent of optimality gap closed over the best pure advisor: The training performance of the other approaches appears worse, and their test performance appears better. Overall, we find that SOP (using clustering) outperforms all other approaches, leaving an average optimality gap of 1% to 1.3% after only 50 nodes of search. In column 'Offline' we show the performance when using the DoHs which were found in the experiment corresponding to Table 1. While these DoHs still perform on par or better than a uniform selection of advisors, learning DoHs which perform well within the tree-search framework is clearly better.

We repeat the same experiment for the heterogeneous benchmarks Set 4 and Set 5. In Table 4 we see once more that offline learning based on training instances that are stochastically related to the test cases offers the possibility of significantly boosting the performance of combinatorial algorithms.

6. Enhancements

We have shown that by clustering training instances it is possible to train a solver in an instance-specific manner that will have much better performance than the traditional best single advisor approach. Furthermore, we show that by splitting the training data into clusters and training on each set separately it is even possible to outperform a DoH that was tuned on all of the training data.

There are however two major drawbacks to the SOP methodology as was pre-

sented so far. First, in order to correctly cluster the instances, it is necessary to learn the weights for the distance metric to insure that instances that behave well under a particular DoH are all in the same cluster. This computation is costly as it requires several iterations of tuning and re-clustering.

The second issue is that it is not clear how many clusters to split the data into. If there are too few clusters, we lose some of our potential to tune parameters more precisely for different parts of the instance feature space. On the other hand, if there are too many clusters, there can be very few instances in each cluster which jeopardizes the robustness and generality of the parameter sets that we optimize for these clusters.

In this section we present extensions that resolve these issues. Following the description of the proposed extensions we present experimental results of applying these techniques to SOP.

6.1. *Normalizing Features*

Instead of learning a feature metric over several iterations, we propose to normalize the features so that, over the set of training instances, each feature spans exactly the interval $[-1, 1]$. That is, for each feature there exists at least one instance for which this feature has value 1 and at least one instance where the feature value is -1. For all other instances, the value lies between these two extremes.

Through this normalization, we reduce the influence of the larger valued features. So far in SOP, features with high values (e.g. average cost per bag) tend to dominate the low value features (e.g. average number of items per bag) when it comes to the outcome of the initial clusterings. This is because the distance between two feature vectors can be very large solely because a single feature value in one instance is much larger than the corresponding value in another instance, even though all of the smaller valued features are almost identical.

By scaling the feature space after every iteration, the SOP procedure gradually learns which of the features are important to the clustering. Larger valued features are given smaller weights, while the crucial smaller valued features are given a much higher weight. However, even though it proves to be accurate in our experiments, the drawback is that this process needs several iterations to converge, where each such iteration is computationally expensive.

By normalizing the features before the initial clustering step, all features are given equal importance. Thus when clustering, the algorithm focuses on separating instances where the features vary drastically. In our experiments we found that the resulting clusters are similar to those found at the later stages of the original algorithm, but by using normalization only a single tuning step is required.

6.2. *G-means Clustering*

We have shown that using k-means over the more traditional regression learning yields superior performance. Yet as we stated earlier, the choice of the k used by

1: g-**Means**(X)
2: $k \leftarrow 1$, $i \leftarrow 1$
3: $(C, S) \leftarrow k$-Means(X, k)
4: **while** $i \leq k$ **do**
5: $(\bar{C}, \bar{S}) \leftarrow k$-Means$(S_i, 2)$
6: $v \leftarrow \bar{C}_1 - \bar{C}_2$, $w \leftarrow \sum v_i^2$
7: $y_i \leftarrow \sum v_i x_i / w$
8: **if** Anderson-Darling-Test(y) failed **then**
9: $C_i \leftarrow \bar{C}_1$, $S_i \leftarrow \bar{S}_1$
10: $k \leftarrow k + 1$
11: $C_k \leftarrow \bar{C}_2$, $S_k \leftarrow \bar{S}_2$
12: **else**
13: $i \leftarrow i + 1$
14: **end if**
15: **end while**
16: **return** (C, S, k)

Algorithm 4: g-Means Clustering Algorithm

this approach is crucial for optimal performance.

We address this issue by using g-means,[9] a clustering algorithm which automatically determines a favorable number of clusters. The underlying assumption here is that a good cluster exhibits a Gaussian distribution around the cluster center. The algorithm, presented in Algorithm 4, first considers all inputs as forming one large cluster. In each iteration, we pick one of the current clusters and try to assess whether this cluster is already sufficiently Gaussian. To this end, g-means splits the cluster in two by running 2-means clustering. We can then project all points in the cluster onto the line that runs through the centers of the two sub-clusters. In this way, we obtain a one-dimensional distribution of points. g-means then checks whether this distribution is normal using the widely accepted Anderson-Darling statistical test.[9] If the current cluster does not pass the test then it is split into the two previously computed clusters, and we continue with the next cluster in our current set.

We found that the g-means algorithm works very well for our purposes. The only problem we encountered was that sometimes clusters could be very small and contain only very few instances. Therefore, to obtain robust parameter sets we do not allow clusters that contain fewer than a certain threshold of instances. Once g-means finishes, for those clusters failing this size requirement, beginning with the smallest, we re-distribute the corresponding instances to the nearest bigger clusters, where proximity is measured by the distance of each instance to the center of the bigger cluster.

6.3. *Experimental Results*

Table 5 presents the evaluation of the proposed enhancements on Set 4. Here SOP is the original algorithm that over a series of iterations learns a distance metric to be used by k-means to find the best DoH. SOP-N first normalizes the features of the training sets before proceeding to using k-means to cluster. SOP-N is run for only a single tuning step. Alternatively, SOP-G learns a distance metric over the course of several iterations, but instead of using k-means for re-clustering, it uses g-means. SOP-NG combines the two enhancements and performs only a single iteration where it first normalizes the features and then employs g-means to cluster. For UB, we assume that the clustering algorithm correctly separated all of the instances into three clusters each containing instances only from Set 1, Set 2, and Set 3 respectively.

It is important to note here that simply by normalizing the features once before clustering, we are able to achieve performance comparable to running multiple iterations of SOP until the distance metric converges. Furthermore, by automatically selecting the number of clusters we use, we do not lose out on performance. It is also important to note that all versions of SOP are performing very close to UB, leading to the conclusion that the clusters that are being used are close to optimal.

Table 6 presents the evaluation of the proposed enhancements on Set 5. The algorithm names remain the same as in Table 5, except since we are unable to know the perfect clustering as was the case with UB, we compare the algorithms to Oracle, which computes the best DoH for each instance.

Looking at SOP-N, we notice a drop in performance of the distributions when evaluated on the OR Library benchmarks SCP 4 and 5. This is caused due the equal weight on all of the features forcing some of the instances to be assigned to the wrong cluster. However, we also note that once g-means splits the data into more clusters, the performance on these benchmarks improves.

Table 5. Percent of optimality gap closed over Best-Single Advisor Heuristic on the Set 4 Benchmark. The standard deviations are provided inside the parentheses.

Benchmark		Gap	SOP	SOP-N	SOP-G	SOP-NG	UB
Set 4	Train	7.3%	46.0 (2.4)	44.0 (2.7)	45.2 (2.5)	44.3 (3.3)	45.8 (2.5)
	Test	8.2%	45.4 (2.5)	45.1 (2.8)	43.0 (2.7)	43.9 (2.7)	45.0 (2.2)

Table 6. Percent of optimality gap closed over Best-Single Advisor Heuristic on the Set 5 Benchmark. The standard deviations are provided inside the parentheses.

Benchmark		Gap	SOP	SOP-N	SOP-G	SOP-NG	Oracle
Set 5	Train	7.2%	47.7 (2.4)	44.4 (3.3)	44.0 (2.6)	44.7 (3.4)	64.1 (2.5)
	Test	7.6%	50.3 (3.7)	51.3 (3.7)	50.3 (3.6)	50.4 (3.6)	63.9 (2.9)
	SCP 4	12.4%	41.1 (2.3)	32.2 (3.4)	43.5 (2.2)	43.8 (2.3)	49.9 (2.2)
	SCP 5	11.8%	44.9 (2.3)	22.0 (3.3)	41.5 (2.4)	41.3 (1.9)	47.1 (2.4)
	SCP 6	12.4%	42.7 (3.7)	45.6 (3.9)	36.3 (3.7)	36.0 (4.2)	48.5 (3.5)

Table 7. Percent of Optimality Gap Closed over the Best Single-Advisor Heuristic using tree search on Heterogeneous Benchmarks. The standard deviations are provided inside the parentheses.

Benchmark		Gap	SOP	SOP-NG
Set 4	Train	1.3%	21.4 (1.5)	20.7 (1.8)
	Test	1.5%	25.1 (1.7)	25.3 (1.9)
Set 5	Train	2.7%	25.3 (1.5)	24.4 (1.4)
	Test	3.8%	24.8 (3.4)	26.0 (3.7)

Finally, as can be seen in Table 7, normalizing features and using g-means also correlates to the tree search solver introduced in Section 5.

7. Conclusions and Future Work

We have introduced the idea of stochastic offline programming, a programming framework for automatically choosing and combining different heuristic advisors. The instance-specific randomized combination of advisors is based on offline experience gathered on a training set which is sampled from the same distribution as the test sets that the algorithm is expected to perform well on.

Extensive tests on a greedy incomplete algorithm and a systematic tree search algorithm for the set covering problem provide a proof of concept: It is indeed possible to combine heuristic advisors in a randomized fashion, and favorable instance-specific distributions can be learned which clearly outperform the best pure advisor as well as a uniform combination of advisors. Moreover, on heterogenous benchmarks we found that there is no one best distribution of heuristics (DoH) that works well on all instances. For this case, we showed that we can learn how to select a DoH based on the features of a given problem instance.

Our future work regards the test of SOP on algorithms for other problems. The SOP framework is general enough to cope with any combination of advisors and any objective. For example, we intend to use it to combine branching heuristics to minimize the expected runtime of a constraint solver. As this will require a lot more CPU time that we could afford in the development and calibration of the homogeneous and heterogenous SOP algorithms, we are working towards an efficient parallel implementation of SOP. Moreover, we intend to generalize the framework so that it can learn more than one DoH simultaneously when heuristic guidance is needed for more than one task within a combinatorial algorithm.

References

1. B. Adenso-Diaz and M.Laguna. Fine-tuning of Algorithms using Fractional Experimental Design and Local Search. *Operations Research*, 54(1):99–114, 2006.
2. E. Balas and M. Carrera. A dynamic subgradient-based branch-and-bound procedure for set covering. *Operations Research*, 44:875–890, 1996.
3. R Battiti and G Tecchiolli. The reactive tabu search. *ORSA Journal on Computing*, 6(2):126–140, 1994.

4. M. Birattari, T. Stuetzle, L. Paquete, K. Varrentrapp. A Racing Algorithm for Configuring Metaheuristics. *GECCO*, 11-18, 2002.
5. J.A. Boyan and A.W. Moore. Learning Evaluation Functions to Improve Optimization by Local Search. *Journal of Machine Learning Research*, 1:77–112, 2000.
6. S.P. Coy, B.L. Golden, G.C. Runger, E.A. Wasil. Using Experimental Design to Find Effective Parameter Settings for Heuristics. *Journal of Heuristics*, 7(1):77–97, 2001.
7. A. Fukunaga. Automated discovery of local search heuristics for satisfiability testing. *Evolutionary Computation*, 16(1):31–61, 2008.
8. C. Gomes and B. Selman. Algorithm Portfolios. *Artificial Intelligence*, 126(1–2):43–62, 2001.
9. G. Hamerly and C. Elkan. Learning the K in K-Means. *NIPS* 2003.
10. F. Hutter and Y. Hamadi. Parameter Adjustment Based on Performance Prediction: Towards an Instance-Aware Problem Solver. Technical Report MSR-TR-2005-125, Microsoft Research Cambridge, UK, 2005.
11. F. Hutter, Y. Hamadi, H.H. Hoos and K. Leyton-brown. Performance prediction and automated tuning of randomized and parametric algorithms. *CP 06*, pp. 213–228, 2006.
12. F. Hutter, H. Hoos, T. Stuetzle. Automatic Algorithm Configuration based on Local Search. *AAAI*, 1152–1157, 2007.
13. M. Jordan. Why the logistic function? A tutorial discussion on probabilities and neural networks. *MIT Computational Cognitive Science Report*, 9503:1995.
14. C.M. Li and Anbulagan. Heuristics Based on Unit Propagation for Satisfiability. *IJCAI*, 366–371, 1997.
15. S.P. Lloyd. Least squares quantization in PCM. *IEEE Transactions on Information Theory* 28(2):129–137, 1982.
16. S. Minton. Automatically Configuring Constraint Satisfaction Programs. *Constraints*, 1(1):1–40, 1996.
17. E. Nudelman, K. Leyton-Brown, H.H. Hoos, A. Devkar, Y. Shoham. Understanding Random SAT: Beyond the Clauses-to-Variables Ratio. *CP*, 438–452, 2004.
18. J.E. Beasley. OR-Library. *Journal of the Operational Research Society*, 41(11):1069–1072, 1990.
19. Ph. Refalo. Impact-Based Search Strategies for Constraint Programming. *CP*, pp. 557–571, 2004.
20. R. Wunderling. Paralleler und objektorientierter Simplex-Algorithmus *Technische Universität Berlin*, 1996.
21. L. Xu, F. Hutter, H. H. Hoos and K. Leyton-Brown. SATzilla: Portfolio-based Algorithm Selection for SAT. *JAIR*, Volume 32, pages 565-606, 2008.
22. A. Zanarini and G. Pesant. Solution Counting Algorithms for Constraint-Centered Search Heuristics. *CP*, 743–757, 2007.