AllDifferent Inside Circuit: An Experimental Evaluation of Propagation Strength and Explanation Quality in Lazy Clause Generation

Mireia Carrió Cortada
Supervisor(s): Emir Demirović, Imko Marijnissen
[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: Mireia Carrió Cortada
Final project course: CSE3000 Research Project
Thesis committee: Emir Demirović, Imko Marijnissen, Andreea Costea

Abstract

Embedding AllDifferent within Circuit is standard practice for routing problems in Constraint Programming, but in a Lazy Clause Generation (LCG) solver this creates a tension: stronger propagators prune more but produce larger, more situation-specific explanations, yielding learned clauses that generalise poorly. AllDifferent and Circuit have only ever been studied separately, under incomparable conditions, leaving practitioners without an evidence-based choice of AllDifferent propagator inside Circuit in LCG. We close this gap with a systematic experimental evaluation of four propagator variants of increasing strength: a decomposed baseline, a matching-based conflict detector, a full GAC propagator, and a circuit-aware extension of GAC, implemented in the Pumpkin LCG solver and tested on structured geographic and unstructured random-graph benchmarks under fixed and activity-based search. Full GAC (V2) is the strongest choice overall, cutting runtime by up to two orders of magnitude by eliminating conflicting branches before the solver enters them; the theoretically stronger circuit-aware variant fails to improve on it because its extra pruning condition rarely fires in practice. V2's runtime advantage holds even where explanation quality favours weaker propagators, but narrows sharply on unstructured graphs as its clause reusability degrades, showing that propagation strength alone cannot predict LCG performance. We recommend full GAC as the default AllDifferent propagator for Circuit in LCG solvers.

1 Introduction

Constraint Programming (CP) is a powerful paradigm for solving combinatorial problems. Rather than relying solely on exhaustive enumeration, CP models a problem through variables and constraints, and combines constraint propagation with systematic search [1]. Propagation removes values that cannot participate in any solution, thus reducing the search space. This combination makes CP particularly effective for solving large, structured problems in domains such as routing [16], scheduling [2], and resource allocation [20].

Routing and sequencing problems form a major application within CP, since many real-world problems involve constructing a valid tour or cycle. The Circuit constraint is the standard way to express such a structure, since it ensures that a selected set of edges forms a single Hamiltonian cycle [9]. Because every node must have a unique successor in such a tour, the Circuit constraint is often paired with the AllDifferent constraint, which requires all variables to take distinct values, thus enforcing that no two nodes point to the same next node.

In recent years, CP solvers have made substantial progress on routing problems through a technique called Lazy Clause Generation (LCG) [14, 7]. LCG enhances standard CP search by learning from failures. Whenever the solver reaches a conflict, it analyses why and records a reason, known as a learned clause, that prevents it from making the same mistake in a different part of the search. This mechanism has delivered major performance improvements across a wide range of problems [17]. Both AllDifferent [6] and Circuit [9] have been adapted to work within this framework.

Yet stronger propagation does not automatically lead to better solving in this setting: a propagator that reasons about many variables at once must produce larger, more situation-specific explanations, yielding clauses that are less reusable elsewhere in the search. This tension is especially sharp when AllDifferent is embedded within Circuit, and its practical significance has never been studied.

To see why this matters, consider the example in Figure 1. Five nodes must be arranged into a single cycle, and each node's arrow can only point to certain successors, as shown on the left. The right side reframes this as a matching problem: each variable (left column) must be paired with a unique successor value (right column). Four variables – x_1, x_2, x_3, x_4 – compete for only three available values – x_1, x_4, x_5 – a contradiction visible immediately from this structure. A propagator that only checks pairs of nodes misses this entirely and must explore many wrong paths before discovering the conflict. A stronger propagator detects it immediately by reasoning about all four nodes as a group. However, in an LCG solver, it must also explain why, referencing all four nodes at once. This produces a learned clause tightly tied to this specific situation, providing little reusable guidance as search continues.

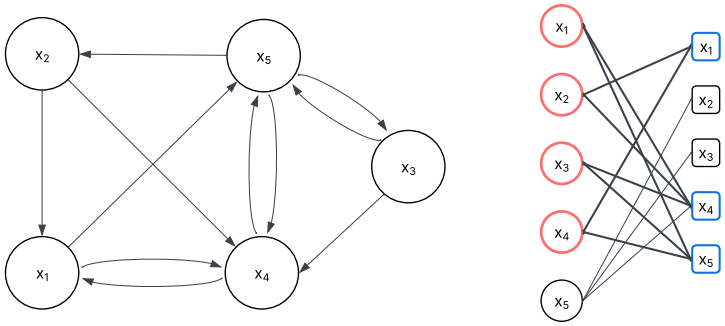

Figure 1: Five-node Circuit instance where x_1, x_2, x_3, x_4 collectively compete for only three successor values x_1, x_4, x_5, making the instance infeasible. A decomposed propagator must branch to discover this, whilst a global propagator detects this immediately.

Prior work leaves this tension unresolved. Studies of strong AllDifferent propagation evaluate it as a standalone constraint [6, 15, 19] while studies of Circuit assume a fixed AllDifferent implementation [9, 3, 12]. These bodies of work were conducted under different conditions and benchmarks, making their conclusions impossible to directly compare. On top of that, the LCG setting introduces explanation costs that are invisible to classical CP analysis and were not a concern in earlier work. Practitioners currently have no evidence-based answer to a basic design question: which AllDifferent propagator should be used inside Circuit in an LCG solver?

This paper is a systematic experimental evaluation that directly answers this question. We implement four propagator variants of increasing strength: a simple decomposed baseline (V0), one that adds global conflict detection (V1), one that adds full domain pruning (V2), and one that further combines this with circuit-specific reasoning based on a recent integration by Bertagnon and Gavanelli [4] (V3). We evaluate them across two families of benchmarks chosen to test whether results depend on problem structure or reflect more general behaviour.

Our evaluation shows that the relationship between propagation strength and solver performance is complex and has direct implications for solver design. Full domain pruning (V2) is the strongest choice across most configurations, improving runtime by up to two orders of magnitude over the baseline, at the cost of a small per-call overhead that fails to pay off only on the easiest, lowest-density instances. Stronger propagation does not always help: V1 can be harmful on structured instances, and V3, despite being strictly stronger than V2, fails to improve on it because its additional pruning step rarely fires in practice. On

structured instances, V2's advantage over V1 comes almost entirely from structural search-space reduction rather than explanation quality, which differs only marginally between them; on unstructured graphs, by contrast, V2's pruning remains just as active but its explanation cost rises sharply, eroding most of that advantage.

In summary, our contributions are as follows. We provide the first experimental evaluation of how AllDifferent propagation strength interacts with the Circuit constraint in an LCG solver, giving practitioners a concrete evidence-based answer to a design question prior literature leaves open. We show that V2 should be the default choice when embedding AllDifferent within Circuit regardless of graph density or search strategy, and that propagation strength alone does not predict solver performance in the LCG setting. Finally, we show this explanation-quality tension is benchmark-dependent: negligible on structured instances, where V2 wins through structural pruning alone, but decisive on unstructured graphs, where V2's pruning produces clauses of lower quality than V1's.

The remainder of this paper is structured as follows. Section 2 introduces preliminaries. Section 3 describes our implementation for the four variants. Section 4 presents our experimental setup and results. Section 5 discusses responsible research considerations, and Section 6 concludes with directions for future work.

2 Preliminaries

Constraint Programming & Lazy Clause Generation: A Constraint Satisfaction Problem (CSP) is a triple (X, D, C) where $X = \{x_1, ..., x_n\}$ is a set of variables, $D = \{D(x_1), ..., D(x_n)\}$ is a set of domains where each $D(x_i)$ is the finite set of values that x_i can take, and $C = \{C_1, ..., C_m\}$ is a set of constraints. Each constraint C_j is defined over a subset of variables and specifies which combinations of values for those variables are allowed. CP solvers find solutions by interleaving both *propagation* and *search*. A propagator for a constraint C removes values from variable domains that cannot participate in any solution to C, reducing search space before and during branching. A propagator achieves *generalised arc-consistency* (GAC), the strongest standard level of filtering, if for every variable x_i and every remaining value $v \in D(x_i)$, there exists a satisfying assignment of all other variables in C.

Standard CP solvers discard all propagation work upon backtracking, so the same or similar conflicts can be rediscovered repeatedly with no way to generalise from past failures. Lazy Clause Generation (LCG) [14] addresses this by integrating SAT-style clause learning [7] into CP: every domain reduction a propagator makes must come with an *explanation* justifying why it was necessary. Formally, an explanation is an implication $l_1 \wedge \cdots \wedge l_k \Rightarrow y$, where each l_t is a literal representing an atomic constraint (e.g. $[x_i = v]$) holding at the current search state and y is the literal denoting the value removed. When the solver reaches a contradiction, conflict analysis combines the explanations responsible into a nogood: a set of literals whose simultaneous truth is inconsistent with any solution. The nogood is stored as a learned clause, letting the solver backjump to the earliest decision implicated in the conflict rather than backtracking step by step, skipping branches that would only rediscover the same failure. A learned clause's quality depends entirely on the explanations that built it: a short, general explanation yields a clause whose literals stay relevant across many branches, while a long, situation-specific one applies only narrowly. The *Literal Block Distance* (LBD) [18] captures this, counting the distinct decision levels spanned by a clause's literals: low LBD indicates a reusable clause, high LBD a narrowly applicable one.

Graph-Theoretic Background: AllDifferent propagation is naturally modelled on a *bipartite graph* $G = (U \cup V, E)$, where U is the set of variables, V is the union of their domains, and an edge $(x_i, v) \in E$ exists whenever $v \in D(x_i)$. A *matching* $M \subseteq E$ pairs each vertex with at most one neighbour; a *perfect* matching on U pairs every variable with a distinct value, which is exactly what AllDifferent requires. Orienting matched edges from V to U and unmatched edges from U to V yields the *residual graph* of G with respect to M; its strongly connected components (SCCs) identify which edges can still participate in some perfect matching. For $S \subseteq U$, its *neighbourhood* is $N(S) = \{v \in V \mid \exists\, u \in S,\ (u, v) \in E\}$. By Hall's theorem, a perfect matching on U exists iff $|N(S)| \geq |S|$ for every $S \subseteq U$: a set with $|N(S)| < |S|$ is a *Hall set violation*, and one with $|N(S)| = |S|$ is a *tight Hall set*, whose variables are confined to exactly the values in $N(S)$.

Constraints: The AllDifferent constraint over $X = \{x_1, \ldots, x_n\}$ requires all variables to take distinct values ($x_i \neq x_j$ for all $i \neq j$). The propagators in this paper realise this with differing strength: decomposed AllDifferent enforces it via $\binom{n}{2}$ independent pairwise inequalities, which are cheap but cannot detect Hall set violations; full GAC, achieved via Régin's algorithm [15], instead removes exactly those values excluded by the bipartite matching structure above.

The Circuit constraint over successor variables $x_1, \ldots, x_n$, with $D(x_i) \subseteq \{1, \ldots, n\} \setminus \{i\}$, requires that the assignments $\{x_i = j\}$ form a single Hamiltonian cycle over all n nodes. Since determining whether a Hamiltonian cycle exists is NP-hard, Circuit typically combines an AllDifferent component enforcing unique successors with a subtour-prevention mechanism.

3 Propagator Variants for Circuit with AllDifferent in LCG

No prior work directly compares AllDifferent propagation strategies when embedded within Circuit in an LCG solver: existing studies either evaluate AllDifferent as a standalone constraint [6, 15, 19] or study Circuit with a fixed AllDifferent implementation [9, 3, 12]. These bodies of work were conducted under different conditions and benchmarks, making their conclusions impossible to directly compare. To fill this gap, we implement four propagator variants of strictly increasing strength, each adding exactly one enhancement over its predecessor so that the contribution of each component can be isolated. **V0** (decomposed baseline) establishes our weakest correct implementation. **V1** (conflict-only matching) adds global Hall-set detection. **V2** (full GAC) extends this with complete SCC-based domain pruning. **V3** (Hall-circuit pruning) further combines AllDifferent and Circuit reasoning into a unified pruning step. For each upgrade, the central question is whether stronger inference justifies its additional propagation cost and, critically in the LCG setting, its explanation complexity.

3.1 V0: Check-Prevent with Decomposed AllDifferent

V0 acts as our weakest correct implementation, establishing the reference point against which the other variants are measured.

Subtour prevention: The circuit component follows the check-prevent algorithm [9]. The *check* procedure traverses the chain of fixed assignments $i \to x_i \to x_{x_i} \to \cdots$ until it either reaches an unfixed variable or detects a cycle. A cycle of size $|C| < n$ constitutes a subtour and raises a conflict. The *prevent* procedure examines the directed graph induced

by currently fixed successors and identifies maximal fixed paths starting from nodes with no fixed predecessor. For a path $i_1 \to \cdots \to i_k$, any value j for x_{i_k} that would close a cycle of size less than n is pruned from $D(x_{i_k})$.

AllDifferent: The AllDifferent component is decomposed into $\binom{n}{2}$ pairwise inequality constraints. Each pair (x_i, x_j) removes v from $D(x_i)$ whenever x_j is assigned v, and symmetrically.

Explanations: Pairwise removal of v from $D(x_i)$ because $x_j = v$ is explained by the single literal $[x_j = v]$. For subtour prevention, V0 uses the cycle-literal scheme of Francis and Stuckey [9]: explanations consist solely of the fixed successor assignments along the relevant path. The left clause explains a detected subtour conflict; the right clause explains the pruning of a single value that would close a premature cycle.

$$\bigwedge_{t=1}^{k} [x_{i_t} = \mathrm{val}(x_{i_t})] \Rightarrow \bot \qquad\qquad \bigwedge_{t=1}^{k} [x_{i_t} = \mathrm{val}(x_{i_t})] \Rightarrow [x_{i_k} \neq i_1]$$

Francis and Stuckey found the no-exit scheme to be empirically stronger, as its clauses tend to generalise better across branches. V0 deliberately uses the cycle-literal scheme instead, since it arises directly from the check-prevent traversal with no additional bookkeeping, keeping V0 as a minimal correct baseline. Adopting no-exit here would partially optimise V0's explanation quality independently of its AllDifferent component, which would confound the comparison.

3.2 V1: Conflict-Only Matching-Based Propagator

V1 replaces the decomposed AllDifferent with a matching-based feasibility check while leaving the subtour-prevention component unchanged. Its purpose is to isolate the contribution of global conflict detection from that of domain pruning. If V1 already captures most of the benefit over V0, the additional cost of full SCC computation in V2 would be unjustified; conversely, if V1 causes harm, the source of that harm must be explained before V2's improvement can be properly understood.

Propagation: V1 constructs the bipartite variable-value graph and computes a maximum matching M using Hopcroft-Karp [11]. If M is not a perfect matching, Hall's theorem guarantees the existence of a violating set $S \subseteq X$ with $|N(S)| < |S|$, and V1 immediately reports infeasibility.

Explanation: The Hall set S is extracted from the graph. The explanation clause states that all variables in S are confined to an insufficient value set:

$$\bigwedge_{x_j \in S} [x_j \in N(S)] \;\Rightarrow\; \bot.$$

In the implementation, confinement to $N(S)$ is encoded through bound and hole predicates rather than a single set-membership literal. For example, in the five-node instance of Figure 1, $S = \{x_1, x_2, x_3, x_4\}$ and $N(S) = \{x_1, x_4, x_5\}$, giving $|S| = 4 > |N(S)| = 3$, and the explanation clause references all four variables confined to these three values.

Moreover, unlike the decomposed AllDifferent, which explains each removal with a single literal, V1's explanation must reference all variables in S and their collective confinement to $N(S)$. This means explanation size grows with the Hall set involved, and since larger Hall sets are more likely to span multiple decision levels, this tends to produce higher-LBD learned clauses that are more specific to the situation that triggered them and provide less reusable guidance elsewhere in the search.

3.3 V2: Full GAC with Matching-Based Pruning

V2 extends V1 with complete domain pruning, implementing Régin's GAC algorithm [15] in full. Beyond detecting infeasibility, V2 removes individual values from domains that cannot participate in any perfect matching, eliminating candidate values before branching. This is the key difference from V1: rather than waiting for a Hall violation to become a full conflict, V2 prunes away the values that would eventually lead to such conflicts.

Propagation: V2 proceeds in four steps:

1. Find a maximum matching M. If M is not a perfect matching, report infeasibility (identical to V1).
2. Construct the directed residual graph with matched edges oriented from V to U and unmatched edges from U to V.
3. Compute the SCCs of the residual graph using Tarjan's algorithm.
4. Remove every unmatched edge $(x_i, v) \notin M$ whose endpoints belong to different SCCs. Such edges do not lie on any directed cycle and cannot participate in any perfect matching.

Explanation: For each pruned value $v \in D(x_i)$, V2 identifies a tight Hall set T that witnesses the removal. The set T satisfies $|N(T)| = |T|$, meaning T's variables are already committed to exactly the values in $N(T)$. Since $v \in N(T)$ but $x_i \notin T$, assigning $x_i = v$ would force $|T| + 1$ variables to share only $|T|$ values, violating Hall's condition.

$$\bigwedge_{x_j \in T} [x_j \in N(T)] \;\Rightarrow\; [x_i \neq v].$$

3.4 V3: Hall-Circuit Pruning

V3 extends V2 with an additional pruning step that combines Hall-set reasoning with circuit structure, based on Bertagnon and Gavanelli [4]. The key observation is that a tight Hall set H not only constrains the AllDifferent component but also imposes structural restrictions on the circuit: any Hamiltonian cycle must enter and exit H the same number of times. When H has exactly one entry node x_s and exactly one exit value v_e, assigning $x_s = v_e$ would trap the nodes of H into a subtour and is therefore pruned. This reasoning goes strictly beyond what V2's SCC-based pruning achieves, since it exploits the global circuit structure rather than just the local bipartite graph.

Propagation: V3 identifies all tight Hall sets from the SCC decomposition already computed by V2 and checks each for a unique entry-exit pair (x_s, v_e).

Explanation: To explain the pruning of $x_s \neq v_e$, V3 constructs the witness set $W = H \backslash \{x_s\}$, which is itself a tight Hall set with $|W| = |N(W)|$ and $v_e \notin N(W)$. The explanation clause is:

$$\bigwedge_{x_j \in W} [x_j \in N(W)] \;\Rightarrow\; [x_s \neq v_e].$$

The explanation is structurally identical to V2's: confinement of W to $N(W)$ witnesses the pruning, with the same bound-and-hole encoding. However, V3's tight Hall sets are identified through circuit structure rather than purely from the bipartite graph, meaning they may capture different, and potentially larger, subsets of variables than V2's pruning would. Whether this additional pruning reduces failures enough to offset any increase in explanation size is a question our experiments directly address.

4 Experimental Evaluation

This section addresses three research questions across two benchmark families. RQ1: does matching-based AllDifferent propagation improve performance within Circuit, and which variant is best? RQ2: do the results generalise to learning-driven (VSIDS) search, or are they specific to fixed-order search? RQ3: do the effects depend on geometric structure, or generalise to unstructured graphs? Experiment 1.1 (Section 4.3) answers RQ1 on geographic instances: V2 is the strongest variant, cutting runtime by up to two orders of magnitude, while V1 is unreliable and V3's extra pruning step rarely fires. Experiment 1.2 (Section 4.4) answers RQ2: V2 remains the default under VSIDS, though V3 narrows its gap. Experiment 2 (Section 4.5) answers RQ3 on random graphs: V2 still wins, but its margin over V1 shrinks as the explanation cost of its pruning rises.

4.1 Benchmark Design and Instance Generation

A central design goal is to separate the effect of graph density from the effect of geometric problem structure. Using only one benchmark family would leave open whether any observed effect is specific to that family's structure or reflects a general property of the propagators. We therefore use two complementary families: one that is spatially structured and one that is purely random.

Geographic k-nearest instances (primary benchmark) We generate structured instances following Francis and Stuckey [9], placing n locations uniformly at random and connecting each node to its k nearest neighbours. A random walk over the graph guarantees the existence of at least one Hamiltonian cycle. These instances model realistic routing structure: edges reflect spatial proximity, and k directly controls domain size while keeping the graph sparse in a geometrically coherent way. Using the same benchmark family as Francis and Stuckey [9] also ensures that our results are directly comparable to the most relevant prior work on Circuit propagation in LCG.

We treat k as the primary independent variable and vary n and k over the grid in Appendix A.1, covering $n \in \{20, 50, 100, 150\}$ and $k \in \{5, ..., 20\}$. This range provides enough domain variation for propagator differences to emerge, while remaining tractable within the 1800s timeout. Certain (n, k) combinations are excluded: very small k at large n risks near-degenerate instances dominated by the random walk, while very large k at small n approaches a complete graph where all variants behave identically. Instance counts per configuration (Appendix A.1) are determined by computational budget and produce stable median and inter-quartile range.

We do not evaluate on TSPLIB instances because their near-complete graph structure prevents any study of density effects, and LCG solvers are not competitive on dense TSP optimisation, which would obscure propagator differences through uniform timeout behaviour.

We similarly do not draw on the standard MiniZinc benchmark suite, as its Circuit-related instances are limited to stochastic VRP and CVRP variants. These are heavily side-constrained (capacity, time windows, demand stochasticity) in ways that dominate solver behaviour independently of AllDifferent propagation strength, and like TSPLIB they tend to produce uniform timeout behaviour that would obscure rather than reveal propagator differences.

Random Erdős–Rényi graphs (secondary benchmark) To test whether the effects observed on structured instances generalise to problems without spatial regularity, we additionally evaluate on directed Erdős–Rényi random graphs, where each directed edge is

included independently with probability p. Unlike the primary benchmark, these instances have no geometric locality, no guaranteed Hamiltonian cycle, and domain sizes determined by random edge inclusion rather than spatial layout. Any propagator effect that appears on both families can be attributed to general algorithmic properties rather than geometric artifacts.

We use $n \in \{20, 50, 100, 150\}$, $p \in \{0.05, 0.10, 0.15\}$, and 20 instances per configuration. The n values are chosen to match the primary benchmark directly, enabling a controlled comparison at the same problem sizes. The p values are chosen to span the density threshold around which random directed graphs reliably contain a Hamiltonian cycle, allowing us to observe propagator behaviour across both degenerate and solvable regimes.

4.2 Experimental Procedure

All instances are compiled from MiniZinc [13] to FlatZinc using the Pumpkin backend and solved with the Pumpkin LCG solver [8]. All instances use a minimum-cost optimisation objective following the model of Francis and Stuckey [9] (Appendix A.2), which forces the solver to prove optimality rather than stop at the first solution, generating substantially more propagation activity and making propagator differences more visible. Unless otherwise specified, experiments use input-order as the variable-selection heuristic and indomain-min as the value-selection heuristic, following Francis and Stuckey [9]. Input-order assigns variables in a fixed sequence regardless of domain state, avoiding any interaction between variable selection and propagator behaviour and ensuring that differences in search effort are attributable to the propagator alone. Indomain-min is used for the same reason: it is a deterministic choice that keeps the comparison clean. Experiment 1.2 (Section 4.4) departs from this default to test whether the propagator ranking holds under activity-based VSIDS search.

Experiments were run on DelftBlue [5], the TU Delft high-performance computing cluster, using Phase 2 compute nodes (Intel Xeon Gold 6448Y, 64 cores, 256 GB RAM) via the SLURM workload manager. Each job was allocated a single CPU core and 4 GB of memory. We impose a per-instance timeout of 1800s for the primary benchmark [9]. For the secondary random graph benchmark a shorter timeout of 300s is used, since instances at the sizes studied are either solved quickly by the stronger variants or remain unsolved regardless of time, making a longer timeout uninformative.

Timed-out instances are recorded separately and excluded from aggregate statistics. Since CP runtime distributions are heavy-tailed [10], we report medians rather than means. When timeout rates differ across variants, we additionally report PAR-2 scores: each timed-out instance is penalised at twice the timeout limit (3600s and 600s respectively) and the mean is taken over all instances including penalties. Spread per variant is reported as the interquartile range.

4.3 Experiment 1.1: Structured Instances under Fixed Search

***RQ1**: Does matching-based AllDifferent propagation improve solver performance when embedded within Circuit in an LCG solver, and if so, which variant is most effective?*

The results deliver a clear answer: full domain pruning (V2) is the best choice across all non-trivial configurations. V1 detects conflicts faster than V0 but intervenes too late: after the solver has already entered wrong branches. V2 prunes impossible values before branching, structurally reducing the search space the solver ever explores. V3, despite being strictly stronger than V2, fails to improve on it because its additional pruning step almost never fires on this benchmark family. The remainder of this section builds the

evidence for this narrative, progressing from solve rates through failure analysis to the internal measurements that explain why.

V2 achieves the highest solve rate at every configuration. Figure 2 shows how V2 achieves the highest proven-optimal rate at every configuration and is the only variant that scales beyond $n = 50$ with meaningful solve rates. At $n = 20$, all variants solve all 30 instances optimally, providing no discrimination between them. Differences emerge sharply at $n = 50$, where, at $k = 5$, V0 proves only 24% of instances optimal, V1 reaches 52%, and V2 reaches 68%; V3 (the theoretically strongest variant) solves only 32%, strictly worse than V2 and comparable to V1 at higher densities. At $n = 100$, V2 is the only variant proving optimality at $k = 15$ and $k = 20$, and at $n = 150$, solve rates are low across all variants, but V2 still proves 3 instances at $k = 10$ while V3 proves none.

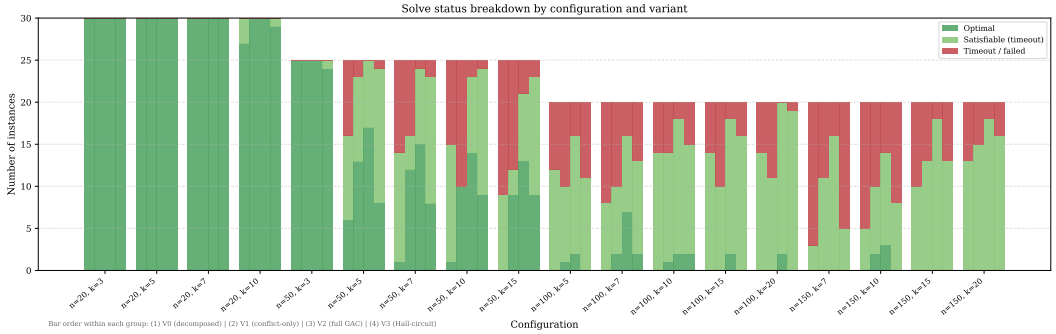

Figure 2: Solve status across all (n, k) configurations (bars left to right: V0, V1, V2, V3). Green = proven-optimal, light green = satisfiable (timeout), red = timeout/failure. At $n = 20$ all variants solve every instance; from $n = 50$ onward, V2 achieves the highest optimal rate.

PAR-2 scores confirm that V2 is not merely better on average but more reliably so (Appendix B.1). At $n = 50$, $k = 5$, V2 achieves a median PAR-2 of 46.6s versus V0's penalty ceiling of 3600s, which is an improvement of nearly two orders of magnitude. At $n = 50$, $k = 7$, V1's median PAR-2 hits 3600s despite solving 48% of instances optimally, revealing extreme variance. This variance reflects a fundamental timing problem: V1 only intervenes when a full Hall violation is already present, meaning the solver has already entered wrong branches before V1 contributes anything. Whether V1 helps therefore depends entirely on which branches the solver happens to explore. V2's PAR-2 at the same configuration is 292s. This likely reflects its continuous pruning (59–65% of all propagation calls at $n = 50$), which eliminates wrong branches structurally before the solver ever enters them.

The exception to V2's dominance is $n = 50$, $k = 3$, where all matching-based variants solve 100% of instances and V1's median solve time (2.37s) beats V2's (4.78s). This is expected: when no instance is hard, V2's per-call SCC computation accumulates as overhead without benefit. The crossover occurs between $k = 3$ and $k = 5$, after which V2's structural pruning outweighs its per-call cost.

Failure counts reveal V2's structural advantage. The solve rate results might suggest that stronger propagation leads to better performance but V3 already breaks that pattern. Failure counts complicate the picture further.

At $n = 20$, where timeouts do not confound comparisons, V1 reduces propagations over V0 substantially, confirming that global Hall set detection eliminates wasted steps. Beyond $n = 20$, this clean comparison no longer holds: timeout rates diverge sharply between variants (Figure 2), so failure and propagation counts for V0 and V1 at $n \leq 50$ are computed only over the subset of instances those variants manage to solve – a subset that is both smaller and likely easier than V2's. The PAR-2 scores reported above mitigate this by penalising timeouts directly, but the raw count comparisons elsewhere in this section should be read with this selection effect in mind. Yet failure counts tell a different story: V1 never approaches V2's failure reduction regardless of density, and at $k = 3$ actually incurs more failures than V0. V2 consistently achieves the fewest failures across all configurations (Figure 3), reducing them a further 5-10× over V1 at k =5-10. V1's propagation count decrease therefore does not translate into a failure count decrease, raising the question of what V2 is doing differently to achieve one.

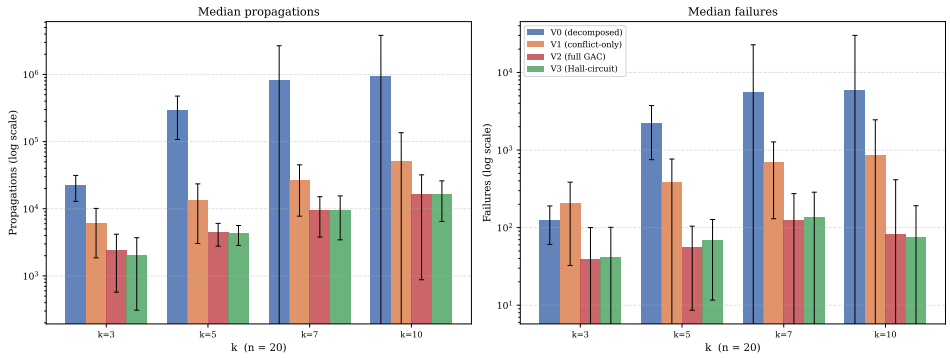

Figure 3: Median propagation counts (left) and failure counts (right) at n=20, where all variants solve all instances and comparisons are unconfounded by timeouts. Both axes use a log scale. V2 consistently achieves the fewest failures across all density levels, reducing them a further 5-10× over V1 despite V1 substantially reducing propagations over V0.

Pruning explains the failure-count gap. The answer is not what V2 learns but what it prevents. V2 prunes on roughly 53-65% of all propagation calls across all configurations at $n = 20$ and $n = 50$ (Appendix B.2). By contrast, V1 only intervenes when a full Hall violation forces a conflict, meaning the solver has already committed to an assignment sequence before V1 contributes anything. Every V2 call that prunes removes a value and the branch beneath it before the solver ever commits to that branch, whereas V1's intervention only ever arrives after the fact. This high, steady firing rate is what the failure counts above are measuring the effect of: detection alone, is not enough to stop the solver from entering most of the branches it eventually backs out of, while proactive pruning is.

The SCC structure underlying V2's pruning shows how its branch elimination capacity scales with density. At $n = 20$, $k = 3$, the SCCs driving pruning average 8.10 nodes but only 3.14 variables per call actually receive a value removal (Appendix B.2), showing how pruning is concentrated in the minority of SCC nodes holding cross-SCC edges. As k increases to 10, this ratio collapses: SCC size drops to 3.33 while variables involved rise to 4.92, meaning at higher density essentially all nodes in a pruning SCC lose a value. Each pruning call therefore removes proportionally more of the remaining search space as density grows, which plausibly contributes to the widening failure-count and PAR-2 advantage V2

holds over V1 as k increases, even though the per-call cost of the SCC computation itself does not grow with density.

Clause quality plays a role, but cannot explain the performance gap. Given that V2 wins through structural search-space reduction, a natural question is whether clause quality plays any independent role. The answer turns out to depend on problem size in a way that's worth treating carefully.

At $n = 20$, where all variants solve every instance and the comparison is therefore unconfounded by timeouts, V1 achieves lower average LBD than V2 at every density, alongside a consistently shorter average nogood length (Appendix B.3). The LBD gap is small in absolute terms (0.06 at $k = 3$, widening to 2.31 at $k = 10$) and the nogood length gap, while larger, follows the same direction throughout. At this size, V1's clauses are a bit more generalisable and noticeably more compact than V2's.

This advantage does not scale cleanly to $n = 50$, where the ordering reverses at most densities and V2 instead attains the lower LBD (Appendix B.3); nogood length shows the same inconsistency, favouring V1 at low density but flipping toward V2 at higher k. We are cautious about reading this as a genuine trend, since a substantial fraction of instances time out at this size, particularly for V0 and V1, so these averages are computed over different, and likely non-comparable, subsets of solved instances per variant. Whether this reflects a real shift in explanation behaviour at scale or simply a difference in which instances each variant manages to solve cannot be determined from the present data, and we flag it as a limitation for future work.

What both regimes agree on, regardless of which direction the gap points at a given size, is the main conclusion: the clause-quality difference between V1 and V2 is far too small to account for the order-of-magnitude difference in solver performance between them. V2 eliminates conflict-definite branches before the solver ever enters them; V1 only intervenes once a conflict has already occurred. At this scale, V2's structural advantage (a 5-10× failure reduction) is large enough to outweigh the clause-quality gap between the two variants.

V3 shows that stronger propagation can backfire. V3 underperforms V2 at almost every configuration, illustrating that strictly stronger propagation does not guarantee better performance. Interestingly, V3's weakness, as we show below, traces back to firing frequency rather than explanation cost. V3 is a strictly stronger propagator than V2: every pruning V2 performs, V3 also performs, plus additional Hall-circuit pruning; however, it is consistently worse.

At $n = 50$, V3's additional cost is rarely justified by its own pruning step: the Hall-circuit condition fires in at most 8.6% of checks across all densities, dropping to 2.8% at $k = 3$ (Appendix B.4), so V3 pays for SCC computation plus entry-exit checking on every propagation call for a step that contributes in fewer than 1 in 10 invocations. This cost is not recovered through explanation quality either: V3's average LBD is consistently higher than V2's at this size (Appendix B.3), meaning the extra pruning condition adds overhead without making clauses any more reusable.

The structural reason is geometric. V3's pruning step requires a tight Hall set with exactly one entry node and exit value, but tight Hall sets in geographic k-nearest instances arise from spatially proximate clusters, which typically have multiple neighbouring nodes pointing in and multiple values pointing out, violating this condition. V3 is not wrong; the geometric structure of this benchmark family simply prevents the condition it needs from arising.

4.4 Experiment 1.2 - Generalisation to Learning Driven Search

RQ2: *Do the results generalise to learning-driven (VSIDS) search, or are they specific to fixed-order search?* The primary experiments deliberately use fixed-order search to isolate propagator effects from search-strategy effects; we now repeat the same benchmark suite under VSIDS, an activity-based heuristic that branches preferentially on variables recently involved in conflicts, making it sensitive to learned clause structure in a way that input-order is not. The core finding is that V2 remains the recommended default under VSIDS, but two nuances emerge that are worth noting.

V2 remains dominant, and its advantage widens with scale. The propagator ordering (V2 best, V1 inconsistent, V0 weakest) is preserved across all non-trivial configurations. At $n = 50$, $k = 5$, V2 achieves a median solve time of 0.053s versus V1's 0.711s and V0's 27.6s, and V2's failure reduction over V1 remains consistent (see Appendix C.2). This gap widens sharply with problem size: V1's PAR-2 is up to 66× worse than V2's at $n = 100$, and up to 540× worse at $n = 150$, where V1 hits the timeout ceiling at several densities while V2 stays under 35s (Appendix C.1). The practical recommendation is therefore unchanged regardless of search strategy, and becomes more decisive as problem size increases.

V2-V3 ordering becomes inconsistent under VSIDS. While the $V0 \leq V1 \leq V2$ ordering from fixed search is preserved, V2 and V3 become statistically indistinguishable: V3 achieves a lower PAR-2 than V2 in 6 of 18 configurations, sometimes by a sizeable margin (Appendix C.1), while V2 leads in the remaining majority. We attribute this primarily to VSIDS's conflict-driven branching being inherently more stochastic than fixed-order search (different runs encounter different conflicts and trigger different tight Hall sets) rather than to a reliable increase in the value of V3's extra pruning step. A secondary possibility is that VSIDS increases the frequency with which V3's entry-exit condition is satisfied, since it concentrates branching on the same constrained clusters that step targets; confirming this would require direct fire-rate measurements under VSIDS, which we leave to future work. Since V3's advantage is inconsistent rather than systematic, V2 remains the recommended default.

LBD drops for all variants under VSIDS, and V2 becomes consistently the most reusable. Every matching-based variant produces lower LBD under VSIDS than under fixed search (Appendix C.3), confirming that search strategy alone substantially affects explanation quality. More notably, the V1-V2 ordering itself reverses: V1 led inconsistently under fixed search, but under VSIDS, V2's LBD is lower than V1's in every configuration, often by a wide margin (Appendix C.3). We hypothesise this reversal occurs because VSIDS concentrates branching on variables recently involved in conflicts, so V2's proactive pruning repeatedly references the same high-activity variables, which VSIDS then assigns at similar search depths, shrinking the decision-level span of its explanations. V1's explanations, by contrast, only materialise once a full Hall violation has occurred, often after the solver has drifted into unrelated branches, so its literals may still span scattered decision levels even under VSIDS. This remains a plausible mechanism rather than an established one, as we have not verified it against decision-level data directly.

4.5 Experiment 2: Unstructured Random graphs

RQ3: *Do the performance differences between variants observed on structured instances generalise to unstructured graphs without geometric locality?*

V2 remains the strongest overall choice, but its advantage over V1 is substantially reduced compared to structured instances. Understanding why reveals how a reduction in clause quality leads to decreased performance.

V2 still wins at scale, but V1 is genuinely competitive at moderate sizes. V2 achieves the highest solve rates at large n, solving 100% of instances at $n = 150$ across all p values, versus V1's 75% at $p = 0.10$ and 55% at $p = 0.15$ (Appendix D.1). However, at moderate problem sizes where both variants solve all instances, V1 can be faster: at $n = 50$, $p = 0.15$, V1 solves in a median of 1.72s against V2's 6.13s. This reversal (not observed on any structured configuration) points to a difference in how the two variants interact with unstructured properties.

Pruning remains active; the cost lies elsewhere. V2's pruning rate stays stable at 50-57% of propagation calls on random graphs, comparable to the 53-65% seen on structured instances; thus, the branch-elimination mechanism is still firing at a similar rate. Failure counts confirm this directly: V2 reduces failures relative to V1 by more than 40× across configurations at $n \geq 50$ (Appendix D.2), a larger margin than observed on structured instances at $n = 20$. Strikingly, even at $n = 50, p = 0.15$, the one configuration where V1 has the faster median wall time (1.72s vs. V2's 6.13s), V2 still needs only 14.5 median failures against V1's 584. Pruning is therefore not weakening on this benchmark; the narrowing or reversal of the runtime gap comes entirely from the rising cost of explaining each prevented failure, not from any loss of pruning effectiveness.

Clause quality explains the narrowing gap. The reason V2's advantage shrinks is clause quality, and the LBD gap between V1 and V2 is substantially larger on random graphs than on structured ones. On k-nearest instances, V1 produced better learned clauses but lost because structural pruning mattered more, and the gap itself was small. Here, the gap widens sharply: V2's average LBD is 1.7-3.1× higher than V1's across configurations (Figure 4), compared to a much smaller and often reversed gap on structured instances. We conjecture this is because, without geometric clustering, V2's tight Hall sets are encountered at points in the search where the variables involved span more decision levels than on k-nearest instances, but we have not verified this against decision-level data directly. Whatever the precise cause, the result is that V2's pruning mechanism is paying a substantially higher explanation-quality cost per pruning call than it did on structured instances, high enough that it erodes most of V2's advantage at moderate sizes, and at $n = 50$, $p = 0.15$ even reverses it in V1's favour, even though pruning itself fires just as often as before.

V3 offers no benefit on random graphs. Its Hall-circuit fire rate is effectively zero across all configurations (≤ 0.0011), compared to 2.8-8.6% on k-nearest instances. The unique entry-exit condition V3 requires almost never arises on unstructured topology, making V3 slower or equal to V2, with no compensating benefit (see Appendix D.3).

5 Responsible Research

To support reproducibility, all propagator implementations, benchmark generators, experiment scripts, and analysis tools are made publicly available in a dedicated evaluation repository: `https://github.com/mcc0112/thesis-evaluation`. Each propagator variant is pinned to an exact solver commit, and every experiment run records the commit used at that time. The repository also records the MiniZinc compiler version and the random seeds used for benchmark generation, so that experiments and benchmark instances can be reproduced precisely.

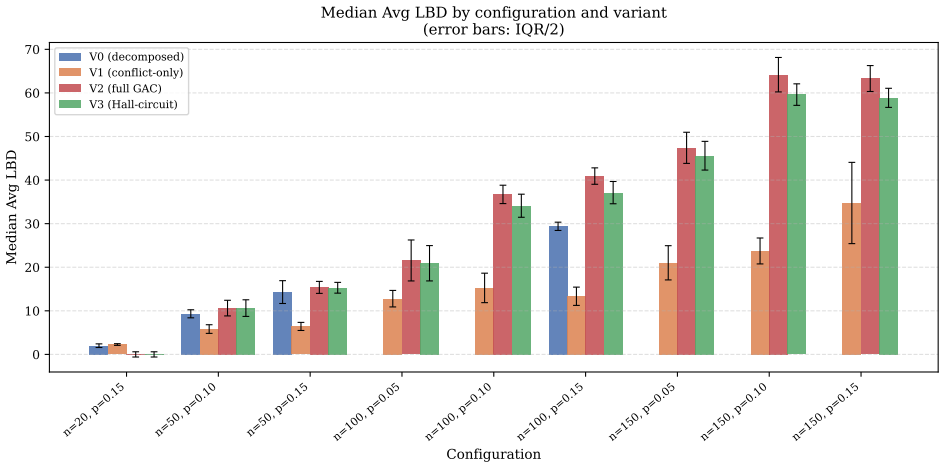

Figure 4: Median Avg LBD by configuration (V0-V3) on unstructured random-graph instances, with error bars showing IQR/2. V2's LBD is consistently 1.7-3.1× higher than V1's across configurations, in contrast to the smaller, often-reversed gap observed on structured k-nearest instances.

6 Conclusions and Future Work

This paper presented a systematic experimental evaluation of how AllDifferent propagation strength interacts with the Circuit constraint inside a Lazy Clause Generation solver – a design question left open by prior literature, with direct relevance to CP-based routing and scheduling applications. Full domain pruning (V2) is the recommended default for embedding AllDifferent within Circuit in an LCG solver, improving runtime by up to two orders of magnitude over the decomposed baseline across almost all configurations and both search strategies, except at very low-density instances, where there is little search space for its structural advantage to act on, so per-call overhead dominates instead. Weaker matching-based detection (V1) is less reliable: its PAR-2 scores show high variance, since it intervenes only once a conflict has occurred rather than pruning proactively like V2. V3, the theoretically strongest variant, fails to improve on V2 under fixed-order search because its additional pruning condition rarely fires in practice, though this gap narrows and occasionally reverses under VSIDS. On unstructured random graphs, the explanation-quality cost of V2's pruning rises enough to erode most of its advantage over V1, underscoring that explanation quality is an important concern in LCG solver design even when pruning itself is unaffected.

Several directions remain open for future work. The subtour-prevention component was held fixed throughout this study; varying its explanation scheme (e.g. adopting no-exit explanations in V0) or using stronger subtour propagation could further isolate the contribution of the AllDifferent component. The conditions under which V3's Hall-circuit pruning reliably fires (suggested to depend on search strategy) call for a dedicated investigation with direct fire-rate measurements under VSIDS. Evaluating all variants on larger instances and denser benchmark families, including real-world problems, would test whether the ordering observed here generalises beyond the problem sizes studied. Finally, exploring simpler or lazier explanation schemes for V2 could reduce its clause length disadvantage on unstructured graphs while preserving its structural pruning benefit.

References

[1] Krzysztof R. Apt. *Principles of constraint programming*. Cambridge University Press, 2003. ISBN 978-0-521-82583-2.

[2] Philippe Baptiste, Philippe Laborie, Claude Le Pape, and Wim Nuijten. *Constraint-Based Scheduling and Planning*. Foundations of Artificial Intelligence. Elsevier, 2006. doi: 10.1016/S1574-6526(06)80026-X. URL https://doi.org/10.1016/S1574-6526(06)80026-X.

[3] Pascal Benchimol, Willem Jan van Hoeve, Jean-Charles Régin, Louis-Martin Rousseau, and Michel Rueher. Improved filtering for weighted circuit constraints. *Constraints An Int. J.*, 17(3):205–233, 2012. doi: 10.1007/S10601-012-9119-X. URL https://doi.org/10.1007/s10601-012-9119-x.

[4] Alessandro Bertagnon and Marco Gavanelli. Stronger integration of circuit and alldifferent propagators for the hamiltonian cycle problem. Technical report, 2024. URL https://ceur-ws.org/Vol-3883/paper2_RCRA7.pdf.

[5] Delft High Performance Computing Centre (DHPC). DelftBlue Supercomputer (Phase 2). https://www.tudelft.nl/dhpc/ark:/44463/DelftBluePhase2, 2024.

[6] Nicholas Downing, Thibaut Feydy, and Peter J. Stuckey. Explaining alldifferent. In Mark Reynolds and Bruce H. Thomas, editors, *Thirty-Fifth Australasian Computer Science Conference, ACSC 2012, Melbourne, Australia, January 2012*, CRPIT, pages 115–124. Australian Computer Society, 2012. URL http://crpit.scem.westernsydney.edu.au/abstracts/CRPITV122Downing.html.

[7] 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*, Lecture Notes in Computer Science, pages 352–366. Springer, 2009. doi: 10.1007/978-3-642-04244-7_29. URL https://doi.org/10.1007/978-3-642-04244-7_29.

[8] 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.

[9] Kathryn Glenn Francis and Peter J. Stuckey. Explaining circuit propagation. *Constraints An Int. J.*, 19(1):1–29, 2014. doi: 10.1007/S10601-013-9148-0. URL https://doi.org/10.1007/s10601-013-9148-0.

[10] Carla P. Gomes, Bart Selman, and Nuno Crato. Heavy-tailed distributions in combinatorial search. In Gert Smolka, editor, *Principles and Practice of Constraint Programming - CP97, Third International Conference, Linz, Austria, October 29 - November 1, 1997, Proceedings*, Lecture Notes in Computer Science, pages 121–135. Springer, 1997. doi: 10.1007/BFB0017434. URL https://doi.org/10.1007/BFb0017434.

[11] John E. Hopcroft and Richard M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM J. Comput.*, 2(4):225–231, 1973. doi: 10.1137/0202019. URL https://doi.org/10.1137/0202019.

[12] Latife Genç Kaya and John N. Hooker. A filter for the circuit constraint. In Frédéric Benhamou, editor, *Principles and Practice of Constraint Programming - CP 2006, 12th International Conference, CP 2006, Nantes, France, September 25-29, 2006, Proceedings*, Lecture Notes in Computer Science, pages 706–710. Springer, 2006. doi: 10.1007/11889205_55. URL https://doi.org/10.1007/11889205_55.

[13] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. Minizinc: Towards a standard CP modelling language. In Christian Bessiere, editor, *Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings*, Lecture Notes in Computer Science, pages 529–543. Springer, 2007. doi: 10.1007/978-3-540-74970-7_38. URL https://doi.org/10.1007/978-3-540-74970-7_38.

[14] Olga Ohrimenko, Peter J. Stuckey, and Michael Codish. Propagation via lazy clause generation. *Constraints An Int. J.*, 14(3):357–391, 2009. doi: 10.1007/S10601-008-9064-X. URL https://doi.org/10.1007/s10601-008-9064-x.

[15] Jean-Charles Régin. A filtering algorithm for constraints of difference in csps. In Barbara Hayes-Roth and Richard E. Korf, editors, *Proceedings of the 12th National Conference on Artificial Intelligence, Seattle, WA, USA, July 31 - August 4, 1994, Volume 1*, pages 362–367. AAAI Press / The MIT Press, 1994. URL http://www.aaai.org/Library/AAAI/1994/aaai94-055.php.

[16] Louis-Martin Rousseau, Michel Gendreau, Gilles Pesant, and Filippo Focacci. Solving vrptws with constraint programming based column generation. *Ann. Oper. Res.*, 130 (1-4):199–216, 2004. doi: 10.1023/B:ANOR.0000032576.73681.29. URL https://doi.org/10.1023/B:ANOR.0000032576.73681.29.

[17] Andreas Schutt, Thibaut Feydy, Peter J. Stuckey, and Mark G. Wallace. Solving rcpsp/max by lazy clause generation. *J. Sched.*, 16(3):273–289, 2013. doi: 10.1007/S10951-012-0285-X. URL https://doi.org/10.1007/s10951-012-0285-x.

[18] Tomohiro Sonobe. Looking inside literal blocks: Towards mining more promising learnt clauses in SAT solving. In *28th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2016, San Jose, CA, USA, November 6-8, 2016*, pages 972–979. IEEE Computer Society, 2016. doi: 10.1109/ICTAI.2016.0150. URL https://doi.org/10.1109/ICTAI.2016.0150.

[19] Willem Jan van Hoeve. The alldifferent constraint: A survey. *CoRR*, cs.PL/0105015, 2001. URL https://arxiv.org/abs/cs/0105015.

[20] Mark Wallace. Practical applications of constraint programming. *Constraints An Int. J.*, 1(1/2):139–168, 1996. doi: 10.1007/BF00143881. URL https://doi.org/10.1007/BF00143881.

A Experimental Setup

A.1 Instance Generation Parameters

Table 1: Parameter grid for geographic k-nearest instances. Each cell reports the number of instances generated per propagator configuration. Cells marked – are excluded for the reasons described in the text.

	$k = 3$	$k = 5$	$k = 7$	$k = 10$	$k = 15$	$k = 20$
$n = 20$	30	30	30	30	–	–
$n = 50$	25	25	25	25	25	–
$n = 100$	–	20	20	20	20	20
$n = 150$	–	–	20	20	20	20

A.2 MiniZinc Model

```
include "globals.mzn";

int: n;
set of int: Locations = 1..n;

int: maxLegLen;

array[Locations, Locations] of int: travelTime;

%% Successor variables: succ[i] = next location after i in the tour.
array[Locations] of var Locations: succ;

%% Only use allowed legs (edges present in the transport network).
constraint forall(loc1, loc2 in Locations)(
  travelTime[loc1, loc2] < 0 -> succ[loc1] != loc2
);

%% Successors must form a Hamiltonian circuit.
constraint circuit(succ);

%% Variable for the length of the longest leg.
var 1..maxLegLen: maxleg;

%% Link maxleg to actual travel times used in the tour.
constraint forall(loc1, loc2 in Locations)(
  succ[loc1] = loc2 -> maxleg >= travelTime[loc1, loc2]
);

solve :: int_search(succ, input_order, indomain_min)
     minimize maxleg;
```

B Supplementary Results – Experiment 1.1 (Structured Instances, Fixed Search)

B.1 PAR-2 Scores at $n = 50$

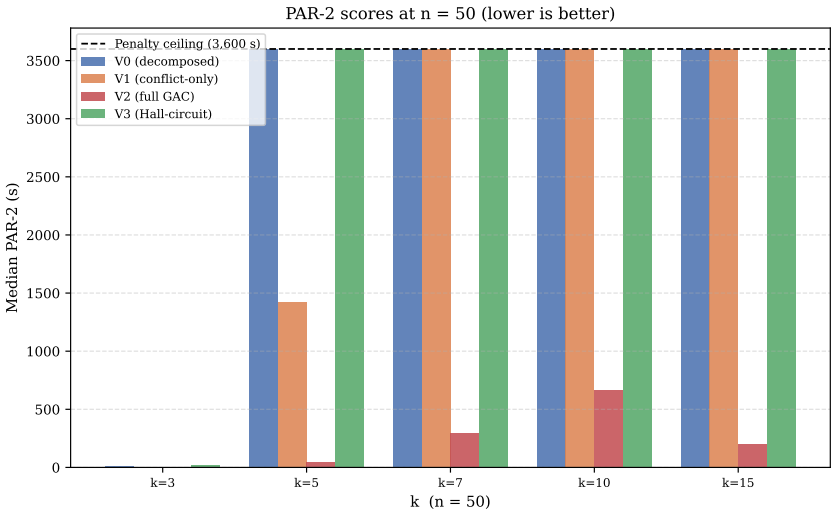

Figure 5: Median PAR-2 scores at $n = 50$ for each propagator variant across density levels, k timed-out instances are penalised at twice the timeout limit (3600s, dashed line). Lower is better. V2 achieves the lowest PAR-2 at every non-trivial configuration; V1 reaches the penalty ceiling at $k = 7$ despite solving 48% of instances, reflecting extreme variance in its behaviour.

B.2 V2 – SCC and Pruning Statistics

Table 2: V2 (full GAC) pruning statistics on geographic k-nearest instances at $n = 20$ and $n = 50$. Pruning rate is the fraction of propagation calls on which at least one value is removed. Vars. involved is the average number of variables receiving a pruning per active call. SCC size is the average size of the strongly connected component driving the pruning.

n	k	Pruning rate	Vars. involved	SCC size	SCC/Vars. ratio
20	3	62.0%	3.14	8.10	2.58
	5	56.6%	3.33	4.70	1.41
	7	56.1%	3.87	4.37	1.13
	10	53.2%	4.92	3.33	0.68
50	3	65.1%	3.63	15.49	4.26
	5	60.7%	3.20	7.22	2.26
	7	60.4%	3.53	9.12	2.59
	10	61.1%	3.42	6.14	1.80
	15	59.0%	4.94	6.66	1.35

B.3 Clause Quality: LBD and Nogood Length

Table 3: Median Literal Block Distance (LBD) and nogood length for all propagator variants on geographic k-nearest instances. Rows correspond to problem configurations (n, k).

	Average LBD				Average Nogood Length			
Configuration	V0	V1	V2	V3	V0	V1	V2	V3
$n = 20, k = 3$	3.88	3.71	3.77	3.67	7.79	5.10	7.53	7.37
$n = 20, k = 5$	5.71	5.02	5.81	6.08	18.51	8.85	14.13	14.95
$n = 20, k = 7$	6.94	5.81	7.10	7.28	28.43	12.18	21.28	20.61
$n = 20, k = 10$	7.04	5.68	7.99	8.13	42.12	17.37	34.76	32.81
$n = 50, k = 3$	6.98	7.46	5.88	7.30	15.38	11.23	14.32	15.44
$n = 50, k = 5$	10.37	11.86	13.38	15.71	31.19	19.35	28.25	25.66
$n = 50, k = 7$	17.90	17.54	17.31	21.54	39.88	37.24	40.08	35.76
$n = 50, k = 10$	21.56	22.53	18.43	24.46	39.58	59.58	50.63	44.77
$n = 50, k = 15$	–	18.70	17.94	22.35	–	68.16	61.76	59.94

B.4 V3 – Hall-Circuit Fire Rate

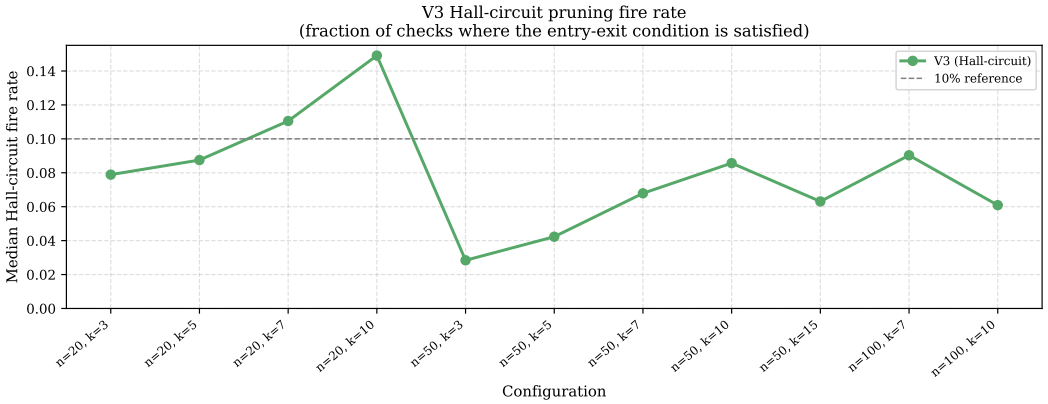

Figure 6: Median Hall-circuit pruning fire rate for V3 across all configurations, defined as the fraction of propagation checks where the unique entry-exit condition is satisfied. The dashed line marks 10% as a reference. The fire rate remains most of the time below 10% at $n = 50$ and drops to 2.8% at low density, which might explain why V3's additional pruning step cannot recover its overhead cost on geographic k-nearest instances.

C Supplementary Results - Experiment 1.2 (Structured Instances, VSIDS Search)

C.1 Solve Rates and PAR-2 Scores

Table 4: Proven-optimal rate and median PAR-2 score under VSIDS search for all propagator variants on geographic k-nearest instances. Timed-out instances are penalised at twice the 1800 s timeout (3600 s ceiling), consistent with the fixed-search experiment.

		Proven-optimal rate				PAR-2 (s)			
n	k	V0	V1	V2	V3	V0	V1	V2	V3
20	3	100%	100%	100%	100%	1.70	9.39	2.06	5.45
	5	100%	100%	100%	100%	0.57	1.94	10.66	7.00
	7	100%	100%	100%	100%	1.24	1.69	1.87	0.46
	10	90%	100%	100%	100%	6.17	1.88	5.93	7.62
50	3	100%	100%	100%	100%	1.77	6.38	2.89	5.03
	5	72%	80%	80%	88%	110.68	2.88	3.36	7.98
	7	28%	96%	96%	92%	3600.00	3.02	5.45	9.83
	10	20%	88%	92%	92%	3600.00	3.89	13.40	7.12
	15	20%	92%	92%	92%	3600.00	4.10	2.25	6.50
100	5	55%	90%	90%	95%	1413.68	24.86	11.16	13.69
	7	0%	90%	80%	85%	3600.00	66.19	5.57	8.90
	10	0%	95%	90%	90%	3600.00	195.94	9.48	9.98
	15	0%	85%	85%	95%	3600.00	198.12	2.98	9.71
	20	0%	80%	95%	95%	3600.00	328.04	8.32	8.16
150	7	0%	55%	95%	100%	3600.00	1452.03	18.23	26.34
	10	0%	30%	90%	100%	3600.00	3600.00	25.11	14.16
	15	0%	20%	100%	100%	3600.00	3600.00	6.62	24.04
	20	0%	10%	95%	90%	3600.00	3600.00	34.37	20.56

C.2 Failure Counts

Table 5: Median number of failures (backtracks) under VSIDS branching for all propagator variants on geographic k-nearest instances. Values are computed over solved instances only; "–" indicates that no instances were solved and hence no median failure count could be computed.

		Median failures (backtracks)			
n	k	V0	V1	V2	V3
20	3	712.5	642.5	638.5	592.0
	5	2323.5	502.0	410.0	399.0
	7	6072.5	492.5	407.0	404.0
	10	7050.0	533.0	412.5	401.5
50	3	1544.0	1466.0	811.0	904.0
	5	129977.5	3390.0	333.5	342.5
	7	145606.0	4955.0	474.0	338.0
	10	233708.0	6078.0	864.0	885.0
	15	79463.0	6185.0	771.0	371.0
100	5	988009.0	35390.5	3113.5	11227.0
	7	–	104341.5	627.5	918.0
	10	–	191032.0	809.0	1533.5
	15	–	176155.0	622.0	3001.0
	20	–	180439.5	1008.0	1430.0
150	7	–	940666.0	11071.0	13939.5
	10	–	574477.0	3791.5	10283.0
	15	–	506286.0	3470.0	11133.0
	20	–	810764.5	20696.0	6949.0

C.3 LBD: Fixed Search vs. VSIDS

Table 6: Average Literal Block Distance (LBD) under fixed (input-order) search versus VSIDS branching, for all propagator variants on geographic k-nearest instances at $n = 20$ and $n = 50$. Lower LBD indicates more reusable learned clauses.

	LBD (fixed search)				LBD (VSIDS)			
Configuration	V0	V1	V2	V3	V0	V1	V2	V3
$n = 20, k = 3$	3.88	3.71	3.77	3.67	1.73	1.50	1.12	1.12
$n = 20, k = 5$	5.71	5.02	5.81	6.08	5.97	1.97	1.16	1.06
$n = 20, k = 7$	6.94	5.81	7.10	7.28	8.02	1.97	1.15	1.11
$n = 20, k = 10$	7.04	5.68	7.99	8.13	7.88	2.31	1.21	1.16
$n = 50, k = 3$	6.98	7.46	5.88	7.30	4.23	3.84	2.06	3.00
$n = 50, k = 5$	10.37	11.86	13.38	15.71	10.90	10.65	3.03	3.47
$n = 50, k = 7$	17.90	17.54	17.31	21.54	11.01	13.60	3.54	4.18
$n = 50, k = 10$	21.56	22.53	18.43	24.46	10.74	16.69	6.19	8.96
$n = 50, k = 15$	–	18.70	17.94	22.35	11.73	17.94	6.41	4.95

D Supplementary Results – Experiment 2 (Random Graphs)

D.1 Solve Rates and PAR-2 Scores

Table 7: Solve rates (fraction proven optimal within 300 s) and median PAR-2 scores (timeouts penalised at 600 s) for all variants on Erdős–Rényi instances; configurations with no solves are omitted. V2 leads at large n; V1 is competitive at moderate sizes but degrades by $n = 150$; V0 solves at most 30%; V3 matches V2's solve rate but is slower.

		V0		**V1**		**V2**		**V3**	
n	p	Rate	PAR-2	Rate	PAR-2	Rate	PAR-2	Rate	PAR-2
20	15%	15%	0.47	15%	2.11	15%	0.56	15%	0.76
50	10%	25%	600.00	75%	4.27	75%	0.23	75%	1.91
50	15%	30%	600.00	100%	1.72	100%	6.13	100%	2.43
100	5%	0%	600.00	20%	0.46	20%	4.68	20%	7.71
100	10%	0%	600.00	90%	3.05	100%	4.14	100%	1.85
100	15%	10%	600.00	100%	5.42	100%	1.86	100%	9.31
150	5%	0%	600.00	73.7%	6.07	90%	2.08	90%	4.34
150	10%	0%	600.00	75%	23.13	100%	6.84	100%	7.47
150	15%	0%	600.00	55%	90.46	100%	7.34	100%	14.89

D.2 Failure Counts

Table 8: Median failure counts (backtracks) on Erdős–Rényi instances for all propagator variants. Medians are computed only over instances solved within the timeout; – indicates no instances were solved for that variant/configuration.

n	p	V0	V1	V2	V3
20	15%	12.0	44.0	0.0	0.0
50	10%	20410.0	276.0	5.0	5.0
50	15%	13541.0	584.0	14.5	14.5
100	5%	–	680.5	6.5	6.0
100	10%	–	2422.5	52.5	52.5
100	15%	135788.0	3721.5	81.0	81.0
150	5%	–	2441.0	53.0	53.0
150	10%	–	5985.0	135.5	136.0
150	15%	–	17571.0	223.5	221.5

D.3 V2/V3 – Internal Propagation Statistics

Table 9: Internal propagation statistics for V2 and V3 on Erdős–Rényi instances. *V2 pruning rate*: fraction of calls removing at least one value. *Vars involved*: variables pruned per active call. *SCC size*: average SCC size driving the pruning. *V2 tight HS size*: average tight Hall-set size witnessing each pruning. *V3 fire rate*: fraction of calls satisfying the Hall-circuit entry-exit condition. Pruning rate stays near 50% throughout; V3's fire rate is effectively zero (≤ 0.0011)

		V2 Pruning Statistics				**V3**
n	p	Pruning Rate	Vars Involved	SCC Size	Tight HS Size	Fire Rate
20	15%	0.57	5.75	10.50	5.37	0.0000
50	10%	0.53	5.33	15.29	7.89	0.0000
50	15%	0.52	5.57	11.76	5.46	0.0011
100	5%	0.52	5.20	31.18	15.67	0.0000
100	10%	0.51	6.49	13.09	6.92	0.0000
100	15%	0.51	7.96	6.83	3.28	0.0000
150	5%	0.51	6.52	26.25	11.88	0.0000
150	10%	0.51	8.31	8.56	4.39	0.0000
150	15%	0.51	10.33	6.56	2.81	0.0001