# Variable Ordering for Decision Diagrams: A Portfolio Approach*

Anthony Karahalios 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: an offline predictive model of the single best algorithm using classifiers, an online low-knowledge algorithm selection, a static uniform time-sharing portfolio, 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.

## 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, 20]. Likewise, for relaxed decision diagrams the variable ordering is often of crucial importance for their effectiveness. For example, Bergman et al. [2, 4] 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 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,

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

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 [16]. 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 [19, 18]. 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 a few key insights. 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. 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 [5] 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 an objective function $f(X)$ to be minimized or maximized.

**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 [19, 18]. 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.

**Compilation Methods** We limit our discussion to the two most popular decision diagram compilation methods in the context of discrete optimization [5]: 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).

**Variable Ordering** Finding the variable ordering that yields the smallest exact decision diagram is an NP-hard problem [20]. 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 [3, 4]. 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 [3, 4]. 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 approach, we can compose a portfolio of static and/or dynamic orderings, and may restrict ourselves to static orderings in case compilation is done via iterative refinement.

## 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, 21]. 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 [16] 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), as well as a maximum overall time limit.

**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.

**Offline Predictive Models Via Classifiers** This approach uses classification models to predict the optimal algorithm to run on a given problem instance [21, 17]. 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. 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 algoritm from the portfolio to solve a given test instance.

**Online 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). 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.

**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. 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.

## 4 Case Study: Graph coloring

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 [19, 18], 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. The optimal solution corresponds to a collection of $r$-$t$ paths that cover all vertices. We use the following six variable orderings in our portfolio:

**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 [19].
**DSATUR:** Use the classic graph coloring heuristic from Brélaz [7].
**Maximal Paths:** Use a maximal path decomposition to order the variables [2].
**Maximal Cliques:** Use a maximal clique decomposition to order the variables.
**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].

In our evaluation, we will refer to these orderings as 'lex', 'max_degree', 'dsatur', 'max_path', 'max_clique', and 'min_width', respectively. We note that the latter two orderings have not been applied before to decision diagram compilation, to the best of our knowledge.

**Training and Testing Data** 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 1800 seconds. We also used a set of 137 graphs from the coloring and clique part of the well-established Dimacs Challenge [15] as another, completely independent, test set. The Dimacs experiments ran with a time limit of 3600 seconds.

**Uniform Time Allocator Ordering** For the uniform time allocator, the order the heuristics run in each round is: min_width, max_clique, dsatur, max_degree, max_path, lex.

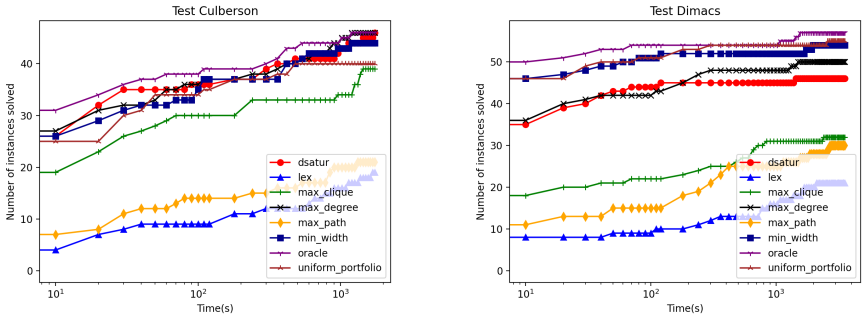
**Predictive Model Features** For the predictive model, we calculate 50 characteristics of each problem instance. We use a subset of the features from Musliu and Schwengerer [17], by including only these categories: graph size features, node degree statistics, maximal clique statistics, local clustering coefficient statistics, weighted local clustering coefficient statistics, and dsatur greedy coloring statistics. 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 [17].

**Dynamic Time Allocator Update Function** 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.

## 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 evaluate the variable orderings (and portfolios) in terms of their performance: how many instances can be solved within a given time limit? Before running our portfolio methods, we confirmed that none of the individual orderings always dominates the others, and that each ordering can be the best for at least one instance. We also compare the portfolios with the hypothetical 'oracle' portfolio, which selects the best variable ordering for each instance. Fig. 1 presents the performance profile (number of instances solved within a given time) for each of the individual orderings, as well as the oracle. The figure shows that an algorithm portfolio has the potential to solve more instances to optimality, and in quicker times than any one algorithm on its own. We next present an assessment of each of the four portfolio approaches from Section 3.



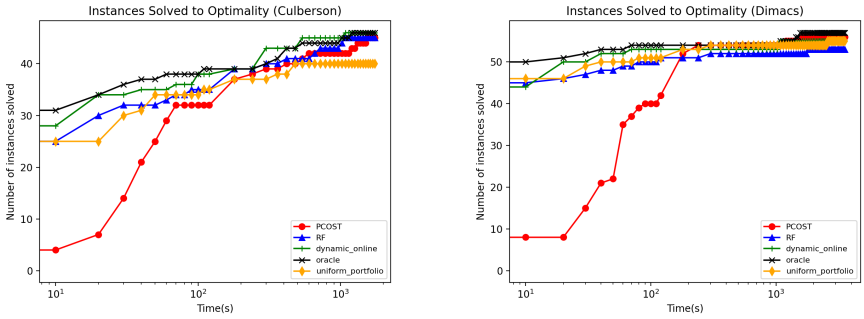

(a) Test Culberson Instances (b) Dimacs Instances

**Fig. 1.** 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.

**Experiment 1: Uniform time-sharing portfolio** 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. 1(b). This method also works well, but not as well, on the Test Culberson instances, as presented in Fig. 1(a).

**Experiment 2: Predictive model** 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 MP took 10 minutes to train, while the other models needed less than one minute to train. 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, with the Random Forests (RF) as the overall best performing method. We present its performance (relative to the other portfolios) in Fig. 2. Note 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. This discrepancy is apparent in the Fig. 2, where the RF's predictions show good late runtime performance for Culberson, but not for Dimacs.

**Experiment 3: Low-knowledge 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 $T = (1800, 3600)$ for the Dimacs and Culberson Test sets respectively. Based on the results of the training data, we set $t = (30, 10, 10)$ for pcost_max, pslope_mean, and pextrap respectively. Among these, pcost_max (select the ordering with the best lower bound) performed best. We present its performance in Fig. 2. It is apparent that although the 30-second




(a) Test Culberson Instances (b) Dimacs Instances

**Fig. 2.** The number of instances solved to optimality within $t$ seconds for one setting of each portfolio method in each category. 'RF' stands for the predictive model using random forests. 'PCOST' stands for the low-knowledge method using best lower bound.

exploitation period slows the initial progress of pcost_max, it later shows best performance, with 56 instances solved for the Dimacs instances, more than any of the other portfolio approaches.

**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. Fig. 2 shows that this method is overall the best performing portfolio. Since this method can combine lower and upper bounds from different variable orderings, it can prove optimality earlier than individual orderings, and can even outperform the oracle (Fig. 2)(a)). However, it solves one fewer Dimacs instance (55 total) than pcost_max.

## 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. 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. We hope that this study will help make decision-diagram based optimization methods more robust in general.

## References


1. J. C. Beck and E. C. Freuder. Simple rules for low-knowledge algorithm selection. In *International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming*, pages 50–64. Springer, 2004.
2. D. Bergman, A. A. Cire, W.-J. van Hoeve, and J. N. Hooker. Variable Ordering for the Application of BDDs to the Maximum Independent Set Problem. In *Proceedings of CPAIOR*, volume 7298 of *LNCS*, pages 34–49. Springer, 2012.
3. D. Bergman, A. A. Cire, W.-J. Van Hoeve, and J. N. Hooker. 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*, pages 34–49. Springer, 2012.
4. D. Bergman, A. A. Cire, W.-J. van Hoeve, and J. N. Hooker. Optimization Bounds from Binary Decision Diagrams. *INFORMS Journal on Computing*, 26(2):253–268, 2014.
5. D. Bergman, A. A. Cire, W.-J. van Hoeve, and J. N. Hooker. *Decision Diagrams for Optimization*. Springer, 2016.
6. R. R. Bouckaert, E. Frank, M. Hall, R. Kirkby, P. Reutemann, A. Seewald, and D. Scuse. Weka manual for version 3-9-1. *University of Waikato, Hamilton, New Zealand*, 2016.
7. D. Brélaz. New methods to color the vertices of a graph. *Communications of the ACM*, 22(4):251–256, 1979.
8. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.
9. R. E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, 24:293– 318, 1992.
10. Q. Cappart, E. Goutierre, D. Bergman, and L.-M. Rousseau. Improving Optimization Bounds Using Machine Learning: Decision Diagrams Meet Deep Reinforcement Learning. In *Proceedings of AAAI*, pages 1443–1451. AAAI Press, 2019.
11. J. C. Culberson and F. Luo. Exploring the k-colorable landscape with iterated greedy. *Cliques, coloring, and satisfiability: second DIMACS implementation challenge*, 26:245–284, 1996.
12. E. C. Freuder. A sufficient condition for backtrack-free search. *Journal of the ACM (JACM)*, 29(1):24–32, 1982.
13. M. Gagliolo and J. Schmidhuber. Algorithm portfolio selection as a bandit problem with unbounded losses. *Annals of Mathematics and Artificial Intelligence*, 61(2):49–86, 2011.
14. C. P. Gomes and B. Selman. Algorithm portfolios. *Artificial Intelligence*, 126(1–2):43–62, 2001.
15. D. S. Johnson and M. A. Trick. *Cliques, coloring, and satisfiability: second DIMACS implementation challenge, October 11-13, 1993*, volume 26. American Mathematical Soc., 1996.
16. L. Kotthoff. Algorithm selection for combinatorial search problems: A survey. In *Data Mining and Constraint Programming*, pages 149–190. Springer, 2016.
17. N. Musliu and M. Schwengerer. Algorithm selection for the graph coloring problem. In *International Conference on Learning and Intelligent Optimization*, pages 389–403. Springer, 2013.
18. W.-J. van Hoeve. Graph Coloring with Decision Diagrams. *Under review*. `http://www.optimization-online.org/DB_HTML/2021/01/8215.html`.

19. W.-J. van Hoeve. Graph Coloring Lower Bounds from Decision Diagrams. In *Proceedings of IPCO*, volume 12125 of *LNCS*, pages 405–419. Springer, 2020.
20. I. Wegener. *Branching Programs and Binary Decision Diagrams: Theory and Applications*. SIAM monographs on discrete mathematics and applications. Society for Industrial and Applied Mathematics, 2000.
21. L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown. Satzilla: portfolio-based algorithm selection for sat. *Journal of artificial intelligence research*, 32:565–606, 2008.