Articulation-Based Propagation for the Circuit Constraint

Implementation and Evaluation in the Pumpkin LCG Solver

Ágoston Szabó[1]

Supervisor(s): Emir Demirović [1]**, Imko Marijnissen**[1]

[1]**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: Ágoston Szabó
Final project course: CSE3000 Research Project
Thesis committee: Emir Demirović, Imko Marijnissen, Andreea Costea

Abstract

Constraint Programming (CP) solvers rely on propagation algorithms to reduce the search space of combinatorial problems. For graph constraints such as the `circuit` constraint, infeasibility often depends on global connectivity properties that are difficult to capture using local propagation alone. In this paper, we investigate articulation-based graph reasoning for the `circuit` constraint in a Lazy Clause Generation (LCG) solver. We introduce propagation techniques based on articulation points and strong articulation points, using connectivity bottlenecks and strongly connected component structure to detect infeasible states and derive additional pruning. We further describe explanation-generation methods that allow these propagations to participate in clause learning within the LCG framework. To evaluate the practical usefulness of this reasoning, we implement the proposed propagators in an LCG solver and compare them against a baseline cycle-prevention propagator on graph instances with varying sizes and densities. The experiments show that articulation-based reasoning substantially reduces search effort and often improves runtime performance on sparse and moderately dense benchmark instances. These results demonstrate that global connectivity reasoning can provide practical benefits beyond traditional cycle-prevention propagation in modern LCG solvers.

1 Introduction

Constraint Programming (CP) is a powerful paradigm for solving NP-hard combinatorial problems by combining search and propagation. Propagation reduces the search space by removing values that cannot participate in any feasible solution, allowing many infeasible partial assignments to be eliminated before they are explored. Although solver performance depends on both search and propagation, stronger propagation can substantially reduce the amount of search required to solve a problem. Consequently, the design of efficient propagators remains a central topic in Constraint Programming research [9; 10].

An important topic in CP is the design of global constraints and their associated propagators. These constraints play a crucial role in modern CP solvers. They capture recurring combinatorial substructures, allowing solvers to reason more effectively than if the same logic were expressed through many small primitive constraints. For this reason, the study of advanced propagators is important both theoretically and practically [12].

One important global constraint is the `circuit` constraint, which requires a set of successor variables to form a Hamiltonian cycle. The constraint appears naturally in routing, scheduling, and sequencing applications [9], including variants of the Traveling Salesperson Problem [5]. Propagation for the circuit constraint is challenging because feasibility depends on global graph structure rather than only local variable relationships. While existing propagators can prevent premature cycles and reason about strongly connected components, many infeasible structures remain undetected until relatively late in the search process.

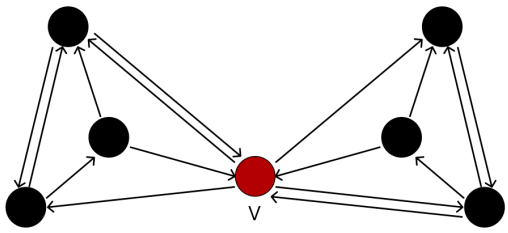

Figure 1: Example of a connectivity bottleneck. Removing vertex V separates the graph into multiple components. Any Hamiltonian cycle would need to visit vertices on both sides of V while entering and leaving each region exactly once. In this case, we can see this is impossible as the cycle would need to go through V twice.

Consider the support graph shown in Figure 1. Vertex V is a connectivity bottleneck: removing it separates the graph into multiple regions. Any Hamiltonian cycle would need to visit the vertices on both sides of the bottleneck while entering and leaving each region exactly once. Since V is an articulation point, this is impossible, immediately proving that no feasible circuit exists. More generally, connectivity bottlenecks can also provide useful propagation even when they do not immediately imply infeasibility. This reasoning is not captured by traditional cycle-prevention propagation and motivates the use of articulation-based graph analysis within the circuit constraint.

Although graph connectivity has been extensively studied in both graph theory and Constraint Programming, the practical usefulness of articulation-based reasoning for the circuit constraint remains largely unexplored. In particular, little is known about whether articulation points and strong articulation points can provide useful propagation in a modern LCG solver, how explanations for such reasoning can be generated, and whether the resulting reduction in search effort justifies the additional graph-analysis overhead.

This paper investigates articulation-based graph reasoning for the circuit constraint. We introduce propagation techniques based on both articulation points and strong articulation points. The proposed propagators exploit connectivity bottlenecks and strongly connected component structure to detect infeasible states and derive additional pruning. We further develop explanation-generation techniques that allow these propagations to participate in clause learning within the LCG framework.

Experimental results show that articulation-based reasoning can substantially reduce the explored search space by identifying structural infeasibilities earlier during search. Across the evaluated benchmark classes, the proposed propagators often reduce the number of search nodes and failures by one or more orders of magnitude. These reductions also translate into improved runtime performance on many medium-sized and large instances, while significantly reducing the number of solver timeouts. The results suggest that articulation points and strong articulation points provide a

practical source of additional propagation for the `circuit` constraint and demonstrate the value of incorporating global connectivity reasoning into modern Lazy Clause Generation solvers.

The remainder of this paper is organised as follows. Section 2 discusses previous work on this topic. Section 3 introduces the preliminaries required for this paper. Section 4 presents the details of the propagator and implementation. Section 5 evaluates the propagator experimentally. Section 6 discusses responsible research. Section 7 concludes the paper and discusses future work.

2 Related Work

The `circuit` constraint has been widely studied within Constraint Programming (CP) due to its importance in routing problems such as the Traveling Salesperson Problem and in sequencing applications [7; 8; 4; 5]. Early approaches focused primarily on cycle prevention, removing assignments that would create subtours before all vertices could be included in a single Hamiltonian cycle. While computationally inexpensive, these approaches reason only about local cycle structure and do not exploit more global connectivity information.

Francis and Stuckey [4] extended propagation for the `circuit` constraint to the Lazy Clause Generation (LCG) framework [2]. Their work introduced explanation-generating versions of both cycle-prevention and strongly-connected-component (SCC) propagation, allowing graph-based reasoning to participate in clause learning. SCC propagation reasons about global connectivity by ensuring that the support graph remains strongly connected. However, it does not exploit how the connectivity structure changes after the removal of individual vertices. In contrast, this paper investigates reasoning based on articulation points and strong articulation points, which capture connectivity bottlenecks that may reveal infeasibility or imply additional propagation.

Graph connectivity reasoning has also been studied extensively outside CP. Articulation points and strong articulation points can be identified efficiently using depth-first-search-based algorithms originating from Tarjan's work [11]. These algorithms form the basis of many graph-analysis techniques used in optimisation and network analysis [1]. In this work, these graph-theoretic concepts are incorporated into propagation for the `circuit` constraint.

Isoart and Régin [6] demonstrated the effectiveness of separator-based reasoning through the k-cutset constraint, showing how graph cutsets can be used to derive strong propagation. The articulation-point reasoning investigated in this paper is related in spirit, as an articulation point corresponds to a 1-vertex cut. However, unlike the k-cutset constraint, which introduces a dedicated connectivity constraint, our work integrates separator-based reasoning directly into propagation for the `circuit` constraint.

Explanation generation is another important aspect of graph-based propagation in LCG solvers. Francis and Stuckey [4] showed how explanations can be generated for connectivity-based propagation within the `circuit` constraint. We build on this line of work by developing explanations for articulation-point-based and strong-articulation-point-based reasoning.

Existing work has demonstrated the value of both connectivity-based propagation for the `circuit` constraint and separator-based reasoning in graph constraints more generally. However, the use of articulation points and strong articulation points as propagation mechanisms for the `circuit` constraint has received little attention, particularly in the context of Lazy Clause Generation. This paper investigates whether articulation-based graph reasoning can provide useful propagation for the `circuit` constraint and how such reasoning can be integrated into an LCG solver through efficient explanations.

3 Preliminaries

3.1 Constraint Programming

A Constraint Satisfaction Problem (CSP) consists of a set of variables, $X = X_1, X_2, \ldots, X_n$, where each variable X_i is associated with a finite domain $D(X_i)$ of possible values, together with constraints restricting the combinations of assignments that may appear in a valid solution.

Constraint Programming (CP) solvers combine propagation and search. Propagation removes values that cannot participate in any feasible solution, thereby reducing the search space. Formally, propagation replaces a domain $D(X_i)$ with a smaller domain $D'(X_i) \subseteq D(X_i)$, while preserving all solutions of the CSP. In the context of the `circuit` constraint, removing a value $v \in D(X_i)$ corresponds to removing the possibility that vertex i selects vertex v as its successor.

When propagation determines that only a single value remains in a domain, the corresponding variable is assigned. For a variable X_i, this can be written as $D(X_i) = \{v\} \implies X_i = v$.

Search is used when propagation alone cannot determine a solution. The solver selects a variable, chooses a value, and recursively explores the resulting subproblems.

A typical approach in modern CP solving is Lazy Clause Generation (LCG), which combines CP propagation with Boolean Satisfiability (SAT)-style clause learning [2]. In LCG, every propagation or conflict is associated with an explanation, which is a logical justification describing why the inference is valid. These explanations are used during conflict analysis to learn clauses that prevent the same reasoning from being repeated later during search.

3.2 Circuit Constraint

The `circuit` constraint is defined over a set of successor variables $X = X_1, X_2, \ldots, X_n$, where each variable X_i represents the successor of vertex i. The domain $D(X_i)$ contains all vertices that may be selected as successors of i. Consequently, each value $j \in D(X_i)$ corresponds to a directed edge (i, j) in the underlying graph, which is called the support graph.

The `circuit` constraint models Hamiltonian cycles over this support graph. A solution corresponds to a selection of one successor for every vertex such that the chosen edges form a single cycle that visits every vertex exactly once.

Formally, the `circuit` constraint requires that:

1. Every vertex has exactly one successor.
2. Every vertex has exactly one predecessor.
3. The selected edges form a single Hamiltonian cycle.

A common propagation technique is cycle prevention. If assigning a value $X_i = j$ would close a cycle before all vertices are included, j can be removed from the domain $D(X_i)$. This propagation is sound because a small cycle would make the existence of a Hamiltonian cycle impossible.

3.3 Cycle-Prevention Propagation

The baseline propagator prevents premature cycles. During the search, some successor variables may already be fixed. These fixed assignments form directed paths and possibly cycles. If assigning a specific value $v \in D(X_i)$ would close a cycle that does not contain all vertices, then that value cannot be part of a feasible solution and is removed. So the domain becomes $D'(X_i) = D(X_i) \setminus \{v\}$.

This propagation is sound because the `circuit` constraint requires one Hamiltonian cycle containing all vertices. If a smaller cycle is formed, it is no longer possible to connect the rest of the nodes to the ones in this premature cycle, as they all have a successor already.

The propagator loops over all fixed paths in the support graph and removes all the edges that would close any of these paths.

However, cycle prevention is limited because it only detects infeasibility once a smaller cycle is about to be closed. It does not reason about whether the remaining support graph is structurally capable of containing a Hamiltonian cycle.

3.4 Articulation and Strong Articulation Points

An articulation point is a vertex whose removal increases the number of connected components in the graph. A similar concept in a directed graph is a strong articulation point, whose removal increases the number of strongly connected components (SCCs) in the graph. Such vertices identify connectivity bottlenecks in the graph, since all paths between certain regions must pass through them.

Figure 2 illustrates a simple example. In the left graph, the red vertex is an articulation point because removing it separates the graph into two disconnected components. In the right graph, the red vertex is a strong articulation point, as its removal increases the number of SCCs from one to two.

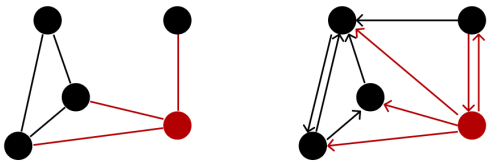

Figure 2: The red vertices are examples of articulation points (left) and strong articulation points (right). On the left graph, removing the red vertex disconnects the graph into two connected components. On the right graph, removing the red vertex increases the number of SCCs in the graph.

3.5 Explanations in Lazy Clause Generation

Lazy Clause Generation (LCG) combines Constraint Programming propagation with SAT-style clause learning [2]. In an LCG solver, every propagation and conflict must be accompanied by an explanation that justifies the inference.

An explanation is a conjunction of predicates that logically implies a propagation or conflict. For example, if a propagator removes a value from a domain, the explanation identifies the conditions under which that removal is valid. Similarly, when a propagator detects a conflict, the explanation describes why the current domains cannot contain a feasible solution.

During conflict analysis, explanations are used to derive learned clauses. These clauses allow the solver to avoid revisiting similar infeasible states later in the search. Consequently, explanation quality can have a significant impact on solver performance. Smaller explanations are generally preferred because they can lead to shorter learned clauses and more effective learning.

The basic cycle-prevention propagator generates the explanations as described in the literature by Francis and Stuckey [4].

4 Articulation-Based Propagator

This section presents the proposed articulation-based propagators for the `circuit` constraint described in Section 3.2. Let $G = (V, E)$ denote the current support graph induced by the domains of the successor variables. The propagators strengthen the baseline cycle-prevention propagator explained in Section 3.3 by exploiting connectivity bottlenecks in G. Two forms of reasoning are considered. The first uses articulation points in an undirected representation of the support graph. The second reasons directly about strong articulation points and the strongly connected component structure of the directed support graph.

4.1 Undirected Articulation Reasoning

The first propagation rule reasons about articulation points in an undirected version of the support graph. Every directed edge in the support graph is treated as an undirected edge, producing an undirected graph G_u.

The key observation is that a Hamiltonian cycle cannot exist if G_u is disconnected or contains an articulation point. If an articulation point separates the graph into multiple regions, any Hamiltonian cycle would need to traverse the articulation point multiple times in order to visit all vertices. Since every vertex in a Hamiltonian cycle has exactly one predecessor and one successor, this is impossible.

Conflict Detection

The propagator first checks whether the undirected support graph satisfies this necessary connectivity condition.

The procedure is as follows:

1. Construct the undirected graph G_u from the current support graph.
2. Check whether G_u is connected.
3. Identify articulation points in G_u.

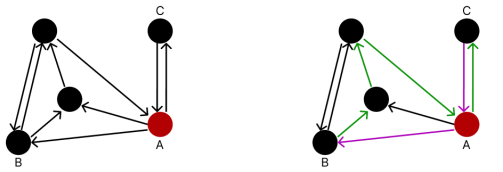

Figure 3: Example of articulation-based conflict detection. Vertex A is an articulation point, so both paths from B to C (green path as an example) and C to B (purple path as an example) need to go through A. However, this breaks the `circuit` constraint as vertex A can only have one successor and only one vertex can have A as its successor.

4. If G_u is disconnected or contains any articulation points, report a conflict.

Figure 3 shows an example where vertex A is an articulation point, whose removal separates the graph into two components. Vertices B and C are in different components. To find a Hamiltonian cycle, we need a path from B to C and a path from C to B. Since vertex A separates them, both of these paths have to go through A. But this breaks the constraint, as our cycle would go through vertex A twice.

The purpose of this check is not to construct a Hamiltonian cycle. Instead, it verifies a necessary structural condition for the existence of one. If the condition fails, the current domains cannot contain any feasible solution.

Articulation-Based Forcing

The same structural condition can also be used to derive forced assignments.

Consider a candidate edge (u, v) that has not yet been fixed. If removing (u, v) causes the resulting support graph to violate the articulation condition described above, then no feasible Hamiltonian cycle can exist without that edge. Consequently, every feasible solution must contain (u, v) and the assignment $X_u = v$ can be enforced.

The propagator therefore follows the forcing procedure, which works as follows:

1. Loop over all of the edges in the support graph that are not yet fixed.
2. For each edge, pretend that we remove it and build the remaining support graph.
3. Check if this graph has any articulation points.
4. If it has any, then we know it is impossible to find a Hamiltonian cycle without this edge, so we need to enforce it.

Unlike cycle-prevention propagation, which reacts only when an invalid cycle is about to be created, articulation-based forcing derives mandatory assignments from global connectivity structure before any explicit infeasibility is observed.

4.2 Strong Articulation Point Reasoning

Undirected articulation reasoning ignores edge direction and therefore captures only a subset of the connectivity information present in the support graph. To exploit directed connectivity more effectively, we additionally reason about strong articulation points introduced in Section 3.4.

Let v be a strong articulation point of the support graph. Removing v produces a graph whose strongly connected components form a condensation graph, which is a directed acyclic graph (DAG). The structure of this DAG provides information about how a Hamiltonian cycle would need to traverse the remaining vertices.

Conflict Detection

For every strong articulation point v, the propagator removes v and computes the SCC decomposition of the remaining graph. Let D_v denote the resulting condensation DAG.

Removing a vertex from a Hamiltonian cycle yields a Hamiltonian path on the remaining $|V| - 1$ vertices. Consequently, a necessary condition for the existence of a Hamiltonian cycle is that the SCC structure induced by the removal of v admits a traversal through all remaining vertices. The propagator therefore performs three checks. If any of them fails, the propagator returns conflict.

First, D_v must contain exactly one source SCC. Every source SCC can only be entered through the strong articulation point v, as it has no incoming edges from other SCCs in the condensation DAG. Since v can have only one successor in a Hamiltonian cycle, the cycle can enter at most one source SCC. Therefore, if multiple source SCCs exist, it is impossible to visit all SCCs and a conflict is reported.

Second, D_v must contain exactly one sink SCC. Every sink SCC can only be exited through the strong articulation point v, as it has no outgoing edges to other SCCs in the condensation DAG. Since v can have only one predecessor in a Hamiltonian cycle, the cycle can leave at most one sink SCC. Therefore, if multiple sink SCCs exist, it is impossible to visit all SCCs and a conflict is reported.

Finally, now that we know we only have one source and one sink SCC, the propagator can easily compute the longest weighted path in the condensation DAG, where the weight of each SCC corresponds to the number of vertices it contains. If the longest path contains fewer than $|V| - 1$ vertices, then not all remaining vertices can be traversed in a single directed walk through the SCC structure. In this case, a Hamiltonian cycle is impossible and a conflict is reported.

Together, these conditions provide a stronger directed connectivity test than the undirected articulation analysis.

Edge Pruning

The SCC structure can also be used to derive additional pruning.

Suppose that removing a strong articulation point v produces a condensation DAG that passes the conflict detection described earlier in Section 4.2. This means it has a unique source SCC S, a unique sink SCC T and a longest weighted path that contains $|V| - 1$ vertices. The only way to have a Hamiltonian cycle in the support graph is to use one of these longest traversals and extend it with v. For this to happen, the successor of v needs to be the start of this traversal, and v has to be the successor of the end of it. There can be multiple longest traversals, so we cannot know exactly which node is the start and which is the end. But we know that the start must belong to S and the end must belong to T.

Therefore, the propagator removes:

- all edges (v, u) where u does not belong to the source SCC S,
- all edges (u, v) where u does not belong to the sink SCC T.

These edges are infeasible because selecting them would force the Hamiltonian cycle to enter or leave the SCC structure at an incorrect location, making it impossible to traverse all SCCs in a consistent order.

This pruning exploits information that is not visible in the undirected graph and therefore complements the articulation-point reasoning described in Section 4.1.

4.3 Explanation Generation

The articulation-based propagator uses a uniform explanation scheme for conflicts, pruning, and edge forcing. Rather than constructing a specialized explanation for each individual inference, it records the domain removals that define the graph on which the connectivity analysis is performed. Concretely, for every edge that is absent from the support graph, the explanation contains the corresponding domain-removal predicate.

These predicates allow Pumpkin's explanation checker to reconstruct the graph used by the propagator. The checker can then independently verify the graph-theoretic property that justifies the inference. For conflict detection, this may correspond to a disconnected support graph, the presence of an articulation point, or a violation of one of the strong articulation point conditions. For pruning, the checker verifies that the SCC structure implied by a strong articulation point justifies the removal of the reported edge.

Edge forcing is handled in a similar way. To determine whether an edge (u,v) must be present in every feasible solution, the propagator temporarily removes the edge and analyzes the resulting graph. If the reduced graph contains an articulation point, every Hamiltonian cycle must contain (u,v). The explanation records the domain removals that define this reduced graph, allowing the checker to reconstruct it and verify the inference.

This explanation scheme is straightforward to generate and verify, although it is not guaranteed to produce minimal explanations. Investigating explanation minimization techniques for articulation-based propagation is left for future work.

5 Experimental Setup and Results

The experimental results show that the articulation-based reasoning substantially strengthens the propagator. Across all benchmark classes, it consistently reduced the number of explored search nodes and solver failures, often by several orders of magnitude. Furthermore, the number of timeouts decreased from 57 to 13, demonstrating that the stronger propagation remains effective even on the most difficult instances.

The goal of this section is to evaluate the practical impact of articulation-based connectivity reasoning on the `circuit` constraint. More specifically, the experiments investigate whether articulation points and strong articulation points provide useful additional propagation that reduces the size of the search space. Since stronger propagation introduces additional graph-analysis overhead, the evaluation also examines its impact on runtime and overall solver performance.

The evaluation compares the baseline cycle-prevention propagator (CP) against the extended version of it (SAP), which includes both articulation and strong articulation-based reasoning, described in Section 4.

5.1 Experimental Questions

The experimental evaluation is designed to answer the following questions:

1. How much do articulation-point and strong-articulation-point propagation reduce the size of the search space compared to the baseline cycle-prevention propagator?
2. What is the impact of the additional graph analyses on runtime performance?
3. How does graph size and density affect the effectiveness of the propagator?

To answer these questions, the propagator configurations are compared using runtime, number of search nodes, failures, and propagation statistics.

5.2 Experimental Setup

All experiments were performed using the Pumpkin [3] Lazy Clause Generation solver extended with the articulation-based propagator described in Section 4. The propagator was implemented in Rust and integrated directly into the existing `circuit` constraint infrastructure.

The evaluation compares two solver configurations:

1. **Baseline (CP):** the standard cycle-prevention propagator introduced in Section 3.3.
2. **SAP:** the full articulation-based propagator described in Section 4, including articulation-point conflict detection, edge forcing, strong-articulation-point conflict detection, and SCC-based edge pruning.

All configurations select variables in their declaration order and assign the smallest value first. This is to ensure that all of the differences in performance come from the propagation itself, rather than from changes in search behaviour.

The benchmark instances were generated using the graph generator proposed by Francis and Stuckey [4]. Following their methodology, the instances were generated in the following way, where n denotes the number of nodes in the graph and k denotes the density.

- Generate n points in 2D where both coordinates of the points are in the $[0, 1)$ range.
- From every node, create a directed edge to its k nearest neighbours.
- Ensure that there exists a Hamiltonian cycle via a random walk over the edges. If the walk gets stuck in node v, add a new random edge from v to a node that has not been visited yet.

For each parameter configuration, 20 random seeds were used to reduce the influence of individual graph structures. After a small initial batch of experiments with a standard 30-minute timeout, we decided to reduce this to 15 because less than 1% of solvetimes fell between 15 and 30 minutes. This way, we can ensure that difficult instances can still be solved while keeping the total experimental runtime manageable. The presented results are the arithmetic means of these random seeds for every configuration, excluding the instances for which at least one of the propagators has timed out. This ensures that averages are computed over identical benchmark sets for both propagators. Each instance was solved independently using both evaluated propagators.

The following performance metrics were recorded:

- Number of explored search nodes,
- Number of solver failures,
- Total runtime,
- Number of propagation operations.

The experiments were executed on a 2.4GHz 13th Gen Intel(R) Core(TM) i7-13700H CPU with 16GB of RAM on Windows 11 using the release-mode builds of Pumpkin [3].

5.3 Results and Discussion

Across all 400 instances, the number of timeouts decreased from 57 to 13, corresponding to a 77% reduction. Looking at only the 339 instances for which none of the solvers timed out, the SAP propagator reduced the total number of explored search nodes by a factor of approximately 560 and reduced total runtime by a factor of approximately 7.7, though these aggregate figures are dominated by the larger, sparser instances where improvements are most dramatic.

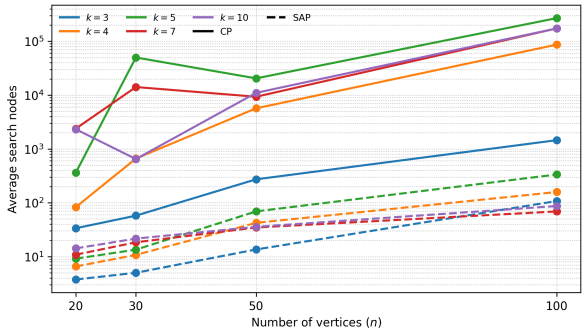

Figure 4: Average number of explored search nodes on a log scale. The SAP propagator consistently explores substantially smaller search trees than the baseline cycle-prevention propagator.

Figures 4, 5 and 6 compare the baseline cycle-prevention propagator (CP) with the full articulation-based propagator (SAP) across benchmark instances of varying size and density on the metrics described in Section 5.2. Table 1 reports the arithmetic means of strong-articulation-point analyses, pruning operations, and conflicts generated by the SAP propagator.

The results show that the SAP propagator consistently reduces the size of the search space. Figures 4 and 5 show that

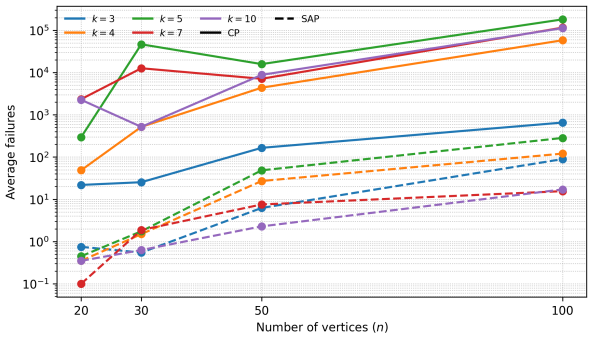

Figure 5: Average number of solver failures on a log scale. The SAP reasoning reduces the number of backtracks across all benchmark classes.

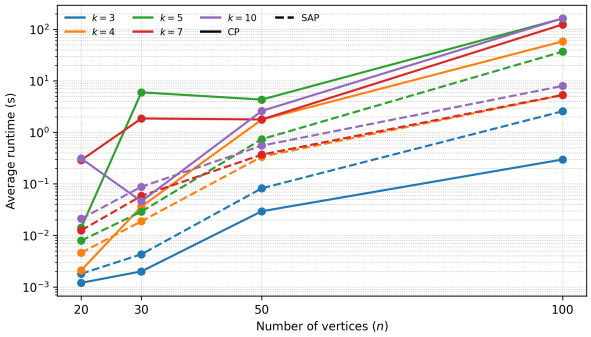

Figure 6: Average runtime on a log scale. Despite the additional graph analysis, the SAP propagator improves runtime on most medium-sized and large instances.

across all benchmark classes, the number of explored search nodes and solver failures decreases by one to several orders of magnitude. The most dramatic improvements occur on the larger instances. For example, for the $(n = 100, k = 5)$ benchmark class, the average number of explored nodes decreases from approximately $269K$ to only 335, while the number of failures decreases from approximately $181K$ to 283. Similar reductions can be observed for the $(n = 30, k = 5)$ and $(n = 50, k = 5)$ benchmark classes, where the search tree becomes hundreds or even thousands of times smaller.

A similar trend can be observed for propagation activity. Despite performing additional graph analyses, the SAP propagator generally executes substantially fewer propagation operations than the baseline configuration. This reduction is a direct consequence of the smaller search trees produced by the stronger propagation. For example, on the $(n = 100, k = 5)$ benchmark class, the average number of propagations decreases from approximately $172M$ to $97K$, while on the $(n = 50, k = 5)$ benchmark class it decreases from approximately $7.1M$ to $15.8K$.

These reductions demonstrate that the articulation-based reasoning successfully identifies infeasible regions of the search space much earlier than cycle prevention alone. The propagator exploits global connectivity information to detect structural bottlenecks, derive additional pruning, and force

edges that must belong to every feasible Hamiltonian cycle. As a result, many partial assignments that would otherwise require extensive search are eliminated during propagation.

Table 1: SAP activity statistics averaged over included runs.

n	k	SAP checks	Prunings	Conflicts
20	3	126.60	31.65	0.20
20	4	158.35	27.85	0.10
20	5	202.40	24.45	0.30
20	7	195.75	21.30	0.05
20	10	228.15	13.45	0.15
30	3	243.90	45.70	0.15
30	4	389.80	50.45	0.55
30	5	431.80	48.95	0.95
30	7	539.29	47.35	1.59
30	10	515.74	21.05	0.16
50	3	1132.35	134.55	2.00
50	4	3460.89	271.58	14.00
50	5	5395.63	295.79	39.21
50	7	1808.31	90.81	2.44
50	10	1480.53	52.00	1.12
100	3	19305.45	662.40	16.65
100	4	30421.67	1070.58	71.67
100	5	57253.38	1150.88	240.88
100	7	7121.80	451.20	5.00
100	10	8295.71	92.43	14.86

The SAP activity statistics, shown in Table 1, provide further evidence for this explanation. Instances exhibiting the largest reductions in search effort also generate the largest numbers of strong-articulation-point analyses, pruning operations, and conflicts. For example, the $(n = 100, k = 5)$ benchmark class performs more than $57K$ SAP analyses on average, resulting in over $1.1K$ pruning operations and approximately 241 detected conflicts. This suggests that the observed reductions in search are directly attributable to the additional graph-based reasoning introduced by the propagator.

The stronger propagation often translates into improved runtime performance, as presented by Figure 6. Although the SAP propagator performs substantially more graph analysis than the baseline propagator, the reduction in search effort is large enough to compensate for this overhead. For example, on the $(n = 100, k = 5)$ benchmark class, the average runtime decreases from approximately 162 seconds to 37 seconds. Similar improvements can be observed on most medium-sized and large benchmark classes.

Another notable result is the reduction in timeouts. Across all benchmark instances, the baseline propagator timed out 57 times, whereas the SAP propagator timed out only 13 times. This indicates that the additional propagation remains effective even on the most challenging instances and allows the solver to solve many instances that would otherwise exceed the time limit.

The results also reveal an interesting interaction between graph structure and propagation strength. The largest improvements are observed for sparse and moderately dense graphs, where articulation points and strong articulation points occur frequently. In such instances, the support graph contains connectivity bottlenecks that can be exploited by the propagator. As graph density increases, these bottlenecks become less common, reducing the opportunities for additional propagation. Nevertheless, even on the denser benchmark classes, the SAP propagator generally maintains a significant advantage over the baseline configuration.

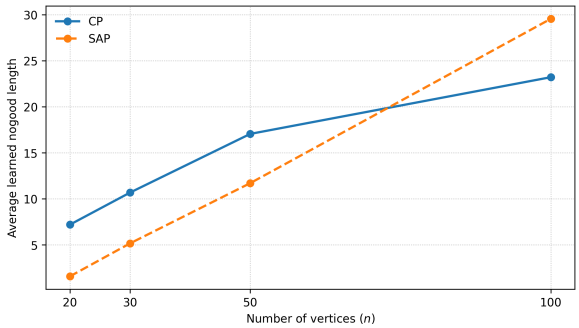

Figure 7: Average learned nogood length as a function of graph size. While SAP explanations become moderately larger for the largest instances, the increase remains limited relative to the reduction in search effort.

An additional aspect that was briefly examined is the effect of the propagator on learned clause characteristics. Figure 7 indicates that explanation size increases with graph size for both propagators. While the SAP propagator produces shorter learned nogoods on smaller benchmark classes, its learned nogoods become moderately larger than those of the cycle-prevention propagator for $n = 100$. This trend is expected, as articulation-based reasoning relies on increasingly large graph structures and therefore requires more information to justify individual inferences.

The primary objective of this work was to investigate stronger propagation for the `circuit` constraint rather than the generation of minimal explanations. Consequently, only a high-level analysis of learned nogood lengths was performed. A more detailed study of explanation quality could provide valuable insights into the interaction between articulation-based reasoning and Lazy Clause Generation. Future work could investigate metrics such as Literal Block Distance (LBD), conflict size, explanation minimality, and the long-term impact of learned clauses on solver performance.

Overall, the experimental evaluation demonstrates that articulation-based reasoning is an effective technique for strengthening the `circuit` constraint. The propagator substantially reduces the size of the search space, decreases the number of solver failures, improves runtime performance on most benchmark classes, and significantly reduces the number of timeouts. These results suggest that global connectivity reasoning can provide practical benefits beyond traditional cycle-prevention propagation in modern Lazy Clause Generation solvers.

6 Responsible Research

The main responsible research concern in this work is solver correctness. The proposed propagator performs global graph-based reasoning inside a Lazy Clause Generation framework,

meaning that every propagation and conflict must be accompanied by a sound explanation. Incorrect explanations could cause the solver to learn invalid clauses and incorrectly remove feasible solutions from the search space.

To reduce this risk, all propagations and conflicts generated by the propagator were validated using Pumpkin's internal explanation checker. This checker verifies that the generated explanation logically implies the corresponding propagation or conflict under the current solver state. During development, this validation proved especially important because articulation-based reasoning depends on complex graph structure rather than only local variable interactions.

The benchmark instances were generated using a graph generator previously used in the literature. Nevertheless, generated instances may not fully represent real-world applications of the `circuit` constraint. Consequently, the conclusions of this study should be interpreted with respect to the evaluated benchmark family.

Reproducibility is another important consideration. Solver performance can depend heavily on implementation details, benchmark generation, random seeds, and hardware configuration. To improve reproducibility, all propagator configurations were evaluated using the same hardware, software, search strategy, solver version, and benchmark generation process. The benchmark instances were generated using multiple random seeds, and the reported results are arithmetic means over these runs.

Reproducibility is supported by the publicly available implementation, benchmark generator, experimental scripts and the random seeds, along with the corresponding instances used in the evaluation.

The experiments also aim to report results transparently. Although the articulation-based propagator performs additional graph analysis and therefore incurs extra computational overhead, this cost is evaluated alongside its benefits in search reduction. For this reason, the evaluation reports multiple performance metrics, including runtime, search nodes, failures, propagation counts, learned nogood lengths, and timeout counts, rather than focusing only on favourable outcomes. This provides a more complete picture of the trade-offs associated with stronger propagation.

Generative AI tools were used during the writing process of this paper. In particular, they were used to rephrase sentences, improve readability, and correct spelling and grammar mistakes. They were also used to assist with formatting LaTeX.

7 Conclusion and Future Work

This paper presented an articulation-point and strong-articulation-point-based propagator for the `circuit` constraint implemented in a Lazy Clause Generation solver. The propagator extends the standard cycle-prevention approach by reasoning about global graph connectivity properties of the support graph. In particular, articulation points are used for conflict detection and edge forcing, while strong articulation points are used for additional conflict detection and SCC-based edge pruning.

The experimental evaluation showed that the propagator can significantly reduce the explored search space across all benchmark classes. Compared to the baseline cycle-prevention propagator, the SAP propagator consistently reduced the number of explored search nodes, failures, and propagation events, often by several orders of magnitude. These reductions also translated into improved runtime performance on most medium-sized and large benchmark classes, while significantly reducing the number of solver timeouts, from 57 to only 13. These results demonstrate that global graph structure can provide propagation that is not visible to purely local reasoning.

Although the SAP propagator performs substantially more graph analysis than the baseline propagator, the resulting reduction in search effort was generally large enough to compensate for the additional overhead. The strongest improvements were observed on sparse and moderately dense graphs, where articulation points and strong articulation points occur frequently and expose structural bottlenecks in the support graph. These results suggest that global connectivity reasoning can provide practical benefits beyond traditional cycle-prevention propagation in modern Lazy Clause Generation solvers.

Several directions for future work remain open. One important improvement would be the introduction of incremental graph analysis in order to avoid reconstructing support graphs repeatedly during propagation. It would also be interesting to experiment with using this propagator only conditionally, specifically when the support graph is already sparse enough. This could be measured by the number of edges still in the graph or by the degrees of the nodes.

It would also be interesting to evaluate the propagator on additional benchmark families and real-world routing instances. This would provide further insight into the relationship between graph structure and propagation effectiveness, and help identify the classes of problems for which articulation-based reasoning is most beneficial.

Finally, while this work includes a preliminary analysis of learned nogood lengths, the primary focus remains on propagation strength and solver performance rather than explanation minimisation. The explanations generated by the propagator are not guaranteed to be minimal, and it would be interesting to investigate whether smaller explanations can be derived without sacrificing propagation strength. Such a study could evaluate the impact of explanation minimisation using metrics such as learned nogood length, Literal Block Distance (LBD), clause reuse and overall solver performance within the Lazy Clause Generation framework.

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009.

[2] Thibaut Feydy and Peter J. Stuckey. Lazy clause generation reengineered. In Ian P. Gent, editor, *Principles and Practice of Constraint Programming - CP 2009, 15th International Conference, CP 2009, Lisbon, Portugal, September 20-24, 2009, Proceedings*, volume 5732

of *Lecture Notes in Computer Science*, pages 352–366. Springer, 2009.

[3] 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.

[4] Kathryn Glenn Francis and Peter J. Stuckey. Explaining circuit propagation. *Constraints An Int. J.*, 19(1):1–29, 2014.

[5] Bezalel Gavish and Stephen C. Graves. The Travelling Salesman Problem and Related Problems. Technical report, Massachusetts Institute of Technology, Operations Research Center, July 1978.

[6] Nicolas Isoart and Jean-Charles Régin. A linear time algorithm for the k-cutset constraint. In Laurent D. Michel, editor, *27th International Conference on Principles and Practice of Constraint Programming, CP 2021, Montpellier, France (Virtual Conference), October 25-29, 2021*, volume 210 of *LIPIcs*, pages 29:1–29:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[7] Michael Jünger, Gerhard Reinelt, and Giovanni Rinaldi. Chapter 4 The traveling salesman problem. In *Network Models*, volume 7 of *Handbooks in Operations Research and Management Science*, pages 225–330. Elsevier, 1995.

[8] Francesca Rossi, Peter van Beek, and Toby Walsh, editors. *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*. Elsevier, 2006.

[9] Francesca Rossi, Peter van Beek, and Toby Walsh. Constraint programming. In Frank van Harmelen, Vladimir Lifschitz, and Bruce W. Porter, editors, *Handbook of Knowledge Representation*, volume 3 of *Foundations of Artificial Intelligence*, pages 181–211. Elsevier, 2008.

[10] Christian Schulte and Peter J. Stuckey. Efficient constraint propagation engines. *ACM Trans. Program. Lang. Syst.*, 31(1):2:1–2:43, 2008.

[11] Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

[12] Willem-Jan van Hoeve and Irit Katriel. Global constraints. In Francesca Rossi, Peter van Beek, and Toby Walsh, editors, *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*, pages 169–208. Elsevier, 2006.