# Variable Ordering for Decision Diagrams: A Portfolio Approach*

Anthony Karahalios[0000−0001−9479−4080] and
Willem-Jan van Hoeve[0000−0002−0023−753X]

Carnegie Mellon University, Pittsburgh PA 15213, USA
akarahal@andrew.cmu.edu,vanhoeve@andrew.cmu.edu

**Abstract.** Relaxed decision diagrams have been successfully applied to solve combinatorial optimization problems, but their performance is known to strongly depend on the variable ordering. We propose a portfolio approach to selecting the best ordering among a set of alternatives. We consider several different portfolio mechanisms: a static uniform time-sharing portfolio, an offline predictive model of the single best algorithm using classifiers, a low-knowledge algorithm selection, and a dynamic online time allocator. As a case study, we compare and contrast their performance on the graph coloring problem. We find that on this problem domain, the dynamic online time allocator provides the best overall performance.

**Keywords:** Decision Diagrams · Graph Coloring · Variable Ordering.

## 1 Introduction

Relaxed decision diagrams have recently been successfully applied within a range of solution methodologies for discrete optimization, including constraint programming, integer linear programming, integer nonlinear programming, and combinatorial optimization. For exact decision diagrams (e.g., reduced ordered binary decision diagrams), it is well known that the variable ordering greatly influences the size of the diagram [8, 9, 22]. Likewise, for relaxed decision diagrams the variable ordering is often of crucial importance for their effectiveness. For example, Bergman et al. [2, 3] demonstrate that a variable ordering that yields a small exact diagram typically also provides stronger dual bounds from the relaxed diagram.

In some cases, e.g., for sequential scheduling problems, the variable ordering is prescribed by the sequential nature of the application. In most cases, however, we must design and/or select a variable ordering that we expect to perform well. In the literature several variable ordering strategies, generic as well as problem-specific, have been proposed. When decision diagrams are built from a single top-to-bottom compilation, dynamic variable orderings can be very effective. For

* Partially supported by Office of Naval Research Grants No. N00014-18-1-2129 and N00014-21-1-2240 and National Science Foundation Award #1918102.

example, a recent work by Cappart et al. [10] deploys deep reinforcement learning to dynamically select the next variable during compilation. Dynamic variable orderings are less applicable, however, to compilation via iterative refinement, in which case the ordering must be specified in advance. Oftentimes there is no single variable ordering strategy that dominates all others, and the challenge in practice is to select a strategy that works well for a specific instance. This is a well-studied problem in artificial intelligence, in the context of *algorithm portfolios*.

There are several ways to construct an algorithm portfolio: using static or dynamic features, formulating predictive models at the algorithm or portfolio level, predicting one algorithm to run per instance or creating a schedule of algorithms to run, using a fixed portfolio or updating it online [19]. In this work, as we consider variable ordering strategies for relaxed decision diagrams, our goal is to study which portfolio design leads to the best performance of the diagram.

As a case study, we consider the graph coloring problem, for which a decision diagram approach was recently introduced [16, 17]. It uses an iterative refinement procedure much like Benders decomposition or lazy-clause generation, by repeatedly refining conflicts in the diagram until the solution is conflict free. Our experimental results show several insights, at least for this problem domain: First, even the simplest portfolio (the static uniform time allocation) can already outperform all individual orderings. Second, predictive methods using classification models or exploration phases can lead to more instances solved optimally. However, these methods may lead to delayed optimality results on problem instances that are easy to solve. Third, allocating time to more than one variable ordering can yield a solution with a unique best upper bound from one ordering and a unique best lower bound from a different ordering. This indicates that it may be advantageous to use one variable ordering to obtain a lower bound and another to obtain the upper bound.

## 2 Decision Diagrams

We follow the framework of Bergman, Cire, van Hoeve, and Hooker [4] and introduce decision diagrams as a graphical representation of a set of solutions to a discrete optimization problem $P$ defined on an ordered set of decision variables $X = \{x_1, x_2, \ldots, x_n\}$ and (optionally) an objective function $f(X)$ to be minimized or maximized.

### 2.1 Definitions

A *decision diagram* for $P$ is a layered directed acyclic graph $D = (N, A)$ with node set $N$ and arc set $A$. Diagram $D$ has $n+1$ layers of nodes, where a node in layer $j$ represents a state associated with variable $x_j$. Layer 1 contains a single root node $r$, and layer $n+1$ contains a single terminal node $t$. Arcs are directed from a node $u$ in layer $j$ to a node $v$ in layer $j+1$ and labeled with a decision value for variable $x_j$. The outgoing arcs for each node must have unique labels.

Hence, an arc-specified $r$-$t$ path $p = (a_1, a_2, \ldots, a_n)$ defines a complete variable assignment by setting $x_j$ to be the label of $a_j$ for $j = 1, \ldots, n$. We let $\mathrm{Sol}(D)$ be the set of solutions represented by all $r$-$t$ paths of $D$. We will slightly abuse notation and denote by $\mathrm{Sol}(P)$ the set of feasible solutions to problem $P$. We say that $D$ is an *exact* decision diagram for $P$ if $\mathrm{Sol}(D) = \mathrm{Sol}(P)$. $D$ is a *relaxed* decision diagram for $P$ if $\mathrm{Sol}(P) \subseteq \mathrm{Sol}(D)$.

The objective function $f(X)$ can be represented in $D$ by appropriately associating a 'weight' to each arc in the diagram. We define the weight of an $r$-$t$ path as a function (e.g., the sum) of its arc weights, and require that the weight of the path is equal to the objective value of the solution it encodes. The shortest (or longest) path in $D$ can be computed in linear time since $D$ is acyclic. Such path corresponds to an *optimal* solution if $D$ is exact, and yields a *dual bound* if $D$ is relaxed.

We can extend the application of decision diagrams to let *multiple* paths in $D$ represent the solution to an optimization problem, as proposed in [16, 17]. In that case, an optimal solution can be computed as a constrained network flow. We will use this application in our case study in Section 4.

### 2.2 Compilation Methods

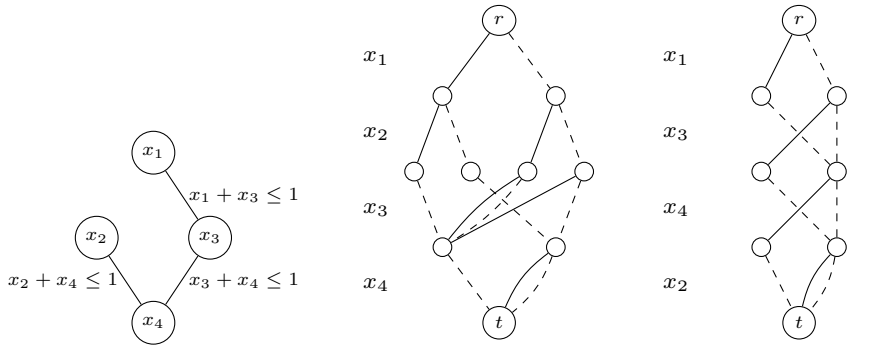
We limit our discussion to the two most popular decision diagram compilation methods in the context of discrete optimization [4]: top-down compilation and iterative refinement. Both methods rely on an underlying recursive formulation of the problem $P$, using states (associated with each node in $N$) and labeled transition functions (represented by the arcs in $A$).

*Top-down compilation* expands the diagram one layer at the time. It considers the nodes (states) in the previous layer, and creates all possible states according to the transition function. Equivalent states are merged. For relaxed decision diagrams, it is typical to impose a maximum size (or 'width') on the layers, in which case non-equivalent nodes may need to be merged. This compilation method can be applied recursively in a branch-and-bound like scheme to obtain an exact solution method.

*Iterative refinement* alternatively starts with an initial relaxed decision diagram in which each layer contains a single node, and all possible arcs between the nodes in subsequent layers are present. The diagram is then iteratively refined by splitting nodes and/or removing infeasible arcs. This is the method of choice for MDD-based constraint propagation, in which case refinement is again limited until a maximum width is reached. It can also be applied as a stand-alone exact solution method, by repeated computation of the optimal solution (which provides a dual bound) and refining any constraints that are violated along the optimal path(s).

### 2.3 Variable Ordering

As mentioned in Section 1, the variable ordering can have a crucial impact on the size of the decision diagram. This is illustrated in the following example:




a. Constraint graph b. Lexicographic ordering b. Path ordering

**Fig. 1.** Constraint graph (a) and exact decision diagrams using the lexicographic variable ordering (b) and the 'path' ordering (c), for the problem in Example 1. In the decision diagrams, dashed arcs represent arcs with label 0, while solid arcs represent arcs with label 1.

*Example 1.* Consider the following constraint satisfaction problem:

$$x_1 + x_3 \leq 1, \quad x_2 + x_4 \leq 1, \quad x_3 + x_4 \leq 1,$$
$$x_1, x_2, x_3 \in \{0, 1\}$$

The constraint graph for this problem is shown in Fig. 1(a). The associated exact decision diagram following the lexicographic variable ordering $(x_1, x_2, x_3, x_4)$ is shown in Fig. 1(b). In Fig. 1(c) we show a smaller exact decision diagram using the path-ordering $(x_1, x_3, x_4, x_2)$ that follows from the constraint graph.

Finding the variable ordering that yields the smallest exact decision diagram is an NP-hard problem [22]. In practice, one therefore typically relies on heuristic variable ordering strategies. An example of a *problem-specific* variable ordering is the *maximal path decomposition* heuristic for compiling the independent sets of a graph [5, 3]. It relies on an a priori computed path decomposition of the input graph, and selects the next variable according to this decomposition. An example of a *generic* variable ordering is the *k-look ahead* ordering [5, 3]. It selects the variable that yields the smallest-width layer when $k = 1$, and evaluates a subset of $k$ variables in general. We will present several more variable ordering heuristics for our case study in Section 4.

The maximal path decomposition heuristic is *static* as the ordering is determined once in advance. In contrast, the $k$-look ahead ordering is *dynamic* because the selection of the next variable is determined during the compilation and depends on the previous choices. Likewise, the reinforcement learning approach of Cappart et al. [10] is a dynamic variable ordering heuristic by design. It uses an action-value function, based on neural fitted Q-learning, to determine the best variable to add to the ordering at each step. Due to its dynamic nature, it can however not be effectively applied when the decision diagram is compiled

using iterative refinement (as in our case study). In our case study, we compose a portfolio of static orderings for settings where iterative refinement is used.

## 3 Algorithm Portfolio Design

Algorithm portfolios have been studied widely in artificial intelligence, and have been shown to be particularly effective for combinatorial optimization and Boolean satisfiability [14, 23, 13]. While many variants exist, most approaches either select one algorithm among a set of alternatives to solve a given problem, or run multiple algorithms (in parallel or sequentially) in dedicated time schedules. Typically one needs to trade off time for exploration (learning the performance of each method) and exploitation (executing the selected algorithm). We refer to Kothoff [19] for a recent survey.

For our purposes we made a selection of four methods from the literature, which contrast offline versus online learning, single versus multiple algorithm selection, and low-level versus high-level knowledge utilization. We assume that we are given a set of variable ordering heuristics (each leading to a different algorithm) and a maximum overall time limit. We explain each portfolio using a contrived example instance in Fig. 2(a). Notice how one variable ordering may have exponentially longer runtime to reach the optimal value compared to other variable orderings.

### 3.1 Static Uniform Time Allocator

This multiple-algorithm selection approach proceeds in rounds; in round $t$, each algorithm is given $2^t$ seconds to solve the problem [13]. We continue until the time limit is reached. As an example, see Fig. 2(b), where the optimal value of 10 will be reached in the round of 64s once V.O. 1 has run for 20 seconds total. So, the total runtime will be (1s+2s+4s+8s+16s+32s+5.25s)=67.25 seconds.

| Runtimes | V.O. 1 | V.O. 2 | V.O. 3 | V.O. 4 |
|---|---|---|---|---|
| 1s | 5 | 5 | 5 | 3 |
| 2s | 5 | 5 | 5 | 6 |
| 4s | 7 | 6 | 5 | 6 |
| 20s | 10 | 8 | 6 | 8 |
| 30s | 10 | 9 | 8 | 10 |
| 60s | 10 | 10 | 8 | 10 |
| 3000s | 10 | 10 | 10 | 10 |

(a) Variable Ordering Runtimes

| Uniform | V.O. 1 | V.O. 2 | V.O. 3 | V.O. 4 |
|---|---|---|---|---|
| 1s | 1/4 | 1/4 | 1/4 | 1/4 |
| 2s | 1/4 | 1/4 | 1/4 | 1/4 |
| 4s | 1/4 | 1/4 | 1/4 | 1/4 |
| 8s | 1/4 | 1/4 | 1/4 | 1/4 |
| 16s | 1/4 | 1/4 | 1/4 | 1/4 |
| 32s | 1/4 | 1/4 | 1/4 | 1/4 |
| ... | ... | ... | ... | ... |

(b) Uniform Time Allocator

**Fig. 2.** An example of lower bounds for 4 different variable orderings at various runtimes, where the optimal value is 10, and the distribution of runtimes using a uniform time allocator.

### 3.2 Offline Predictive Models Via Classifiers

This approach uses classification models to predict the optimal algorithm to run on a given problem instance [23, 20]. As input, the method requires several easily computable features of a problem instance and logic to label the best algorithm for an instance given performance data. These features and labels are computed for a training dataset, then discretized using MDL with Kononenko's criteria, and a greedy forward feature selection process. Pairwise products are computed for this subset of features as in a similar work by Xu et al. [23]. Then, the same discretization and feature selection process is performed to obtain the final features and labels used to train classification models. Several classification models can be applied including Bayesian Networks (BN), Decision Trees (DT), k-Nearest Neighbor (kNN), Multilayer Perceptrons (MP), Random Forests (RF), and Support-Vector-Machines (SVM). The trained classification model is used to select one algorithm from the portfolio to solve a given test instance. For the example in Fig. 2(a), suppose the model takes $t$ seconds to predict V.O. 1. The total runtime will be $t + 20$s. If alternatively the model predicts V.O. 3, then the runtime will be $t + 3000$s. This demonstrates two things: $t$ affects the overall performance of the predictive model approach, and the predictive model choosing one single variable ordering could be detrimental.

### 3.3 Low-Knowledge Single Algorithm Selection

This is a single-algorithm selection method that runs in two phases [1]. An exploration phase runs each algorithm for a time $t$, and then an exploitation phase chooses one algorithm to run for the remaining time based on the results of the exploration phase. In [1], three prediction rules for the exploitation phase are proposed: pcost_max (select algorithm with best lower bound), pslope_mean (maximum mean of the change in the best lower bound), and pextrap (extrapolate pcost_max with pslope_mean to find the maximum lower bound at the time limit). Ties are broken by choosing the ordering with the best mean performance at the time limit for the training data. For each prediction rule, the optimal time $t$ to use on the testing data is found by running $t = 10, 20, \ldots, 300$ on the training instances and choosing the $t$ that gives the maximum number of optimal lower bound results. For the example in Fig. 2(a), suppose the model trains for 30 seconds with each variable ordering. Using pcost_max, the model would choose the ordering with the better mean performance on the training data between V.O. 1 and V.O. 4 as these have the highest bounds, pcost_slope would choose V.O. 4 as it has the highest mean change of 0.2 per second if we start from an offset of 10 seconds, and pcost_extrap would choose V.O. 4 by calculating the highest extrapolated value of 10 + 0.2*(timeout - 30s).

### 3.4 Dynamic Online Time Allocator

This is a multi-algorithm selection method following a dynamic online schedule [13]. It proceeds in rounds, such that round $t$ has a limit of $2^t$ seconds. We

initially assign to each algorithm a share of the runtime. After each round, the time share for each algorithm is updated based on a function of the problem instance features, the current runtime for each algorithm, and the performance of each algorithm. For our purposes, we use an updating function with three parameters: maximum lower bound (lb_bonus), maximum change in lower bound (delta_bonus), and a tie parameter (tie_bonus) that encourages reversion to the uniform time allocator. Given a time share allocation ($vo_1$, $vo_2$, ..., $vo_k$) at the beginning of a round, this function adds lb_bonus to the $vo_i$ for the variable ordering $i$ with the maximum lower bound at the end of the round. Similarly, delta_bonus adds to the maximum change in lower bound from the beginning of the round to the end of the round. In the case of any ties, the bonus is divided evenly amongst the tied variable orderings. In the case that all variables tie for both lb_bonus and delta_bonus, tie_bonus is added to all $vo_i$. After adding bonuses, all $vo_i$ are re-normalized so that they sum to 1. Similar to the Low-Knowledge method of [1], we use the training instances to tune the parameters of the updating function to use on the test instances. For the example in Fig. 2(a), the table in Fig. 3 shows the distribution of runtimes for each round. Suppose lb_bonus=delta_bonus=tie_bonus=1. Then, no bonus is given until each variable ordering runs for 1 second, where then the lb_bonus and delta_bonus are split between V.O. 1, V.O. 2, and V.O. 3, creating the distribution for the 8s round. Then, V.O. 1 receives the lb_bonus as it passes 4s total runtime with a bound of 7 while V.O. 4 receives the delta_bonus as it passes 2s total runtime with a bound of 6. The next round would use the tie_bonus, as no variable ordering improves in the 16s round. This portfolio reaches the optimal lower bound at 39.8 seconds in the 32s round when V.O. 1 reaches 20 seconds of runtime.

| Dynamic | V.O. 1 | V.O. 2 | V.O. 3 | V.O. 4 |
|---|---|---|---|---|
| 1s | 1/4 | 1/4 | 1/4 | 1/4 |
| 2s | 1/4 | 1/4 | 1/4 | 1/4 |
| 4s | 1/4 | 1/4 | 1/4 | 1/4 |
| 8s | 11/36 | 11/36 | 11/36 | 3/36 |
| 16s | 47/108 | 11/108 | 11/108 | 39/108 |
| 32s | 155/540 | 119/540 | 119/540 | 147/540 |

**Fig. 3.** The distribution of runtimes using the Dynamic Online Time Allocator.

## 4 Case Study: Graph coloring

We next apply the variable ordering portfolios for decision diagrams to graph coloring as a case study. Given a graph, the graph coloring problem is to minimize the number of colors necessary to color all vertices such that no vertices sharing an edge have the same color.

A decision diagram approach for graph coloring was proposed in [16, 17], using iterative refinement based on conflict resolution. The decision diagram

represents the independent sets (color classes) of the graph, where each layer corresponds to a vertex of the input graph. That is, each $r$-$t$ path in the decision diagram correspond to a color class defined by the vertices that take an arc with label 1 in the path. A graph coloring solution consists of a set of color classes such that each vertex belongs to one color class. To find such solution, we can define a network flow optimization model on the decision diagram, that 1) minimizes the total amount of flow out of the root node, while 2) ensuring that in each layer at least one arc with label 1 is traversed. The optimal network flow solution thus corresponds to a collection of $r$-$t$ paths that 'cover' all vertices. If one of these paths contains a conflict, the decision diagram is refined accordingly, and a new network flow solution is computed. This process iterates until a conflict-free solution is found or a stopping criterion is met. The experimental results in [17] demonstrate that the performance of this approach relies strongly on the variable ordering, which makes this a relevant case study for our portfolio approach.

### 4.1 Variable Orderings

We consider the following six variable orderings, the first three of which were also studied in [17]:

**Lexicographic:** Order the variables as they are input into the problem.

**Maximum Connectivity/Degree:** Add vertices one at a time, choosing the one with the maximum number of dependencies already in the ordering, and the one with the largest degree as a tie-breaker [16].

**DSATUR:** Use the classic graph coloring heuristic from Brélaz [7].

**Maximal Paths:** Use a maximal path decomposition to order the variables [2]. Start by considering the variables in the order they were entered. While not all vertices are in the ordering, choose the first unchosen vertex and create a maximal path, adding vertices to the ordering as they are added to the path, then remove that maximal path from the graph. To create the maximal path, choose an unchosen vertex that is adjacent to the most recent vertex added to the path, or if this does not exist, add an unchosen vertex adjacent to the first vertex in the path until there are no more possible vertices to add.

**Maximal Cliques:** Use a maximal clique decomposition to order the variables. Sort the vertices from largest degree to smallest degree. Construct a maximal clique decomposition on the graph of variable dependencies by choosing one vertex at a time, and then the first neighbor in the adjacency list that maintains a clique if one exists. Order the vertices by starting with the largest clique, and continue with cliques that share as many edges with the previous clique as possible.

**Minimum Width:** Apply a variable ordering with minimum *width*, that is, the maximum number of dependencies for a variable that come before that variable in the ordering [12]. The algorithm is described in Algorithm 1.

In our evaluation, we will refer to the above orderings as 'lex', 'max_degree', 'dsatur', 'max_path', 'max_clique', and 'min_width', respectively. We note that

**Algorithm 1** Minimum Width Variable Ordering Algorithm

**Input:** Graph $G = (V, E)$
**Output:** Ordered list of vertices $L$
**Definition:** $\deg(v, G)$ is the degree of $v$ in $G$.

```
L ← ∅
while V not empty do
  N ← argmin_{v∈V} {deg(v, G)}
  V ← V − N
  E ← E − {(i, v) : (i, v) ∈ E, v ∈ N}
  L ← N:L {add N to front of L}
  G ← (V, E)
end while
return L
```

the latter two orderings have not been applied before to decision diagram compilation, to the best of our knowledge. We give details for constructing each type of portfolio below.

### 4.2 Algorithm Portfolios

**Static Uniform Time Allocator** For the uniform time allocator, the order the heuristics run in each round is: min_width, max_clique, dsatur, max_degree, max_path, lex. This order was chosen based on which variable orderings solved the most instances of the Dimacs benchmark set within a 3600s time limit.

**Offline Predictive Models Via Classifiers** We used Culberson's random instance generator [11] to generate 432 graphs. We generated 4 graphs of each type in the cross product of $n$=(100, 250, 500, 1000), density=(0.1, 0.5, 0.9), embedded colorings of (0, 10, 20), (0, 25, 50), (0, 25, 100) and (0, 50, 100) for each $n$ respectively, and variability=(0, 1) when the embedding does not equal 0. We use 3 graphs of each type as a training set (324 graphs), and the 4th graphs as a testing set (108 graphs). We ran each algorithm on these graphs for a maximum of 1,800 seconds. We also used a set of 137 graphs from the coloring and clique part of the well-established Dimacs Challenge [18] as another, completely independent, test set. The Dimacs experiments ran with a time limit of 3,600 seconds.

For the features, we calculate 50 characteristics of each problem instance. We use a subset of the features from Musliu and Schwengerer [20], by including only these categories: graph size features, node degree statistics, maximal clique statistics, local clustering coefficient statistics, weighted local clustering coefficient statistics, and dsatur greedy coloring statistics. Graph Size features and node degree statistics use their common definitions. Maximal clique uses a simple greedy algorithm for each node. Clustering coefficients use their classic definition [21], and weighted clustering coefficients multiplies each clustering coefficient for a node by its degree. DSATUR runs the common algorithm mentioned earlier

in this paper. Problem instances were labelled with a best algorithm based first on maximum lower bound, then best time to the best lower bound, and then most instances solved to optimality. To simplify parameter configuration for the classification models, we used parameters recommended in [20]. For the BN, the maximum number of parent nodes is set to 5. For the DT, the minimum number of objects per leaf was set to 3. For kNN, the size of the neighborhood is set to 5. For the RF, the number of trees was set to 15. For the MP and SVM, and other remaining parameters, we used the default settings from the Weka system [6].

**Low-Knowledge Single Algorithm Selection** We order the variables as we did in the Static Uniform Time Allocator, and we use the training set from the Offline Predictive Models Via Classifiers to find the best parameter $t$. When determining the mean change for pslope_mean, we consider the interval from 10 seconds to the end of the training phase, as all variable orderings start with a lower bound of 1.

**Dynamic Online Time Allocator** We order the variables as we did in the Static Uniform Time Allocator, and we use the training set from the Offline Predictive Models Via Classifiers to find the best set of bonus parameters.

## 5 Experimental Evaluation

All variable orderings and iterative refinement algorithms are written in C++. The data evaluation scripts are written in Python, using a wrapper around the Weka data mining library version 1.0.6 for the machine learning models used [6]. Following previous studies, we assume an "ideal" machine with no task switching overhead [13]. Therefore, our experiments were run for each single variable ordering, and this data was compiled to simulate each portfolio method. All experiments were run on an Intel Xeon 2.33GHz CPU with Ubuntu 18.04. We will evaluate each algorithm (i.e., the individual variable orderings and portfolios) in terms of their performance: how many instances can be solved within a given time limit? We first consider the performance of the individual variables orderings, and then assess each of the four portfolio approaches from Section 3.

### 5.1 Performance of individual variable orderings

Before running our portfolio methods, we confirm that none of the individual orderings always dominates the others, and that each ordering can be the best for at least one instance. We show that this is the case for both the Test Culberson instances and Dimacs instances through Figure 4(a), which plots the frequency that a variable ordering achieved the best bound amongst all variable orderings within three time ranges of when the quickest variable ordering achieved that bound. For the 108 Test Culberson instances, any one variable ordering achieves the best lower bound within 60 seconds of the quickest variable ordering to achieve this bound for less than 80 instances. Similarly, this number is 100 of the 137 Dimacs instances.

A more detailed comparison of the individual variable orderings is given in Fig. 5 by presenting their performance plots, i.e., the number of instances solved by a given time limit. We show these plots separately for the test Culberson instances (a) and the Dimacs instances (b). The best performing individual variable orderings for the Culberson instances are dsatur and max_degree (both solve 46 instances). While max_degree solves 46 instances in 1200 seconds, dsatur solves 46 instances in 1740 seconds. For the Dimacs instances the min_width ordering performs best (solving 54 instances). The min_width ordering also performs best overall, solving 98 instances in total compared to 96 for the runner up max_degree.

### 5.2 Experiment 1: Static Uniform Time Allocator

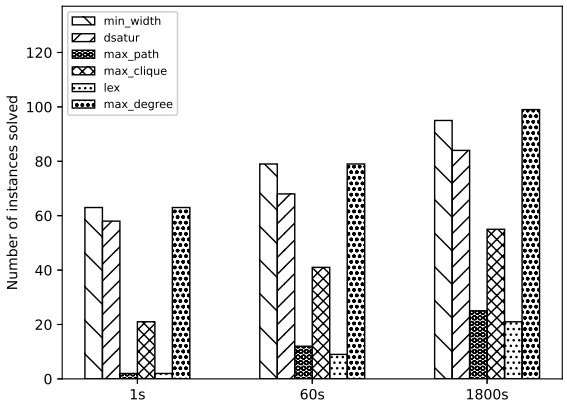
We included this method as a baseline comparison. Despite its simplicity, the uniform time-sharing portfolio solves 55 Dimacs instances optimally, and solves more Dimacs instances in faster times than all of the variable orderings individually, as can be seen in Fig. 5(b). This method also works well, but not as well, on the Test Culberson instances, as presented in Fig. 5(a). In both cases, there is at least one instance that a hypothetical 'oracle' portfolio, which selects the best variable ordering for each instance can solve, but the uniform portfolio cannot.

### 5.3 Experiment 2: Offline Predictive Models Via Classifiers

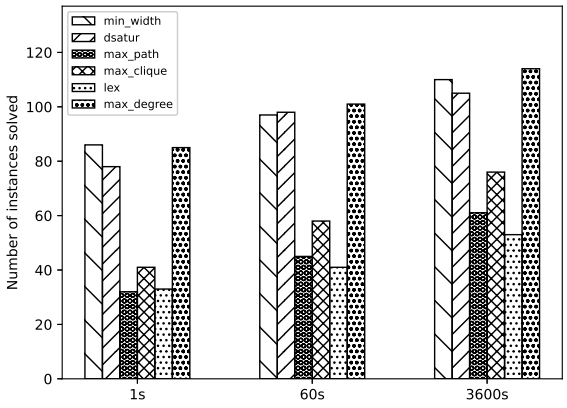
The predictive model used a greedy forward feature selection which chose 28 features (4 basic features and 24 product features) ranging over all of the feature categories (the same features were used for all models). The Multilayer Perceptrons model (MP) took 10 minutes to train, while the other models needed less than one minute to train, so we chose to not include results for MP. All of the testing took less than a second. Among all instances, the median time taken to compute all features for an instance is 1 second, the 75th percentile is 19 seconds, and the maximum is 2562 seconds. Most classifiers showed similar performance as seen in Fig. 6. Random Forests (RF) performed best for Culberson instances solving 45 instances, and Bayesian Network (BN) for Dimacs instances solving 55 instances. The results highlight the fact that the models are trained on Culberson data, so the Culberson test results simulate a user having access to results from a similar problem set, while the Dimacs results simulate a user lacking similar training data.

### 5.4 Experiment 3: Low-Knowledge Single Algorithm Selection

Recall that for six orderings and a time limit $T$, the training phase for this method takes $6 * t$ seconds, while the the final selected algorithm runs for a total of $t + (T - 6 * t)$ seconds. As stated before, we use $T = (3600, 1800)$ for the Dimacs and Culberson Test sets respectively. Based on the results of the training data,



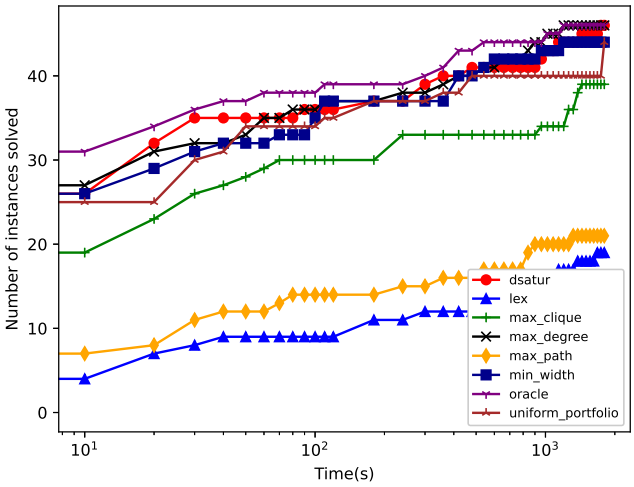

(a) Test Culberson instances



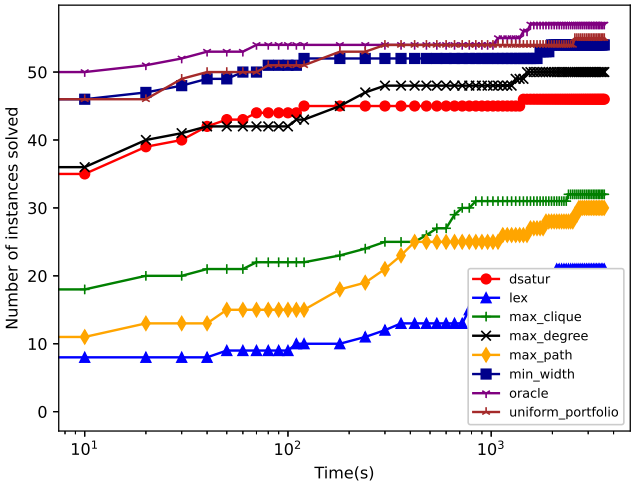

(b) Dimacs instances

**Fig. 4.** The frequency that a variable ordering yields the best lower bound within a time range of (1s, 60s, 1800/3600s) from the fastest time of any ordering, for the Test Culberson instances (a) and the Dimacs instances (b).

we set $t = (30, 10, 10)$ for pcost_max, pslope_mean, and pextrap respectively. We present the performance for the three possible settings for this type of portfolio in Fig. 7. We see that pcost_slope performs best for Culberson instances solving 46 instances, while pcost_max performs best for Dimacs instances, solving 56 instances. The choice of 10 seconds for pslope_mean will always select the top tie-breaker option (which was dsatur), because there is no slope to calculate

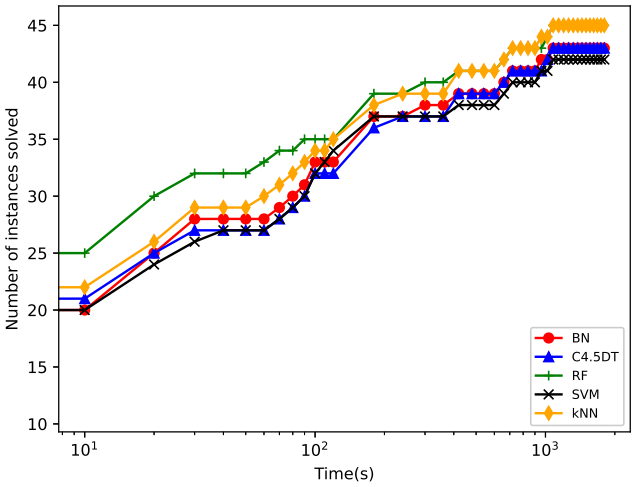



(a) Test Culberson Instances

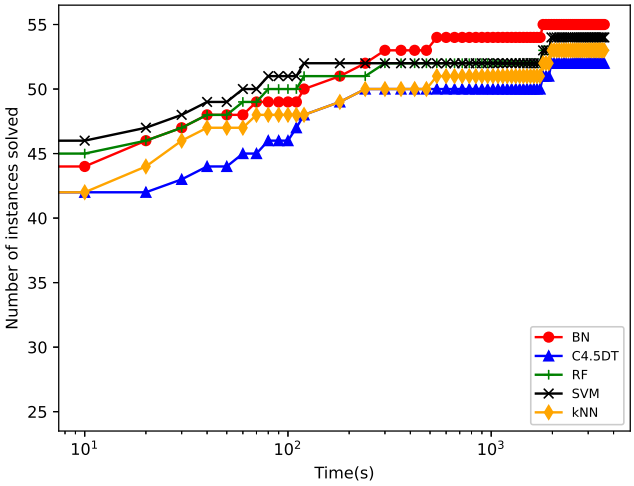



(b) Dimacs Instances

**Fig. 5.** The number of instances solved to optimality within $t$ seconds for each variable ordering, the oracle, and the uniform time-sharing portfolio. The time is in log-scale.



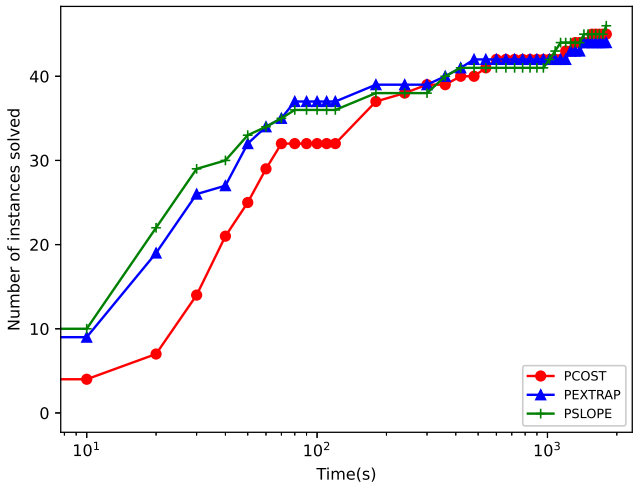

(a) Test Culberson Instances



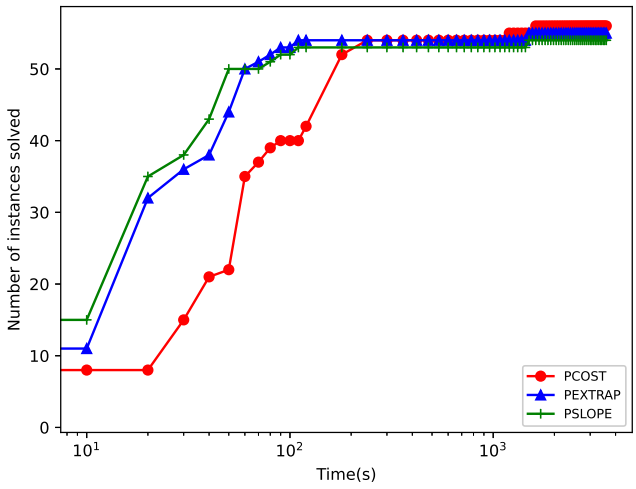

(b) Dimacs Instances

**Fig. 6.** The number of instances solved to optimality within $t$ seconds for each type of classifier used in the predictive method and the oracle. The time is in log-scale.

with only one bin, so we use pcost_max for both types of instances in our final comparisons. The results show that while this type of portfolio may take a while to train, its late performance looks strong for both types of instances.




(a) Test Culberson Instances




(b) Dimacs Instances

**Fig. 7.** The number of instances solved to optimality within $t$ seconds for each function that can be used in the low knowledge portfolio and the oracle. The time is in log-scale.

### 5.5 Experiment 4: Dynamic Online Time Allocator

We ran this method on the training data using values of (0, 2, 4, 6) for each possible bonus value. Based on those results, for the testing sets we used lb_bonus = 6, delta_bonus = 6, and tie_bonus = 6. These large yet equal bonuses made it quick to either converge to an optimal allocation share or revert back to the uniform distribution. The overall comparison in the next section includes these results.

### 5.6 Overall Comparison

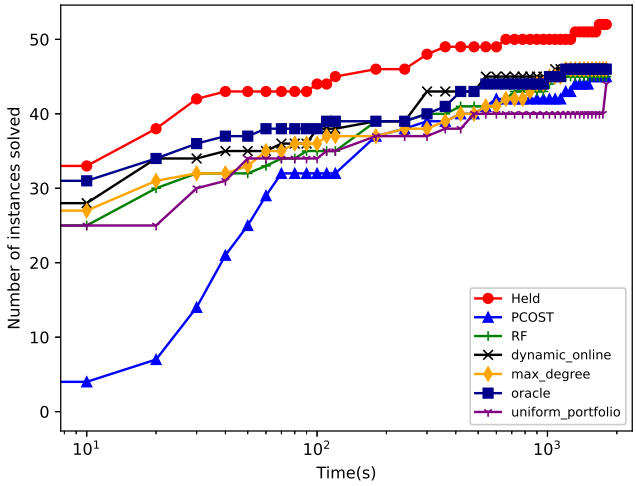
**Table 1.** Number of instances solved to optimality within three time limits for the Culberson and Dimacs instance types. We compare the best single variable ordering, each of the portfolio approaches, the oracle, and the method by Held et al. [15].

| | Culberson | | | Dimacs | | |
|---|---|---|---|---|---|---|
| | 100s | 750s | 1,800s | 100s | 1,000s | 3,600s |
| Single Variable Ordering (max_degree/min_width) | 36 | 44 | 46 | 51 | 52 | 54 |
| Static Uniform Time Allocator | 35 | 40 | 44 | 51 | 54 | 55 |
| Offline Predictive Model (RF/BN) | 35 | 43 | 45 | 49 | 54 | 55 |
| Low-Knowledge Single Algorithm (PCOST) | 32 | 42 | 45 | 40 | 54 | 56 |
| Dynamic Online Time Allocator | 37 | 45 | 46 | 53 | 53 | 55 |
| Oracle | 39 | 44 | 46 | 54 | 54 | 57 |
| Held et al. | 44 | 50 | 52 | 53 | 56 | 60 |

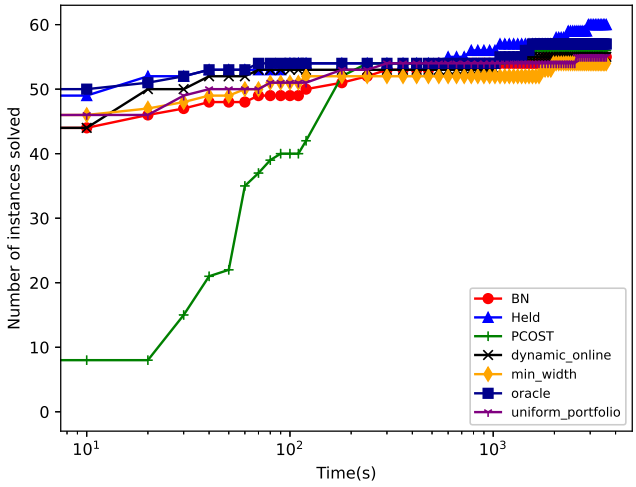
Lastly, we compare the performance of the best settings for each type of portfolio against the oracle, the best performing individual ordering (max_degree, resp. min_width), and the state-of-the-art graph coloring solver by Held et al. [15]. The latter solver is based on integer linear programming, and implements a branch-and-price algorithm.[1] The performance plot for each method is given in Fig. 8 for the Culberson instances (a) and Dimacs instances (b). In addition, Table 1 presents the number of instances solved to optimality within three different time limits for each algorithm and each instance type.

Table 1 shows that the best performing portfolios at the time limit are PCOST and dynamic_online for the Culberson instances (solving 46 instances), and PCOST again for the Dimacs instances (solving 56 instances). Fig. 8 furthermore shows that also in terms of overall performance (across varying time limits and for both instance types), both the low-knowledge PCOST and dynamic online portfolios perform well. However, one may favor the dynamic online portfolio because the low-knowledge method is often slower to reach optimality due to

[1] The code has been downloaded from `https://github.com/heldstephan/exactcolors`.




(a) Test Culberson instances




(b) Dimacs instances

**Fig. 8.** The number of instances solved to optimality within $t$ seconds for the best performing individual variable ordering (max_degree/min_width), the best setting for each portfolio method, the oracle, and the state-of-the-art code by Held et al. [15], for the Test Culberson instances (a) and the Dimacs instances (b).

its training phase. Notice that for the Culberson instances, the dynamic_online portfolio solves the same number of instances as the single variable ordering max_degree. More granularity shows that the dynamic_online portfolio solves 46 instances in 1,080 seconds while the max_degree ordering takes 1,200 seconds. This indicates that for a set of similar graph instances, one variable ordering might work just as well as a portfolio, but when the set of instances is more diverse like in the Dimacs set, the portfolio can become more helpful.

The predictive method shows slightly stronger relative performance to the other portfolio methods on the Culberson instances than for the Dimacs instances, especially looking at 100s. This is likely because the training set consists of Culberson instances. Also notice from comparing the performance of the dynamic online portfolio and the oracle at 750s for Culberson instances that portfolios using more than one ordering can even outperform the oracle when one variable ordering finds the best lower bound and another finds the matching upper bound.

In a comparison to the state of the art, the dynamic online portfolio improves the performance of the decision diagram approach to be competitive with Held et al. for the Dimacs instances. However, overall Held et al. solve more instances within the time limit. For the Culberson instances, the approach by Held et al. clearly outperforms the best portfolio approach.

## 6 Conclusion

We presented a portfolio approach to selecting the best variable ordering for relaxed decision diagrams in the context of combinatorial optimization. We considered four approaches: uniform time allocation, predictive modeling, a low-knowledge selection procedure, and a dynamic online time allocator. We compared the performance of these methods on the graph coloring problem, and find that even the simplest portfolio (uniform time allocation) already outperforms all individual orderings for the Dimacs benchmark set of instances. The dynamic online time allocator showed the best overall performance. As it can combine lower and upper bounds from different orderings, it is even able to outperform an oracle that selects the best single ordering for each instance.

## References

1. Beck, J.C., Freuder, E.C.: Simple rules for low-knowledge algorithm selection. In: International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming. pp. 50–64. Springer (2004)
2. Bergman, D., Cire, A.A., van Hoeve, W.J., Hooker, J.N.: Variable Ordering for the Application of BDDs to the Maximum Independent Set Problem. In: Proceedings of CPAIOR. LNCS, vol. 7298, pp. 34–49. Springer (2012)
3. Bergman, D., Cire, A.A., van Hoeve, W.J., Hooker, J.N.: Optimization Bounds from Binary Decision Diagrams. INFORMS Journal on Computing **26**(2), 253–268 (2014)

4. Bergman, D., Cire, A.A., van Hoeve, W.J., Hooker, J.N.: Decision Diagrams for Optimization. Springer (2016)
5. Bergman, D., Cire, A.A., Van Hoeve, W.J., Hooker, J.N.: Variable ordering for the application of bdds to the maximum independent set problem. In: International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming. pp. 34–49. Springer (2012)
6. Bouckaert, R.R., Frank, E., Hall, M., Kirkby, R., Reutemann, P., Seewald, A., Scuse, D.: Weka manual for version 3-9-1. University of Waikato, Hamilton, New Zealand (2016)
7. Brélaz, D.: New methods to color the vertices of a graph. Communications of the ACM **22**(4), 251–256 (1979)
8. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Transactions on Computers **C-35**(8), 677–691 (1986)
9. Bryant, R.E.: Symbolic boolean manipulation with ordered binary decision diagrams. ACM Computing Surveys **24**, 293– 318 (1992)
10. Cappart, Q., Goutierre, E., Bergman, D., Rousseau, L.M.: Improving Optimization Bounds Using Machine Learning: Decision Diagrams Meet Deep Reinforcement Learning. In: Proceedings of AAAI. pp. 1443–1451. AAAI Press (2019)
11. Culberson, J.C., Luo, F.: Exploring the k-colorable landscape with iterated greedy. Cliques, coloring, and satisfiability: second DIMACS implementation challenge **26**, 245–284 (1996)
12. Freuder, E.C.: A sufficient condition for backtrack-free search. Journal of the ACM (JACM) **29**(1), 24–32 (1982)
13. Gagliolo, M., Schmidhuber, J.: Algorithm portfolio selection as a bandit problem with unbounded losses. Annals of Mathematics and Artificial Intelligence **61**(2), 49–86 (2011)
14. Gomes, C.P., Selman, B.: Algorithm portfolios. Artificial Intelligence **126**(1–2), 43–62 (2001)
15. Held, S., Cook, W., Sewell, E.C.: Maximum-weight stable sets and safe lower bounds for graph coloring. Mathematical Programming Computation **4**(4), 363–381 (2012)
16. van Hoeve, W.J.: Graph Coloring Lower Bounds from Decision Diagrams. In: Proceedings of IPCO. LNCS, vol. 12125, pp. 405–419. Springer (2020)
17. van Hoeve, W.J.: Graph coloring with decision diagrams. Mathematical Programming pp. 1–44 (To appear)
18. Johnson, D.S., Trick, M.A.: Cliques, coloring, and satisfiability: second DIMACS implementation challenge, October 11-13, 1993, vol. 26. American Mathematical Soc. (1996)
19. Kotthoff, L.: Algorithm selection for combinatorial search problems: A survey. In: Data Mining and Constraint Programming, pp. 149–190. Springer (2016)
20. Musliu, N., Schwengerer, M.: Algorithm selection for the graph coloring problem. In: International Conference on Learning and Intelligent Optimization. pp. 389–403. Springer (2013)
21. Watts, D.J., Strogatz, S.H.: Collective dynamics of 'small-world'networks. nature **393**(6684), 440–442 (1998)
22. Wegener, I.: Branching Programs and Binary Decision Diagrams: Theory and Applications. SIAM monographs on discrete mathematics and applications, Society for Industrial and Applied Mathematics (2000)
23. Xu, L., Hutter, F., Hoos, H.H., Leyton-Brown, K.: Satzilla: portfolio-based algorithm selection for sat. Journal of artificial intelligence research **32**, 565–606 (2008)