Upward Book Embeddings of DAGs: Constraint-Based Methods and Embeddability Analysis

by

Rustem Kakimov

A thesis submitted to the Faculty of Graduate Studies
Lakehead University
in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science

Department of Computer Science
Lakehead University
April 2025

Examining Committee

Dr. Xing Tan, Supervisor
Department of Computer Science, Lakehead University

Dr. Ruizhong Wei
Department of Computer Science, Lakehead University

Dr. Kai Huang
DeGroote School of Business, McMaster University

Acknowledgments

I would like to express my deepest gratitude to my supervisor, Dr. Xing Tan, for his continuous support and encouragement throughout the course of this research. His insightful feedback, invaluable ideas, and unwavering patience have been instrumental to the development of this work. Without his guidance, this thesis would not have been possible.

I would also like to sincerely thank Dr. Ruizhong Wei and Dr. Kai Huang for serving on my examining committee.

I am thankful to Dr. Rachael Wang and Jay Patel for their work as Graduate Coordinators. Their assistance and support throughout my studies have been greatly appreciated.

In addition, I wish to thank the Department of Computer Science and the Faculty of Graduate Studies at Lakehead University for the appointment of my Graduate Assistantship. Beyond financial support, the Graduate Assistantship provided valuable professional training and opportunities that have contributed significantly to my academic and personal growth.

Abstract

The k-page upward book embedding (kUBE) problem is a fundamental challenge in graph theory with applications in circuit layout, scheduling, and hierarchical visualization. Despite its relevance, the problem—particularly for $k \geq 2$—remains underexplored. This thesis develops practical methods for solving kUBE and conducts a detailed investigation of how graph structural properties influence upward embeddability.

We first propose a Boolean satisfiability (SAT) encoding, SAT-1, that extends existing k-page book embedding techniques to the general kUBE setting. For the special case of $k = 2$ (2UBE), we introduce SAT-2, a more compact SAT encoding exploiting the fixed number of pages, and a constraint programming (CP) model as an alternative formulation. Empirical evaluation shows that SAT solvers consistently outperform CP, with SAT-2 achieving up to 40% faster runtimes on large instances and up to 30× speedups on hard instances from the North dataset compared to SAT-1.

Beyond solving efficiency, we systematically analyze how upward book embeddability depends on structural parameters such as the edge-to-vertex ratio (m/n). Through exhaustive enumeration and sampling, we identify sharp phase transition phenomena across different values of k (up to $k = 6$) and model the phase transition threshold as a function of graph size and page count using a power-law relationship, providing the first quantitative characterization of this phenomenon.

Contents

List of Tables

List of Figures

Publications

Parts of this thesis are based on the following publications and submissions:

- Rustem Kakimov and Xing Tan. "SAT for Upward Book Embedding: An Empirical Study." Accepted at the *38th Canadian Conference on Artificial Intelligence (Canadian AI 2025).* [29]
- Rustem Kakimov and Xing Tan. "On Upward Book Embeddability of DAGs." Submitted to the *The 37th Canadian Conference on Computational Geometry (CCCG 2025).*

Chapter 1

Introduction

The *book embedding problem (kBE)* is a fundamental topic in graph theory that deals with arranging the vertices and edges of a graph into a k-page book structure. For an undirected graph $G = (V, E)$, a k-page book embedding consists of two components:

1. a **vertex ordering**, which determines the position of vertices along a linear spine, and

2. a **page assignment**, which places each edge on one of the k pages so that no two edges on the same page cross.

Compact book embeddings that minimize k are useful in a wide range of applications, such as circuit layout design [13, 26], parallel scheduling [5], graph visualization [40], and data structure optimization [17, 38].

An important extension of kBE is the *k-page upward book embedding (kUBE)* problem, which is applicable to directed graphs (DAGs). In this variant, all edges must be oriented upward along the spine—a restriction that naturally aligns with hierarchical layouts, network diagrams, and 3D graph drawing [18]. A visual example of a 2-page upward book embedding (2UBE) for a simple grid DAG is shown in Figure 1.1.

While the 1-page cases of kBE and 1UBE can be solved in linear time [27], the general k-page versions become NP-complete for $k \geq 2$ [43]. In the case of 2UBE, this complexity classification was only recently established in 2023 by Bekos et al. [6]. Given this computational intractability, understanding the structural properties that

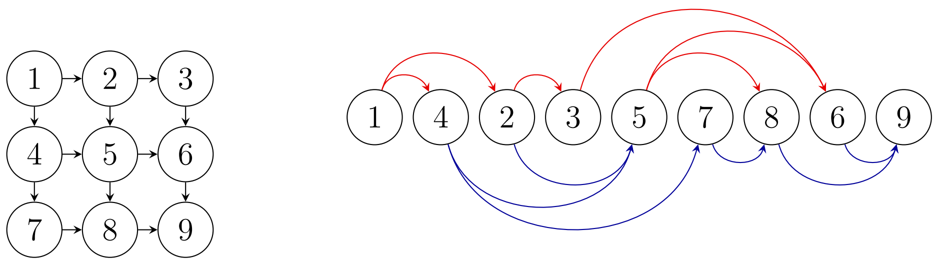


Figure 1.1: A 3×3 directed grid and its corresponding 2-page "upward" book embedding.

influence embeddability—and finding practical solution methods—has become increasingly important.

Like many other combinatorial problems in graph theory, k-Page Book Embedding (kBE) and Upward Book Embedding (kUBE) are naturally suited to declarative approaches such as *Constraint Programming (CP)* and *Boolean Satisfiability (SAT)*. CP allows the problem to be expressed through high-level constraints that guide the search process via propagation and domain-specific heuristics. In contrast, SAT reduces the problem to a purely Boolean formulation, enabling the use of modern solvers that efficiently explore the entire solution space and are capable of proving unsatisfiability. A key advantage of both approaches is their completeness—failure to find a solution implies its non-existence.

This work aims to address a gap in the literature by presenting three general-case solutions for the 2-Page Upward Book Embedding (2UBE) problem—two SAT-based (SAT-1 and SAT-2) and one based on Constraint Programming (CP)—where SAT-1 is applicable for arbitrary k-page embeddings. All approaches are evaluated empirically using the North dataset, a standard benchmark for graph embedding problems [24].

A further objective is to demonstrate how the SAT-1 encoding can be leveraged to investigate the relationship between structural properties of the graph and its embeddability for different values of k. Our results indicate that the edge-to-vertex ratio (m/n) is a strong predictor of embeddability: certain ranges guarantee 100% embeddability for all DAGs, while others result in none. We also observe a clear phase transition, whose location depends on the number of vertices (n) and the number of available pages (k).

The main contributions of this thesis are summarized as follows:

- **Formulations and Modeling**
  - Adaptation of an existing kBE SAT encoding to kUBE (SAT-1),
  - Introduction of a novel SAT encoding optimized for 2UBE (SAT-2),
  - Formulation of a CP model for 2UBE,
  - Formal proofs of soundness and completeness of all the proposed methods, and empirical evaluation on standard benchmark datasets,
- **Dataset Construction and Empirical Analysis**
  - Enumeration of all DAGs up to $n = 6$ and analysis of their kUBE embeddability,
  - Sampling-based study of larger DAGs ($n = 10, 15, 20$) using topological and uniform generation methods,
  - Empirical identification of phase transition points (50% satisfiability, peak runtime) at specific m/n values for $n \leq 20$,
  - Modeling the phase transition threshold in kUBE embeddability as a function of the number of vertices n and pages k using a power-law relationship,
- **Theoretical Contributions**
  - Proof that all DAGs of size $n \geq 6$ are embeddable in $n - 3$ pages,
  - Conjecture that $k = \lceil n/2 \rceil$ is sufficient for an upward book embedding (kUBE) of any DAG with n vertices.

The remainder of this thesis is organized as follows. Chapter 2 introduces the fundamental definitions and notation required throughout the work and reviews key complexity results related to the k-Page Book Embedding (kBE) and k-Page Upward Book Embedding (kUBE) problems. Chapter 3 describes three alternative formulations of the 2UBE problem: the SAT-1 encoding, the SAT-2 encoding, and a Constraint Programming (CP) model. This chapter also discusses the principal design choices and highlights the key differences between the proposed approaches. Chapter 4 details the benchmark datasets, solver configurations, and experimental results. Chapter 5 outlines the methodologies employed for generating small and large Directed Acyclic Graphs (DAGs) via exhaustive enumeration and sampling techniques. Chapter 6 presents the analysis of embeddability trends observed in the datasets, including empirical identification of phase transitions and the derivation of an approximate formula relating the m/n phase transition threshold to the number of vertices n and pages k. Finally, Chapter 7 summarizes the key findings of this research and proposes several avenues for future investigation.

Chapter 2

Background

This chapter establishes foundational concepts required for this study. First, we introduce the classical book embedding problem, followed by the *upward* variant which is the focus of this research. We provide formal definitions, complexity results, known algorithms, and other relevant findings for these problems and their domains. We then describe two solving methodologies used in this work: SAT and CP. We briefly outline how problems can be encoded in either framework, why these approaches were selected, and how they have been successfully applied in prior literature on related graph problems. All of the concepts introduced here set the stage for the discussion in later chapters.

2.1 The k-Page Book Embedding Problem (kBE)

The *k-Page Book Embedding* problem (kBE) is a classical graph layout problem that arises in applications such as VLSI design and graph visualization. The core idea is to place the vertices of a graph along a linear spine and partition the edges across k pages in such a way that no two edges on the same page intersect. This abstraction allows for reducing edge clutter and preserving readability in layered graph drawings.

In the following, we provide formal definitions of the k-Page Book Embedding problem, review known complexity results, describe algorithmic approaches, and discuss recent findings related to *book thickness*.

Formal Definitions

Definition 2.1 (Book Embedding for Undirected Graphs). *A book embedding of a graph $G = (V, E)$ consists of*

1. *a linear ordering $\pi : V \to \{1, 2, \ldots, |V|\}$ of vertices along the line called the spine of a book, and*

2. *an assignment $\sigma : E \to \{1, \ldots, k\}$ of individual edges on one of the k pages such that no edges assigned to the same page geometrically cross. More precisely, for any two edges (u_1, v_1) and (u_2, v_2) on the same page, such that $\pi(u_1) < \pi(v_1)$ and $\pi(u_2) < \pi(v_2)$, the following two conditions are not allowed: $\pi(u_1) < \pi(u_2) < \pi(v_1) < \pi(v_2)$ and $\pi(u_2) < \pi(u_1) < \pi(v_2) < \pi(v_1)$.*

Definition 2.2 (kBE Problem). *A k-Page Book Embedding Problem (kBE) is defined as follows: Given an undirected graph G, does there exist a k-page book embedding for G?*

Complexity and Algorithms

Book embedding has been extensively studied in both theoretical and practical contexts. The 1-page variant, referred to as 1BE, is computationally simple and can be solved in linear time. The class of graphs that admit a 1-page embedding is exactly the outerplanar graphs [13]. However, for $k \geq 2$, the problem becomes NP-complete [13, 43], introducing considerable computational difficulty.

In recent years, more efficient approaches have been explored. One of the most promising developments is a sub-exponential algorithm for the 2-page case [23]. This method runs in $2^{O(\sqrt{n})}$ time and provides a more tractable solution for medium-sized instances.

Theoretical Properties

The *book thickness* (also known as the *page number*) of a graph is defined as the smallest k for which a valid k-page embedding exists. For certain graph families, the exact book thickness is known. For example, the complete graph K_n has book thickness given by:

Theorem 2.1 (Overbay [36]). *If $n \geq 4$, then $bt(K_n) = \lceil \frac{n}{2} \rceil$.*

Series-parallel graphs are always embeddable in two pages [13], and the NP-completeness result holds for general graphs when $k \geq 2$ [13]. These findings provide important benchmarks for understanding the difficulty of the problem in both practical and theoretical contexts.

2.2 The k-Page Upward Book Embedding Problem (kUBE)

The *k-Page Upward Book Embedding* problem (kUBE) is a directed version of the classical book embedding problem and forms the main focus of this study. In this variant, the input is a directed acyclic graph (DAG), and the vertex ordering along the spine must be a topological ordering—meaning that every edge (u, v) must satisfy $\pi(u) < \pi(v)$. As with kBE, edges are assigned to one of k pages, and no two edges on the same page may geometrically cross.

This directionality constraint adds an extra layer of complexity to the problem. While kBE permits undirected graphs and focuses purely on edge-crossing minimization, kUBE also enforces a semantic flow from sources to sinks in the graph. This makes upward embeddings especially suitable for hierarchical structures like workflows and dependency trees, but also substantially reduces the number of possible valid embeddings and increases computational difficulty.

In the remainder of this section, we formally define the k-Page Upward Book Embedding problem, review known complexity results and algorithmic approaches, and summarize relevant theoretical findings from recent literature, including bounds on book thickness and embeddability.

Formal Definitions

Definition 2.3 (Upward Book Embedding for Directed Graph). *An upward book embedding of a directed graph $G = (V, E)$ consists of:*

1. *a linear ordering $\pi : V \to \{1, 2, \ldots, |V|\}$ of vertices along the spine of the book, and*

2. *an assignment $\sigma : E \to \{1, \ldots, k\}$ of edges to one of k pages such that no two edges on the same page geometrically cross. Additionally, for every edge (u, v), it must hold that $\pi(u) < \pi(v)$.*

Definition 2.4 (kUBE Problem)**.** *The k-Page Upward Book Embedding Problem (kUBE) is defined as follows: Given a directed graph G*[1]*, does there exist a k-page upward book embedding for G?*

Complexity and Algorithms

Theorem 2.2 (Bekos et al. [6]; Heath and Pemmaraju [27])**.** *For any $k \geq 2$, the kUBE problem is NP-complete. Meanwhile, 1UBE can be solved in $O(n)$ time.*

The complexity profile of kUBE mirrors that of kBE in several ways. Like kBE, the 1-page case is tractable, while the problem becomes NP-complete as soon as multiple pages are involved. However, the additional topological constraints in kUBE make algorithmic solutions more sensitive to graph structure.

Binucci et al. [9] show that for *planar st-graphs*, the 2UBE problem can be solved in $O(f(\beta) \cdot n + n^3)$ time, where β is the branchwidth and f is a singly-exponential function.

These findings suggest that while kUBE is NP-complete in the general case, it may still be efficiently solvable for graphs with special structural properties.

Theoretical Properties

While the book thickness of many undirected graphs is known (e.g., $\lceil n/2 \rceil$ for K_n), the upward book thickness of general DAGs remains largely unexplored. No tight bounds are currently known for this variant.

Nevertheless, several important subclasses of DAGs have been shown to always admit a 2-page upward book embedding:

- N-free upward planar digraphs [34]
- Plane st-graphs with a special face structure [9]

[1] An upward book embedding can exist only if G is acyclic.

- DAGs with constant branchwidth [9]

Example

Figure 2.1 illustrates a 3UBE instance. The input DAG has 6 nodes and 15 edges (Figure 2.1a). In the embedding (Figure 2.1b), nodes are placed in topological order along a horizontal spine, and edges—colored to indicate page assignment—are drawn such that no two edges on the same page cross.

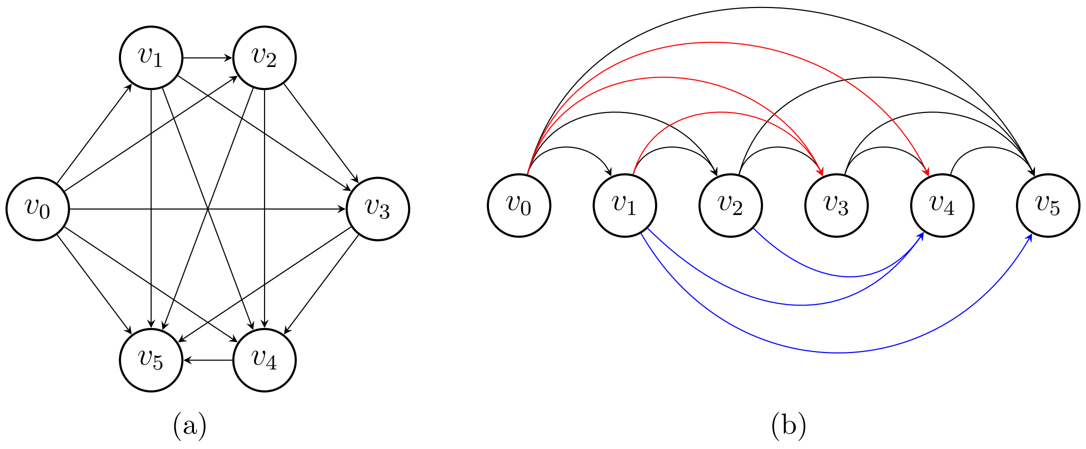


Figure 2.1: (a) A DAG with 6 nodes and 15 edges. (b) Its 3-page Upward Book Embedding (3UBE), with edge partitions colored black, blue, and red.

2.3 Other Book Embedding Variants and Complexity Landscape

Some variants of the book embedding problem introduce the notion of *partitioning*, where the assignment of edges to pages is provided as part of the input. For example, in the case of $k = 2$, each edge is already assigned to either page 1 or page 2. The task then reduces to finding a vertex ordering along the spine such that no two edges on the same page cross. Since the partitioning is fixed, these problems avoid

Problem	$k = 1$	$k = 2$	$k = 3$	$k \geq 4$
kBE	$O(n)$ [42]	NP-c. [13] $2^{O(\sqrt{n})}$ [23]	NP-c. [13]	NP-c. [13]
kUBE	$O(n)$ [27]	NP-c. [6]	NP-c. [9]	NP-c. [9]
kPBE	$O(n)$ [42]	$O(n)$ [28]	NP-c. [4]	NP-c. [4]
kUPBE	$O(n)$ [27]	OPEN	NP-c. [3]	NP-c. [3]
kUMPBE	$O(n)$ [27]	$O(n)$ [3]	OPEN	NP-c. [3]

Table 2.1: Complexities of book embedding problems.

Note: ”NP-c.” stands for ”NP-complete”.

the combinatorial overhead of exploring multiple edge assignments and are therefore expected to be computationally simpler in some cases.

These are some of the partitioned variants described in literature:

- **k-Page Partitioned Book Embedding (kPBE)**: the undirected version with fixed page assignments [28].
- **k-Page Upward Partitioned Book Embedding (kUPBE)**: the directed acyclic variant with upward edge constraints [3].
- **k-Page Upward Matching-Partitioned Book Embedding (kUMPBE)**: a restricted case of kUPBE where each page contains a matching [3].

The complexity results for all the book embedding problems discussed so far, including these partitioned variants, are summarized in Table 2.1. The table also highlights several open cases that remain unresolved at the time of writing.

2.4 Boolean Satisfiability (SAT)

The Boolean Satisfiability Problem (SAT) is a classical problem of determining whether a given boolean formula can be satisfied, that is, whether there is an assignment of true/false values to variables such that the entire formula evaluates to true. SAT was the first problem proven to be NP-complete [15]. Despite the high computational complexity, advances in algorithms used for SAT solving over the past two decades have made it possible to apply SAT solvers to a wide variety of real-world and theoretical problems.

SAT is typically applicable to problems that can be expressed through a finite set of binary decisions. Applications range from hardware verification and scheduling to graph problems and cryptography. In the context of our work, SAT is used to determine whether a given directed acyclic graph can be embedded in an upward book layout with a given number of pages.

Satisfiability and Unsatisfiability

A Boolean formula is satisfiable if there exists at least one assignment of truth values to its variables that makes the entire formula evaluate to true. For instance, the formula

$$(x \vee y) \wedge (\neg x \vee y) \tag{2.1}$$

is satisfiable. Assigning $x =$ false and $y =$ true satisfies both clauses. In contrast, a formula such as

$$x \wedge \neg x \tag{2.2}$$

is unsatisfiable, since no assignment can make both clauses true simultaneously.

SAT solvers accept formulas expressed in a particular format, most often Conjunctive Normal Form (CNF), where the formula is a conjunction of disjunctions of literals. While this format is restrictive, it has become the standard input representation due to its compatibility and alignment with efficient solving techniques.

Encoding Problems into SAT

The process of encoding a problem into SAT typically involves three steps. First, a set of boolean variables is defined to describe the decisions and elements of the original problem at hand. It is common for a group of boolean variables to represent a single concept. For example, three boolean variables can be used to represent three different possible label assignments for a particular node. Second, constraints that define valid configurations are expressed as logical relationships among those variables. Finally, these constraints are converted into CNF format.

Many well-known combinatorial problems can be encoded in this way. For example, graph coloring problems can be reduced to SAT by introducing variables for each vertex-color pair and adding clauses to ensure adjacent vertices do not share the same color. In a similar fashion, SAT-based methods have been successfully used to tackle scheduling, resource allocation, and routing problems. In our case, node ordering and edge-to-page assignments are encoded as boolean variables, and structural embedding constraints on upwardness and non-crossing of edges are transformed into clauses.

SAT Solvers

Once the problem is encoded into CNF, it can be passed to a SAT solver. These solvers apply a combination of techniques to explore the space of possible assignments efficiently. The most widely used class of solvers today follow the *Conflict-Driven Clause Learning* (CDCL) paradigm. These solvers make decisions by selecting a variable and assigning it a value, propagating the consequences of that decision, and backtracking when a conflict is found. It is interesting to note that they learn from conflicts by creating and adding new clauses that prevent the same failure from occurring again.

When problem is executed, the solver returns either `SAT` or `UNSAT`. If the result is `SAT`, the output includes a model—a list of integers representing a satisfying assignment. Each integer corresponds to a variable index, with a minus sign indicating that the variable is assigned `false`, and a positive value indicating `true`. This output can then be decoded back into a solution to the original problem. In our case, this involves reconstructing a linear order of the nodes and a page assignment for each edge in the graph.

Numerous SAT solvers have been developed and evaluated through annual competitions, with many optimized for specific classes of problems. Solvers such as MiniSAT, Glucose, and CaDiCaL have historically performed well, each implementing variants of CDCL with different heuristics for branching, restart policies, and clause learning.

In this study, we utilize *Kissat* [8], a modern SAT solver known for its efficiency and performance in recent SAT competitions. Kissat incorporates several state-of-the-art techniques such as inprocessing, phase saving, and aggressive clause minimization, making it well-suited for structured SAT instances like those arising from graph embedding problems.

DIMACS CNF Format

SAT solvers typically operate on input written in the DIMACS CNF format. This format represents each clause as a space-separated list of integers, each corresponding to a variable or its negation, terminated by a zero. When the satisfiable boolean expression in Equation 2.1 is encoded in DIMACS CNF format, it takes the following form:

```
p cnf 2 2
1 2 0
-1 2 0
```

Where the header `p cnf 2 2` indicates that the formula contains 2 variables and 2 clauses, the line `1 2 0` represents the clause $x \vee y$, with the terminating `0` marking the end of the clause. The line `-1 2 0` corresponds to the clause $\neg x \vee y$. Note that the minus sign indicates logical negation.

A minimal example of an unsatisfiable DIMACS CNF file is an encoding of expression in Equation 2.2:

```
p cnf 1 2
1 0
-1 0
```

Why Use SAT for kUBE

The k-page upward book embedding problem is NP-complete for $k \geq 2$, which most certainly indicates that direct algorithmic solutions would scale poorly with instance size. Encoding the problem into SAT allows us to utilize the power of modern solvers to either find valid embeddings or prove that none exist. The completeness of the SAT approach is a major advantage, as proving the non-embeddability of a graph can be as important as finding a valid embedding.

Through reducing the embedding problem to a SAT instance, we transfer the complexity of the search to a well-tested and optimized solving engine. This approach not only provides a practical way to solve larger instances, but also allows for easier experimentation with different encoding strategies.

2.5 Constraint Programming (CP)

In constraint programming (CP), users define a set of decision variables along with constraints that must be satisfied. These variables can take on various forms such as boolean, integer, and others. Constraints may include boolean logic, arithmetic equalities and inequalities, set-based conditions, and more. CP subsumes satisfiability problems (SAT) as a subset, and therefore can be used to solve all problems solvable by SAT solvers. While SAT solvers are often more optimized in practice due to decades of refinement, CP offers more flexibility and can be more efficient for problems that are naturally expressed using integer variables and arithmetic relations.

Several researchers have successfully applied CP to solve complex graph-related problems, demonstrating its suitability for structured combinatorial domains, including our target book embedding problem [22], [25].

Encoding Problems into CP

Constraint Programming (CP) is an effective method for solving combinatorial problems by using decision variables and constraints. To encode a problem into CP, one must define decision variables, specify their domains, and establish constraints that correctly and completely describe the problem.

Decision variables represent unknown values within the problem. Each decision variable is associated with a domain, which is the set of possible values it can take. Constraints are then defined to capture relationships between decision variables using arithmetic or logical rules.

For example, constraints may require decision variables to take on distinct values (using the `alldifferent` constraint), or to lie within specific numerical bounds. An objective function can be introduced in cases where optimization is necessary—such as minimizing or maximizing a particular outcome.

To demonstrate, we provide a sample encoding of the Sudoku problem in CP. This encoding captures the full structure of the puzzle and can serve as valid input to CP solvers when converted to the appropriate syntax required by the solver in use.

Sudoku Example: We define a constraint programming (CP) model for the standard 9×9 Sudoku puzzle. The problem consists of assigning a digit from 1 to 9 to each cell in a 9×9 grid such that each row, column, and 3×3 subgrid contains all digits exactly once.

Variables:

$$S[i][j] \in \{1, 2, \ldots, 9\} \quad \forall i, j \in \{1, \ldots, 9\}$$

where $S[i][j]$ denotes the value assigned to the cell located at row i and column j of the grid. Each decision variable represents a single cell and must take an integer value between 1 and 9.

Constraints:

- **Row constraints:**

$$\texttt{AllDifferent}(S[i][1], S[i][2], \ldots, S[i][9]) \quad \forall i \in \{1, \ldots, 9\}$$

  For each row i, the nine cells must take distinct values, ensuring that each digit from 1 to 9 appears exactly once in every row.

- **Column constraints:**

$$\texttt{AllDifferent}(S[1][j], S[2][j], \ldots, S[9][j]) \quad \forall j \in \{1, \ldots, 9\}$$

For each column j, the nine cells must also take distinct values, guaranteeing that each digit from 1 to 9 appears exactly once in every column.

- **Block constraints (3×3 subgrids):**

$$\texttt{AllDifferent}\left(\{S[i][j] \mid i \in \{r+1, \ldots, r+3\},\right.$$
$$\left. j \in \{c+1, \ldots, c+3\}\}\right) \quad \forall r, c \in \{0, 3, 6\}$$

  For each 3×3 subgrid defined by the top-left corner coordinates $(r+1, c+1)$, the nine cells within the block must take distinct values, ensuring that each digit from 1 to 9 appears exactly once within every subgrid.

CP Solvers

Modern CP solvers have demonstrated considerable advances in addressing real-world combinatorial problems. Among these, Google's CP-SAT solver—part of the OR-Tools suite—has consistently ranked at the top of the MiniZinc Challenge since 2013 [35]. CP-SAT adopts a hybrid strategy by integrating classical CP propagation with SAT-solving techniques.

Although several high-quality solvers exist on the market—including PicatSAT and IBM ILOG CP Optimizer—in this work, we have chosen to focus exclusively on Google's CP-SAT. The decision stems not only from its proven track record in benchmark competitions but also from the broader goal of the study. The primary objective is not to compare individual solvers within the same paradigm, but rather to contrast the paradigms themselves—Constraint Programming versus pure SAT-solving. Hence, it is only logical to employ the most capable solver representative of each paradigm. Accordingly, our experiments are based on using a pure SAT encoding evaluated with a state-of-the-art SAT solver, and a CP encoding evaluated with CP-SAT, which is widely recognized as best-in-class for constraint-based models.

Chapter 3

Upward Book Embedding Encoding Techniques

In this chapter, we first formally introduce a kUBE SAT encoding, which is an adapted version of the kBE SAT encoding described by Bekos et al. [7]. Given a graph $G = (V, E)$, this encoding determines whether the edges in E can be embedded in a book with k pages. This encoding is code-named **SAT-1**.

In an effort to improve upon this encoding, we examine the constrained case of $k = 2$ (which is known to be NP-complete) and investigate whether the number of variables can be reduced. This endeavor has proven successful, as we were able to significantly reduce the number of required variables. The resulting encoding for 2UBE is then formally described and code-named **SAT-2**.

Finally, we introduce a novel but straightforward CP encoding, which uses a different approach for encoding the positioning of nodes: integer variables for node indices instead of boolean variables for relative ordering. The logical constraint for ensuring no edge crossings is mostly identical to that of SAT-1.

3.1 SAT-1 for kUBE

Let $G = (V, E)$, where $V = \{v_1, v_2, \ldots, v_n\}$ and $E = \{e_1, e_2, \ldots, e_m\}$, be a directed acyclic graph for which we seek to decide whether it admits an upward book embedding in $k \geq 2$ pages. We define a logical formula $\mathcal{F}_1(G, k)$ that encodes this decision problem as a SAT instance. This encoding, referred to as *SAT-1*, follows the approach introduced by Bekos et al. [7], and is specified by a set of propositional variables and rules. These rules capture relative vertex ordering, transitivity, edge-to-page assignments, and a simplified prohibition of edge crossings. Since each rule is expressed in propositional logic, the full formula can be translated into conjunctive normal form (CNF) in a straightforward manner.

Relative ordering of vertices. The vertices of G must be arranged along the book spine in a specific order. For each pair of vertices (v_i, v_j) with $i < j$, we define a variable $L(v_i, v_j)$, which is true if and only if $\pi(v_i) < \pi(v_j)$. To enforce asymmetry, we define $L(v_i, v_j)$ only for $i < j$, ensuring that $L(v_i, v_j) \iff \neg L(v_j, v_i)$.

Transitivity. The ordering must be transitive, meaning cyclic dependencies such as $L(v_i, v_j) \wedge L(v_j, v_k) \wedge L(v_k, v_i)$ are not allowed. Thus, for all pairwise distinct vertices $v_i, v_j, v_k \in V$, it is required that $L(v_i, v_j) \wedge L(v_j, v_k) \rightarrow L(v_i, v_k)$. That is, the following transitivity constraint, presented in conjunctive normal form (CNF), must hold:

$$[\neg L(v_i, v_j), \neg L(v_j, v_k), L(v_i, v_k)] \tag{3.1}$$

Topological ordering. kUBE requires that all directed edges are oriented in a singular direction along the spine of the book. This property is also referred to as *topological ordering* of vertices. To enforce this, we add this clause for each of the graph's edges:

$$L(u, v) \quad \forall (u, v) \in E \tag{3.2}$$

Edge assignment. A set of variables to represent the assignment of edges to pages. For each $e_i \in E$ and each $p \in [0, k)$, where k is the maximum number of available pages, the variable $EP(e_i, p)$ is true if and only if the edge e_i is assigned to page p. Every edge must be assigned to at least one page, ensuring the following constraint holds for all $e_i \in E$:

$$EP(e_i, 0) \vee EP(e_i, 1) \vee \cdots \vee EP(e_i, k-1) \tag{3.3}$$

Same page edges. For any two edges e_i and e_j, the variable $X(e_i, e_j)$ is true if and only if both edges belong to the same page. Formally, for all $e_i, e_j \in E$ with $i < j$, the following constraint holds:

$$\big(EP(e_i, p) \wedge EP(e_j, p)\big) \rightarrow X(e_i, e_j) \tag{3.4}$$

No edge crossings on the same page. As we are dealing with DAGs only in kUBE problem, the edge crossing prohibition rule introduced by Bekos et al. [7] for undirected graphs is no longer applicable. That is, given two undirected edges $e_i = (a, b)$ and $e_j = (c, d)$, the permutations would be (a, c, b, d), (a, d, b, c), (b, c, a, d), (b, d, a, c), (c, a, d, b), (c, b, d, a), (d, a, c, b), and (d, b, c, a). However, if e_i and e_j are directed edges, we do not need to consider permutations that would inverse the direction of edges, reducing the cases with (a, c, b, d) and (c, a, d, b) only. To prohibit the illegal subsequences of vertices (a, c, b, d) and (c, a, d, b), for any two edges $e_i = (a, b)$ and $e_j = (b, c)$ such that $i < j$, we encode the rule:

$$X(e_i, e_j) \rightarrow \neg Cross(e_i, e_j) \tag{3.5}$$

where $Cross(e_i, e_j)$, indicating that two edges intersect, is defined as:

$$Cross(e_i, e_j) \equiv (L(a, c) \wedge L(c, b) \wedge L(b, d)) \vee (L(c, a) \wedge L(a, d) \wedge L(d, b)) \tag{3.6}$$

That is, for any two edges $e_i = (a, b),\ e_j = (c, d)$ such that $a \neq b,\ b \neq c,\ c \neq d,\ a \neq d$, we have the following two CNF clauses:

$$\begin{gathered}[\neg X(e_i, e_j), \neg L(a, c), \neg L(c, b), \neg L(b, d)] \\ [\neg X(e_i, e_j), \neg L(c, a), \neg L(a, d), \neg L(d, b)]\end{gathered} \tag{3.7}$$

We finish this section with a justification on the soundness and completeness of the encoding and the fact that it is bounded by the polynomial total number of variables and clauses.

Theorem 3.1. *Given a DAG $G = (V, E)$, where $|V| = n$ and $|E| = m$, and $k \in \mathbb{N}$, the graph G admits an upward book embedding on k pages if and only if $\mathcal{F}_1(G, k)$ is satisfiable. More precisely, G admits kUBE if and only if there exists a model $\mathcal{M}$ for $\mathcal{F}_1(G, k)$, i.e., $\mathcal{M} \models \mathcal{F}_1(G, k)$. Further, the size of $\mathcal{F}_1(G, k)$ is bounded by*

$O(n^2 + m^2 + mk)$ variables and $O(n^3 + m^2)$ clauses.

Proof. We establish both directions of the equivalence.

$\Rightarrow$: If G admits an upward book embedding on k-pages, $\mathcal{M} \models \mathcal{F}_1(G, k)$. Suppose G has a valid k-page upwards book embedding $\mathcal{E}(G, k)$ and let $(\hat{L}, \hat{EP}, \hat{X})$ be an assignment to variables L, EP, and X of $\mathcal{F}_1(G, k)$ with their meaning as follows: (i) $\hat{L}(v_i, v_j) = true$, if and only if v_i is positioned before v_j; (ii) $\hat{EP}(e, p) = true$, if and only if edge e is assigned to page p; (iii) $\hat{X}(e_i, e_j) = true$, if and only if both edges e_i and e_j lie on the same page.

For satisfaction of the rules of *transitivity*, *edge assignment*, and *same page edges* by $(\hat{L}, \hat{EP}, \hat{X})$, the arguments by [Theorem 1]bekos2015book can be exactly adopted. We have our focus on the remaining two rules of *topological ordering* and *no edge crossings on the same page.*

The *topological ordering rule* is satisfied by $\hat{L}$ because, for all G edges (u, v), the condition $\pi(u) < \pi(v)$ holds according to the definition of upward book embedding. Thus, $\hat{L}(u, v)$ holds true for all edges.

Finally, we prove that *no edge crossings on the same page rule* is satisfied by contradiction. Suppose that $(\hat{L}, \hat{EP}, \hat{X})$ assignment violates the no-crossing rule for some pair of edges (a, b) and (c, d), which means $\hat{X}((a, b), (c, d)) = true$ and vertices form a subsequence (a, c, b, d) or (c, a, d, b). For (a, c, b, d), the spine of the book together with the edge (a, b) in $\mathcal{E}(G, k)$ forms an enclosed region. Accordingly, vertex c lies within the region while d lies outside. That is, for d to connect with c, the edge (c, d) must cross (a, b). The argument on (c, a, d, b) is similar. As a result, edges must cross in $\mathcal{E}(G, k)$, leading to contradiction.

$\Leftarrow$: If $\mathcal{M} \models \mathcal{F}_1(G, k)$, G admits an upward book embedding on k-pages.

Let $(\hat{L}, \hat{EP}, \hat{X})$ be a satisfying assignment to $\mathcal{F}_1(G, k)$. As shown by Bekos et al. [7], every satisfying assignment to $\mathcal{F}_1(G, k)$ corresponds to a valid k-page book embedding. Since $\hat{L}$ assignment satisfies the *topological ordering rule*, $\pi(u) < \pi(v)$ must hold for all edges (u, v). This guarantees that the resulting vertex ordering is *upward*, as required by 2UBE definition. Thus, any satisfying assignment to $\mathcal{F}_1(G, k)$ represents a valid upward book embedding on k pages.

The formula $\mathcal{F}_1(G, k)$, defines three sets of variables: L, EP, and X. Their cardinality are respectively upper-bounded by $\frac{n(n-1)}{2}$, mk, and $\frac{m(m-1)}{2}$. Thus, the total count of variables is bounded by $O(n^3 + m^2 + mk)$. The number of clauses is bounded by $O(n^3 + m^2)$, due to the *transitivity* rules $O(n^3)$ and the *no-edge-crossings-on-the-same-page* rules $O(m^2)$. □

3.2 SAT-2 for 2UBE

We begin the definition of $\mathcal{F}_2(G, 2)$ by including the *relative ordering of vertices*, *transitivity*, and *topological ordering* constraints from $\mathcal{F}_1(G, k)$.

Since 2UBE is restricted to only two pages, we can conceptualize one page being above and another below the spine of the book. Therefore, a single boolean variable, denoted as $T(e_i)$, suffices to represent the page assignment of edge e_i. More precisely, the variable $T(e_i)$ is true if and only if the edge e_i is assigned to the top page, i.e., the page above the spine of the book. Due to the compact nature of the variable set T, we no longer need additional rules to enforce the correct representation of the edge-to-page assignment as in the case of EP. Any configuration of T inherently represents a valid distribution of edges between the two available pages.

No edge crossings on the same page. To account for the absence of variable $X(e_i, e_j)$, which indicates if two edges belong to the same page, we need to separately handle cases where both edges lie either on the top page or the bottom one. For that, we need two implicative rules, which use $Cross(e_i, e_j)$, defined in Equation (3.6). For the top page:

$$T(e_i) \land T(e_j) \rightarrow \neg Cross(e_i, e_j) \tag{3.8}$$

and for the bottom page:

$$\neg T(e_i) \land \neg T(e_j) \rightarrow \neg Cross(e_i, e_j) \tag{3.9}$$

This completes the construction of $\mathcal{F}_2(G, 2)$.

Finally, we provide a formal proof of soundness and completeness of the above-described encoding, as well as the polynomial bounds on the number of variables and clauses.

Theorem 3.2. *Given a DAG $G = (V, E)$, where $|V| = n$ and $|E| = m$, G admits an upward book embedding on 2 page if and only if $\mathcal{F}_2(G, 2)$ is satisfiable. More precisely, G admits 2UBE if and only if there exists a model $\mathcal{M}$ for $\mathcal{F}_2(G, 2)$, i.e., $\mathcal{M} \models \mathcal{F}_2(G, 2)$. Further, the size of $\mathcal{F}_2(G, 2)$ is bounded by $O(n^3 + m^2)$ clauses and $O(n^2 + m)$ variables.*

Proof. We again establish both directions of the equivalence.

$\Rightarrow$: If G admits an upward book embedding on 2 pages, $\mathcal{M} \models \mathcal{F}_2(G, 2)$. Assume that G admits an upward 2-page book embedding $\mathcal{E}(G, 2)$. By Theorem 3.1, a valid 2-page upward embedding yields a satisfying assignment $(\hat{L}, \hat{EP}, \hat{X})$ to $F_1(G, 2)$.

The rules of *relative ordering of vertices*, *transitivity*, and *topological ordering* in $F_2(G, 2)$ are identical to the ones in $F_1(G, 2)$, they are thus satisfied by $\hat{L}$.

Further, $F_1(G, 2)$ encodes edge-to-page assignment with $EP(e, p)$, where $p \in [0, 1]$. We can define $\hat{T}$ assignments from $\hat{EP}$ using this formula:

$$\hat{T}(e) = \begin{cases} \texttt{true} & \text{if } \hat{EP}(e, 0) \text{ is true,} \\ \texttt{false} & \text{if } \hat{EP}(e, 1) \text{ is true.} \end{cases} \tag{3.10}$$

To show that the rules of *no-edge-crossings-on-the-same-page* in $F_2(G, 2)$ (3.8 and 3.9) are equivalent to 3.5 in $F_1(G, 2)$, we show that the following is true:

$$X(e_i, e_j) \iff (T(e_i) \wedge T(e_j)) \vee (\neg T(e_i) \wedge \neg T(e_j)). \tag{3.11}$$

From $T(e) \equiv EP(e, 0)$ and $\neg T(e) \equiv EP(e, 1)$, we make substitutions to obtain

$$X(e_i, e_j) \iff (EP(e_i, 0) \wedge EP(e_j, 0)) \vee (EP(e_i, 1) \wedge EP(e_j, 1)) \tag{3.12}$$

The right-hand side now matches exactly the definition of $X(e_i, e_j)$, meaning that the *no-edge-crossing* constraint is satisfied.

$\Leftarrow$: If $\mathcal{M} \models \mathcal{F}_2(G, 2)$, G admits an upward book embedding on 2 pages.

That is, suppose there is a satisfying assignment $(\hat{L_1}, \hat{T})$ for $\mathcal{F}_2(G, 2)$, there is a satisfying assignment $(\hat{L_1}, \hat{T})$ corresponding to a satisfying assignment $(\hat{L_2}, \hat{EP}, \hat{X})$ for $\mathcal{F}_1(G, 2)$.

Since $\mathcal{F}_1(G, 2)$ and $\mathcal{F}_2(G, 2)$ impose identical constraints for vertex ordering, it follows that $\hat{L_2} = \hat{L_1}$. Next, we construct $\hat{EP}$ from $\hat{T}$ assignment for each edge e, as follows:

$$\hat{EP}(e,0) = \hat{T}(e), \quad \hat{EP}(e,1) = \neg\hat{T}(e) \tag{3.13}$$

The $\hat{X}$ assignment can be derived from $\hat{EP}$ according to its definition. The *no-edge-crossings-on-the-same-page* rules for $F_1(G,2)$ and $F_2(G,2)$ are equivalent as shown above. Therefore, the rules are satisfied. To conclude, we have a satisfying assignment $(\hat{L_2}, \hat{EP}, \hat{X})$ for $\mathcal{F}_1(G,2)$, it follows from Theorem 3.1 that G has a valid 2-page upward book embedding.

$\mathcal{F}_2(G,2)$ defines the sets of variables L and T, with their cardinality are respectively upper-bounded by $\frac{n(n-1)}{2}$ and m. Therefore, the number of variables is bounded by $O(n^2 + m)$. The number of clauses is dominated by the *transitivity rules* $O(n^3)$ and the *no-edge-crossings-on-the-same-page rules* $O(m^2)$, which results in $O(n^3+m^2)$ total clauses. □

3.3 A CP Encoding for 2UBE

Compared to SAT, Constraint Programming (CP) can leverage integer variables and algebraic constraints that are often difficult or verbose to express solely through boolean expressions. Therefore, to compare the effectiveness of both of these approaches, we describe an alternative CP encoding for the 2UBE problem. Later in this study, we refer to this encoding as simply **CP**.

Given a DAG $G = (V, E)$, where $|v| = n$ and $|E| = m$, the CP encoding introduces two sets of decision variables to model the 2UBE problem. The first set describes the positions of vertices on the spine of the book:

$$\text{pos}[i] \in \{0, \ldots, n-1\}, \quad \forall i \in \{0, \ldots, n-1\} \tag{3.14}$$

The second set consists of m Boolean variables for the m edge-to-page assignments.

$$\text{page}[i] \in \{0, 1\}, \quad \forall i \in \{0, \ldots, m-1\} \tag{3.15}$$

To ensure that each node has a unique position on the spine, or, in other words, no two nodes share the same position, we add the following constraint:

$$\text{AllDifferent}(\text{pos}[0], \text{pos}[1], \dots, \text{pos}[n-1]) \tag{3.16}$$

Furthermore, we must enforce the topological ordering of nodes by following the direction of edges:

$$\text{pos}[u] < \text{pos}[v] \quad \forall (u, v) \in E \tag{3.17}$$

Finally, no two edges assigned to the same page are allowed to cross. That is, for any two edges $e_i, e_j \in E$, where $i \neq j$, $e_i = (a, b)$, and $e_j = (c, d)$, we have that:

$$\begin{aligned}\big(\text{page}[i] = \text{page}[j]\big) \implies \big(&\neg(\text{pos}[a] < \text{pos}[c] < \text{pos}[b]) \\ &\wedge \neg(\text{pos}[c] < \text{pos}[a] < \text{pos}[d] < \text{pos}[b])\big)\end{aligned} \tag{3.18}$$

In closing, we provide a formal proof of completeness and soundness of the proposed CP encoding, thus validating its correctness in the context of solving 2UBE problems.

Theorem 3.3. *Given a DAG $G = (V, E)$, where $|V| = n$ and $|E| = m$, the graph G admits a 2-page upward book embedding if and only if the CP encoding is solvable.*

Proof. We establish both directions of the equivalence.

$\Rightarrow$: Suppose that G admits a valid 2-page upward book embedding $\mathcal{E}(G, 2)$. We show that the CP model is solvable.

Assign to each vertex v_i a value pos$[i]$ representing its position along the book spine, and to each edge e_i a page assignment page$[i]$, where 0 denotes the bottom page and 1 denotes the top page. Since $\mathcal{E}(G, 2)$ is valid, all vertex positions are distinct, satisfying the `AllDifferent` constraint.

Moreover, for each edge $(u, v) \in E$, the topological order of the embedding ensures that pos$[u] <$ pos$[v]$, satisfying the upwardness constraint. Finally, since no two edges placed on the same page cross, the disjunctive no-crossing constraints are satisfied. Thus, the assignment derived from $\mathcal{E}(G, 2)$ constitutes a valid solution to the CP encoding.

$\Leftarrow$: Suppose that the CP model has a solution. We show that G admits a valid 2-page upward book embedding.

The `AllDifferent` constraint ensures that vertex positions are distinct, yielding a unique ordering along the spine. The constraint $\text{pos}[u] < \text{pos}[v]$ for each $(u, v) \in E$ ensures that the ordering respects the direction of edges, satisfying upwardness. Furthermore, the no-crossing constraints guarantee that any two edges assigned to the same page do not cross.

Therefore, the solution to the CP model defines a valid 2-page upward book embedding, with vertex positions given by pos and page assignments given by page.

$\square$

Chapter 4

Empirical Evaluation of Encodings

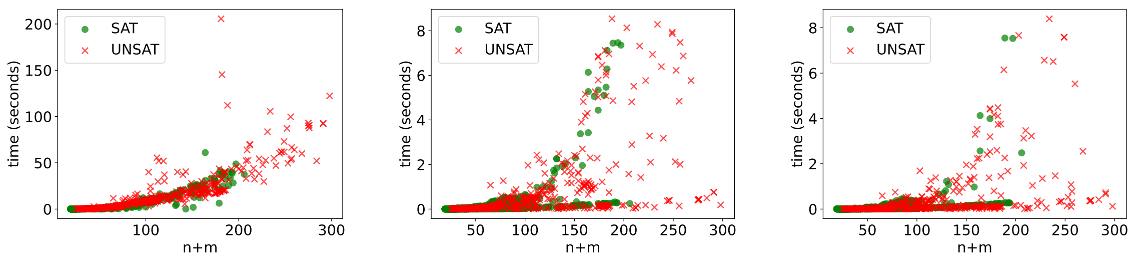
In this section, we evaluate the performance of the proposed methods: SAT-1, SAT-2, and CP. For the CP solver, we use Google's open source OR-Tools CP-SAT Solver [37]. For the SAT solver, we use Kissat [8], an efficient single-threaded SAT solver that is considered state-of-the-art at the time of writing. All experiments/tests were run on a 6-core Intel Core i7 2.6GHz CPU. Timeout for a single instance was set to 3600 seconds and the memory was limited to 16 GB. To measure the time required to solve each instance with statistical rigor, we used Hyperfine, a popular command-line benchmarking tool. Hyperfine was configured with a single untimed warmup run prior to measurement to mitigate the influence of startup overhead and disk I/O.

We first assess the performance of SAT-1, SAT-2, and CP on the North Graph dataset, a widely used benchmark for graph embedding problems, focusing on runtime distributions and detailed comparisons in Section 4.1. Section 4.2 extends the analysis to larger, structured instances by benchmarking scalability on directed grid graphs. Finally, Section 4.3 summarizes the key observations and conclusions drawn from the empirical results.

4.1 Benchmarking on "North" DAG Dataset

North Graphs (downloaded from [24]) is a popular dataset consisting of 1277 DAGs collected by Stephen North with vertex counts ranging from 10 to 100 and edge counts ranging from 9 to 241 [12]. Processing of the entire dataset took approximately 9 minutes for SAT-1, 4.5 minutes for SAT-2, and 2.8 hours for CP. Exactly 800 DAGs are satisfiable (i.e., allowing a 2-page upwards book embedding), while the remaining 477 are unsatisfiable.

All instances are solvable by SAT-1, SAT-2, and CP, but as shown in Figure 4.1, SAT-based methods are significantly faster than CP. While SAT-1 and SAT-2 perform similarly, SAT-2 appears faster. Most instances are relatively easy to solve, as indicated by the clustering of data points near the horizontal axis, meaning shorter runtimes. However, despite all methods solving the same set of instances, individual runtimes vary significantly, leading to different point distributions across the three subfigures in Figure 4.1.

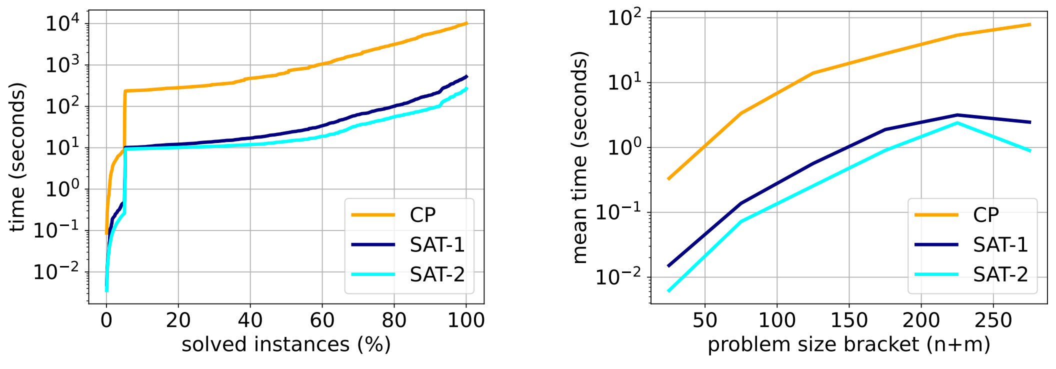


(a) CP (b) SAT-1 (c) SAT-2

Figure 4.1: Runtime of CP (a), SAT-1 (b), and SAT-2 (c) for North Graph DAGs. Each graph instance is sized by $m + n$, where n is the number of nodes and m is the number of edges. Data points: green dots represent satisfiable instances; red crosses indicate unsatisfiable ones.

To quantify runtime, cumulative and mean runtime as Cactus plots are presented in Figures 4.2a and 4.2b, respectively. Note that the y-axis uses a logarithmic scale. We observe that CP is significantly slower than the SAT approaches: approximately 20 times slower than SAT-1 and 40 times slower than SAT-2. Additionally, SAT-2 consistently outperforms SAT-1 across problems of all sizes (Figure 4.2b). Next, we investigate this performance difference in closer detail.

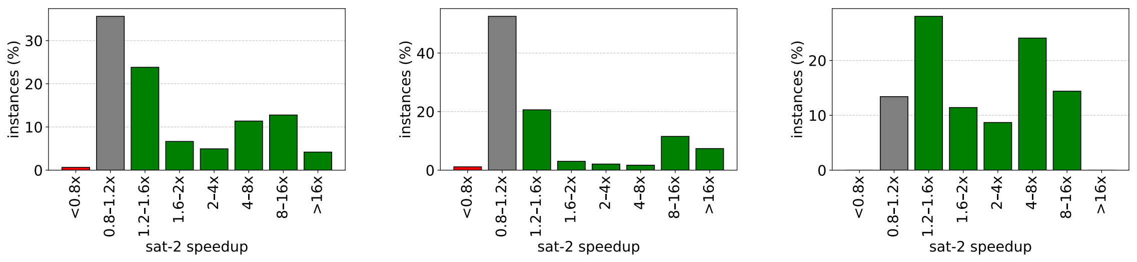
We first define “significant performance improvement” as a speedup factor between $0.8x$ and $1.2x$. Using this measure, approximately 65% of instances show a significant improvement of SAT-2 over SAT-1, while less than 1% show deteriorated performance. About one-third of problems show no significant improvement. The results are plotted as histograms in Figure 4.3, where satisfiable and unsatisfiable in-



(a) (b)

Figure 4.2: (a) Cumulative runtime. (b) Mean runtime per problem bracket.

stances are further distinguished. Notably, there is a significant difference in speedup distribution between satisfiable and unsatisfiable instances. Over 85% of UNSAT instances show significant performance improvements with SAT-2 over SAT-1, with no deterioration. In contrast, only around half of SAT instances show meaningful improvement using SAT-2, and around 1% experience slight performance deterioration. Interestingly, all problems with a speedup factor above $16x$ are SAT instances.



(a) All instances (b) SAT instances (c) UNSAT instances

Figure 4.3: Speedup distribution of SAT-2 over SAT-1 across North Graph DAGs (Green: improved performance, gray: similar performance, red: deteriorated).

These observations of the large disparity in runtime improvement raise the question: Why SAT-2 performs much better for certain problems while offering no improvement over SAT-1 for others? In hopes of answering this question, we started by exploring the relationship of SAT-1 runtime with various parameters.

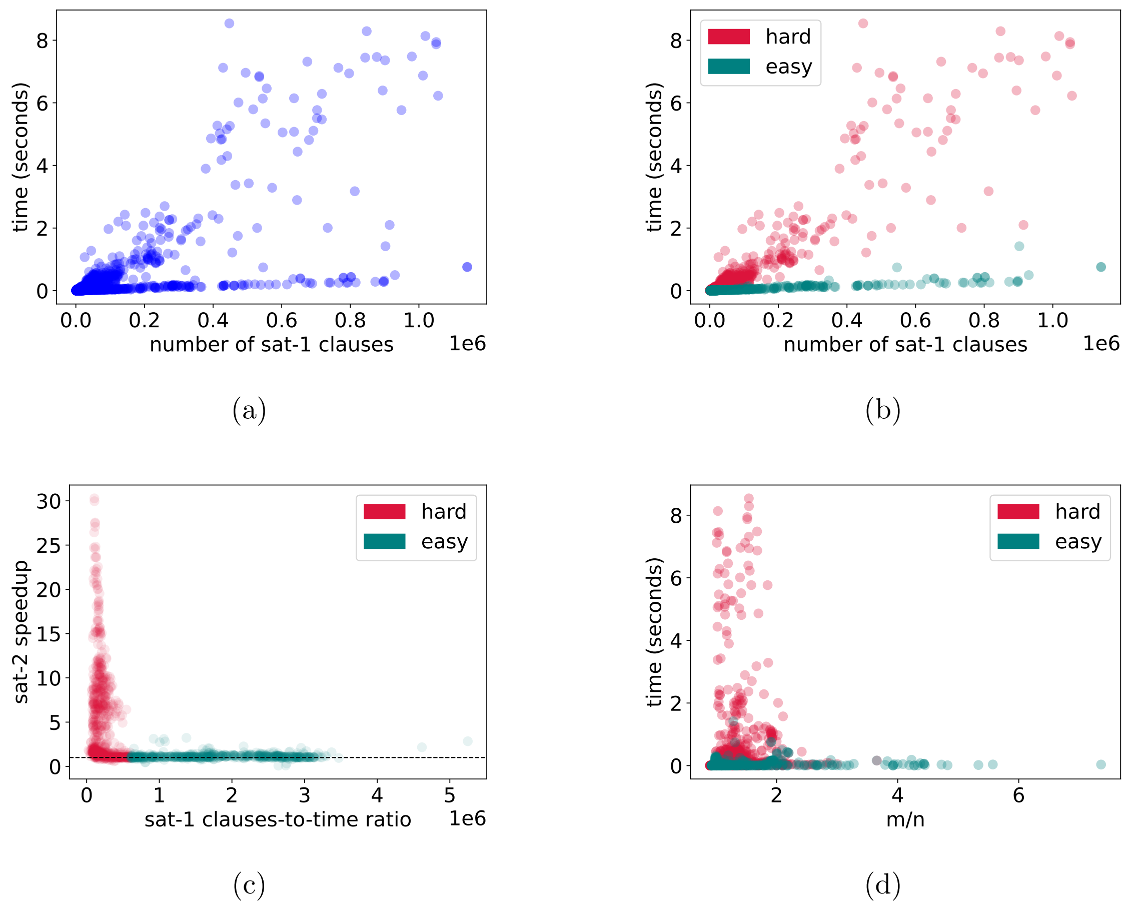


Figure 4.4: (a) SAT-1 runtime vs. clause count. (b) Identified “hard” and “easy” instances on plot a. (c) Relationship of SAT-2 speedup to SAT-1 clause-to-time ratio. (d) SAT-1 runtime vs. edge density (m/n), highlighting hard vs. easy instances.

When plotting SAT-1 runtime against the number of generated clauses in its CNF representation (Figure 4.4a), we observe two potential linear trends. Based on this visual pattern, we select a threshold of 6×10^5 for the clause-to-time ratio to split the dataset into two groups (Figure 4.4b). The group with slower runtime growth is labeled “easy”, while the group with a significantly higher and varied growth rate is labeled “hard”.

Following established practices in the statistical literature [14], we validate our categorization of data by performing standard tests to showcase the distinct nature of the two resulting groups. Levene’s test ($p < 10^{-190}$) shows a significantly increased variance for the ”hard” group, the Mann-Whitney U test ($p < 10^{-200}$) indicates the two groups have different distributions, and the t-test ($p \approx 0$) shows a statistically significant difference of mean values.

Having established two distinct groups with differing SAT-1 performance profiles, we next examine their relationship to SAT-2 speedup. When SAT-2 speed gain is plotted against the SAT-1 clause-to-time ratio (Figure 4.4c), we observe that all instances exhibiting significant speedup fall below the defined threshold and correspond to the “hard” class. In contrast, instances classified as “easy” show minimal to no performance gain, as indicated by the dotted line representing equal runtimes. These findings suggest that certain graph instances are inherently more difficult for SAT-1, while SAT-2 consistently outperforms on this subset, often by a wide margin.

To explore whether this problem hardness distinction is linked to structural properties of the graph, we visualize SAT-1 runtime versus graph edge density, calculated as m/n (Figure 4.4d). What we found is that a majority of hard instances are concentrated in the $1 \leq m/n \leq 2$ range, implying a potential connection between edge density and problem hardness.

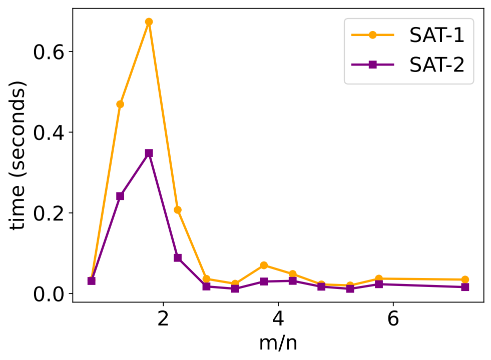


Figure 4.5: Mean SAT-1 and SAT-2 runtime vs. edge density (m/n).

Finally, by comparing average runtimes of SAT-1 and SAT-2 across binned values of m/n (Figure 4.5), we observe that most of the performance gains achieved by SAT-2 occur in the same density range identified earlier. This reinforces the conclusion that SAT-2 encoding significantly reduces average solve time of structurally difficult instances, which is consistent with the observations in Figure 4.4c.

4.2 Scalability Analysis on Grid Graphs

To see whether the performance gains from SAT-2 will carry over to large instances, we benchmark all encodings on grid graphs of increasing size. A grid graph, also referred to as a two-dimensional lattice graph in the literature, is an $m \times n$ planar graph formed as the Cartesian product of two path graphs with m and n vertices [1].

We convert undirected grid graphs into DAGs by orienting their edges in two directions (rightward and downward). This construction yields a family of planar st-graphs—directed graphs with a single source and a single sink—which always admit a 2-page upward book embedding [9]. The example illustrated in Figure 1.1 is, in fact, a directed grid graph of size 3 constructed using this method.

Why select grid DAGs for scalability analysis? In this experiment, our primary goal is to measure how the two encodings perform as instance size increases. To ensure that performance differences reflect encoding efficiency rather than solver-specific behavior, we restrict our analysis to satisfiable instances. In particular, UNSAT performance is often influenced by solver-specific pruning heuristics and conflict learning strategies, which are highly sensitive to variable ordering and other internal mechanisms. In contrast, SAT instances require the solver to construct a complete solution, offering a more meaningful basis for comparison. By fixing as many parameters as possible, we isolate the effect of instance size and provide a fair comparison between the two encodings. Grid DAGs are well-suited for this purpose: they are easy to generate, they scale naturally with size, and—unlike trees or paths—their embeddings are not trivial, placing greater demand on the encoding logic and making them more appropriate for a meaningful scalability study.

All three methods were evaluated on grid graph instances of sizes up to 28×28. The results are presented in Table 4.1 and Figure 4.6. Consistent with the results from the North Graph DAG dataset benchmark, the CP encoding performed poorly, reaching the timeout limit as early as $n = 20$. To solve the largest graph instance (28×28, 784 nodes) SAT-1 took around 48 minutes, whereas SAT-2 took 27.5 minutes. This represents a 40% reduction in runtime over SAT-1, with an overall average speedup across all instances of approximately 30%. These results suggest that SAT-2's advantage persists at scale. Moreover, we can observe in Figure 4.6 that the runtime gap between the two SAT encodings widens as instance size increases. This

Table 4.1: Mean execution times of SAT-1, SAT-2, and CP for grid graphs of size $n \times n$. The second column shows the number of nodes. TO stands for timed-out (> 3600s).

n	Nodes (n^2)	SAT-1 (s)	SAT-2 (s)	CP (s)
2	4	0.0016	0.0030	0.0109
3	9	0.0053	0.0033	0.1007
4	16	0.0290	0.0071	0.6595
5	25	0.0503	0.0116	3.2400
6	36	0.1006	0.0274	7.8882
7	49	0.2406	0.1536	15.0825
8	64	0.5470	0.3536	28.3582
9	81	1.209	0.7173	46.2965
10	100	2.565	1.266	74.8024
11	121	4.601	2.225	108.6383
12	144	5.478	4.491	148.0552
13	169	7.062	6.676	183.4610
14	196	8.929	8.370	334.5505
15	225	11.362	10.457	538.5593
16	256	15.145	13.859	798.9257
17	289	19.617	18.148	1039.6657
18	324	25.703	23.611	1355.6614
19	361	34.164	31.033	2293.8375
20	400	44.323	41.848	TO
21	441	77.498	35.553	TO
22	484	111.974	54.986	TO
23	529	152.683	97.159	TO
24	576	237.621	143.109	TO
25	625	377.786	241.390	TO
26	676	570.635	363.598	TO
27	729	1314.026	978.565	TO
28	784	2874.723	1649.362	TO

trend indicates that SAT-2's performance gains could be even more pronounced for larger instances, validating its potential as a practical and scalable approach.

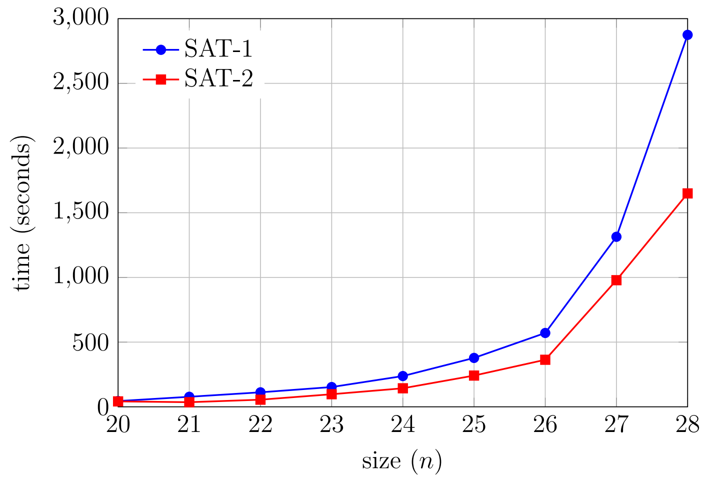


Figure 4.6: Mean runtimes for SAT-1 and SAT-2 across grid graphs of size $n \times n$, where $n \in [20, 28]$.

4.3 Evaluation Conclusion

Empirical evaluations were done on the North dataset and synthetic (grid) graphs. SAT methods outperformed CP by an order of magnitude. When it comes to the difference between SAT approaches, SAT-2 on average outperformed SAT-1, with around 65% of instances showing measurable improvement (over 20%) over SAT-1, and around 5% of instances showing a 16× improvement or more. Notably, a larger proportion (85%) of UNSAT instances benefited from significant runtime speedup (between 20% and 1600%). Less than 1% of instances (almost exclusively SAT ones) showed slight slowdown when switching from SAT-1 to SAT-2, but it was not significant.

Through a combination of empirical observations and statistical analysis, we identified two distinct categories of graph instances—“easy” and “hard.” This classification led to a further insight: the runtime behavior of both SAT encodings appears to correlate with graph density. In particular, hard instances tend to cluster in the $1 \leq m/n \leq 2$ range, where both SAT-1 and SAT-2 show peak runtimes. Moreover, it is within this range that SAT-2 achieves the most significant performance gains over SAT-1.

These findings motivate a deeper investigation into how embeddability correlates with graph density (Chapter 6: Analyzing Upward Book Embeddability of DAGs). To conduct this analysis without structural bias, the North dataset is no longer sufficient. Instead, we propose and evaluate new DAG generation strategies in the following chapters to produce a more representative and unbiased sampling of the DAG space.

Chapter 5

Generating DAGs

In the previous chapter, we hypothesized that the North graph dataset may be biased toward certain structural families. Therefore, to formally and conclusively investigate the relationship between graph structure and problem hardness (solver runtime growth), we must shift toward generating our own test cases—ones that are representative of the entire DAG space without introducing bias. This chapter presents two approaches we used to achieve that.

The first approach is brute-force enumeration of all possible DAGs for a given number of nodes. This method is exhaustive and, by design, produces an unbiased dataset. However, it is limited by two related factors: the exponential storage space required and the exponential time needed to process all instances.

To overcome these limitations, we adopt a hybrid sampling strategy that generates diverse and representative DAGs. This method combines a uniform sampling algorithm proposed by Kuipers–Moffa [33] with a simpler, more easily controlled algorithm of our own. Together, they allow us to produce a sufficient number of samples for any given node count, avoiding heavily undersampled regions that would arise with a purely uniform method.

5.1 Enumerating All Small DAGs

The algorithm which generates all DAGs for n nodes is presented in Algorithm 1. The algorithm initializes a set of nodes $V = \{0, 1, \ldots, n-1\}$ and the set of all possible directed edges E_{all} between distinct nodes. It then iterates over all subsets of E_{all}, constructs a graph for each, checks if it is a DAG (i.e., has no cycles) and collects valid edge sets in a list which is returned. Recognizing that the number of DAGs grows exponentially with n, the algorithm iterates over all subsets of E_{all}, constructs a graph for each subset, and verifies if it is a DAG (i.e. acyclic) using a check such as topological sort. Valid edge sets are returned.

Given the exponential number of edge subsets—reflecting the inherent complexity of enumerating all DAGs—the algorithm adopts a brute-force approach. Each DAG verification requires $O(n^2)$ time, resulting in exponential time complexity overall. Additionally, storing all DAGs demands exponential space, limiting practical use to $n \leq 10$. Nonetheless, our implementation emphasizes simplicity and completeness. By exhaustively exploring edge combinations, it ensures every DAG is generated, making it well-suited for small n or theoretical analysis where clarity and correctness outweigh efficiency considerations.

Algorithm 1 Generate all DAGs with n nodes

```
1: *Initialization* : dags ← []; $V \leftarrow \{0, 1, \ldots, n-1\}$; $E_{\text{all}} \leftarrow \{(u,v) \mid u \neq v, u, v \in V\}$
2: **for all** edge subsets $E'$ of $E_{\text{all}}$ **do**
3:     $G \leftarrow$ CreateGraph$(V, E')$
4:     **If** (IsDAG$(G)$)  dags ← dags $\cup \{E'\}$
5: **end for**
6: **return** dags
```

According to the On-Line Encyclopedia of Integer Sequences (A003024 [41]), the total number of DAGs is 25 for $n = 3$, 543 for $n = 4$, 29,281 for $n = 5$, and 3,781,503 for $n = 6$. In fact, we employ Algorithm 1 to determine their detailed breakdown across different edge counts (m), with the results presented in Tables 5.1, 5.2 and 5.3.

Table 5.1: Number of DAGs with (a) $n = 3$, and (b) $n = 4$ nodes, for different edge counts

(a) $n = 3$

Edges	0	1	2	3
# of DAGs	1	6	12	6

(b) $n = 4$

Edges	0	1	2	3	4	5	6
# of DAGs	1	12	60	152	186	108	24

Table 5.2: Number of DAGs with $n = 5$ nodes for different edge counts

Edges	0	1	2	3	4	5	6	7	8	9	10
# of DAGs	1	20	180	940	3050	6180	7960	6540	3330	960	120

Table 5.3: Number of DAGs with $n = 6$ nodes for different edge counts

Edges	0	1	2	3	4	5	6	7
# of DAGs	1	30	420	3600	20790	83952	240480	496680
Edges	8	9	10	11	12	13	14	15
# of DAGs	750810	838130	691020	416160	178230	51480	9000	720

5.2 Generating Random DAGs via Sampling

When n is large, generating and analyzing all possible DAGs becomes computationally infeasible due to the exponential growth of the DAG instance space. Instead, we sample a subset of DAGs to study their properties, making the approach computationally manageable. It is crucial, however, to ensure that the sampled subset is representative, as an inappropriate sampling method can skew results and misrepresent the underlying properties of the DAG population. For example, a biased sampling method might over-represent certain structures—such as those with more valid topological orderings—leading to inaccurate conclusions about phenomena like satisfiability transitions.

To balance computational efficiency and statistical precision, we adopt a two-tiered strategy that integrates existing methods with our own adaptations into a cohesive framework. First, we propose Algorithm 2, an efficient, approximate topological-order-based algorithm that generates a broad set of DAGs across all edge counts m (ranging from 0 to $n(n-1)/2$) for each n. We then complement this with targeted uniform sampling in the critical phase-transition region using the Kuipers–Moffa method [33]. The combination of approximate sampling for broad coverage and uniform sampling for key areas ensures both efficiency and accuracy in our experimental analysis.

The topological-order method, detailed in Algorithm 2, generates DAGs by shuffling node orders and uniformly selecting m edges from all possible forward edges, producing 100 DAGs per m (or 30 for high n and k). Meanwhile, the Kuipers–Moffa

Algorithm 2 Generate a random DAG of n nodes and m edges

```
Require: $n$, $m$
Ensure: A list of $m$ edges forming a random DAG
1: $order \leftarrow$ Shuffle($[0, 1, \dots, n-1]$)    ▷ Random permutation of nodes
2: $all\_edges \leftarrow \{(order[i], order[j]) \mid 0 \le i < j < n\}$    ▷ All forward edges
3: $chosen\_edges \leftarrow$ UniformSampling($all\_edges, m$)    ▷ Select $m$ edges uniformly
4: return Sort($chosen\_edges$)    ▷ Return sorted edge list
```

method, implemented via the `unifDAG` R package [30], ensures uniform sampling by recursively constructing DAGs through outpoint removal and reverse connection sampling. We specifically use the approximate method for broad exploration across all m, reducing to 30 samples for computationally intensive cases ($n = 20, k \geq 6$), and validate its empirical adequacy against uniform sampling for selected n. We switch to Kuipers–Moffa sampling in the phase-transition region, where satisfiability is about 50%. In this study, we used this sampling approach to generate DAGs for $n \in \{7, 8, \dots, 20\}$, storing them as edge lists for analysis.

Chapter 6

Analyzing Upward Book Embeddability of DAGs

One well-known phenomenon in NP-complete problems is the presence of a phase transition—a sharp threshold where the probability of a property being satisfied drops suddenly. Phase transitions are well-documented in graph theory and constraint satisfaction problems, such as the connectivity threshold in random graphs [20], the k-colorability threshold in graph coloring [2], and the satisfiability threshold in random k-SAT problems [11, 39].

We hypothesize that embeddability is influenced by graph density, with fewer edges leading to fewer potential edge crossings and thus a higher likelihood of finding a valid embedding. This suggests the presence of a phase transition, where embeddability drops sharply as density increases.

As an initial validation, we examined the North graph dataset for $k = 2$ to test whether this phase transition behavior appears. The results supported the hypothesis, motivating a broader and more systematic analysis across a larger and more representative space of DAGs.

Guided by this preliminary result, we formed two questions:

- How does increasing graph density affect kUBE embeddability?
- How does increasing the number of pages affect kUBE embeddability?

To address these questions, we adopt a systematic approach starting with small values of n, where the number of possible DAGs is manageable. For these small DAGs, we enumerate all instances using affordable computational resources. This allows us to analyze embeddability patterns exhaustively and derive initial insights into the effects of varying m and k.

Building on these findings, we extend our study to larger DAGs, where enumeration becomes impractical but one can sample representatively. We investigate how the trends observed in smaller cases scale with increasing n. Our analysis combines empirical observations with theoretical considerations, shedding light on the behavior of kUBE embeddability as graph size and structure evolve.

6.1 Embeddability of North Graphs

For each North Graph DAG, we compute the edge-to-node ratio m/n. We next encode each graph as a SAT instance using SAT-2 and solve it using Kissat. After solving, we aggregate the data by computing the percentage of graphs that are embeddable in two pages for binned values of m/n. This provides an empirical probability distribution of 2UBE embeddability as a function of density m/n.

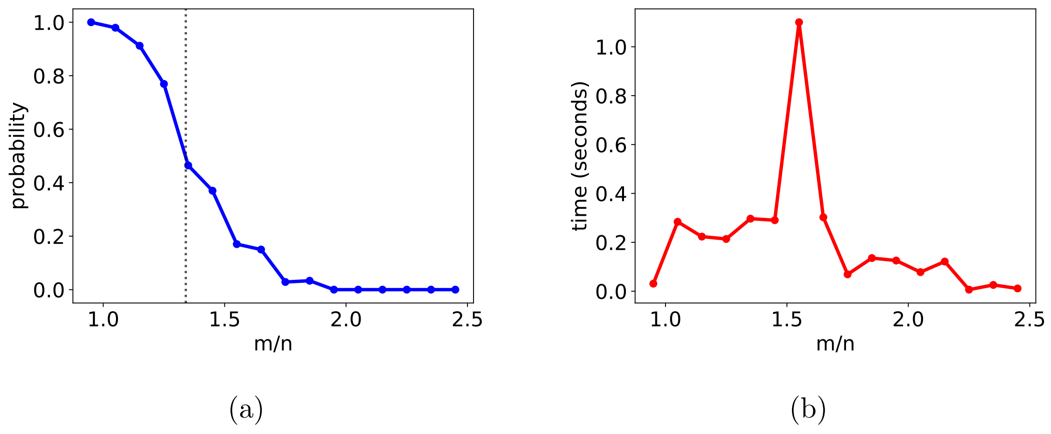


Figure 6.1: On North dataset: (a) Probability of satisfiability of 2UBE vs. m/n (dotted line marks the 50% satisfiable point). (b) Mean SAT-2 runtime vs. m/n.

As shown in Figure 6.1a, a phase transition is clearly observed in the 2UBE embeddability as a function of graph density (m/n). The transition occurs roughly in the range $1.2 \leq m/n \leq 1.5$, where the embeddability probability drops from 0.769

at $m/n \in (1.2, 1.3]$ to 0.370 at $m/n \in (1.4, 1.5]$. Beyond $m/n \approx 1.5$, the probability of embeddability sharply approaches zero, reaching 0% at $m/n > 1.9$.

In Figure 6.1b, we plot the mean SAT-2 runtime against the edge-to-node ratio for $m/n \in [1, 2.5]$, using bin width of 0.1. Although values of m/n extend up to 6, for all instances with $m/n > 2$ average solve time remains below 50ms. In contrast, there is a significant runtime spike around $m/n \approx 1.55$, which coincides with the drop of probability in Figure 6.1a.

These observations suggest a phase transition threshold near $m/n \approx 1.5$ for 2UBE, but a broader investigation across the full DAG space is needed to draw a stronger conclusion.

6.2 Embeddability of Small DAGs

In small cases ($n \leq 6$), all possible DAGs can be exhaustively enumerated. This allows us to conclusively investigate the upward book embeddability of the complete DAG space for these values of n. This examination provides significant insight and valuable perspective on the broader context of larger instances.

We assess the embeddability of all enumerated DAGs for $n = 4, 5, 6$ and different values of k using the SAT-1 approach. For each n, we generate all possible DAGs with Algorithm 1, categorize them by edge count m (as shown in Tables 5.1, 5.2, and 5.3), and test each instance for k-page upward book embeddability with $k = 1$ and $k = 2$. The SAT-1 encoding translates a kUBE problem into a Boolean satisfiability problem, which is then solved using the Kissat solver. For each combination of n, m, and k, we compute the percentage of satisfiable DAGs by dividing the number of SAT outcomes (indicating embeddability) by the total number of DAGs at that m, providing the data points to plot the curves in Figure 6.2.

Figure 6.2 illustrates the percentage of DAGs that can be satisfied using k pages as a function of the total number of edges m, for varying numbers of nodes n. The x-axis represents m, ranging from 0 to 15, while the y-axis shows the percentage of satisfiable DAGs, from 0% to 100%. Note that when $n = 6$, the maximal possible m value is 15; when $n = 5$, the maximal m value is 10; and when $n = 4$, the maximal m is 6. The data is categorized by n and k: filled circles denote $n = 6$, crosses denote $n = 5$, and triangles denote $n = 4$, with blue representing $k = 1$ and orange representing $k = 2$.

For all configurations, we have the following observations:

1. The percentage starts at 100% when $m = 0$, as a DAG with no edges is trivially embeddable. As m increases, the percentage decreases steadily and smoothly, reflecting the growing complexity of the DAG. For any specific curve with a fixed n, this indicates that embeddability decreases as the graph density (defined as m/n) increases.

2. For a fixed n, the percentage of satisfiable DAGs is consistently higher for $k = 2$ (orange) than for $k = 1$ (blue). For instance, the solid orange line ($n = 6, k = 2$) remains above the solid blue line ($n = 6, k = 1$) across all m, indicating that using two pages provides greater flexibility to satisfy the DAG.

3. When n is fixed and k is repeatedly increased (i.e., the total number of pages allowed for the embedding), there exists a threshold k value at which all DAGs of size n are embeddable. For small DAGs (where $n = 4, 5, 6$), these threshold k values are obtained using the SAT-1 encoding and the Kissat SAT solver, and are presented in Table 6.1.

Table 6.1: Minimal number of pages k required for all size-n DAGs to be embeddable ($n = 4, 5, 6$).

DAG size (n)	4	5	6
Minimal number of pages required (k)	2	3	3

The table explains why there are only five curves in Figure 6.2. In addition, we also manually proved that when $k \geq 3$, all $n = 6$ DAGs are embeddable in the following theorem.

Theorem 6.1. *All $n = 6$ DAGs can be upward book-embedded using 3 pages.*

Proof. We first note that all DAGs can be topologically sorted. That is, given a DAG G, the nodes in G can be arranged from left to right, and all edges in G can be arranged to point from left to right. In Figure 2.1, for example, the left subfigure

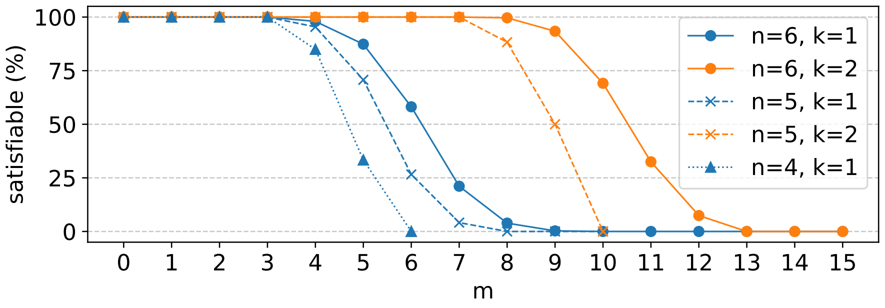


Figure 6.2: Percentage of satisfiable DAGs as a function of the number of edges m, for different numbers of nodes n and pages k. Filled circles represent $n = 6$, crosses represent $n = 5$, and triangles represent $n = 4$. Blue denotes $k = 1$, and orange denotes $k = 2$.

shows a DAG, and the right subfigure shows the same but topologically sorted graph, with vertices ordered from v_0 to v_5.

A greedy approach that works for almost all cases: We present an algorithm where all edges leaving v_0 are placed on the first page (denoted as the black page), all edges leaving v_1 go to the blue page, and all edges leaving v_2 are placed on the yellow page. The edge from v_4 to v_5 can be assigned to any of the three pages without crossing any existing edges on that page.

One exception: The only potential issue is the edge from v_3 to v_5, which, if placed on any of the three pages, may cross with existing edges. However, for complete cases (where each node has an edge to every node to its right), we are able to find a 3-upward book embedding, as shown in the right subfigure of Figure 2.1. The edge of any $n = 6$ DAG contains fewer edges than the "maximal DAG", so the edge from v_3 to v_5 cannot cause any issues for other DAGs either.

We conclude that all $n = 6$ DAGs can be upward book-embedded using three pages. □

6.3 Bounds on Minimal k for Embeddability

Theorem 6.1 can be utilized to establish general cases, as shown below in Theorem 6.2. Specifically in the theorem, we prove an upper bound on the minimal number of pages required to upward-embed a DAG with n vertices.

Theorem 6.2. *All DAGs of size $n \geq 6$ can be upward book-embedded using $(n-3)$ pages.*

Proof. We employ mathematical induction for the proof on the number n.

Base Case: The base case, where $n = 6$, is established in Theorem 6.1, ensuring that any DAG with 6 vertices can be upward book embedded using $6 - 3 = 3$ pages.

Induction Step: Let us assume that for some $k \geq 6$, any DAG with k vertices can be upward book embedded using $k - 3$ pages, and consider a DAG G with $k + 1$ vertices. We must show that G can be upward book embedded using: $(k+1)-3 = k-2$ pages.

Since G is a DAG, it has at least one sink vertex v (a vertex with out-degree 0). Define $G' = G - \{v\}$, obtained by removing v and all edges incident to it from G. Thus, G' has k vertices. By the induction hypothesis, G' admits an upward book embedding with spine order $u_1, u_2, \ldots, u_k$ and edges partitioned into $k-3$ pages, each containing non-crossing upward edges.

Construct an embedding for G by appending v to the spine: $u_1, u_2, \ldots, u_k, v$. This ordering preserves topological consistency, as v has no outgoing edges, and for any edge $u_i \to v$, u_i precedes v on the spine. Assign all edges incident to v, of the form $u_i \to v$, to a single new page. For any two edges $u_i \to v$ and $u_j \to v$ with $i < j$, the spine order $u_i < u_j < v$ ensures their upward arcs converge to v without crossing.

The embedding of G thus uses $k - 3$ pages for the edges of G', which remain non-crossing, plus one additional page for all edges $u_i \to v$, also non-crossing. Total pages: $(k - 3) + 1 = k - 2$. Hence, G is upward book embedded in $k - 2$ pages, and the statement holds for $n = k + 1$.

By induction, starting from $n = 6$ and extending to all $n > 6$, every DAG with n vertices can be upward book embedded using $n - 3$ pages. □

Note that the statement does not hold as n decreases below the base case. For instance, not all DAGs with $n = 5$ vertices can be upward book embedded in $5-3 = 2$ pages, as demonstrated in Figure 6.2.

Subsequently, we explore the lower bound on the minimal number of pages required for upward book embedding. For each n from 2 to 20, we select the maximal

DAG (i.e., a DAG of size n with the maximum possible edges) and incrementally reduce the allowed number of pages, using a SAT solver to test embeddability at each page count. Accordingly, we identify the lower bound on the minimal pages necessary for each n. Experiments are conducted with increasing n values up to 20, and the resulting lower bounds are reported in Table 6.2.

Table 6.2: Lower bound on pages required for upward book embedding of DAGs with n vertices.

n	2-3	4	5-6	7-8	9-10	11-12	13-14	15-16	17-18	19-20
number of pages	1	2	3	4	5	6	7	8	9	10

From the initial values of n, a pattern emerges suggesting that the lower bound on the minimal number of pages required to embed a DAG with n vertices is $\lceil n/2 \rceil$, for all $n \geq 4$. However, whether this pattern holds has not yet been formally proved or disproved. We therefore propose the following conjecture:

Conjecture 6.3. *For all $n \geq 4$, to guarantee the upward book embeddability of a DAG with n vertices, $\lceil n/2 \rceil$ pages are required.*

A Parallel Between BE and kUBE: Our investigation suggests a possible alignment in the page requirements of the classical book embedding (BE) problem for undirected graphs and the upward book embedding (kUBE) problem for directed acyclic graphs (DAGs). It is a well-established result that any undirected graph with n vertices has book thickness at most $\lceil n/2 \rceil$, with equality for complete graphs. In our empirical analysis of kUBE instances, we observe that the same upper bound may suffice for embedding all DAGs of size n. Although kUBE imposes stricter conditions—most notably the requirement that all edges respect a topological ordering—the number of pages required does not appear to increase beyond that of the undirected case, at least for the range of n we considered.

In conclusion, this section investigates the number of pages required to embed any DAG with n vertices. We proved that, for all $n \geq 6$, this number is at most $n - 3$, while conjecturing that it is at least $\lceil n/2 \rceil$ for all $n \geq 4$.

6.4 Embeddability of Random DAGs

When n only increases slightly, enumerating all DAGs quickly becomes impractical. For instance, when $n = 10$, the total number of DAGs is approximately 4.18×10^{18} (A003024 [41]). To explore the embeddability properties of larger DAGs, we must rely on the sampling technique introduced in the previous chapter. We used this method to obtain 4,600 DAGs with $n = 10$, 10,600 with $n = 15$, and 19,100 with $n = 20$, the latter of which was downsampled to 5,730 instances for $k \geq 6$ to reduce total execution time.

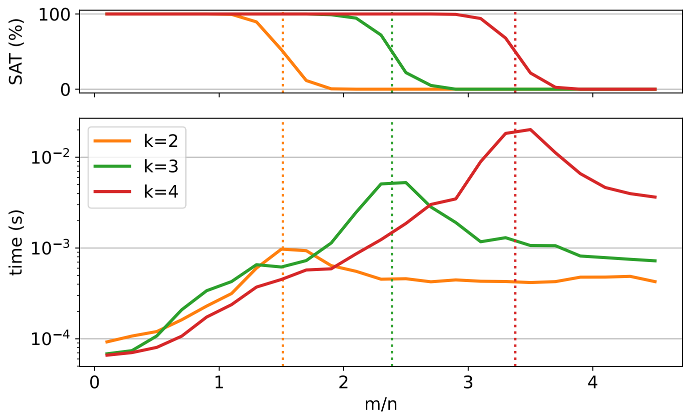
We then benchmarked SAT solver runtimes across increasing values of k. The experimental setup covered the following configurations, with the number of edges m ranging from 0 to its maximum in all cases: (a) $n = 10$, with $k = 2, 3, 4$; (b) $n = 15$, with $k = 2$ to 7; and (c) $n = 20$, with $k = 2$ to 7.

For $n = 20$, the minimum number of pages required to embed all DAGs is 10 (provided that Conjecture 6.3 is valid). Thus, satisfiability curves with respect to edge density could, in principle, be computed up to $k = 9$. However, due to the prohibitively high computational cost at these values, results for $k = 8$ and 9 were not included in the current paper.

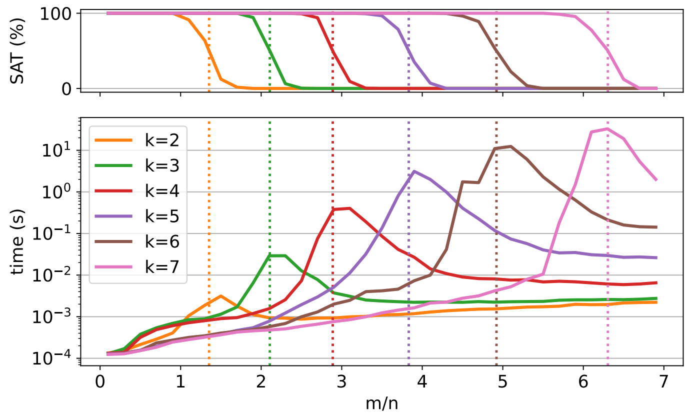
For each value of n, we analyzed how increasing the edge count m, and thus the graph density m/n, impacts both the embeddability of DAGs in kUBE and the run-time of the SAT solver. Results are presented in Figure 6.3, where each subfigure corresponds to a fixed n and consists of two aligned plots: the top shows the satisfiability rate as a function of normalized edge density m/n, and the bottom shows the mean run time of the SAT solver. In both plots, the x-axis is divided into bins of width 0.2, and the mean is calculated per bin.

Recall that (Section 6.2) when DAGs are small and full enumeration was feasible, we identified two key patterns: 1) embeddability starts at 100% with $m = 0$, dropping smoothly as m increases due to the increasing density of the graph (m/n), and 2) higher k values, such as $k = 2$ versus $k = 1$, consistently improve embeddability by offering greater flexibility. Interestingly, these patterns hold in our experiments with sampled larger DAGs at $n = 10$, 15, and 20, where we observe a comparable gradual decline in embeddability as edge density grows, alongside a significant improvement in embedding success with larger k values, closely mirroring the behavior of their

(a) $n = 10$



(b) $n = 15$



(c) $n = 20$

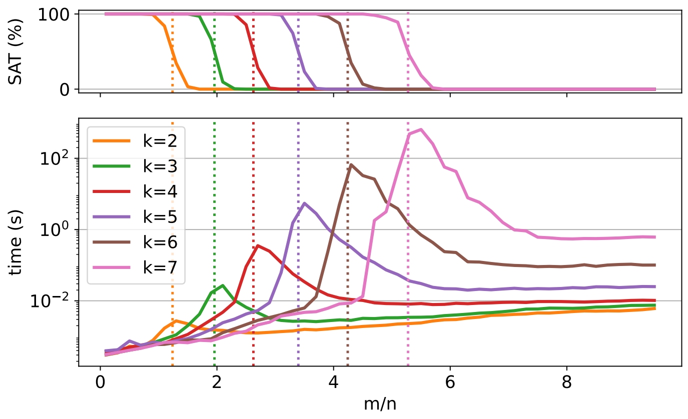


Figure 6.3: Embeddability and computational cost for sampled DAGs across $n = 10$, 15, and 20. Each subfigure consists of: (Top) Fraction of embeddable DAGs versus graph density m/n for varying k. (Bottom) Mean computation time versus graph density m/n. Vertical dotted lines mark the 50% embeddability threshold on both subplots.

smaller, fully enumerated counterparts.

The bottom plots of the three subfigures, which depict mean computation time, exhibit a consistent pattern across all configurations of n and k:

1. For each (n, k) combination, the peak runtime is aligned with the 50% embeddability threshold (indicated by the vertical dotted line). This suggests that computational complexity is maximized when the proportion of SAT to UNSAT instances is approximately balanced.

2. For larger values of k, the peak runtime increases substantially compared to smaller k. As the y-axis uses a logarithmic scale, we can observe that the peak runtime for k is typically one order of magnitude higher than that for $k - 1$.

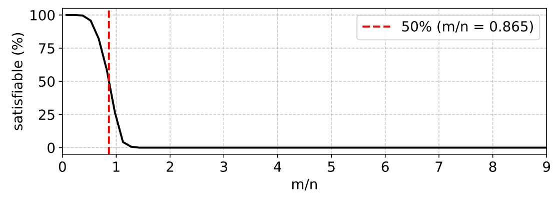
3. Prior to the peak, the runtime differences across k values are relatively small. Notably, there is consistently a range of m/n values where higher k instances are solved faster than those with lower k. This range tends to occur near the peak runtime for the lower k, likely because at those m/n values, the higher k instances are nearly always embeddable—possibly even trivially so.

4. After the peak, as m/n continues to increase, instances become predominantly non-embeddable. In this regime, higher k instances consistently take longer to solve. This is likely due to the increased number of SAT variables and constraints introduced at higher k, which makes proving unsatisfiability more computationally demanding.

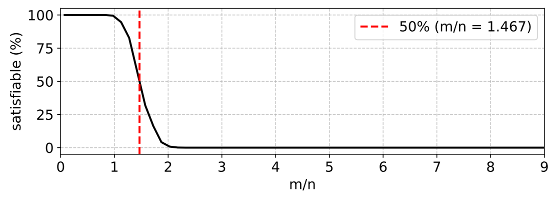
6.5 Phase Transition

We previously observed phase transition behavior and runtime peaks for individual (n, k) pairs. To study how these transitions behave across different graph sizes, we combined sampled DAGs with n between 7 and 20, excluding smaller n due to the high relative variability of runtimes caused by their extremely short execution times, which made precise measurement infeasible. This aggregation resulted in a dataset of 45,000 DAGs, enabling us to examine whether embeddability exhibits a sharp threshold that is largely independent of n.

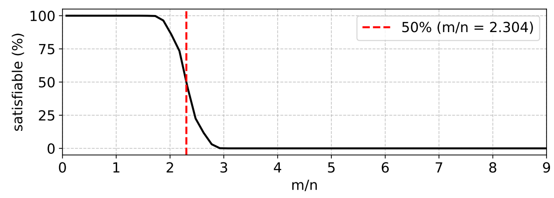
Analyzing this combined dataset, we observe that embeddability drops abruptly from nearly 100% to near 0% as the edge density m/n crosses a critical threshold. We characterize this behavior by using m/n as the control parameter and plotting embeddability curves across varying k. These transitions are clearly visible in Figures 6.4 (a–f), and corresponding peaks in solver runtime are observed in Figures 6.5 (a–f), particularly for $k \geq 2$.



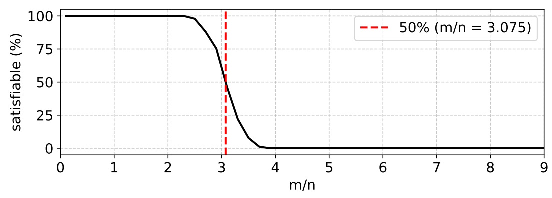
(a) $k = 1$



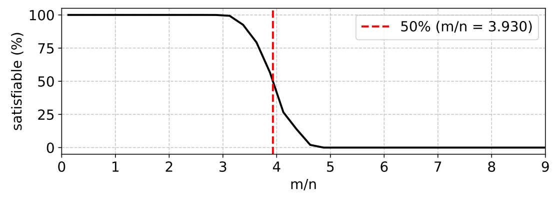
(b) $k = 2$



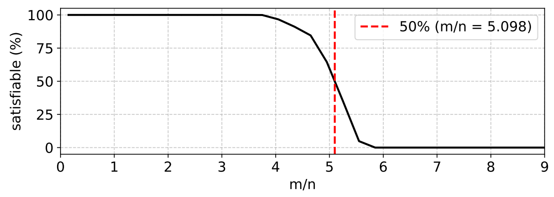
(c) $k = 3$



(d) $k = 4$



(e) $k = 5$



(f) $k = 6$

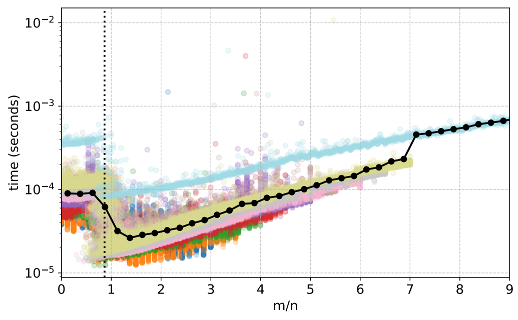
Figure 6.4: Percentage of DAGs embeddable in a k-page upward book embedding as a function of graph density (m/n). Vertical, red and dashed lines mark the 50% embeddability thresholds.

Figure 6.4 in particular shows the percentage of embeddable DAGs decreasing rapidly at critical m/n values: 0.865 for $k = 1$, 1.467 for $k = 2$, 2.304 for $k = 3$, 3.075 for $k = 4$, 3.930 for $k = 5$, and 5.098 for $k = 6$, as marked by the 50% embeddability thresholds. These thresholds reveal a clear relationship: the critical m/n at which the phase transition occurs increases near-linearly with k, suggesting that each additional page allows the DAG to sustain a higher density before embeddability collapses, a trend we explore further to quantify its implications across varying n.

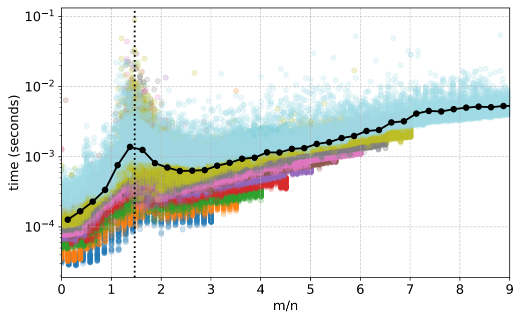
Figure 6.5 illustrates runtime behavior across $k = 1$ to 6 in subfigures (a–f), where each subfigure employs a scatter plot to depict runtime (in seconds, on a logarithmic scale) versus m/n for DAGs with $n = 7$ to 20, using distinct colors to differentiate node sizes and highlight trends across graph scales. For $k = 2$ to 6, the scatter plots reveal a pronounced peak in runtime that aligns closely with the critical m/n values from Figure 4—1.467 for $k = 2$, 2.304 for $k = 3$, 3.075 for $k = 4$, 3.930 for $k = 5$, and 5.098 for $k = 6$—where the phase transition occurs, underscoring the solver's peak complexity during the embeddability shift. Notably, the right side of each peak exhibits higher runtimes, as the increasing m/n corresponds to a larger number of edges in the graph, thereby requiring more time to verify embeddability.

The runtime behavior for $k = 1$ in Figure 6.5 differs markedly from cases where $k \geq 2$, due to differences in computational complexity. For $k \geq 2$, the NP-completeness of kUBE leads to runtime peaks at the phase transition (e.g., $m/n = 1.467$ for $k = 2$, increasing to 5.098 for $k = 6$), as shown in subfigures (b–f). In contrast, for $k = 1$, its polynomial-time solvability results in a gradual runtime increase without a peak at $m/n = 0.865$ in subfigure (a). This contrast underscores that when only one page is involved, a simpler decision process suffices, avoiding the exponential complexity spike that arises with multiple pages.

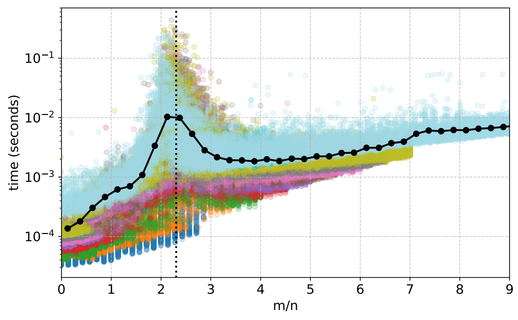
This behavior mirrors the distinction between 2SAT and 3SAT. While 3SAT is NP-complete [31] and exhibits a pronounced computational peak near its phase transition [10, 39], 2SAT is solvable in linear time [32, 21] (e.g., via the implication graph and strongly connected components). As the clause-to-variable ratio in 2SAT grows, contradictions emerge more quickly, allowing for early solver termination. The lack of a runtime peak for the $k = 1$ case similarly reflects its polynomial solvability: contradictions or valid embeddings become apparent and are resolved efficiently in denser instances.



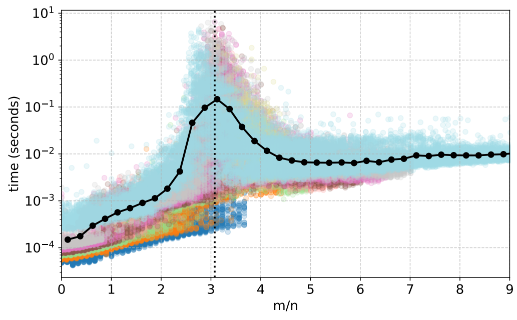
(a) k = 1



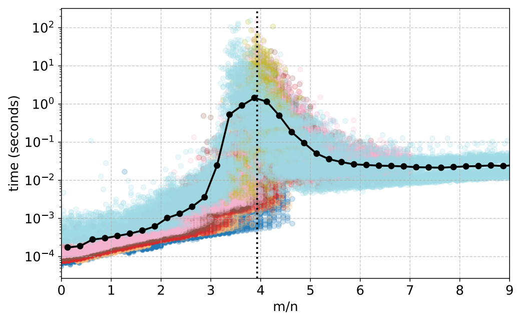
(b) k = 2



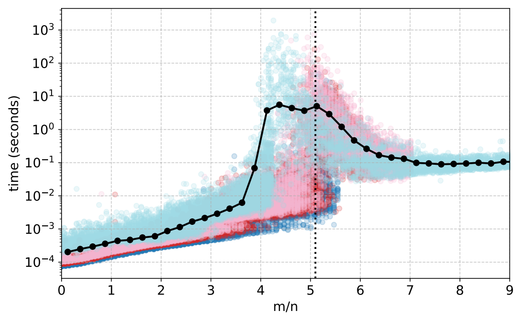
(c) k = 3



(d) k = 4



(e) k = 5



(f) k = 6

● n = 7 ● n = 9 ● n = 11 ● n = 13 ● n = 15
● n = 8 ● n = 10 ● n = 12 ● n = 14 ● n = 20

Figure 6.5: Run-time versus m/n for $k \in \{1, 2, \ldots, 6\}$. Each subplot presents a scatter plot of all combined instances for a fixed k, with points colored according to the value of n. The black line denotes the mean run-time per bin of m/n, and the vertical dotted line indicates the 50% embedding threshold.

6.6 Modeling the Phase Transition Threshold

While the previous section identified approximate phase transition points for each fixed k by aggregating across DAG sizes, we now model the threshold more precisely as a function of both n and k, capturing how graph size interacts with flexibility in page count.

To help us identify patterns, we plot the exact 50% embeddability thresholds for each pair of n and k in Figure 6.6. A clear trend can be observed: the threshold is consistently higher for larger k values, and it decreases gradually as n increases.

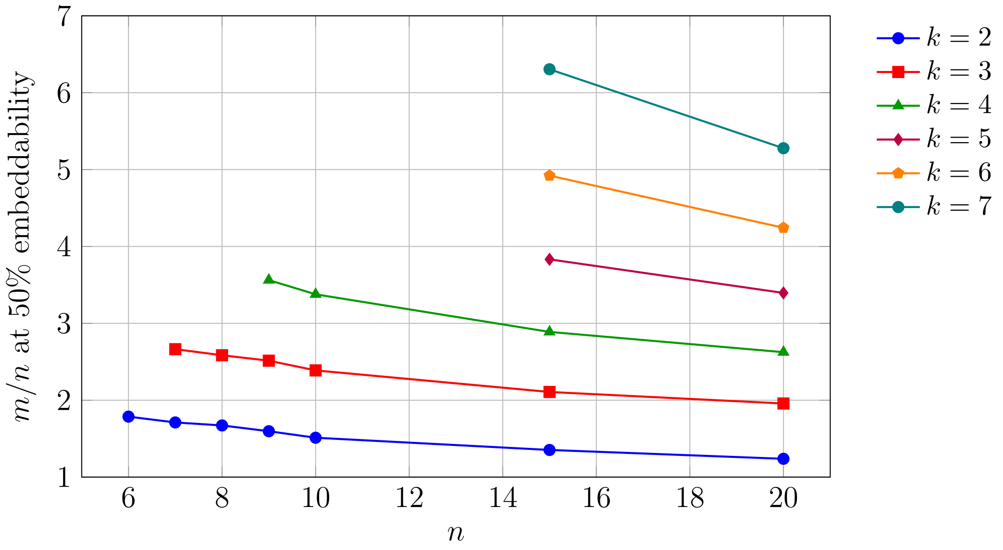


Figure 6.6: Values of m/n at 50% embeddability threshold across various DAG sizes n and number of pages k.

Inspired by the use of power-law scaling to model phase transitions in physics [19], we attempt to fit the data using the following model:

$$\frac{m}{n} = C \cdot k^{\beta} \cdot n^{\gamma} \tag{6.1}$$

After fitting the model using nonlinear least squares regression, we obtain the following values for C, β, and γ:

$$\frac{m}{n} \approx 1.5 \cdot k^{1.26} \cdot n^{-0.39} \tag{6.2}$$

To measure the effectiveness of our model in predicting observed outcomes, we calculate the coefficient of determination (R^2) [16]. The model given in Equation 6.2 achieves a R^2 of 0.993, indicating that it explains 99.3% of the variance in threshold values. The quality of the fit can be observed in Figure 6.7.

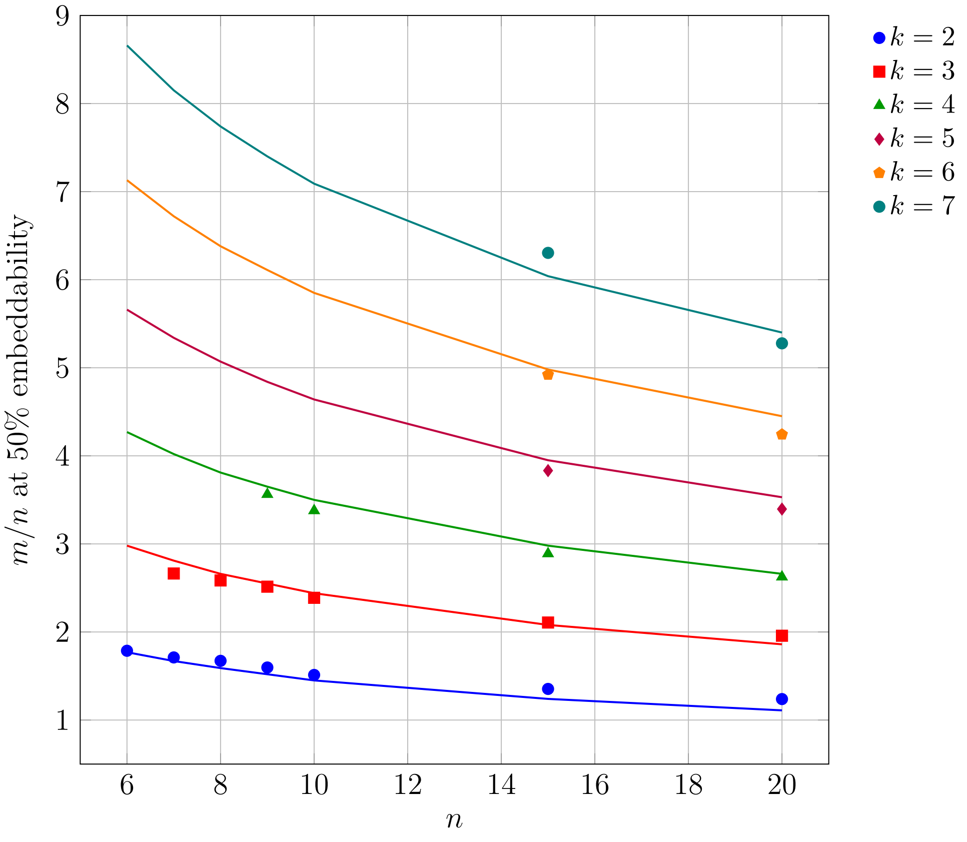


Figure 6.7: Markers represent m/n embeddability thresholds from empirical analysis. Lines show predicted values from the fitted model presented in Equation 6.2.

Chapter 7

Conclusion

This thesis addressed the *k-Page Upward Book Embedding* (kUBE) problem, which is a variant of the well-studied book embedding problem applied to directed graphs and requiring a topological ordering of nodes. We proposed, implemented, and empirically evaluated practical solution methods based on Boolean Satisfiability (SAT) and Constraint Programming (CP). The work was motivated by both the theoretical interest in upward book embeddings and the lack of general-purpose techniques capable of handling arbitrary DAGs. This is an important gap given the prevalence of DAGs in fields such as scheduling, data flow analysis, and network visualization.

We presented three encoding strategies:

- **SAT-1**, an adaptation of an existing kBE SAT encoding to the upward case;
- **SAT-2**, a more compact SAT encoding tailored specifically to 2UBE;
- A straightforward **CP model** utilizing integer variables for representing vertex positioning.

All approaches were formally proven sound and complete, ensuring that every SAT or UNSAT result corresponds to the embeddability or non-embeddability of the original graph.

7.1 Empirical Evaluation Results

The empirical evaluation focused on two datasets: the North dataset and synthetically generated grid graphs. Across both benchmarks, SAT-based methods consistently outperformed the CP-based approach by a substantial margin. This highlights that SAT solvers, particularly with the refined SAT-2 encoding, are better suited for the kUBE problem under the current experimental setup.

Among the SAT approaches, SAT-2 demonstrated a clear advantage over SAT-1. Instances that were difficult to solve for SAT-1 were observed to cluster around a specific range of graph densities. A closer examination revealed that instance hardness correlates strongly with edge density: for two-page upward book embedding, instances with an edge-to-vertex ratio between one and two, and particularly around 1.5, consistently posed the greatest computational challenge. This density range was later identified as corresponding to the phase transition region, where the probability of satisfiability is near fifty percent and computational difficulty peaks. Notably, the majority of instances that exhibited significant performance gains under SAT-2 fall within this region. These observations suggest that improvements in encoding efficiency yield their greatest benefits near the phase transition, where instances are significantly harder to solve, while instances far from this threshold show minimal impact from such optimizations.

7.2 Embeddability Analysis Results

Our analysis demonstrates a strong relationship between graph density and embeddability in k-page upward book embeddings, supported by exhaustive enumeration, random sampling, and evaluation of real-world DAGs.

Embeddability becomes less likely as edge density (m/n) increases and as the number of vertices n grows. In contrast, increasing the number of pages k improves embeddability substantially. In other words, k is positively correlated with embeddability, while m, n, and m/n are negatively correlated. This relationship holds consistently across all datasets and experimental settings.

Based on these trends, we derived a power-law model fitted to the empirical data, which predicts the critical edge density at which the phase transition occurs. In other words, it estimates the m/n value where approximately half of all DAGs are embeddable and half are not. The model achieved an R^2 of 0.993 when fitted to the observed data, indicating an excellent match between the predicted and empirical phase transition thresholds. It provides a practical tool for approximating embeddability thresholds without requiring full enumeration or exhaustive sampling.

An additional insight arises when examining solver runtimes. For $k \geq 2$, runtimes exhibit a clear peak near the phase transition. This indicates increased computational difficulty at critical densities. However, for $k = 1$, no such runtime peak appears. Runtimes increase gradually with density instead. This distinction mirrors the behavior observed in 2SAT and 3SAT. 2SAT exhibits a satisfiability phase transition but no computational hardness peak, consistent with its polynomial-time solvability. 3SAT exhibits both a phase transition and a hardness peak, consistent with its NP-completeness. This parallel suggests that observing whether a runtime peak occurs near the phase transition could serve as a heuristic for predicting whether a problem is in P or NP. However, it does not constitute a formal proof of membership in either class.

Finally, theoretical analysis complements these empirical findings. We proved an upper bound of $k \leq n - 3$ for guaranteeing upward book embeddability. Empirical results suggest that $\lceil n/2 \rceil$ pages suffice for all tested DAGs up to $n = 20$. Although upward book embeddings impose stricter conditions compared to classical book embeddings of undirected graphs, our findings indicate that the number of required pages remains comparable, at least within the tested range. This motivates Conjecture 9, proposing that the minimal number of pages needed for upward book embeddability grows predictably with the number of vertices.

7.3 Future Work

Several promising directions remain for future research. While this thesis focused on developing and evaluating SAT and CP approaches for 2-page upward book embedding, many natural extensions and open questions arise. Future work could explore scaling the empirical analysis and investigating unresolved complexity classifications. Each of these directions offers opportunities to deepen understanding of the problem space and to advance the study of upward book embeddings.

Analysis on Scaled-Up Dataset

Extending the experiments to larger DAGs (e.g., $n > 20$) would allow a more thorough validation of the proposed phase transition model and provide deeper insights into solver performance at scale. With additional data for new (n, k) combinations, we could test whether the observed power-law relationship for the phase transition threshold continues to hold, and whether the phase transition behavior maintains the same progression relative to increasing k and n, or exhibits new phenomena. We expect the trends to persist, but empirical validation is necessary to confirm these hypotheses.

Scaling up would also allow us to investigate solver behavior under more extreme conditions. In particular, it remains an open question whether the performance gap between SAT-2 and SAT-1 continues to widen as n increases. Larger instances may reveal new solver bottlenecks or shifts in relative efficiency that are not visible at the scales studied so far.

In the current work, this direction was not pursued due to practical limitations. Available hardware (an average laptop) and limited project time made it infeasible to conduct large-scale experiments. To fully pursue this direction, future work could leverage high-performance servers, parallelize the processing of different instances, and extend the sampling over a longer period. The same sampling strategy used here could be applied to larger spaces.

Tightening Theoretical Bounds

In this thesis, we established a general upper bound on the number of pages required for an upward book embedding (kUBE) of a DAG with n vertices: $k \leq n - 3$ for all $n \geq 6$. We also proposed Conjecture 6.3, which suggests that $k \geq \lceil n/2 \rceil$ pages are necessary for upward book embeddability for all $n \geq 4$.

This creates a gap between the proven upper bound and the conjectured lower bound. Tightening these bounds, either by improving the current upper bound or by proving Conjecture 6.3, remains a key theoretical challenge.

In the context of standard book embeddings (kBE) for undirected graphs, it is known that the book thickness of the complete graph K_n is exactly $\lceil n/2 \rceil$. This sharp result motivates the search for similarly tight bounds in the upward setting. Resolving the general bounds on the page number would represent a significant theoretical advancement.

Runtime as Complexity Indicator

The observed runtime behavior, specifically the absence of a peak for $k = 1$ and the presence of sharp peaks for $k \geq 2$, suggests that empirical runtime profiles could serve as indicators of computational hardness. This observation raises the possibility of using runtime analysis to assist in classifying the complexity of a problem without a complete theoretical understanding.

In particular, partitioned variants such as 2UPBE and 3UMPBE, whose computational complexity remains unresolved, could be investigated through their phase transition behavior. However, the broader idea is not limited to book embedding problems. It could extend to other combinatorial and graph problems, where the presence of a sharp runtime peak near a phase transition may indicate NP-completeness, while a smooth transition may suggest polynomial-time solvability.

This direction was not pursued in the present work because studying partitioned variants would require different encoding strategies and was beyond the scope of this thesis. Nonetheless, using empirical runtime behavior to assist in empirical complexity classification remains an interesting and promising research direction. It may aid theoretical efforts by providing early evidence of problem complexity, guiding where to attempt formal classifications, and contributing to a deeper understanding of phase transitions in combinatorial optimization.

References

[1] BD Acharya and MK Gill. On the index of gracefulness of a graph and the gracefulness of two-dimensional square lattice graphs. *Indian J. Math*, 23(81-94):14, 1981.

[2] Dimitris Achlioptas and Ehud Friedgut. A sharp threshold for k-colorability. *Random Structures & Algorithms*, 14(1):63–70, 1999.

[3] Hugo A Akitaya, Erik D Demaine, Adam Hesterberg, and Quanquan C Liu. Upward partitioned book embeddings. In *International Symposium on Graph Drawing and Network Visualization*, pages 210–223. Springer, 2017.

[4] Patrizio Angelini, Giordano Da Lozzo, and Daniel Neuwirth. Advancements on sefe and partitioned book embedding problems. *Theoretical Computer Science*, 575:71–89, 2015.

[5] Patrizio Angelini, Giuseppe Di Battista, Fabrizio Frati, Maurizio Patrignani, and Ignaz Rutter. Testing the simultaneous embeddability of two graphs whose intersection is a biconnected or a connected graph. *Journal of Discrete Algorithms*, 14:150–172, 2012.

[6] Michael A. Bekos, Giordano Da Lozzo, Fabrizio Frati, Martin Gronemann, Tamara Mchedlidze, and Chrysanthi N. Raftopoulou. Recognizing DAGs with page-number 2 is NP-complete. *Theoretical Computer Science*, 946:113689, 2023.

[7] Michael A Bekos, Michael Kaufmann, and Christian Zielke. The book embedding problem from a SAT-solving perspective. In *Graph Drawing and Network Visualization: 23rd International Symposium, GD 2015, Los Angeles, CA, USA,*

September 24-26, 2015, Revised Selected Papers 23, pages 125–138. Springer, 2015.

[8] Armin Biere, Tobias Faller, Katalin Fazekas, Mathias Fleury, Nils Froleyks, and Florian Pollitt. CaDiCal, Gimsatul, IsaSAT and Kissat entering the SAT competition 2024. *SAT Competition 2024*, page 8, 2024.

[9] Carla Binucci, Giordano Da Lozzo, Emilio Di Giacomo, Walter Didimo, Tamara Mchedlidze, and Maurizio Patrignani. Upward book embeddability of st-graphs: Complexity and algorithms. *Algorithmica*, 85(12):3521–3571, 2023.

[10] Peter Cheeseman, Brian Kanefsky, and William M. Taylor. Where the really hard problems are. In *Proceedings of the 12th IJCAI*, pages 331–337, 1991.

[11] Peter C Cheeseman, Bob Kanefsky, William M Taylor, et al. Where the really hard problems are. In *Ijcai*, volume 91, pages 331–337, 1991.

[12] Markus Chimani, Carsten Gutwenger, Petra Mutzel, and Hoi-Ming Wong. Layer-free upward crossing minimization. *Journal of Experimental Algorithmics (JEA)*, 15:2–1, 2010.

[13] Fan RK Chung, Frank Thomson Leighton, and Arnold L Rosenberg. Embedding graphs in books: a layout problem with applications to VLSI design. *SIAM Journal on Algebraic Discrete Methods*, 8(1):33–58, 1987.

[14] William Jay Conover. *Practical nonparametric statistics.* john wiley & sons, 1999.

[15] Stephen A Cook. The complexity of theorem-proving procedures (1971). 2021.

[16] Alessandro Di Bucchianico. Coefficient of determination (r 2). *Encyclopedia of statistics in quality and reliability*, 2008.

[17] Josep Díaz, Jordi Petit, and Maria Serna. A survey of graph layout problems. *ACM Computing Surveys (CSUR)*, 34(3):313–356, 2002.

[18] Vida Dujmović and David R Wood. Stacks, queues and tracks: Layouts of graph subdivisions. *Discrete Mathematics & Theoretical Computer Science*, 7, 2005.

[19] DG Enzer, MM Schauer, JJ Gomez, MS Gulley, MH Holzscheiter, PG Kwiat, SK Lamoreaux, CG Peterson, VD Sandberg, D Tupa, et al. Observation of power-law scaling for phase transitions in linear trapped ion crystals. *Physical review letters*, 85(12):2466, 2000.

[20] Paul Erdős and Alfréd Rényi. On the evolution of random graphs. *Publications of the Mathematical Institute of the Hungarian Academy of Sciences*, 5:17–61, 1960.

[21] Shimon Even, Alon Itai, and Adi Shamir. On the complexity of timetable and multi-commodity flow problems. *SIAM Journal on Computing*, 5(4):691–703, 1976. `doi:10.1137/0205048`.

[22] Michel Gangnet and Burton Rosenberg. Constraint programming and graph algorithms. *Annals of Mathematics and Artificial Intelligence*, 8:271–284, 1993.

[23] Robert Ganian, Haiko Mueller, Sebastian Ordyniak, Giacomo Paesani, and Mateusz Rychlicki. A tight subexponential-time algorithm for two-page book embedding. *arXiv preprint arXiv:2404.14087*, 2024.

[24] Graph Drawing Community. The International Symposium on Graph Drawing and Network Visualization. Accessed: 2025-02-20. URL: `http://www.graphdrawing.org`.

[25] Stefano Gualandi and Federico Malucelli. Exact solution of graph coloring problems via constraint programming and column generation. *INFORMS Journal on Computing*, 24(1):81–100, 2012.

[26] Lenwood S Heath, Frank Thomson Leighton, and Arnold L Rosenberg. Comparing queues and stacks as machines for laying out graphs. *SIAM journal on discrete mathematics*, 5(3):398–412, 1992.

[27] Lenwood S Heath and Sriram V Pemmaraju. Stack and queue layouts of directed acyclic graphs: Part ii. *SIAM Journal on Computing*, 28(5):1588–1626, 1999.

[28] Seok-Hee Hong and Hiroshi Nagamochi. Simpler algorithms for testing two-page book embedding of partitioned graphs. *Theoretical Computer Science*, 725:79–98, 2018.

[29] Rustem Kakimov and Xing Tan. SAT for upward book embedding: An empirical study. In *Proceedings of the 38th Canadian Conference on Artificial Intelligence (Canadian AI 2025)*, 2025.

[30] Markus Kalisch. *unifDAG: Uniform sampling of directed acyclic graphs*, 2024. R package version 1.0.4. URL: `https://CRAN.R-project.org/package=unifDAG`.

[31] Richard M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, pages 85–103. Springer, 1972.

[32] Melven R. Krom. The decision problem for a class of first-order formulas in which all disjunctions are binary. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 13(1–2):15–20, 1967.

[33] Jack Kuipers and Giusi Moffa. Uniform random generation of large acyclic digraphs. *Statistics and Computing*, 25:227–242, 2015.

[34] Tamara Mchedlidze and Antonios Symvonis. Crossing-free acyclic hamiltonian path completion for planar st-digraphs. In *International Symposium on Algorithms and Computation*, pages 882–891. Springer, 2009.

[35] MiniZinc Challenge Committee. Minizinc challenge 2024 results, 2024. Accessed: 2025-04-10. URL: `https://www.minizinc.org/challenge/2024/results/`.

[36] Shannon Overbay. Graphs with small book thickness. *Missouri Journal of Mathematical Sciences*, 19(2):121–130, 2007.

[37] Laurent Perron and the OR-Tools Team. Or-tools, 2011–2025. URL: `https://developers.google.com/optimization`.

[38] Arnold L Rosenberg and Lenwood S Heath. *Graph separators, with applications.* Springer Science & Business Media, 2005.

[39] Bart Selman and Hector J. Levesque. The hard and easy distribution of SAT problems. In *Proceedings of the 10th AAAI*, pages 440–446, 1992.

[40] Roberto Tamassia. *Handbook of graph drawing and visualization.* CRC press, 2013.

[41] The On-Line Encyclopedia of Integer Sequences. Sequence A003024: Number of directed acyclic graphs (DAGs) with n labeled nodes, 2024. Accessed: 2024-03-20. URL: `https://oeis.org/A003024`.

[42] Manfred Wiegers. Recognizing outerplanar graphs in linear time. In *Graph-Theoretic Concepts in Computer Science: International Workshop WG'86 Bernried, Federal Republic of Germany, June 17–19, 1986 Proceedings 12*, pages 165–176. Springer, 1987.

[43] Avi Wigderson. The complexity of the Hamiltonian circuit problem for maximal planar graphs. *EECS Department Report*, 298, 1982.

Appendix A

Supplementary Code

This appendix presents the core source code used to encode and solve the Upward Book Embedding problems discussed in this thesis. We provide both SAT and CP models for completeness and reproducibility.

A.1 SAT-1

The first script implements the SAT-1 model, which encodes general k-Page Upward Book Embedding instances into CNF formulas. Additionally, it provides a decoding function to reconstruct the book spine order and page assignments from a satisfying solver output.

```python
from pysat.formula import CNF
from functools import cmp_to_key


def get_variables(N, M, P):
    """
    Generates Boolean variables for the encoding:
    - L(i, j): whether vertex i is to the left of vertex j
    - EP(i, p): whether edge i is assigned to page p
    - X(i, j): whether edges i and j are assigned to the same
       page
    """
    variable_count = 0

    is_left_to = {}
    for i in range(N):
        for j in range(i + 1, N):
            variable_count += 1
            is_left_to[(i, j)] = variable_count
    L = lambda i, j: is_left_to[(i, j)] if i < j else -
       is_left_to[(j, i)]

    edge_to_page = {}
    for i in range(M):
        for p in range(P):
            variable_count += 1
            edge_to_page[(i, p)] = variable_count
    EP = lambda i, p: edge_to_page[(i, p)]

    edges_on_same_page = {}
    for i in range(M):
        for j in range(i + 1, M):
            variable_count += 1
            edges_on_same_page[(i, j)] = variable_count
    X = lambda i, j: edges_on_same_page[(i, j)] if i < j else
       edges_on_same_page[(j, i)]

    return L, EP, X


def encode_upward_book_embedding(digraph, P):
    """
    SAT encodes Upward Book Embedding for P pages.
    """
    cnf = CNF()
    nodes = list(digraph.nodes)
```

```python
    edges = list(digraph.edges())
    N = len(nodes)
    M = len(edges)

    node_index = {nodes[i]: i for i in range(N)}
    L, EP, X = get_variables(N, M, P)

    # Enforce transitivity: if i < j and j < k then i < k
    for i in range(N):
        for j in range(N):
            for k in range(N):
                if len({i, j, k}) == 3:
                    cnf.append([-L(i, j), -L(j, k), L(i, k)])

    # Enforce topological order: for every edge (u,v), u must
       come before v
    for u, v in edges:
        i, j = node_index[u], node_index[v]
        cnf.append([L(i, j)])

    # Ensure every edge is assigned at least one page
    for i in range(M):
        cnf.append([EP(i, p) for p in range(P)])

    # Define X(i,j): edges i and j share a page if they are
       assigned to the same page
    for i in range(M):
        for j in range(i + 1, M):
            for p in range(P):
                cnf.append([X(i, j), -EP(i, p), -EP(j, p)])

    # Planarity constraints: prevent crossings on the same
       page
    for a in range(M):
        for b in range(a + 1, M):
            i, j = map(lambda x: node_index[x], edges[a])
            k, l = map(lambda x: node_index[x], edges[b])
            if len({i, j, k, l}) == 4:
                forbidden_orders = [
                    (i, k, j, l), (i, l, j, k),
                    (j, k, i, l), (j, l, i, k),
                    (k, i, l, j), (k, j, l, i),
                    (l, i, k, j), (l, j, k, i)
                ]
                for (a1, a2, a3, a4) in forbidden_orders:
                    cnf.append([-X(a, b), -L(a1, a2), -L(a2,
                        a3), -L(a3, a4)])

    return cnf


def decode_book_embedding(graph, P, solution):
    """
    Decodes SAT solver output into vertex spine order and edge
        -to-page assignments.
    """
    if not solution:
        return

    vertices = list(graph.nodes)
    edges = list(graph.edges)
    N, M = len(vertices), len(edges)

    L, EP, X = get_variables(N, M, P)

    value_of = {}
    for var in solution:
        value_of[abs(var)] = var > 0
```

```python
            value_of[-abs(var)] = var < 0

    # Determine spine order by sorting according to L(i,j)
      relations
    print('Book spine vertex order:')
    vertices.sort(key=cmp_to_key(lambda i, j: -1 if value_of[L
      (i, j)] else 1))
    vertices_str = ', '.join(f'v{i}' for i in vertices)
    print(vertices_str)
    print()

    # Determine page assignment for each edge
    print('Edge to page assignment:')
    by_pages = [[] for _ in range(P)]
    for i in range(M):
        assigned_pages = [p for p in range(P) if value_of[EP(i
          , p)]]
        pages_str = ', '.join(f'p{p}' for p in assigned_pages)
        u, v = edges[i]
        print(f'e{i} (v{u}, v{v}) - {pages_str}')
        if assigned_pages:
            by_pages[assigned_pages[0]].append((u, v))

    return vertices, by_pages
```

Listing A.1: Python script for SAT-1 encoding and decoding.

A.2 SAT-2

The second script implements the SAT-2 model, focused specifically on encoding 2-Page Upward Book Embedding instances into CNF.

```python
from pysat.formula import CNF


```

```python
def get_variables(N):
    """
    Generates Boolean variables for the 2-Page Upward Book
       Embedding encoding:
    - L(i, j): whether vertex i is to the left of vertex j
    - TOP(i): whether edge i is assigned to the top page
    """
    variable_count = 0

    is_left_to = {}
    for i in range(N):
        for j in range(i + 1, N):
            variable_count += 1
            is_left_to[(i, j)] = variable_count
    L = lambda i, j: is_left_to[(i, j)] if i < j else -
       is_left_to[(j, i)]

    # Edge page assignment variables start after L variables
    TOP = lambda i: variable_count + i + 1

    return L, TOP


def encode_2UBE(digraph):
    """
    SAT encodes the 2-Page Upward Book Embedding (2UBE)
       problem.
    Assumes exactly two pages: top and bottom.
    """
    cnf = CNF()
    nodes = list(digraph.nodes)
    edges = list(digraph.edges())
    N = len(nodes)
    M = len(edges)

    node_index = {nodes[i]: i for i in range(N)}
    L, TOP = get_variables(N)

    # Enforce transitivity: if i < j and j < k then i < k
    for i in range(N):
        for j in range(N):
            if i == j:
                continue
            for k in range(N):
                if i != k and j != k:
                    cnf.append([-L(i, j), -L(j, k), L(i, k)])

    # Enforce topological order: for every edge (u,v), u must
       come before v
    for u, v in edges:
        i, j = node_index[u]
        j = node_index[v]
        cnf.append([L(i, j)])

    # Planarity constraints based on page assignments
    for a in range(M):
        for b in range(a + 1, M):
            i, j = map(lambda x: node_index[x], edges[a])
            k, l = map(lambda x: node_index[x], edges[b])

            if len({i, j, k, l}) == 4:  # ensure pairwise
               distinct vertices
                # Crossing when both edges on top page
                cnf.append([-TOP(a), -TOP(b), -L(i, k), -L(k,
                   j), -L(j, l)])
                cnf.append([-TOP(a), -TOP(b), -L(k, i), -L(i,
                   l), -L(l, j)])

                # Crossing when both edges on bottom page
                cnf.append([TOP(a), TOP(b), -L(i, k), -L(k, j)
                    , -L(j, l)])
                cnf.append([TOP(a), TOP(b), -L(k, i), -L(i, l)
                    , -L(l, j)])

    # Arbitrarily fix one edge to the top page for symmetry
        breaking
    cnf.append([TOP(0)])

    return cnf
```

Listing A.2: Python script for SAT-2 encoding.

A.3 CP

The third script provides a Constraint Programming (CP) model using Google's OR-Tools CP-SAT Solver. It encodes 2UBE as a constraint satisfaction problem, using integer variables to represent vertex positions along the spine instead of boolean variables for pairwise relative positioning.

```
from ortools.sat.python import cp_model


def solve_2UBE(n, edges):
    """
    Solves the 2-Page Upward Book Embedding (2UBE) problem
        using Constraint Programming (CP).
    - Determines a linear spine order of vertices.
    - Assigns edges to two pages to avoid crossings.
    """
    # Create the model
    model = cp_model.CpModel()
    m = len(edges)

    # Variables:
    # pos_of_node[i] = position of vertex i along the spine (0 to n-1)
    pos_of_node = [model.NewIntVar(0, n - 1, f'pos_{i}') for i in range(n)]
    # page_of_edge[i] = 0 (bottom page) or 1 (top page)
    page_of_edge = [model.NewBoolVar(f'edge_page_{i}') for i in range(m)]

    # All vertices must have different spine positions
    model.AddAllDifferent(pos_of_node)

    # Enforce topological order: u must come before v for each edge (u,v)
    for u, v in edges:
        model.Add(pos_of_node[u] < pos_of_node[v])

    # Planarity constraints: avoid overlaps on same page
    for i, (u, v) in enumerate(edges):
        for j, (w, x) in enumerate(edges):
            if i != j and len({u, v, w, x}) == 4:
                # Create auxiliary Boolean variables for overlap conditions
                overlap1 = model.NewBoolVar(f'overlap1_{u}_{v}_{w}_{x}')
                overlap2 = model.NewBoolVar(f'overlap2_{u}_{v}_{w}_{x}')

                # Individual comparisons
                u_lt_w = model.NewBoolVar(f'u_lt_w_{u}_{w}')
                w_lt_v = model.NewBoolVar(f'w_lt_v_{w}_{v}')
                v_lt_x = model.NewBoolVar(f'v_lt_x_{v}_{x}')

                w_lt_u = model.NewBoolVar(f'w_lt_u_{w}_{u}')
                u_lt_x = model.NewBoolVar(f'u_lt_x_{u}_{x}')
                x_lt_v = model.NewBoolVar(f'x_lt_v_{x}_{v}')

                # Define conditions for overlap1: u < w < v <
                   x
                model.Add(pos_of_node[u] < pos_of_node[w]).
                   OnlyEnforceIf(u_lt_w)
                model.Add(pos_of_node[u] >= pos_of_node[w]).
                   OnlyEnforceIf(u_lt_w.Not())

                model.Add(pos_of_node[w] < pos_of_node[v]).
                   OnlyEnforceIf(w_lt_v)
                model.Add(pos_of_node[w] >= pos_of_node[v]).
                   OnlyEnforceIf(w_lt_v.Not())

                model.Add(pos_of_node[v] < pos_of_node[x]).
                   OnlyEnforceIf(v_lt_x)
                model.Add(pos_of_node[v] >= pos_of_node[x]).
                   OnlyEnforceIf(v_lt_x.Not())

                model.AddBoolAnd([u_lt_w, w_lt_v, v_lt_x]).
                   OnlyEnforceIf(overlap1)
                model.AddBoolOr([u_lt_w.Not(), w_lt_v.Not(),
                   v_lt_x.Not()]).OnlyEnforceIf(overlap1.Not()
                   )

                # Define conditions for overlap2: w < u < x <
                   v
                model.Add(pos_of_node[w] < pos_of_node[u]).
                   OnlyEnforceIf(w_lt_u)
                model.Add(pos_of_node[w] >= pos_of_node[u]).
                   OnlyEnforceIf(w_lt_u.Not())

                model.Add(pos_of_node[u] < pos_of_node[x]).
```

```
OnlyEnforceIf(u_lt_x)
                model.Add(pos_of_node[u] >= pos_of_node[x]).OnlyEnforceIf(u_lt_x.Not())

                model.Add(pos_of_node[x] < pos_of_node[v]).OnlyEnforceIf(x_lt_v)
                model.Add(pos_of_node[x] >= pos_of_node[v]).OnlyEnforceIf(x_lt_v.Not())

                model.AddBoolAnd([w_lt_u, u_lt_x, x_lt_v]).OnlyEnforceIf(overlap2)
                model.AddBoolOr([w_lt_u.Not(), u_lt_x.Not(), x_lt_v.Not()]).OnlyEnforceIf(overlap2.Not())

                # If two edges are on the same page, they must not overlap
                same_page = model.NewBoolVar(f'same_page_{i}_{j}')
                model.Add(page_of_edge[i] == page_of_edge[j]).OnlyEnforceIf(same_page)
                model.Add(page_of_edge[i] != page_of_edge[j]).OnlyEnforceIf(same_page.Not())

                model.AddImplication(same_page, overlap1.Not())
                model.AddImplication(same_page, overlap2.Not())

    # Solve the model
    solver = cp_model.CpSolver()
    status = solver.Solve(model)

    # Output results
```

```python
    if status == cp_model.OPTIMAL or status == cp_model.FEASIBLE:
        print("Solution found:")
        # Retrieve and sort nodes by position
        node_positions = [(i, solver.Value(pos_of_node[i])) for i in range(n)]
        node_positions.sort(key=lambda x: x[1])
        order = [node for node, pos in node_positions]

        return (order, [solver.Value(page) for page in page_of_edge])
    else:
        print("No solution found.")
```

Listing A.3: Python script for CP model solving 2UBE.