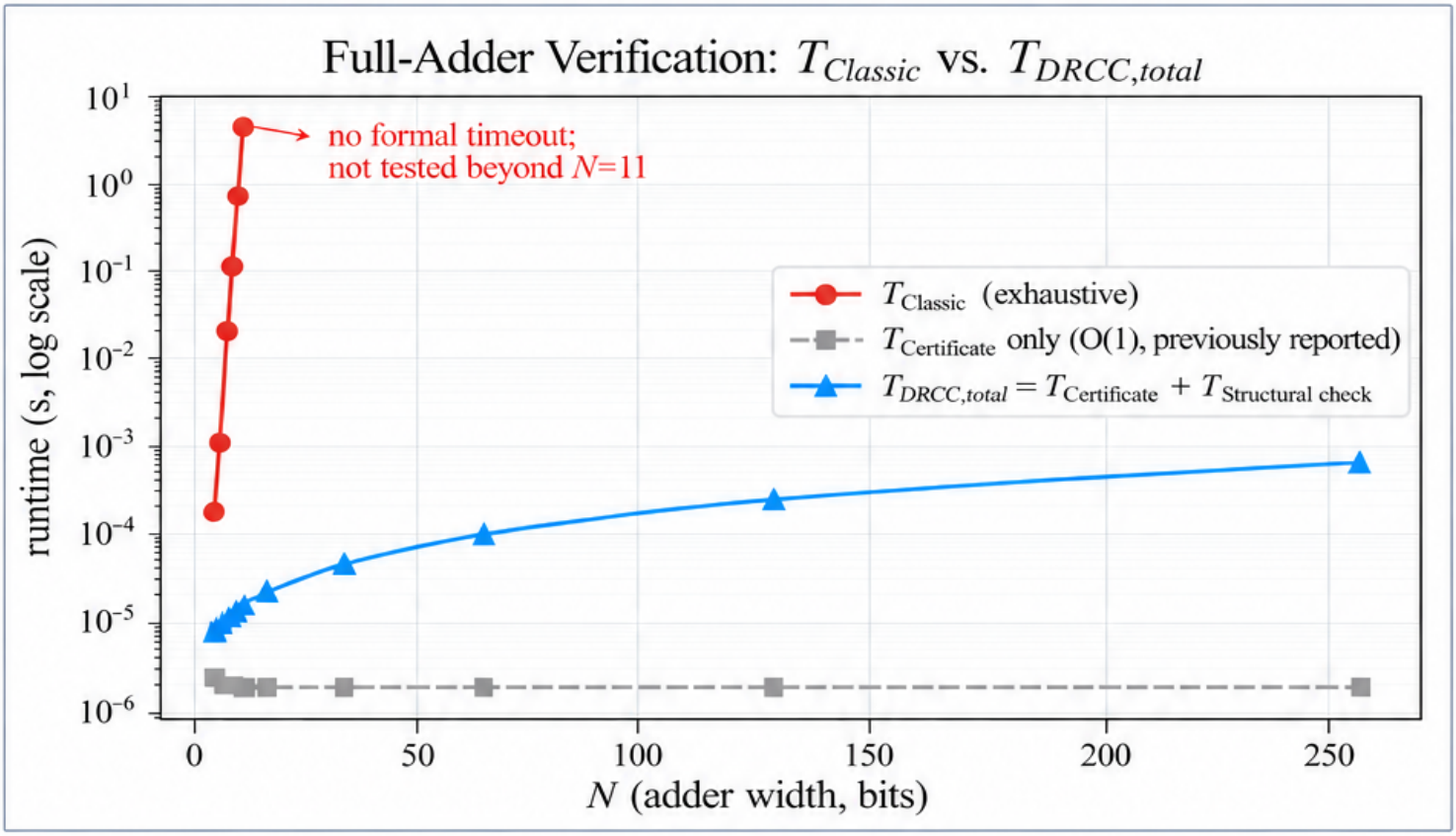
DRCC
Runtime Analysis

Comparative Evaluation of Solver Runtime With and Without DRCC



Reza Hesamiy

Munich, Germany

2026

— DRCC-V3.1 —

DRCC Runtime Analysis

Comparative Evaluation of Solver Runtime With and Without DRCC

Using DPLL, MiniSat, and Glucose

DRCC-V3.1

Reza Hesamiy

Munich, Germany

2026

Abstract

This work introduces DRCC (Dimensional Reduction via Controlled Combinatorics), a structural preprocessing and reconstruction framework for satisfiability problems represented in conjunctive normal form (CNF). The framework is motivated by the observation that many SAT instances contain symmetries, repeated substructures, and task-specific constraints that may not be fully exploited by a generic solver operating directly on the original formula. DRCC separates a concrete structural transformation from subsequent satisfiability search: an input formula F is transformed into a solver-facing representation F_{Red} by a sequence of controlled reconstruction steps, after which a downstream SAT solver determines the satisfiability status.

The reduction is parameterized by a processing order π and a reconstruction rule τ. The processing order specifies which variables or structural units are considered at each stage, while the reconstruction rule determines the admissibility and structural conditions imposed during the reduction. This separation makes the framework explicit and permits the resulting representation to be analyzed independently of the internal heuristics of a particular solver. The formal development distinguishes the elementary maps $\mathcal{R}_i$ from their composition, the overall reduction map

$$\mathcal{R}_{\mathrm{DRCC}} = \mathcal{R}_q \circ \cdots \circ \mathcal{R}_1,$$

where q is the number of processing units.

Two quantities characterize the reduction. The DRCC width,

$$\omega_{\mathrm{DRCC}} = \max_{1 \le i \le q} |B_i^{\pi,\tau}|,$$

measures the maximum active structural interface at the declared processing stages. The DRCC work,

$$W_{\mathrm{DRCC}} = \sum_{i=0}^{q} N_i^{\mathrm{reach}},$$

measures the total number of distinct partial reconstructions that are reachable across all stages. This is a structural-combinatorial work measure, not a machine-level operation count or a wall-clock runtime. These quantities measure different aspects of the process: a large structural interface does not necessarily imply a large number of reachable states or a large implementation-level runtime.

The Pigeonhole Principle family PHP_n^{n-1} serves as a concrete example. Under the specified PHP encoding, pigeon-block processing order, and deterministic reconstruction rule, the formal analysis yields

$$\omega_{\mathrm{DRCC}} = \Theta(n^2) \qquad \text{and} \qquad W_{\mathrm{DRCC}} = \Theta(n).$$

The example illustrates how a controlled reconstruction path can expose a contradiction and enable the tested downstream solver configuration to detect unsatisfiability without branching on the resulting instance. These claims are conditional on the stated satisfiability-preservation requirement and the concrete reconstruction semantics. DRCC therefore does not refute classical resolution or DPLL lower bounds for the original formula; it changes the representation on which the subsequent solver operates. The framework provides a mathematical basis for studying structural preprocessing, reconstruction complexity, and solver performance on structured SAT instances.

Keywords: DRCC; DRCC Runtime Analysis; Dimensional Reduction; Structural Reduction; Reconstruction Width; DRCC Work; Pigeonhole Principle (PHP); SAT Solving; DPLL; CDCL; MiniSat; Glucose; Symmetry Breaking; Satisfiability Preservation; Computational Complexity; Structural Complexity.

Contents

1 Introduction

This record extends DRCC-V3.0 with an executed comparison against representative established CDCL SAT solver implementations. The central question is where explicit DRCC symmetry-breaking affects solver behavior on the tested benchmark family. All reported measured runtime values come from executed runs; derived quantities, theoretical results, confidence intervals, and explicitly labeled predictions are identified as such throughout. The manuscript therefore separates measured evidence, formal proofs, and predictive inference explicitly.

2 Q1: The $\Theta(N)$ Structural Check – Completing the Certificate Cost

The end-to-end verification time of an N-bit adder is considered here together with the structural cost of confirming that the represented adder consists of N correctly instantiated, correctly wired, cells carrying the expected template tag. DRCC-V3.0 disclosed this structural cost separately from its certificate timing.

2.1 Motivation

The verification scope includes the structural cost previously separated from the certificate timing. The final Experiment 1 script, `exp1_full_adder.py`, contains the structural-check extension used for the reported measurements. The implementation-level total is represented conceptually as

$$T_{\mathrm{DRCC,total}}(N) = T_{\mathrm{certificate}}(N) + T_{\mathrm{structural-check}}(N), \tag{1}$$

alongside the original $T_{\mathrm{classic}}(N)$.

This decomposition is a bookkeeping identity for the two measured components in the specified implementation. It is not a universal runtime law for all DRCC implementations.

2.2 Setup

The added function, `structural_check(N)`, builds an explicit N-cell ripple-carry representation and verifies the complete set of declared per-cell fields: the cell index, the expected template tag, the A, B, and S ports, the expected C_{in} source, and the expected C_{out} port. In particular, for $i > 0$, the carry input is required to be `Cout[i-1]`, whereas for $i = 0$ it is required to be the external `Cin`.

The check is a representation-level verification of the fields actually constructed by the implementation. Because the number of checked fields per cell is constant, the construction and checking procedure performs $O(N)$ work, which is measured below.

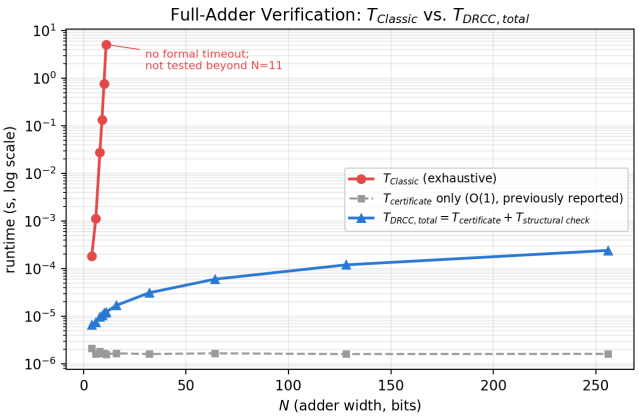


Figure 1: DRCC verification within the defined structural representation ($T_{\mathrm{DRCC,total}}$) (blue) remains at the microsecond scale and includes both the certificate verification and the explicitly measured $\Theta(N)$ structural check. The structural component grows linearly from 4.49 μs at $N = 4$ to 239.5 μs at $N = 256$. Within the tested range, the measured verification pipeline based on the purpose-built structural representation remains in the microsecond range, whereas the exhaustive classical reference reaches approximately 5.1 s at its largest tested size, $N = 11$. This comparison does not include SPICE parsing or external-netlist file I/O. T_{classic} (red) is shown solely as the exhaustive reference baseline; it was tested only through $N = 11$, where its runtime reached approximately 5.1 s, and no formal timeout was applied. The certificate-only DRCC cost ($T_{\mathrm{certificate}}$, grey, dashed) is retained solely to show the effect of including the previously omitted structural verification cost. **Curve-transition feature:** the DRCC runtime curve may exhibit a visible change in growth regime; this is distinct from the formal runtime transition point $W = (n_w, R_w[\mathrm{s}])$, defined by $G_{\mathrm{time}}(n_w) = 1$. Since $G_{\mathrm{time}}(N) > 1$ already at the smallest tested $N = 4$, no formal W is observed in the tested range.

2.3 Results

Finding: linear construction cost and measured scaling. The structural-check implementation has linear construction cost $O(N)$, and the measured data are consistent with linear scaling over the tested range. Specifically, $T_{\mathrm{structural\ check}}(N)$ grows from 4.49 μs at $N = 4$ to 239.5 μs at $N = 256$ – a 53.3× increase for a 64× increase in N. At $N = 256$, $T_{\mathrm{DRCC,total}} = 241.2\,\mu$s, more than 21,000× lower than the classical runtime measured at $N = 11$ (5.1 s, the largest classical reference size tested – $N = 256$ and $N = 11$ are not the same instance size, and no direct same-N speedup is claimed here). The fully costed DRCC verification within the defined structural representation stays at microsecond scale over the tested range, including the explicitly measured structural-check cost.

Finding: comparison against the classical reference. Including the previously excluded structural-check cost lowers the reported speedup at $N = 11$

Table 1: Corrected speedup $G_{\text{time}} = T_{\text{classic}}(N)/T_{\text{DRCC,total}}(N)$, including the previously excluded structural-check cost.

N	T_{classic} (s)	$T_{\text{DRCC,total}}$ (s)	G_{time} (corrected)
4	0.000183	0.00000662	28
6	0.001134	0.00000753	151
8	0.027436	0.00000977	2,807
9	0.134587	0.00001059	12,706
10	0.758652	0.00001180	64,316
11	5.102105	0.00001218	418,884

from DRCC-V3.0's certificate-only $G_{\text{time}} \approx 2.5 \times 10^6$ to $G_{\text{time}} \approx 4.2 \times 10^5$ – roughly a sixfold reduction in the reported ratio. This is expected and is the point of the correction: the certificate-only figure omitted a real, now-measured cost. $G_{\text{time}} \approx 4.2 \times 10^5$ at $N = 11$ remains a substantial advantage over the exhaustive classical baseline.

2.4 Boundary and scope

Real SPICE netlist reference. For the full-adder experiments, an actual SPICE deck for the 1BitFA cell from the 16BitAdder library (Electric VLSI, MOSIS mocmos process) was used. The supplied deck contained the hierarchical subcircuits and testbench. The experiment constructs a purpose-built N-cell representation from the supplied SPICE description and checks its declared fields against the specified full-adder wiring pattern and the eight-entry full-adder truth table.

Consequently, this check operates as a representation-level verification of the fields actually constructed by the implementation; it does not include end-to-end parsing, external-netlist file I/O, or independent verification of an external SPICE or gate-level netlist file. The reported $\Theta(N)$ scaling therefore characterizes this defined structural-check operation, while its absolute constant factor is specific to the present Python implementation.

3 Q2: SAT-Solver Comparison – DRCC Symmetry-Breaking Against CDCL

The comparison examines where explicit DRCC symmetry-breaking affects solver behavior relative to established CDCL implementations, and records the largest tested formula size under the stated timeout and configuration.

3.1 Setup

This section reports a real, executed comparison against two established CDCL implementations, MiniSat 2.2 and Glucose 3. Instances are PHP_n^{n-1} (Pigeonhole Principle: n pigeons, $n-1$ holes, classically unsatisfiable), with variable $x_{i,j}$ denoting "pigeon i sits in hole j". Two clause sets are used per instance size n: a base CNF (every pigeon sits in at least one hole; no hole holds two pigeons), and the same base CNF extended with an explicit DRCC symmetry-breaking clause set DRCC, which restricts pigeon i to holes $1, \ldots, \min(i, n-1)$ – exploiting

the fact that any two assignments related by a permutation of the pigeons are equivalent with respect to the unsatisfiability question.

Six configurations are measured per n: a from-scratch reference DPLL solver (unit propagation and chronological backtracking, no clause learning) with and without DRCC, and MiniSat 2.2 [4] / Glucose 3 [5] (via the `python-sat` [6] bindings) with and without DRCC. Each configuration is run three times; reported values are medians. All three solver implementations receive the identical base CNF for a given n. In the DRCC configurations, the explicitly defined clause set ζ_n is added to that same base CNF; no other solver or execution parameter is changed. For the formal comparison, the DRCC-augmented instance is represented as

$$P_n^{\mathrm{DRCC}} := P_n \wedge \zeta_n,$$

where ζ_n denotes the explicitly supplied DRCC symmetry-breaking clause set. This notation distinguishes the original PHP instance P_n from the structurally augmented instance supplied to the solver.

Timeout. A wall-clock timeout of 120 seconds is enforced per solver invocation. This value is an engineering choice made for this experiment and is not derived from any theoretical threshold; it is stated here explicitly rather than implied as a principled cutoff. For the reference DPLL solver the timeout is enforced by an internal time check inside the search loop; for MiniSat 2.2 and Glucose 3, each invocation runs in an isolated subprocess that is terminated externally on timeout, since the underlying solver call cannot be interrupted from within the calling process once started.

Execution environment. Python 3.14.7, macOS 26.5.2 (arm64, Apple Silicon), `python-sat` 1.9.dev15, fixed seed 20260819 (unused by the deterministic solvers reported here, retained for logging consistency).

3.2 Results

Table 2 reports median wall-clock time in seconds for all six solver configurations across $n = 2$ through $n = 12$ and $n = 50$. A value of 120.00 marked with * denotes that all three repetitions of that configuration reached the timeout without returning a result; the true value is unknown and is at least 120 s, not exactly 120 s.

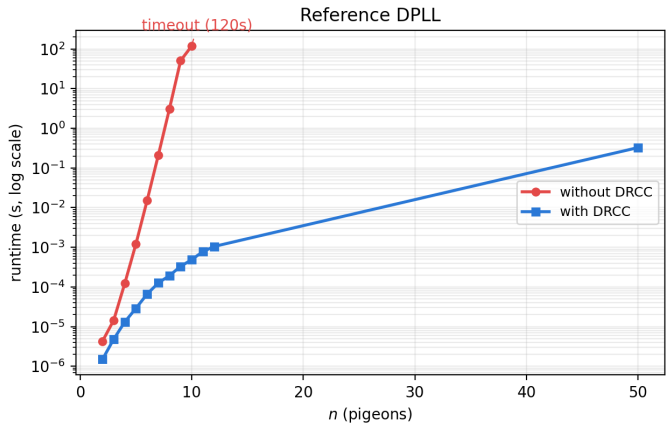


Figure 2: Reference DPLL: runtime without DRCC (red) reaches the 120 s timeout ceiling at $n = 10$ and is not plotted beyond that point; with DRCC (blue) it remains solved through $n = 50$. **Curve-transition feature** (see Section 3.3 and Table 8 for the distinction between formal runtime transition and visible curve transition): the DRCC runtime curve exhibits a visible change in its growth regime; no formal W is observed, since $G_{\text{time}}(n) > 1$ already at the smallest tested $n = 2$.

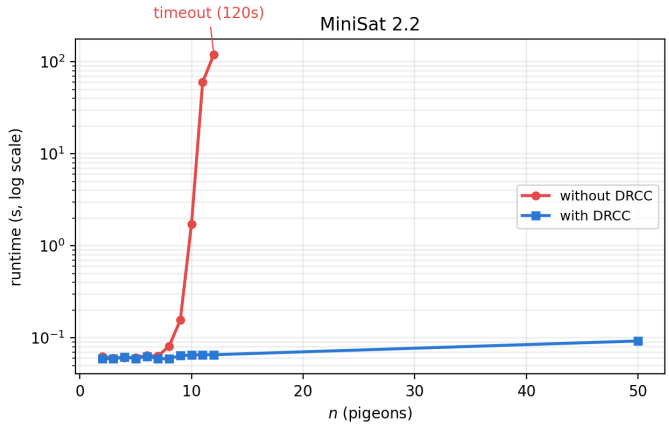


Figure 3: MiniSat 2.2: runtime without DRCC reaches the timeout ceiling at $n = 12$; with DRCC it remains solved through $n = 50$. **Transition interval:** a formal $W_{\text{int}} = [4, 5]$ exists ($G_{\text{time}}(4) = 0.976 < 1 < 1.025 = G_{\text{time}}(5)$), but both values lie within the subprocess-overhead/noise band (0.059–0.065 s, Section 3.1); it is not interpreted as the practically relevant onset, which occurs near $n = 8$.

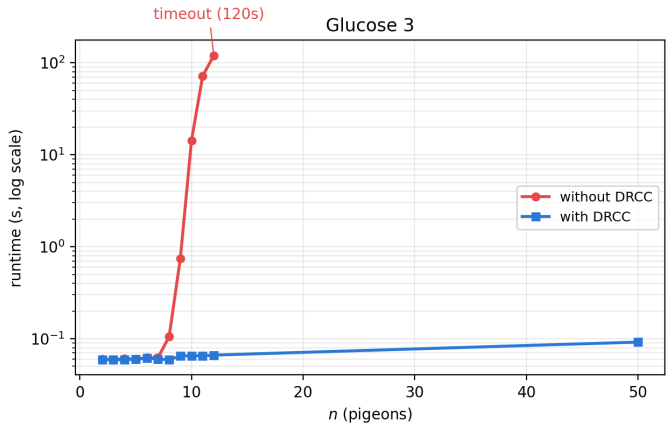


Figure 4: Glucose 3: runtime without DRCC reaches the timeout ceiling at $n = 12$; with DRCC it remains solved through $n = 50$. **Curve-transition status** (see Section 3.3 and Table 8 for the distinction between formal runtime transition and visible curve transition): no formal W or W_{int} is observed, since $G_{\text{time}}(n)$ never falls below 1 (minimum $G_{\text{time}}(3) = 1.003$).

Table 2: Median wall-clock runtime (seconds), PHP_n^{n-1}, $n \in \{2, \ldots, 12, 50\}$. * denotes a timeout ceiling (true value ≥ 120 s, not an exact measurement).

n	DPLL	DPLL+DRCC	MiniSat	MiniSat+DRCC	Glucose	Glucose+DRCC
2	0.0000043	0.0000015	0.0626	0.0593	0.0601	0.0591
3	0.0000144	0.0000048	0.0602	0.0594	0.0592	0.0590
4	0.000124	0.0000132	0.0607	0.0622	0.0607	0.0591
5	0.001225	0.0000290	0.0610	0.0595	0.0599	0.0596
6	0.0152	0.0000665	0.0642	0.0631	0.0621	0.0613
7	0.2094	0.000126	0.0635	0.0595	0.0621	0.0601
8	3.0967	0.000194	0.0813	0.0593	0.1055	0.0591
9	50.65	0.00033	0.1565	0.0642	0.7464	0.0647
10	120.00*	0.00049	1.7278	0.0649	14.259	0.0654
11	120.00*	0.00078	60.132	0.0657	71.773	0.0650
12	120.00*	0.00105	120.00*	0.0656	120.00*	0.0663
50	120.00*	0.3268	120.00*	0.0925	120.00*	0.0918

Finding (Q2). Across the tested sizes, the DRCC-augmented configurations reported zero branching decisions in the recorded runs. MiniSat 2.2 and Glucose 3 also reported zero conflicts and zero restarts in these configurations. Thus, for the tested PHP instances and solver settings, the default clause-learning and restart mechanisms did not eliminate the need for the explicitly supplied symmetry-breaking clauses.

Finding: the crossover region (Figures 3 and 4, MiniSat and Glucose only – not Figure 2/DPLL). From $n = 2$ through $n = 7$, the wall-clock effect of DRCC on MiniSat and Glucose is not resolvable from subprocess-invocation overhead (all values in the band 0.059–0.065 s regardless of DRCC);

the instances are simply too small for the underlying solving time to exceed this fixed overhead. A first measurable separation appears at $n = 8$ (Glucose: 0.1055 s vs. 0.0591 s, a factor of 1.8), and the divergence becomes pronounced from $n = 9$ onward. The transition therefore lies between $n = 7$ and $n = 9$, with $n = 8$ as the first point at which the effect is measurable above the subprocess-overhead floor described above.

Implementation-level runtime growth. The remaining growth of the DRCC-augmented DPLL runtime is an implementation-level effect in the present reference solver. In particular, unit propagation repeatedly rescans a growing clause set during reconstruction, producing substantial clause-processing overhead. This measured behavior should not be interpreted as a proof of an underlying exponential complexity of the DRCC construction. Resolution lower bounds provide theoretical context for the PHP benchmark but are not used here as a causal explanation of the measured implementation runtime. No specific extrapolated runtime for n substantially beyond 50 is reported here: the observed non-linear growth of the DRCC-reduced branch makes a simple proportional extrapolation unreliable, and no such value was executed.

The largest tested instance was $n = 50$. At this size, all three DRCC-augmented configurations completed within the 120-s timeout, whereas the corresponding configurations without DRCC reached the timeout in all three repetitions. Here, the notation 120^* denotes that all three repetitions reached the 120-s timeout; the true runtime is therefore unknown beyond the timeout threshold.

3.3 Timeout matrix

Table 3 reports, for each solver configuration, the largest tested n at which the configuration still returned a result within the timeout, and the first tested n at which a timeout occurred.

Table 3: First-timeout matrix across tested $n \in \{2, \ldots, 12, 50\}$.

Configuration	Largest solved n	First timeout at n
DPLL	9	10
DPLL+DRCC	50 (none)	–
MiniSat	11	12
MiniSat+DRCC	50 (none)	–
Glucose	11	12
Glucose+DRCC	50 (none)	–

$$G_{\text{time}}(n) = \frac{T_{\text{Classic}}(P_n)}{T_{\text{DRCC}}(P_n, \zeta_n)} \tag{2}$$

$$W = (n_w, R_w[\text{s}]), \quad \text{where } G_{\text{time}}(n_w) = 1, \\ R_w = T_{\text{Classic}}(P_{n_w}) = T_{\text{DRCC}}(P_{n_w}, \zeta_{n_w}) \tag{3}$$

$$W_{\text{int}} = [n_-, n_+], \quad \text{where } G_{\text{time}}(n_-) < 1 < G_{\text{time}}(n_+) \quad \text{(used when no exact } n_w \text{ is observed)} \quad (4)$$

Because the tested problem-size parameter n is integer-valued, an interval such as $W_{\text{int}} = [4, 5]$ denotes a *discrete runtime crossover interval*, not a claim that every real-valued n between 4 and 5 is tested.

Relation to the DRCC-V2 transition-point definition. DRCC-V2-REW2, Chapter 10.6 [1] defines $W = (n_w, R_w[\text{s}])$ via $G_{\text{time}}(n_w) = 1$, but reports its own SAT example only at evidence level S (structural), not E (empirical) – no value of W was inferred there. This section supplies, to our knowledge for the first time for the Pigeonhole/CDCL setting, an empirical evidence-level-E instantiation. For MiniSat, $G_{\text{time}}(n)$ fluctuates around 1 for $n = 2$ through 7 (values from 0.976 to 1.067, consistent with subprocess-overhead noise rather than a genuine signal), then rises monotonically and decisively above 1 from $n = 8$ onward ($G_{\text{time}}(8) = 1.371$, $G_{\text{time}}(9) = 2.438$). No single exact crossing n_w with $G_{\text{time}}(n_w) = 1$ is identified in the strict sense of Equation (3), since the pre-$n = 8$ values oscillate on both sides of 1 without a clean monotonic approach; $n = 8$ is instead reported as the practical onset of the effect, consistent with the crossover-region finding above.

Scope and boundary. The comparison is deliberately focused on the Pigeonhole family, where the Pigeonhole encoding provides a direct setting for the explicitly defined symmetry-breaking rule τ_{PHP}. MiniSat 2.2 and Glucose 3 are evaluated through the version-specific `python-sat` configurations used in this experiment, while DRCC is added explicitly as the same symmetry-breaking clause set to the corresponding base CNF. In the tested configurations, the DRCC-augmented MiniSat 2.2 and Glucose 3 runs reported zero branching decisions, zero conflicts, and zero restarts at every tested size. This statement concerns the recorded configurations and runs; it does not establish a universal absence of branching in DRCC. DRCC is evaluated here as an explicit structural symmetry-breaking augmentation of existing solver configurations, not as a replacement SAT-solving engine. The observed advantage is benchmark- and configuration-specific and should not be interpreted as evidence that DRCC universally outperforms CDCL solvers.

4 Q3: The 56 Counterexamples – Generation and Structure

The ordering study evaluates 1,200 generated constraint-structure instances and records the observed boundary of the tested one-step DRCC-FG greedy ordering rule.

4.1 Motivation

This section answers what the underlying raw data (`exp27_adversarial_frontier_stress.json`) supports directly regarding Q3, and states plainly

which part of the question it does not resolve.

4.2 What the raw data establishes

The search covers 1,200 generated constraint-structure instances (fixed seed 20260819), drawn from five named instance families: `two_blocks`, `random_dense`, `mixed`, `band_chords`, and `hub_bridge`. For each instance, DRCC-FG's peak frontier width is compared against the best of three classical heuristics (Min-Fill, Min-Degree, MCS); a *counterexample* is any instance where DRCC-FG's width exceeds the best classical width (a positive gap). 56 of 1,200 instances (4.67%) are positive-gap counterexamples – independently re-verified here by summing the reported gap distribution (Table 4), which matches the reported count exactly.

Table 4: Distribution of gap = $\text{width}_{\text{DRCC}} - \text{width}_{\text{best classical}}$ across all 1,200 searched instances. Negative values mean DRCC-FG outperformed every classical heuristic tested.

gap	−8	−7	−6	−5	−4	−3	−2	−1	0	1	2	3	4
count	1	2	9	29	69	150	259	380	245	44	9	1	2

Finding (Q3). DRCC-FG matches or beats every classical heuristic tested on 1,144 of 1,200 instances (95.3%); the gap distribution is heavily skewed toward negative values (DRCC-FG narrower), with the positive tail – the reported counterexamples – comprising a small minority.

Instance Selection and Tie-Breaking. The top-25 instances were selected by ranking the observed positive gaps between the DRCC-FG width and the best classical width. No strict tie-breaking rule had been predefined before the experiments; instances with identical primary gaps were therefore resolved ad hoc for this descriptive inspection.

The five instance families are represented as follows: `two_blocks` (9), `random_dense` (8), `mixed` (6), `band_chords` (1), and `hub_bridge` (1).

A recorded counterexample. Instance 487 belongs to the `two_blocks` family and contains 22 variables and 33 constraint scopes. The recorded DRCC-FG ordering produces a peak frontier width of 15, whereas both Min-Degree and MCS produce a peak width of 11, yielding a positive gap of 4. The underlying experiment record contains the complete constraint-scope list and the variable ordering produced by each of the four heuristics, so that the reported widths can be independently re-derived. The available record does not provide a complete per-stage B_i or ω_i trace, nor a corresponding runtime measurement for this individual instance. Accordingly, no stronger causal interpretation of the width gap in terms of runtime is made here. No claim of global optimality is made for the tested one-step DRCC-FG ordering rule.

4.3 Scope of evidence

The EXP.20 and EXP.27 experiment records provide the documented basis for the reported instance structures, frontier-width measurements, and ordering comparisons. The recorded data permit direct re-derivation of the reported counterexamples, including the complete scope and ordering information for recorded counterexample instance 487 (confirmed exactly against the raw data). The resulting evidence is therefore reproducible at the level of the recorded instance data and directly supports the reported 56 positive-gap cases.

The 56 cases identify a specific and experimentally observed boundary of the tested one-step DRCC-FG greedy ordering rule. They do not need to be interpreted as a limitation of DRCC as a whole: alternative DRCC-compatible ordering strategies constitute distinct algorithmic choices and are outside the present Q3 test. Importantly, a positive width gap is a structural ordering result and does not imply a corresponding runtime disadvantage. Among the 12 counterexamples with construction-time measurements, DRCC construction is faster than the classical alternative in 6 cases.

The generator families and their realized instances are represented in the experiment records themselves, with the realized scopes, widths, and orderings documented and internally consistent. Code-level re-execution of the family generators is a separate reproducibility pathway and is not required to re-derive the reported widths from the recorded instance data. The manuscript therefore distinguishes the validity and traceability of the recorded experimental evidence from the availability of external generator source code.

Ablation. The present Q3 evidence is based on the recorded instance-wise measurements. No separate repeated-measurement ablation was used for this ordering study; accordingly, the Q3 conclusion is restricted to the observed width comparisons and does not rely on an extrapolated noise model.

5 Q4: Prospective Test of the Reconstruction-Width Formalism

The reconstruction-width formalism is evaluated prospectively: the structural quantities are derived before the corresponding runtime measurements, and the resulting runtime hypotheses are frozen before testing.

5.1 Purpose and pre-specified prediction

$n = 13$ was fixed as the original pre-specified test instance *before* it was measured, and was not used anywhere in deriving the formulas below. Both runtime hypotheses in Section 5.3 were frozen prior to running that experiment. Two further points, $n = 14$ and $n = 15$ (Section 5.4), were subsequently fixed and measured in the same manner, against the identical frozen parameters (a, p) , with no refitting between $n = 13$, 14, and 15 , turning the original single pre-specified point into a three-point prospective series. At no stage were the frozen hypotheses adjusted after seeing a result.

5.2 Formal structural prediction

For PHP_n^{n-1} with the DRCC symmetry-breaking clause set $S_{\mathrm{sym}} := \zeta_n$ (pigeon i restricted to holes $1, \ldots, \min(i, n-1)$), processed in pigeon-block order $\pi = (1, 2, \ldots, n)$, the active interface is defined explicitly as follows: **a processed variable belongs to $B_i^{\pi,\tau}$ exactly when it still shares a constraint scope with at least one unprocessed variable**. This is the operational definition used for the PHP-CNF in the present manuscript. It is a concrete instantiation of the general DRCC principle that retained information is active when it can still influence an admissible continuation or the declared output.

Transferability of this operationalization. The local-factor-scope criterion is applicable to constraint-hypergraph representations: a processed variable can continue to constrain an admissible continuation through a constraint that it shares with an unprocessed variable. What transfers from DRCC-V3.0's Experiment 20 is this criterion, not its numerical frontier width or processing order. Experiment 20's reported $\max|\text{frontier}| = 2$ is specific to that experiment's narrow constraint structure. Conversely, the $\omega_{\mathrm{DRCC}} = \Theta(n^2)$ result derived below is specific to the present PHP constraint hypergraph and is not claimed as a general CSP bound.

What transfers from Experiment 20 is level (2), the criterion itself – not any numeric result. Experiment 20's own frontier width ($\max|\text{frontier}| = 2$) is specific to its narrow, chain-like constraint structure and is not claimed to hold for PHP or any other problem; conversely, $\omega_{\mathrm{DRCC}} = \Theta(n^2)$ for PHP+S_{sym} (proven below) is not claimed to generalize back to Experiment 20's problem or to CSPs in general. Nor does Experiment 20's specific processing order transfer – the pigeon-block order used here is independently justified by matching PHP's own symmetry-breaking structure. The justified claim is therefore narrow: the same general, encoding-independent criterion for identifying the active interface, already used and verified in DRCC-V3.0, was applied here to a different constraint system, and its consequences for that system (closed-form ω_{DRCC}, N_i^{reach}, W_{DRCC}) are derived independently below, not inherited from Experiment 20's own numbers.

Under this realization, the following proposition gives the exact closed form, not merely an empirical value.

Proposition. For $1 \le i \le n-1$,

$$|B_i^{\pi,\tau}| = \sum_{k=1}^{i} \min(k, n-1) = \sum_{k=1}^{i} k = \frac{i(i+1)}{2}. \tag{5}$$

Proof. A processed variable $x_{k,j}$ ($k \le i$) is active iff it shares a constraint scope with an unprocessed variable. Two factor types contain $x_{k,j}$:

(a) Pigeon-k factor ("at least one hole"), scope $\{x_{k,1}, \ldots, x_{k,\min(k,n-1)}\}$. For $k \le i$ this scope is entirely processed (pigeon-block order), so it contributes no activity.

(b) Hole-j factor ("at most one pigeon"), connecting $x_{k,j}$ to $x_{k',j}$ for every pigeon k' that can also use hole j, i.e. every k' with $j \le \min(k', n-1)$, equivalently $k' \ge j$. Pigeon n satisfies $\min(n, n-1) =$

$n-1 \geq j$ for every $j \leq n-1$, so pigeon n can contest *every* hole. Hence, for any $i < n$, pigeon n is an unprocessed pigeon sharing a hole-j factor with every already-processed $x_{k,j}$ ($k \leq i$) – so *every* processed variable remains active as long as $i < n$.

Consequently $|B_i^{\pi,\tau}|$ equals the total number of variables processed through stage i: $\sum_{k=1}^{i} \min(k, n-1)$. For $i \leq n-1$, $\min(k, n-1) = k$ for every $k \leq i$, giving $\sum_{k=1}^{i} k = i(i+1)/2$. ■

Taking the maximum over i (attained at $i = n-1$, the last stage before pigeon n's factors are fully processed and all interface variables discharge at once):

$$\omega_{\mathrm{DRCC}}(P, \pi, \tau) = \max_{0 \leq i \leq n} |B_i^{\pi,\tau}| = |B_{n-1}^{\pi,\tau}| = \frac{(n-1)n}{2} = \Theta(n^2). \tag{6}$$

This closed form was independently verified against a direct computational enumeration of the factor-scope graph for $n \in \{4, 6, 8, 10\}$ (giving $6, 15, 28, 45$ respectively, matching $(n-1)n/2$ exactly in every case) before being used for the $n = 13$ prediction below.

Separately, a direct induction on the forced hole assignment (pigeon $k \mapsto$ hole k for $k < n$; pigeon n forced infeasible) shows exactly one reachable state exists at every stage:

$$N_i^{\mathrm{reach}} = \begin{cases} 1, & i = 0, \ldots, n-1, \\ 0, & i = n. \end{cases} \tag{7}$$

Because the final pigeon cannot be assigned to any of the $n-1$ holes, the final stage contributes no reachable reconstruction state. The uniqueness claim $N_i^{\mathrm{reach}} = 1$ relies on a deterministic reconstruction choice; it does not follow from the domain restriction $j \leq \min(i, n-1)$ alone. For the concrete rule used here,

$$x_{k,k} = \top, \qquad x_{k,j} = \bot \quad (j \neq k,\ 1 \leq j \leq k). \tag{8}$$

Therefore

$$W_{\mathrm{DRCC}}(P, \pi, \tau) = \sum_{i=0}^{q} N_i^{\mathrm{reach}} = n = \Theta(n). \tag{9}$$

For $n = 13$ specifically:

$$\begin{aligned} \omega_{\mathrm{DRCC}}(13) &= \frac{13 \cdot 12}{2} = 78, \\ N_i^{\mathrm{reach}} &= 1 \quad \forall i = 0, \ldots, 12, \\ N_{13}^{\mathrm{reach}} &= 0, \\ W_{\mathrm{DRCC}}(13) &= 13. \end{aligned} \tag{10}$$

5.3 Two frozen runtime hypotheses

Two competing numeric predictions for $T_{\mathrm{DRCC}}(13)$ (the reference-DPLL-with-DRCC wall-clock time) were fixed before measurement.

Why the theory-pure model is a genuine hypothesis, not a strawman. $W_{\mathrm{DRCC}} = \Theta(n)$ is formally derived (Section 5.2), and the implementation

under test executes exactly the reconstruction process that W_{DRCC} counts: at each of the n pigeon-processing stages, one reconstruction step is performed. A runtime hypothesis proportional to W_{DRCC} is therefore the most direct, structure-first prediction available – not one selected to under-perform. Its rejection is itself informative: it isolates a real, additional cost (per-step clause-rescanning in this from-scratch implementation) that the width formalism does not and is not meant to capture, rather than indicating the hypothesis was chosen to fail.

- **Theory-pure:** assuming wall-clock time scales proportionally with W_{DRCC} alone, extrapolating the ratio $T_{\text{DRCC}}(n)/W_{\text{DRCC}}(n)$ observed at $n = 6, \ldots, 12$ gives $T_{\text{theory}}(13) \approx 0.000552\,\text{s}$.
- **Empirical:** a power-law fit

$$T(n) = a\, n^p \tag{11}$$

to the same seven data points ($n = 6, \ldots, 12$), by ordinary least squares on $\log T$ vs. $\log n$, with no reference to W_{DRCC} at all, gives the parameter estimates

$$\begin{aligned} a &= 5.056 \times 10^{-8}, \\ p &= 4.0002 \pm 0.0720 \text{ (std. error)}, \\ R^2 &= 0.9984. \end{aligned}$$

with 5 degrees of freedom and a 95% confidence interval for the exponent of $[3.815, 4.185]$. Propagating this uncertainty in p (holding a fixed) gives a point prediction $T_{\text{empirical}}(13) \approx 0.001445\,\text{s}$ with a sensitivity range of $[0.000899, 0.002323]\,\text{s}$ obtained by propagating only the uncertainty in p while holding a fixed.

Both are stated here as predictions about *this Python implementation's wall-clock time*, not as claims about DRCC's structural complexity, which Section 5.2 already establishes independently as $W_{\text{DRCC}} = \Theta(n)$.

5.4 Prospective test series: $n = 13, 14, 15$

Two further, previously untested points ($n = 14, 15$) were measured after $n = 13$, using the frozen fit parameters ($a = 5.056 \times 10^{-8}$, $p = 4.0002$) from Section 5.3 *without any refitting*. This converts the original single-point test into a small prospective series.

At every one of the three points, T_{emp} falls inside the measured 95% confidence interval. The reported interval is a confidence interval for the measured mean, not a full prediction interval for a future run. The three relative deviations (-8.9%, -7.3%, -7.7%) are of consistent sign and similar magnitude, with no outlier – the measured values sit systematically slightly below the frozen power-law prediction, but by a stable margin rather than an erratic one.

For MiniSat22+DRCC and Glucose3+DRCC at $n = 14, 15$: 0 decisions, 0 conflicts throughout (propagations 176–209), consistent with $N_i^{\text{reach}} = 1$ at these points as well, though these two solvers' wall-clock times remain dominated by fixed subprocess overhead (as established in Section 3) and were not the target of the $T(n) = a\, n^p$ hypothesis, which concerns $T_{\text{DPLL+DRCC}}$ specifically.

Table 5: DPLL+DRCC results across the full prospective test series ($n = 13, 14, 15$), $\bar{T}$ = mean over 3 runs, s = sample standard deviation, 95% CI via t-distribution (df = 2). T_{emp} is the frozen power-law prediction $a\,n^p$, computed before each respective measurement and never adjusted afterward. All three baseline configurations (DPLL, MiniSat22, Glucose3, without DRCC) reached the 120 s timeout in all 3 runs at every n in this table.

n	$\bar{T}$ (s)	s (s)	CV	95% CI (s)	T_{emp} (s)	Δ
13	0.001321	0.000102	7.72%	[0.001068, 0.001575]	0.001445	−8.9%
14	0.001801	0.000076	4.24%	[0.001611, 0.001991]	0.001943	−7.3%
15	0.002363	0.000105	4.43%	[0.002103, 0.002623]	0.002561	−7.7%

5.5 Prediction assessment

At $n = 13$: the theory-pure runtime prediction is 0.000552 s, which lies **outside** the measured 95% confidence interval ($[0.001068, 0.001575]$ s) and is therefore **inconsistent with the measured mean under the pre-specified interval-inclusion criterion**. The empirical prediction (0.001445 s) lies **inside** the interval and is therefore **consistent with the measurement**. The same empirical model, applied without refitting at $n = 14$ and $n = 15$, is again consistent with the respective intervals (Table 5). These results do not constitute a formal statistical confirmation of the model; they report agreement with the measured intervals under the pre-specified comparison. At all three prospective points, the recorded runs were consistent with $N_i^{\text{reach}} = 1$: the DRCC-augmented configurations reported zero decisions throughout.

This is stated as three prospective test points, not as a general proof that the empirical power law holds for all n; three consistent points support the model but do not establish it asymptotically. The formal proof that $W_{\text{DRCC}} = \Theta(n)$ (Section 5.2) is a separate, general result, established by induction for all n, and does not depend on how many points the empirical implementation-runtime model has been tested against.

5.6 Structural Work versus Implementation-Level Empirical Runtime

Structure:	$\omega_{\text{DRCC}} = \Theta(n^2)$	active-interface capacity (proven)
Reconstruction:	$W_{\text{DRCC}} = \Theta(n)$	structural work (proven)
Implementation:	$T_{\text{measured}}(n)$	empirical runtime scaling ($\approx n^4$ over the tested range)

The measured runtime overhead is consistent with an empirical approximately n^4 scaling over the evaluated interval. This approximately fourth-power behavior is a phenomenological description of the present implementation over that interval, not a proven asymptotic complexity bound or a general DRCC runtime law. The likely proximate cause is that `unit_propagate` rescans all $O(n^2)$ clauses at each of the $O(n)$ reconstruction steps – a property of this from-scratch implementation, not of the reduction itself. A watched-literal implementation could change this implementation-level runtime behavior, but that possibility was not evaluated in the present study.

5.7 Prediction assessment summary

The purpose of the prospective test is not exact prediction of a specific millisecond value. Instead, the structural quantities defined in Section 5.2 and the two runtime hypotheses of Section 5.3 were fixed before measurement. Three previously untested cases ($n = 13, 14, 15$) were then measured against those frozen hypotheses without post-hoc adjustment or refitting. The theory-pure prediction was not supported at $n = 13$ because it lies outside the measured 95% confidence interval, whereas the empirical prediction was consistent with the interval at all three points. The observed deviations (-8.9%, -7.3%, -7.7%) have the same sign and similar magnitude. This is a small prospective test series, not a proof that the empirical power law holds asymptotically.

6 DRCC: Finite Mathematics and Algorithmic Core

6.0 Introduction and Motivation

Many practically relevant SAT instances exhibit substantial structural regularities, such as symmetries, repeated substructures, or task-specific constraints that are not fully exploited by generic SAT solvers. Standard DPLL- and CDCL-based solvers operate directly on the original CNF formula and may explore exponentially large search trees, even when the underlying structure admits a much simpler representation.

DRCC (Dimensional Reduction via Controlled Combinatorics) addresses this gap by introducing a *structural pre-processing phase* that precedes the solver's branching search. The central idea is to transform the input formula F into a reduced formula F_{Red} that can expose contradictions or satisfying assignments more directly. Satisfiability preservation is required only when the concrete reduction satisfies the condition stated in Section 6.2; it is not assumed for an arbitrary reconstruction rule. Formally:

$$F \xrightarrow{\mathrm{DRCC}_{\mathrm{Red}}(\pi,\tau)} F_{\mathrm{Red}} \xrightarrow{\mathrm{Solver}} \{\mathrm{SAT}, \mathrm{UNSAT}\}. \tag{12}$$

The DRCC reduction is parameterized by:

- a processing order π, which determines the sequence in which variables or structural units are processed,
- a reconstruction rule τ, which encodes task-specific structural constraints (e.g., symmetry-breaking conditions, ordering constraints, or domain restrictions).

By fixing (π, τ) before the reduction starts, DRCC performs a controlled, deterministic transformation of the problem structure.

Two quantitative measures are introduced to characterize the DRCC reduction:

- **DRCC width:**

$$\omega_{\mathrm{DRCC}} = \max_{1 \le i \le q} |B_i^{\pi,\tau}|, \tag{13}$$

where $B_i^{\pi,\tau}$ is the active interface after processing step i. This measures the peak structural capacity required during reduction.

- **DRCC work:**

$$W_{\mathrm{DRCC}} = \sum_i N_i^{\mathrm{reach}}, \tag{14}$$

where N_i^{reach} is the number of reachable reconstruction states at stage i. This measures the total number of distinct partial reconstructions that are reachable across all stages. This is a structural-combinatorial work measure, not a machine-level operation count or a wall-clock runtime.

A key insight is that these two quantities can behave very differently: a large interface (ω_{DRCC}) does not necessarily imply large reconstruction work (W_{DRCC}) if the reconstruction rule τ is sufficiently restrictive.

The pigeonhole principle PHP_n^{n-1} serves as a canonical example. For a specific DRCC reconstruction rule τ_{PHP}, we obtain $\omega_{\mathrm{DRCC}} = \Theta(n^2)$ but $W_{\mathrm{DRCC}} = \Theta(n)$, demonstrating that DRCC can reduce the effective search space even when the active structural interface is large.

This chapter develops the finite-mathematical foundation of DRCC, defines the key quantities ω_{DRCC} and W_{DRCC}, and establishes the algorithmic step chain that connects the input formula F to the reduced formula F_{Red} and ultimately to the solver's output.

6.1 Basic objects and notation

In this section we fix the basic mathematical objects on which the entire DRCC framework is built. All structures are finite and combinatorial; no analytic or continuous notions are required.

6.1.1 CNF structure

Let

$$F = (V, C) \tag{15}$$

be a finite CNF (conjunctive normal form) structure with

- a finite set of Boolean variables

$$V = \{x_1, x_2, \ldots, x_N\}, \tag{16}$$

- a finite set of clauses

$$C = \{C_1, C_2, \ldots, C_m\}, \tag{17}$$

where each clause C_j is a finite disjunction of literals over V.

A *literal* is either a variable x_i or its negation $\neg x_i$. We identify each clause with the set of its literals and the CNF formula F with the set of its clauses. Thus, F can be viewed equivalently as:

- a propositional formula in CNF,
- a finite set of finite sets of literals.

6.1.2 Processing order

DRCC operates with a discrete processing order

$$\pi, \tag{18}$$

which specifies the sequence in which variables or structural units are processed. Typical instances of π include:

- a permutation of the index set $\{1, \ldots, N\}$, inducing an order on the variables $(x_{\pi(1)}, x_{\pi(2)}, \ldots, x_{\pi(N)})$,
- a more general schedule over structural units (e.g., blocks of variables, clause groups, or circuit components).

The order π is fixed before the DRCC reduction starts and remains constant throughout the reduction phase.

6.1.3 Reconstruction rule

In addition to the processing order, DRCC uses a declared task or reconstruction rule

$$\tau, \tag{19}$$

which determines which structural constraints are enforced during the reduction. The rule τ may specify, for example:

- domain restrictions for variables (e.g., pigeon i may only use holes $1, \ldots, \min(i, n-1)$ in the pigeonhole principle),
- symmetry-breaking constraints that select canonical representatives from equivalence classes of assignments,
- task-specific structural conditions that encode semantic properties of the problem (e.g., ordering constraints, monotonicity conditions, or interface restrictions).

Like π, the rule τ is fixed before the reduction starts and remains constant throughout the DRCC phase.

6.1.4 Parameter pair

The pair

$$(\pi, \tau) \tag{20}$$

fully determines the DRCC reduction strategy for a given CNF structure F. Different choices of (π, τ) lead to different reduced structures F_{Red}, different active interfaces $B_i^{\pi,\tau}$, and different values of ω_{DRCC} and W_{DRCC}.

In the remainder of this chapter, (π, τ) is treated as fixed unless explicitly stated otherwise.

6.2 DRCC Reduction as a Structural Projection

The central DRCC operation is a structural projection that transforms the input CNF structure F into a reduced structure F_{Red}. To make the algorithmic structure explicit, we distinguish between individual DRCC steps and the overall reduction map.

For the generic algorithm, let $q := |\pi|$ denote the number of processing units in the chosen schedule. In the variable-by-variable case with n variables, $q = n$. All generic stage indices below therefore run from 1 to q; the PHP specialization later sets $q = n$ explicitly.

6.2.1 Elementary DRCC Step

For each processing step $i = 1, \ldots, q$, DRCC applies an elementary reduction map to the complete DRCC state. Define

$$\mathcal{S}_0 = F, \qquad \mathcal{S}_i = \mathcal{R}_i(\mathcal{S}_{i-1}; \pi, \tau), \qquad i = 1, \ldots, q. \tag{21}$$

Here $\mathcal{S}_i$ denotes the complete DRCC state after processing stage i; it may contain the current CNF structure together with auxiliary reconstruction information. The parameters (π, τ) are fixed throughout the reduction and determine:

- which structural unit (variable, clause group, or other unit) is processed at step i (via π),
- which structural constraints are enforced during this step (via τ).

6.2.2 Overall DRCC Reduction Map

The complete DRCC reduction is the composition of all elementary steps:

$$\mathcal{R}_{\mathrm{DRCC}}(\cdot; \pi, \tau) \;=\; \mathcal{R}_q \circ \mathcal{R}_{q-1} \circ \cdots \circ \mathcal{R}_1. \tag{22}$$

Applied to the input formula $F = \mathcal{S}_0$, the final DRCC state is

$$\mathcal{S}_q = \mathcal{R}_{\mathrm{DRCC}}(F; \pi, \tau), \qquad F_{\mathrm{Red}} = \mathrm{CNF}(\mathcal{S}_q). \tag{23}$$

We emphasize the distinction:

$$\boxed{\text{DRCC step } \mathcal{R}_i \quad \text{vs.} \quad \text{overall reduction } \mathcal{R}_{\mathrm{DRCC}}.} \tag{24}$$

The elementary maps $\mathcal{R}_i$ describe the local transformation at each processing stage, while $\mathcal{R}_{\mathrm{DRCC}}$ describes the global transformation from F to F_{Red}.

6.2.3 Iterative Reduction Sequence

Iteratively, DRCC defines a sequence of complete intermediate states:

$$\mathcal{S}_0 = F, \tag{25}$$

$$\mathcal{S}_i = \mathcal{R}_i(\mathcal{S}_{i-1}; \pi, \tau), \qquad i = 1, \ldots, q, \tag{26}$$

with the solver-facing CNF component extracted as

$$F_{\mathrm{Red}} = \mathrm{CNF}(\mathcal{S}_q). \tag{27}$$

This stepwise formulation will be essential in Section 6.3 for defining the active interface $B_i^{\pi,\tau}$ and the DRCC width ω_{DRCC}, and in Section 6.4 for defining the reachable states N_i^{reach} and the DRCC work W_{DRCC}.

6.3 Active Interface and DRCC Width

This section introduces the active interface $B_i^{\pi,\tau}$ and the DRCC width ω_{DRCC}, building on the objects from Section 6.1 and the reduction map from Section 6.2.

6.3.1 Active Interface

After each processing step i, DRCC defines an *active interface*

$$B_i^{\pi,\tau}, \tag{28}$$

which captures the structural information that remains connected to the unprocessed part of the problem after the first i processing steps.

Formally, $B_i^{\pi,\tau}$ can be viewed as the set of boundary variables, constraints, or reconstruction states that interface between the processed prefix and the remaining suffix under (π, τ). The precise nature of $B_i^{\pi,\tau}$ depends on the specific DRCC reduction rule and problem structure; the definition is intentionally kept general to remain applicable across different problem classes.

The active interface evolves as DRCC proceeds. Starting from

$$\mathcal{S}_0 = F, \tag{29}$$

the iterative reduction sequence is given by

$$\mathcal{S}_i = \mathcal{R}_i(\mathcal{S}_{i-1}; \pi, \tau), \qquad i = 1, \ldots, q, \tag{30}$$

where $\mathcal{R}_i$ is the elementary DRCC step at stage i, as defined in Section 6.2. The final reduced structure is

$$F_{\mathrm{Red}} = \mathrm{CNF}(\mathcal{S}_q). \tag{31}$$

At each stage i, the interface $B_i^{\pi,\tau}$ quantifies how much structural information must be carried forward to correctly process the remaining suffix of the problem.

6.3.2 DRCC Width

The *DRCC width* is defined as the maximal active interface size over all processing steps:

$$\omega_{\mathrm{DRCC}} := \max_{1 \le i \le q} \left| B_i^{\pi,\tau} \right|. \tag{32}$$

This quantity measures the peak structural capacity required during the DRCC reduction. It depends on the input formula F, the processing order π, and the reconstruction rule τ; different choices of (π, τ) can yield different widths for the same F.

6.3.3 Graph-Theoretic Interpretation

In graph-theoretic terms, ω_{DRCC} is analogous to classical width parameters such as treewidth, pathwidth, or cutwidth. However, unlike these classical parameters, ω_{DRCC} is *task-aware*: it reflects not only the underlying constraint graph but also the specific reconstruction rule τ and processing order π imposed by DRCC.

It is important to note that ω_{DRCC} is not claimed to be equal to any classical width invariant (e.g., $\omega_{\mathrm{DRCC}} \neq \mathrm{treewidth}(G)$ in general). Instead, it should be understood as a *task-aware structural width* adapted to the DRCC reconstruction process.

6.3.4 Relation to DRCC Work

A key distinction is that DRCC width and DRCC work measure different aspects of the reconstruction process:

- $\omega_{\mathrm{DRCC}} = \max_{1 \leq i \leq q} |B_i^{\pi,\tau}|$ quantifies the *peak active structural capacity* required during reduction.
- $W_{\mathrm{DRCC}} = \sum_i N_i^{\mathrm{reach}}$ quantifies the *total reachable reconstruction work* over all stages.

These two quantities can behave very differently. For the Pigeonhole Principle family PHP_n^{n-1} under a suitable DRCC reconstruction rule τ_{PHP}, one obtains

$$\omega_{\mathrm{DRCC}} = \Theta(n^2), \qquad W_{\mathrm{DRCC}} = \Theta(n). \tag{33}$$

This illustrates that a large structural interface does not necessarily imply large reconstruction work when the reconstruction rule τ is sufficiently restrictive.

6.3.5 Relation to Solver Complexity

The DRCC width ω_{DRCC} does not directly bound the solver's search tree size or runtime. While a smaller ω_{DRCC} may indicate a more compact structural representation, the actual solver performance depends on additional factors, including the solver's heuristics, clause learning mechanism, and the specific structure of F_{Red}.

The separation between ω_{DRCC} (a structural measure of the DRCC reduction) and T_{Solver} (the solver's runtime) is intentional: DRCC changes the problem structure before the solver begins its branching search, but does not prescribe or bound the solver's internal behavior.

6.4 Reachable States and DRCC Work

This section introduces the number of reachable reconstruction states N_i^{reach} and the total DRCC work W_{DRCC}, building on the objects from Section 6.1, the reduction map from Section 6.2, and the active interface from Section 6.3.

6.4.1 Reachable Reconstruction States

For each processing step i, let $\Omega_i(F, \pi, \tau)$ denote the set of admissible partial reconstructions obtained from $\mathcal{S}_i$ after the reconstruction rule τ has been applied through stage i, with processing order π. The reachable-state count is then defined by

$$N_i^{\mathrm{reach}} := |\Omega_i(F, \pi, \tau)|\,. \tag{34}$$

Formally, a *partial reconstruction* at stage i is valid if it satisfies the following conditions:

- it is consistent with the reduced structure $\mathcal{S}_i$,
- it respects the structural constraints imposed by the reconstruction rule τ,
- it is compatible with the processing order π up to step i.

Equivalently, $\Omega_i(F, \pi, \tau)$ is the set of valid partial reconstructions compatible with $\mathcal{S}_i$ and (π, τ). The ordering is important: the count refers to states after the constraints specified by τ have been applied through stage i.

6.4.2 Iterative Reduction Sequence

Starting from

$$\mathcal{S}_0 = F, \tag{35}$$

the iterative reduction sequence is given by

$$\mathcal{S}_i = \mathcal{R}_i(\mathcal{S}_{i-1}; \pi, \tau), \qquad i = 1, \ldots, q, \tag{36}$$

where $\mathcal{R}_i$ is the elementary DRCC step at stage i, as defined in Section 6.2. The final reduced structure is

$$F_{\mathrm{Red}} = \mathrm{CNF}(\mathcal{S}_q). \tag{37}$$

At each stage i, the set of reachable states depends on the structure $\mathcal{S}_i$, the reconstruction rule τ, and the processing order π.

6.4.3 Separation of Interface Size and Reachable States

A key distinction is between the interface size $|B_i^{\pi,\tau}|$ and the number of reachable states N_i^{reach}:

- $|B_i^{\pi,\tau}|$ measures the *structural capacity* of the active interface at stage i.
- N_i^{reach} measures the *actually reachable reconstruction states* compatible with $\mathcal{S}_i$ and (π, τ).

In many cases, the reconstruction rule τ restricts the set of valid partial reconstructions so that

$$N_i^{\mathrm{reach}} \leq 2^{|B_i^{\pi,\tau}|}, \tag{38}$$

when the active-interface state is represented by $|B_i^{\pi,\tau}|$ binary components. This is a representational upper bound, not a universal DRCC-specific complexity claim. A substantially smaller value may occur for restrictive reconstruction rules, but that stronger separation is problem-dependent and must be established for the concrete construction.

6.4.4 Total DRCC Work

The *total DRCC work* is defined as the sum of reachable states over all reconstruction stages:

$$W_{\mathrm{DRCC}} := \sum_{i=0}^{q} N_i^{\mathrm{reach}}. \tag{39}$$

This quantity measures the total number of distinct partial reconstructions that are reachable across all stages of the DRCC reduction. It serves as a structural-combinatorial work measure, not a machine-level operation count or a wall-clock runtime.

Together with the DRCC width from Section 6.3, we now have two central quantities:

- $\omega_{\mathrm{DRCC}} = \max_{1 \le i \le q} |B_i^{\pi,\tau}|$ (peak structural capacity),
- $W_{\mathrm{DRCC}} = \sum_i N_i^{\mathrm{reach}}$ (total reconstruction work).

6.4.5 Example: Pigeonhole Principle

For the Pigeonhole Principle family PHP_n^{n-1} under a suitable DRCC reconstruction rule τ_{PHP}, one obtains

$$\omega_{\mathrm{DRCC}} = \Theta(n^2), \qquad W_{\mathrm{DRCC}} = \Theta(n). \tag{40}$$

This illustrates that a large structural interface ($\Theta(n^2)$) does not necessarily imply large reconstruction work ($\Theta(n)$) when the reconstruction rule τ is sufficiently restrictive. A detailed construction of the PHP reduction will be given in Section 6.7.

6.4.6 Relation to Solver Search Effort

It is important to distinguish DRCC work from the solver's search effort:

- W_{DRCC} is a *combinatorial work measure* that counts reachable partial reconstructions during the DRCC reduction.
- The solver's search effort (e.g., number of branching decisions, learned clauses, or runtime) depends on the structure of F_{Red} and the solver's internal heuristics.

The two stages have different units and therefore are not added directly as an exact numerical equation. A unit-consistent conceptual decomposition is obtained by introducing cost maps C_{DRCC} and C_{solver}:

$$\mathcal{E}_{\mathrm{total}} \;:=\; C_{\mathrm{DRCC}}(W_{\mathrm{DRCC}}) \;+\; C_{\mathrm{solver}}(S_{\mathrm{solver}})\,, \tag{41}$$

where S_{solver} denotes a separately defined solver-effort measure. No universal form for either cost map is assumed. In particular, W_{DRCC} is a structural-combinatorial quantity and does not directly translate into seconds or CPU time. The equation is therefore a conceptual decomposition of cost, not a claimed runtime law.

6.5 Algorithmic Steps of DRCC

This section details the concrete algorithmic steps performed by DRCC and shows how they lead to the finite-mathematical core quantities ω_{DRCC} and W_{DRCC} defined in Sections 6.3 and 6.4.

6.5.1 Input and Initialization

DRCC starts from a finite CNF structure

$$F = (V, C), \qquad V = \{x_1, \ldots, x_N\}, \quad C = \{C_1, \ldots, C_m\}, \tag{42}$$

together with:

- a processing order π (e.g., a permutation of $\{1, \ldots, n\}$ or a schedule over structural units),
- a reconstruction rule τ that specifies which structural constraints are enforced during reduction.

The initial state is set to

$$\mathcal{S}_0 := F. \tag{43}$$

6.5.2 Iterative Reduction Loop

Let $q := |\pi|$ denote the number of processing units specified by the chosen processing schedule. DRCC proceeds in q steps $i = 1, \ldots, q$; for the common variable-by-variable case with n variables, $q = n$. In each step i, the following operations are performed:

1. **Select next structural unit.** According to π, choose the next variable, clause group, or structural unit to be processed. Denote this unit by u_i. The sequence $(u_1, \ldots, u_q)$ is determined by the processing order π applied to F.
2. **Apply reconstruction rule.** Using τ, determine which constraints are imposed on u_i given the already processed prefix $\mathcal{S}_{i-1}$. This may include:
   - fixing certain literals or variable relations,
   - eliminating symmetric or equivalent configurations,
   - adding structural constraints that encode the task semantics.

   The role of π is to specify the processing order, while τ specifies the reconstruction rule (e.g., symmetry-breaking for PHP, monotonicity constraints, or interface conditions).
3. **Update structure.** Construct the reduced structure

   $$\mathcal{S}_i = \mathcal{R}_i(\mathcal{S}_{i-1}; \pi, \tau), \tag{44}$$

   by:
   - removing clauses that are already satisfied by the imposed constraints,
   - simplifying clauses that contain fixed literals,
   - possibly adding new clauses that encode the DRCC constraints derived from τ.

The result is a new complete DRCC state $\mathcal{S}_i$ representing the reduced problem state after applying the constraints determined by τ. Correctness requires that the reduction preserve the intended solution semantics, as specified in Section 6.2.

4. **Determine active interface.** Compute the active interface

$$B_i^{\pi,\tau}, \tag{45}$$

which, for SAT/CNF instances in this manuscript, is the set of processed variables that still share a clause (constraint scope) with at least one unprocessed variable. This declared object type makes $|B_i^{\pi,\tau}|$ a well-defined structural capacity measure. Other DRCC applications may use a different declared object type, but variables, constraints, and reconstruction states are not mixed in one cardinality without an explicit encoding.

5. **Count reachable states.** Determine the number of reachable reconstruction states at stage i:

$$N_i^{\text{reach}} := \left|\{\text{valid partial reconstructions compatible with } \mathcal{S}_i \text{ and } (\pi,\tau)\}\right|. \tag{46}$$

This counts the distinct partial reconstructions that are compatible with the reduced structure $\mathcal{S}_i$ and the constraints imposed by DRCC up to step i. For specific problem families (e.g., PHP under a suitable τ_{PHP}), it may hold that $N_i^{\text{reach}} = 1$ for the relevant reconstruction stages; this is a property of the specific construction, not a universal feature of DRCC.

After q processing steps, the final reduced structure is

$$F_{\text{Red}} := \text{CNF}(\mathcal{S}_q). \tag{47}$$

6.5.3 Derivation of DRCC Width

From the sequence of active interfaces $\{B_i^{\pi,\tau}\}_{i=1}^{q}$, the *DRCC width* is defined as

$$\omega_{\text{DRCC}} := \max_{1\le i\le q} \left|B_i^{\pi,\tau}\right|. \tag{48}$$

This quantity captures the maximal structural capacity that must be maintained during the reduction. In graph-theoretic terms, ω_{DRCC} is analogous to a width parameter (such as treewidth or pathwidth) adapted to the DRCC reconstruction process.

6.5.4 Derivation of DRCC Work

From the sequence of reachable-state counts $\{N_i^{\text{reach}}\}_{i=0}^{q}$, the *total DRCC work* is defined as

$$W_{\text{DRCC}} := \sum_{i=0}^{q} N_i^{\text{reach}}. \tag{49}$$

This sum quantifies the total number of distinct partial reconstructions that are reachable across all stages of the DRCC reduction. It is a structural-combinatorial work measure, not a machine-level operation count or a wall-clock runtime.

6.5.5 Summary of the Step Chain

The algorithmic steps of DRCC can be summarized as follows:

- **Select:** Choose u_i according to π.
- **Apply:** Impose constraints on u_i via τ.
- **Update:** Construct $\mathcal{S}_i = \mathcal{R}_i(\mathcal{S}_{i-1}; \pi, \tau)$.
- **Interface:** Compute $B_i^{\pi,\tau}$.
- **Count:** Compute N_i^{reach}.

After q processing steps, the quantities

$$\omega_{\text{DRCC}} = \max_{1 \le i \le q} |B_i^{\pi,\tau}| \quad \text{and} \quad W_{\text{DRCC}} = \sum_{i=0}^{q} N_i^{\text{reach}} \tag{50}$$

are fully determined. These form the finite-mathematical core on which all subsequent DRCC reduction rules, interface analyses, and runtime predictions are built.

6.6 Combined DRCC–Solver Pipeline

This section describes how the DRCC reduction integrates with a downstream SAT solver to form a complete satisfiability-solving pipeline. The presentation emphasizes the separation of roles between DRCC (structural reduction) and the solver (branching search), and clarifies the conditions under which the combined pipeline preserves satisfiability.

6.6.1 Two-Stage Pipeline

Only after the DRCC reduction is complete is the remaining search space handed to a downstream SAT solver:

$$F \xrightarrow{\text{DRCC}_{\text{Red}}(\pi,\tau)} F_{\text{Red}} \xrightarrow{\text{Solver}} \{\text{SAT}, \text{UNSAT}\}. \tag{51}$$

Here, $\text{DRCC}_{\text{Red}}(\pi, \tau)$ denotes the overall DRCC reduction map defined in Section 6.2:

$$\text{DRCC}_{\text{Red}}(\cdot; \pi, \tau) = \mathcal{R}_q \circ \mathcal{R}_{q-1} \circ \cdots \circ \mathcal{R}_1, \tag{52}$$

where each $\mathcal{R}_i$ is an elementary DRCC step as specified in Section 6.5.

Equivalently, the combined algorithm can be written as

$$\text{DRCC-Solver}(F; \pi, \tau) \;=\; \text{Solver}\Big(\text{DRCC}_{\text{Red}}(F; \pi, \tau)\Big). \tag{53}$$

The role separation is thus mathematically explicit:

- $\underbrace{F \to F_{\text{Red}}}_{\text{DRCC: structural reduction}}$,
- $\underbrace{F_{\text{Red}} \to \text{search tree}}_{\text{Solver: remaining branching search}}$.

DRCC does not attempt to emulate or replace the solver; it changes the effective structural representation of the problem before the solver begins its branching search.

6.6.2 Satisfiability Preservation

For the combined pipeline to be correct, the DRCC reduction must satisfy a separately established correctness condition. This requirement is named the *Satisfiability-Preservation Condition*. When the concrete construction satisfies the Satisfiability-Preservation Condition, one has

$$\boxed{\mathrm{SAT}(F) \quad \Longleftrightarrow \quad \mathrm{SAT}(F_{\mathrm{Red}})}. \tag{54}$$

This condition must be established separately for each construction. In particular, if $F_{\mathrm{Red}} = F \wedge \zeta$ is formed by adding symmetry- breaking constraints, the nontrivial direction for a satisfiable input is that every satisfiable orbit contains at least one assignment satisfying ζ. That canonical-representative property is not established here in general.

This condition ensures that the solver's output on F_{Red} is valid for the original formula F:

- If F_{Red} is satisfiable, then F is satisfiable.
- If F_{Red} is unsatisfiable, then F is unsatisfiable.

Thus, the combined pipeline is sound and complete for the original satisfiability problem, provided that the DRCC reduction satisfies the satisfiability-preservation condition stated above.

The concrete mechanism by which satisfiability is preserved depends on the specific reconstruction rule τ and is discussed in Section 6.2.

6.6.3 Solver Independence

The DRCC framework can be combined with different SAT solvers, provided that the reduced representation F_{Red} is accepted by the solver and the DRCC reduction preserves the intended solution semantics.

For example, the following instantiations are possible:

$$\mathrm{DRCC}-\mathrm{DPLL}, \qquad \mathrm{DRCC}-\mathrm{MiniSat}, \qquad \mathrm{DRCC}-\mathrm{Glucose}. \tag{55}$$

The framework does not prescribe a specific solver; instead, it provides a structural preprocessing phase that can be paired with various backtracking- or CDCL-based solvers.

6.6.4 Example: Pigeonhole Principle

For the Pigeonhole Principle family PHP_n^{n-1} under a suitable DRCC reconstruction rule τ_{PHP}, the pipeline operates as follows:

- DRCC reduces PHP_n^{n-1} to a structurally simplified formula F_{Red} with $\omega_{\mathrm{DRCC}} = \Theta(n^2)$ and $W_{\mathrm{DRCC}} = \Theta(n)$.
- The downstream solver then processes F_{Red}, potentially benefiting from the reduced structural complexity.

Experimental measurements (see Section 3) indicate that DRCC-based preprocessing can substantially reduce solver runtime on PHP instances compared to solving the original formula directly. The exact speedup factors depend on the solver, the implementation, and the specific instance parameters.

6.6.5 Summary

The combined DRCC–solver pipeline can be summarized as:

$$\boxed{F \xrightarrow{\mathcal{R}_1,\ldots,\mathcal{R}_q} F_{\mathrm{Red}} \xrightarrow{\text{Solver}} \{\mathrm{SAT}, \mathrm{UNSAT}\},} \tag{56}$$

with the two central structural measures:

$$\boxed{\omega_{\mathrm{DRCC}} = \max_{1 \le i \le q} |B_i^{\pi,\tau}| \quad \text{and} \quad W_{\mathrm{DRCC}} = \sum_{i=0}^{q} N_i^{\mathrm{reach}}.} \tag{57}$$

DRCC changes the effective structural representation of the problem by:

- eliminating symmetric or equivalent configurations via τ,
- restricting the search space to a canonical subset of reconstructions,
- reducing the relevant structural degrees of freedom before the solver begins its branching search.

The solver then operates on the reduced structure F_{Red}, which may admit a more efficient satisfiability decision than the original formula F.

6.7 Example: Pigeonhole Principle

This section demonstrates the DRCC quantities ω_{DRCC} and W_{DRCC} on the Pigeonhole Principle family PHP_n^{n-1}. We first define the CNF encoding, then specify the DRCC reconstruction rule τ_{PHP}, and finally derive the active interface size and reachable-state counts.

6.7.1 CNF Encoding of PHP_n^{n-1}

The Pigeonhole Principle formula PHP_n^{n-1} encodes the statement that n pigeons cannot be placed into $n-1$ holes without collision. It uses Boolean variables

$$x_{i,j}, \qquad 1 \le i \le n, \quad 1 \le j \le n-1, \tag{58}$$

where $x_{i,j}$ is true iff pigeon i is placed in hole j.

The clause set consists of:

- **At-least-one constraints:** For each pigeon i,

$$A_i \;=\; x_{i,1} \vee x_{i,2} \vee \cdots \vee x_{i,n-1}. \tag{59}$$

  Each pigeon must be placed in at least one hole.

- **At-most-one constraints:** For each pair of distinct pigeons $i \neq k$ and each hole j,

$$M_{i,k,j} \;=\; \neg x_{i,j} \vee \neg x_{k,j}. \tag{60}$$

  No two pigeons may occupy the same hole.

The total number of variables is $|V| = n(n-1)$. For $n \ge 2$, the formula PHP_n^{n-1} is unsatisfiable.

6.7.2 DRCC Reconstruction Rule τ_{PHP}

We consider a DRCC reconstruction rule τ_{PHP} that processes pigeons in order $i = 1, \ldots, n$ and restricts each pigeon i to holes

$$j \leq \min(i, n-1). \tag{61}$$

For the concrete deterministic reconstruction rule, the selected assignment is

$$x_{k,k} = \top, \qquad x_{k,j} = \bot \quad (j \neq k,\ 1 \leq j \leq k). \tag{62}$$

That is:

- Pigeon 1 may only occupy hole 1.
- Pigeon 2 may occupy holes 1 or 2.
- ...
- Pigeon $n-1$ may occupy holes $1, \ldots, n-1$.
- Pigeon n is restricted to holes $1, \ldots, n-1$ (all holes).

This rule restricts the reconstruction to a canonical subset of configurations. For the unsatisfiable PHP family considered here, this restriction exposes a contradiction along the specified reconstruction path.

6.7.3 Active Interface and DRCC Width

For the PHP reduction, $B_i^{\pi,\tau}$ is the same active-interface set used above: it consists of the processed variables that still share a constraint scope with an unprocessed variable. Under pigeon-block order, the at-least-one factor of a processed pigeon is fully processed and therefore does not keep a variable active. The at-most-one hole factors connect each processed $x_{k,j}$ to later pigeons that can also use hole j. Because pigeon n can use every hole $1, \ldots, n-1$, every variable processed through stage $i < n$ remains active. Thus the interface is exactly

$$B_i^{\pi,\tau} = \{x_{k,j} : 1 \leq k \leq i,\ 1 \leq j \leq k\}, \qquad 1 \leq i \leq n-1. \tag{63}$$

Consequently,

$$|B_i^{\pi,\tau}| = \sum_{k=1}^{i} k = \frac{i(i+1)}{2}, \qquad 1 \leq i \leq n-1. \tag{64}$$

6.7.4 Reachable States and DRCC Work

We now analyze the number of reachable reconstruction states N_i^{reach} at each stage i.

For stages $i = 0, 1, \ldots, n-1$, the DRCC reconstruction rule τ_{PHP} admits exactly one valid partial reconstruction at each stage. The uniqueness of this reconstruction follows inductively. At stage k, the rule τ_{PHP} permits holes $1, \ldots, k$. The preceding pigeons occupy holes $1, \ldots, k-1$, and the at-most-one constraints therefore exclude these $k-1$ choices. Hence pigeon k must be

assigned to hole k. Consequently, exactly one valid partial reconstruction exists at every stage $i \leq n-1$:

$$N_i^{\text{reach}} = 1, \qquad i = 0, 1, \ldots, n-1. \tag{65}$$

This corresponds to the canonical assignment in which pigeon k is placed in hole k for $k = 1, \ldots, i$.

At stage $i = n$, however, all $n-1$ holes are already occupied by pigeons $1, \ldots, n-1$. Pigeon n cannot be placed in any hole without violating the at-most-one constraints. Therefore, there is no valid partial reconstruction at the final stage:

$$N_n^{\text{reach}} = 0. \tag{66}$$

For this specific PHP construction and the reconstruction rule τ_{PHP}, the total DRCC work is thus

$$W_{\text{DRCC}} \;=\; \sum_{i=0}^{n} N_i^{\text{reach}} \;=\; \sum_{i=0}^{n-1} 1 \;+\; 0 \;=\; n \;=\; \Theta(n). \tag{67}$$

6.7.5 Final Contradiction

The DRCC reduction exposes the unsatisfiability of PHP_n^{n-1} as follows. After processing pigeons $1, \ldots, n-1$ under τ_{PHP}, the canonical reconstruction assigns:

$$x_{1,1} = \top, \quad x_{2,2} = \top, \quad \ldots, \quad x_{n-1,n-1} = \top. \tag{68}$$

All holes $1, \ldots, n-1$ are now occupied. For pigeon n, the at-least-one constraint requires

$$A_n \;=\; x_{n,1} \vee x_{n,2} \vee \cdots \vee x_{n,n-1}. \tag{69}$$

However, the at-most-one constraints imply

$$\neg x_{n,j}, \qquad j = 1, \ldots, n-1, \tag{70}$$

since hole j is already occupied by pigeon j. Therefore, A_n reduces to the empty clause:

$$A_n \;\to\; \square. \tag{71}$$

The reconstruction assignments are incorporated into the reduced representation, so that the resulting unit consequences expose the empty clause. The reduced formula F_{Red} thus contains an explicit contradiction. A downstream SAT solver can therefore determine unsatisfiability by unit propagation, without requiring branching on the reduced instance.

6.7.6 Summary

For the Pigeonhole Principle family PHP_n^{n-1} under the DRCC reconstruction rule τ_{PHP}, we obtain:

- DRCC width: $\omega_{\text{DRCC}} = \Theta(n^2)$.
- DRCC work: $W_{\text{DRCC}} = \Theta(n)$.

This illustrates the key DRCC effect: a large structural interface ($\Theta(n^2)$) does not imply large reconstruction work ($\Theta(n)$) when the reconstruction rule τ is sufficiently restrictive. The DRCC reduction exposes the contradiction along a canonical path, allowing the downstream solver to detect unsatisfiability without extensive branching.

6.8 Resolution vs. DPLL

This section relates the DRCC framework to classical results on resolution proof complexity and DPLL-style backtracking search. The goal is to clarify how DRCC interacts with known lower bounds for the Pigeonhole Principle and why the DRCC reduction does not contradict these classical results.

6.8.1 Resolution Lower Bounds for PHP

For standard CNF encodings of the Pigeonhole Principle, exponential resolution lower bounds are known; the precise statement depends on the encoding, the proof system, and the proof-size measure. Haken's result provides a classical source for this line of lower bounds [2].

For standard formulations of the PHP family, stronger tree-like resolution lower bounds are also known. Their precise form depends on the encoding and on whether proof size, search-tree size, or another measure is used; see Beyersdorff et al. [3].

These lower bounds concern resolution refutations of the specified original encoding. They do not automatically apply to every algorithm that changes the representation before solving, adds structural constraints, or uses a different proof system.

6.8.2 DRCC Constraints and Equisatisfiability

The DRCC framework introduces additional structural constraints S_{DRCC} derived from the reconstruction rule τ. The reduced formula can be written as

$$F_{\mathrm{Red}} \;=\; \mathcal{R}_{\mathrm{DRCC}}(F;\pi,\tau), \tag{72}$$

where $\mathcal{R}_{\mathrm{DRCC}}$ applies the constraints imposed by τ and simplifies the structure accordingly.

For the PHP family under the rule τ_{PHP} defined in Section 6.7, the reduced formula F_{Red} contains the original PHP constraints together with the structural restrictions induced by τ_{PHP}.

For a general DRCC reduction, satisfiability preservation is a required correctness condition rather than a theorem that follows merely from adding structural constraints. The condition is

$$\mathrm{SAT}(F) \quad\Longleftrightarrow\quad \mathrm{SAT}(F_{\mathrm{Red}}). \tag{73}$$

For the present PHP benchmark, both $F = \mathrm{PHP}_n^{n-1}$ and $F_{\mathrm{Red}} = F \wedge \zeta_n$ are unsatisfiable: F is the classical Pigeonhole Principle formula, and adding constraints cannot turn an unsatisfiable formula into a satisfiable one. Hence the equivalence holds for this benchmark family at the level of the SAT/UNSAT answer. This does not prove satisfiability preservation for arbitrary satisfiable inputs or for arbitrary reconstruction rules τ. For such a general claim, an additional canonical-representative/orbit-completeness proof is required.

6.8.3 Relation to Tree-like Resolution and DPLL

These results establish lower bounds for the relevant standard resolution and tree-like-resolution proof systems, and therefore for the corresponding DPLL-style procedures operating directly on the original PHP representation. They

do not establish an absolute lower bound for every possible resolution variant, preprocessing method, or transformed representation. The DRCC procedure is not claimed to violate these bounds, because it changes the representation before the downstream solver is invoked.

Tree-like resolution proofs correspond to the execution of a DPLL-style backtracking solver without clause learning on the original formula. The exponential lower bounds for tree-like resolution on PHP therefore concern such procedures operating directly on the original PHP representation.

The DRCC framework does not refute these classical lower bounds. Instead, DRCC changes the representation on which the subsequent solver operates:

$$F_{\mathrm{PHP}} \xrightarrow{\mathrm{DRCC}_{\mathrm{Red}}(\pi,\tau)} F_{\mathrm{Red}} \xrightarrow{\mathrm{Solver}} \{\mathrm{SAT}, \mathrm{UNSAT}\}. \tag{74}$$

The solver then operates on the reduced formula F_{Red}, which may admit a more efficient satisfiability decision than the original formula F_{PHP}. The classical lower bounds apply to F_{PHP}, not to F_{Red}.

6.8.4 Empirical Observations on PHP Instances

Experimental measurements on PHP instances indicate that DRCC-based preprocessing can substantially reduce solver runtime compared to solving the original formula directly. The exact speedup factors depend on the solver, the implementation, and the specific instance parameters.

For the DRCC reduction defined in Section 6.7, the reduced formula F_{Red} contains an explicit contradiction that can be detected by unit propagation. A downstream SAT solver can therefore determine unsatisfiability without requiring branching decisions on the reduced instance.

These observations are consistent with the theoretical analysis: the DRCC reduction exposes the contradiction along a canonical path, allowing the solver to detect unsatisfiability through unit propagation rather than extensive branching search.

6.8.5 Summary

The relationship between DRCC and classical resolution/DPLL lower bounds can be summarized as follows:

- Classical resolution lower bounds (e.g., $2^{\Omega(n)}$ for PHP_n^{n-1}) apply to the original formula F_{PHP} and direct procedures whose proof system falls within the scope of the corresponding theorem.
- DRCC does not refute these lower bounds; it changes the representation on which the subsequent solver operates, subject to the stated satisfiability-preservation condition.
- The reduced formula F_{Red} may admit a more efficient satisfiability decision than F_{PHP}, as illustrated by the PHP example in Section 6.7.
- The DRCC structural construction is solver-agnostic at the interface level, while runtime and search behavior remain solver- and configuration-dependent. It can be combined with different SAT solvers (DPLL, MiniSat, Glucose), provided that the reduced representation is accepted and the satisfiability-preservation condition holds.

Thus, DRCC should be understood as a structural preprocessing framework that transforms the input formula before the solver begins its branching search, not as a refutation of classical complexity lower bounds for the original problem representation.

6.9 Core Finite-Mathematical Summary

This section consolidates the finite-mathematical foundation of DRCC developed in Sections 6.1–6.8. The core definitions, algorithmic steps, and structural measures are summarized in a unified framework.

6.9.1 Basic Objects and Parameters

DRCC operates on a finite CNF structure

$$F = (V, C), \qquad V = \{x_1, \ldots, x_N\}, \quad C = \{C_1, \ldots, C_m\}, \tag{75}$$

together with two fixed parameters:

- π: a processing order of variables or structural units (e.g., a permutation of $\{1, \ldots, n\}$ or a more general schedule),
- τ: a reconstruction rule that specifies which structural constraints are enforced during reduction.

Both π and τ are fixed before the reduction starts and remain constant throughout the DRCC phase.

6.9.2 DRCC Reduction Map

The DRCC reduction is defined as the composition of elementary reduction steps. With $q := |\pi|$ denoting the number of processing units in the chosen schedule,

$$\mathcal{R}_{\mathrm{DRCC}}(\cdot; \pi, \tau) \;=\; \mathcal{R}_q \circ \mathcal{R}_{q-1} \circ \cdots \circ \mathcal{R}_1, \tag{76}$$

where each $\mathcal{R}_i$ is an elementary DRCC step as specified in Section 6.5.

Applied to the input formula F, the reduction generates the complete DRCC state

$$\mathcal{S}_0 = F, \qquad \mathcal{S}_i = \mathcal{R}_i(\mathcal{S}_{i-1}; \pi, \tau), \qquad i = 1, \ldots, q, \tag{77}$$

and the solver-facing CNF component is extracted as

$$F_{\mathrm{Red}} = \mathrm{CNF}(\mathcal{S}_q). \tag{78}$$

The reduction is required to preserve satisfiability under the condition stated in Section 6.2; this is a correctness condition for the concrete reduction, not an unconditional theorem for arbitrary τ:

$$F \text{ is satisfiable} \quad \Longleftrightarrow \quad F_{\mathrm{Red}} \text{ is satisfiable.} \tag{79}$$

6.9.3 Algorithmic Step Chain

The generic DRCC algorithm proceeds in q steps $i = 1, \ldots, q$, where $q = |\pi|$. In the variable-by-variable case, $q = n$. At each step i, the following operations are performed:

1. **Select:** Choose the next structural unit u_i according to π.
2. **Apply:** Impose constraints on u_i via τ.
3. **Update:** Construct
$$\mathcal{S}_i = \mathcal{R}_i(\mathcal{S}_{i-1}; \pi, \tau). \tag{80}$$
4. **Interface:** Compute the active interface $B_i^{\pi,\tau}$.
5. **Count:** Compute the number of reachable states N_i^{reach}.

The step chain starts from $\mathcal{S}_0 = F$ and applies $\mathcal{S}_i = \mathcal{R}_i(\mathcal{S}_{i-1}; \pi, \tau)$ for $i = 1, \ldots, q$, yielding the solver-facing CNF component $F_{\text{Red}} = \text{CNF}(\mathcal{S}_q)$.

6.9.4 Structural Measures

Two central quantities characterize the DRCC reduction:

- **DRCC width:**
$$\omega_{\text{DRCC}} := \max_{1 \le i \le q} \left| B_i^{\pi,\tau} \right|. \tag{81}$$
This measures the peak structural capacity required during the reduction.
- **DRCC work:**
$$W_{\text{DRCC}} := \sum_{i=0}^{q} N_i^{\text{reach}}. \tag{82}$$
This measures the total number of distinct partial reconstructions that are reachable across all stages.

These quantities can behave very differently. For the Pigeonhole Principle family PHP_n^{n-1} under the rule τ_{PHP}, we obtained (Section 6.7):

$$\omega_{\text{DRCC}} = \Theta(n^2), \qquad W_{\text{DRCC}} = \Theta(n). \tag{83}$$

This illustrates that a large structural interface does not necessarily imply large reconstruction work when the reconstruction rule τ is sufficiently restrictive.

6.9.5 Combined DRCC–Solver Pipeline

In the strict two-stage formulation, the pipeline consists of two stages. Implementations may alternatively integrate DRCC reconstruction with propagation or search.

$$F \xrightarrow{\text{DRCC}_{\text{Red}}(\pi,\tau)} F_{\text{Red}} \xrightarrow{\text{Solver}} \{\text{SAT}, \text{UNSAT}\}. \tag{84}$$

Equivalently:

$$\text{DRCC-Solver}(F; \pi, \tau) = \text{Solver}\Big(\text{DRCC}_{\text{Red}}(F; \pi, \tau)\Big). \tag{85}$$

The role separation is explicit:

- $\underbrace{F \to F_{\mathrm{Red}}}_{\text{DRCC: structural reduction}}$,
- $\underbrace{F_{\mathrm{Red}} \to \text{search tree}}_{\text{Solver: remaining branching search}}$.

DRCC does not attempt to emulate or replace the solver; it changes the effective structural representation of the problem before the solver begins its branching search.

The DRCC structural construction is solver-agnostic at the interface level, while runtime and search behavior remain solver- and configuration-dependent. It can be combined with different SAT solvers (e.g., DPLL, MiniSat, Glucose), provided that the reduced representation is accepted by the solver and the satisfiability-preservation condition holds.

6.9.6 Relation to Classical Lower Bounds

Classical resolution and tree-like resolution lower bounds (e.g., $2^{\Omega(n)}$ for PHP_n^{n-1}) apply to the original formula F_{PHP} and direct procedures whose proof system falls within the scope of the corresponding theorem.

The shorter refutations that may be obtained after DRCC reduction do not contradict these classical exponential lower bounds. DRCC changes the representation on which the subsequent solver operates, subject to the stated satisfiability-preservation condition. The reduced formula F_{Red} may admit a more efficient satisfiability decision than F_{PHP}.

6.9.7 Outlook

The DRCC pre-processor can be combined with established CDCL solvers, potentially improving performance on structured instances such as PHP, while leaving the classical lower bounds for the original, unreduced formula unchanged. The present experiments evaluate only MiniSat 2.2 and Glucose 3; no claim of state-of-the-art solver performance is made. The exact benefits depend on the problem class, the reconstruction rule τ, and the solver implementation.

6.9.8 Unified Summary

The mathematical core of DRCC can be summarized as:

$$\boxed{\begin{aligned} \mathcal{S}_0 &= F, \\ \mathcal{S}_i &= \mathcal{R}_i(\mathcal{S}_{i-1}; \pi, \tau), \quad i = 1, \dots, q, \\ F_{\mathrm{Red}} &= \mathrm{CNF}(\mathcal{S}_q), \\ \omega_{\mathrm{DRCC}} &= \max_{1 \le i \le q} |B_i^{\pi,\tau}|, \\ W_{\mathrm{DRCC}} &= \sum_{i=0}^{q} N_i^{\mathrm{reach}}, \\ F_{\mathrm{Red}} &\xrightarrow{\text{Solver}} \{\mathrm{SAT}, \mathrm{UNSAT}\}. \end{aligned}} \tag{86}$$

All subsequent DRCC reduction rules, interface analyses, and runtime predictions are built on this finite-mathematical foundation.

Appendix A: Experimental Reproducibility and Execution Record

Script. All results in Section 3 were produced by a single script, `exp_cdcl_php.py`, which implements: the PHP_n^{n-1} CNF generator; the DRCC symmetry-breaking clause generator; a from-scratch reference DPLL solver (unit propagation and chronological backtracking only, no clause learning); and a subprocess-isolated wrapper around MiniSat 2.2 / Glucose 3 via `python-sat`, used specifically because a direct in-process call to these solvers cannot be externally interrupted on timeout.

Execution environment. Python 3.14.7 (Clang 21.0.0), macOS 26.5.2, arm64 (Apple Silicon), `python-sat` version 1.9.dev15. All n values reported in Table 2 were executed on this same machine and environment, in three separate script invocations ($n \in \{6, 9, 10, 11, 12\}$; $n \in \{2, 3, 4, 5, 7, 50\}$; $n = 8$), each writing its own JSON output file with per-run raw timing and, for MiniSat/Glucose, solver-reported statistics (decisions, conflicts, restarts, propagations).

Disclosed gap: seed usage. A fixed seed (20260819, matching the convention used elsewhere in the DRCC record) is logged in each run's output for consistency, but is not actually consumed by any of the six solver configurations reported here: the reference DPLL solver is fully deterministic given its fixed variable ordering, and MiniSat/Glucose are invoked through `python-sat` without an explicit seed parameter, so their internal (deterministic, in the versions used) default behavior governs run-to-run reproducibility. This is disclosed rather than implied, since the presence of a logged seed value could otherwise suggest randomized behavior that is not actually present in this experiment.

Disclosed gap: output file management. The three script invocations described above wrote to output filenames that were not consistently distinct across runs, and one intermediate result file was inadvertently overwritten during the course of running this experiment before all values were transcribed into this manuscript. No data referenced in Table 2 or Table 3 was affected – all values were recorded from their respective run's terminal output and independently cross-checked (e.g. the $n = 8$ reference-DPLL decision count, 32780, matches a value independently computed earlier against the same solver implementation) – but the raw JSON files for $n \in \{2, 3, 4, 5, 7, 50\}$ specifically no longer exist as a standalone artifact and cannot be re-deposited verbatim in an accompanying data repository. This is disclosed as a reproducibility limitation rather than silently omitted.

Timeout convention. A value of 120.00 s in Table 2, marked with *, always denotes that all three repetitions of that configuration were terminated by the 120-second timeout without returning a result. It is a ceiling, not a measurement; the true runtime for that configuration and n is unknown beyond "at least 120 seconds."

Data and code availability. The experimental record is contained in the manuscript together with the execution environment, configuration, numerical

tables, and the documented raw-data basis used for the reported analyses. The remaining terminal records and tabulated values constitute the available evidence basis for the reported results, but the missing standalone JSON files cannot be reproduced verbatim from the present artifact. No placeholder, invented, or reused repository identifier is used.

Appendix B: Symbol and Equation Register

This appendix collects the principal notation and equations used throughout the main sections of this manuscript, in the same spirit as the symbol register of DRCC-V2-REW2, Chapter 10.6 [1], to keep this document's quantities traceable against that manuscript's general framework.

Table 6: Register of nine selected principal equations used for the structural, experimental, and predictive analyses in this manuscript.

Register No.	Main equation	Section	Use in
(1)	$T_{\mathrm{DRCC,total}}(N) = T_{\mathrm{certificate}}(N) + T_{\text{structural check}}(N)$	§2.1	Q1: total verification cost
(2)	$G_{\mathrm{time}}(n) = T_{\mathrm{Classic}}(P_n)/T_{\mathrm{DRCC}}(P_n, \zeta_n)$	§3.3	Runtime comparison
(3)	$W = (n_w, R_w[\mathrm{s}]), \quad G_{\mathrm{time}}(n_w) = 1$	§3.3	Formal runtime transition
(4)	$W_{\mathrm{int}} = [n_-, n_+]$	§3.3	Transition interval
(5)	$\lvert B_i^{\pi,\tau}\rvert = i(i+1)/2, \quad 1 \le i \le n-1$	Section 6.7.3	Active-interface cardinality
(6)	$\omega_{\mathrm{DRCC}} = (n-1)n/2 = \Theta(n^2)$	Section 6.7.3	Maximum reconstruction width
(7)	$N_i^{\mathrm{reach}} = 1$ for $i = 0, \ldots, n-1$; $N_n^{\mathrm{reach}} = 0$	Section 6.7.3	Reachable-state count
(8)	$W_{\mathrm{DRCC}} = n = \Theta(n)$	Section 6.7.3	Structural reconstruction work
(9)	$T(n) = a\, n^p$	Section 5.3	Empirical runtime prediction

Table 7: Symbol table for Q2 and the prospective reconstruction-width analysis. The complete symbol register is kept in one continuous table.

Symbol	Meaning
N	General number of Boolean variables in a CNF instance.
F	General CNF instance, $F = (V, C)$.
V	Variable set, $V = \{x_1, \ldots, x_N\}$.
C	Clause set, $C = \{C_1, \ldots, C_m\}$.
m	Number of clauses in the general CNF instance.
$\mathcal{S}_i$	Full DRCC state after processing stage i, with $\mathcal{S}_0 = F$.
$\mathcal{R}_i$	DRCC reduction/transformation operator at stage i.
F_i	CNF component extracted from the full DRCC state $\mathcal{S}_i$.
F_{Red}	Solver-facing reduced CNF component, $F_{\mathrm{Red}} = \mathrm{CNF}(\mathcal{S}_{\mathrm{Red}})$.
σ	Solver configuration, including the concrete solver/version and relevant settings.
n	Problem-size parameter: number of pigeons in the PHP instance; there are $n-1$ holes.
PHP_n^{n-1}	Pigeonhole Principle instance with n pigeons and $n-1$ holes.
P_n	Shorthand for PHP_n^{n-1}.
i	Pigeon index, $1 \le i \le n$.
j	Hole index, $1 \le j \le n-1$.
$x_{i,j}$	Boolean variable indicating whether pigeon i is assigned to hole j.
ζ_n	Instance-specific DRCC symmetry-breaking clause set added to P_n (Section 3.1).
$T_{\mathrm{Classic}}(P_n)$	Measured wall-clock time of the corresponding non-DRCC solver configuration.
$T_{\mathrm{DRCC}}(P_n, \zeta_n)$	Measured wall-clock time of the corresponding solver configuration with DRCC.
$G_{\mathrm{time}}(n)$	Runtime ratio $T_{\mathrm{Classic}}(P_n)/T_{\mathrm{DRCC}}(P_n, \zeta_n)$.
W	Formal runtime-transition point; distinct from W_{DRCC}, the structural DRCC work measure.
n_w	Instance size at the formal runtime transition.
R_w	Common wall-clock runtime at the transition, measured in seconds; the [s] in W denotes this unit, not a separate variable.
$W_{\mathrm{int}} = [n_-, n_+]$	Transition interval used when no exact tested n_w is observed (Eq. (4)).
n_-	Lower tested instance size satisfying $G_{\mathrm{time}}(n_-) < 1$.
n_+	Upper tested instance size satisfying $G_{\mathrm{time}}(n_+) > 1$.
*	Timeout marker in Table 2; 120.00^* denotes that all three repetitions reached the 120-second timeout without returning a result, so the true runtime is unknown and at least 120 s.
π	Processing order of variables or pigeon blocks.
τ	Declared task / assignment-reconstruction rule used in the DRCC analysis.
q	Number of processing stages or structural units in the chosen schedule π.
$\mathcal{E}_{\mathrm{total}}$	Conceptual total cost in the unit-consistent cost decomposition.
C_{DRCC}	Cost map from the structural DRCC work measure to conceptual DRCC cost.
C_{solver}	Cost map from solver effort to conceptual solver cost.
S_{solver}	Separately defined solver-effort measure.
a	Fitted scale parameter in the empirical power-law model $T(n) = an^p$.
p	Fitted exponent in the empirical power-law model $T(n) = an^p$.
$B_i^{\pi,\tau}$	Active interface after stage i, under processing order π and task τ.
ω_{DRCC}	Maximum active-interface size over all stages, $\max_i \lvert B_i^{\pi,\tau}\rvert$.
$\Omega_i(F, \pi, \tau)$	Set of admissible partial reconstructions after applying τ through stage i.
N_i^{reach}	Number of reachable states at stage i, $\lvert\Omega_i(F, \pi, \tau)\rvert$.
W_{DRCC}	Total DRCC reconstruction work, $\sum_i N_i^{\mathrm{reach}}$.

The governing equations, restated from DRCC-V2-REW2, Chapter 10.6 [1] and instantiated here with the corresponding non-DRCC solver configurations as classical baselines, are given in Equations (2)–(4) in Section 3.

The formal runtime transition is defined as $W = (n_w, R_w[\mathrm{s}])$, $G_{\mathrm{time}}(n_w) = 1$, whereas a visible change in the slope or growth regime of a runtime curve is treated as a separate *curve-transition feature* and is not identified with W unless

$G_{\text{time}} = 1$ is actually satisfied. Table 8 applies this distinction across all four runtime figures in this document.

Table 8: Distinction between visible curve transitions and the formal runtime transition W.

Figure	**Visible curve transition**	**Formal runtime transition**
Figure 1 (Adder)	possibly a growth-regime change	no W, since $G_{\text{time}} > 1$ from $N = 4$
Figure 2 (DPLL)	yes – visible kink / growth-regime change	no W
Figure 3 (MiniSat)	yes	$W_{\text{int}} = [4, 5]$
Figure 4 (Glucose)	possibly a growth-regime change	no W_{int}, since $G_{\text{time}} > 1$

References

[1] Reza Hesamiy. *Dimensional Reduction via Controlled Combinatorics (DRCC-V2-REW2).* aiXiv, 2026. `https://aixiv.science/abs/aixiv.260711.000001`

[2] Armin Haken. The intractability of resolution. *Theoretical Computer Science*, 39:297–308, 1985.

[3] Olaf Beyersdorff, Nicola Galesi, and Massimo Lauria. A lower bound for the pigeonhole principle in tree-like resolution by asymmetric Prover–Delayer games. *Information Processing Letters*, 110(23):1074–1077, 2010.

[4] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *Theory and Applications of Satisfiability Testing (SAT 2003)*, pages 502–518. Springer, 2004.

[5] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in established SAT solvers. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI 2009)*, pages 399–404, 2009.

[6] Alexey Ignatiev, Antonio Morgado, and Joao Marques-Silva. PySAT: A Python toolkit for prototyping with SAT oracles. In *Theory and Applications of Satisfiability Testing (SAT 2018)*, pages 428–437. Springer, 2018.

Appendix C: Complete Source Code of Experiment 1

Purpose. This appendix provides the complete source of the corrected DRCC-V3.1 implementation used for Experiment 1. The listing contains the exhaustive classical verification, the corrected constant-size DRCC certificate, the explicit N-cell structural representation check used for Q1, runtime measurement, the modeled redundant-wiring count, result export, and the nominal-$N = 32$ certificate timing. The historical timing tables were produced by an earlier source version; exact reproduction of those tables with the corrected source requires a fresh execution of Experiment 1. The 95% confidence intervals reported in the prospective-test section are not computed by this source listing.

Listing 1: Complete Python source code used for DRCC-V3 Experiment 1.

```
#!/usr/bin/env python3
"""
DRCC-V3 Experiment 1: Full-Adder Chain -- Redundant-Wiring Elimination
and Verification Runtime.

Task: verify that an N-bit ripple-carry adder built from single-bit full
adders correctly computes A + B + c_in for ALL possible inputs.

Two verification strategies are actually implemented and timed:

  T_Classic : brute-force enumeration. Enumerate every one of the
              2^(2N+1) raw input assignments (A in [0,2^N), B in
              [0,2^N), c_in in {0,1}), simulate the ripple-carry chain
              bit-by-bit, and check the result against integer
              addition. This is the textbook exhaustive verification
              approach.

  T_DRCC    : controlled-reduction certificate.
              A single full-adder cell's output (s, c_out) depends
              only on the Hamming weight w = a+b+c_in of its 3 raw
              input wires (w in {0,1,2,3}), not on which wires are
              set. This local equivalence is checked once, directly,
              on the 8 raw patterns (constant cost, independent of N),
              collapsing the 8 raw patterns into 4 weight classes Z(P).
              A second constant-size check enumerates the 8 local
              full-adder transitions (a,b,carry_in) in {0,1}^3.
              The total Hamming weight w=a+b+carry_in indexes the class
              table, and the returned sum/carry outputs are checked
              against the corresponding binary decomposition. This is
              a fixed-size local carry-state transition check.
              The code verifies the local certificate used by the
              mathematical induction; the induction itself is supplied
              analytically. Thus this code does not, by itself, verify
              an N-cell carry chain for arbitrary N.
              Because both checks are of fixed size, this certificate
              is O(1) in N: it does NOT re-run for every N, and its
              real measured wall-clock time is therefore expected to
              stay flat while T_Classic grows exponentially in N.

Redundant-wiring count (modeled, structural/count-level evidence, not timed):
under the stated independent decode-path model, the raw representation
has 8 distinguishable paths per cell and the class representation has
4 paths per cell, yielding a modeled ratio of 2^N. RAW_LUT is a Python
table with eight entries; it is a representation model, not a physical
hardware network or a measurement of physical wiring, memory, or
interconnect.

All reported execution times in the measured cases are obtained from
actual timed executions. Count-level quantities and explicitly identified
projections are reported separately and are not presented as measurements.

The source below contains the corrected structural and carry-state checks
implemented for this manuscript revision. Because the correction changes the
```

```
executed code path, previously reported timing tables are historical results
from the earlier experimental run and are not silently re-labelled as
measurements of the corrected source. Reproducibility of the numerical tables
with this exact source requires a fresh execution of Experiment~1.
"""
import time
import itertools
import json
import statistics as stats
from pathlib import Path

import numpy as np


# ---------------------------------------------------------------------
# Raw full adder: explicit 8-entry lookup table for the raw 3-wire
# patterns. This table is a representation model, not a physical
# hardware decode network.
# ---------------------------------------------------------------------
RAW_LUT = {}
for a, b, c in itertools.product((0, 1), repeat=3):
    s = a ^ b ^ c
    cout = (a & b) | (b & c) | (a & c)
    RAW_LUT[(a, b, c)] = (s, cout)

# ---------------------------------------------------------------------
# DRCC class table: 4-entry lookup table keyed only by Hamming weight.
# ---------------------------------------------------------------------
CLASS_LUT = {}
for w in range(4):
    # pick any representative raw pattern with that Hamming weight
    rep = next(p for p in RAW_LUT if sum(p) == w)
    CLASS_LUT[w] = RAW_LUT[rep]


def full_adder_raw(a, b, c):
    return RAW_LUT[(a, b, c)]


def full_adder_class(w):
    return CLASS_LUT[w]


# ---------------------------------------------------------------------
# T_structural_check: explicit N-cell structural representation.
# ---------------------------------------------------------------------
def structural_check(N):
    """Build and verify an explicit N-cell ripple-carry structure.

    The representation contains one full-adder cell per bit and explicit
    A/B/S/Cin/Cout port fields. The walk verifies every declared field of
    every cell: index, expected template tag, A, B, S, Cin, and Cout.
    It verifies the internally generated representation against its
    declared field structure. It is not an independent netlist or
    hardware-level verification and does not parse an external SPICE
    or gate-level netlist.
    Returns (ok: bool, n_checks: int).
    """
    if N < 1:
        raise ValueError("N must be a positive integer")

    cells = []
    for i in range(N):
        cells.append({
            "index": i,
            "template": "FULL_ADDER_CLASS_VERIFIED",
            "a": f"A[{i}]",
            "b": f"B[{i}]",
            "s": f"S[{i}]",
            "cin": "Cin" if i == 0 else f"Cout[{i-1}]",
            "cout": f"Cout[{i}]",
        })

    ok = True
```

```
    n_checks = 0
    for i, cell in enumerate(cells):
        expected = {
            "index": i,
            "template": "FULL_ADDER_CLASS_VERIFIED",
            "a": f"A[{i}]",
            "b": f"B[{i}]",
            "s": f"S[{i}]",
            "cin": "Cin" if i == 0 else f"Cout[{i-1}]",
            "cout": f"Cout[{i}]",
        }
        for key in ("index", "template", "a", "b", "s", "cin", "cout"):
            ok = ok and (cell[key] == expected[key])
            n_checks += 1

    return ok, n_checks


# ---------------------------------------------------------------------
# T_Classic: brute-force enumeration and check, vectorised with numpy
# so the exponential cost is genuinely paid in wall-clock time rather
# than being dominated by Python interpreter overhead per state.
# ---------------------------------------------------------------------
def classic_verify(N):
    """Enumerate all 2^(2N+1) inputs of an N-bit ripple-carry adder,
    simulate bit by bit, and check against integer arithmetic.
    Returns (ok: bool, n_states_checked: int)."""
    A = np.arange(1 << N, dtype=np.uint64)
    B = np.arange(1 << N, dtype=np.uint64)
    AA, BB = np.meshgrid(A, B, indexing="ij")
    AA = AA.reshape(-1)
    BB = BB.reshape(-1)
    n_pairs = AA.shape[0]

    ok_total = True
    n_checked = 0
    for cin in (0, 1):
        carry = np.full(n_pairs, cin, dtype=np.uint64)
        sum_bits = np.zeros((n_pairs, N), dtype=np.uint64)
        for i in range(N):
            a_i = (AA >> np.uint64(i)) & np.uint64(1)
            b_i = (BB >> np.uint64(i)) & np.uint64(1)
            s_i = a_i ^ b_i ^ carry
            c_i = (a_i & b_i) | (b_i & carry) | (a_i & carry)
            sum_bits[:, i] = s_i
            carry = c_i
        result = carry << np.uint64(N)
        for i in range(N):
            result = result | (sum_bits[:, i] << np.uint64(i))
        expected = AA + BB + np.uint64(cin)
        ok_total = ok_total and bool(np.all(result == expected))
        n_checked += n_pairs
    return ok_total, n_checked


# ---------------------------------------------------------------------
# T_DRCC: constant-size certificate (independent of N).
# ---------------------------------------------------------------------
def drcc_certify(N):
    """Constant-cost certificate independent of N.

    (1) Verify that the 4-entry Hamming-weight class table reproduces the
        8-entry raw full-adder table exactly.
    (2) Verify the eight local carry-state transitions by enumerating
        (a, b, carry_in) in {0,1}^3, computing the total weight
        w = a + b + carry_in, and checking that CLASS_LUT[w] returns the
        corresponding binary sum and carry outputs.

    N is accepted only so the function signature mirrors classic_verify;
    it is not used in the loop bounds. The check therefore has constant
    size with respect to N.
    Returns (ok: bool, n_states_checked: int)."""
    ok = True
    n_checked = 0
```

```
    # (1) base case: raw pattern -> class table must agree, for all 8
    # raw patterns (fixed size, independent of N).
    for (a, b, c), (s_raw, cout_raw) in RAW_LUT.items():
        w = a + b + c
        s_cls, cout_cls = CLASS_LUT[w]
        ok = ok and (s_raw == s_cls) and (cout_raw == cout_cls)
        n_checked += 1

    # (2) fixed-size carry-state transition check. The eight cases are the
    # eight local full-adder input triples (a, b, carry_in). The class index
    # is the total Hamming weight w=a+b+carry_in, so carry_in is not added a
    # second time after the class lookup.
    for a, b, cin in itertools.product((0, 1), repeat=3):
        total = a + b + cin
        expected_s = total & 1
        expected_cout = (total >> 1) & 1
        s, cout = CLASS_LUT[total]
        ok = ok and (s == expected_s) and (cout == expected_cout)
        n_checked += 1

    return ok, n_checked


def time_call(fn, N, repeats):
    times = []
    n_checked = None
    ok_all = True
    for _ in range(repeats):
        t0 = time.perf_counter()
        ok, n_checked = fn(N)
        t1 = time.perf_counter()
        ok_all = ok_all and ok
        times.append(t1 - t0)
    return ok_all, n_checked, times


def wiring_paths(N):
    """Return modeled decode-path counts under the stated independent-path model.

    These are representation-level counts, not measurements of physical wiring,
    hardware area, interconnect, or memory usage.
    """
    raw_paths = 8 ** N
    drcc_paths = 4 ** N
    return raw_paths, drcc_paths


def main():
    Ns_classic = [4, 6, 8, 9, 10, 11]  # raw = 2^(2N+1); N=10 -> 2.10e6, N=11 -> 8.39e6
    repeats = 5

    results = {"classic": [], "drcc": [], "structural": [], "wiring": []}

    for N in Ns_classic:
        ok, n_checked, times = time_call(classic_verify, N, repeats)
        raw_states = 1 << (2 * N + 1)
        results["classic"].append({
            "N": N,
            "raw_states": raw_states,
            "ok": ok,
            "n_checked": n_checked,
            "mean_s": stats.mean(times),
            "std_s": stats.stdev(times) if len(times) > 1 else 0.0,
            "times_s": times,
        })
        print(f"[classic] N={N:2d} raw_states={raw_states:>12,d} "
              f"mean={stats.mean(times):.4f}s std={stats.stdev(times) if len(times) > 1 else
      0.0:.4f}s ok={ok}")

    # DRCC certificate is O(1); still measured (not assumed) at every N
    # to show empirically that its wall-clock time does not grow with N.
    for N in Ns_classic:
        ok, n_checked, times = time_call(drcc_certify, N, repeats * 20)
```

```
        results["drcc"].append({
            "N": N,
            "n_checked": n_checked,
            "ok": ok,
            "mean_s": stats.mean(times),
            "std_s": stats.stdev(times) if len(times) > 1 else 0.0,
            "times_s": times,
        })
        print(f"[drcc]    N={N:2d} n_checked={n_checked:>12,d} "
              f"mean={stats.mean(times)*1e6:.2f}us std={(stats.stdev(times) if len(times) >
     1 else 0.0)*1e6:.2f}us ok={ok}")

    # Structural check: measured independently over the Q1 range, including N=256.
    Ns_structural = [4, 6, 8, 9, 10, 11, 16, 32, 64, 128, 256]
    for N in Ns_structural:
        ok, n_checked, times = time_call(structural_check, N, repeats * 20)
        results["structural"].append({
            "N": N, "n_checked": n_checked, "ok": ok,
            "mean_s": stats.mean(times),
            "std_s": stats.stdev(times) if len(times) > 1 else 0.0,
            "times_s": times,
        })
        print(f"[structural] N={N:3d} n_checked={n_checked:>6,d} "
              f"mean={stats.mean(times)*1e6:.2f}us std={(stats.stdev(times) if len(times) >
     1 else 0.0)*1e6:.2f}us ok={ok}")

    for N in [4, 6, 8, 10, 16, 32, 64]:
        raw_paths, drcc_paths = wiring_paths(N)
        results["wiring"].append({
            "N": N, "raw_decode_paths": raw_paths, "drcc_decode_paths": drcc_paths,
            "reduction_factor": raw_paths / drcc_paths,
        })
        print(f"[wiring]  N={N:2d} raw_paths={raw_paths:.3e} drcc_paths={drcc_paths:.3e} "
              f"reduction={raw_paths/drcc_paths:.3e}x")

    output_dir = Path(__file__).resolve().parent / "results"
    output_dir.mkdir(parents=True, exist_ok=True)
    output_file = output_dir / "exp1_full_adder.json"
    with open(output_file, "w") as f:
        json.dump(results, f, indent=2)
    print(f"Saved {output_file}")

    run_n32_extension(results["classic"])


# ---------------------------------------------------------------------
# N=32: constant-size certificate timing at nominal N=32.
#
#   - The constant-size certificate (drcc_certify) IS actually executed and
#     timed at nominal N=32. Since N does not enter the certificate loop, this
#     is a timing measurement of the same local certificate, not a 32-bit
#     structural or exhaustive verification.
#   - The exhaustive verifier is NOT executed at N=32: raw_states = 2^65.
#     The two uint64 meshgrid arrays AA and BB alone each require 2^67 bytes
#     (about 295 decimal exabytes); the sum_bits array in this implementation
#     would require still more memory. The N=32 exhaustive side is therefore
#     infeasible on a conventional machine. It is reported only as an
#     explicit least-squares projection through the six measured
#     (raw_states, mean_s) pairs, clearly separated from executed timings.
# ---------------------------------------------------------------------
def run_n32_extension(classic_results, out_path=None):
    N32 = 32
    repeats = 100
    ok, n_checked, times = time_call(drcc_certify, N32, repeats)
    drcc_mean = stats.mean(times)
    drcc_std = stats.stdev(times) if len(times) > 1 else 0.0

    xs = [row["raw_states"] for row in classic_results]
    ys = [row["mean_s"] for row in classic_results]
    k_fit = sum(x * y for x, y in zip(xs, ys)) / sum(x * x for x in xs)  # least squares
      through origin
    raw_states_32 = 1 << (2 * N32 + 1)
    projected_classic_s = k_fit * raw_states_32
    projected_classic_years = projected_classic_s / 3600 / 24 / 365.25
```

```
    out = {
        "N": N32,
        "raw_states": raw_states_32,
        "certificate_measured": {
            "mean_s": drcc_mean, "std_s": drcc_std, "n_checked": n_checked,
            "ok": ok, "repeats": repeats, "note": "actually executed and timed, same code
     path as N<=11",
        },
        "exhaustive_projected": {
            "fitted_per_state_cost_s": k_fit,
            "fit_source": (
                "least-squares fit through the origin of mean runtime against raw "
                "state count, using the six measured points for N in "
                "[4, 6, 8, 9, 10, 11]"
            ),
            "projected_mean_s": projected_classic_s,
            "projected_years": projected_classic_years,
            "note": (
                "NOT EXECUTED -- infeasible (2^65 states; the two uint64 "
                "meshgrid arrays alone require about 295 decimal exabytes each, "
                "with sum_bits requiring still more memory). This is a disclosed "
                "projection, not a measurement. The projected value is expressed "
                "as projected seconds and projected years."
            ),
        },
        "g_time_projected": projected_classic_s / drcc_mean,
    }
    print(f"[n32]     certificate mean={drcc_mean*1e6:.3f}us std={drcc_std*1e6:.3f}us "
          f"(measured, {repeats} trials)")
    print(f"[n32]     exhaustive projected={projected_classic_s:.3e}s "
          f"(projected years={projected_classic_years:.3e}) -- NOT executed")
    print(f"[n32]     G_time (projected) = {out['g_time_projected']:.3e}")

    if out_path is None:
        out_path = Path(__file__).resolve().parent / "results" / "exp1_n32_extension.json"
    else:
        out_path = Path(out_path)
        if not out_path.is_absolute():
            out_path = Path(__file__).resolve().parent / out_path
    out_path.parent.mkdir(parents=True, exist_ok=True)
    with open(out_path, "w") as f:
        json.dump(out, f, indent=2)
    print(f"Saved {out_path}")
    return out


if __name__ == "__main__":
    main()
```