Automated Proofs of Lattice Equations

Mikyle Shaelen Singh

Student Number: 2465557

Supervised by

Professor Clint Van Alten
Associate Professor Dmitry Shkatov

A research report submitted in partial fulfillment of the requirements
for the degree of Bachelor of Science with Honours in Computer Science

in the
School of Computer Science & Applied Mathematics
University of the Witwatersrand, Johannesburg

November 10, 2025

Abstract

This research report presents a unified, reproducible empirical study of four decision procedures for the equational theory of lattices, namely Whitman's recursive search, Freese's memoised recursion, Cosmadakis' closure via derivation graphs, and the height-stratified dynamic program of Hunt et al. Implementations share a common front end and run on both canonical DAG and tree representations structures. Using synthetically generated term pairs with controlled depth, arity, alternation, and sharing, we measure runtime, peak memory, and recursion/table activity across configuration families.

On alternating inputs, Freese overtakes Whitman at depth $\approx$ 6–7 and reaches roughly $300\times$ faster by the deepest points. Canonicalisation further removes subterm duplication, yielding additional speedups of up to $\sim 110\times$ for Freese (about $85\times$ on the C1 average over the tree representation), with moderate gains for Cosmadakis and Hunt and negligible effect on Whitman. Among the polynomial methods, Cosmadakis is competitive at moderate sizes but its bit-matrix footprint grows to $\sim 10^5$ MB (≈ 100 GB) at depth thereby negatively affecting its runtime, whereas Hunt offers a steadier low-GB memory profile at somewhat higher runtime. Two structural factors mainly explain rankings: (i) *sharing* strongly predicts performance for Freese by collapsing the memo domain and (ii) increasing *arity* penalises closure/table methods (Cosmadakis, Hunt) more than branching/memoised recursion (Whitman, Freese). Whitman remains competitive on thin spines and small same-polarity slabs.

The study bridges theory and practice by quantifying how representation and input structure govern efficiency, and it suggests guidance for procedure selection under resource constraints.

Declaration

I, **Mikyle Shaelen Singh**, hereby declare the contents of this research report to be my own work. This report is submitted in partial fulfillment of the requirements for the degree of Bachelor of Science with Honours in Computer Science at the University of the Witwatersrand. This work has not been submitted to any other university, or for any other degree.

Mikyle Shaelen Singh
November 10, 2025

Acknowledgements

I would like to sincerely thank my supervisors, **Professor Clint Van Alten** and **Associate Professor Dmitry Shkatov**, for their valuable guidance, feedback, and continuous support throughout the entirety of this research project. Their expertise and mentorship have been instrumental in shaping the direction and quality of the research project in its entirety.

I am also grateful to my peers for their insightful suggestions, review comments, and support throughout the research process and the preparation of this report.

Contents

Abstract i
Declaration ii
Acknowledgements iii
Table of Contents iv

1 Introduction 1

2 Theoretical and Algorithmic Foundations 2
2.1 Theoretical Background 2
2.2 Algorithmic Approaches to the Equational Theory of Lattices 3
2.2.1 Whitman's Recursive Algorithm 3
2.2.2 Freese's Polynomial-Time Optimisation 4
2.2.3 Cosmadakis' Graph-Based Proof System 4
2.2.4 Hunt et al: Complexity-Theoretic Characterization 5
2.2.5 Galatos' Proof-Theoretic Decision Procedure 5
2.2.6 Comparative Summary 6

3 Algorithmic Implementation and Data Representation 7
3.1 Term Representation and Canonicalisation 7
3.2 Expression Generation 8
3.3 Uniform Procedure Interface 9
3.4 Decision Procedures 9
3.4.1 Whitman: depth first branching without memoisation 9
3.4.2 Freese: top down recursion with memoisation 9
3.4.3 Cosmadakis: worklist-driven closure 10
3.4.4 Hunt-Rosenkrantz-Bloniarz: height stratified dynamic programming 10
3.5 Determinism, tie breaking, and fairness 11

4 Experimental Methodology 12
4.1 Objectives and Hypotheses 12
4.2 Input Families and Factors 12
4.2.1 Named configurations and parameter ranges 13
4.3 Measurement Protocol 13
4.4 Instrumentation and recorded metrics 13
4.5 Analysis methods 13
4.6 Environment and reproducibility 14
4.6.1 Final experimental configuration 14
4.6.2 Platform specification 14
4.7 Deliverables and figure generation 15
4.7.1 Companion DAG versus tree runs 15
4.8 Limitations 15

5 Results and Empirical Findings 16

5.1 Runtime performance comparison . . . 16
5.2 Factors affecting performance . . . 16
5.3 Memory profile . . . 19
5.4 Recursive work . . . 19
5.5 Representation effect: DAG versus Tree . . . 19
5.6 Budget scaling and dominance . . . 20
5.7 Summary . . . 21

6 Analysis and Discussion . . . 22
6.1 Depth scaling on alternating inputs . . . 22
6.2 Structural drivers: why shape matters . . . 23
6.2.1 Sharing versus runtime . . . 23
6.2.2 Arity and fan-out . . . 23
6.2.3 Per-algorithm extremes and their causes . . . 23
6.3 Memory behaviour . . . 23
6.4 Recursive work: concentration and collapse . . . 24
6.5 Representation effect: tree versus DAG . . . 24
6.6 Deviations from theory . . . 24
6.7 Budget scaling and dominance . . . 24
6.8 Further suggestions . . . 25
6.9 Validity, limits, and testable predictions . . . 25
6.10 Future directions . . . 26

7 Conclusion . . . 27

Bibliography . . . 27

A Decision Procedures Pseudocode . . . 29

B Code Availability . . . 33

C - AI Declaration Form . . . 33

Chapter 1

Introduction

Lattices, which are algebraic structures, arise naturally in various fields across mathematics and computer science, particularly in domains where ordering, hierarchy, or some structured combination of data is required. Such structures provide a framework of formal reasoning in several areas such as type theory, program analysis, language semantics, database theory, and distributed systems, thereby bridging a vital gap between abstract algebra and order theory. These mathematical structures formalize the intuitive notion of well-defined ordering, abstracting binary operations such as ”meet” and ”join”, which allow for systematic reasoning over partially ordered sets with unique least upper bounds and greatest lower bounds [1].

Despite their structural simplicity, determining whether a given lattice equation holds universally (i.e. across all lattices) is both a theoretically intricate and computationally challenging problem. The problem, known as the **equational theory of lattices**, is especially important in automated reasoning systems, where symbolic expressions must be validated mechanically. Manual approaches are not only tedious and error-prone but also infeasible at scale. Automated proof systems for lattice equations address this problem by offering efficient and reliable algorithms that reduce the burden of manual proof construction and verification. Such systems also enable the systematic coverage of complex equation spaces that would otherwise be impractical to traverse manually, underpinning foundational aspects of automated theorem provers and symbolic computation engines integral to verifying critical software systems.

Foundational work, such as that of Whitman, introduced a recursive decision procedure for evaluating inequalities in free lattices [2], while also establishing canonical forms and covering relations. Subsequent works by Freese [3], Cosmadakis [4], Hunt et al. [5], and more recently, Galatos [6], progressively refined these ideas — ranging from optimizing decision procedures and exploring graph-based representations, to formal proof systems establishing the decidability of the equational theory.

This research project implements and empirically compares four automated decision procedures for lattice equations in a unified infrastructure, quantifying how input depth, arity, sharing, and representation (DAG vs. tree) shape runtime and memory. The remainder of this report is organised as follows. *Theoretical and Algorithmic Foundations* reviews background and algorithms, Chapter 3 details the implementation and representations, Chapter 4 sets out configurations and measurement, Chapter 5 reports results and Chapter 6 provides an in-depth analysis and discussion of the decision procedures. The report concludes with a brief conclusion and directions for future work.

Chapter 2

Theoretical and Algorithmic Foundations

2.1 Theoretical Background

Lattice theory stems from the study of ordered structures and their algebraic properties.

Consider the notion of a partially ordered set. That is, a set P equipped with some binary relation $\leq$ that satisfies **reflexivity**($a \leq a$), **antisymmetry** ($a \leq b$ and $b \leq a$ implies $a = b$), and **transitivity** ($a \leq b$ and $b \leq c$ implies $a \leq c$). The set P is usually referred to as an ordered set, or a **poset** [2]. The above properties define a partial ordering i.e. not every pair of elements needs to be comparable (unlike in total orders), which is key to capturing structures and maintaining order relationships.

Within a poset $(P, \leq)$, let there exist a subset S, that is, $S \subseteq P$. There exists a set of upper bounds and a set of lower bounds of S, defined as follows:

$$\text{Upper Bound Set: } \{x \in P \mid \forall s \in S,\ s \leq x\}$$
$$\text{Lower Bound Set: } \{x \in P \mid \forall s \in S,\ x \leq s\}$$

Since $\leq$ is transitive, the set of upper bounds of S (denoted $S^{\uparrow}$) forms an up-set, and the set of lower bounds ($S^{\downarrow}$) forms a down-set. If $S^{\uparrow}$ has a least element x, then x is the least upper bound or **supremum** of S, written $\sup S$. Likewise, if $S^{\downarrow}$ has a greatest element x, then x is the greatest lower bound or **infimum**, written $\inf S$.

Therefore, the supremum and infimum for these subsets can formally be defined [1]:

$$\sup S \coloneqq \min\{x \in P \mid \forall s \in S,\ s \leq x\}$$
$$\inf S \coloneqq \max\{x \in P \mid \forall s \in S,\ x \leq s\}$$

Such elements provide a structured way of combining and comparing elements for larger ordered structures.

As Freese [3] demonstrates, the poset P can define a lattice. A lattice is a poset in which every pair of elements $a, b \in P$, has both an infimum (inf) and a supremum (sup), denoted $a \wedge b$ (meet) and $a \vee b$ (join), respectively, as defined in [1]. Such operations enable systematic reasoning over ordered sets, and their existence for every pair of elements generates a finite combination of operations that exists between elements of a set. The finite combination of operations for every pair of elements generates the poset into a lattice.

Formally, lattices are generally treated algebraically as structures $\langle L, \wedge, \vee \rangle$, where the binary operators satisfy the following identities:

- **Commutativity:** $a \vee b = b \vee a, \quad a \wedge b = b \wedge a$

- **Associativity:** $a \vee (b \vee c) = (a \vee b) \vee c, \quad a \wedge (b \wedge c) = (a \wedge b) \wedge c$
- **Idempotency:** $a \vee a = a, \quad a \wedge a = a$
- **Absorption:** $a \vee (a \wedge b) = a, \quad a \wedge (a \vee b) = a$

The equational properties not only define the lattice but are also central in the algorithms that attempt to prove whether certain lattice equations are valid. Building on this algebraic view, lattice expressions can be formed, i.e. symbolic expressions composed of variables joined by the binary operators $\langle \wedge, \vee \rangle$. A typical objective in this domain is to determine whether two terms of an equation are equivalent ($s = t$) or an inequality $s \leq t$ holds under the semantics of all lattices - a universal validity. This objective defines the **equational theory of lattices**, which decides whether an expression such as

$$(x \wedge y) \vee z = (x \vee z) \wedge (y \vee z)$$

holds in every lattice, or only in specific lattices. For instance, the aforementioned equation holds in distributive lattices but not all lattices [1]. Such theory forms the basis of automated reasoning over symbolic lattice terms, with varying applications.

To address equational problems without reference to a specific model, the concept of a free lattice is used. A **free lattice** is the "most general" lattice generated from a set of variables with no relations beyond the lattice axioms. In a free lattice, if two expressions are equal, they are equal in all lattices. Such an abstraction is very useful for algorithmic decision methods since free lattices allow decision problems to be studied abstractly and operated on symbolically, without referring to a particular concrete lattice. If an inequality $s \leq t$ holds in the free lattice, then it holds in all lattices. For example, to verify whether $(x \wedge y) \vee z = (x \vee z) \wedge (y \vee z)$ holds universally, one can check if it holds in the free lattice, as demonstrated by Whitman [2]. Thus, algorithmic methods often focus on manipulating and comparing expressions within free lattices. Whitman [2] provided an algorithm to decide such inequalities in free lattices, using recursive rules to explore valid derivations and establishing canonical forms as minimal representations of lattice expressions, which was later optimized by Freese [3] and extended by Cosmadakis [4] to broader decision problems.

2.2 Algorithmic Approaches to the Equational Theory of Lattices

The equational theory of lattices, which asks whether an equation or inequality holds universally across all lattices, has motivated the development of a sequence of increasingly efficient decision procedures. This section reviews the most significant algorithmic approaches.

2.2.1 Whitman's Recursive Algorithm

The foundational work of Whitman [2] introduced the first general decision procedure for lattice inequalities in the setting of free lattices. In this model, all valid lattice equations derived from the axioms of lattice theory are preserved - no additional identities are assumed. The key problem is to determine whether a term A is less than, or equal to, another term B, written $A \leq B$, in the free lattice, which implies that the inequality holds in all lattices [2].

Whitman's method decomposed the lattice terms using a set of structural conditions - known as Whitman's conditions. For instance,

- If $A = A_1 \vee A_2$, then $A \leq B$ **iff** $A_1 \leq B$ **and** $A_2 \leq B$
- Similarly, if $A = A_1 \wedge A_2$, then $A \leq B$ **iff** $A_1 \leq B$ **or** $A_2 \leq B$

The rules were similar for dual forms, capturing associativity, distributivity, and absorption behaviour [3].

These recursive syntactic rules decompose the comparison into checks over smaller subterms. However, although the procedure is sound, terms may be nested and reused in structurally identical problems. Hence the algorithm suffers from severe computational inefficiency since the number of recursive checks tends to grow exponentially with the size of the terms. Freese later formalized this by demonstrating that even a simple worst-case implementation of Whitman's procedure (alternating meet-join constructions) can perform up to 2^n checks for terms with n deep-nested subterms [3]. Nonetheless, Whitman's methods remains significant as it provides a generalized criterion for universal validity in lattices, and it establishes the conceptual foundations on which later, more efficient algorithms have been built.

2.2.2 Freese's Polynomial-Time Optimisation

Freese [3] provided a major improvement to Whitman's algorithm by introducing memoization to avoid redundant subterm analysis. He noted that in Whitman's recursive structure, the same comparisons such as $A_i \leq B_j$, would often recur in multiple branches of the recursion tree. Therefore, Freese proposed caching the results of the comparisons to avoid re-computation. This yielded a dynamic programming-style algorithm that operated on a table of previously computed subresults.

Freese's insight was to treat term comparisons as nodes in a term-product graph, where each node represents a subproblem pair (A, B) of subterms being compared (i.e. whether $A \leq B$). By storing the result of each $A \leq B$ test in a lookup table, the algorithm would avoid recomputing the same value redundantly. Freese demonstrates that this modified version reduces the worst-case time complexity from exponential to polynomial, specifically $O(n^2)$, where n denotes the number of distinct subterm comparisons. Moreover, the algorithm can be implemented with basic pattern matching and lookup strategies.

Freese also observed that, in practice, the algorithm can perform significantly better on structured or sparse inputs. If a subterm ordering comparison fails early on, the search tree can be pruned. Thus, the best-case runtime is essentially constant time, specifically $O(1)$, and performance is heavily dependent on the input terms structure. However, there is a space complexity tradeoff: the memoization table requires space proportional to the number of unique subterm pairs, which could pose a limitation for larger values of n. Additionally, the implementation of a dynamic programming-style algorithm has introduced some additional complexity. Thus, the algorithm's effectiveness primarily depends on efficient table implementation.

Freese's implementation effectively preserves the recursive benefits of Whitman's approach while drastically improving efficiency. However, while sufficient for moderately sized lattices, it can become limiting in large or highly recursive structures [3].

2.2.3 Cosmadakis' Graph-Based Proof System

Cosmadakis [4] took a fundamentally different approach to the equational theory problem: introducing a proof-theoretic and graph-theoretic framework for solving the uniform word problem for lattices. He formulated a complete system for deciding the uniform word problem for lattices: given a set E of equations and a target equation $s = t$, decide whether $E \models s = t$.

Rather than evaluating comparisons recursively, Cosmadakis proposed constructing a directed derivation graph whose nodes are subterms and edges represent inference rules derived from the lattice axioms - lattice inference rules [4] such as:

- $p \leq q, p' \leq q \Rightarrow p \vee p' \leq q$
- $s \leq p, s \leq q \Rightarrow s \leq p \wedge q$
- If $p = q \in E$, then $p \leq q$, etc.

can be encoded as rewrite rules over the graph. The inference rules are iteratively applied to propagate valid inequalities through the graph, and hence generate derivable inequalities among subterms. Determining whether an equation $s \leq t$ holds is now reduced to testing whether there exists a directed path (arc sequence) from s to t in the graph.

Cosmadakis' approach is shown to solve the uniform word problem in polynomial time by operating within a fixed-size subterm space. It also uses logarithmic space, making it log-space complete for P i.e. the algorithm only requires $O(\log n)$ working memory (allows for efficient scalability to larger n inputs). Cosmadakis also extends the framework to the generator problem, which asks whether a term lies in the sublattice generated by a given set of terms. This generator problem also has a log-space complexity bound, especially under different shared or reused input representations (eg. DAG (directed acyclic graph)-based term representations [4]).

Relative to Freese's [3] approach, Cosmadakis' approach minimizes memory consumption and avoids explicit dynamic programming structures. However, while it tends to incur some construction overhead during graph generation, the graph representation simplifies the implementation details in comparison to Whitman's [2]. The approach is favourable for memory-constrained systems or highly reusable subterm graphs, where implications can be determined quickly through graph traversal.

2.2.4 Hunt et al: Complexity-Theoretic Characterization

Hunt, Rosenkrantz, and Bloniarz, contributed a comprehensive complexity classification of various lattice decision problems. Rather than proposing a specific decision procedure, they focused on delineating the boundaries between tractable and intractable problems. Hunt et al. examined the equivalence problem ($F = G$), the inequality problem ($F \leq G$), and the operator minimization problem (finding an equivalent expression with the fewest operations). They found that the general lattice equivalence problem is coNP-hard, even for constant-free expressions on both finite and distributive lattices [5]. Hunt et al. proceeded to introduce an efficient expressibility condition C, under which these problems become coNP-hard. The condition C characterizes when a lattice supports compact encoding of hard logical problems. As such, they demonstrate that any finite or distributive lattice satisfying C admits reductions from classic coNP-complete problems (such as DNF tautology testing). However, despite these findings, Hunt et al. identified special cases where the problems become tractable:

- When lattice expressions are restricted to disjunctive normal form (DNF) or conjunctive normal form (CNF), equivalence testing becomes polynomial-time solvable.
- For constant-free expressions in free lattices with three or more generators, simplification and equivalence are efficient.

Hunt et al. further demonstrate that approximate minimization, which simplifies expressions within a given operator range, is as hard as exact minimization, implying approximations are coNP-hard with the exemption for when $P = NP$ (eg. free lattices with ≥ 3 generators [5]).

The findings from Hunt et al. place earlier algorithmic approaches in context. While Whitman, Freese, and Cosmadakis' offer efficient solutions for structured inputs, they are not universally applicable since structure and representation matter. General cases require caution, as they may be intractable without further restrictions.

2.2.5 Galatos' Proof-Theoretic Decision Procedure

Galatos [6] proposed a formal minimalistic proof-theoretic system, called **Lat**, for reasoning about lattice equations using deductive inference rules. The system defines derivability $s \leq t$ through standard logical rules such as reflexivity, associativity, and join-meet distributivity, and inference rules such as:

- $s \leq t, t \leq r \Rightarrow s \leq r$ (transitivity)

- $s \leq r, t \leq r \Rightarrow s \vee t \leq r$, etc. [6]

The main foundation of the framework is the subsytem **Lat**$^-$, which omits the transitivity rule. Galatos proves that this smaller system is complete (for the equational theory) and terminating, and is thus decidable [6]. Every valid equation is derivable in **Lat**$^-$, and all derivations terminate. This provides a decidable proof framework, independent of algorithmic representations.

The construction defines derivability as a binary relation $s\mathcal{N}_t$, and constructs a closure operator γ over sets of terms. The closure algebra, when formed this way, is itself now a complete lattice Tm^+, and the term $s \leq t$ holds if and only if $s \in \gamma(\{t\})$. This demonstrates a deep connection between logical derivability and lattice-theoretic structure.

Though not intended for practical implementation, Galatos' **Lat**$^-$ system is a viable alternative to recursive or graph-based decision procedures. It provides a foundational framework that complements computational approaches, offering a proof-theoretic solution for lattice equations that bypasses term rewriting and graph traversal.

2.2.6 Comparative Summary

The following table summarizes the complexity characteristics of the approaches discussed in preceding subsections.

Algorithm	Time Complexity	Space Complexity
Whitman (1941)	$O(1)$ (best), $O(2^n)$ (worst)	Low
Freese (1987)	$O(1)$ (best), $O(n^2)$ (worst)	$O(n^2)$
Cosmadakis (1988)	$O(\lvert\mathrm{sub}(u)\rvert \, \lvert\mathrm{sub}(v)\rvert \, \bar{k})$	$O(\lvert\mathrm{sub}(u)\rvert \, \lvert\mathrm{sub}(v)\rvert)$
Hunt et al. (1987)	$O(\lvert\mathrm{sub}(u)\rvert \, \lvert\mathrm{sub}(v)\rvert \, \bar{k})$ on restricted forms	$O(\lvert\mathrm{sub}(u)\rvert \, \lvert\mathrm{sub}(v)\rvert)$
Galatos (2024)	Derivation length bounded by rule depth	Proof search state exponential in worst-case depth

Notes: Freese achieves best-case constant time via early pruning. Cosmadakis and Hunt both operate over unique subterms; $\bar{k}$ denotes average branching. Hunt et al.'s bounds apply to their efficiently expressible fragments (e.g., CNF/DNF). Galatos' proof system terminates with derivations whose depth bounds the search space; explicit complexity depends on the chosen proof strategy.

Table 1: Comparison of lattice equation decision methods by time and space complexity.

Each method offers a distinct viewpoint: recursive decomposition (Whitman), optimisation through caching (Freese), derivation graphs (Cosmadakis), complexity boundary analysis (Hunt et al.), and proof-theoretic derivability (Galatos). These approaches provide a balance of general applicability, computational efficiency, and foundational clarity. Depending on the intended use case - whether algorithmic implementation, formal proof verification, or theoretical complexity analysis - different trade-offs will be appropriate.

Chapter 3

Algorithmic Implementation and Data Representation

This chapter describes the implementation infrastructure supporting the empirical comparison of four decision procedures for the word problem in free lattices, namely Whitman [2], Freese [3], Cosmadakis [4], and Hunt et al. [5]. The chosen data representation structures and expression generators are presented first, establishing the framework within which the four algorithms operate. High-level algorithmic implementations are then provided and discussed.

3.1 Term Representation and Canonicalisation

We implement two internal representations in order to isolate the cost and benefit of subterm sharing.

Canonical DAG representation: Terms are interned into a directed acyclic graph where each structurally unique subterm is represented by a single node that is shared by all occurrences. Canonicalisation enforces the lattice identities needed for structural sharing, namely flattening of same kind operations (associativity), stable sorting of children (commutativity), duplicate elimination (idempotence), and hash consing into a unique node pool. Variables are mapped deterministically to integer symbols upon first sight, and this mapping is reused across all experiments. These invariants guarantee that equivalent inputs yield byte identical node identities, which maximises the effectiveness of memoisation and closure style algorithms.

Tree representation: To measure the cost of not sharing subterms, we also build a classical abstract syntax tree with no hash consing. Multi-ary joins and meets are expanded into left associated binary chains in order to mimic pointer based implementations. In this mode two isomorphic subtrees have different identities, which defeats DAG level sharing. The same front end parser and variable mapping are used as in DAG mode in order to keep input semantics identical.

Parsing and determinism Inputs are represented as S-expressions (Lisp-style prefix notation) over $\{\vee, \wedge\}$ with variables drawn from a fixed alphabet. For example, the lattice term $(x \wedge y) \vee z$ serialises as $(\vee(\wedge xy)z)$. The parser is deterministic, so whitespace and tokenisation cannot change the variable to symbol mapping or the child ordering after canonicalisation. An optional knob *ensure unique leaves* replaces repeated variables at leaves with fresh symbols when we need to suppress accidental sharing.

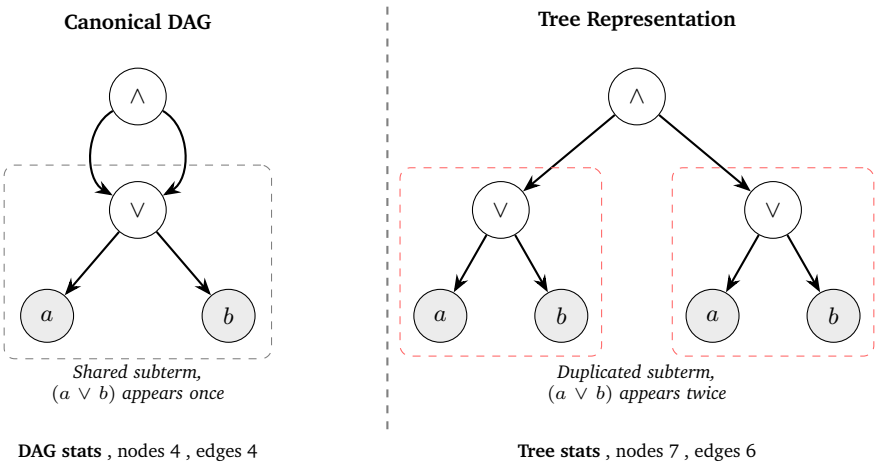

Figure 1: Canonical DAG mode shares the subterm $(a \vee b)$ inside $(a \vee b) \wedge (a \vee b)$, while a tree representation duplicates it. Shared structure reduces both memory traffic and the number of distinct subproblems exposed to memoised or closure style procedures.

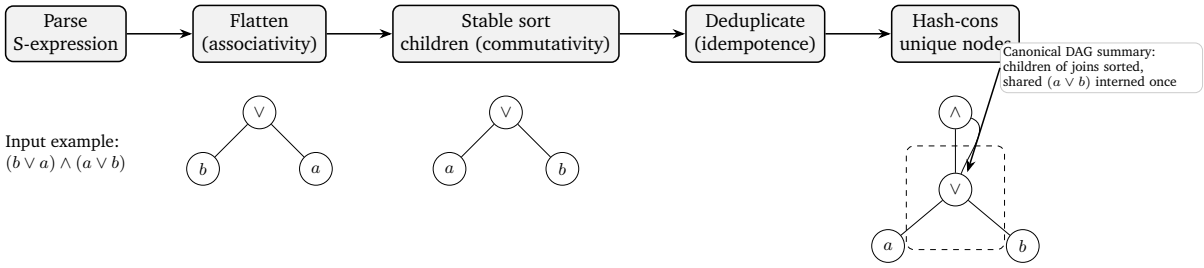

Figure 2: Canonicalisation pipeline: parse, flatten, stable sort, deduplicate, then intern to a unique node pool. Two occurrences of $(a \vee b)$ collapse to one DAG node.

3.2 Expression Generation

We use a configurable generator to produce pairs (u, v) with controlled structure. The generator is implemented as a small library that emits S-expressions and metadata, and it is called from the harness that runs each engine. The design goals are reproducibility, separation of concerns, and the ability to dial specific structural features without entangling them with the decision logic.

Construction modes: Two complementary construction modes are provided. In height mode, the caller specifies a target depth and the generator recurses until the depth reaches zero, which produces balanced shapes when arity is held fixed. In budget mode, the caller specifies a target count of internal nodes, and the generator maintains a work queue of partially built subtrees while allocating the remaining budget across children in proportion to simple heuristics, which produces variable width trees that match the requested size.

Structural controls: The generator exposes a small set of orthogonal knobs. Root polarity can be fixed so that u starts with a meet and v starts with a join (or the converse), which forces the critical branching case for Whitman style recursion. An alternation index controls the probability of flipping the operator along an edge, which lets us move between long homogeneous runs and frequent operator changes. Arity can be fixed or sampled from a discrete distribution, which changes branching factor without changing depth. The variable alphabet size controls the chance of repeated leaves, which influences the amount of structural sharing after canonicalisation. The *ensure unique leaves* option can be enabled to eliminate accidental sharing at leaves when desired.

Output and annotations: Each generated pair is serialised as an S expression and accompanied by lightweight annotations that are computed at construction time. These include node counts, observed depth, an alternation estimate, and a coarse sharing ratio in canonical mode. The annotations are consumed by the measurement harness, while the parameter grids and concrete settings are defined in Chapter 4.

3.3 Uniform Procedure Interface

All engines expose `leq(u, v)` on a common node interface. Engines do not mutate the term graph, do not rely on representation specific side effects, and do not require per engine preprocessing beyond reading node kind (variable, join, meet) and iterating over children in the stable canonical order. This keeps comparisons fair across representations and procedures.

3.4 Decision Procedures

We summarise each procedure as implemented and we point to its pseudocode in Appendix A. Throughout this section, U denotes the multiset of children of a join node, and V denotes the multiset of children of a meet node.

3.4.1 Whitman: depth first branching without memoisation

We implement Whitman as a pure depth first backtracker that applies the structural rules directly to (u, v) without any caching (see Algorithm 1). After discharging trivial equalities and symbol containment checks, deterministic directions are taken greedily, namely if $u = \bigvee U$ we must establish $u_i \leq v$ for all $u_i \in U$, and if $v = \bigwedge V$ it suffices to show $u \leq v_j$ for some $v_j \in V$. The only genuine nondeterminism arises at opposite polarity where u is a meet and v is a join. We fix a branch order that first explores the right side, that is $u \leq v_j$ for each $v_j \in V$, then explores the left side, that is $u_i \leq v$ for each $u_i \in U$. This ordering is chosen to make runs deterministic across seeds and to keep the comparison with Freese clean. Termination is guaranteed because every recursive call strictly reduces the height of u or of v, we short circuit as soon as a disjunct succeeds, and we abandon a branch immediately upon a base case contradiction. Because we intentionally do not cache, identical subproblems reappear even in DAG mode, and they reappear more often in tree mode. Counters recorded for this procedure include `pairs_visited`, `max_stack`, and branching statistics, and Chapter 4 explains how these drive the plots.

3.4.2 Freese: top down recursion with memoisation

Freese augments Whitman style recursion with a memo table on canonical pairs (u, v), which turns repeated subproblems on DAGs into table lookups and collapses the search to polynomial time in the number of unique subterms (see Algorithm 2). Before any recursive work, we query a flat hash map keyed by a stable pair of node identities, on a miss we perform the same structural case analysis as in Section 3.4.1, we store the boolean result, and we return it. We cache booleans rather than a tri state value because on a finite DAG the recursion is acyclic in the product poset of subterms, which removes the need for an in progress marker. We keep Whitman branch order so that any differences in behaviour can be attributed to memoisation rather than to search heuristics.

Representation matters: in DAG mode the number of distinct questions is $|\mathrm{sub}(u)| \cdot |\mathrm{sub}(v)|$, in tree mode isomorphic subtrees have different identities, which lowers hit rates and pulls the procedure back toward Whitman. In code the switch is gated by representation, since we enable memoisation with `(use_memo = I && I->canonicalize)`, therefore in tree mode distinct node identities cause hits to collapse and behaviour approaches Whitman. Counters include `recursive_calls`, `max_stack`, `memo_hits`, `memo_size`, and branch attempt or success counts, with usage described in Chapter 4.

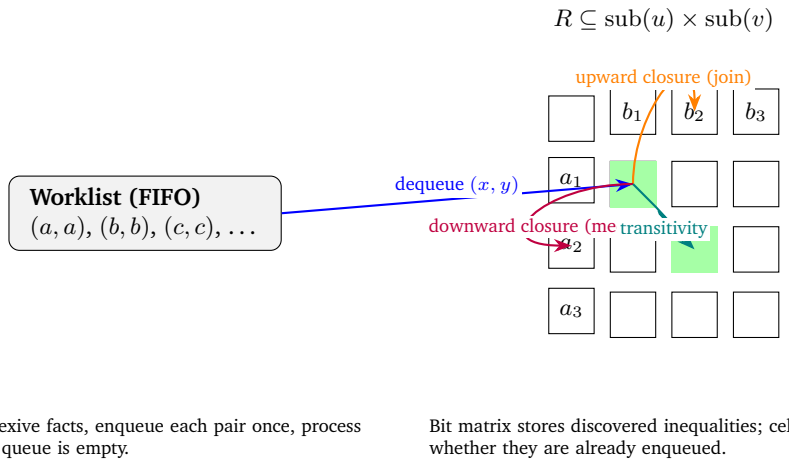

Figure 3: Cosmadakis as worklist saturation over a bit matrix. Processing (a, b) may add consequences by upward closure along join, downward closure along meet, and transitivity.

3.4.3 Cosmadakis: worklist-driven closure

Cosmadakis is implemented as saturation of a binary relation $R \subseteq \mathrm{sub}(u) \times \mathrm{sub}(v)$ using a worklist (see Algorithm 3). We enumerate unique subterms of u and v and we allocate a bit matrix for R. We then seed reflexive facts, and we push those seeds into a FIFO queue. Processing a pair (a, b) may add new facts by three local mechanisms, namely upward closure along joins where if b is a child of a join c we infer (a, c), downward closure along meets where if (a, c) holds for all children c of a meet d we infer (a, d), and transitivity where we compose through intermediates already known in the matrix. Each cell carries an enqueued mark so we never push the same item twice, and the matrix deduplicates facts, which makes the loop strongly normalising. We choose a FIFO queue because locality tends to make newly created consequences useful immediately, which reduces cold cache probes into the matrix, while still giving a fair traversal of the inference graph.

This design pays for certainty with memory, the state is quadratic in the number of unique subterms, which is why we prefer DAG mode when possible. In tree mode the matrix inflates with duplication and both time and memory scale accordingly. The practical runtime is governed by the sparsity pattern of R and by mean arity. We record worklist size statistics and relation density, and Chapter 4 explains how these statistics flow into the analysis.

3.4.4 Hunt-Rosenkrantz-Bloniarz: height stratified dynamic programming

The Hunt Rosenkrantz Bloniarz procedure is a bottom up dynamic program that fills a truth table $T[a, b]$ in an order aligned with the dependency structure of joins and meets (see Algorithm 4). We compute heights $h(\cdot)$ as distance to the nearest leaf for all unique subterms, and we process pairs in nondecreasing lexicographic order of $(h(a), h(b))$. With this schedule every entry depends only on previously filled entries, namely if $a = \bigvee_i a_i$ then $T[a, b]$ is the conjunction of $T[a_i, b]$, and if $b = \bigwedge_j b_j$ then $T[a, b]$ is the conjunction of $T[a, b_j]$, with base cases for identity and for variable containment. The height order removes any recursion, it guarantees single pass evaluation per cell, and it improves cache behaviour because rows and columns of similar structural depth are touched together. We store T as a compact bit set indexed by canonical node identities in order to keep memory movement predictable.

As with Cosmadakis, the table is quadratic in the number of unique subterms in DAG mode, and larger in tree mode. Unlike Freese, memoisation is implicit in the table, and there is no branch scheduling to tune. The effective cost is $O(|\mathrm{sub}(u)| \cdot |\mathrm{sub}(v)| \cdot \bar{k})$ where $\bar{k}$ is average arity. We record table dimensions and per height fill counts, and Chapter 4 explains how these are utilised.

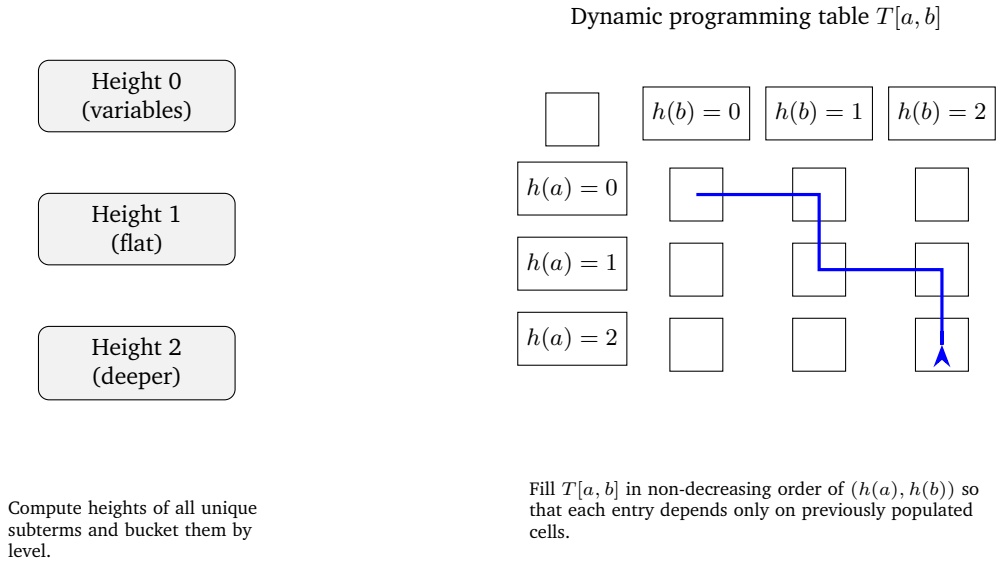

Figure 4: Hunt height stratification and table sweep. Each cell depends only on entries with smaller heights.

3.5 Determinism, tie breaking, and fairness

All procedures use the same stable child order based on canonical node identity, and the same branch ordering in Whitman and in Freese where we explore right then left. The parser symbol assignment is deterministic, and all generator randomness is seeded in a single place in the harness in order to make runs reproducible. We avoid representation specific optimisations beyond what is intrinsic to each method, for example memoisation in Freese, or bit matrix layout in Cosmadakis. This keeps the comparison focused on algorithmic behaviour rather than engineering details.

This chapter fixes representation choices, input construction at a high level, and decision logic interfaces, and it explains why we made these choices. The next chapter specifies the generator parameter grids, the configuration families, the instrumentation and measurement protocol, and how the recorded counters drive the plots in the results.

Chapter 4

Experimental Methodology

This chapter specifies how we generate inputs, run the four decision procedures, and analyse outcomes in a way that is reproducible and fair across algorithms and representations. Each methodological choice is tied to questions answered in the Results and Analysis chapters.

4.1 Objectives and Hypotheses

We evaluate four procedures on families of inputs that stress different structural factors. The research study addresses the following hypotheses:

- **H1:** Structural sharing reduces runtime primarily for memoised recursion and branching search: as sharing increases, median runtime drops sharply for Freese and Whitman, with weaker effects on Cosmadakis and Hunt.
- **H2:** Branching factor changes the relative ranking. Higher arity penalises Cosmadakis and Hunt more than Whitman or Freese.
- **H3:** There is a crossover depth where a polynomial method becomes faster than Whitman under full alternation.
- **H4:** Dominance depends on structure. No single algorithm wins on every configuration.
- **H5:** Memory grows with the number of unique subterms and with arity.

4.2 Input Families and Factors

Parsing, canonicalisation, and internal representations are defined in Sections 3.1 and 3.2. We do not repeat such details here. We vary the following independent factors when generating pairs (u, v).

- **Size:** measured as depth in the full alternating family or as a budget of internal nodes in budget families.
- **Arity:** fixed k or a small discrete range.
- **Alternation:** fraction of edges that flip operator.
- **Root polarity:** opposite or same at the roots.
- **Variable alphabet size:** which affects structural sharing under canonicalisation.
- **Representation mode:** canonical DAG or tree.

Table 2: Configuration families and parameter ranges. Depth is exact in C1. Budgets are target internal node counts in C2 through C6.

Config	Shape	Size parameter	Arity	Alternation	Root polarity
C1	full alternating	depth 1 to 16	2	full	opposite
C2	alternating budget	budgets $\{50, 100, 200, 400, 800\}$	2 to 4	high	opposite
C3	alternating budget	budgets $\{50, 100, 200, 400, 800\}$	2 to 4	high	same
C4	balanced sharing sweep	budgets $\{50, 100, 200, 400\}$	2 to 4	low to high	both
C5	left or right spine	budgets $\{20, 40, 60\}$	chain	low	both
C6	fixed arity	budgets scaled per k	$k \in \{2, 3, 4, 6\}$	high	opposite

4.2.1 Named configurations and parameter ranges

We refer to six named configurations that combine the factors above. Table 2 lists the parameter ranges used. Budget is a structural knob and it is not complexity equivalent across shapes, therefore cross shape overlays are descriptive only.

4.3 Measurement Protocol

For each configuration and size point, we evaluate the four procedures on the number of pairs recorded in the generator metadata. Every pair is evaluated in both directions $(u \leq v)$ and $(v \leq u)$ and we record per direction outcomes separately. For validation we ensure that all 4 decision procedures agree with the same result (i.e. $(u \leq v)$ and $(v \leq u)$ must yield the same outcomes for all 4 algorithms). We use a monotonic clock, report microseconds, and we take the median over pairs at the same size point. The order of algorithms is fixed. The order of pairs is deterministic and seed controlled. We also run a short warmup prior to measurement. A run is valid when the procedure returns a boolean verdict, the counters are internally consistent, and the timing is finite. For full alternating inputs we cap depth at the largest point that completes within the global time budget for Whitman on the chosen platform and we apply the same cap across representations.

4.4 Instrumentation and recorded metrics

Each procedure logs a shared core of fields in a per pair CSV together with counters specific to its evaluation strategy. The shared core includes runtime in microseconds, peak working set per direction, and generator annotations such as alternation index, sharing ratio, observed pair depth, and budget. Per procedure counters are defined in Sections 3.4.1 through 3.4.4 and include, for example, recursive activity and branch statistics for Whitman and Freese, memo hits and table size for Freese, and relation or table occupancy summaries for Cosmadakis and Hunt.

We aggregate by median (to reduce skewness generated by means) at each size point and display interquartile ranges when relevant. All figures in the Results chapter are generated from these CSV files by a single analysis script.

4.5 Analysis methods

We apply the following methods, each tied to a hypothesis.

- **C4 node sharing analysis** for H1: Scatter of sharing ratio versus $\log_{10}$ runtime with a separate colour per algorithm and a non parametric smooth. We report Spearman correlation and a p value per algorithm. Smoothing uses LOWESS as implemented in the plotting script.
- **C6 arity grid** for H2: Four panels for $k \in \{2, 3, 4, 6\}$ with identical axes and a panel annotation for median observed pair depth.
- **C1 crossover annotation** for H3: On the log scale plot we place a vertical marker at the first depth where an algorithm becomes faster than Whitman, estimated by linear interpolation in log space.
- **Winner map** for H4: For each configuration we compute the fraction of size points where each algorithm is the fastest. The main figure uses a strict argmin policy, a tolerance variant within five percent is provided as a supplementary figure.
- **Budget scaling overlay** for structural comparison across shapes: We overlay representative configurations on a log scale and state the caveat that budget is not directly comparable across shapes.
- **Memory profile** for H5: Peak working set versus size for the baseline configuration on linear and log scales.

4.6 Environment and reproducibility

We build all binaries with a single toolchain and a single set of optimisation flags. We run on a fixed machine profile, pin processes to an exclusive set of cores, use a monotonic clock, and all randomness is seeded. The parser is deterministic, the canonical node interner is global within a process and cleared between experiments that are not intended to share state. Each figure is produced by a script that takes a results root and writes a directory of images and tables, listing its inputs and a commit hash. Raw CSV files and generated images are archived with the report. All code and the generator are version controlled.

4.6.1 Final experimental configuration

The plots and tables in Chapters 5–6 are drawn from two scheduled batches:

- **Canonical DAG:** For configuration C1 (fully alternating), depths 1–11 use 50 expression pairs per depth per direction; depths 12–16 retain five pairs in order to maintain reasonable experimental runtimes. The best- and worst-case suites (E_bestcase/E_worstcase) contribute 30 pairs per depth; E_worstcase is similarly thinned to five pairs at depths 12–16. Configurations C2–C6 and the three sharing sweeps use the budget grids listed in Table 2 with 30 pairs per size point. Various metrics such as medians, means, and IQR's are computed for the relevant metrics to ensure robust results.
- **Abstract Syntax Tree:** Each configuration is rerun in tree representation mode with the identical seeds and pair budgets. Depth ranges for C1 in this mode stop at $d = 14$ because the tree representation amplifies Whitman's and Cosmadakis runtime and memory usage. Best/worst suites again provide 30 pairs per depth up to $d = 14$, with five pairs kept beyond that where applicable.

Across both batches we therefore analyse 15 main configuration points (C1–C6 plus sharing extremes and spine variants) and two extremal suites, totalling a significant number evaluated pairs on DAGs and the same number on tree representations. No additional post-hoc filtering was applied.

4.6.2 Platform specification

All experiments were run on a single node of the *bigbatch* partition. Nodes have an INTEL CORE I9 10940X CPU with 14 cores, and 128 GB of system memory. The operating system is UBUNTU 22.04.3 LTS with LINUX KERNEL 5.15.0 43 GENERIC ON X86_64. Cluster limits for the partition are MaxTime

equal to 4320 minutes. Our runs use one node and do not use distributed parallelism (due to context switching).

4.7 Deliverables and figure generation

Figures are generated by a single analysis script that reads per pair and summary CSVs and produces plots and configuration tables in both LaTeX and Markdown. The main set includes C1 runtime plots in linear and log scales, best and worst case overlays with theoretical curves, memory profiles in linear and log scales, per algorithm best versus worst case plots, a Whitman versus Freese recursion comparison, Freese memoisation efficiency, a speedup ratio versus Whitman, the C4 node sharing analysis, the C6 arity grid, a budget scaling overlay, a winner map in strict form, and a structure summary table. A tolerance based winner map is also generated.

4.7.1 Companion DAG versus tree runs

For a subset of configurations we repeat the analysis in tree mode in order to quantify the effect of subterm duplication on time and memory directly against canonical DAG runs. We report the companion figures alongside the main DAG results and discuss the deltas.

4.8 Limitations

Budget is a structural knob and it does not align across shapes, therefore cross shape comparisons are descriptive. The environment is a single machine profile, so absolute runtimes are platform specific. We emphasise relative comparisons and structural trends.

The next chapter reports the results that follow from this methodology. We begin with the baseline configuration and then quantitatively report additional factor studies (sharing and arity), memory profiles, representation effects, and winner maps.

Chapter 5

Results and Empirical Findings

This chapter presents the empirical results in a compact, plot-first style. We report medians unless stated otherwise and refer to figures for exact trends.

5.1 Runtime performance comparison

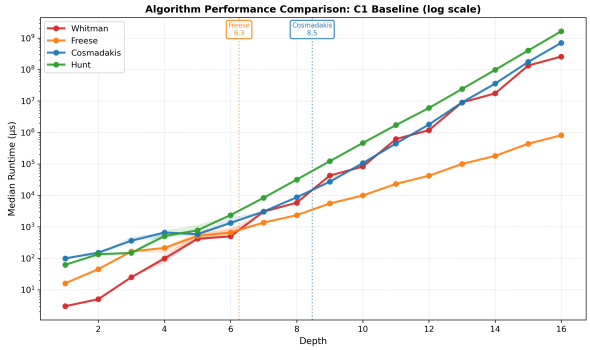

Figure 5: C1 baseline (log scale). Vertical markers show crossovers: Freese near depth 7 and Cosmadakis near depth 8 onwards on full alternation.

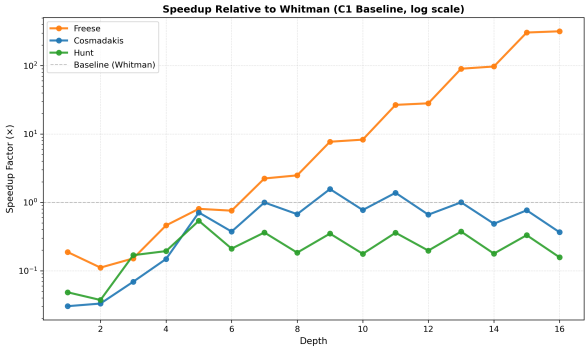

Figure 6: Speedup relative to Whitman (log scale). Freese grows to roughly $300\times$ at $d = 15-16$, Cosmadakis hovers around parity and dips below at the deepest points, Hunt remains below $1\times$.

At shallow depth, Whitman is competitive; Freese overtakes around $d \approx 6$–7 and widens the gap rapidly, reaching two to three orders of magnitude by $d = 14$–16. Cosmadakis crosses near $d \approx 9$ as the relation densifies and memory traffic dominates, and returns close to parity by the deepest points. Hunt is consistently slower on C1.

5.2 Factors affecting performance

Greater DAG sharing is associated with lower runtime for memoising and branching procedures alike (strong for Freese and Whitman; weak for Cosmadakis and Hunt).

Across all four algorithms, best-case medians are essentially flat up to mid-teens depth, while worst-case traces the known asymptotics (exponential for Whitman; high-order polynomial for the others), with the steepest upturns beginning around $d = 12$–14.

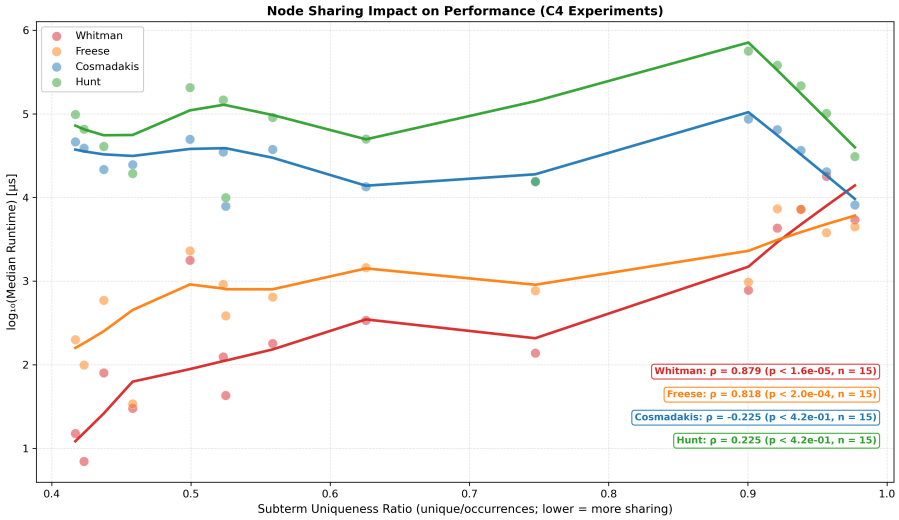

Figure 7: C4 node sharing study. Spearman correlations (as annotated): Whitman $\rho = 0.879$ ($p < 1.6 \cdot 10^{-5}$, $n = 15$), Freese $\rho = 0.818$ ($p < 2.0 \cdot 10^{-4}$, $n = 15$) show strong association with sharing; Cosmadakis and Hunt show weak, non-significant trends. Lower x-axis implies more sharing.

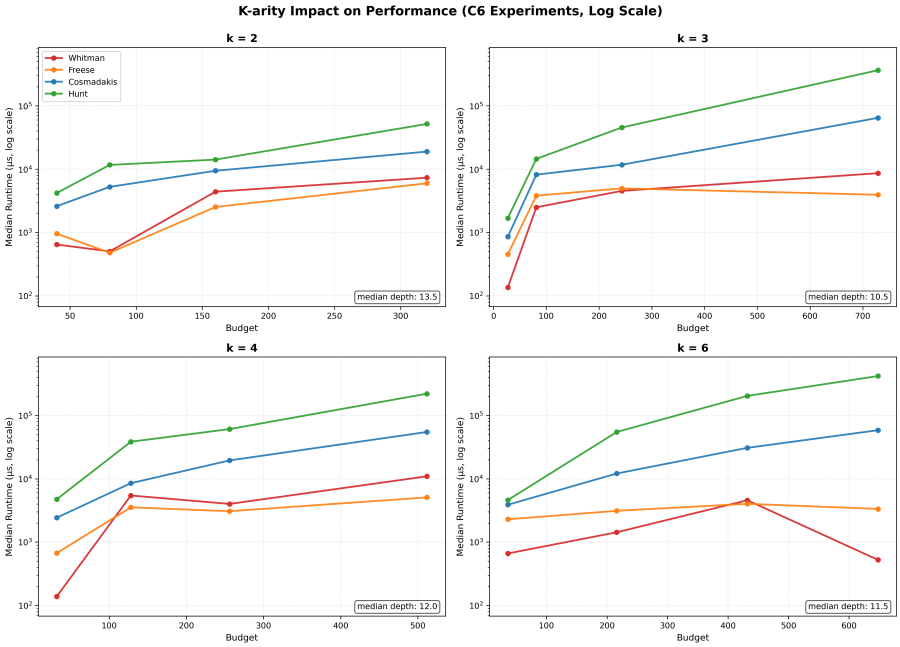

Figure 8: C6 arity grid (log scale). Increasing k materially penalises Cosmadakis and Hunt, with smaller effects on Whitman and Freese.

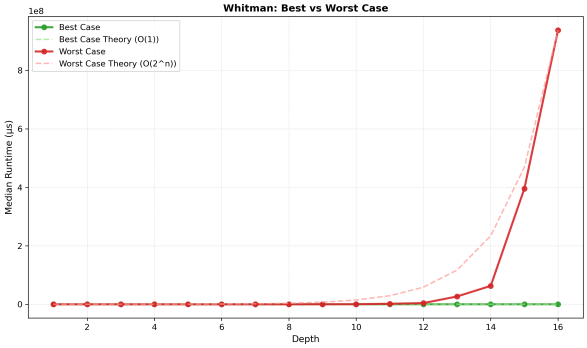

Figure 9: Whitman Best vs Worst Case Runtimes: worst-case exponential growth dominates beyond $d \approx 12$.

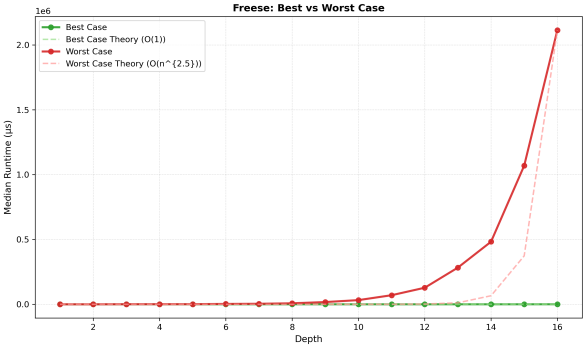

Figure 10: Freese Best vs Worst Case Runtimes: worst-case follows a high-degree polynomial; best-case is flat.

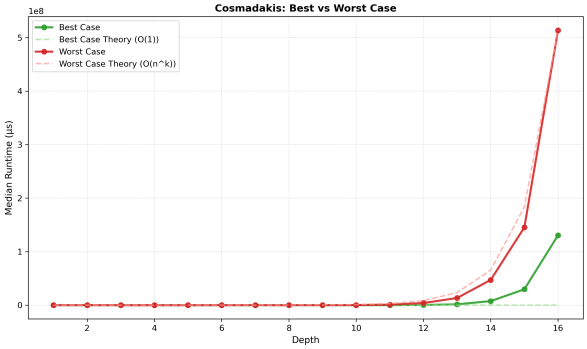

Figure 11: Cosmadakis Best vs Worst Case Runtimes: sharp rise after $d \approx 12$ in worst-case; best-case flat until deep points.

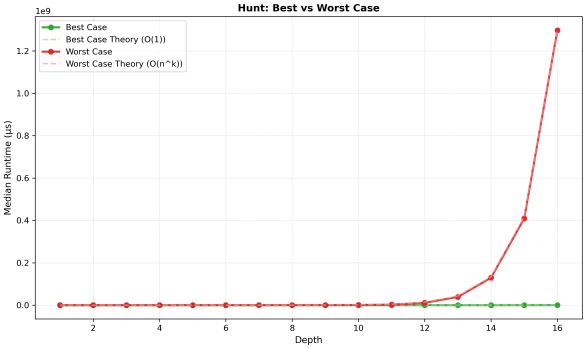

Figure 12: Hunt Best vs Worst Case Runtimes: worst-case steep for deep, wide inputs; best-case near flat.

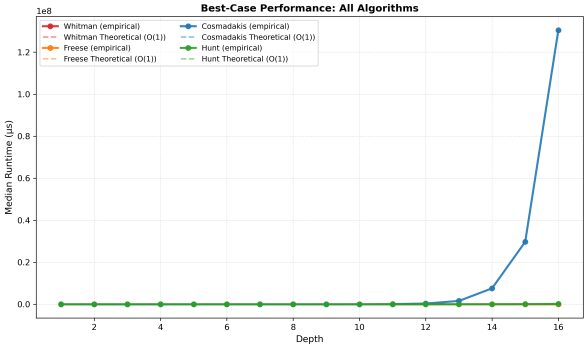

Figure 13: Combined Best-case: Whitman, Freese, Hunt remain near-constant; Cosmadakis exhibits a spike at $d = 16$ (table blow-up).

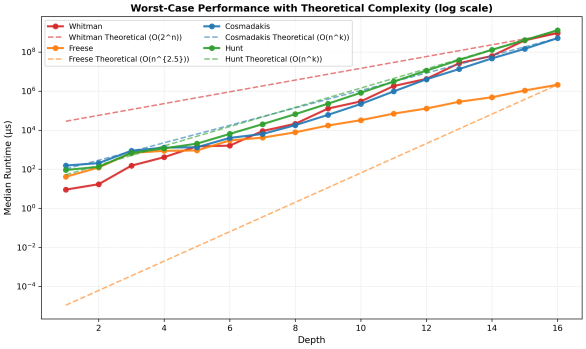

Figure 14: Combined Worst-case (log scale): Whitman becomes intractable first; others rise later.

5.3 Memory profile

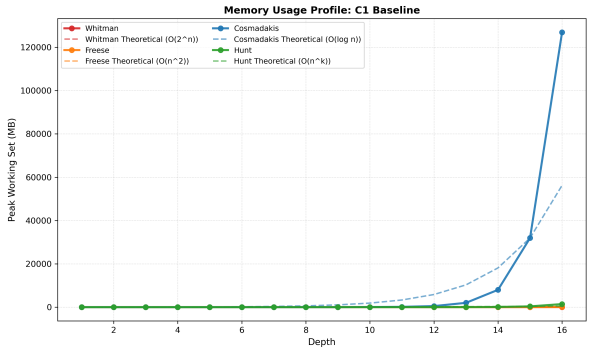

Figure 15: Peak working set (linear).

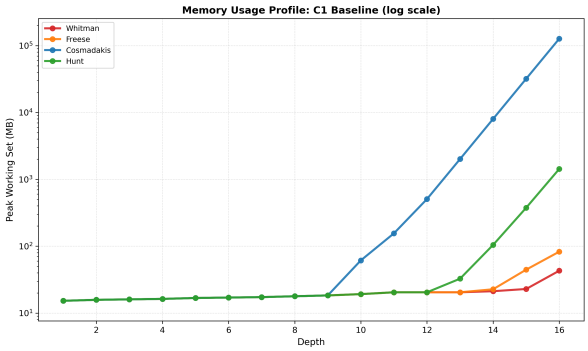

Figure 16: Peak working set (log).

Cosmadakis' bit-matrix dominates memory at depth ($\sim 10^5$ MB at $d = 16$), while Hunt stays around the low-GB range. Freese and Whitman remain in the tens-of-MB range on this platform.

5.4 Recursive work

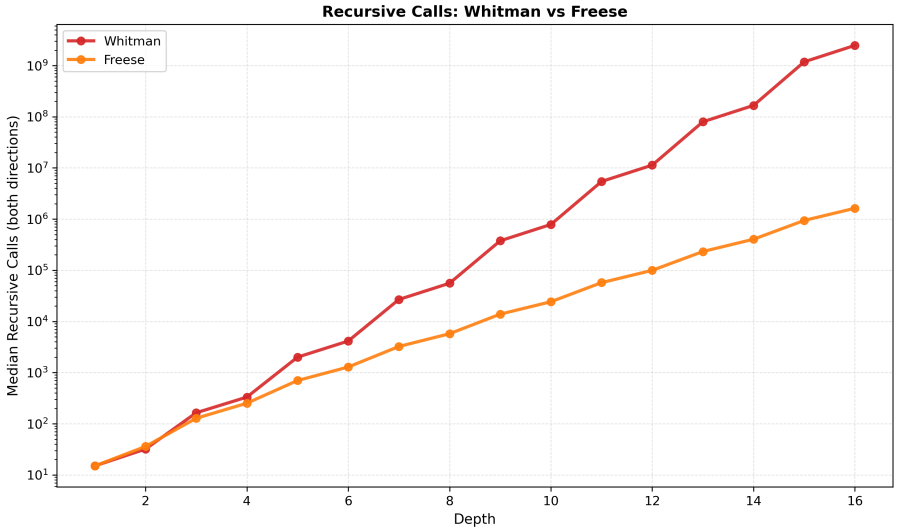

Figure 17: Median recursive calls (both directions, log scale) for C1. Whitman escalates from 10^2 at $d = 4$ to $> 10^9$ by $d = 16$; Freese scales to $\sim 10^6$ calls at $d = 16$.

Memoisation collapses repeated subproblems substantially: at $d = 16$, Whitman exceeds a billion calls while Freese remains around the low millions.

5.5 Representation effect: DAG versus Tree

Canonical DAGs yield the largest gains for Freese (tens to low-hundreds $\times$ by depth 14), moderate gains for Cosmadakis and Hunt, and little change for Whitman where subproblem duplication is explored rather than cached.

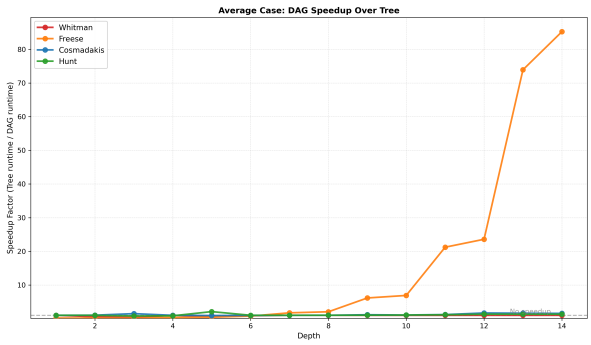

Figure 18: Average case (C1): DAG speedup over tree. Freese reaches $\sim 85\times$ by $d = 14$; Cosmadakis ~ 2–$3\times$; Hunt near $\sim 2\times$.

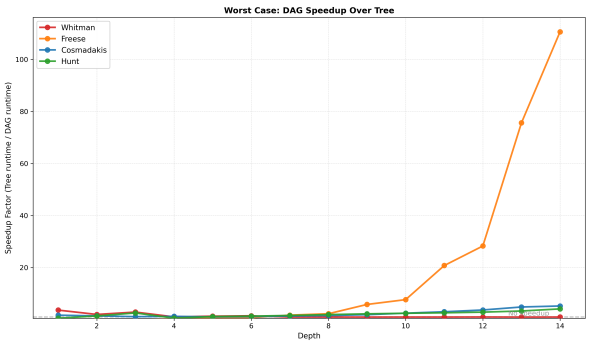

Figure 19: Worst case: DAG speedup peaks near $110\times$ for Freese at $d = 14$; Cosmadakis about 5–$6\times$; Hunt about 4–$5\times$.

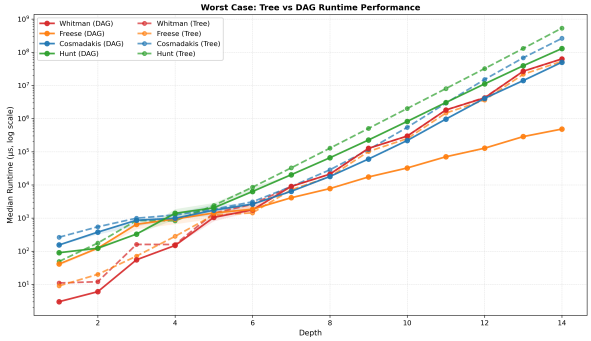

Figure 20: Worst-case runtime, tree vs. DAG (log scale). Large separations open from $d \approx 9$ onward.

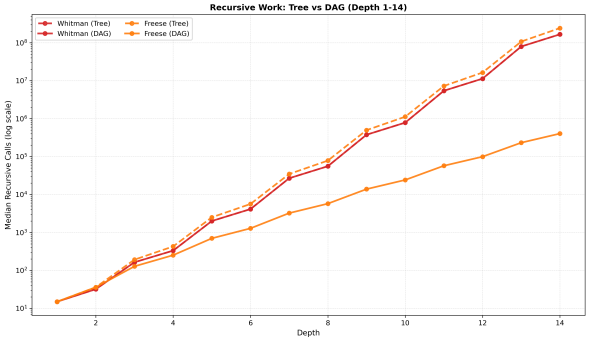

Figure 21: Recursive work, tree vs. DAG. Freese benefits strongly; Whitman shows minimal change.

5.6 Budget scaling and dominance

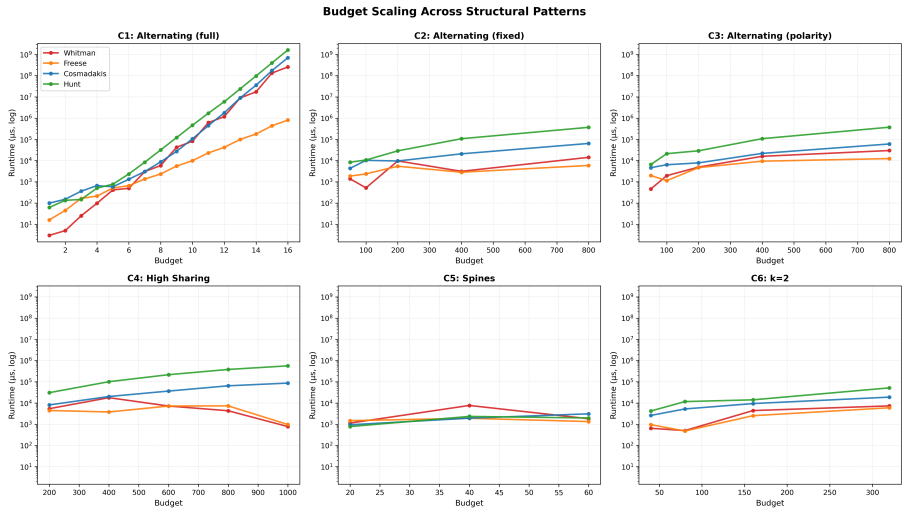

Figure 22: Budget scaling across structural patterns (log scale). Shapes are not complexity-equivalent, so overlays are descriptive.

Budget increases move all procedures upward, most sharply for closure and DP on the widest shapes.

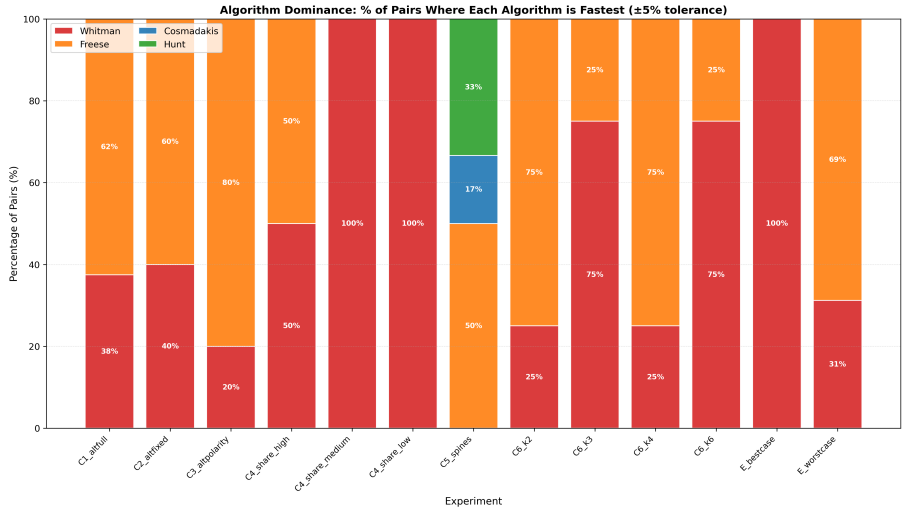

Figure 23: Dominance summary with 5% tie tolerance (algorithms within 5% of minimum runtime share credit). Freese dominates alternating families (e.g., 60–80% of pairs on C2–C3), Whitman dominates low-sharing families (C4 medium/low), spines are mixed (Whitman 50%, Cosmadakis 17%, Hunt 33%), and worst-case is largely Freese ($\sim 69\%$).

5.7 Summary

Freese overtakes Whitman around depth 6–7 on C1 and reaches around $300\times$ speedup by the deepest points. Sharing is a strong predictor of runtime for Whitman and Freese, less so for Cosmadakis and Hunt. Memory usage aligns with state size: Cosmadakis grows to $O(10^5)$ MB, Hunt to $O(10^3)$ MB, while Whitman and Freese remain in the tens of MB. Canonical DAGs provide substantial acceleration for memoised and table-based methods, especially in worst-case structures (up to $\sim 110\times$ for Freese at $d = 14$). The next chapter analyses these patterns in detail and relates them to the structural mechanics of each procedure.

Chapter 6

Analysis and Discussion

This chapter interprets the results from Chapter 5. We evaluate why the curves take the shapes they do, where they align with theory, and where they diverge due to representation, constant factors, or machine effects. We also give simple growth models that predict the observed separations and crossovers, followed by practical guidance.

6.1 Depth scaling on alternating inputs

Figure 5 shows depth scaling on perfectly alternating binary trees with opposite root polarity. The trends follow classical expectations. For Whitman, each meet–join encounter splits the obligation $u \leq v$ into families of subgoals, so the search tree grows exponentially with the smaller of $|u|$ and $|v|$. On full alternation the node count at depth d scales like 2^d, therefore even small increases in d multiply the recursive exploration. This explains why Whitman is competitive at shallow depth but becomes orders of magnitude slower by $d = 15$ to $d = 16$ in Fig. 6.

Freese replaces repeated obligations by memoisation over canonical node IDs. On a DAG, each structural pair $(a, b) \in \mathrm{sub}(u) \times \mathrm{sub}(v)$ is computed once. With full alternation in canonical form, the number of unique subterms grows proportionally to the input size N, so the memo table has $O(N^2)$ entries. Because $N \asymp 2^d$, this yields an $O(4^d)$ envelope in depth. This is still exponential because the input itself grows exponentially in d, but the base is much smaller than Whitman's combinatorial branching. The measured crossover near depth 6 to 7 is where the benefit from avoiding duplicate work outweighs hash lookups and bookkeeping costs.

Cosmadakis and Hunt are also bounded by the number of unique subterms. Both have $O(|\mathrm{sub}(u)| \cdot |\mathrm{sub}(v)|)$ state with an average arity factor in the work. Cosmadakis briefly crosses Whitman near $d \approx 8$ to 9 in Fig. 5 because closure can propagate easy consequences quickly, but as depth increases the relation densifies, the bit matrix grows large, and memory traffic dominates, which brings the curve back toward parity with Whitman. Hunt remains slower than Freese on C1 since it must touch many table cells and perform per-cell conjunctions. At shallow depths Hunt is sometimes close to Whitman due to small constants, but it drifts upward as layers widen.

The polynomial bounds for Cosmadakis and Hunt are in the number of unique subterms N. Our C1 family makes $N \approx 2^d$, so all methods rise exponentially in depth d. Thus at large d, constants and memory effects determine the wall-time ranking.

6.2 Structural drivers: why shape matters

6.2.1 Sharing versus runtime

Figure 7 correlates runtime with a sharing metric, where smaller x-axis values implies more sharing. For Freese the effect is direct: canonicalisation maps isomorphic subterms to the same IDs, so the number of distinct pairs (a, b) shrinks as sharing rises, which strongly reduces total work. This is why the Spearman correlation is large in magnitude and significant. For Whitman the improvement is indirect. Highly shared DAGs generally arise from less alternation or more idempotence; both conditions increase deterministic pushes and reduce the number of hard meet–join face-offs, hence fewer branch points and faster runs. Cosmadakis and Hunt are largely insensitive to repetition frequency because their dominant costs scale with the number of unique subterms rather than with how often those subterms are reused. Sharing helps them only insofar as it reduces $|\mathrm{sub}(t)|$.

A simple back-of-the-envelope model clarifies the Freese trend. Let:

$$r_t = \frac{\text{occurrences}}{\text{unique}}$$

be the DAG compression ratio for term t. The memo domain is a Cartesian product of unique subterms, therefore the expected reduction in pair count when moving from tree to DAG behaves like $r_u \times r_v$. This multiplicative dependence matches the strong monotone trend in Fig. 7.

6.2.2 Arity and fan-out

The arity grid in Fig. 8 shows that increasing the branching factor penalises Cosmadakis and Hunt more than Whitman or Freese. For Cosmadakis, more children increase the degree of upward and downward propagation. Each dequeue touches a larger neighbourhood, transitivity has more opportunities to fire, and the matrix lights up more cells sooner. For Hunt, each table entry requires conjunctions over all children, so the cost per cell grows with k even when the number of layers is unchanged. Whitman and Freese also incur overhead from wider nodes, but the hit is smaller: Whitman's branching is bounded by the per-node fan-out rather than by a global table footprint, and Freese amortises much of the expansion through caching of identical pairs.

6.2.3 Per-algorithm extremes and their causes

The best–worst layouts in Figs. 9, 10, 11, and 12 show when each algorithm meets its theoretical behaviour. Whitman's worst case is alternating with opposite polarity, which forces branching at each level; its best case is same polarity or thin spines where deterministic pushes collapse the tree. Freese is worst when $|\mathrm{sub}(u)|$ and $|\mathrm{sub}(v)|$ are large and diverse, so the table is large and there is little reuse; it is best when many pairs repeat and the table remains small. Cosmadakis and Hunt achieve their best behaviour when the relation or table stays sparse and the wavefront is narrow, and their worst behaviour when closures saturate and layers are wide, which forces quadratic state and substantial per-cell work. The overlays in Figs. 13 and 14 reflect this pattern: best-case lines are essentially flat, whereas worst-case lines exhibit the expected exponential growth for Whitman and steep polynomial growth for the others.

6.3 Memory behaviour

Figures 15 and 16 show space scaling. The differences follow from data structure design and memory access patterns. Cosmadakis builds a bit matrix for $R \subseteq \mathrm{sub}(u) \times \mathrm{sub}(v)$ and drives it with a worklist. As the number of unique subterms grows, the footprint alone approaches the order of $10^5\,\mathrm{MB}$, which is approximately $100\,\mathrm{GB}$, by $d = 16$ on C1. At that point memory bandwidth and cache-miss penalties dominate, which explains the sharper curvature in runtime. Hunt also stores a table of similar cardinality,

but its layer-ordered sweeps produce more sequential access and fewer re-visits than closure, so the live footprint is smaller in practice and locality is better → the range is typically low-GB. Freese stores a compact boolean value per 64-bit key in a hash map, and Whitman holds recursion state and small identity or containment caches, so both remain in tens of MB and are not memory-bound over our range. The small best-case spike for Cosmadakis at high depth in Fig. 13 is not caused by logical complexity but by space blow-up and the resulting cache behaviour.

6.4 Recursive work: concentration and collapse

Figure 17 quantifies recursive calls. Whitman escalates from 10^2 at $d = 4$ to beyond 10^9 by $d = 16$, which is consistent with repeated binary branching. Freese remains around 10^6 calls at $d = 16$ because each structural pair is resolved once and then reused. The residual work that remains after memoisation concentrates exactly where polarity mismatches force decomposition on both sides and where multiple children must be conjoined. Memoisation eliminates duplication, not necessity (on full alternation there are still $\Theta(n^2)$ distinct pairs to settle).

6.5 Representation effect: tree versus DAG

The DAG advantage in Figs. 18 to 21 decomposes along algorithmic lines. For Freese, tree mode assigns distinct pointer identities to isomorphic subterms, so identical obligations (a, b) are recomputed under different addresses and cache hits fail. Canonical DAG mode performs hash-consing and ACId normalisation, therefore each structural pair is computed once and re-used. This is why average-case speedup reaches about $85\times$ by $d = 14$ and why the worst-case panel peaks near $110\times$. For Cosmadakis and Hunt, canonical DAGs reduce the number of unique nodes, which shrinks matrix or table dimensions and brings speedups of two to six times; the gains are moderate because both already compute each cell once in either representation, and DAGs merely reduce the number of cells. Whitman sees minimal change in Fig. 21 because without pair caching, collapsing identities does not prevent the recursive exploration; it only changes pointer equality.

Again, $r_t = \frac{\text{occurrences}}{\text{unique}}$ is the compression ratio of term t under canonicalisation. For Freese, the expected speedup from tree to DAG is proportional to $r_u \times r_v$, since the memo domain changes from pairs of occurrences to pairs of uniques.

6.6 Deviations from theory

The main deviations have principled explanations. The brief interval where Cosmadakis beats Whitman arises because closure can propagate along joins and meets quickly, which short-circuits regions of the matrix before Whitman's backtracking locates a witness; at greater depths the $O(N^2)$ state and irregular access patterns dominate and erase this advantage. Hunt often sits above Cosmadakis on C1 even though both are quadratic in principle, because Hunt pays a per-cell conjunction cost that scales with arity and does not benefit from opportunistic transitivity shortcuts that sometimes help Cosmadakis at intermediate depths. Finally, the precise crossover depth between Whitman and Freese is platform sensitive; hash throughput and cache capacity shift the crossover by about one level either way, but the existence of an early crossover is robust.

6.7 Budget scaling and dominance

The budget overlay in Fig. 22 should be read descriptively because the budget knob is structural and does not align across shapes. Even so, as structures widen, closure and DP climb faster due to table or matrix growth and increasing per-cell work, while memoised recursion remains bounded by the number of

unique pairs. The dominance summary in Fig. 23 reflects the same regimes. Freese dominates alternating and mixed families at medium to high depth. Whitman can win on spines or same-polarity slabs at small sizes where deterministic pushes remove most branching and overheads are tiny. Cosmadakis and Hunt are competitive when memory is ample and structures are not too wide. Under tighter memory contexts, Hunt tends to be the safer polynomial choice.

6.8 Further suggestions

For alternating inputs with opposite polarity in canonical DAG form, Freese should be preferred above depth about six to seven, whereas below that threshold Whitman's lower overhead may be competitive. When high DAG sharing is expected, for example due to a small variable alphabet or frequent idempotence, Freese gains multiplicatively with the sharing ratios r_u and r_v and typically provides large advantages. If arity is large or layers are wide, Cosmadakis should be avoided unless memory is ample; among table methods, Hunt gives smaller and more predictable footprints. For thin chains or same-polarity regions, Whitman is often the simplest and fastest by virtue of deterministic pushes. In the absence of canonicalisation, large slowdowns should be expected for Freese and moderate slowdowns for table methods, with little change for Whitman.

6.9 Validity, limits, and testable predictions

Budgets are structural and not directly comparable across shapes, therefore cross-shape overlays are descriptive. Absolute times are platform specific, although structure-driven patterns are robust. Inputs are synthetic free-lattice terms and may differ from downstream workloads in sharing and arity distributions. Parser determinism, seeded randomness, and medians reduce variance, but memory hierarchy and allocator differences can shift constants, particularly for Cosmadakis and for the memo table in Freese.

Three predictions follow directly from the mechanisms: Increasing the variable alphabet, which lowers sharing, should move Freese upward and push the Whitman–Freese crossover deeper. Enabling *ensure unique leaves* on C1 should also reduce DAG benefits for Freese by reducing repeated leaves, narrowing the DAG versus tree speedups. Raising arity at fixed depth should penalise Cosmadakis and Hunt superlinearly in practice due to memory traffic, with smaller effects on Whitman and Freese.

Additional limitations.

- **Shared front end as a single point of failure:** All four engines consume the same parser/canonicaliser. Agreement among procedures is a strong sanity check, but a front-end bug (e.g., in ACId normalisation) could make all four agree on a wrong instance. A future cross-check against an independent tool (e.g., Prover9/OTTER or a minimal Coq encoding) would reduce this risk.
- **Depth–size coupling:** In C1, depth is the size knob and $N \approx 2^d$, so the "polynomial" methods appear exponential in d. This is expected but makes asymptotics easy to misread across shapes. Results should be interpreted in terms of unique-subterm count N, not depth alone.
- **Sample size at deep points:** For $d \geq 12$ we thinned to five pairs. Medians and IQRs damp noise, but tail behaviour (esp. Cosmadakis near memory limits) may be under- or over-represented.
- **Schedule sensitivity not explored:** We fixed one branch order for Whitman/Freese and a FIFO worklist for Cosmadakis. Alternative schedules (e.g., best-first branching; different queue policies; height/blocking orders for closure/DP) can move constants and crossovers.
- **Engineering choices conflate with "algorithm":** Hash design and table layout (Freese), bit-matrix representation (Cosmadakis), and bit-table packing (Hunt) all influence cache and band-

width. We did not vary these (e.g., sparse/blocked matrices, compressed bitsets), so some "memory blow-ups" are partly representation choices rather than inherent limits.

- **Tree model bias:** Our tree mode uses left-associative binaries. Other tree encodings (e.g., balanced binaries) could alter unique-subterm counts and locality, especially for table methods.
- **Preprocessing cost omitted:** We report solver runtimes post-parse. Canonicalisation (flatten/sort/dedup/intern) can be non-trivial for very large terms; we did not separate or optimise this cost.
- **Uniform word-problem scope:** We studied $u \leq v$ over free lattices. Cosmadakis' system is designed for the uniform word problem with a hypothesis set E; we did not vary $|E|$ or compare under large premise sets, which can change practical behaviour.

What likely limited peak depth / breadth.

- **Whitman time cap** (exponential branching) and **Cosmadakis memory cap** (quadratic state) bounded the deepest C1 points.
- Lack of **sparse/blocked matrices** and **parallel fills** constrained Cosmadakis/Hunt scalability.
- Using a single **memo table implementation** (hash, probing policy) constrained Freese's high-N throughput.

6.10 Future directions

Hybrid dispatchers that recognise local structure could combine Whitman's deterministic pushes on easy regions with Freese's memo table on the hard regions, or switch to Hunt when layers densify. Sparse or blocked representations for Cosmadakis and layer-batched dynamic programming for Hunt may reduce memory traffic. Porting to real datasets would calibrate sharing and arity distributions and validate the guidance in downstream workloads.

Chapter 7

Conclusion

This study implemented and benchmarked four decision procedures for the equational theory of lattices in a controlled, reproducible framework, comparing behaviour across depth, arity, sharing, and term representation (canonical DAGs versus trees). Empirically, Freese's memoised recursion overtakes Whitman around depth 6–7 on alternating inputs and grows to roughly $300\times$ faster by the deepest points tested. Representation further amplifies this advantage: moving from trees to canonical DAGs removes subterm duplication and yields speedups of up to $\sim 110\times$ for Freese (about $85\times$ on the balanced average), with moderate gains for Cosmadakis and Hunt and negligible effect on Whitman.

The polynomial procedures differ in their resource profiles: Cosmadakis is often competitive at moderate sizes but its bit-matrix footprint grows to approximately 10^5 MB (~ 100 GB) at depth, whereas Hunt delivers a steadier, low-GB memory profile at the cost of somewhat higher runtime. Two structural factors chiefly explain the rankings observed: (i) *sharing* strongly predicts performance for Freese by collapsing the memo domain, and (ii) increasing *arity* penalises closure/table methods (Cosmadakis, Hunt) more than branching/memoised recursion (Whitman, Freese). Consistent with this, the dominance map shows Freese prevailing on alternating and mixed families at medium–high depth, while Whitman remains competitive on thin spines and small same-polarity slabs where deterministic pushes minimise overhead.

Beyond the curves, the analysis links mechanisms to outcomes: memo-table saturation versus exponential branching, closure wavefront density and memory traffic, table fill order and cache locality, and representation effects via canonicalisation. These observations support practical guidance: prefer Freese on canonical DAGs above shallow depth or when sharing is high; consider Hunt as the safer polynomial choice under tighter memory; reserve Whitman for small or structurally easy instances.

Natural extensions include hybrid dispatch (combining Whitman's deterministic pushes with Freese's caching on hard regions), sparse or blocked layouts for closure/table methods, and evaluation on domain-specific benchmarks to calibrate sharing and arity distributions. Taken together, the results bridge theory and practice for lattice decision procedures and provide a foundation for informed algorithm selection in automated reasoning systems.

Bibliography

[1] B. A. Davey and H. A. Priestley, *Introduction to Lattices and Order*, 2nd edition, Cambridge: Cambridge University Press, 2002. Available: https://www.cambridge.org/core/books/introduction-to-lattices-and-order/946458CB6638AF86D85BA00F5787F4F4, Accessed: March 12, 2025.

[2] P. M. Whitman, *Free Lattices*, Annals of Mathematics, 2nd series, vol. 42, no. 1, Mathematics Department: Princeton University, 1941, pp. 325-330. ISSN: 0003-486X. Available: https://www.jstor.org/stable/1969001, Accessed: March 15, 2025.

[3] R. Freese, *Free Lattice Algorithms*, Order, vol. 3, no. 4, pp. 331–344, December 1987. Publisher: Springer. ISSN: 0167-8094, 1572-9273. DOI: 10.1007/BF00340775. Available: https://doi.org/10.1007/BF00340775, Accessed: March 15, 2025.

[4] S. S. Cosmadakis, *The Word and Generator Problems for Lattices*, Information and Computation, vol. 77, no. 3, 1988, pp. 192-217. Publisher: Academic Press. ISSN: 0890-5401. DOI: 10.1016/0890-5401(88)90048-X. Available: https://doi.org/10.1016/0890-5401(88)90048-X, Accessed: March 15, 2025.

[5] H. B. Hunt, D. J. Rosenkrantz, and P. A. Bloniarz, "On the Computational Complexity of Algebra on Lattices," *SIAM Journal on Computing*, vol. 6, no. 1, 1987, pp. 129–146. Publisher: Society for Industrial and Applied Mathematics (SIAM). ISSN: 0097-5397. DOI: 10.1137/0216011. Accessed: March 15, 2025.

[6] N. Galatos, *Decidability of Lattice Equations*, Studia Logica, vol. 112, 2024, pp. 607-610. Publisher: Springer Nature. ISSN: 1572-8730. DOI: 10.1007/s11225-023-10063-4. Available: https://doi.org/10.1007/s11225-023-10063-4, Accessed: March 15, 2025.

[7] L. de Moura, N. Bjørner, *Z3: An Efficient SMT Solver*, Ramakrishnan, C.R., Rehof, J. (eds) Tools and Algorithms for the Construction and Analysis of Systems, vol. 4963, 2008. Publisher: Springer, Berlin, Heidelberg. DOI: 10.1007/978-3-540-78800-3_24. Available: https://doi.org/10.1007/978-3-540-78800-3_24, Accessed: April 27, 2025.

Appendix A

Decision Procedures Pseudocode

The following algorithms mirror the implementations described in Chapter 3, with explicit evaluation order, stopping conditions, and representation-sensitive details (unique-subterm enumeration vs. raw trees). We write KIND(x) $\in \{\text{VAR}, \text{JOIN}, \text{MEET}\}$, CHILDREN(x) for the stable child list, and CONTAINSVAR(y,x) to test whether variable x appears among the children of a join y (dually for meets).

Algorithm 1 Whitman: depth-first backtracking (no memoisation)

```
1:  function WhitmanLeq(u, v)
2:      if u = v then
3:          return true
4:      end if
5:      if Kind(u)=Var and Kind(v)=Join then
6:          return ContainsVar(v, u)
7:      end if
8:      if Kind(u)=Meet and Kind(v)=Var then
9:          return ContainsVar(u, v)
10:     end if
11:     if Kind(u)=Join then
12:         for all a ∈ Children(u) do
13:             if not WhitmanLeq(a, v) then
14:                 return false
15:             end if
16:         end for
17:         return true
18:     end if
19:     if Kind(v)=Meet then
20:         for all b ∈ Children(v) do                    ▷ right branch first
21:             if WhitmanLeq(u, b) then
22:                 return true
23:             end if
24:         end for
25:         return false
26:     end if
27:     if Kind(u)=Meet and Kind(v)=Join then
28:         for all b ∈ Children(v) do                    ▷ right branch
29:             if WhitmanLeq(u, b) then
30:                 return true
31:             end if
32:         end for
33:         for all a ∈ Children(u) do                    ▷ left branch
34:             if WhitmanLeq(a, v) then
35:                 return true
36:             end if
37:         end for
38:         return false
39:     end if
40:     return false
41: end function
```

Algorithm 2 Freese: top-down recursion with memoisation

```
1:  function FreeseLeq(u, v)
2:      if (u, v) in Memo then
3:          return Memo[(u, v)]
4:      end if
5:      if u = v then
6:          Memo[(u, v)]←true; return true
7:      end if
8:      if Kind(u)=Var and Kind(v)=Join then
9:          r ← ContainsVar(v, u);  Memo[(u, v)]← r; return r
10:     end if
11:     if Kind(u)=Meet and Kind(v)=Var then
12:         r ← ContainsVar(u, v);  Memo[(u, v)]← r; return r
13:     end if
14:     if Kind(u)=Join then
15:         for all a ∈ Children(u) do
16:             if not FreeseLeq(a, v) then
17:                 Memo[(u, v)]←false; return false
18:             end if
19:         end for
20:         Memo[(u, v)]←true; return true
21:     end if
22:     if Kind(v)=Meet then
23:         for all b ∈ Children(v) do                    ▷ right branch first
24:             if FreeseLeq(u, b) then
25:                 Memo[(u, v)]←true; return true
26:             end if
27:         end for
28:         Memo[(u, v)]←false; return false
29:     end if
30:     if Kind(u)=Meet and Kind(v)=Join then
31:         for all b ∈ Children(v) do                    ▷ right branch
32:             if FreeseLeq(u, b) then
33:                 Memo[(u, v)]←true; return true
34:             end if
35:         end for
36:         for all a ∈ Children(u) do                    ▷ left branch
37:             if FreeseLeq(a, v) then
38:                 Memo[(u, v)]←true; return true
39:             end if
40:         end for
41:         Memo[(u, v)]←false; return false
42:     end if
43:     Memo[(u, v)]←false; return false
44: end function
```

Algorithm 3 Cosmadakis: worklist-driven closure on unique subterms

function CosmaLeq(u, v)
  $U \leftarrow$ UniqueSubterms(u); $V \leftarrow$ UniqueSubterms(v)
  $S \leftarrow U \cup V$ with stable indices; $R \leftarrow$ BitMatrix(—S—,—S—, all **false**)
  for all $x \in S$: $R[x, x] \leftarrow$**true**; Enqueue(x, x)
  while worklist not empty **do**
    $(a, b) \leftarrow$ Dequeue() ▷ Upward (join) propagation
    for all join nodes c with $b \in$ Children(c) **do**
      if not $R[a, c]$ **then**
        $R[a, c] \leftarrow$**true**; Enqueue(a, c)
      end if
    end for ▷ Downward (meet) propagation
    for all meet nodes d **do**
      if $\forall c \in$ Children(d) : $R[a, c] =$ **true** and **not** $R[a, d]$ **then**
        $R[a, d] \leftarrow$**true**; Enqueue(a, d)
      end if
    end for ▷ Transitivity
    for all $c \in S$ **do**
      if $R[a, c]$ and **not** $R[c, b]$ **then**
        $R[c, b] \leftarrow$**true**; Enqueue(c, b)
      end if
      if $R[c, b]$ and **not** $R[a, c]$ **then**
        $R[a, c] \leftarrow$**true**; Enqueue(a, c)
      end if
    end for
  end while
  return $R[u, v]$
end function

Algorithm 4 Hunt–Rosenkrantz–Bloniarz: height-stratified dynamic program

function HuntLeq(u, v)
  $U \leftarrow$ UniqueSubterms(u); $V \leftarrow$ UniqueSubterms(v)
  Compute heights $h(\cdot)$ for all $x \in U \cup V$; build layers $L_U[h], L_V[h]$
  Allocate table $T[U][V] \leftarrow$**false** ▷ Base facts
  for all $a \in U$ **do**
    $T[a, a] \leftarrow$**true**
  end for
  for all $b \in V$ with Kind(b)=Join **do**
    for all variables x among Children(b) **do**
      $T[x, b] \leftarrow$**true**
    end for
  end for ▷ Height-nondecreasing fill
  for $h_u = 0..H_U$ **do**
    for $h_v = 0..H_V$ **do**
      for all $a \in L_U[h_u]$ **do**
        for all $b \in L_V[h_v]$ **do**
          if Kind(a)=Join **then**
            $T[a, b] \leftarrow \bigwedge_{c \in \text{Children}(a)} T[c, b]$
          else if Kind(b)=Meet **then**
            $T[a, b] \leftarrow \bigwedge_{d \in \text{Children}(b)} T[a, d]$
          end if
        end for
      end for
    end for
  end for
  return $T[u, v]$
end function

Appendix B

Code Availability

The full source code (engines, generators, configs, and plotting scripts) is publicly available at:

https://github.com/DeathSnip3r/Automated-Proofs-of-Lattice-Equations

Optional metadata for permanence: commit/tag used to produce figures <commit-or-tag>, archived release DOI <doi-if-any>.

Wits University Faculty of Science post-graduate student AI declaration

I understand that the use of generative AI tools (such as ChatGPT or similar) without explicitly declaring such use constitutes a form of plagiarism and is classified by Wits University as academic misconduct.

I declare that in the course of conducting the research towards my degree or in the preparation of this thesis/dissertation/research report (select one by marking with an X):

I **did not** make use of generative AI tools ☐

I **did** make use of generative AI tools for the following (tick all that apply):

1. Idea Generation (research problem/design, hypothesis) ☐
2. Sourcing Related Work (summarising, identifying sources) ☐
3. Methods and Experiment Design (experiment setup, model tuning) ☐
4. Data Analysis (presentation, coding, interpretation) ☒
5. Theoretical Development (theorem proving, conceptual analysis) ☐
6. Code Development (generating algorithms, writing scripts) ☐
7. Presentation (rendering graphics, formatting) ☒
8. Editing (grammar, readability) ☐
9. Writing (text generation, document structuring) ☐
10. Citation Formatting (structuring, organising) ☒

If other uses were involved, please specify below:

Generative AI tool used (list all)	**Used for?**

If generative AI tools were used as an integral part of the experimental design or in the direct execution of my research, I confirm that details of this use are clearly outlined in the relevant experimental/methodology chapters of my thesis/dissertation/research report.

Student number: 2465557

Candidate signature: 

Date: November 7, 2025