Clique Probing for Mixed-Integer Programs

Jacob von Holly-Ponientzietz[1], Alexander Hoen[1,2], Mark Turner[1], and
Ambros Gleixner[1,2]

[1] Zuse Institute Berlin, Takustr. 7, 14195 Berlin, Germany
`von.holly-ponientzietz@zib.de`,`turner@zib.de`
[2] HTW Berlin, 10313 Berlin, Germany
`hoen@htw-berlin.de`,`gleixner@htw-berlin.de`

Abstract. Probing is an important presolving technique in mixed-integer programming solvers. It selects binary variables, tentatively fixes them to 0 and 1, and performs propagation to deduce additional variable fixings, bound tightenings, substitutions, and implications. In this work, we propose *clique probing*: instead of probing on individual variables, we select cliques, a set of binary variables of which at most one can be set to one, and systematically probe on all variables of a clique. Experiments with our implementation in the open-source presolve library PaPILO demonstrate that exploiting clique information in this form significantly increases the number of reductions. When integrated into the MIP solver SCIP, we observe a 3% performance improvement on MIPLIB instances containing cliques.

Keywords: mixed integer programming, presolving, probing

1 Introduction

Mixed Integer Programming (MIP) is widely used in industry to solve optimization problems, with competitive MIP solvers typically implementing some variant of *branch-&-cut* as the backbone of the solving process. Before initializing the branch-&-cut process, however, *presolving* (or *preprocessing*) is called. The goal of presolving is to eliminate redundant information and strengthen the problem formulation, thereby accelerating the subsequent solution process. Therefore, presolving is a crucial part of modern MIP solvers and, more often than not, it is the deciding factor between a problem being solvable or not [2,3].

One important presolving technique is *probing* [11,1]. The central idea of probing is to select binary variables and temporarily fix them to 0 and 1. After each tentative fixing, a propagation step is performed, and the resulting deductions are analyzed in order to strengthen the formulation. In [2], disabling probing led to a slowdown of 7% to 33% on instances with more than 1000 seconds solving time and rendered 11 of 3155 models unsolvable.

Definition 1. Formally, let x be a binary variable and z an arbitrary variable with bounds $[l_z, u_z]$. We denote the variable bounds of z after propagating $x = 0$ as $[l_z^0, u_z^0]$ and the variable bounds of z after propagating $x = 1$ as $[l_z^1, u_z^1]$. The following four rules can be applied while probing:

inf If setting $x = 0$ ($x = 1$) makes the problem infeasible, x can be fixed to 1 (0).

bds Global bounds for z can be updated to $[\min(l_z^0, l_z^1), \max(u_z^0, u_z^1)]$.

sub If $l_z^0 = u_z^0$ and $l_z^1 = u_z^1$, the variable z can be substituted by $z = l_z^0 + (l_z^1 - l_z^0)\, x$.

imp Implications can be collected in a central data structure of the solver. An example for an implication would be $x_k = 0 \rightarrow z \geq l_z^0$.

Performing probing on all variables can be time-consuming and may outweigh its potential benefits to the solver. To address this, probing orders the variables according to a scoring function and employs a termination criterion to stop early. Thus, designing an effective scoring function and appropriate termination criteria is crucial for an efficient implementation. Despite the importance of both probing and the subsequent solving process, we are not aware of any published work describing scoring methods or termination criteria.

Open-source solvers like SCIP [6] and HiGHS [9] assign scores to each binary variable and rank them by this score. The score tries to reflect the impact that fixing the variables has on the model. These solvers abort if a certain absolute or relative limit is reached, typically a limit on time, number of variables, number of fixings, or a combination of these limits. In order to prevent stalling, an additional termination criterion is typically added, which interrupts probing if the last iterations were barely successful. MIP solvers typically exploit problem structure to accelerate the solving process. Examples include the use of clique tables [10,1], specialized handling of SOS constraints, and the identification of knapsack structure for generating strong cover cuts [5,4,1]. However, to the best of our knowledge, this has not yet been used for probing. To improve the efficiency of probing, we propose leveraging structural information from the model. In particular, we focus on cliques to speed up the procedure, although the underlying idea can be generalized to other types of structure. Therefore, we propose *Clique Probing*: During our probing routine, we use the information provided by cliques to adjust the fixings done during probing. By this, we get stronger reductions and are able to speed up the subsequent solving process.

The paper is organized as follows: In Section 2, we present the idea of clique probing and its implementation. In Section 3, we evaluate the performance of clique probing, and summarize our results in Section 4.

2 Clique Probing

2.1 General idea

A *clique* is a set of n binary variables $x_1, \ldots, x_n$ of which at most one variable is allowed to be active, i.e., to be set to one. We distinguish *at-most-one cliques*, for which

$$x_1 + \cdots + x_n \leq 1$$

must hold, and *exactly-one cliques*, for which

$$x_1 + \cdots + x_n = 1$$

must hold. During presolving, cliques are found explicitly as part of the constraints of the original model or derived from other problem structures that imply mutually exclusive assignments.

Standard probing selects variables based on the score, fixes them to one and zero, and performs propagation on both assignments. Assuming all variables of a clique are selected for probing, probing performs $2n$ propagation calls, two for each variable. However, a clique only allows at most $n + 1$ different assignments for the variable vector $(x_1, \ldots, x_n)$, and for exactly-one cliques, even one less, because the all-zero assignment is excluded. Therefore, it would be sufficient to just probe these $n + 1$, respectively n, cases. This has the advantage that we can (a) reduce the number of probing and subsequent propagation calls needed to get the same results, (b) propagate faster since we can fix multiple variables initially, (c) obtain tighter bounds for cliques involving more than two variables, since fixing a variable inside the clique to zero does not determine the remaining variables within the clique, and (d) exploit that the all-zero assignment is not feasible for exactly-one cliques.

The following definition adjusts the evaluation of the results of standard probing in Definition 1 for probing on cliques.

Definition 2. Formally, let c be an at-most-one clique consisting of n variables. Let c^i be the variable assignment, where x_i is set to one, and all other variables of clique c are set to zero. Let c^0 be the variable assignment, where all variables of a clique c are set to 0. Let z be an arbitrary variable with bounds $[l_z, u_z]$, and let $[l_z^i, u_z^i]$ be the variable bounds of z after propagating c^i. Then a new set of four rules can be applied while probing:

inf If propagating c^i with $i \in \{1, \ldots, n\}$ makes the problem infeasible, x_i can be fixed to zero. If propagating c^0 makes the problem infeasible, c can be upgraded to an exactly-one clique.

bds Global bounds for z can be updated to $[\min_{k\in\{0,\ldots,n\}} \ell_z^k, \max_{k\in\{0,\ldots,n\}} u_z^k]$.

sub Variable z can be substituted by $z = \ell_z^j + (\ell_z^i - \ell_z^j)\, x_i$ with $j \in \{1, \ldots, n\}\backslash\{i\}$ if the following rules hold:

1. $\ell_z^k = u_z^k$ for all $k \in \{0, 1, \ldots, n\}$
2. $\ell_z^j = \ell_z^k$ and $u_z^j = u_z^k$ for all $j, k \in \{0, 1 \ldots, n\}\backslash\{i\}$
3. $\ell_z^k \neq \ell_z^i$ and $u_z^k \neq u_z^i$ for all $k \in \{0, 1 \ldots, n\}\backslash\{i\}$

imp Implications for c^i with $i \in \{1, \ldots, n\}$ can still be stored similarly to Definition 1.

For evaluating exactly-one cliques, the variable assignment c^0 is not considered, and the rules are adapted accordingly: in the *bds* and *subs* rules, the index sets range over $\{1, \ldots, n\}$ instead of $\{0, \ldots, n\}$. Further, the sub-rule can be more generalized, allowing the aggregation of multiple variables.

2.2 Data structure

In formulating the rules for analyzing substitutions and bounds, it is necessary to know all bounds of each variable after the propagation of variable assignments. For long cliques, however, this can be memory-intensive and cause a significant slowdown.

To address this, we maintain a global data structure within clique probing instead of storing all bounds for each propagation. For the global bounds, we record for each variable the current maximum upper and minimum lower bound, denoted as B in Algorithm 1 below. To detect substitutions, we additionally store the second lowest and highest values, as well as the indices (argmin/argmax) of the minimum and maximum bounds, denoted as I in Algorithm 1.

Rather than duplicating bounds for every possible variable assignment, this approach limits the overhead to six values per variable. Additionally, after propagating each assignment, these values allow us to calculate the maximal bound changes and substitutions as if no further assignments were to be considered. For example, if a variable's bound is not tightened after propagating a single assignment, then no global tightening of that bound can occur. These values can be used as an upper bound on the number of potential bound changes and substitutions, which can then be used as a termination criterion for large cliques.

2.3 Algorithm

Algorithm 1 shows the function to perform probing on a single clique. First, the data structures B (for tracking global bound changes) and I (for detecting substitutions) are initialized (Line 1). For at-most-one cliques, all variables in the clique are fixed to zero and propagated. If this all-zero propagation is infeasible, the clique is upgraded to an exactly-one clique (Line 5). Otherwise, B and I are updated according to rules *(bds, subs)* (Line 7). Next, each variable in the clique is probed by setting it to one while fixing all others to zero, followed by propagation. If infeasibility is detected, the corresponding variable is fixed to zero (Line 13); otherwise, the data structures B and I are updated (Line 15).

Since this process can become computationally expensive for large cliques, we introduce an early termination criterion. As explained in Section 2.2, we can derive an upper bound on how many bound changes can be applied from B and I. If this falls below a certain threshold, we abort clique probing (Line 10). Finally, after probing all cases, the results stored in B and I are analyzed, and the corresponding bound changes and substitutions are applied (Lines 16 and 17). If each propagation within the clique proves infeasible, the algorithm concludes global infeasibility (Line 19).

A full run of clique probing is outlined in Algorithm 2. First, we need to select the cliques to be probed on. In Line 1, we sort cliques by their average variable score, and the highest-scoring cliques Then, we perform clique probing on the selected cliques until either all the cliques have been probed or two consecutive cliques have been unsuccessful (Line 7). Probing on a clique is considered unsuccessful if a certain threshold of bound changes and substitutions per propagated

Algorithm 1 PROBESINGLECLIQUE

```
Require: MIP with global variable bound vectors $l, u$; clique $c$ with the set of all
    variables in the clique $N_c$; abort threshold $\delta$
 1: init $B, I$ with global bounds
 2: if $c$ is an at-most-one clique then
 3:     set all $x \in N_c$ to 0; propagate
 4:     if infeasible then
 5:         upgrade clique to an exactly-one clique
 6:     else
 7:         update $B, I$
 8: for $x \in N_c$ do
 9:     set $x$ to 1; set all $y \in N_c / \{x\}$ to 0; propagate
10:     if MAXGLOBALBOUNDCHANGESANDIMPLICATIONS$(B, I) \leq \delta$ then
11:         abort
12:     if infeasible then
13:         fix $x$ to 0
14:     else
15:         update $B, I$
16: analyze $B$ and apply global valid variable bounds
17: analyze $I$ and apply substitutions
18: if all propagations infeasible then
19:     return global infeasibility
```

variable is not reached. Afterwards, traditional probing is continued in Line 10 on the variables (N_P) that have not been touched by clique probing (Line 6). Clique probing is disabled for future runs if it finds no reductions at all (Line 9).

2.4 Implementation

For our implementation, we build upon the open-source presolving C++ library PAPILO [7], which is publicly available on GitHub[3]. PAPILO implements common presolving techniques [2], supports multi-precision, and accelerates the presolving process through parallelization. To minimize synchronization overhead, it adopts a transaction-based architecture: each presolver records its reductions independently in the form of a transaction rather than applying them immediately. These transactions are then returned to the PAPILO core, which sequentially checks their validity and applies them if still applicable.

Whenever probing is invoked in PAPILO, our *clique probing* procedure is executed first, followed by the standard probing routine. In the latter, we exclude variables that have already been probed or fixed by clique probing. Additionally, all bound changes and fixings identified by clique probing are applied locally to the instance used for probing. Candidate variables are ranked using the same scoring function employed by PAPILO's built-in probing routine.

In order to make use of the existing parallelization framework in PAPILO, we implemented the option to parallelize the for loops in Line 4 in Algorithm 2

---

[3] https://github.com/scipopt/papilo

Algorithm 2 CliqueProbing

```
Require: a list of cliques $C$, probing score for binary variables $S$, list of binary variables in the MIP $N$
Sort $C$ by average probing score of variables $S$
$C \leftarrow$ highest-scored cliques, skipping cliques with more than 50% overlapping variables
$N_P \leftarrow N$
for $c \in C$ do
    ProbeSingleClique($c$)
    $N_P \leftarrow N_P \backslash \{N_c\}$
    if two consecutive calls of ProbeSingleClique were unsuccessful then
        break
disable clique probing for future runs if no reductions were found
resume with Probing($N_P$)
```

and Line 8 in Algorithm 1. Each thread receives its own local copy of B and I in order to avoid synchronization overhead for the global data structures B and I in Algorithm 1, Variable assignments are processed in batches, and after each batch, the local data structures are synchronized sequentially with the global ones. With this implementation, no locking during the threads is necessary.

In our experiments, however, all algorithms are executed sequentially in order to determine the pure algorithmic advantage of clique probing. Furthermore, PaPILO currently has no API to pass implications to an interfaced MIP solver and does not use them internally. Hence, we do not consider them in our experiments.

3 Computational Study

In this section, we evaluate how the theoretical speedup translates into practical performance. In particular, we are interested in investigating the following questions: (a) Is clique probing more effective in the sense that it yields more reductions? (b) How does clique probing affect the overall performance of the solver? The first two questions are examined in Section 3.2 and the latter in Section 3.3. First, we explain the experimental setup in Section 3.1.

3.1 Experimental setup

We implemented clique probing in PaPILO [7], which functions as a standalone presolving library and is also integrated as a default plugin within the presolving phase of the open-source MIP solver SCIP. In our experiments, we run SCIP, which invokes PaPILO as part of its presolving routine. It is important to note that PaPILO constitutes only one component of the overall presolving process in SCIP and may be called multiple times during presolving, with presolving as a whole potentially being called multiple times during the solving process. Since presolve routines are called before the invocation of PaPILO within SCIP, PaPILO is already executed on a reduced problem and not on the original problem.

Testset We base our experiments on the MIPLIB 2017 benchmark set [8] consisting of 240 instances and solve each instance with four different random seeds. We exclude instances that can neither be solved with clique probing enabled nor disabled, on four different seeds, in two hours. Further, we exclude instances where PaPILO does not identify cliques since they are not affected by the changes. This results in a total of 93 instances. Each seed combination is treated as a separate observation, resulting in a total number of 372 runs.

Software & Hardware For our experiments, we use our implementation in PaPILO 3.0 githash `0e1165d6` and compare with the default PaPILO githash `32328a5c` as a baseline, which is a verbose variant of `d808e1f3`. We use a development version of SCIP 10.0 with githash `80c1fbfe1b`. The experiments were carried out on identical machines with Intel(R) Xeon(R) CPU E7-8880 v4 @ 2.20 GHz.

Parametrization By default, PaPILO within SCIP is executed sequentially, hence we also used the sequential mode in our experiments. Since probing large cliques is computationally expensive, we focus only on promising candidate cliques and aim to terminate the probing process early and parameterize Algorithms 2 and 1 accordingly. This approach aligns with the general philosophy of probing. To avoid long and unsuccessful runtimes, we limit the maximum size of cliques considered to 150 and the maximum number of variables in cliques probed in a single run to 3000. The initial batch size of cliques is set to 2, which determines how many cliques are considered at the start of the process. Clique probing is aborted if, for two consecutive iterations, fewer than three reductions per propagation are found. Analogous to probing, we define an additional abortion criterion that terminates clique probing if a threshold of reductions or work limit is exceeded. If no reduction is found, we also disable clique probing for future rounds of presolving. Additionally, to avoid probing duplicate variables, we restrict the maximum ratio of probed on variables in a clique to 50%.

3.2 Analysis of clique probing

In this section, we analyze the performance of clique probing in terms of the number of fixings, substitutions, and bound changes found in comparison to default PaPILO for the first call within SCIP. A detailed table containing the propagations performed per second, the reductions found per second, and the reductions found for each instance is provided in the repository[4].

Figure 1 visualizes the total number of reductions found, comparing the default PaPILO version (horizontal axis) with the clique probing version (vertical axis). Each data point represents an observation for one instance. Blue points

---

[4] https://github.com/alexhoen/papilo/tree/clique-probing-paper/Paper

indicate the number of fixings, green crosses show the number of bound changes, and red circles represent substitutions found. Instances for which the numbers of fixings, substitutions, and bound changes are identical, and the runtime differs by less than 5% or by at most 0.2 seconds, are omitted from both the table and the figure. For each of the three categories, the vast majority of data points lie on or above the diagonal, which demonstrates that clique probing is overall more effective than standard probing.

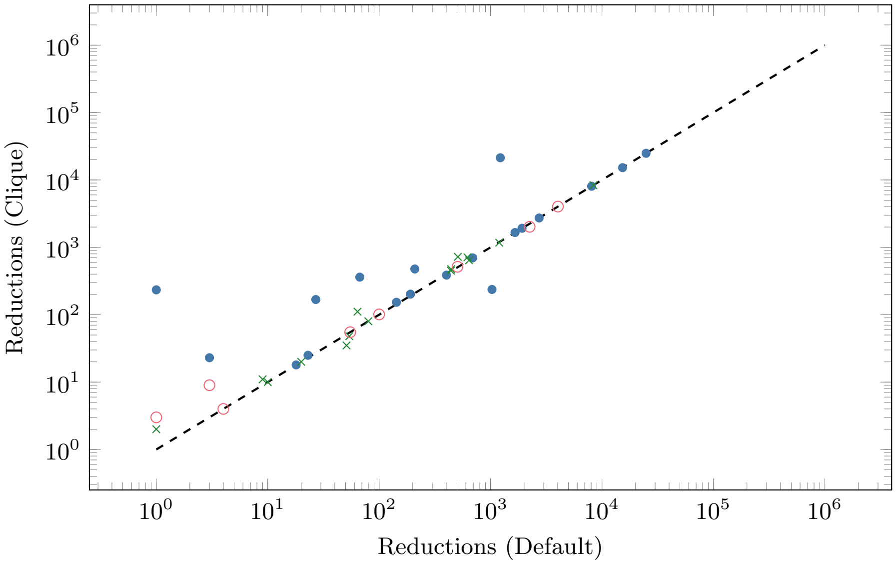

Fig. 1. Visualization of reductions of different types, fixings are marked as blue dots, substitutions as red circles, and bound changes as green crosses.

We highlight several instances with particularly pronounced differences. Two notable negative outliners are the instances NET12 and SQUARE47: On SQUARE47, the runtime of probing increases from 68.5 to 173.8 seconds. While default probing aborts probing on the first batch of ten variables without finding any reductions, Clique probing initially selects two large cliques, which takes a significant amount of time and returns 41 bound changes. On NET12, an unfavorable initial clique selection results in premature termination of clique probing, whereas the variable selection of default probing is comparatively "lucky": it probes 1600 variables instead of 250 and finds more reductions.

In contrast to these outliers, most instances show a clear increase in reductions with only a marginal increase in runtime. In particular, for AIR05, NEOS8, NEOS-860300, PIPEROUT-08, PIPEROUT-27, SUPPORTCASE33 the improvement is pronounced: the number of fixings increases from 0 to 138, from 1233 to 21 340, from 27 to 168, from 209 to 477, from 67 to 361, and from 1 to 234, respectively, with only a small overhead in runtime.

Generally, the runtime of probing increases slightly when clique probing is active. However, the probing process becomes significantly more efficient. Compared to using standard probing, PaPILO with clique probing finds 5% more reductions per second of presolving. A drawback is that aborting the clique probing process discards all progress made for that clique, which can, for example, on the SQUARE47 instance, increase the overall runtime.

Note that even a small number of additional reductions can substantially impact the behavior of the underlying solver since (a) presolvers are executed in a round-based manner to iteratively improve the model and therefore interact with each other, and (b) contrary to what the name might imply, presolving is invoked multiple times during the overall solution process, for example, when solving sub-MIPs or during restarts.

3.3 Impact on overall solver performance

Although the above results are promising, it is difficult to assess solely from the number of reductions and the recorded times how the subsequent solving process behaves with these new changes. While each reduction achieved through (clique-)probing strengthens the problem formulation, we want to analyze the effectiveness of the reductions obtained by the clique probing approach when integrated within a solver.

In Table 1, we compare the performance of clique probing in SCIP against the default configuration. All run times are given in seconds and aggregated using the shifted geometric mean with a shift of 1 (sgm1). Nodes are aggregated using the shifted geometric mean with a shift of 100 (sgm100). The row "all" summarizes results over the full test set, while the rows $[s, tlim]$ include only instances which were solved by at least one configuration and for which at least one configuration took at least s seconds to solve. The row "affected" lists only those instances where the branch-and-bound tree was altered by clique probing.

Overall, clique probing solves 324 instances compared to 321 in the default setting, thus improving the total number of solved instances by three. We also observe a relative speed-up of approximately 3% in terms of solving time. This improvement becomes more pronounced on harder instances: for instances requiring at least 100 seconds to solve, the speed-up increases to 6%, and for those requiring at least 1000 seconds, it reaches 8%. The results, therefore, indicate that clique probing is particularly effective on more challenging instances.

4 Conclusion

In this paper, we presented clique probing, an approach that exploits the model's existing data structures to avoid unnecessary propagations during probing while simultaneously providing a stronger relaxation. As demonstrated in our experiments, this results in a substantial increase in propagations per second and,

Table 1. Performance comparison between SCIP with PaPILO with clique probing enabled vs. default PaPILO. Times are displayed in seconds and aggregated by sgm1. The number of nodes is aggregated by sgm100.

		clique probing			default			relative	
Subset	instances	solved	time	nodes	solved	time	nodes	time	nodes
all	372	324	310.56	2125	321	318.98	2139	1.03	1.01
affected	340	312	350.85	3979	309	372.52	4127	1.06	1.04
[0,tlim]	352	324	214.30	2236	321	223.45	2285	1.04	1.02
[1,tlim]	352	324	214.30	2236	321	223.45	2285	1.04	1.02
[10,tlim]	332	304	330.23	3050	301	343.43	3119	1.04	1.02
[100,tlim]	264	236	758.98	7138	233	804.32	7281	1.06	1.02
[1000,tlim]	176	148	1987.33	19533	145	2142.13	20793	1.08	1.06

consequently, slightly more reductions during probing. Overall, this yields a 3% speed-up in SCIP on instances containing cliques.

There remains, however, room for further improvement. (a) Since PaPILO does not maintain a clique table, and SCIP does not share its clique table with PaPILO, we are currently limited to detecting only those cliques that appear explicitly in the model; and (b) there is presently no mechanism for communicating implications back to the underlying solver within PaPILO.

Despite these limitations, the results are encouraging and suggest that similar techniques could be applied to other data structures, such as SOS2 constraints, opening up promising directions for future research.

Acknowledgements: The work for this article was supported through the Research Campus MODAL funded by the German Federal Ministry of Research, Technology, and Space (fund numbers 05M14ZAM, 05M20ZBM, 05M2025). Alexander was supported by the Bundesministerium für Bildung und Forschung (16DHBKI071).

References

1. Achterberg, T.: Constraint Integer Programming. Ph.D. thesis (01 2007). https://doi.org/10.14279/depositonce-1634
2. Achterberg, T., Bixby, R., Gu, Z., Rothberg, E., Weninger, D.: Presolve reductions in mixed integer programming. INFORMS Journal on Computing **32** (11 2019). https://doi.org/10.1287/ijoc.2018.0857
3. Achterberg, T., Wunderling, R.: Mixed integer programming: Analyzing 12 years of progress. In: Facets of combinatorial optimization: Festschrift for martin grötschel, pp. 449–481. Springer (2013)
4. Atamtürk, A.: Cover and flow cover inequalities for capacitated knapsack polyhedra. Mathematical Programming **99**(1), 63–80 (2004). https://doi.org/10.1007/s10107-003-0467-0
5. Balas, E.: Facets of the knapsack polytope. Mathematical Programming **8**, 146–164 (1975). https://doi.org/10.1007/BF01580442

6. Bolusani, S., Besançon, M., Bestuzheva, K., Chmiela, A., Dionísio, J., Donkiewicz, T., van Doornmalen, J., Eifler, L., Ghannam, M., Gleixner, A., Graczyk, C., Halbig, K., Hedtke, I., Hoen, A., Hojny, C., van der Hulst, R., Kamp, D., Koch, T., Kofler, K., Lentz, J., Manns, J., Mexi, G., Mühmer, E., Pfetsch, M.E., Schlösser, F., Serrano, F., Shinano, Y., Turner, M., Vigerske, S., Weninger, D., Xu, L.: The scip optimization suite 9.0 (2024), https://arxiv.org/abs/2402.17702
7. Gleixner, A., Gottwald, L., Hoen, A.: PaPILO: A parallel presolving library for integer and linear programming with multiprecision support. INFORMS Journal on Computing (2023). https://doi.org/10.1287/ijoc.2022.0171.cd, https://github.com/INFORMSJoC/2022.0171
8. Gleixner, A., Hendel, G., Gamrath, G., Achterberg, T., Bastubbe, M., Berthold, T., Christophel, P.M., Jarck, K., Koch, T., Linderoth, J., Lübbecke, M., Mittelmann, H., Ozyurt, D., Ralphs, T., Salvagnin, D., Shinano, Y.: Miplib 2017: Data-driven compilation of the 6th mixed-integer programming library. Mathematical Programming Computation **13**(3), 443 – 490 (2021). https://doi.org/10.1007/s12532-020-00194-3
9. Huangfu, Q., Hall, J.A.J.: Parallelizing the dual revised simplex method (2018). https://doi.org/10.1007/s12532-017-0130-5, https://doi.org/10.1007/s12532-017-0130-5
10. Nemhauser, G.L., Trotter, L.E.: Vertex packings: structural properties and algorithms. Mathematical Programming **8**, 232–248 (1975). https://doi.org/10.1007/BF01580444
11. Savelsbergh, M.: Preprocessing and probing techniques for mixed integer programming problems. ORSA Journal on Computing **6** (11 1994). https://doi.org/10.1287/ijoc.6.4.445

Table 2. Results of the first probing call of PaPILO in SCIP. Columns show Default vs. Clique Probing for each metric: Time (s), variable fixings (Fixings), substitutions (Subs), bound changes (BChgs), propagations (Props), average propagations per second (Props/s), and reductions per propagation (Reds/Prop).

	Time [s]		Fixings		Subs		BChgs		Props		Reds/Prop	
Instance	Def	Clique	Def	Clique	Def	Clique	Def	Clique	Def	Clique	Def	Clique
30n20b8	0.2	0.2	3	23	0	0	0	1	40	320	0.1	0.1
air05	0.1	0.5	0	138	0	0	0	0	20	392	0.0	0.4
bppc4-08	0.0	0.2	0	0	0	0	0	1	30	543	0.0	0.0
brazil3	0.2	0.3	0	10	0	0	0	0	138	791	0.0	0.0
co-100	3.2	3.9	24869	24869	80	80	0	0	62	95	402.4	262.6
cryptanalysiskb128n5obj16	5.7	6.6	690	697	8382	8348	0	0	51316	43533	0.2	0.2
ex10	141.5	74.2	15297	15206	620	711	0	0	32808	18831	0.5	0.8
ex9	14.8	16.0	8077	8063	443	474	0	0	10216	9138	0.8	0.9
mzzv11	0.3	0.4	191	202	1	2	0	0	262	398	0.7	0.5
mzzv42z	0.2	0.3	143	153	9	11	0	0	258	647	0.6	0.3
neos-3216931-puriri	0.1	0.2	23	25	0	0	0	0	176	235	0.1	0.1
neos-4722843-widden	4.2	10.6	1659	1659	509	723	2243	2017	64928	70320	0.1	0.1
neos-5114902-kasavu	12.2	12.9	0	0	0	0	0	0	2000	2009	0.0	0.0
neos-5195221-niemur	0.5	1.4	1919	1919	441	450	4026	4026	16894	19796	0.4	0.3
neos-662469	0.2	0.4	0	0	0	0	0	0	24	299	0.0	0.0
neos-860300	0.8	1.5	27	168	0	0	0	0	120	384	0.2	0.4
neos8	5.9	1.9	1223	21340	20	20	4	4	3502	3459	0.4	6.2
neos-957323	1.1	1.9	0	40	0	0	0	0	164	519	0.0	0.1
net12	0.2	0.2	1030	237	51	35	0	0	3236	921	0.3	0.3
nexp-150-20-8-5	0.1	0.1	0	0	0	0	3	9	3462	3518	0.0	0.0
ns1644855	10.1	12.4	0	0	0	0	100	101	1950	2004	0.1	0.1
ns1760995	6.8	31.0	0	71	0	0	0	30	266	1815	0.0	0.1
piperout-08	0.2	0.3	209	477	64	111	0	26	682	1024	0.4	0.6
piperout-27	0.2	0.4	67	361	54	48	0	2	266	959	0.5	0.4
rocI-4-11	0.2	0.2	402	387	1196	1173	505	514	6156	5733	0.3	0.4
rocII-5-11	2.9	2.8	2728	2727	10	10	55	55	10068	9237	0.3	0.3
s250r10	13.7	23.0	18	18	641	641	0	0	152	420	4.3	1.6
square47	69.5	173.8	0	0	0	0	0	41	20	377	0.0	0.1
supportcase33	0.3	1.1	1	234	0	0	1	3	62	503	0.0	0.5
wachplan	0.0	0.0	0	0	0	0	0	1	22	64	0.0	0.0