Vertex Sampling for Backdoor Search in Binary Linear Optimization

Chang Liu and Elias B. Khalil

Department of Mechanical and Industrial Engineering,
University of Toronto, Toronto, Canada
changy.liu@mail.utoronto.ca

Abstract. Branching variable selection is a critical decision in branch and bound. A small subset of variables forms a "backdoor" if branch and bound can solve the problem to optimality (or prove infeasibility) by branching only on that subset. Existing methods for backdoor search rely either on identifying such variables through learning techniques or, for binary linear optimization, through collecting vertices encountered in the branch and bound tree and "hitting" them with a small set of variables. Both lack systematic and dynamic information collection. In this work, we propose a novel dynamic approach for sampling fractional vertices for backdoor search in binary linear optimization. Our method relies on a heuristic for uniform vertex sampling, a sweet spot between very slow exhaustive enumeration and fast but arbitrary, possibly biased, sampling. We test our approach on MIPLIB2017 binary instances and are able to consistently generate smaller branch and bound trees than Fischetti and Monaci's method [10]. Our method is sometimes competitive with the SCIP solver's sophisticated branching strategy. We also analyze the existence and size of backdoors on small synthetic instances where exhaustive enumeration is feasible. Results suggest that our heuristic vertex sampling strategy is a good approximation to exhaustive enumeration and that small backdoors do occur naturally in our dataset.

Keywords: Integer Programming · Branch and Bound · Branching · Backdoor Variables.

1 Introduction

The branch and bound framework is a central tool in solving hard combinatorial problems. For an integer linear program, at every node of the branch and bound search tree, the solver must pick a fractional variable and split its domain through the addition of constraints, resulting in two child nodes. Choosing the "right" variable to split on may result in significant reductions in tree size. In many problem instances, it has been observed that there exist small subsets of variables such that if the solver branches on only that subset, it can solve the original problem to optimality [6]. These small subsets of variables are referred to as "backdoor" variables, or simply backdoors. Recent research has shown that only branching on these backdoor variables for combinatorial optimization problems leads to faster total runtime and smaller search trees [16,10].

1.1 Strong Optimality Backdoor

There exist various definitions for backdoors in constraint satisfaction [18] and optimization problems [6]. In this work, we will focus on *strong optimality backdoors* as presented in [10,7].

Given a binary linear problem $(P) : \min_x c^\intercal x$ s.t. $Ax \leq b, x \in \{0,1\}^n$,

- Let $v(P)$ denote the optimal value of the problem (P) or inf if the problem is infeasible.
- Let $L(S) := \min\{c^\top x : Ax \leq b, x_j \text{ integer } \forall j \in S\}$ be the optimal value obtained by only requiring integrality for variables in set S. $L(S) := inf$ if the problem is infeasible by requiring integrality on variables in S.
- As a result, $L(\emptyset)$ corresponds to the root linear relaxation value resulting from dropping the integrality requirement on all variables in (P).

Definition 1. *A set S is a **strong optimality backdoor** iff $L(S) = v(P)$, that is, requiring integrality on the variables in S is enough to obtain a certifiably optimal solution or prove infeasibility.*

1.2 A Hitting-set Approach to Binary Linear Problems

In this paper, we will focus on solving *binary linear programs* (BLP). In addition to their pervasiveness in real-world integer programs, a BLP has the nice characteristic that branching on a variable does not introduce new vertices to the resulting polytopes [2].

Given this property of BLPs, there exists a simple exact method for finding *strong* backdoors for BLPs as illustrated by Fischetti and Monaci [10]. Let $\mathcal{X}$ represent the set of variables in a BLP. We introduce binary variables y_i such that $y_i = 1$ if variable $i \in \mathcal{X}$ is in a strong backdoor set and $y_i = 0$ otherwise. Now, let $\mathcal{V}$ be the set of fractional vertices of the BLP, and for any vertex $v^* \in \mathcal{V}$, let $frac(v^*) := \{i \in \mathcal{X} : v_i^* \text{ is fractional}\}$ denote the set of variables that are fractional in the vertex. Then, finding a minimum-size backdoor amounts to solving the hitting set problem shown in Figure 1.

$$\min \sum_{i \in \mathcal{X}} y_i \tag{1}$$

$$\sum_{i \in frac(v^*)} y_i \geq 1 \qquad \forall v^* \in \mathcal{V} \tag{2}$$

$$y_i \in \{0,1\} \qquad \forall i \in \mathcal{X} \tag{3}$$

Fig. 1: Hitting set model to find a minimum-size backdoor for a BLP.

The hitting set problem can also be solved approximately in polynomial time. The greedy approximation algorithm starts with an empty set of variables, adding in each iteration a variable that hits the largest number of remaining fractional vertices from $\mathcal{V}$. After adding a variable, vertices that are hit are removed from consideration in the next iteration. The algorithm terminates when all vertices have been hit. This greedy hitting set algorithm finds a set that is at most a factor of $O(\log|\mathcal{X}|)$ larger than the smallest hitting set (see Section 31.4 of [8]).

Up to now, we have assumed full access to the set of fractional vertices $\mathcal{V}$. In practice, enumerating this set is at least as hard as solving the optimization problem itself [15]. Fischetti and Monaci [10], whose method we will refer to as *FM* throughout this paper, addresses this issue using a two-stage approach. In the preliminary "sampling" phase, their method collects fractional vertices (solutions) during the branch and bound procedure. These vertices come from the solutions to node LP relaxations using any branching variable selection strategy. Then, a set covering model (after careful study, we believe this model is more accurately described as a hitting set model) is solved on all the collected vertices to find a small subset of variables (a backdoor set) that covers (hits) all collected vertices. Given enough time, the *FM* algorithm could enumerate all vertices and a minimum backdoor set is guaranteed to be found. However, in their work, they set a time budget and only collect a subset of vertices, thus only finding *quasi*-backdoor sets.

1.3 Motivation

To illustrate the shortcomings of *FM*'s vertex sampling strategy, we consider the `glass-sc` instance from MIPLIB2017 [11]. Running *FM* on this instance, we observe quasi-backdoor of 1-2 variables only. Fischetti and Monaci described in the paper that the runtime and number of nodes in the branch and bound trees using *FM* is better than Cplex with heuristics and presolve turned off, but worse than the default Cplex branch and bound by quite a margin. With more investigation, we notice that the vertices found through *FM* are very similar to each other in terms of variable fractionality, i.e., the collected vertices have the same small set of fractional variables. This sampling process is clearly ineffective, as all the sampled vertices can be hit with very few variables, but those do not constitute a backdoor. This could be due to the arbitrary nature of collecting solutions during the branch and bound procedure. This phenomenon motivates our investigation in this paper: we seek vertex sampling strategies that are better guided and that can lead to better backdoors.

1.4 Contributions

In this paper, we propose two backdoor branching strategies based on a new vertex sampling approach. Our sampling method is a heuristic extension of the shake-and-bake method [3] which is originally designed for sampling uniformly from the facets, rather than vertices, of a polytope. The *static* version of our method is analogous in its two-phase structure to *FM*. Rather than sampling arbitrarily through short branch and bound runs as is done in *FM*, we aim for more uniform sampling through our new heuristic. In the second phase, a hitting-set problem is solved to produce a quasi-backdoor. Lastly, the quasi-backdoor variables are prioritized in branching variable selection, with one of the solver's branching strategies being used as a fallback until a termination criterion is met.

The *dynamic* version of our method recursively applies the sample-then-hit approach at nodes of the search tree. Whenever all quasi-backdoor variables are integer in a node's relaxation solution, i.e., they are not candidates for branching, a sampling of vertices in the node's polytope, i.e., the polytope resulting from adding the branching constraints, is triggered, followed by a hitting-set solve. This algorithm incurs a higher computational cost per node on average compared to its static counterpart but relies less on the solver's fallback branching strategy.

Experiments comparing both the static and dynamic backdoor branching approaches with existing backdoor and non-backdoor branching strategies are performed. In Section 5, we show that backdoors found through our vertex sampling approach are good approximations to exact strong backdoors. In Section 6, we show that both our static and dynamic backdoor branching strategies outperform FM on MIPLIB2017 binary instances, and also outperform SCIP's default branching strategies for a subset of the tested instances.

2 Related Work

The existence of backdoor sets has been observed for many integer programs [6] and satisfiability problems [18,17]. While there is little theory supporting their existence in general or for specific structured problems, a variety of backdoor search techniques have been proposed in the literature. They vary in their simplicity and computational complexity.

Basic variable sampling methods. In [6], the authors present two simple approaches to test for small backdoors. In one, they randomly select subset of variables of a given size k and test whether it is a backdoor. This method is further modified to account for the fractionality of the variables in the LP root relaxation solution. A variable that is closer to 0.5 in the root relaxation solution has a higher chance of being picked to be part of a backdoor. These methods were meant to test the existence and likelihood of finding backdoors but are not effective at scale.

Monte Carlo Tree Search. Recent work incorporates learning into the search of backdoor sets for *mixed integer programming* (MIP). Khalil et al. proposed `BaMCTS`, a *Monte Carlo Tree Search* (MCTS) based approach to finding backdoor sets of a specific size K for a MIP instance [16]. In `BaMCTS`, the backdoor search is viewed as a single-player constraint satisfaction game where the goal is to find an order-sensitive subset of variables of a predetermined size that satisfies the definition of a backdoor set. A state in this game is then a set of any permutation of variables of size less than K. An action is simply choosing a variable, for which the current LP relaxation value is not integral, and that is currently not part of the state and appending it to the current state. When the size of the state is equal to K, then it is a terminal state and a MIP is run by only branching on variables in the current terminal state and respecting the ordering. If the MIP is solved, then the current state forms a backdoor set; if branching is not possible in any leaf node, then the current state is not a backdoor set. Information such as pseudocosts and search completion is collected for future runs. The authors then leverage the pseudocosts information in the node selection and expansion steps of the MCTS. The authors showed that by branching on the backdoor set found through `BaMCTS` led to fewer nodes in the search tree, shorter total runtime, and better optimality gap compared with standard MIP solver, though the incremental gain is not large.

Supervised Learning. Ferber et al. have presented a supervised learning framework to find a small set of pseudo-backdoors which are variables that the solver prioritizes whenever it is possible to branch on them [9]. The authors have trained two classifiers: one that predicts if one set of pseudo-backdoors is better or worse than another, and one that predicts whether a set of pseudo-backdoors is better or worse than the default solver settings. The authors looked at two sets of problems, the capacitated facility location problem, and the generalized independent set problem. The authors showed that the combined method of using both classifiers resulted in faster runtime but the improvement is also marginal. In a related line of work, Cai et al. first utilize MCTS to collect backdoor information to train a Graph Attention Network, then predict backdoors [4]. The authors tested their methods on four categories of problems and showed runtime improvements against both the previous model from Ferber et al. [9] and standard Gurobi. However, this method requires offline data collection and training for each problem family.

3 Full Vertex Enumeration

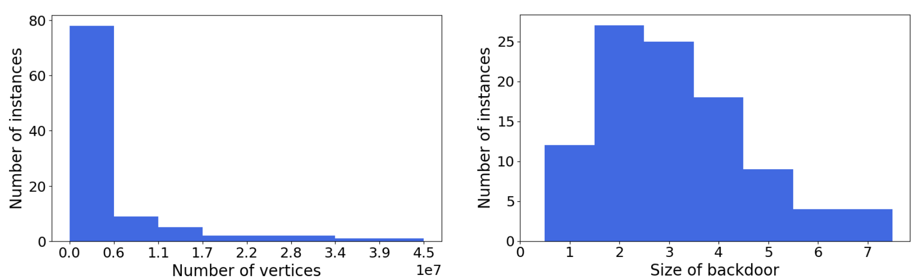
Given the full set of vertices of a BLP, a minimum-size backdoor can be found using the hitting-set approach described in Section 1.2. In order to verify the existence of small backdoors in small instances, we generate 100 integer-infeasible set-covering problem instances, $\{\min \mathbf{1}^T x | x \in \mathbb{R}^d : Ax \leq -\mathbf{1}, x \in \{0,1\}\}$ with 20 variables and 40 constraints, i.e., the matrix A is of size 20×40. The instances are generated as follows: the $n \times d$ matrix A is generated by assigning -1 randomly with a probability of 30% and zero elsewhere. Next, let z denote the optimal value of an instance. We make the instances infeasible by appending the additional optimality constraint of $\mathbf{1}^T x \leq z - \epsilon$, where ϵ is a small tolerance set to be $1e^{-5}$. We work with infeasible instances to isolate the impact of finding backdoors. In solving feasible instances, the change in the primal bound due to finding new incumbents affects the tree size in somewhat arbitrary ways that also depend on the node selection strategy. Whereas in solving infeasible instances, we are guaranteed to only have fractional vertices, thus only infeasible sub-trees are pruned and primal bounds have no effect.

Enumerating all vertices in a convex polytope, when the dimension is greater than two, is a hard problem [15] and can be extremely computationally expensive. Theoretical upper bounds and lower bounds on the number of vertices for a polytope exist but still generally present a large gap between them [12]. Given a polytope P with n constraints and d variables, the bounds are given by:

1. **McMullen upper bound theorem**: the number of vertices in polytope $\mathcal{P}$ is at most $f(n,d) = \binom{n-\lfloor (d+1)/2 \rfloor}{n-d} + \binom{n-\lfloor (d+2)/2 \rfloor}{n-d}$
2. **Barnette lower bound theorem**: the number of vertices in polytope $\mathcal{P}$ is at least $g(n,d) = n(d-1) - (d+1)(d-2)$

As an example, a hypercube in dimension $d = 40$ has exactly $2^{40} \approx 1.09 \times 10^{12}$ vertices, but the lower and upper bounds are 1562 and 5.59×10^{15}, respectively. As a result, we were not able to generate all vertices for instances greater than 20 variables and 40 constraints in the provided time limit.

We used the *lrs* method proposed by Avis and Jordan [1] to enumerate all the vertices for each generated instance. On average, each instance had 4.8M vertices, ranging from 728 to 44.1M (see Figure 2a). After running the exact hitting set model on all the vertices for each instance, the smallest backdoor is of size 1, the largest backdoor has a size of 8, and the median size is 3. This indeed shows that for many of these instances, a small backdoor does exist; the distribution can be seen in Figure 2b.



(a) Vertex count (b) Backdoor size

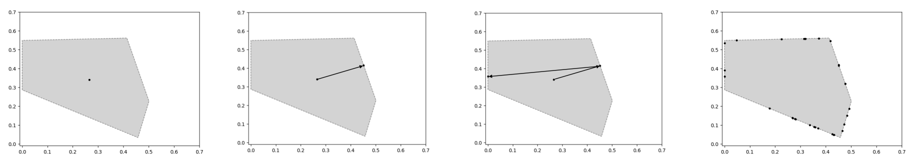
Fig. 2: Vertex count and backdoor size distributions for set cover instances with 20 variables and 40 constraints.

4 Finding Backdoors Through Vertex Sampling

Since vertex enumeration is intractable, and in theory, uniformly sampling from the vertices of a given polytope is known to be hard (see Theorem 1 from [14]), the best one can hope for is to sample said vertices. As such, we will focus on designing a heuristic sampling method.

4.1 Modified Shake-and-bake

Boender et al. introduced the "shake-and-bake" method which generates (asymptotically) uniform points on the *boundary* of a fully-dimensional polytope P [3]. Given a polytope P, the shake-and-bake algorithm, illustrated in Figure 3, starts by finding the Chebyshev centre of P (3a), then hits the boundary of P (3b) in a random direction and outputs point x^0. Next, it generates a direction in the half-space bounded by the binding constraint of the current point x^0 and yields the next point x^1. This process is repeated until a certain number of samples is generated.



(a) Find Chebyshev centre of P (b) Find point x^0 on boundary of P (c) Generate direction from x^0's binding constraint and hits point x^1 (d) 30 samples on boundary of P

Fig. 3: Illustration for the Shake-and-Bake algorithm.

We propose to modify the shake-and-bake method to collect vertices with diverse fractionality and to compute backdoor variables independently of the branch and bound tree. We take every sampled point on a facet from the shake-and-bake algorithm, generate a random objective direction and solve a linear program on the facet of the sampled point to reach to a vertex that is on said facet. This way, we can navigate to diverse vertices and collect them. However, we lose the guarantee of uniform sampling, a question that still deserves attention in its own right. The *"modified shake-and-bake"* (MSB) framework is described in Algorithm 1.

Algorithm 1: Modified shake-and-bake (*MSB*)

Input: $n \times d$ matrix A, n-dimensional vector $\mathbf{b}$, number of samples N.
Sample a set of N points $\mathcal{S}$ on the boundary of the polytope defined by $A(x) \leq \mathbf{b}$ using the "shake-and-bake" method;
for $x^0 \in \mathcal{S}$ **do**
  $A', \mathbf{b}' \leftarrow$ subset of constraints that are binding at x^0
  $\mathbf{c}^0 \leftarrow$ random direction vector defining the objective function
  $x^* \leftarrow$ solution to the LP $\{\max\ \mathbf{c}^T x | x \in \mathbf{R}^d : Ax \leq b, A'x = \mathbf{b}', x \geq 0\}$
end
Return all unique x^*

Solving the hitting set problem on the vertices returned by MSB produces a quasi-backdoor. Running a branch and bound solver that prioritizes the variables in the quasi-backdoor set is referred to as MSB_r, which we show in Section 6 does result in some performance improvement (smaller trees). However, this *static* approach is limited by the number of samples taken at the start and the initial produced quasi-backdoor set is often far from being a strong backdoor. As the branch and bound tree is built, we obtain more information that is relevant to branching decisions that could be exploited. The *static* approach we have just described cannot exploit such additional information.

4.2 Dynamic Backdoor Branching

To that end, we introduce a dynamic sampling approach (MSB_d), in which we start by sampling a number of vertices using *MSB* and compute an initial quasi-backdoor. In the branch and bound tree, we let the algorithm pick variables in this quasi-backdoor first for branching. If this is not a full backdoor set, we will arrive to nodes where none of the branching variable candidates are in the initial quasi-backdoor. At that node, we run *MSB* again, but adding the branching constraints of the node so as to guarantee that new vertices are sampled. We will also include the parent linear programming relaxation solution in the vertex set to ensure that the new quasi-backdoor will at least contain one of the branching variable candidates at this node. At every node, we will also keep copies of the ancestors' backdoors that are passed to the children. There can be situations where backdoor variables found in a previous node have not been used in branching decisions but could be used deeper in the tree. Ideally, we would like to sample as few times as possible as sampling is very expensive.

There are several hyperparameters that control MSB_d, shown in Table 1. We must pick an adequate number of initial samples; as we branch, we want to sample less and less (limited by a minimum number of samples) and this can be controlled through a decay factor, which is dependent on the depth of the tree. We also set a maximum depth of the tree at which we won't perform more sampling. When no branching candidates are in the generated backdoor set, we will refer back to a fallback branching strategy (either most infeasible branching or the solver's default).

Table 1: Hyperparameters used in MSB_d

Hyperparameter	Value
Initial number of samples	500
Initial number of samples when sampled within the branch and bound tree	100
Sampling decay factor	0.8
Sampling max depth	8
Minimum number of samples	20

4.3 Complexity of *MSB*

Given a matrix A of dimension $n \times d$, the runtime to get one sample from the shake-and-bake method is $O(nd)$. In *MSB*, for each sample, we need to 1) find the binding constraints, 2) draw a random objective vector, and 3) solve the LP. The complexity for 1) is approximately $O(nd)$, for 2) is $O(d)$ and for 3) is on average linear or low-degree polynomial in (n, d), say $f(n, d)$. Thus the overall complexity for *MSB* is $O(nd + f(n, d))$. Assuming that the LP solving is efficient in practice, the overall runtime is expected to grow polynomially with n and d.

5 Experiments on Synthetic Instances

We test experimentally how the quasi-backdoors found through *MSB* perform when compared to exact backdoors found by solving the hitting-set problem on the full set of vertices of the 100 synthetic instances described in Section 3.

5.1 Sampled Vertex Hitting Rate

First, to understand how *MSB* performs in approximating uniform random sampling, we varied the number of samples and computed the hitting rate on the fractional vertices of each of the 100 synthetic instances. For each instance, we used five different seeds to sample $\{50, 100, 500, 1000, 5000\}$ points using *MSB*. We can see from Figure 4 that even with just 50 samples, we achieved a median hitting rate of 99%; that rate quickly goes to 100% for many instances with just 500 samples. When compared to uniformly sampling from the existing full set of vertices, our approach seemed to achieve an even higher hitting rate. This indicates that it is a good approximation to uniform vertex sampling.

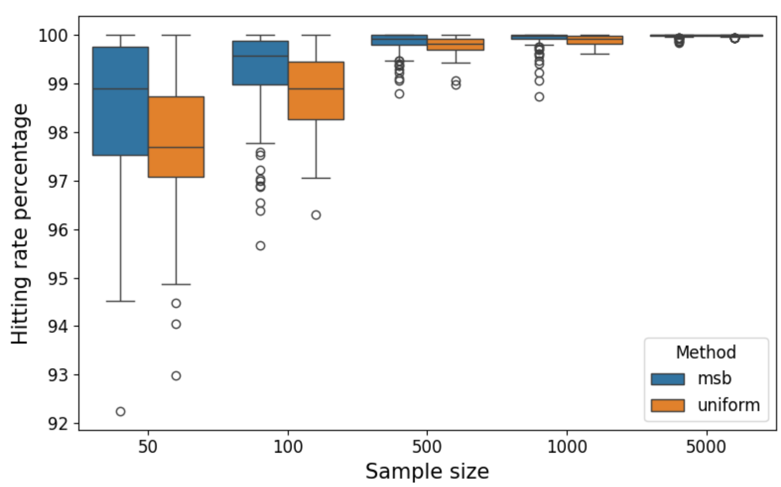


Fig. 4: Vertex hitting rate vs. sample size using *MSB*

5.2 Solving Synthetic Instances using Vertex Sampling Approaches

We then solved each of the 100 generated instances with the exact backdoor branching (*EBB*), quasi-backdoor branching at the root (MSB_r), and the dynamic quasi-backdoor branching (MSB_d) with the hyper-parameters listed in Table 1. For each instance, we compared the nodecounts, i.e., the sizes of the branch and bound trees. From Table 2, we can see that MSB_r and MSB_d manage to tie with *EBB* in 47 and 41 instances. MSB_r and MSB_d could even produce trees smaller than *EBB* in 6 and 3 instances, respectively. This experiment shows that for these small synthetic instances, our vertex sampling approaches do generate good approximate backdoors.

Table 2: Nodecount comparison on synthetic instances. *EBB* is compared with MSB_r and MSB_d, both with *most infeasible* as the fall back branching strategy; w denotes the number of wins, t the number of ties, and l the number of losses by *EBB*. Here, a win is when a method produces a smaller nodecount.

	$\mathbf{MSB_r}$	$\mathbf{MSB_d}$
EBB	w: 47 t: 47 l: 6	w: 56 t: 41 l: 3

6 Experimental Results on MIPLIB2017 Instances

Next, we tested our methods on BLP instances from MIPLIB2017 [11]. Due to the imposed time limit, we only report the results of the smallest 48 instances in this paper, which have less than 3000 variables (see Appendix A for instance details). Although we have yet to test on larger instances, we do not believe the runtime overhead due to sampling will be the bottleneck for solving larger instances; see Section 4.3.

For each of the 48 instances, we ran 7 different methods: two out-of-the-box SCIP branching strategies, the default, *S(D)*, and most infeasible branching, *S(MI)*; the set covering method from Fischetti and Monaci (*FM*); and four variations of our quasi-backdoor branching approaches, as outlined in Table 3. Note that *S(D)* is explicitly used as a reference only and is not considered for result comparison. In this work, as we do not incorporate any learning and do not require any initial search or data collection, we did not include a comparison with other learning-based backdoor methods [16,9,4].

6.1 Experimental Setup

All instances are solved with SCIP version 9.2.2 with Gurobi 12 as the LP solver with vanilla branch and bound, i.e., with presolve, heuristics, cuts, symmetry,

Table 3: Branching strategies

Acronym	**Strategy**
S(D)	SCIP's default branching
S(MI)	SCIP's most infeasible branching
FM	Set covering from Fischetti and Monaci
$\mathrm{MSB_r}$(MI)	Static quasi-backdoor branching, fall back: most infeasible branching
$\mathrm{MSB_d}$(MI)	Dynamic quasi-backdoor branching, fall back: most infeasible branching
$\mathrm{MSB_r}$(D)	Static quasi-backdoor branching, fall back: default branching
$\mathrm{MSB_d}$(D)	Dynamic quasi-backdoor branching, fall back: default branching

etc, switched off. Many of these much larger MIPLIB2017 instances could not be solved within the time limit of two hours. In order to gather more information, we made a few changes to our experimental setup.

1. The hitting set problem was changed was solved approximately using the approach outlined in Section 1.2.
2. For feasible instances, we used the best objective bound found from the MIPLIB2017 library [11] and added an objective bound constraint.

We also ran each instance with 5 different seeds as we noticed that there exists variability due to random seeds in both the solver and the sampling stages. The code to reproduce our results, including the implementation of all reported methods is available at https://github.com/changyliu/backdoorsearch-vertex-sampling.

6.2 Results

Out of the 48 instances, only 29 instances were solved by at least one of the seven methods across all 5 seeds. Further, 10 instances were solved at the root, likely due to the objective bound that we have added. Table 4 shows the mean nodecount across the 5 different seeds for the remaining 19 instances. The mean calculation only takes into account runs with seeds that solved within the time limit. We can see that except for one instance, `f2gap401600`, 18 instances resulted in the smallest nodecount when using *MSB_r(D)* or *MSB_d(D)*. In 10 instances, our methods even resulted with less or as many nbodes as SCIP's default branching. It is especially interesting to note that for instances `opm2-z6-s1` and `opm2-z7-s8`, which are precedence-constrained knapsack problems, we produced trees an order of magnitude smaller than SCIP's default branching strategy.

In Table 5, we summarize the average total runtime (including the time for sampling) in seconds for the 19 MIPLIB2017 instances. Our methods do take a longer time to complete in general, however they are still on the same order of magnitude and with a closer look, we do see that the majority of the runtime for our quasi-backdoor branching approaches is spent in vertex sampling. However, it is still worth noting that for five instances (`iis-glass-cov`, `iis-hc-cov`, `f2gap401600`, `opm2-z6-s1`, `opm2-z7-s8`), even including the vertex sampling time, the average total runtimes to solve these problems were shorter.

Table 4: Mean nodecount for MIPLIB instances across five seeds.

Instance	S (D)	S (MI)	MSB_r (MI)	MSB_d (MI)	MSB_r (D)	MSB_d (D)	FM
stein9inf	13	**13**	**13**	**13**	**13**	**13**	**13**
stein15inf	82	86	105	103	**53**	103	97
stein45inf	502	1071	1032	1317	**440**	531	993
p0201	120	675	1235	953	444	**272**	703
glass-sc	249736	t.l.	t.l.	t.l.	**261353**	t.l.	t.l.
iis-glass-cov	43613	56720	54323	295638	**40916**	174856	62862
iis-hc-cov	64398	125765	125765	125765	**64398**	**64398**	140769
supportcase14	5	154	40	37	**21**	33	53
supportcase16	1	90	39	29	**21**	29	39
mod008inf	151	1365	643	627	179	**83**	961
f2gap40400	4	696	708	691	**9**	32	687
mine-166-5	1942	t.l.	t.l.	t.l.	239434	**74421**	t.l.
opm2-z6-s1	15741	t.l.	36245	50320	1140	**810**	t.l.
f2gap401600	4080739	**4247500**	t.l.	4285822	t.l.	4723333	t.l.
acc-tight2	87	98	195	361	**71**	580	99
opm2-z7-s8	36387	t.l.	52226	t.l.	3779	**3343**	t.l.
neos-859770	47173	127305	99565	103057.4	51308	**40476**	134863
neos-1330346	366712	t.l.	t.l.	t.l.	t.l.	t.l.	t.l.
toll-like	49	t.l.	t.l.	t.l.	t.l.	t.l.	t.l.

*t.l. = reached time limit of 2 hours.

Table 5: Mean runtime in seconds for MIPLIB instances across five seeds

Instance	S (D)	S (MI)	MSB_r (MI)	MSB_d (MI)	MSB_r (D)	MSB_d (D)	FM
stein9inf	0.51	0.61	1.31	0.76	1.34	1.49	1.57
stein15inf	0.84	0.71	1.29	1.25	1.62	1.10	1.52
stein45inf	1.30	0.90	2.49	7.82	2.97	8.82	1.59
p0201	1.06	0.88	4.47	6.76	4.00	15.83	2.78
glass-sc	3704.05	t.l.	t.l.	t.l.	3967.03	t.l.	t.l.
iis-glass-cov	782.36	769.70	773.66	4665.94	611.92	3010.58	974.62
iis-hc-cov	1964.19	3401.02	3678.02	4003.09	1748.56	1673.75	4050.75
supportcase14	1.37	0.78	3.86	5.20	3.96	5.59	3.81
supportcase16	0.80	1.02	3.78	5.14	4.62	5.09	3.13
mod008inf	1.30	1.05	3.68	4.11	3.66	3.83	3.68
f2gap40400	1.75	1.38	4.83	8.86	5.57	8.87	5.41
mine-166-5	19.94	t.l.	t.l.	t.l.	2273.82	269.48	t.l.
opm2-z6-s1	242.97	t.l.	677.05	1335.62	189.67	175.25	t.l.
f2gap401600	2002.58	2279.28	t.l.	3254.60	t.l.	1280.96	t.l.
acc-tight2	14.92	12.60	127.78	247.86	143.24	207.93	123.39
opm2-z7-s8	993.64	t.l.	2462.81	t.l.	546.10	726.84	t.l.
neos-859770	332.83	905.85	1229.60	2195.76	725.22	1863.86	2106.52
neos-1330346	4999.68	t.l.	t.l.	t.l.	t.l.	t.l.	t.l.
toll-like	31.02	t.l.	t.l.	t.l.	t.l.	t.l.	t.l.

*t.l. = reached time limit of 2 hours.

Comparing MSB_r with FM. Looking closer at $MSB_r(D)$ and *FM*, the two methods are exactly the same except for the sampling method for backdoors at the root. The backdoors generated by the two methods are completely different. The mean backdoor size for *MSB* is 14.95 across the 48 instances with a minimum size of one and a maximum size of 69. On the other hand, the backdoor sizes for *FM* are a lot smaller, with a mean of 1.67, a minimum of one, and a maximum of five. This shows that our method covers a more diverse vertex set, thus producing more impactful and meaningful backdoors. Furthermore, in all of the 19 instances that solved to optimality, MSB_r had a smaller or equal nodecount when compared to *FM*. The runtime for $MSB_r(D)$ is also comparable to or smaller than *FM*, as seen in Figure 5. For many instances, the sampling time for our method is much smaller especially for larger instances even though we are collecting 500 vertices whereas the *FM* method only collects at most 100 vertices (10 iterations and at most 10 vertices per iteration).

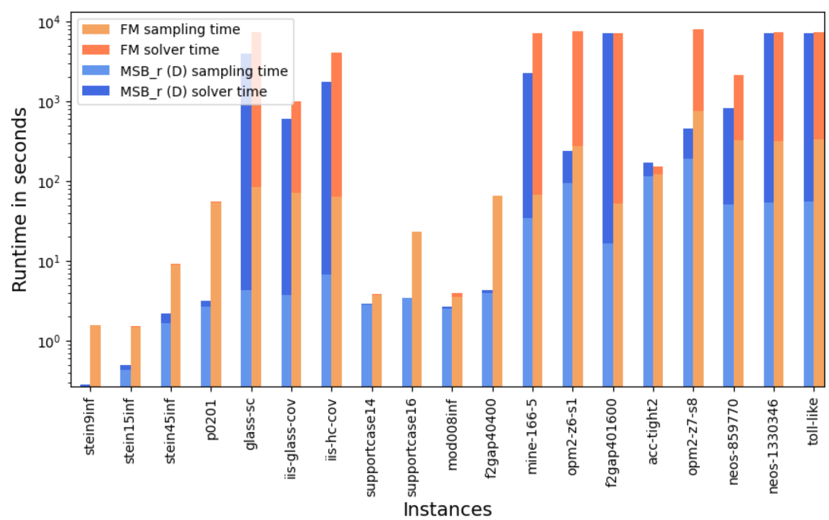


Fig. 5: Runtime breakdown comparison between $MSB_r(D)$ and *FM*

7 Conclusions and Future Work

In this work, we proposed two new methods for backdoor search in BLP, which rely on a form of partial vertex enumeration. While exact enumeration is computationally prohibitive, we show that a heuristic vertex sampling suffices for the purpose of constructing quality quasi-backdoor sets. The dynamic version of our approach continues to update the quasi-backdoor throughout the tree, up to some depth, exploiting additional vertex samples that are collected further down in the search tree.

The first key finding of our experimental evaluation is that vertex sampling using the proposed modified shake-and-bake method is a good proxy to exact enumeration, even when the number of vertices is extremely large. The second finding is that while our sampling procedure has no theoretical guarantees, it performs similarly to true uniform sampling when it comes to backdoor finding. Last, experiments on binary MIPLIB2017 instances show that our proposed methods are competitive with SCIP's default strategy and much better than the backdoor search approach of Fischetti and Monaci. We do note that our method is still limited by the runtime overhead due to sampling, although not substantial, but we do plan to do further work on efficient sampling to be more practically useful and competitive with other approaches.

Going forward, a systematic tuning of the hyperparameters of both variants of our approach could be explored to improve performance. Instance-specific tuning could be performed based on problem type and instance size similar to [13,5]. Our approach performed remarkably well on some structured MIPLIB2017 instances, namely `opm2-z6-s1` and `opm2-z7-s8`, which are precedence-constrained knapsack problems; we are interested in whether this combinatorial structure is particularly amenable to backdoors. Last, our modification of the shake-and-bake method to sample vertices instead of facets proved to be empirically effective. It would be interesting to analyze its theoretical properties in order to further research into sampling-based backdoor search methods.

References

1. Avis, D., Jordan, C.: mplrs: A scalable parallel vertex/facet enumeration code. Mathematical Programming Computation **10**(2), 267–302 (2018)
2. Balas, E., Ceria, S., Cornuéjols, G., Natraj, N.: Gomory cuts revisited. Operations Research Letters **19**(1), 1–9 (1996)
3. Boender, C., Caron, R.J., McDonald, J.F., Kan, A.R., Romeijn, H.E., Smith, R.L., Telgen, J., Vorst, A.: Shake-and-bake algorithms for generating uniform points on the boundary of bounded polyhedra. Operations research **39**(6), 945–954 (1991)
4. Cai, J., Huang, T., Dilkina, B.: Learning backdoors for mixed integer linear programs with contrastive learning. arXiv preprint arXiv:2401.10467 (2024)
5. Di Liberto, G., Kadioglu, S., Leo, K., Malitsky, Y.: Dash: Dynamic approach for switching heuristics. European Journal of Operational Research **248**(3), 943–953 (2016)
6. Dilkina, B., Gomes, C.P., Malitsky, Y., Sabharwal, A., Sellmann, M.: Backdoors to combinatorial optimization: Feasibility and optimality. In: Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems: 6th International Conference, CPAIOR 2009 Pittsburgh, PA, USA, May 27-31, 2009 Proceedings 6. pp. 56–70. Springer (2009)
7. Dilkina, B., Gomes, C.P., Sabharwal, A.: Backdoors in the context of learning. In: International Conference on Theory and Applications of Satisfiability Testing. pp. 73–79. Springer (2009)
8. Erickson, J.: Algorithms. Self-published / Independent (2019)
9. Ferber, A., Song, J., Dilkina, B., Yue, Y.: Learning pseudo-backdoors for mixed integer programs. In: International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research. pp. 91–102 (2022)

10. Fischetti, M., Monaci, M.: Backdoor branching. INFORMS Journal on Computing **25**(4), 693–700 (2013)
11. Gleixner, A., Hendel, G., Gamrath, G., Achterberg, T., Bastubbe, M., Berthold, T., Christophel, P., Jarck, K., Koch, T., Linderoth, J., et al.: Miplib 2017: data-driven compilation of the 6th mixed-integer programming library. Mathematical Programming Computation **13**(3), 443–490 (2021)
12. Gruber, P.M., Wills, J.M.: Handbook of convex geometry, Volume B. North-Holland (1993)
13. Kadioglu, S., Malitsky, Y., Sellmann, M., Tierney, K., et al.: Isac-instance-specific algorithm configuration. In: ECAI. vol. 215, pp. 751–756 (2010)
14. Khachiyan, L.: Transversal hypergraphs and families of polyhedral cones. Advances in Convex Analysis and Global Optimization: Honoring the Memory of C. Caratheodory (1873–1950) pp. 105–118 (2001)
15. Khachiyan, L., Boros, E., Borys, K., Gurvich, V., Elbassioni, K.: Generating all vertices of a polyhedron is hard. Twentieth Anniversary Volume: Discrete & Computational Geometry pp. 1–17 (2009)
16. Khalil, E.B., Vaezipoor, P., Dilkina, B.: Finding backdoors to integer programs: a monte carlo tree search framework. In: Proceedings of the AAAI Conference on Artificial Intelligence. vol. 36, pp. 3786–3795 (2022)
17. Kilby, P., Slaney, J., Thiébaux, S., Walsh, T., et al.: Backbones and backdoors in satisfiability. In: Proceedings of the AAAI Conference on Artificial Intelligence. vol. 5, pp. 1368–1373 (2005)
18. Williams, R., Gomes, C.P., Selman, B.: Backdoors to typical case complexity. In: Proceedings of the International Joint Conference on Artificial Intelligence. vol. 3, pp. 1173–1178 (2003)

A Binary MIPLIB2017 Instances

Below is a list of the binary MIPLIB2017 instances that we have tested our methods on. The instances are sorted in ascending order of number of variables.

Table 6: Binary MIPLIB instances

Instance	**Status**	**Variables**	**Constraints**	**Objective**
stein9inf	easy	9	14	Infeasible
stein15inf	easy	15	37	Infeasible
stein45inf	easy	45	332	Infeasible
pb-market-split8-70-4	hard	71	17	0
v150d30-2hopcds	hard	150	7822	41
2club200v15p5scn	hard	200	17013	-70
p0201	easy	201	133	7615
glass-sc	easy	214	6119	23
iis-glass-cov	easy	214	5375	21
iis-hc-cov	easy	297	9727	17
supportcase14	easy	304	234	288
mod008inf	easy	319	7	Infeasible
supportcase16	easy	319	130	288
f2gap40400	easy	400	40	20772
mine-166-5	easy	830	8429	-566395707.9
mine-90-10	easy	900	6270	-784302337.6
queens-30	hard	900	960	-40
ponderthis0517-inf	easy	975	78	Infeasible
cod105	easy	1024	1024	-12
sorrell3	easy	1024	169162	-16
supportcase30	open	1024	1028	NA
reblock115	easy	1150	4735	-36800603.23
app2-2	easy	1226	335	212040.3571
acc-tight5	easy	1339	3052	0
opm2-z6-s1	easy	1350	15533	-6202
seymour	easy	1372	4944	423
f2gap201600	easy	1600	20	76453
f2gap401600	easy	1600	40	82307
f2gap801600	easy	1600	80	86678.99998
acc-tight2	easy	1620	2520	0
acc-tight4	easy	1620	3285	0
reblock166	easy	1660	17024	-600052168.91
cvs08r139-94	easy	1864	2398	-116
opm2-z7-s8	easy	2023	31798	-11242
sorrell8	easy	2046	18944	-350
sorrell7	open	2048	78848	-198.0*
cvs16r70-62	easy	2112	3278	-42
graph20-20-1rand	easy	2183	5587	-9
ramos3	open	2187	2187	186.0*
neos-2328163-agri	easy	2236	1963	27674
chromaticindex32-8	easy	2304	2111	4
cvs16r89-60	easy	2384	3068	-65
neos-859770	easy	2504	2065	Infeasible
mod010	easy	2655	146	6548
neos-1330346	easy	2664	4248	8
neos-1337307	easy	2840	5687	-202319
cvs16r106-72	easy	2848	3608	-81
toll-like	easy	2883	4408	610