Strong Bridges for the Circuit Constraint

Implementing and Evaluating Strong Bridge Detection in a Lazy Clause Generation Solver

Martijn van Leest
Supervisors: Emir Demirović, Imko Marijnissen
EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 21, 2026

Name of the student: Martijn van Leest
Final project course: CSE3000 Research Project
Thesis committee: Emir Demirović, Imko Marijnissen, Andreea Costea

Abstract

Constraint Programming with Lazy Clause Generation (LCG) relies on effective propagation to reduce the search space. We study the integration of strong bridge detection into a `circuit` propagator, where strong bridges identify edges that must be present to maintain strong connectivity in the graph induced by current domains. Compared to a baseline that only prevents subcycles, this approach substantially reduces search effort, with reductions of up to three orders of magnitude in the satisfaction setting and one order of magnitude in the optimization setting. These gains often translate into runtime improvements, particularly for satisfaction. The extension also produces shorter nogoods, while its effect on LBD is mixed, reflecting more global explanations. Overall, the results demonstrate that identifying necessary edges is an effective way to strengthen propagation and significantly reduce the explored search space.

1 Introduction

Combinatorial optimization problems arise in many real-world applications such as scheduling [1], routing [2], and planning [3], where optimal decisions must be made under complex requirements [4]. Many of these problems are NP-hard, and Constraint Programming (CP) provides an effective paradigm for solving them by modeling relationships between variables as constraints and searching for satisfying assignments [5].

A central idea in CP is propagation, where constraints iteratively reduce variable domains by eliminating infeasible values. By pruning the search space early, propagation plays a crucial role in improving solver efficiency.

In this work, we consider CP in the context of Lazy Clause Generation (LCG), which combines CP with SAT-based learning [6, 7]. In LCG, each propagation must be accompanied by an explanation, from which clauses are learned. These clauses capture the conditions under which a conflict occurs and prevent the solver from repeating similar assignments during subsequent search.

Propagators inherently involve a trade-off between the strength of their inference and the computational effort required to perform it [8]. Strong propagators typically rely on more complex reasoning, increasing the computational cost of propagation.

This work focuses on the `circuit` constraint [9], which enforces a set of variables to form a Hamiltonian cycle over a graph. This problem is NP-complete, making effective propagation essential for practical solving [10]. When combined with an objective function, such as in the Traveling Salesperson Problem (TSP) [11], the problem becomes NP-hard.

Existing propagation techniques for the `circuit` constraint range from local cycle prevention to global connectivity-based reasoning. In particular, Francis and Stuckey [12] propose a method based on strongly connected components (SCCs), which exploits the requirement that a Hamiltonian cycle induces a single SCC. Another approach from Kaya and Hooker [13] eliminates edges that cannot belong to any Hamiltonian cycle. While these methods can prevent infeasible structures or remove impossible edges, they do not explicitly identify which edges are required to maintain connectivity.

This can be addressed by incorporating strong bridge detection into circuit propagation. A strong bridge is an edge whose removal breaks strong connectivity and must therefore appear in any feasible solution. Detecting such edges enables the propagator to enforce them directly, introducing a form of edge-level reasoning that complements existing connectivity-based techniques (see Figure 1 for an example).

In this paper, we investigate the impact of integrating strong bridge detection into the `circuit` propagator in an LCG setting. We analyze how this extension affects search effort, explanation quality, and runtime, and how these effects depend on instance properties such as graph size and density. The evaluation considers both satisfaction and optimization settings to assess whether the observed improvements generalize across different problem formulations.

Our results show that strong bridge propagation substantially reduces search effort, with reductions of at least 80% in number of conflicts. Propagation reductions are typically above 80%, though lower for sparse instances, where they still exceed 40%. These improvements are often accompanied by runtime improvements, particularly in the satisfaction setting, while results in the optimization setting are more variable. The extension also affects explanation quality: it consistently produces shorter nogoods, with reductions exceeding 80% for sparse instances and small to moderate improvements otherwise, while its impact on LBD is more mixed, ranging from substantial reductions for sparse instances to moderate increases for larger and denser configurations, indicating that explanations involve literals from more distinct decision levels and capture less localized dependencies. Overall, the results demonstrate that strong bridge reasoning enables more effective pruning of the search space, with performance gains that depend on instance structure.

The contributions of this paper are threefold. First, we extend a `circuit` propagator with strong bridge detection for LCG. Second, we develop explanations for strong bridge propagation. Third, we provide an empirical evaluation across instances of varying size and density, analyzing search effort, runtime, and explanation characteristics.

The remainder of the paper is structured as follows. Section 2 places this work in context of existing research. Section 3 introduces background concepts. Section 4 presents the propagation algorithms. Section 5 describes the experiments. Section 6 discusses reproducibility, and Section 7 concludes.

(a) Before strong bridge propagation

(b) After strong bridge propagation

Figure 1: Example of strong bridges (red), which are required to maintain strong connectivity and can be enforced without search. In contrast, cycle-prevention does not prune edges in this state without search, as it only detects subcycles following fixed paths.

2 Related work

We review existing propagation techniques for the `circuit` constraint and related concepts from graph theory that inform stronger propagation.

2.1 Circuit constraint propagation

The `circuit` constraint originates from Beldiceanu and Contejean [9], who introduced global constraints to capture structural properties in CP models. It is a special case of the `cycle` constraint, enforcing a single Hamiltonian cycle.

A simple and effective propagation approach is based on subcycle prevention, originally proposed by Caseau and Laburthe [14]. This method removes assignments that would form premature subcycles. While efficient, it is purely local and does not enforce global connectivity.

To strengthen propagation, Francis and Stuckey [12] extend subcycle prevention and introduce reasoning based on strongly connected components (SCCs) within an LCG framework [6], leveraging Tarjan's [15] algorithm for detecting SCCs. Their approach enforces global connectivity of the candidate graph, enabling stronger pruning. However, it does not eliminate subcycles that remain globally connected, nor does it identify which individual edges are required to preserve connectivity.

Work from Kaya and Hooker [13] focuses on removing edges that cannot belong to any Hamiltonian cycle. They propose a filtering algorithm that eliminates edges by exploiting separator-based reasoning, identifying edges that cannot participate in any Hamiltonian cycle due to the connectivity structure they induce. While effective in pruning such edges, this approach does not capture which edges are required in all feasible solutions.

Across these approaches, a distinction arises between local reasoning, which prevents substructures, and global reasoning, which enforces connectivity. However, none explicitly identifies edges that must be present in every feasible circuit.

2.2 Strong bridges

Strong bridges are a classical concept in graph theory, defined as edges whose removal increases the number of strongly connected components in a directed graph [16]. They capture critical dependencies in the graph, as their removal leads to a loss of strong connectivity. In contrast to SCC-based reasoning, which detects when connectivity is violated, strong bridge detection identifies edges whose removal would *cause* such violations.

Despite their relevance for capturing critical connectivity, strong bridges have, to our knowledge, not been explicitly exploited for propagation in the context of the `circuit` constraint.

2.3 Research gap

Existing propagation techniques for the `circuit` constraint either use local reasoning to prevent the creation of subcycles, or exploit global connectivity properties to prune infeasible assignments. While such approaches provide valuable information, they typically reason at the level of groups of nodes or paths, rather than explicitly characterizing individual edges as critical structural elements. As a result, edges may be enforced indirectly as a *consequence* of the propagation process, but they are not identified through a dedicated notion of edge criticality. Strong bridges provide a natural mechanism for identifying such edges, directly addressing this limitation and motivating their integration into `circuit` propagation.

3 Preliminaries

3.1 Constraint Programming

Constraint Programming (CP) is a paradigm for solving combinatorial problems by modeling them as variables with associated domains and constraints restricting their values [5].

Formally, a **Constraint Satisfaction Problem** (CSP) consists of a set of variables $X = \{x_1, \ldots, x_n\}$, corresponding domains $D = \{d_1, \ldots, d_n\}$, and a set of constraints $C = \{c_1, \ldots, c_m\}$. A solution assigns each variable a value from its domain such that all constraints in C are satisfied. A **Constraint Optimization Problem** (COP) extends this by including an objective function f over assignments, which is to be minimized or maximized subject to the constraints.

A key mechanism in CP is **propagation**, where constraints iteratively remove values from domains that cannot be part of any feasible solution. This process is performed by **propagators**.

3.2 Lazy Clause Generation

Lazy clause generation (LCG) extends CP with clause learning techniques from SAT solving. In LCG, each propagation must be accompanied by an **explanation** (or **reason**) that justifies the inference [7].

An **atomic constraint** or **atom** is a basic constraint such as $x = v$, $x \neq v$, or $x \leq v$. Explanations are expressed as an implication in terms of atoms:

$$\bigwedge_{p \in premises} p \rightarrow consequence.$$

From these explanations, the solver derives **nogoods** that prevent the same conflict, or similar conflicting assignments, from being revisited. This enables **backjumping**, allowing the solver to return directly to the source of a conflict and avoid revisiting the same conflicting assignments.

Smaller explanations are generally preferred, as they yield more widely applicable nogoods and improve learning effectiveness. Additionally, the quality of learned nogoods is often measured using the **Literal Block Distance** (LBD) [17], defined as the number of distinct decision levels of the literals in the clause. Lower LBD values generally indicate more effective nogoods, as they depend on fewer decision levels and are thus more likely to propagate again.

3.3 Graph concepts

- A **directed graph** $G = (V, E)$ consists of a set of vertices V and directed edges $E \subseteq V \times V$.
- A **path** from vertex u to vertex v is a sequence of vertices such that each consecutive pair is connected by a directed edge. Vertex v is **reachable** from u if such a path exists.
- A **strongly connected component (SCC)** is a maximal subset of vertices in which every pair of vertices is mutually reachable.

- An edge $(u, v) \in E$ is a **strong bridge** if its removal increases the number of strongly connected components of the graph.

3.4 The `circuit` constraint

The `circuit` constraint is defined over variables $X = \{x_1, \ldots, x_n\}$, where each variable x_i represents the successor of node i, and $\text{dom}(x_i)$ specifies its possible successors.

A solution assigns each variable such that each node has exactly one successor and all nodes form a single cycle, i.e., a Hamiltonian cycle.

This constraint can be represented as a directed graph with nodes V and edges E:

$$V = \{i \mid x_i \in X\}, E = \{(i, j) \mid j \in \text{dom}(x_i)\}$$

A feasible solution corresponds to selecting exactly one outgoing edge per node such that the resulting subgraph forms a single directed cycle over all nodes.

3.5 Baseline `circuit` propagation

One approach to propagating the `circuit` constraint is based on cycle prevention [12, 14].

- A **fixed edge** (u, v) corresponds to an assignment $x_u = v$.
- A **chain** $(v_1, \ldots, v_k)$ consists of fixed edges $x_{v_i} = v_{i+1}$ for $i < k$, where v_k has no fixed successor.
- For a chain from u to v, the value u is removed from $\text{dom}(x_v)$ if the chain does not contain all nodes, as assigning $x_v = u$ would form a subcycle.

This ensures that partial assignments remain extendable to a full Hamiltonian cycle. In LCG, such removals must be explained by the assignments defining the chain, which together imply that the subcycle would occur.

4 Methodology

We describe a propagation algorithm for the `circuit` constraint based on strong bridge detection, together with explanations for integration into a lazy clause generation (LCG) solver.

4.1 Strong Bridge Motivation

For the `circuit` constraint, global connectivity is essential: any solution must form a single directed cycle visiting all nodes, and thus the underlying graph must be strongly connected. This motivates identifying edges that are necessary to preserve connectivity. As shown in Figure 1, strong bridge propagation can enforce such edges directly from the current domain structure, whereas baseline cycle-prevention relies on fixed assignments and cannot prune in that case. More generally, this highlights that strong bridge propagation strengthens the overall propagation by exploiting global structure beyond what cycle-prevention alone captures. As a result, it can enable inferences to be made earlier in the search process compared to relying solely on cycle-prevention.

4.2 Reachability-Based Propagation

Our approach builds on the standard characterization that a directed graph is strongly connected if and only if all nodes are mutually reachable. In practice, this can be verified using two reachability traversals, one in the original graph and one in the reversed graph, ensuring that every node can both reach and be reached from the chosen start node.

Our contribution is to integrate strong bridge detection into the `circuit` propagator by exploiting reachability in the graph. In particular, we use reachability not only to assess connectivity, but to identify edges whose removal would violate this property. At a high level, the propagator considers each candidate edge and checks whether its removal breaks reachability; if so, the edge is identified as necessary and enforced. This reasoning is performed repeatedly during search, adapting to the current domains.

Intuition. Intuitively, an edge (u, v) is a strong bridge if removing it makes node v unreachable from u. In that case, no alternative path exists between these nodes. Since a feasible solution to the `circuit` constraint must correspond to a strongly connected graph, such an edge must be part of every valid solution and can therefore be enforced.

Formalization. Let $G = (V, E)$ be the graph induced by the current domains:

$$V = \{i \mid x_i \in X\}, \quad E = \{(i, j) \mid j \in \mathrm{dom}(x_i)\}.$$

Assuming G is strongly connected, for each edge $(u, v) \in E$ we consider the graph $G' = (V, E \setminus \{(u, v)\})$. If v is not reachable from u in G', then (u, v) is a strong bridge and the assignment $x_u = v$ is enforced. This condition is checked using a depth-first search from node u in G'. In our current implementation, this requires one traversal per edge, resulting in a worst-case complexity of $O(|E|(|V| + |E|))$ per propagation step. While more efficient algorithms for detecting strong bridges exist [16], we adopt this straightforward approach to focus on the propagation strength rather than optimizing the underlying graph computation.

Correctness follows from the fact that any Hamiltonian cycle induces a strongly connected graph. If removing (u, v) breaks reachability from u to v, then no alternative path exists between these nodes, and any feasible solution must include (u, v). Hence, enforcing $x_u = v$ is sound.

4.3 Explanation Construction

In LCG, each propagation or conflict must be accompanied by an explanation expressed as a set of atoms. In principle, explanations can be made minimal by identifying only the atoms necessary for the inference, but computing such minimal explanations is often too expensive in practice. Our approach therefore avoids the additional cost of computing minimal explanations and instead uses two different explanation strategies.

When the graph is no longer strongly connected, we raise a conflict using a coarse explanation based on all current domain restrictions. This approach avoids additional computation but always contains redundant literals. An alternative to be explored is to derive explanations based on minimal separating structures, such as cuts or articulation points, that witness the loss of strong connectivity. This could lead to substantially smaller explanations, but requires additional reasoning to identify such structures during propagation.

For strong bridge propagation, explanations are derived from reachability after removing an edge. Let (u, v) be an edge identified as a strong bridge, and let G' be the graph obtained

by removing this edge. Let R be the set of nodes reachable from u in G', and let $U = V \setminus R$. By construction, $v \in U$, and any path from u to v would require an edge from R to U. Since no such edge exists, (u, v) is necessary to preserve connectivity.

For each $i \in R$, we consider values $j \in U$ that were originally part of $\mathrm{dom}_0(x_i)$ but have since been removed. Each such removal corresponds to a pruned edge (i, j) and contributes the literal $(x_i \neq j)$ to the explanation. The resulting explanation is:

$$\bigwedge_{i \in R,\, j \in U,\, j \in \mathrm{dom}_0(x_i),\, j \notin \mathrm{dom}(x_i)} (x_i \neq j) \;\rightarrow\; (x_u = v).$$

This explanation is sound but not necessarily minimal. The reachability partition may include nodes that are not relevant for paths between u and v, leading to redundant literals. Constructing smaller explanations would require identifying only edges that lie on potential paths, which is left for future work.

Compared to the baseline cycle-prevention propagator which operates using local chain reasoning (see subsection 3.5), the proposed method requires multiple reachability computations per propagation step. This increases computational cost but enables significantly stronger pruning by identifying edges that are necessary for connectivity.

4.4 Explanation Correctness

The correctness of the explanation follows from the LCG framework. The negation of the consequence $(x_u = v)$ is $x_u \neq v$, which removes edge (u, v). Under this assumption and all premises, all edges from R to U are absent. Consequently, no path exists from u to any node in U, including v. This contradicts the requirement that the graph must remain strongly connected in order to allow for a Hamiltonian cycle. Therefore, the propagation is sound.

5 Experimental Setup and Results

The goal of this section is to evaluate the impact of integrating strong bridge detection into the `circuit` propagator under LCG, in both satisfaction and optimization settings, to assess the robustness of the approach across different problem formulations.

The results confirm that strong bridge propagation substantially reduces search effort. In the satisfaction setting, conflicts are typically reduced by over 80% and often approach 99%, while propagation reductions range from around 40-60% for sparse instances to over 90% for denser ones. These gains often translate into runtime improvements, particularly for smaller and lower-density graphs, although the relationship between search reduction and runtime is not uniform. A consistent pattern appears at $k = 3$, where improvements are less pronounced.

In the optimization setting, improvements are smaller and more variable, indicating that overall performance depends not only on search reduction but also on other factors influencing runtime. In particular, for $k = 3$ with $n \geq 60$, slowdowns of up to 37% can occur despite reduced search effort, reflecting cases where the additional propagation cost is not fully offset by the achieved pruning.

The effect on explanation quality is more nuanced. Strong bridge propagation changes the structure of the implication graph, introducing more globally distributed dependencies between assignments. For smaller instances, this often reduces the number of required decisions, leading to lower LBD values. For larger and denser instances, however, these

dependencies span more of the search, causing explanations to involve literals from more decision levels and thus increasing LBD, despite the overall reduction in search effort. At the same time, the extension consistently produces shorter nogoods, as it complements chain-based reasoning with compact structural explanations.

Overall, strong bridge propagation significantly reduces the explored search space, with the magnitude of improvements depending on instance structure. The remainder of this section provides details on instance generation, experimental setup, and analysis with respect to search effort, explanation quality, and runtime.

5.1 Instance Generation

Instances are generated following the approach of Francis and Stuckey [12]. Each instance is a directed graph with n nodes, where each node connects to its k nearest neighbors based on Euclidean distance. Node positions are sampled uniformly in the unit square. Feasibility is ensured by constructing a Hamiltonian cycle via a random walk and adding edges where necessary. This yields structured graph instances with spatial and connectivity properties similar to network-style problems.

For the optimization setting, edge weights are derived from Euclidean distances and define the objective of minimizing total circuit cost. Distances are scaled and converted to integers (via truncation) to match the MiniZinc model (see Appendix E). This introduces a slight downward bias compared to true rounding, but is applied consistently across all configurations. A fixed search strategy with input-order variable selection and minimum value assignment is used to ensure that differences between configurations are primarily attributable to propagation rather than adaptive branching heuristics.

5.2 Experimental Setup

We compare the strong bridge extension against a baseline `circuit` propagator that prevents subcycles (see subsection 3.5). Experiments are conducted using Pumpkin [18] on an Intel i7-13700KF CPU (3.4 GHz, 16 cores, 24 logical processors) with 32GB RAM. Each run is single-threaded, with up to 10 instances processed in parallel. Although parallel execution may introduce minor variability due to shared hardware resources, all configurations are evaluated under identical conditions, ensuring fair comparison.

For satisfaction, we consider $n \in \{20, 40, 60, 80, 100\}$ and $k \in \{2, 3, 4, 5\}$; for optimization, we restrict the parameters to $n \in \{20, 40, 60, 80\}$ and $k \in \{2, 3, 4\}$. Configurations $(100, 5)$ (satisfaction) and $(80, 4)$ (optimization) are excluded because frequent timeouts result in too few solved instances for meaningful comparison.

For each pair of (n, k), 100 instances are generated. Results are averaged over solved instances only, excluding timeouts. Time limit is set to 30 minutes. Parameter n controls graph size, while k controls density, ranging from sparse to dense graphs. Timeout statistics are reported in Appendix D.

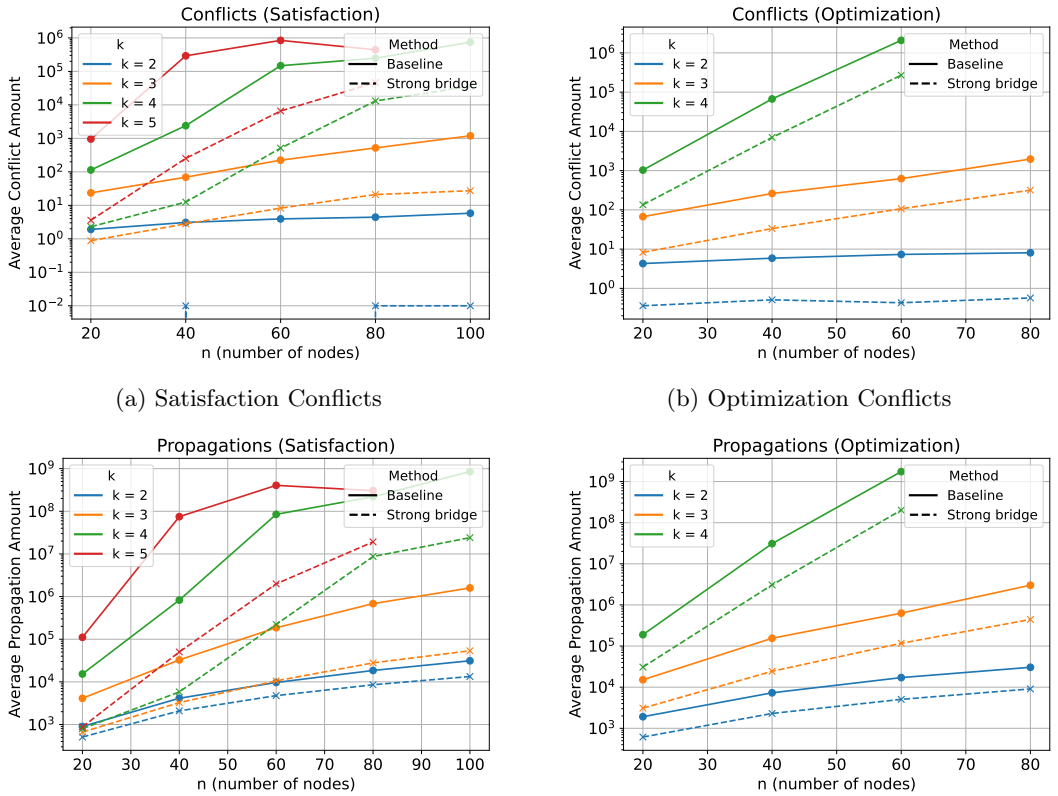
5.3 Experimental Results and Analysis

Search effort. Strong bridge detection substantially reduces search effort in both settings, as shown in Figure 2. In the satisfaction setting, conflict reductions are consistently very high, with average reductions above 88% across all configurations, and often exceeding 95%. Propagation reductions show a clearer dependence on graph structure, ranging from approximately 40-60% for sparse instances ($k = 2$) to over 90% for denser configurations.

In the optimization setting, reductions remain substantial. Conflicts are typically reduced by 80-94%, while propagations show reductions in the range of roughly 67-92%, depending on instance structure.

A notable but less pronounced pattern can be observed around $k = 3$, where reductions are smaller than for both sparser and denser configurations in terms of number of conflicts. Appendix A provides the full numerical details.

These results highlight that strong bridge detection is a highly effective mechanism for reducing search effort, enabling substantial pruning of the search space across both settings.



(a) Satisfaction Conflicts

(b) Optimization Conflicts

(c) Satisfaction Propagations

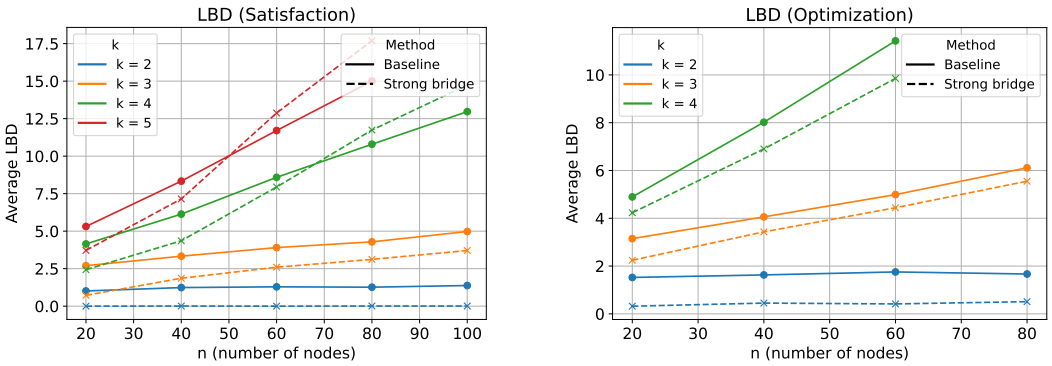
(d) Optimization Propagations

Figure 2: Average number of conflicts and propagations as a function of instance size for both variants. The satisfaction setting shows reductions of up to three orders of magnitude when the extension is used, whereas the optimization setting shows reductions of up to one order of magnitude.

Explanation quality. The effect of strong bridge propagation on explanation quality depends on instance size and density. Figure 3 reports the corresponding LBD values. In the satisfaction setting, the extension initially yields substantially lower average LBD values for smaller and sparser instances, with reductions often exceeding 70–100% for $k = 2$ and around 30–70% for $k \geq 3$. As instance size increases, this effect diminishes and may reverse, with LBD increasing by up to 20% for $n \geq 60$ and $k = 5$ and 12% for $n = 100$ and $k = 4$.

This behavior can be explained by how strong bridge propagation influences the structure of explanations. By enforcing edges based on global connectivity, it introduces more widely distributed dependencies between assignments. For smaller instances, this often reduces the number of required decisions, leading to explanations that involve fewer decision levels and thus lower LBD. As instance size grows, however, these dependencies span a larger portion of the graph, causing explanations to involve literals from more distinct decision levels and increasing LBD, despite the overall reduction in search effort.

Although strong bridge propagations themselves occur relatively infrequently, they influence subsequent propagation and conflict formation, and thus affect LBD beyond their individual occurrences. In the optimization setting, LBD remains lower for the extension, consistent with the corresponding parameter range in the satisfaction setting, as only smaller instances are considered.



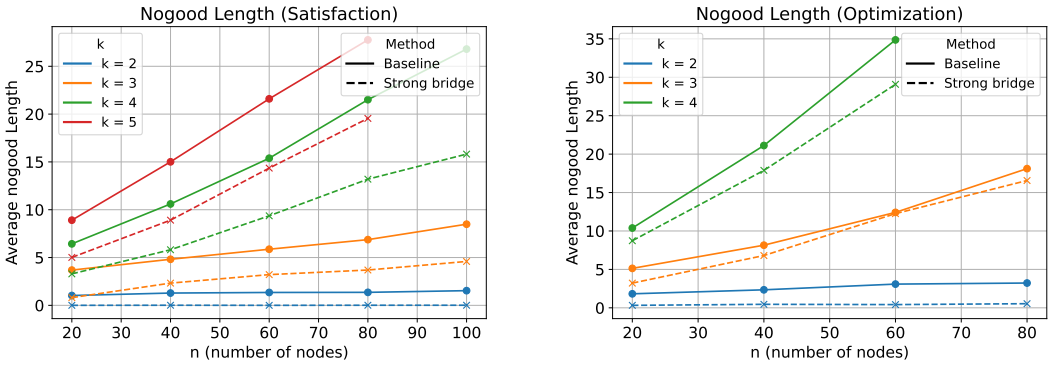
(a) Satisfaction LBD

(b) Optimization LBD

Figure 3: Average LBD value as a function of instance size for both variants in satisfaction and optimization setting. For small to moderate graph size, the extension results in lower LBD values, which is typically associated with better clause quality.

In contrast, the average nogood length is consistently smaller for the extension across both settings, as can be seen in Figure 4. In the satisfaction setting, reductions typically range from approximately 29-50% for moderate densities to over 95% for sparse instances. In the optimization setting, nogood length reductions remain positive but are generally smaller.

This behavior can be explained by the type of reasoning used. Baseline explanations rely on chain-like local reasoning, while strong bridge explanations are based on compact structural arguments over small separating sets. As a result, explanations contain fewer literals but may span more decision levels. Although connectivity-based conflicts can produce large explanations, these occur relatively infrequently and therefore have limited impact on average nogood length.



(a) Satisfaction Nogood Length

(b) Optimization Nogood Length

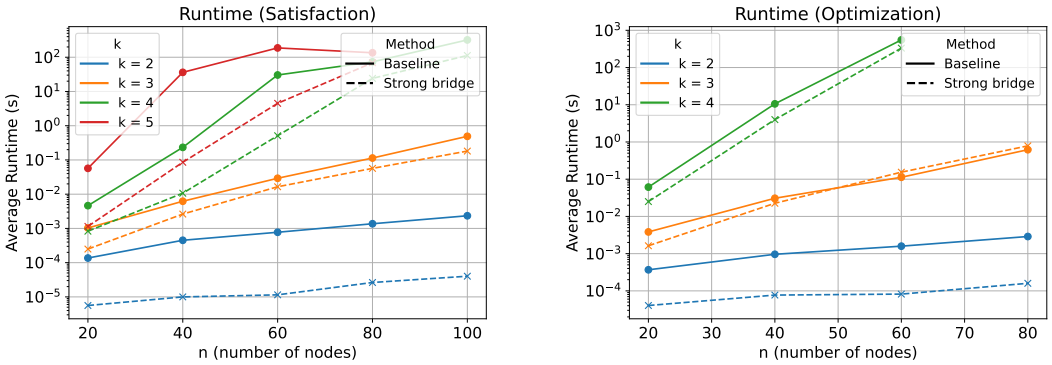
Figure 4: Average nogood length as a function of instance size for both variants in satisfaction and optimization setting. Strong bridge extension results in consistently lower nogood length, which is typically associated with better clause quality.

Runtime. While search effort and explanation quality are the primary focus of this evaluation, we report runtime as a complementary measure. Runtime is influenced by implementation details and overhead, and should therefore be interpreted mainly as a comparative indicator rather than a direct measure of propagation efficiency. Results are shown in Figure 5, with additional details in Appendix C.

Strong bridge propagation generally improves runtime in both settings, with the largest gains observed for sparse instances, with improvements of over 89% for $k = 2$. Improvements decrease with increasing density, and are consistently less pronounced at $k = 3$ compared to other values of k. In the optimization setting, this can occasionally lead to slowdowns when the additional propagation cost is not fully offset by the reduction in search effort, with slowdowns of approximately 25-40% observed for $n \geq 60$ and $k = 3$.

This behavior mirrors the conflict reduction results, and suggests that $k = 3$ represents the least favorable regime for strong bridge propagation in the current range of parameters. Although a comparable number of strong bridge propagations are triggered across different values of k for a given n (see Appendix A), the overall pruning effect appears to be smaller in this regime, indicating that these propagations lead to fewer subsequent inferences.

Overall, these results indicate that runtime improvements correlate closely with reductions in search effort, but should be interpreted as a secondary effect. The primary contribution of this work is to demonstrate that strong bridge propagation alters the structure of the search space and reduces the number of explored states, rather than directly optimizing runtime.



(a) Satisfaction Runtime

(b) Optimization Runtime

Figure 5: Average runtime as a function of instance size for both variants in satisfaction and optimization setting. Strong bridge extension generally results in lower runtime, with diminishing returns on bigger instances.

Limitations. While the results consistently demonstrate the effectiveness of strong bridge propagation, several limitations should be noted. First, the experiments rely on synthetically generated instances based on a single generator inspired by Francis and Stuckey [12]. Although this model produces structured graphs with relevant properties, it does not capture the full diversity of instances encountered in real-world applications, and the observed behavior may therefore be specific to this instance distribution.

Second, the use of a fixed search strategy is intended to isolate the impact of propagation, but does not reflect the behavior of modern CP solvers that employ sophisticated heuristics. In practice, interactions between strong bridge propagation and dynamic branching strategies could lead to different performance characteristics, and the extent to which the observed improvements transfer to such settings remains an open question.

Third, the experiments are limited to moderate instance sizes due to the computational cost of the evaluated configurations. As a result, the scalability of strong bridge propagation for larger graphs remains only partially explored, and the observed trends may not fully generalize to substantially larger instances.

Finally, the implementation includes additional instrumentation to collect detailed statistics, such as measuring time spent in different propagation components. While useful for analysis, this introduces overhead that may affect runtime measurements and potentially skew performance comparisons. Although this overhead is applied consistently across configurations, it should be taken into account when interpreting the results.

6 Responsible Research

All experiments in this work are designed with reproducibility and correctness in mind. Instance generation is controlled by fixed random seeds, ensuring deterministic behavior, and solver configurations are fully specified through command-line parameters, including a flag for enabling the strong bridge extension within a single code base. The entire experimental pipeline, covering generation, execution, and analysis, is automated and publicly available.[1]

[1] `https://github.com/Mvnlst/pumpkin-strong-bridge-experiments`

Reported results are tied to a documented seed, allowing full regeneration of instances and experiments.

Correctness is validated both internally, through Pumpkin's propagation checking mechanism, and externally, by comparing solutions against a reference solver using exhaustive enumeration on small instances, confirming that no valid solutions are lost or incorrectly introduced.

While the work does not involve human subjects or sensitive data, improved optimization techniques may influence real-world applications such as scheduling and network design, highlighting the importance of responsible use in practice.

7 Conclusions and Future Work

In this work, we studied the integration of strong bridge detection into a `circuit` propagator in the context of lazy clause generation, compared to a baseline propagator that only prevents subcycles. This complements existing connectivity-based techniques by incorporating stronger global structural reasoning into propagation.

The results show that strong bridge detection substantially reduces search effort compared to only using subcycle prevention, with reductions of up to three orders of magnitude in the satisfaction setting and up to one order of magnitude in the optimization setting. These improvements arise from more effective pruning of the search space and depend on instance structure, with consistently less pronounced improvements for $k = 3$. The extension also produces shorter, more informative nogoods, although its effect on LBD is more mixed, reflecting explanations that involve literals from more distinct decision levels and are less localized in the search. While the stronger propagation introduces additional computational overhead, this is often compensated by reduced search effort, leading to runtime improvements in many configurations, particularly in the satisfaction setting.

Overall, the findings demonstrate that strong bridge reasoning is an effective mechanism for strengthening propagation and reducing the explored search space.

Future work includes investigating more efficient algorithms for detecting strong bridges to further reduce computational overhead. In particular, linear-time algorithms for strong bridge detection have been proposed [16], and integrating such techniques could significantly improve scalability. It would also be valuable to evaluate the approach on a broader set of problem classes, including real-world instances, to assess general applicability. Another promising direction is the development of hybrid propagation strategies that combine multiple forms of graph-based reasoning, as well as exploring alternative explanation mechanisms to better balance nogood compactness and LBD. Finally, a more detailed statistical analysis using profiling techniques could provide deeper insight into the internal behavior of the propagator, enabling a finer-grained understanding of how strong bridge propagation interacts with the search process and influences performance.

References

[1] Joaquín Rodriguez. A constraint programming model for real-time train scheduling at junctions. *Transportation Research Part B: Methodological*, 41(2):231–245, 2007. ISSN 0191-2615. doi: https://doi.org/10.1016/j.trb.2006.02.006. URL `https://www.sciencedirect.com/science/article/pii/S0191261506000233`. Advanced Modelling of Train Operations in Stations and Networks.

[2] Paul Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In Michael Maher and Jean-Francois Puget, editors, *Principles and Practice of Constraint Programming — CP98*, pages 417–431, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg. doi: 10.1007/3-540-49481-2_30.

[3] Peter Van Beek and Xinguang Chen. Cplan: A constraint programming approach to planning. In *AAAI/IAAI*, pages 585–590, 1999.

[4] Alexander Schrijver. On the history of combinatorial optimization (till 1960). In K. Aardal, G.L. Nemhauser, and R. Weismantel, editors, *Discrete Optimization*, volume 12 of *Handbooks in Operations Research and Management Science*, pages 1–68. Elsevier, 2005. doi: 10.1016/S0927-0507(05)12001-5. URL `https://www.sciencedirect.com/science/article/pii/S0927050705120015`.

[5] Francesca Rossi, Peter van Beek, and Toby Walsh. Chapter 4 constraint programming. In Frank van Harmelen, Vladimir Lifschitz, and Bruce Porter, editors, *Handbook of Knowledge Representation*, volume 3 of *Foundations of Artificial Intelligence*, pages 181–211. Elsevier, 2008. doi: 10.1016/S1574-6526(07)03004-0. URL `https://www.sciencedirect.com/science/article/pii/S1574652607030040`.

[6] Olga Ohrimenko, Peter J. Stuckey, and Michael Codish. Propagation via lazy clause generation. *Constraints*, 14(3):357–391, 2009. doi: 10.1007/s10601-008-9064-x.

[7] Thibaut Feydy and Peter J. Stuckey. Lazy clause generation reengineered. In *Principles and Practice of Constraint Programming (CP 2009)*, volume 5732 of *Lecture Notes in Computer Science*, pages 352–366. Springer, 2009. doi: 10.1007/978-3-642-04244-7_29.

[8] Peter J. Stuckey. Lazy clause generation: Combining the power of sat and cp (and mip?) solving. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, volume 6140 of *Lecture Notes in Computer Science*, pages 5–9. Springer, 2010. doi: 10.1007/978-3-642-13520-0_3.

[9] Nicolas Beldiceanu and Evelyne Contejean. Introducing global constraints in CHIP. *Mathematical and Computer Modelling*, 20(12):97–123, 1994. doi: 10.1016/0895-7177(94)90127-9.

[10] Richard M. Karp. *Reducibility Among Combinatorial Problems*, pages 219–241. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. ISBN 978-3-540-68279-0. doi: 10.1007/978-3-540-68279-0_8. URL `https://doi.org/10.1007/978-3-540-68279-0_8`.

[11] Gregory Gutin and Abraham P. Punnen. *The Traveling Salesman Problem and Its Variations*. Combinatorial Optimization. Springer, 2006. doi: 10.1007/0-306-48213-4.

[12] Kathryn Glen Francis and Peter J. Stuckey. Explaining circuit propagation. *Constraints*, 19(1):1–29, 2014. doi: 10.1007/s10601-013-9148-0.

[13] Latife Genç Kaya and J. N. Hooker. A filter for the circuit constraint. In Frédéric Benhamou, editor, *Principles and Practice of Constraint Programming - CP 2006*, pages 706–710, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. ISBN 978-3-540-46268-2. doi: 10.1007/11889205_55.

[14] Yves Caseau and François Laburthe. Solving small tsps with constraints. In *Proceedings of the 14th International Conference on Logic Programming (ICLP)*, pages 316–330, 1997.

[15] Robert E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972. doi: 10.1137/0201010.

[16] Giuseppe F. Italiano, Luigi Laura, and Federico Santaroni. Finding strong bridges and strong articulation points in linear time. *Theoretical Computer Science*, 447: 74–84, 2012. ISSN 0304-3975. doi: 10.1016/j.tcs.2011.11.011. URL `https://www.sciencedirect.com/science/article/pii/S0304397511009303`. Combinational Algorithms and Applications (COCOA 2010).

[17] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern sat solvers. In *IJCAI*, volume 9, pages 399–404, 2009.

[18] Maarten Flippo, Konstantin Sidorov, Imko Marijnissen, Jeff Smits, and Emir Demirović. A Multi-Stage Proof Logging Framework to Certify the Correctness of CP Solvers. In Paul Shaw, editor, *30th International Conference on Principles and Practice of Constraint Programming (CP 2024)*, volume 307 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 11:1–11:20, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-336-2. doi: 10.4230/LIPIcs.CP.2024.11. URL `https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.CP.2024.11`.

Appendix

A Search Effort Details

In this section, we provide additional insight into the behavior of the strong bridge extension by analyzing both the number of conflicts and propagations and their contribution to the overall search effort.

Tables 1 and 2 report detailed reductions in conflicts and propagations. Across all configurations, reductions are substantial, confirming that strong bridge propagation significantly decreases the amount of search required. A notable but subtle pattern can be observed for $k = 3$, where conflict reductions are smaller compared to $k = 2$ and $k = 4$ in instances with small to moderate graph sizes, mirroring the behavior observed in runtime.

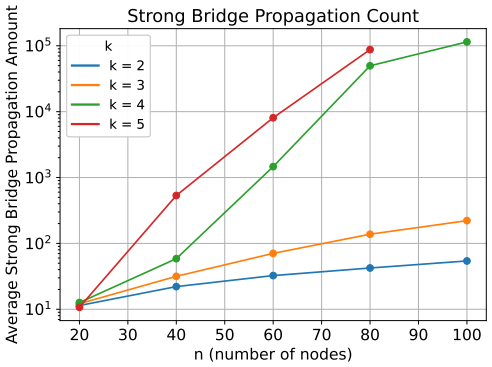
		SAT			OPT		
n	k	Base	SB	(%)	Base	SB	(%)
20	2	1.9	0	100.0	4.3	0.4	91.6
20	3	23.6	0.9	96.3	67.0	8.3	87.7
20	4	113.6	2.3	98.0	1031.6	135.0	86.9
20	5	956.8	3.6	99.6	-	-	-
40	2	3.1	0.0	99.7	5.9	0.5	91.3
40	3	69.2	2.8	96.0	261.4	33.4	87.2
40	4	2398.1	12.5	99.5	6.70×10^4	7017.3	89.5
40	5	2.92×10^5	252.4	99.9	-	-	-
60	2	3.9	0	100.0	7.3	0.4	94.1
60	3	223.4	8.2	96.3	624.9	106.8	82.9
60	4	1.48×10^5	511.8	99.7	2.09×10^6	1.95×10^5	90.6
60	5	7.81×10^5	4452.6	99.4	-	-	-
80	2	4.4	0.0	99.8	8.1	0.6	92.9
80	3	519.1	20.9	96.0	1969.7	317.9	83.9
80	4	2.48×10^5	1.34×10^4	94.6	-	-	-
80	5	4.09×10^5	4.93×10^4	88.0	-	-	-
100	2	5.8	0.0	99.8	-	-	-
100	3	1192.2	27.4	97.7	-	-	-
100	4	6.49×10^5	2.21×10^4	96.6	-	-	-

Table 1: Conflict reduction (SAT vs OPT).

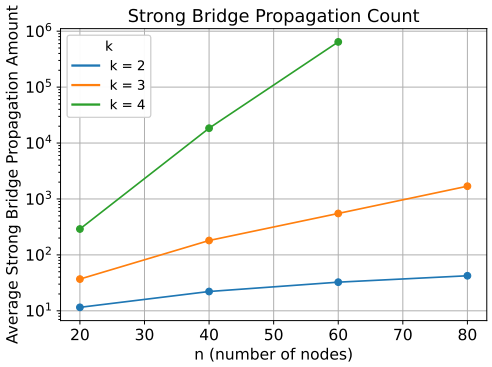
		SAT			OPT		
n	k	Base	SB	(%)	Base	SB	(%)
20	2	898.3	508.6	43.4	1912.2	613.0	67.9
20	3	4115.7	663.1	83.9	1.51×10^4	3090.3	79.6
20	4	1.53×10^4	797.6	94.8	1.89×10^5	3.08×10^4	83.7
20	5	1.11×10^5	877.8	99.2	-	-	-
40	2	4131.5	2089.3	49.4	7294.5	2292.8	68.6
40	3	3.27×10^4	3261.6	90.0	1.55×10^5	2.44×10^4	84.2
40	4	8.27×10^5	5817.5	99.3	3.09×10^7	3.07×10^6	90.1
40	5	7.48×10^7	5.02×10^4	99.9	-	-	-
60	2	9684.8	4759.3	50.9	1.71×10^4	5057.5	70.4
60	3	1.86×10^5	1.06×10^4	94.3	6.31×10^5	1.17×10^5	81.5
60	4	8.48×10^7	2.22×10^5	99.7	1.75×10^9	1.39×10^8	92.0
60	5	3.76×10^8	1.17×10^6	99.7	-	-	-
80	2	1.85×10^4	8547.0	53.8	3.04×10^4	9059.9	70.2
80	3	6.83×10^5	2.79×10^4	95.9	3.02×10^6	4.43×10^5	85.3
80	4	2.19×10^8	8.96×10^6	95.9	-	-	-
80	5	2.78×10^8	2.22×10^7	92.0	-	-	-
100	2	3.12×10^4	1.34×10^4	57.2	-	-	-
100	3	1.59×10^6	5.36×10^4	96.6	-	-	-
100	4	7.07×10^8	1.61×10^7	97.7	-	-	-

Table 2: Propagation reduction (SAT vs OPT).

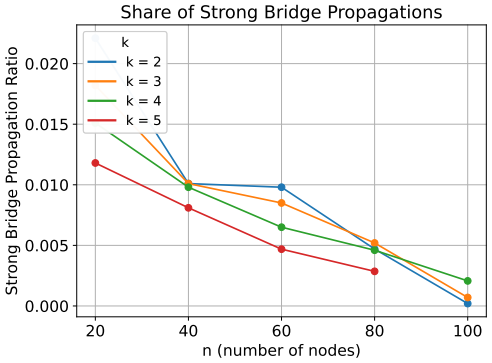
Figure 6a and Figure 6b show the total number of times strong bridge propagation is triggered as a function of n, which increase with instance size, reflecting the growing number of opportunities for applying the extension. However, when considering the relative contribution of these propagations, a different trend emerges: Figure 6c and Figure 6d also show the ratio of strong bridge propagations to the total number of propagations. This ratio decreases as n increases, indicating that although the extension becomes more active in absolute terms, it accounts for a smaller fraction of the overall propagation effort. Nevertheless, despite their relatively low frequency, strong bridge propagations have a disproportionate impact on search reduction, indicating that they contribute more effectively to pruning than their frequency alone would suggest.



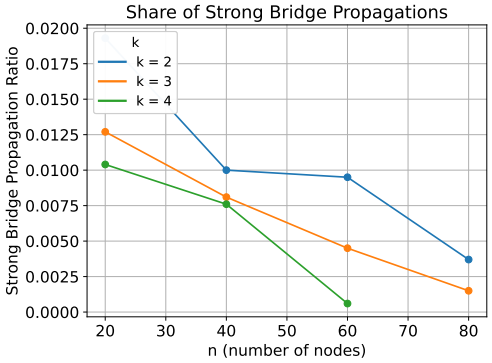
(a) SAT Strong Bridge Propagation Amount



(b) OPT Strong Bridge Propagation Amount



(c) SAT Strong Bridge Propagation Ratio



(d) OPT Strong Bridge Propagation Ratio

Figure 6: Visualization of absolute and relative amount of propagations regarding strong bridges, in satisfaction and optimisation setting. Ratio of strong bridge propagations goes down as problem size increases.

B Explanation Quality Details

Table 3 reports the relative frequency of strong bridge (SB) and SCC propagations as a percentage of the total number of propagations. Across all configurations, strong bridge propagations account for only a small fraction of propagations, typically below 2%, and this fraction decreases as instance size increases. SCC propagations occur even less frequently.

This indicates that the observed changes in explanation quality are not driven by the frequency of these propagations, but rather by their effect on the structure of the search process. Although strong bridge propagations are rare, their influence on subsequent propagation and conflict formation appears to be significant, leading to more globally distributed dependencies between assignments.

As a result, conflicts may involve literals from a wider range of decision levels, leading to increased LBD values, despite the overall reduction in search effort. At the same time, explanations remain relatively compact in terms of the number of literals, reflecting the structural nature of the reasoning used by the extension.

		SAT		OPT	
n	k	SB (%)	SCC (%)	SB (%)	SCC (%)
20	2	2.24	0.00	1.88	0.00
20	3	1.83	0.04	1.20	0.02
20	4	1.58	0.12	0.94	0.06
20	5	1.22	0.15	-	-
40	2	1.06	0.00	0.97	0.00
40	3	0.97	0.03	0.74	0.03
40	4	1.01	0.12	0.60	0.06
40	5	1.06	0.43	-	-
60	2	0.68	0.00	0.64	0.00
60	3	0.67	0.04	0.47	0.02
60	4	0.66	0.21	0.32	0.03
60	5	0.41	0.30	-	-
80	2	0.50	0.00	0.47	0.00
80	3	0.49	0.04	0.38	0.02
80	4	0.57	0.14	-	-
80	5	0.46	0.22	-	-
100	2	0.41	0.00	-	-
100	3	0.41	0.03	-	-
100	4	0.48	0.13	-	-

Table 3: Relative frequency of strong bridge (SB) and SCC propagations as a percentage of total propagations.

Table 4 shows the exact change in average Literal Block Distance values and average nogood length values for the different configurations. The results show a clear divergence between the two metrics: while nogood length is consistently reduced across all configurations, LBD reductions diminish as instance size and density increase, and can become negative for larger and denser graphs. This supports the hypothesis that strong bridge propagation leads to more compact explanations in terms of literal count, while simultaneously introducing more globally distributed dependencies that increase the number of decision levels involved.

		SAT		OPT	
n	k	LBD (%)	NG (%)	LBD (%)	NG (%)
20	2	100.0	100.0	79.0	82.3
20	3	73.3	78.1	28.8	37.4
20	4	41.4	48.8	13.6	15.9
20	5	30.0	43.7	-	-
40	2	99.2	99.2	72.4	80.5
40	3	44.1	51.7	15.4	16.5
40	4	28.9	45.2	13.9	15.3
40	5	14.2	40.6	-	-
60	2	100.0	100.0	76.3	86.4
60	3	33.5	45.3	11.0	1.3
60	4	7.5	39.1	14.2	17.7
60	5	-8.9	33.9	-	-
80	2	99.2	99.3	69.4	83.3
80	3	27.3	46.2	9.2	8.5
80	4	-8.0	39.0	-	-
80	5	-19.6	29.5	-	-
100	2	99.3	99.3	-	-
100	3	25.5	45.9	-	-
100	4	-12.2	41.4	-	-

Table 4: Relative change in LBD and nogood length (SAT vs OPT). Negative values indicate an increase in length.

C Runtime Analysis

Table 5 provides the exact runtime values underlying the trends discussed in the main text. Missing entries indicate configurations that were not evaluated in the optimization setting. Runtime values are measured externally by the solver.

Several configurations exhibit negative runtime improvements in the optimization setting. These correspond to cases where the reduction in search effort is relatively small, while the additional overhead of strong bridge propagation remains. As a result, the extension incurs extra computational cost without sufficient pruning benefit to compensate. Runtime improvements are consistently smaller for $k = 3$ than for $k = 2$ and $k = 4$, indicating that this regime is less favorable for the extension.

The extension generally leads to substantial runtime improvements. This demonstrates that increasing propagation strength can translate into faster solving when the additional computational cost is sufficiently offset by reduced search. At the same time, the variability observed across configurations highlights the trade-off between propagation strength and overhead, as the benefits depend on instance structure.

When computing average runtime, timeouts are excluded to avoid skewing results. As an alternative, penalized metrics (e.g., PAR2) could be used, but are not considered here to avoid introducing additional assumptions. Appendix D does show that the extension achieves strictly fewer timeouts than the baseline, which suggests that the relative performance advantage of the extension would likely be even more pronounced under penalized metrics such as PAR2.

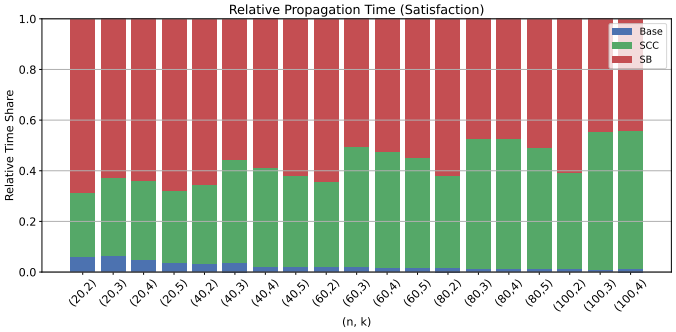
		SAT			OPT		
n	k	Base (s)	SB (s)	(%)	Base (s)	SB (s)	(%)
20	2	1.37e-04	5.60e-06	95.9	3.68e-04	4.04e-05	89.0
20	3	0.0010	2.50e-04	75.8	0.0038	0.0016	57.5
20	4	0.0046	8.30e-04	82.1	0.0613	0.0251	59.1
20	5	0.0569	0.0011	98.0	-	-	-
40	2	4.50e-04	9.99e-06	97.8	9.61e-04	7.71e-05	92.0
40	3	0.0062	0.0026	57.8	0.0308	0.0226	26.7
40	4	0.2331	0.0107	95.4	10.58	3.99	62.3
40	5	36.22	0.0856	99.8	-	-	-
60	2	7.72e-04	1.15e-05	98.5	0.0016	8.19e-05	94.8
60	3	0.0292	0.0165	43.5	0.1127	0.1545	-37.0
60	4	30.30	0.5100	98.3	551.44	222.58	59.6
60	5	172.67	3.14	98.2	-	-	-
80	2	0.0014	2.63e-05	98.1	0.0029	1.60e-04	94.5
80	3	0.1138	0.0567	50.1	0.6193	0.7861	-26.9
80	4	73.27	24.80	66.2	-	-	-
80	5	125.49	74.15	40.9	-	-	-
100	2	0.0023	4.02e-05	98.3	-	-	-
100	3	0.4900	0.1816	62.9	-	-	-
100	4	291.63	70.82	75.7	-	-	-

Table 5: Runtime comparison (SAT vs OPT).

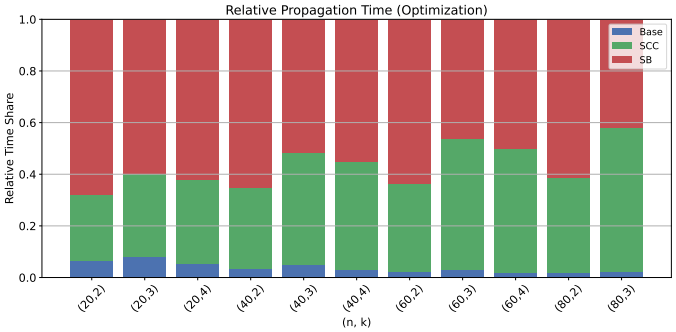
For a more detailed analysis of how computation is distributed within the solver, additional instrumentation is used to measure the time spent in different propagation components. Unlike total runtime, these measurements are collected using internal timers embedded within the propagator. As a result, they should be interpreted with care. In particular, timing of individual components (base cycle prevention, strong bridge propagation, and SCC conflict propagation) may be blown up due to measuring overhead. Consequently, the accumulated time attributed to individual components does not form a disjoint partition of total runtime, and values derived from these measurements do not necessarily sum to 100%. This effect is especially pronounced for very small instances, where absolute runtimes are extremely low and measurement overhead introduces additional noise.

Therefore, the timing breakdown should be interpreted comparatively rather than absolutely. While individual values may not reflect the exact share of total runtime, they remain informative for comparing the relative computational effort of different propagation components across configurations.

Figure 7 illustrates the relative contribution of different propagation components within the extension. The figure shows how time is distributed between base cycle prevention, strong bridge propagation, and SCC checks. As such, the plotted values should be understood as indicative of relative trends rather than precise proportions.



(a) SAT Time Shares per Propagation Type



(b) OPT Time Shares per Propagation Type

Figure 7: Visualization of relative time spent in different propagation types in the satisfaction and optimization setting. Values reflect relative computational effort within propagation and may not correspond to exact proportions of total runtime due to instrumentation overhead.

Beyond illustrating the distribution of effort within propagation, the figure also indicates that strong bridge and SCC propagation dominate the computational cost in the extension. Although these propagations occur less frequently for larger instances (see Appendix B), their relative time contribution increases, suggesting that individual propagation steps become significantly more expensive.

In contrast, the contribution of base cycle prevention decreases, indicating more modest scaling. This suggests that the cost of the extension is driven not by the number of propagations, but by their increasing complexity. Consequently, improving the efficiency of strong bridge and SCC reasoning could allow reductions in search effort to translate more directly into runtime gains.

Taken together with the reductions in conflicts and propagations observed in the main text, these results suggest that the extension shifts computational effort away from search and toward stronger propagation. While this shift cannot be directly inferred from the timing breakdown alone, it is consistent with the combined evidence from search effort metrics and runtime behavior.

D Timeout Analysis

Table 6 reports the number of timeouts observed for each configuration. Overall, the number of timeouts is low, indicating that the chosen instance sizes allow for reliable comparison of solver performance.

The extension achieves strictly fewer timeouts in all configurations where timeouts occur. This suggests that stronger propagation improves robustness by reducing the likelihood of excessively large search trees, allowing the solver to close branches earlier and avoid exploring regions that would otherwise lead to only conflicting assignments.

We note that the instance sizes were chosen to balance difficulty and tractability. Selecting significantly harder instances would likely increase the number of timeouts, but would also reduce the reliability of aggregated performance metrics such as runtime and search effort, as these would become dominated by incomplete runs.

		SAT		OPT	
n	k	Base	SB	Base	SB
20	2	0	0	0	0
20	3	0	0	0	0
20	4	0	0	0	0
20	5	0	0	-	-
40	2	0	0	0	0
40	3	0	0	0	0
40	4	0	0	0	0
40	5	0	0	-	-
60	2	0	0	0	0
60	3	0	0	0	0
60	4	0	0	11	0
60	5	13	4	-	-
80	2	0	0	0	0
80	3	0	0	0	0
80	4	7	0	-	-
80	5	35	9	-	-
100	2	0	0	-	-
100	3	0	0	-	-
100	4	22	13	-	-

Table 6: Timeout counts (SAT vs OPT).

E Minizinc Models

The MiniZinc models used for the both settings are shown in this appendix. Decision variables $x[i]$ represent the successor of node i in the circuit. Domain restrictions are imposed through the sets `allowed[i]`, ensuring that only valid edges are considered. The global `circuit` constraint enforces that the successor variables form a Hamiltonian cycle. A fixed search strategy using input-order variable selection and minimum value assignment is employed to ensure that differences between configurations are primarily attributable to propagation rather than search heuristics.

For the optimization model, edge weights are given by the matrix `dist`, and the objective minimizes the total cost of the circuit.

```
include "globals.mzn";

int: n;
array[1..n] of var 1..n: x;
array[1..n] of set of int: allowed;
array[1..n, 1..n] of int: dist;

constraint forall(i in 1..n)(
    x[i] in allowed[i]
);

constraint circuit(x);

solve :: int_search(x, input_order, indomain_min, complete) satisfy;
```

Model 1: MiniZinc model for the satisfaction setting

```
include "globals.mzn";

int: n;
array[1..n] of var 1..n: x;
array[1..n] of set of int: allowed;
array[1..n, 1..n] of int: dist;

constraint forall(i in 1..n)(
    x[i] in allowed[i]
);

constraint circuit(x);

var int: total_cost;
constraint total_cost = sum(i in 1..n)(dist[i, x[i]]);

solve :: int_search(x, input_order, indomain_min, complete)
      minimize total_cost;
```

Model 2: MiniZinc model for the optimization setting