BACHELOR´S THESIS / ENGINEERING 2025

Deterministic and Randomized Approaches to Constraint Satisfaction Problems

Yuyan Wang

upf. Universitat Pompeu Fabra Barcelona

Deterministic and Randomized Approaches to Constraint Satisfaction Problems

TREBALL FI DE GRAU DE

Yuyan Wang

Director: Victor Dalmau

Grau en Enginyeria Matemàtica en Ciència de Dades

Curs 2024-2025

upf. Universitat Pompeu Fabra Barcelona

Escola d'Enginyeria

To my supervisor Victor Dalmau, whose inspiring classes on algorithms and contagious enthusiasm made this work possible :)

Acknowledgements

I would like to express my deepest gratitude to everyone who has accompanied me throughout my journey; a journey that has just begun.

To my mentors, for guiding me along the right path and for their invaluable advice;
To my professors, for sharing their knowledge and for their inspiring classes;
To my colleagues, for their constant support and encouragement;
To my classmates, for their good energy and companionship;
and to my family and all those who have believed in me, for their trust and unwavering support.

A very special thanks goes to my thesis supervisor, Victor Dalmau, for his inspiring guidance throughout this project, for his passion for algorithms, and for every algorithm session we've shared along the way.

Abstract

This thesis explores algorithmic approaches to solving Constraint Satisfaction Problems (CSPs), with a particular focus on randomized algorithms. The study targets CSP instances that combine three specific classes of constraints: Value Restriction Constraints, 2-Fan Constraints, and Binary Permutation Constraints. These form one of the largest known tractable fragments of the general binary CSP landscape.

To build a solid foundation, the thesis first introduces these constraint families and develops deterministic algorithms for each, gaining deeper insights into their structural properties. The core contribution is the design and analysis of a general randomized algorithm that solves combined instances of these constraints with polynomial expected runtime. Inspired by techniques such as Papadimitriouâs random walk for 2-SAT, this approach demonstrates how randomization can effectively find a solution for such CSPs.

The results align with and extend prior theoretical work on tractable CSP classes, providing an algorithmic solution for this important subclass of problems.

Resum

Aquesta tesi explora algoritmes per a la resolució de Problemes de Satisfacció de Restriccions (CSP), amb un èmfasi particular en els algoritmes randomitzats. L'estudi es centra en instàncies de CSP que combinen tres classes específiques de restriccions: restriccions de valor (Value Restriction Constraints), restriccions 2-Fan i restriccions de permutació binària (Binary Permutation Constraints). Aquestes constitueixen un dels fragments tractables més coneguts dins del panorama general dels CSP binaris.

Per establir una base sòlida, la tesi introdueix primer aquestes famílies de restriccions i desenvolupa algoritmes deterministes per a cadascuna, aprofundint en la comprensió de les seves propietats estructurals. La contribució principal consisteix en el disseny i l'anàlisi d'un algoritme randomitzat general que resol instàncies combinades d'aquestes restriccions amb un temps esperat d'execució polinòmic. Inspirat en tècniques com l'algoritme de Papadimitriou per al problema 2-SAT, aquest algoritme demostra com l'aleatorització pot trobar solucions de manera efectiva per a aquest tipus de CSP.

Els resultats s'alineen amb els treballs teòrics previs sobre classes tractables de CSP, i aporten una solució algorítmica per a aquest important subconjunt de problemes.

Contents

List of Figures

List of Algorithms

Introduction

This thesis explores algorithmic approaches for solving Constraint Satisfaction Problems (CSPs), with a particular emphasis on randomized algorithms. The goal of this work is to investigate how randomized methods can be leveraged to efficiently handle specific classes of CSPs that exhibit certain structural restrictions.

The first part of the thesis introduces and defines the selected constraint families in a unified and consistent framework; defining the base for the algorithms that follow.

Subsequently, the second part of the thesis revisits deterministic algorithms for these constraint types. Although efficient deterministic algorithms already exist in the literature, this work independently constructs and analyzes such algorithms from scratch. This process provides both a personal and conceptual understanding of the problem structure and key algorithmic ideas.

The main contribution of the thesis lies in the development and analysis of a general randomized algorithm capable of solving CSP instances that combine three important classes of constraints: Value Restriction Constraints, 2-Fan Constraints, and Binary Permutation Constraints. The algorithm achieves polynomial expected runtime for this combined class, which covers the tractable (solvable in polynomial time) cases identified in prior theoretical work.

Motivation

The initial motivation for this thesis arose from my prior exposure to the fundamental concepts of algorithms and computational complexity, particularly through a course named *Algorithm Design*. That course introduced me to the rich landscape of algorithmic thinking and formal problem classes such as `P` and `NP`, as well as the central open question of whether `P` equals `NP`. This naturally sparked my curiosity about the boundary between tractable and intractable problems, and the role that algorithm design plays in navigating this boundary.

One area that particularly captured my interest was the study of the SAT (Boolean Satisfiability) problem. During the course, we explored concrete examples of SAT solving, which highlighted both the theoretical importance of the

problem and the practical value of efficient algorithms for handling it. Despite the apparent simplicity of the SAT formulation, it became clear that developing effective solution methods is far from trivial without an understanding of algorithmic techniques. This experience deepened my appreciation for the elegance and power of algorithms in tackling complex combinatorial problems.

Encouraged by this interest, I was motivated to explore broader classes of problems within the same regime, which led me to this study of Constraint Satisfaction Problems (CSPs). In particular, the idea of generalizing beyond Boolean variables (i.e., allowing variables to take values from larger domains) immediately really intrigued me. This generalization significantly increases the modeling power of CSPs and opens the door to representing a wide variety of real-world problems in scheduling, planning, resource allocation, and beyond.

Thus, this thesis represents a natural extension of my curiosity, moving from SAT and 2-SAT to richer and more expressive forms of CSPs (although restricted to some types of constraints), and investigating both deterministic and randomized algorithmic approaches for solving them.

State of Art

A central goal in CSP research is to understand which classes of instances can be solved efficiently, despite the fact that CSPs are NP-complete in general [14]. One major direction in this area focuses on identifying tractable constraint languages; which are sets of allowed constraints that guarantee polynomial-time solvability. A key tool in this characterization is the notion of *polymorphisms*: algebraic operations that preserve solutions [3, 8]. If a set of constraints admits certain polymorphisms, the corresponding CSP is guaranteed to be tractable [8, 9].

One influential result in this line of work is the identification of the so-called 0/1/all constraint class [5]. It was shown that CSP instances built from three types of constraints (value restrictions, 2-fan constraints, and binary permutation constraints) fall into this class and can be solved in polynomial time [15]. These three constraint types are exactly the ones studied in this thesis, and they represent one of the largest known tractable fragments of the general binary CSP landscape.

At the same time, randomized algorithms have become an important tool in CSP solving. They often provide practical advantages when deterministic algorithms are too complex or when heuristic performance is important [11]. In particular, randomized methods can achieve expected polynomial-time performance on certain structured CSPs. A famous example is Papadimitriou's 1991 random walk algorithm for 2-SAT [12], which shows that a simple randomized local search strategy can efficiently solve this well-known CSP class. Such results have inspired the use of randomization as a general strategy in CSP algorithm design.

Contribution of this thesis. This thesis builds upon these foundations by developing and analyzing a *randomized* algorithm that solves CSP instances combining value restrictions, 2-fan constraints, and binary permutation constraints (the entire 0/1/all class) in polynomial expected time.

Work Plan and Task Progression

Since this is a rather theoretical thesis, the process and timeline have remained flexible throughout the project. Unlike more implementation-heavy theses, this work has focused primarily on deepening conceptual understanding and developing algorithms. From the very beginning, the main objective was to explore CSPs more thoroughly, starting from well-known problems like 2-SAT and gradually expanding into more complex constraint classes.

The first phase of this thesis, spanning from October to December 2024, was dedicated to building foundational knowledge. This involved studying existing deterministic algorithms for 2-SAT, understanding Papadimitriou's randomized algorithm, and analyzing why these approaches work. Alongside theoretical explorations, early efforts included independently implementing initial deterministic versions of solvers.

In the subsequent months, the project shifted toward the design of random algorithms tailored to specific constraint classes. From January to April 2025, several algorithms were proposed and formalized. The focus during this phase was on correctness proofs, complexity analysis, and refining the implementation of these algorithms. Special attention was given to adapting the style of randomized analysis introduced by Papadimitriou to this setting.

As the thesis progressed into spring, efforts were made to unify these algorithms into a general approach capable of solving combinations of the three targeted constraint types. By early June 2025, the final version of the combined randomized algorithm had been defined. From that point forward, the focus shifted toward formalizing the thesis document and polishing arguments.

Chapter 1

BACKGROUND AND PRELIMINARIES

1.1 Complexity Classes: P, NP, and NP-complete

In computational complexity theory, problems are classified into various complexity classes based on the resources required to solve them, most notably time and space.

P is the class of decision problems that can be solved in polynomial time by a deterministic algorithm. That is, there exists an algorithm that solves any instance of the problem of size n in $O(n^k)$ time for some constant k. Problems in P are considered efficiently solvable.

NP is the class of decision problems for which a proposed solution can be verified in polynomial time. In other words, given a candidate solution, it is possible to check its correctness efficiently. It is important to note that this does not imply that the problem can be solved efficiently, only that verifying a solution is efficient [1].

A problem is said to be NP-complete if it belongs to NP and is as hard as any other problem in NP. More formally, a problem A is NP-complete if every problem in NP can be reduced to A using a polynomial-time reduction. NP-complete problems are considered the most difficult problems within NP. If a polynomial-time algorithm were found for any NP-complete problem, it would imply that $P = NP$.

1.2 Boolean Satisfiability Problems (SAT)

The Boolean Satisfiability Problem (SAT) is a fundamental decision problem in computational complexity theory. It is defined as follows: given a Boolean for-

mula composed of logical variables, the task is to determine whether there exists an assignment of truth values (true or false) to the variables that makes the entire formula true.

Formally, a Boolean formula is typically expressed in Conjunctive Normal Form (CNF), which is a conjunction (logical AND) of clauses, where each clause is a disjunction (logical OR) of literals. A literal is a variable or its negation. SAT is widely recognized as one of the most prominent NP-complete problems, serving as a foundational benchmark for complexity theory.

Definition 1.2.1 (SAT)**.** The Boolean Satisfiability Problem (SAT) is the problem of determining whether a given CNF is satisfiable. That is, whether there exists an assignment of truth values to the variables such that the formula evaluates to true.

1.2.1 2-SAT

The 2-Satisfiability Problem (2-SAT) is a special case of SAT in which each clause contains at most two literals. Despite the general SAT problem being NP-complete, 2-SAT is solvable in polynomial time. This notable distinction arises from structural properties specific to 2-literal clauses.

Example 1.2.1. (2-SAT) Consider the following Boolean formula in CNF form:

$$(x \vee y) \wedge (\neg x \vee z) \wedge (\neg y \vee \neg z).$$

This formula is an example of a 2-SAT instance, as each clause contains exactly two literals. Determining if there's an assignment of truth values to the variables x, y, z that satisfies this formula is a typical instance of the 2-SAT problem.

Definition 1.2.2 (2-SAT)**.** The 2-SAT is a restriction of the SAT in which each clause contains at most two literals.

Formally, given a CNF where each clause has at most two literals, the goal is to determine whether there exists an assignment of truth values to the variables that satisfies the entire formula.

Deterministic Approaches for 2-SAT A prominent deterministic approach employs graph theory and is based on the construction of an *implication graph* [2, 10]. Specifically, each clause in a 2-SAT instance can be represented by two directed implications. For instance, a clause $(x \vee y)$ is equivalently represented by two implications:

$$(\neg x \to y) \quad \text{and} \quad (\neg y \to x).$$

By constructing such an implication graph for the entire formula, solving 2-SAT reduces to analyzing the structure of this graph. The critical insight used in this approach is the identification of strongly connected components (SCCs). If, within any SCC, a variable and its negation both appear, the formula is unsatisfiable. Otherwise, the instance is satisfiable, and a valid truth assignment can be obtained by processing these SCCs in topological order. This deterministic method operates in linear time $O(|V| + |E|)$, making it highly efficient and practical for solving 2-SAT problems.

Randomized Approaches for 2-SAT (Papadimitriou) Another influential approach to solving 2-SAT efficiently is the randomized algorithm proposed by Papadimitriou [12]. This algorithm is of particular interest because, despite its simplicity, it achieves polynomial expected runtime and offers important insights into randomized strategies for CSPs.

The randomized algotithm proposed by Papadimitriou is straightforward and intuitive, described as follows: *"Start with any truth assignment. While there are unsatisfied clauses, pick one and flip a random literal in it."*

The randomized algorithm is proven to solve any satisfiable 2-SAT instance in expected $O(n^2)$ steps, where n is the number of variables. This result demonstrates how randomized approaches can be highly effective even for problems that also admit deterministic solutions.

In addition to its practical performance, the algorithm serves as an entry point for extending randomized techniques to broader classes of satisfiability and CSPs, a direction explored in this thesis.

1.3 Constraint Satisfaction Problems (CSP)

A Constraint Satisfaction Problem (CSP) is a fundamental computational framework widely used for modeling and solving combinatorial decision problems. Informally, a CSP involves assigning values to a set of variables subject to certain constraints, which restrict allowable combinations of assignments.

This thesis focuses specifically on *Binary CSPs*, a well-studied class of CSPs in which all constraints involve at most two variables. In this setting, constraints can be either unary (restricting the value of a single variable) or binary (restricting the joint assignment of a pair of variables). In Binary CSPs, the formalism generalizes beyond Boolean variables and logical ORs; it also defines allowed pairs of values through explicit binary constraints C_{ij}[1] $\subseteq D_i \times D_j$.

[1]Same notation may appear as C_{xy}, etc., with variable names used in the subscript, instead of numbers. In all cases, this notation refers to a binary constraint between a pair of variables.

More formally, in the context of this thesis, let's define a finite Binary CSP as follows:

Definition 1.3.1 (Binary Constraint Satisfaction Problem)**.** A finite Binary Constraint Satisfaction Problem consists of:

- A finite set of variables $X = \{x_1, x_2, \ldots, x_n\}$.
- A finite domain D shared by all variables, i.e., $D_i = D$ for every $x_i \in X$; where D_i represents the possible values that variable x_i can take.
- A finite set of constraints C, where each constraint is either:
  - a unary constraint $C_i \subseteq D$, specifying the allowed values for variable x_i, or
  - a binary constraint $C_{ij} \subseteq D \times D$, specifying the allowed pairs of values for variables x_i and x_j.

A solution to a CSP is an assignment of values from the domain D to all variables such that every constraint is satisfied. More explicitly, an assignment $\{(x_1, v_1), (x_2, v_2), \ldots, (x_n, v_n)\}$ is a solution if the following holds:

- For every unary constraint C_i, $v_i \in C_i$.
- For every binary constraint C_{ij}, $(v_i, v_j) \in C_{ij}$.

A CSP instance is said to be *satisfiable* if at least one valid solution exists, and *unsatisfiable* otherwise.

In the context of this thesis, the constraints are defined explicitly and pairwise (or unary where needed), with a focus on binary relationships. The work specifically explores particular restricted classes of CSPs characterized by special structural constraints, namely: *Value Restricted CSP*, *2-Fan CSP*, and *Binary Permutation CSP*.

Each of these classes imposes distinct restrictions on the allowed assignments, significantly affecting both their solvability and the complexity of algorithms designed to solve them. The specific definitions of these constraints will be presented in the subsequent subsections.

1.3.1 Value Restricted CSP

A Value Restricted Constraint is a type of constraint applied individually to a single variable within a CSP. Unlike binary constraints, it directly limits the domain of the variable, allowing only a specific subset of its original values. Formally, a Value Restricted Constraint is defined as follows:

Definition 1.3.2 (Value Restricted Constraint)**.** Given a variable x_i with finite domain D_i, a *Value Restricted Constraint* $C_i \subseteq D_i$ is defined as a subset of its original domain. The constraint explicitly limits variable x_i to only take values within the restricted domain C_i.

An assignment (x_i, v_i) satisfies the Value Restricted Constraint if and only if:

$$v_i \in C_i.$$

Example 1.3.1. (Value Restriction) Suppose a variable x_i originally has a domain $D_i = \{1, 2, 3\}$. A value restriction could specify that x_i can only take values from a subset $D_i' = \{2, 3\}$. Consequently, any valid assignment involving x_i must select either 2 or 3.

1.3.2 2 Fan CSP

A 2-Fan Constraint is a specialized type of binary constraint between two variables, characterized by the existence of exactly one designated "wild" value for each variable involved. If one variable is assigned its wild value, then any assignment to the other variable satisfies the constraint, irrespective of its chosen value. Formally, the 2-Fan Constraint is defined as follows:

Definition 1.3.3 (2-Fan Constraint)**.** Given two variables x_i and x_j with respective finite domains D_i and D_j, a constraint $C_{ij} \subseteq D_i \times D_j$ is called a *2-Fan Constraint* if there exist unique values $w_i \in D_i$ and $w_j \in D_j$ (the *"wild values"*) such that:

$$C_{ij} = (\{w_i\} \times D_j) \cup (D_i \times \{w_j\}).$$

In other words, the constraint is satisfied if and only if at least one of the variables is assigned its wild value.

Example 1.3.2. (2-Fan Constraint) Consider two variables x and y, each with the domain $D_x = D_y = \{1, 2, 3\}$. Suppose the wild values are defined as $w_x = 1$ and $w_y = 3$. The 2-Fan constraint between these two variables is then explicitly given by: $C_{xy} = (\{1\} \times \{1, 2, 3\}) \cup (\{1, 2, 3\} \times \{3\})$.

Thus, this constraint will be satisfied if $x = 1$ (regardless of the value of y), or if $y = 3$ (regardless of the value of x). Assignments such as $(1, 2)$, $(1, 1)$, or $(2, 3)$ satisfy the constraint, whereas an assignment such as $(2, 2)$ does not satisfy it, since neither variable holds its respective wild value.

1.3.3 Binary Permutation CSP

A Binary Permutation Constraint is a specialized binary constraint defined between two variables, characterized by a strict permutation relationship between

the allowed values of these variables. Formally, a binary permutation constraint explicitly requires that each value in the domain of the first variable appears exactly once paired with a unique value from the domain of the second variable, and vice versa.

Definition 1.3.4 (Binary Permutation Constraint)**.** Given two variables x_i and x_j with finite and equally-sized domains D_i and D_j (i.e., $|D_i| = |D_j|$), a constraint $C_{ij} \subseteq D_i \times D_j$ is called a ***Binary Permutation Constraint*** if and only if there exists a bijection $\pi : D_i \to D_j$ such that:

$$C_{ij} = \{(v, \pi(v)) \mid v \in D_i\}.$$

Equivalently, every value from the domain D_i appears exactly once as the first element of each tuple in C_{ij}, and every value from the domain D_j appears exactly once as the second element.

Example 1.3.3. (Binary Permutation Constraint) Consider two variables x, y with domains $D_x = D_y = \{1, 2, 3\}$. Examples of valid Binary Permutation Constraints could be:

$$C_{xy} = \{(1, 1), (2, 2), (3, 3)\}$$

or another permutation such as:

$$C'_{xy} = \{(1, 2), (2, 3), (3, 1)\}.$$

In both cases, each value from D_x $(1, 2, 3)$ appears exactly once in the first position, and each value from D_y appears exactly once in the second position. These examples illustrate the strict permutation relationship inherent in Binary Permutation Constraints.

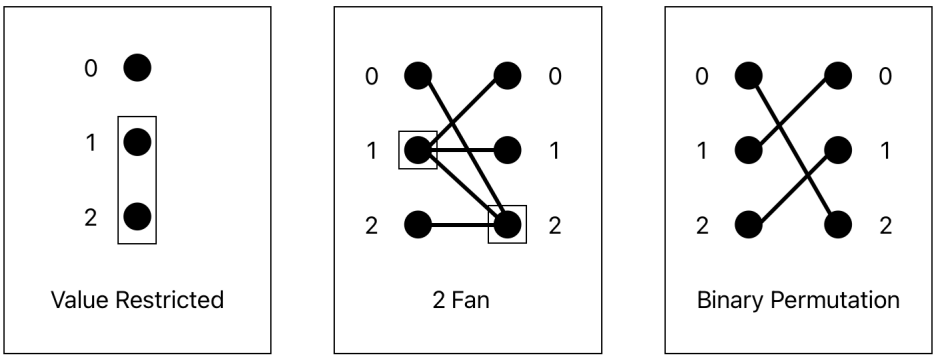

Figure 1.1: 3 types of constraints

1.3.4 Complexity of Binary CSP

As stated earlier, this thesis focuses on particular classes of constraints: Value Restricted, 2-Fan, and Binary Permutation Constraints. These special cases enable the design of efficient algorithms. In contrast, the general problem of solving Binary Constraint Satisfaction Problems (Binary CSP) is computationally hard. This section formally establishes that the general Binary CSP is NP-complete.

Proof outline. The proof of NP-completeness proceeds in two steps:

- **Binary CSP is in NP:** Given an assignment to all variables, it is possible to verify in polynomial time whether all binary constraints are satisfied.
- **Binary CSP is NP-hard:** A polynomial-time reduction is provided from a known NP-complete problem, namely 3-SAT, to Binary CSP.

Binary CSP is in NP. The verification step for a Binary CSP instance is straightforward. A candidate assignment consists of an assignment of one value $v_i \in D$ to each variable $x_i \in X$. Verification proceeds by checking each constraint:

- For every unary constraint $C_i \subseteq D$, it is verified whether $v_i \in C_i$.
- For every binary constraint $C_{ij} \subseteq D \times D$, it is verified whether the assigned pair $(v_i, v_j) \in C_{ij}$.

Since the number of unary constraints is $O(n)$ and the number of binary constraints is at most $O(n^2)$, and since each check can be performed in constant time, the total verification process runs in polynomial time with respect to n.

Binary CSP is NP-hard. The NP-hardness of Binary CSP is a well-known result (see, for example, [4, 6]), but for completeness, an independently derived proof is included here.

NP-hardness is established by providing a reduction from the 3-SAT problem to Binary CSP. Recall that 3-SAT is the problem of deciding whether a Boolean formula in CNF, where each clause is a disjunction of exactly three literals, is satisfiable.

The reduction proceeds as follows. For each clause in the 3-SAT formula, a set of binary constraints is introduced on new variables, ensuring that any satisfying assignment to the resulting CSP instance corresponds to a satisfying assignment of the original 3-SAT formula.

Construction. Consider a clause of the form $x \vee \neg y \vee z$. An auxiliary variable w is introduced to encode the partial truth value of the first two literals $x \vee \neg y$. The domain of w is defined to capture all possible combinations of truth assignments to x and y, as follows:

– $w = 0$: $x =$ False, $y =$ True
– $w = 1$: $x =$ True, $y =$ False
– $w = 2$: $x =$ False, $y =$ False
– $w = 3$: $x =$ True, $y =$ True

Next, the following binary constraints are introduced:

- A constraint between x and w, enforcing consistency between the value of x and the encoding stored in w.
- A constraint between y and w, likewise enforcing consistency.
- A constraint between w and z, ensuring that the clause $x \vee \neg y \vee z$ is satisfied. Specifically, the pair $(w, z) = (2, 0)$, corresponding to the assignment $x =$ False, $y =$ True, $z =$ False, must be excluded; as this is the only assignment that would falsify the clause.

By applying this construction to each clause in the 3-SAT formula, an instance of Binary CSP is obtained whose solutions correspond exactly to the satisfying assignments of the original formula.

This reduction clearly runs in polynomial time, as it introduces a constant number of variables and constraints per clause. Since 3-SAT is NP-complete and a polynomial-time reduction to Binary CSP has been provided, it follows that Binary CSP is NP-hard.

Overall result. Since Binary CSP is both in NP and NP-hard, it is NP-complete.

Chapter 2

DETERMINISTIC ALGORITHMS

In this chapter, deterministic algorithms are studied for solving Constraint Satisfaction Problems (CSPs) of the types introduced in the previous chapter: 2-SAT, 2-Fan Constraints, and Binary Permutation Constraints.

A deterministic algorithm is an algorithm that, given a particular input, always produces the same output and follows the same sequence of computational steps. Unlike randomized algorithms, which incorporate probabilistic choices and may exhibit variability in runtime or output, deterministic algorithms provide fully predictable behavior. In the context of CSPs, deterministic algorithms either return a valid satisfying assignment (if one exists) or report unsatisfiability, with a guaranteed worst-case time complexity.

Deterministic algorithms for CSPs have been extensively studied in the literature (see, e.g., [2, 15]). The results presented in this section are not novel in a strict sense; however, the algorithms and proofs included here have been derived independently in the course of this thesis work. Presenting them explicitly serves two purposes. First, it establishes clear and efficient baseline algorithms for the specific families of constraints under consideration in this thesis. Second, it provides insight into the structural properties of these problems, as deterministic algorithms often exploit particular symmetries or combinatorial features of the constraints.

The following subsections present and analyze deterministic algorithms for 2-SAT, 2-Fan Constraints, and Binary Permutation Constraints. For each case, the algorithmic approach is described, its correctness is discussed, and its computational complexity is analyzed.

2.1 2-SAT

As discussed in previous sections, the 2-SAT is a special case of Boolean satisfiability where each clause consists of at most two literals. Unlike the general SAT problem, which is NP-complete, 2-SAT is known to be solvable in polynomial time.

In this section, an efficient deterministic algorithm for 2-SAT based on graph theory is presented. The approach leverages the structure of *implication graphs* to model the dependencies between variable assignments, allowing to detect contradictions and construct valid solutions in a systematic way.

Note on Algorithm Provenance. The deterministic algorithm presented here is not an original design of this thesis. After initially developing a custom algorithm to solve 2-SAT, it became evident that the approach could be simplified by consulting existing literature. The final version presented is a definition based on the well-established graph-based method introduced by Aspvall, Plass, and Tarjan [2], which achieves linear-time complexity. The current description and implementation follow the structure of that approach.

Algorithm Description (Algorithm 1)

The deterministic algorithm solves 2-SAT by first representing the formula as a directed graph, known as the implication graph.Each literal ℓ (whether a variable or its negation) is represented as a vertex in the graph. For each clause $(\ell_1 \vee \ell_2)$, we add two directed edges:

$$(\neg\ell_1 \to \ell_2), \quad (\neg\ell_2 \to \ell_1).$$

These edges encode the logical implications implied by the clause. For example, if $\neg\ell_1$ is true, then ℓ_2 must also be true to satisfy the clause.

Once the implication graph is constructed, its strongly connected components (SCCs) are computed. The crucial property is that if a variable x and its negation $\neg x$ appear in the same SCC, then the formula is unsatisfiable. In this case, there exists a cycle of implications forcing x to simultaneously be assigned both `true` and `false`, which is impossible.

If no such contradiction is detected, a satisfying assignment is constructed by processing the SCCs in inverse topological order. In this context, topological order refers to an ordering of SCCs such that all directed edges between different components point from earlier to later components in the order. The inverse topological order thus visits components in the opposite direction of the edges, ensuring that no dependencies are violated when making assignments. During this process, if a variable x remains unassigned when its SCC is visited, it is assigned the value `true`, and its negation $\neg x$ is assigned `false`. This guarantees

that all implications are respected and the resulting assignment satisfies the entire formula.

Algorithm 1 Deterministic 2-SAT Solver (Implication Graph)

Input: A 2-SAT formula represented as a list of clauses.
Output: A satisfying assignment, or a determination of unsatisfiability.

procedure SOLVER2SAT
Construct the implication graph for the formula.
for each clause $(\ell_1 \vee \ell_2)$ **do**
    Add edges $(\neg\ell_1 \to \ell_2)$ and $(\neg\ell_2 \to \ell_1)$.
end for
Compute strongly connected components (SCCs) of the graph.
for each variable x **do**
    if x and $\neg x$ are in the same SCC **then**
        return "Unsatisfiable".
    end if
end for
Initialize all variables as unassigned.
for each SCC in inverse topological order **do**
    for each variable x in the SCC **do**
        if the literal x (non-negated) belongs to the SCC **then**
            Assign x to `true`.
        else
            Assign x to `false`.
        end if
    end for
end for
return final assignment.

Algorithm Correctness

The correctness of the algorithm relies on the well-established relationship between the structure of the implication graph and the logical satisfiability of the 2-SAT formula. The key property is that satisfiability can be fully determined by analyzing the strongly connected components (SCCs) of the graph.

If a variable x and its negation $\neg x$ appear in the same SCC, this signals a contradiction: there exists a cycle of implications that simultaneously forces x to be both `true` and `false`. In this case, the formula is unsatisfiable, and the algorithm terminates correctly.

If no such contradiction is present, the structure of the SCCs ensures that a consistent assignment of truth values can be constructed. By processing the SCCs in inverse topological order, the algorithm guarantees that variable assignments are made in a manner that respects all logical dependencies encoded in the graph. Since no SCC can influence a previously assigned SCC in this order, the resulting assignment will be globally consistent. The convention used to assign variables (`true` for the non-negated literal when first encountered, `false` otherwise) ensures that the chosen assignment satisfies all clauses of the original formula.

Thus, the algorithm correctly determines whether the 2-SAT formula is satisfiable, and if so, produces a valid satisfying assignment.

Algorithm Complexity

The algorithm runs in linear time with respect to the size of the input formula.

The implication graph contains $2n$ vertices (one for each literal x and $\neg x$), where n is the number of distinct variables. For each clause, two edges are added, resulting in $O(m)$ edges, where m is the number of clauses.

Strongly connected components can be computed in $O(n + m)$ time using standard algorithms such as Kosaraju's or Tarjan's algorithm [13]. The topological ordering of SCCs, as well as the assignment propagation, can also be performed in linear time.

Therefore, the overall time complexity of the algorithm is $O(n+m)$, which is linear in the size of the formula representation. This matches the known optimal complexity for solving 2-SAT deterministically.

2.2 2-Fan CSP

This section presents a deterministic algorithm for solving CSP instances in which all constraints are 2-Fan Constraints. The key observation is that the structure of 2-Fan Constraints allows the construction of a solution through a simple propagation process that respects the semantics of the wild values.

The algorithm proceeds by iteratively assigning values to variables in a way that ensures all constraints are satisfied. In particular, any variable that is required (through constraint propagation) to take its wild value is immediately assigned accordingly, simplifying the assignment process for connected variables.

Algorithm Description (Algorithm 2 and 3)

The deterministic algorithm for solving 2-Fan CSP builds on the observation that each variable x_i can be assigned either its wild value w_i or a non-wild value, and

that 2-Fan Constraints propagate forced assignments depending on these choices.

Algorithm 2 Deterministic 2-Fan CSP Solver

Input: A 2-Fan CSP represented as a set of variables and 2-Fan Constraints.
Output: A set of valid assignments for solvable components, or a determination of unsatisfiability for any component.

procedure SOLVER2FANCSP
Construct the constraint graph.
Compute the connected components of the graph.
for each connected component $\mathcal{C}$ **do**
  Initialize all variables in $\mathcal{C}$ as unassigned.
  Select an arbitrary 2-Fan Constraint C_{ij} in $\mathcal{C}$.
  for each possible initial assignment for x_i: w_i or any $v_i \neq w_i$ **do**
    Save current assignment state.
    Assign x_i to the chosen value in CurrentAssignment.
    (Result, FinalAssignment) $\leftarrow$ `Propagate(`x_i`, CurrentAssignment)`.
    if Result == Success **then**
      for each variable $x_k \in \mathcal{C}$ **do**
        if x_k is unassigned in FinalAssignment **then**
          Assign $x_k \leftarrow w_k$ in FinalAssignment.
        end if
      end for
      Record $FinalAssignment$ as valid assignment for $\mathcal{C}$.
      Proceed to the next component.
    else
      Restore previous assignment state.
    end if
  end for
  If no initial assignment leads to Success, report $\mathcal{C}$ as unsatisfiable.
end for
return assignments for solvable components.

In addition, the algorithm exploits the fact that the constraint graph of the CSP can be decomposed into connected components. The constraint graph is an undirected graph where each node represents a variable, and an edge is added between two variables if they participate in the same 2-Fan Constraint. A connected component of this graph is a set of variables that are linked through sequences of shared constraints; that is, there exists a path of constraints connecting any pair of variables in the component.

Since 2-Fan Constraints only affect variables within the same connected component, the problem can be solved independently on each component. Once a valid assignment is found for a component, this assignment is fixed and not modified even if other components turn out to be unsatisfiable. This ensures that the algorithm can correctly report which parts of the CSP are solvable, while preserving valid partial assignments.

Example 2.2.1. (Connected Component in the 2-Fan Constraints Setting) Consider a 2-Fan CSP with variables x, y, z, u, and the following constraints:

- C_{xy} connects x and y
- C_{yz} connects y and z
- C_{xu} connects x and u

In this case, $\{x, y, z, u\}$ forms a single connected component, because all four variables are linked through the chain of constraints. Conversely, if there is an additional constraint C_{vt} that is not involved in any constraint with x, y, z, u, it would form its own separate connected component composed by variables v, t.

The algorithm proceeds by solving each connected component separately. For a given component, an arbitrary 2-Fan Constraint C_{ij} is selected, and one of its variables, say x_i, is chosen as the initial variable to assign. The algorithm first attempts to assign x_i to its wild value w_i, and propagates the resulting forced assignments throughout the component. If this leads to a contradiction, the algorithm backtracks and tries assigning x_i to each of its non-wild values, one by one, again propagating consequences in each case. If all possible initial assignments for x_i lead to contradictions, the component is declared unsatisfiable.

Propagation at each step enforces the semantics of 2-Fan Constraints:

- For each constraint C_{ij} involving x_i and its neighbor x_j, consider the wild value w_i^{ij} defined by that constraint:
  - Case 1: If $x_i = w_i^{ij}$, then the constraint C_{ij} is satisfied without requiring x_j to take any particular value.
  - Case 2: If $x_i \neq w_i^{ij}$, then to satisfy C_{ij}, the neighbor x_j must be assigned w_j^{ij} (its wild value in this constraint), unless this leads to a conflict.
- This propagation is recursive: any newly assigned variable x_j must be processed similarly, considering all constraints C_{jk} in which it participates.
- The process continues until either:
  - All constraints are satisfied consistently with the current partial assignment, or
  - A contradiction is detected (an inconsistent forced assignment).

The process continues component by component. The algorithm returns a set

of valid assignments for all solvable components, and a report of unsatisfiability for any component that cannot be solved under any initial choice.

Algorithm 3 Helper Procedure: Propagate Forced Assignments

Input: Variable x_k, CurrentAssignment.
Output: (Success, CurrentAssignment) or (Conflict, null).

procedure PROPAGATE(x_k, CURRENTASSIGNMENT)
for each constraint C_{kj} involving x_k and neighbor x_j **do**
  Let w_k^{kj} and w_j^{kj} be the wild values of x_k and x_j in C_{kj}.
  if CurrentAssignment of $x_k \neq w_k^{kj}$ **then**
    if x_j is already assigned and $x_j \neq w_j^{kj}$ **then**
      return (Conflict, null).
    else if x_j is unassigned **then**
      Assign $x_j \leftarrow w_j^{kj}$ in CurrentAssignment.
      (Result, UpdatedAssignment) $\leftarrow$ `Propagate(`x_j`, CurrentAssignment).`
      if Result == Conflict **then**
        return (Conflict, null).
      else
        CurrentAssignment $\leftarrow$ UpdatedAssignment.
      end if
    end if
  end if
end for
return (Success, CurrentAssignment).

Algorithm Correctness

The correctness of the algorithm follows from two key properties of 2-Fan Constraints and the structure of the propagation process.

Soundness. The algorithm only returns an assignment for a component if all constraints within that component are satisfied. This is ensured by the propagation procedure, which enforces the semantics of 2-Fan Constraints: if a variable x_i is assigned to a non-wild value $v_i \neq w_i$, then each neighbor x_j involved in a constraint with x_i is forced to take its wild value w_j. The recursive propagation ensures that this implication is consistently applied throughout the component. If a conflict is detected (i.e. a variable is forced to simultaneously take two incompatible values), the current search branch is abandoned. Thus, any assignment

produced by the algorithm is guaranteed to satisfy all constraints in the corresponding component.

Completeness. The algorithm systematically explores all initial choices for one variable x_i in each connected component: first trying w_i, then each non-wild value $v_i \neq w_i$. For each such choice, it fully propagates the resulting forced assignments. If a satisfying assignment exists for the component, this process will find it. Conversely, if all initial choices for x_i lead to conflicts, the component is correctly declared unsatisfiable. Since each component is treated independently, valid assignments found for solvable components are preserved, even if other components turn out to be unsatisfiable.

In summary, if the algorithm outputs a valid assignment for a component, that assignment is guaranteed to satisfy all constraints within the component. If it reports a component as unsatisfiable, no valid assignment exists for that component.

Algorithm Complexity

The overall time complexity of the algorithm depends on two main factors: the decomposition of the constraint graph into connected components, and the cost of solving each component through recursive propagation and backtracking.

Decomposition: Constructing the constraint graph takes $O(n+m)$ time, where n is the number of variables and m is the number of constraints. Computing the connected components of an undirected graph can also be performed in $O(n+m)$ time using standard graph traversal algorithms such as Depth-First Search (DFS) or Breadth-First Search (BFS).

Per-component search: For each connected component, the algorithm explores all possible initial assignments for one selected variable x_i. The number of initial choices is at most $|D_i|$, where D_i is the domain of x_i. For each initial choice, the algorithm performs recursive propagation.

Propagation: Within each propagation step, the recursive `Propagate` procedure assigns values to variables and enforces consistency according to the semantics of 2-Fan Constraints. The key property of the algorithm is that each variable x_k is assigned at most once along a single propagation path. Once x_k is assigned, it will not be revisited in further recursive calls, ensuring that the assignment process proceeds in a forward, acyclic manner.

When a variable x_k is processed, the algorithm scans all constraints involving x_k, which corresponds to inspecting all of its neighbors in the constraint graph.

The number of such neighbors equals the degree d_k of x_k. Across the entire connected component, the sum of the degrees of all variables is proportional to the total number of constraints m_c. Each constraint can be accessed at most twice (once when processing each of its two participating variables). Therefore, the total number of constraint processed during a complete propagation is $O(2m_c) = O(m_c)$.

Furthermore, since each variable is assigned at most once, the number of recursive calls performed is bounded by $O(n_c)$, where n_c is the number of variables in the connected component.

Combining these observations, the overall cost of a single run of the propagation initiated by one initial assignment is $O(n_c + m_c)$. This reflects the fact that the algorithm performs one recursive visit per variable and scans each constraint at most twice across the entire propagation path.

Backtracking: In the worst case, the algorithm explores all possible initial choices for the selected variable x_i. For each choice, it performs a full propagation. Therefore, the worst-case time complexity per component is $O(|D| \cdot (n_c + m_c))$.

Since components are processed independently, the total time complexity of the algorithm is: $O\left(n + m + \sum_{\mathcal{C}} |D| \cdot (n_c + m_c)\right) = O\left(n + m + |D| \cdot (n + m)\right) = O\left((|D| + 1) \cdot (n + m)\right)$, where the sum is taken over all connected components $\mathcal{C}$, and x_i is the selected initial variable in component $\mathcal{C}$.

In many practical cases, domain sizes $|D|$ are small, and connected components are relatively sparse. In such cases, the algorithm runs efficiently and can quickly find valid assignments or detect unsatisfiable components.

2.3 Binary Permutation CSP

As previously defined, Binary Permutation CSP imposes that each value of each variable must appear exactly once across the set of allowed assignments for each pair of variables.

Solving Binary Permutation CSPs presents unique challenges compared to problems like 2-SAT or 2-Fan CSP. The permutation structure introduces strong global dependencies between variable assignments, requiring careful coordination to ensure that no value is repeated or omitted in the final solution.

The designed final algorithm for Binary Permutation CSP adopts a search-based approach using backtracking, enhanced with heuristic ordering and constraint propagation. While early designed prototypes by the author explored more explicit graph-based techniques, these proved unnecessarily complex for this problem class. The backtracking framework, provides an efficient mechanism to explore the solution space while pruning infeasible branches early.

Algorithm Description (Algorithm 4 and 5)

Algorithm 4 Deterministic Binary Permutation CSP Solver

Input: A set of Binary Permutation Constraints, each defined as a pair of variables (x, y) with a list of allowed value pairs.
Output: A complete variable assignment satisfying all constraints, or a determination that no solution exists.

procedure SOLVERBINARYPERMUTATIONCSP
Initialize domains of all variables according to their associated Binary Permutation Constraints.
Construct data structures to maintain current domain sizes and constraint links.
Call BACKTRACK($\{\}$)

The Binary Permutation CSP solver presented in this section is based on a recursive backtracking approach enhanced with constraint propagation. The central feature of this kind of contraints is their bijective restrictions. While restrictive, it is in fact what enables an efficient solution. It imposes strong consistency requirements that propagate deterministically through the constraint graph, significantly reducing the search space. In contrast, if the constraints allowed a larger or more flexible set of combinations, the problem would generally be harder to solve.

The algorithm begins by initializing the domains of all variables according to the specified Binary Permutation Constraints. Each constraint between a pair of variables (x_i, x_j) defines a set of allowed value pairs corresponding to a bijection $\pi : D_i \to D_j$, as described in the formal definition. Consequently, when a value is assigned to one variable, the possible values for its neighbors are immediately constrained.

At each step of the recursion, the algorithm iterates over the possible values for the selected variable. For each tentative assignment, it checks whether the current partial assignment remains consistent with all Binary Permutation Constraints. If so, the assignment is temporarily added to the current solution, and the constraints are propagated: this involves updating the domains of all connected variables to reflect the consequences of the new assignment, effectively reducing the search space. The algorithm then recursively proceeds to assign the next variable.

If a contradiction is encountered at any point (e.g., a variable's domain becomes empty), the algorithm backtracks: it removes the last assignment and restores the previous state of the domains. This ensures that only consistent assignments are pursued.

The process continues until either a complete and valid assignment is found, or all possibilities have been exhausted, where the algorithm concludes that the instance is unsatisfiable.

Algorithm 5 Backtrack Procedure for Binary Permutation CSP

```
0:  procedure BACKTRACK(assignment)
1:  if all variables are assigned then
2:      return assignment.
3:  end if
4:  Select an unassigned variable v.
5:  for each value val in the current domain of v do
6:      if assigning val to v does not violate any Binary Permutation Constraints
        then
7:          Add {v ↦ val} to the current assignment.
8:          Propagate constraints: update the domains of all connected variables to
            reflect this assignment.
9:          Call BACKTRACK(assignment).
10:         if BACKTRACK returns a valid assignment then
11:             return assignment.
12:         end if
13:         Remove {v ↦ val} from the assignment and restore domains to their
            previous state.
14:     end if
15: end for
16: return failure.
```

Algorithm Correctness

The correctness of the Binary Permutation CSP solver is grounded in the soundness of the search and constraint propagation mechanisms employed by the backtracking algorithm.

At each stage of the recursion, the algorithm maintains a partial assignment that is consistent with all Binary Permutation Constraints seen so far. When a variable v is assigned a value val, the constraint propagation step ensures that this assignment remains consistent with the corresponding bijective relationships defined by the Binary Permutation Constraints. In particular, for any constraint C_{ij} involving v, the domains of the neighboring variable are updated so that the required bijection property is preserved.

The use of backtracking guarantees that the algorithm explores the entire space of consistent assignments. If a contradiction is detected at any point, meaning that the domain of some variable becomes empty or a constraint cannot be satisfied;

the algorithm abandons the current branch and backtracks to try alternative assignments. This ensures that no invalid assignment is ever accepted.

When the algorithm reaches a state where all variables are assigned, the solution is necessarily valid. Since the Binary Permutation Constraints enforce that each value appears exactly once in the first and second positions of each constraint relation, and since constraint propagation preserves this structure during the search, the final assignment will satisfy all constraints if the algorithm terminates successfully. Conversely, if no assignment is found after exploring all branches, the instance is indeed unsatisfiable. The backtracking algorithm is exhaustive: failure to find a solution implies that no consistent global assignment exists.

Algorithm Complexity

The runtime complexity of the deterministic Binary Permutation CSP solver can be analyzed by examining the structure of its backtracking search and the impact of domain pruning during constraint propagation.

Consider an instance with n variables $x_1, x_2, \ldots, x_n$, where each variable x_i has a finite domain D. The Binary Permutation Constraints between pairs of variables are represented explicitly as allowed pairs of variable-value combinations.

Recursive search tree. At each step of the search, when a variable x_i is assigned a specific value a, the algorithm examines all Binary Permutation Constraints involving x_i and prunes inconsistent values from the domains of connected variables. Crucially, once a value b is removed from a domain during a particular recursive branch, it cannot be reintroduced later along the same search path. Thus, each value removal occurs at most once per variable along any single path in the recursion tree.

The worst-case recursion tree has depth at most n, since each recursive level corresponds to assigning a value to one additional variable. Without pruning, the naive number of branches at each level would be D, leading to a brute-force complexity of $O(D^n)$.

However, pruning has a significant effect on both the breadth and depth of the recursion tree. The process can be conceptualized through a state-transition graph, where each state is characterized by:

- The current set of variable assignments.
- The remaining domain values for each unassigned variable.

Transitions between states correspond to selecting a value for an unassigned variable and propagating the resulting domain reductions. To analyze progress,

one can define the metric T as the sum of remaining domain sizes across all unassigned variables:

$$T = \sum_{x_i \text{ unassigned}} |D(x)|$$

Each state transition strictly decreases T, since assigning a value to a variable and propagating constraints reduces at least one domain size. Initially, $T \leq n \times D$. Since T decreases monotonically and cannot become negative, the number of transitions along any single path in the search is bounded by $n \times D$.

Consequently, although a naive analysis of the recursion tree suggests an exponential upper bound of $O(D^n)$, this is only a loose bound on the number of explored states. A more refined analysis based on the strictly decreasing T shows that the number of effective state transitions along any search path is in fact bounded by $O(n \times D)$, leading to a polynomial upper bound on the total running time. Thus, while the underlying recursion tree has exponential size in theory, the pruning induced ensures that the actual running time of the algorithm is polynomial.

Total running time. The total running time of the algorithm can be analyzed by combining the bound on the number of state transitions and the cost per transition. As shown, the total number of distinct state transitions that occur during the entire execution of the algorithm is bounded by $O(nD)$.

At each state transition, the algorithm performs an assignment and propagates its effects to the domains of neighboring variables. This propagation involves examining all Binary Permutation Constraints connected to the currently assigned variable, which requires at most $O(n)$ time per transition.

Combining these two observations, the total running time of the algorithm is thus $O(nD)$ transitions $\times O(n)$ time per transition, yielding an overall bound of $O(n^2 D)$.

Chapter 3

RANDOMIZED ALGORITHMS

This chapter constitutes the main focus of the thesis and presents its most original contributions.

Following the analysis of deterministic algorithms, this section explores the use of randomized algorithms for solving CSPs. Randomized approaches introduce probabilistic elements into the algorithmic process, typically offering different trade-offs in terms of simplicity, performance, and guarantees compared to deterministic methods.

For the purpose of this thesis, a randomized algorithm is considered valid if its expected solving time is finite (i.e., strictly less than infinite) even if it may not always succeed within a fixed number of steps. Randomized algorithms can be broadly classified into two main categories [7, 11]:

- **Monte Carlo Algorithms:** These algorithms are noted for their typically quick execution times but carry a risk of incorrect outputs. Specifically, they may not find an existing solution to a problem even though one is indeed present. The defining feature of Monte Carlo algorithms is that they always operate within a predetermined bounded timeframe but are subject to a probability of producing an erroneous result.

- **Las Vegas Algorithms:** Conversely, Las Vegas algorithms guarantee the correctness of the output, with the variability affecting only their execution time. The performance of these algorithms is probabilistically fast but their primary characteristic is that they do not fail in finding a response if there exist such one.

Randomized algorithms often provide practical advantages for solving large or complex CSP instances by enabling efficient exploration of the solution space or by escaping local minima that might hinder deterministic approaches. The algorithms presented in this section are designed with this perspective in mind.

In parallel with the previous chapter, the following subsections will present randomized algorithms tailored for three problem classes: 2-SAT, 2-Fan CSP, and Binary Permutation CSP. For each case, the algorithm will be described and analyzed in terms of its behavior, correctness, and expected complexity. All these algorithms fall under the Monte Carlo class: they may return failure (i.e., not finding a solution) even if one exists, although their expected running time is finite and the probability of failure can be made arbitrarily small by increasing the number of iterations.

3.1 2-SAT

As previously discussed, the 2-SAT problem constitutes a restricted version of the Boolean satisfiability problem where each clause contains at most two literals. While this problem is solvable deterministically in linear time, it also serves as a classical example for exploring randomized algorithmic techniques.

This section presents an analysis of Papadimitriou's randomized algorithm for solving 2-SAT [12]. The algorithm exemplifies a Monte Carlo method, where the runtime is bounded, but there is a small probability of failing to find a satisfying assignment even when one exists. However, this probability can be made arbitrarily small through repeated trials. Importantly, the algorithm is one-sided in its error: if it returns a satisfying assignment, then the instance is guaranteed to be satisfiable. The only source of error lies in potentially reporting failure when a solution does exist.

Papadimitriou's algorithm is particularly notable for its conceptual simplicity and pedagogical value: it performs a random walk in the space of possible variable assignments, guided by unsatisfied clauses. Its analysis provides key insights into the behavior and guarantees of randomized algorithms, which will be relevant when designing similar strategies for more complex CSP variants.

Algorithm Description (Algorithm 6)

Papadimitriou's algorithm begins by assigning each variable in the 2-SAT instance a random initial truth value. The algorithm then proceeds in iterations, repeatedly modifying the current assignment to move toward a satisfying solution.

At each iteration, if the current assignment satisfies all clauses of the formula, the algorithm terminates and returns the solution. If any clause is unsatisfied, the algorithm selects one such clause uniformly at random. It then flips the truth value of one of the two variables in this clause, choosing the variable uniformly at random. This flipping operation is intended to resolve the conflict caused by the unsatisfied clause, thereby moving the assignment closer to a satisfying one.

The process continues for a fixed number of iterations depending on n (the number of variables), which is set to ensure a high probability of success. If no satisfying assignment is found after this threshold is reached, the algorithm reports failure.

Algorithm 6 Papadimitriou's Algorithm for 2-SAT

Input: A 2-SAT formula with n variables and m clauses.
Output: A satisfying assignment, or a determination of unsatisfiability.

procedure PAPADIMITRIOU2SATSOLVER
Assign each variable a random `true`/`false` value.
while iteration threshold not reached **do**
  if current assignment satisfies all clauses **then**
    return "Satisfiable" with the current assignment.
  end if
  Select an unsatisfied clause uniformly at random.
  With probability $\frac{1}{2}$:
    Flip the value of the first variable in the chosen clause (i.e. if it was assigned to `true`, it should now be reassigned to `false`; and viceversa)
  Else:
    Flip the value of the second variable in the chosen clause
end while
return "Unsatisfiable".

Analytical Framework

The complexity of Papadimitriou's randomized algorithm for 2-SAT can be formally analyzed by modeling its behavior as a random walk over a discrete state space. Each state in this process corresponds to the number of variables that are currently assigned correctly with respect to a fixed satisfying assignment.

Let's define:

$$X_i := \text{Number of steps to reach state } n \text{ from state } i$$

Here, state i denotes the number of variables currently assigned correctly, relative to a fixed (though unknown) satisfying assignment. The target state is n, which represents the fully correct assignment where all n variables match the satisfying assignment and the 2-SAT formula is solved.

Two boundary conditions characterize this process:

- $X_n = 0$ (absorbing state), since the assignment would already be in the desired state (valid assignment).

- $X_0 = 1 + X_1$; from state 0, the only move is to state 1. This is because the algorithm always flips a literal in an unsatisfied clause, which will improve the current assignment with probability 1).

Papadimitriou's algorithm proceeds by selecting a clause that is unsatisfied under the current assignment and flipping one of its literals uniformly at random; which is equivalent to a Markov chain with states $0, 1, 2, \ldots, n$. Under the assumption that the formula is satisfiable, there exists at least one satisfying assignment, which is fixed for the purpose of this analysis.

At any step, if the current assignment agrees with the satisfying assignment on i variables, the expected behavior is as follows:

- With probability at least $\frac{1}{2}$, flipping a variable increases the number of correctly assigned variables (transition to state $i + 1$)
- With probability at most $\frac{1}{2}$, the number of correct assignments decreases (transition to state $i - 1$).

This behavior can be viewed as a symmetric random walk with a reflecting boundary at state 0 and an absorbing boundary at state n.

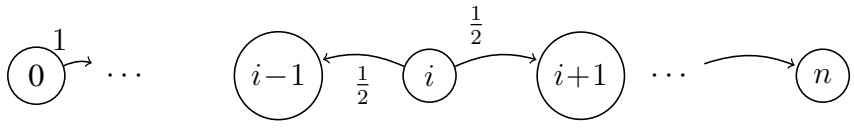

Figure 3.1: Markov chain model of the state space explored by Papadimitriou's 2-SAT algorithm.

Algorithm Complexity

Now, consider the number of steps to reach state n starting from state i in the previous defined random walk. Classical results on symmetric random walks show that the expected hitting time for such a walk is:

$$\max_{0 \leq i \leq n} \mathbb{E}[X_i] = O(n^2)$$

That is, the maximum expected number of steps to reach the absorbing state n (i.e., solving the 2-SAT formula) is upper bounded by n^2.

Let's proceed to formally bound the expected number of steps needed to reach the target state n in the random walk. As defined, let X_i denote the expected number of steps from state i to state n, where $i \in \{0, 1, \ldots, n\}$. Recall:

$$\mathbb{E}[X_n] = 0 \quad \text{(absorbing state)} \quad \text{and} \quad \mathbb{E}[X_0] = 1 + \mathbb{E}[X_1]$$

For general $i \in \{1, 2, \ldots, n-1\}$, the recurrence can be written:

$$\mathbb{E}[X_i] \leq 1 + \frac{1}{2}\mathbb{E}[X_{i-1}] + \frac{1}{2}\mathbb{E}[X_{i+1}]$$

The inequality accounts for the fact that in practice, the probability of improving the assignment can exceed $\frac{1}{2}$, but this bound suffices for worst-case analysis.

From the previous inequality, an upper bound on $\mathbb{E}[X_i]$ can be established by induction:

$$\mathbb{E}[X_i] \leq \mathbb{E}[X_{i+1}] + 2i + 1$$

Proof by induction on i:

- *Base case.* For $i = 0$,

$$\mathbb{E}[X_0] = 1 + \mathbb{E}[X_1] \leq \mathbb{E}[X_1] + 1 = \mathbb{E}[X_1] + 2 \cdot 0 + 1$$

- *Inductive step.* Assume the induction hypothesis holds for $i-1$, i.e., $\mathbb{E}[X_{i-1}] \leq \mathbb{E}[X_i] + 2(i-1) + 1$. The goal is to show: $\mathbb{E}[X_i] \leq \mathbb{E}[X_{i+1}] + 2i + 1$.

  Starting from the recurrence:

$$\mathbb{E}[X_i] \leq 1 + \frac{1}{2}\mathbb{E}[X_{i-1}] + \frac{1}{2}\mathbb{E}[X_{i+1}]$$

  Apply the inductive hypothesis:

$$\begin{aligned}
\mathbb{E}[X_i] &\leq 1 + \frac{1}{2}\left(\mathbb{E}[X_i] + 2(i-1) + 1\right) + \frac{1}{2}\mathbb{E}[X_{i+1}] \\
&\leq 1 + \frac{1}{2}\mathbb{E}[X_i] + (i-1) + \frac{1}{2} + \frac{1}{2}\mathbb{E}[X_{i+1}] \\
&\leq \frac{1}{2}\mathbb{E}[X_i] + \frac{3}{2} + (i-1) + \frac{1}{2}\mathbb{E}[X_{i+1}] \\
&\leq \frac{1}{2}\mathbb{E}[X_i] + (i + \frac{1}{2}) + \frac{1}{2}\mathbb{E}[X_{i+1}]
\end{aligned}$$

  Now subtract $\frac{1}{2}\mathbb{E}[X_i]$ from both sides:

$$\frac{1}{2}\mathbb{E}[X_i] \leq \left(i + \frac{1}{2}\right) + \frac{1}{2}\mathbb{E}[X_{i+1}]$$

Multiply both sides by 2:

$$\mathbb{E}[X_i] \leq \mathbb{E}[X_{i+1}] + 2i + 1$$

Thus, the inductive step holds for all i.

Finally, by summing the recurrence over $i = 0, 1, \ldots, n-1$, we obtain:

$$\mathbb{E}[X_0] \leq \sum_{i=0}^{n-1}(2i+1) = \frac{n(1+2n-1)}{2} = n^2$$

Hence, the expected number of steps required to solve a satisfiable 2-SAT instance using Papadimitriou's algorithm is upper bounded by n^2.

Algorithm Success Probability

Having established that the expected number of steps required by Papadimitriou's algorithm to solve a satisfiable 2-SAT instance is at most $O(n^2)$, the next goal is to quantify the probability of success within a finite number of steps. Specifically, it is desirable to show that the algorithm succeeds with probability at least $1 - \delta$ after performing $t = 2n^2\lceil \log_2(1/\delta) \rceil$ steps.

This guarantee is obtained by partitioning the total number of steps into several independent trials, each consisting of $2n^2$ steps. Let the number of such trials be:

$$t' := \lceil \log_2 \left(\frac{1}{\delta} \right) \rceil$$

Let A_i denote the event that trial i fails to find a satisfying assignment. The probability that *all* trials fail is given by:

$$\mathbb{P}\left(\bigcap_{i=1}^{t'} A_i \right) = \prod_{i=1}^{t'} \mathbb{P}(A_i)$$

Now, since the expected number of steps to find a satisfying assignment is $\mathbb{E}[X] \leq n^2$, Markov's inequality can be used to bound the failure probability of a single trial of $2n^2$ steps:

$$\mathbb{P}(X \geq 2n^2) \leq \frac{\mathbb{E}[X]}{2n^2} \leq \frac{n^2}{2n^2} = \frac{1}{2}$$

Therefore, the probability that all t' trials fail is at most:

$$\left(\frac{1}{2} \right)^{t'} = \frac{1}{2^{\lceil \log_2(1/\delta) \rceil}} = \delta$$

Consequently, the probability that *at least one* trial succeeds is at least:

$$1 - \mathbb{P}(\text{all trials fail}) \geq 1 - \delta$$

Hence, if the Papadimitriou's algorithm is runned for $2n^2\lceil \log_2(1/\delta) \rceil$ steps, then with probability at least $1 - \delta$, the algorithm finds a satisfying assignment.

This proves that Papadimitriou's algorithm is not only expected to run in polynomial time, but also succeeds with high probability in polynomially many steps.

3.2 2-Fan CSP

Following the analysis of Papadimitriou's randomized approach for 2-SAT, the attention now turns to designing an analogous randomized strategy for 2-Fan CSPs. The structure of 2-Fan Constraints lends itself naturally to a simple probabilistic correction mechanism: each constraint can be satisfied by forcing either variable involved to take its associated "wild" value. This observation motivates a randomized local search procedure that repeatedly attempts to correct unsatisfied constraints by selectively applying such corrections.

The randomized algorithm for 2-Fan CSP operates by iteratively refining an assignment of variable values. Unlike the deterministic approaches explored in previous sections, this method explicitly embraces randomness both in the selection of which constraint to correct and in deciding how to perform the correction.

Algorithm Description (Algorithm 7)

The procedure begins by initializing a random assignment of values to all variables. Since 2-Fan Constraints require at least one variable in each pair to assume its *wild value*, many constraints will typically be unsatisfied under this initial random configuration. Recall that in a 2-Fan Constraint between variables x_i and x_j, the wild values $w_i \in D_i$ and $w_j \in D_j$ are special values such that the constraint is satisfied if either $x_i = w_i$ or $\mathrm{x}_j = w_j$ (or both). This simple structure enables an intuitive local repair strategy.

At each iteration, the algorithm selects an unsatisfied constraint C_{ij} to address. Given the selected constraint C_{ij}, which connects variables x_i and x_j, the algorithm then chooses how to resolve the violation through a randomized decision. With probability $\frac{1}{2}$, the value of x_i is set to its wild value w_i, immediately satisfying the constraint. With complementary probability $\frac{1}{2}$, the algorithm instead forces x_j to its wild value w_j.

This correction mechanism has two desirable properties. First, it guarantees that at least one unsatisfied constraint is corrected in each iteration. Second, the

stochastic choice prevents the search from becoming trapped in rigid, deterministic update patterns that might cause cycling or local stagnation.

Naturally, forcing one variable to its wild value may cause other constraints to become unsatisfied, as 2-Fan Constraints introduce global dependencies through shared variables. The process thus performs a random walk over the space of assignments, with each step locally increasing the likelihood of global consistency.

The search continues iteratively until either a fully satisfying assignment is discovered, or a predetermined iteration threshold is reached, in which case the algorithm halts and reports failure to find a solution.

The simplicity of this randomized approach makes it both flexible and easy to implement. Moreover, as with Papadimitriou's algorithm for 2-SAT, it offers a promising framework for probabilistic analysis, which will be explored in the following sections.

Algorithm 7 Randomized 2-Fan CSP Solver

Input: A set of 2-Fan Constraints between pairs of variables.
Output: A complete assignment satisfying all constraints, or a determination that no solution exists.

procedure RANDOMIZEDSOLVER2FANCSP
Initialize a random assignment of values to all variables.
while iteration threshold not reached **do**
  if current assignment satisfies all 2-Fan Constraints **then**
    return current assignment.
  end if
  Select an unsatisfied 2-Fan Constraint C_{ij} at random.
  With probability $\frac{1}{2}$:
    Set $x_i \leftarrow w_i$ (x_i's "wild" value).
  With probability $\frac{1}{2}$:
    Set $x_j \leftarrow w_j$.
end while
return âUnsatisfiableâ.

Analytical Framework

The behaviour of the RANDOMIZEDSOLVER2FANCSP can be analysed, in close analogy to Papadimitriouâs treatment of 2-SAT, by viewing the algorithm as a random walk on a one-dimensional state space. In the present analysis, it is assumed that variable domains have size 3. This simplifies the structure of transitions, and can be later adapted to larger domain sizes.

As in the 2-SAT analysis, the idea is to *fix* an arbitrary satisfying assignment and analyze how the algorithm progresses toward this target. It is important to acknowledge that there may exist multiple satisfying assignments, but for the purpose of analysis, the current assignment is always compared to one such fixed reference assignment. Let:

$$X_i := \text{expected number of steps to reach the target state } n \text{ from state } i$$

State 0 corresponds to an assignment in which none of the variables is currently assigned the correct value (relative to the fixed reference); state n is absorbing, once all variables match their correct values (i.e., each variable is in the value assigned to it in the fixed satisfying assignment), the formula is trivially satisfied and the algorithm halts.

At each iteration, the algorithm selects an unsatisfied constraint C_{ij} uniformly at random and forces one of the two variables to its wild value, each with probability $\frac{1}{2}$. The impact of this update on the current state depends on both the current assignment and the structure of the reference assignment. Three distinct transition patterns are identified:

Case A (Guaranteed improvement). The selected constraint C_{ij} is such that neither x_i nor x_j is currently assigned its correct value. Furthermore, the correct assignment (in the reference solution) for this constraint is that *both* variables should take their wild values.

In this case, forcing either variable to its wild value always results in gaining one additional correctly assigned variable (progress toward state i+1). Therefore:

$$\mathbb{E}[X_i] \;\leq\; 1 + \mathbb{E}[X_{i+1}] \tag{2Fan.A}$$

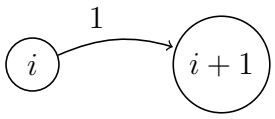

Figure 3.2: Transition structure (2Fan.A).

Case B (Symmetric step). The selected constraint C_{ij} is such that exactly *one* of the variables is currently assigned its correct value, while the reference assignment prescribes that this variable remains and the other variable should be set to its wild value.

In this case, the random flip has the following behavior:

- With probability $\frac{1}{2}$, the already correct variable is flipped (incorrectly), causing a loss of one correct variable (transition to $i - 1$).

- With probability $\frac{1}{2}$, the other variable is set to its wild value (transition to $i+1$).

Thus, the expected progress satisfies:

$$\mathbb{E}[X_i] \;\leq\; 1 + \tfrac{1}{2}\,\mathbb{E}[X_{i-1}] + \tfrac{1}{2}\,\mathbb{E}[X_{i+1}] \tag{2Fan.B}$$

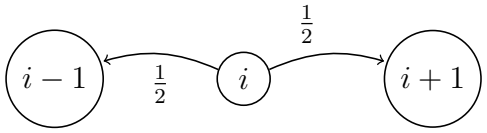

Figure 3.3: Transition structure (2Fan.B).

Case C (No loss, possible stagnation). The selected constraint C_{ij} is such that neither x_i nor x_j is currently assigned its correct value. However, in the reference assignment, only one of the two variables is required to be wild, and the other variable is meant to hold a non-wild value.

In this situation:

- With probability $\frac{1}{2}$, the algorithm selects and correctly updates the variable that should be wild (progress to $i+1$).
- With probability $\frac{1}{2}$, the other variable is set to its wild value, which does not help toward reaching the reference state (state remains at i).

The resulting expected progress satisfies:

$$\mathbb{E}[X_i] \;\leq\; 1 + \tfrac{1}{2}\,\mathbb{E}[X_i] + \tfrac{1}{2}\,\mathbb{E}[X_{i+1}] \tag{2Fan.C}$$

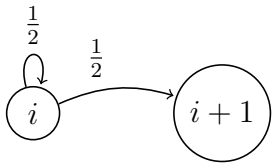

Figure 3.4: Transition structure (2Fan.C).

These three cases capture all possible behaviors of the algorithm in terms of state transitions during its execution. The subsequent analysis will show that for each case, an inequality of the form

$$\mathbb{E}[X_i] \leq \mathbb{E}[X_{i+1}] + ai + b$$

can be established, with suitable choices of a and b, ensuring that the total expected number of steps is polynomial in n. This will imply that the randomized algorithm solves any satisfiable instance of the 2-Fan CSP in expected polynomial time.

Algorithm Complexity

Recall that if for each case (A, B, C), an inequality of the form:

$$\mathbb{E}[X_i] \leq \mathbb{E}[X_{i+1}] + ai + b$$

can be established, with suitable constants a and b, then it becomes possible to bound the overall expected number of steps of the algorithm. summing them over the entire state space $i = 0, 1, \ldots, n-1$ leads to a total expected runtime that is polynomial in n.

Case A Analysis Taking inequality (2Fan.A), where $\mathbb{E}[X_i] \leq 1 + \mathbb{E}[X_{i+1}]$, the goal is to express this inequality in the form: $\mathbb{E}[X_i] \leq \mathbb{E}[X_{i+1}] + ai + b$

It is clear that this is equivalent to: $1 \leq ai + b$. This condition must hold for all $i \geq 0$. In particular:

- For $i = 0$, the inequality becomes $1 \leq b$, which requires $b \geq 1$.
- For $i \geq 1$, the inequality becomes $1 \leq ai + b$, which is automatically satisfied if $b \geq 1$ and $a \geq 0$.

Thus, any choice of $a \geq 0$ and $b \geq 1$ satisfies the required condition for Case A.

Case B Analysis Consider inequality (2Fan.B). Let's proceed by assuming that the inductive hypothesis holds at states $i-1$ and i, that is: $\mathbb{E}[X_{i-1}] \leq \mathbb{E}[X_i] + a(i-1) + b$

Remark: It is important to note that Case B only applies when $i \geq 1$, since this case corresponds to configurations where *at least one* variable is correctly assigned. At $i = 0$, the algorithm transitions according to Case (2Fan.C), so no issue arises regarding the undefined value $\mathbb{E}[X_{-1}]$. This clarifies that the following analysis for Case B can safely assume $i \geq 1$.

Now substitute this inequality into the right-hand side of (2Fan.B):

$$\begin{aligned}\mathbb{E}[X_i] &\leq 1 + \tfrac{1}{2}\left(\mathbb{E}[X_i] + a(i-1) + b\right) + \tfrac{1}{2}\mathbb{E}[X_{i+1}] \\ &= 1 + \tfrac{1}{2}\mathbb{E}[X_i] + \tfrac{1}{2}a(i-1) + \tfrac{1}{2}b + \tfrac{1}{2}\mathbb{E}[X_{i+1}]\end{aligned}$$

Simplify further:

$$\tfrac{1}{2}\mathbb{E}[X_i] \leq 1 + \tfrac{1}{2}a(i-1) + \tfrac{1}{2}b + \tfrac{1}{2}\mathbb{E}[X_{i+1}]$$

$$\mathbb{E}[X_i] \leq \mathbb{E}[X_{i+1}] + a(i-1) + b + 2$$

$$\mathbb{E}[X_i] \leq \mathbb{E}[X_{i+1}] + ai + (b + 2 - a)$$

Hence, it implies that

$$ai + (b + 2 - a) \leq ai + b$$

$$2 \leq a$$

To ensure this holds for all $i \geq 0$, it suffices to ensure $a \geq 2$.

Case C Analysis Starting from inequality (2Fan.C), isolate $\mathbb{E}[X_i]$ on one side, subtract $\frac{1}{2}\mathbb{E}[X_i]$ from both sides and multiply by 2 to get:

$$\mathbb{E}[X_i] \leq 2 + \mathbb{E}[X_{i+1}]$$

It implies that:

$$2 \leq ai + b$$

In particular:

- For $i = 0$, this requires $2 \leq b$, so $b \geq 2$.
- For $i \geq 1$, the inequality is automatically satisfied if $b \geq 2$ and $a \geq 0$.

Final Conclusion of the Algorithm Complexity Analysis Summarizing the results of the three cases:

- Case A requires: $1 \leq ai + b$
- Case B requires: $2 \leq a$
- Case C requires: $2 \leq ai + b$

To ensure that a common pair of constants (a, b) satisfies all three inequalities simultaneously for all $i \geq 0$, it suffices to find values such that the most restrictive inequalities are satisfied. These are $2 \leq ai + b$ and $2 \leq a$. Thus, the choice $a \geq 2$ and $b \geq 2$ satisfies the inequalities for all three cases.

Therefore, the expected number of steps to reach a satisfying assignment is bounded by $O(n^2)$. This establishes that the RANDOMIZEDSOLVER2FANCSP algorithm solves any satisfiable 2-Fan CSP instance in expected polynomial time.

3.3 Binary Permutation CSP

Inspired by the success of randomized algorithms in the simpler 2-SAT setting, a randomized strategy is proposed to navigate the space of assignments for Binary Permutation CSPs. The goal is to progressively move toward satisfying all Binary Permutation Constraints through probabilistic local adjustments, starting from an arbitrary initial assignment.

Algorithm Description (Algorithm 8)

The randomized algorithm for solving Binary Permutation CSP follows the same general pattern as the randomized algorithms presented for 2-SAT and 2-Fan CSP. It adopts an iterative improvement strategy: starting from an arbitrary initial assignment, the algorithm progressively refines this assignment through local randomized corrections. The key idea is to leverage the structure of Binary Permutation Constraints to guide updates, with the aim of converging toward a globally valid solution.

The process begins by generating an initial arbitrary assignment of values to all variables. Given the highly constrained nature of Binary Permutation CSPs, this initial assignment is unlikely to satisfy all constraints. However, it serves as a useful starting point for a guided stochastic search.

At each iteration, the algorithm examines the current assignment and identifies Binary Permutation Constraints that are currently unsatisfied. The algorithm then selects one of these unsatisfied constraints (the selection strategy can be arbitrary), thereby focusing on correcting one inconsistency at a time.

Once an unsatisfied constraint C_{ij} is selected, the algorithm applies one of two update strategies, controlled by a parameter p. With probability p, a targeted local correction is performed: one of the two variables involved in the constraint is chosen uniformly at random, and the other variable is updated to match the value prescribed by the bijection π defining the Binary Permutation Constraint. In other words, the constraint is enforced explicitly for this pair of variables, thereby correcting one inconsistency.

With probability $1-p$, a more exploratory update is applied: a new pair of values $(v_i, v_j) \in C_{ij}$ is selected uniformly at random, under the additional condition that neither v_i nor v_j matches the current assignment of the corresponding variable. This mechanism serves to introduce controlled randomness into the search process, enabling the algorithm to escape from local minima and explore alternative regions of the solution space.

The algorithm iteratively applies these corrective and exploratory updates until either a satisfying assignment is found (i.e., all Binary Permutation Constraints are simultaneously satisfied), or a predefined iteration threshold is reached. If no

solution is found within the allotted number of iterations, the algorithm terminates with a failure message.

Algorithm 8 Randomized Binary Permutation CSP Solver

Input: A set of Binary Permutation Constraints between pairs of variables.
Output: A complete assignment satisfying all constraints, or a determination that no solution exists.

procedure RANDOMIZEDSOLVERBINARYPERMUTATIONCSP
Initialize a random assignment of values to all variables.
while iteration threshold not reached **do**
  if current assignment satisfies all Binary Permutation Constraints **then**
    return current assignment.
  end if
  Select an unsatisfied Binary Permutation Constraint C_{ij} at random.
  With probability p:
    Select one of x_i or x_j uniformly at random.
    Update the other variable's value to match the corresponding value prescribed by the constraint.
  With probability $1 - p$:
    Select a value pair $(v_i, v_j) \in C_{ij}$ such that $v_i \neq$ current(x_i) and $v_j \neq$ current(x_j).
    Update $x_i \leftarrow v_i, x_j \leftarrow v_j$.
end while
return "Unsatisfiable"

Analytical Framework

The behaviour of the RANDOMIZEDSOLVERBINARYPERMUTATIONCSP can be also analysed, in close analogy to Papadimitriou's treatment of 2-SAT and the analysis developed for 2-Fan CSPs. In the present analysis, it is assumed that variable domains have size 3.

As in the previous analysis, the idea is to *fix* an arbitrary satisfying assignment and analyze how the algorithm progresses toward this target. It is important to acknowledge that there may exist multiple satisfying assignments, but for the purpose of analysis, the current assignment is always compared to one such fixed reference assignment.

The impact of the algorithm's updates on the current state depends on the current assignment relative to the reference assignment. Two distinct transition patterns arise:

Case A (Symmetric step). The selected constraint C_{ij} is such that one of the two variables is currently assigned its correct value, while the other variable is not. The reference assignment prescribes that the correctly assigned variable should remain unchanged, and the other variable should be updated to its corresponding correct value via the bijection π.

In this situation, the algorithm's behaviour decomposes as follows:

- With probability $\frac{1}{2}p$, the correct variable is selected for the update, leading to an incorrect flip (transition to $i-1$).
- With probability $\frac{1}{2}p$, the incorrect variable is selected and updated correctly (transition to $i+1$).
- With probability $1-p$, an exploratory update is applied, which does not change the count of correct variables (remaining at i).

This yields the following recurrence:

$$\mathbb{E}[X_i] \leq 1 + \tfrac{1}{2}p\mathbb{E}[X_{i-1}] + \tfrac{1}{2}p\mathbb{E}[X_{i+1}] + (1-p)\mathbb{E}[X_i] \qquad \text{(BPerm.A)}$$

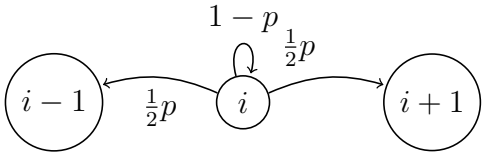

Figure 3.5: Transition structure (BPerm.A).

Case B (Jump step). The selected constraint C_{ij} is such that neither x_i nor x_j is currently assigned its correct value. In the reference assignment, both variables should take values that differ from their current assignment.

In this case:

- With probability p, a local correction is performed by updating one variable according to π, which does not result in progress toward the reference assignment, leaving the state unchanged (remaining at i).
- With probability $1-p$, an exploratory update is performed: a new pair $(v_i, v_j) \in C_{ij}$ is selected uniformly at random, under the condition that both v_i and v_j differ from their current assignments. This update can cause a "jump" to state $i+2$, because both variables simultaneously match the reference assignment if the correct pair is selected.

This leads to the following recurrence:

$$\mathbb{E}[X_i] \leq 1 + p\mathbb{E}[X_i] + (1-p)\mathbb{E}[X_{i+2}] \qquad \text{(BPerm.B)}$$

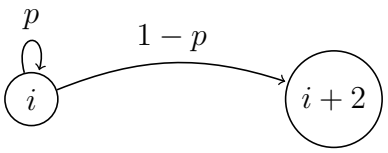

Figure 3.6: Transition structure (BPerm.B).

Algorithm Complexity

Recall that if for each case (A, B), an inequality of the form:

$$\mathbb{E}[X_i] \leq \mathbb{E}[X_{i+1}] + ai + b$$

can be established, with suitable constants a and b. Once such inequalities are derived, they can be telescoped over the entire state space $i = 0, 1, \ldots, n-1$, which implies that the algorithm has expected polynomial runtime.

Case A Analysis Consider inequality (BPerm.A). Proceed by isolating terms involving $\mathbb{E}[X_i]$, and move the $(1-p)\mathbb{E}[X_i]$ term to the left-hand side:

$$\begin{aligned} \mathbb{E}[X_i] - (1-p)\mathbb{E}[X_i] &\leq 1 + \tfrac{1}{2}p\mathbb{E}[X_{i-1}] + \tfrac{1}{2}p\mathbb{E}[X_{i+1}] \\ p\mathbb{E}[X_i] &\leq 1 + \tfrac{1}{2}p\mathbb{E}[X_{i-1}] + \tfrac{1}{2}p\mathbb{E}[X_{i+1}] \end{aligned}$$

Simplify:

$$\mathbb{E}[X_i] \leq \frac{1}{p} + \tfrac{1}{2}\mathbb{E}[X_{i-1}] + \tfrac{1}{2}\mathbb{E}[X_{i+1}]$$

$$\mathbb{E}[X_i] \leq \frac{1}{p} + \tfrac{1}{2}\left(\mathbb{E}[X_i] + a(i-1) + b\right) + \tfrac{1}{2}\mathbb{E}[X_{i+1}]$$

$$\begin{aligned} \mathbb{E}[X_i] - \tfrac{1}{2}\mathbb{E}[X_i] &\leq \frac{1}{p} + \tfrac{1}{2}\left(a(i-1) + b\right) + \tfrac{1}{2}\mathbb{E}[X_{i+1}] \\ \tfrac{1}{2}\mathbb{E}[X_i] &\leq \frac{1}{p} + \tfrac{1}{2}a(i-1) + \tfrac{1}{2}b + \tfrac{1}{2}\mathbb{E}[X_{i+1}] \end{aligned}$$

$$\mathbb{E}[X_i] \leq \mathbb{E}[X_{i+1}] + \frac{2}{p} + a(i-1) + b$$

It implies that:

$$\frac{2}{p} + a(i-1) + b \leq ai + b$$

$$\frac{2}{p} \leq a$$

When $i = 0$, by the structure of the algorithm:

- With probability $1 - p$, the algorithm remains in state $i = 0$.
- With probability p, it moves to state $i = 1$.

Therefore, the recurrence at $i = 0$ is: $\mathbb{E}[X_0] \leq 1 + (1 - p)\mathbb{E}[X_0] + p\mathbb{E}[X_1]$. Rearranging it's obtained that: $\mathbb{E}[X_0] \leq \frac{1}{p} + \mathbb{E}[X_1]$. This corresponds to the inductive form $\mathbb{E}[X_0] \leq \mathbb{E}[X_1] + b$, provided that $b \geq \frac{1}{p}$.

Case B Analysis Consider inequality (BPerm.B). Isolate $\mathbb{E}[X_i]$ and divide both sides by $1 - p$:

$$(1 - p)\mathbb{E}[X_i] \leq 1 + (1 - p)\mathbb{E}[X_{i+2}]$$

$$\mathbb{E}[X_i] \leq \mathbb{E}[X_{i+2}] + \frac{1}{1 - p}$$

Now it can be observed that $\mathbb{E}[X_{i+2}] \leq \mathbb{E}[X_{i+1}]$ is satisfied always, by definition of the random walk, giving:

$$\mathbb{E}[X_i] \leq \mathbb{E}[X_{i+1}] + \frac{1}{1 - p}$$

It implies that:

$$\frac{1}{1 - p} \leq ai + b$$

Final Conclusion of the Algorithm Complexity Analysis At this point, three key inequalities have been established for the expected number of steps $\mathbb{E}[X_i]$ in the two transition cases of the algorithm:

$$\frac{2}{p} \leq a$$

$$\frac{1}{p} \leq b$$

$$\frac{1}{1 - p} \leq ai + b$$

If constants $a \geq 0, b \geq 0$ and $p \in [0, 1]$ can be found such that both inequalities hold for all $i \geq 0$, then the randomized Binary Permutation CSP solver is proven to operate in expected polynomial time.

Example 3.3.1. (Values for a, b, p) Take $p = \frac{1}{2}, a = 4, b = 2$

First consider the inequalities from Case A: $\frac{2}{p} = \frac{2}{\frac{1}{2}} = 4 \leq a = 4$; $\frac{1}{p} = \frac{1}{\frac{1}{2}} = 2 \leq b = 2$. Thus, the first two inequalities are satisfied.

Now consider the inequality from Case B: $\frac{1}{1-p} = \frac{1}{1-\frac{1}{2}} = 2 \leq ai + b = 4i + 2$. For $i \geq 0$, this inequality holds trivially, since $4i + 2 \geq 2$ for all $i \geq 0$.

Therefore, the choice $p = \frac{1}{2}$, $a = 4$, $b = 2$ satisfies both inequalities simultaneously.

3.4 Combination of 3 types of Constraint

After analyzing randomized solvers for individual classes of constraints (2-Fan, Binary Permutation), the next natural step is to consider CSP instances that may simultaneously include these two types of constraints and the Value Restricted ones. This setting arises in practical problems where different structural patterns coexist, requiring an algorithm that can handle arbitrary mixtures of these constraints.

Fortunately, the design of the randomized solvers for each individual class follows a common analytical framework: the algorithm performs local updates driven by unsatisfied constraints, and its expected progress toward a fixed reference solution can be modeled as a random walk over a one-dimensional state space. In each case, the analysis reduces to establishing inequalities of the form:

$$\mathbb{E}[X_i] \leq \mathbb{E}[X_{i+1}] + ai + b$$

where constants $a \geq 0$, $b \geq 0$ depend on the update rules for the particular constraint type.

Crucially, this analytical form is compatible across the three solvers. When combining all three constraint types in a single instance, the algorithm proceeds in the same manner: at each step, it selects an unsatisfied constraint uniformly at random (regardless of its type), and applies the corresponding update strategy. The effect on $\mathbb{E}[X_i]$ in each case is governed by the corresponding inequality:

- For 2-Fan CSPs: $\mathbb{E}[X_i] \leq \mathbb{E}[X_{i+1}] + a_{\text{2Fan}} i + b_{\text{2Fan}}$
- For Binary Permutation CSPs: $\mathbb{E}[X_i] \leq \mathbb{E}[X_{i+1}] + a_{\text{BPerm}} i + b_{\text{BPerm}}$
- For Value Restriction CSPs: $\mathbb{E}[X_i] \leq \mathbb{E}[X_{i+1}] + a_{\text{VR}} i + b_{\text{VR}}$

The random walk as a whole remains valid if a single pair of constants (a, b) can be chosen to simultaneously satisfy all three inequalities. In other words, one must select $a \geq \max a_{\text{2Fan}}, a_{\text{BPerm}}, a_{\text{VR}}$ and $b \geq \max b_{\text{2Fan}}, b_{\text{BPerm}}, b_{\text{VR}}$.

This establishes that the randomized solver operates in expected polynomial time even when faced with mixed CSP instances combining arbitrary numbers of 2-Fan, Binary Permutation, and Value Restriction constraints.

Algorithm Description (Algorithm 9)

Algorithm 9 Combined Randomized CSP Solver (2-Fan, Binary Permutation, Value Restriction)

Input: A set of variables and a set of constraints of types {2-Fan, Binary Permutation, Value Restriction}.
Output: A complete assignment satisfying all constraints, or a determination that no solution exists.

procedure COMBINEDRANDOMIZEDSOLVERCSP
Initialize a random assignment of values to all variables.
while iteration threshold not reached **do**
  if current assignment satisfies all constraints **then**
    return current assignment.
  end if
  Select an unsatisfied constraint C at random.
  if C is a Value Restriction constraint **then**
    Select a value v uniformly at random among the allowed values for that variable.
  else if C is a 2-Fan constraint **then**
    RANDOMIZEDSOLVER2FANCSP(C)
  else if C is a Binary Permutation constraint **then**
    RANDOMIZEDSOLVERBINARYPERMUTATIONCSP(C)
  end if
end while
return "Unsatisfiable"

Example 3.4.1. (Example of values for a, b that satisfy all cases) Take $p = \frac{1}{2}$ for the Binary Permutation CSP part. Then the following values work: $a = 4, b = 4$

These values satisfy the following:

- For the 2-Fan CSP, the required inequalities are of the form $1 \leq ai + b$ or$2 \leq a(i+1) + b$, which are satisfied by $a = 4, b = 4$.
- For the Binary Permutation CSP: $\frac{2}{p} = 4 \leq a = 4$; $\frac{1}{p} = 2 \leq b = 4$; $\frac{1}{1-p} = 2 \leq 4i + 4, \quad \forall i \geq 0$ which is always satisfied.

- For Value Restriction constraints, updates only guarantee improvement or neutral steps, so the same inequality form $\mathbb{E}[X_i] \leq \mathbb{E}[X_{i+1}] + ai + b$ can be satisfied with any $a \geq 0, b \geq 1$; thus a = 4, b = 4 works.

Thus, the combined randomized algorithm operates in expected polynomial time.

Chapter 4

CONCLUSIONS

4.1 Interpretation & Limitations

The main contribution of this thesis lies in the construction and analysis of the final randomized algorithm 9. This algorithm provides an efficient randomized procedure capable of solving Constraint Satisfaction Problems (CSPs) involving combinations of three key types of restricted constraints: Value Restriction Constraints, 2-Fan Constraints, and Binary Permutation Constraints.

This result connects directly with the broader theoretical landscape established in *Characterising Tractable Constraints* [5]. In particular, this set of CSPs is known to be maximally tractable; meaning that as long as one restricts to constraints within this set, the problem remains solvable in polynomial time. However, if one allows the addition of any constraint outside this set, the problem becomes NP-complete. Consequently, this thesis provides a complete randomized polynomial-time solution for the tractable fragment (that is, the class of CSPs solvable in polynomial time) identified, thereby covering all efficiently solvable combinations of these restricted constraint types. In this sense, the work positions itself naturally at the boundary between tractable and intractable CSPs.

It is important to highlight certain limitations of the present analysis. First, while the randomized algorithms analyzed guarantee polynomial expected runtime, they remain Monte Carlo in nature. That is, although they are designed to find a solution with high probability if one exists, they may fail to do so in a given execution. Moreover, this thesis does not include a detailed analysis of the success probability of the algorithms, nor an explicit derivation of how many repeated trials would be required to achieve a desired confidence level. This is left as a direction for possible future work.

Second, the analysis throughout this thesis has been conducted under the assumption that the domain size of variables is equal to 3. While the techniques

generalize conceptually to larger domains, the specific constants derived in the recurrence relations (e.g., in Section 3.2 and Section 3.3) depend quantitatively on this assumption. Extending the results to arbitrary fixed domain sizes would require a careful generalization of the analysis.

Third, the algorithm relies on the ability to select unsatisfied constraints and perform updates efficiently. It assumes that the set of unsatisfied constraints can be maintained and queried in constant or logarithmic time (or even linear), which is reasonable in practical implementations but implicitly assumes appropriate data structures.

Finally, the randomized analysis has been performed assuming that the input CSP instance lies entirely within the tractable class. The algorithm itself does not explicitly verify whether the input respects the required constraint forms (Value Restriction, 2-Fan, Binary Permutation), and correctness depends on this precondition being satisfied.

4.2 Future Work

The work presented in this thesis opens several natural directions for future research and refinement.

First, as noted in the previous section, this thesis does not include a detailed analysis of the success probability of the proposed randomized algorithms. While the expected runtime is shown to be polynomial, a formal bound on the number of trials required to achieve a given success probability would make the results more complete from a probabilistic algorithm analysis standpoint.

Second, the current analysis has been developed under the assumption that all variable domains have size 3. It would be valuable to extend the analysis to domains of arbitrary finite size. Such an extension could clarify how the dependency on domain size affects both the algorithm's design and its runtime guarantees.

Third, the algorithms presented focus exclusively on the three restricted constraint types analyzed (Value Restriction, 2-Fan, and Binary Permutation). An interesting line of future work would be to study whether similar randomized techniques can be adapted to other classes of CSPs, not necessarily including all of 0/1/all constraints. Or to explore how hybrid strategies combining deterministic and randomized components could perform in other tractable subclasses.

Finally, while this thesis has focused on theoretical analysis, an important direction for future work is to implement and experimentally test the algorithms on CSP instances in code. Running experiments would provide insights into how well the algorithms perform in practice; for example, how many steps they typically take to find a solution, and how different choices of parameters (such as the probability p in the randomized steps) affect performance. It would also be

interesting to explore whether adding variable selection heuristics (which are not needed to prove the theoretical results) could improve the algorithms' speed when applied to real-world problems.

Appendix A

MATHEMATICAL TOOLS

The analysis and discussion of several algorithms presented in this thesis require the introduction of several essential mathematical tools.

Markov's Inequality: Let X be a non-negative random variable, and let $a > 0$. Then Markov's inequality states that:

$$\mathbb{P}(X \geq a) \leq \frac{\mathbb{E}[X]}{a}$$

Arithmetic Series Formula: The sum of the first n terms of an arithmetic sequence with first term a_1 and last term a_n is given by:

$$\sum_{i=1}^{n} a_i = \frac{n(a_1 + a_n)}{2}$$

In particular, the sum of the first n odd numbers (i.e., $1+3+5+\cdots+(2n-1)$) equals n^2, and the sum of the first n terms of the form $2i + 1$ can be simplified using this formula.

Logarithmic Identity: This follows directly from the definition of the logarithm base 2.:

$$\frac{1}{2^{\log_2(1/\delta)}} = \delta$$

Bibliography

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Reading, Mass.: Addison-Wesley, 1974.

[2] Bengt Aspvall, Michael F. Plass, and Robert Endre Tarjan. "A linear-time algorithm for testing the truth of certain quantified boolean formulas". In: *Information Processing Letters* 8.3 (1979), pp. 121–123. ISSN: 0020-0190. DOI: https://doi.org/10.1016/0020-0190(79)90002-4. URL: https://www.sciencedirect.com/science/article/pii/0020019079900024.

[3] ClÃ©ment Carbonnel, Martin C. Cooper, and Joao Marques-Silva. "Tractable Explaining of Multivariate Decision Trees". In: *Proceedings of the 20th International Conference on Principles of Knowledge Representation and Reasoning*. Aug. 2023, pp. 127–135. DOI: 10.24963/kr.2023/13. URL: https://doi.org/10.24963/kr.2023/13.

[4] David A. Cohen et al. "Tractable Classes of Binary CSPs Defined by Excluded Topological Minors". In: *IJCAI'15 : Proceedings of the 24th International Conference on Artificial Intelligence*. Ed. by Hong Kong University of Science Qiang Yang, Technology, and University of Oxford Michael Wooldridge. International Joint Conferences on Artificial Intelligence (IJCAI). Buenos Aires, Argentina: International Joint Conferences on Artificial Intelligence (IJCAI), July 2015, pp. 1945–1951. URL: https://hal.science/hal-04079418.

[5] Martin C. Cooper, David A. Cohen, and Peter G. Jeavons. "Characterising tractable constraints". In: *Artificial Intelligence* 65.2 (1994), pp. 347–361. ISSN: 0004-3702. DOI: https://doi.org/10.1016/0004-3702(94)90021-3. URL: https://www.sciencedirect.com/science/article/pii/0004370294900213.

[6] Martin C. Cooper and Guillaume Escamocher. "Characterising the complexity of constraint satisfaction problems defined by 2-constraint forbidden patterns". In: *Discrete Applied Mathematics* 184 (2015), pp. 89–113. ISSN: 0166-218X. DOI: https://doi.org/10.1016/j.dam.

2014.10.035. URL: https://www.sciencedirect.com/science/article/pii/S0166218X14004806.

[7] Sanjoy Dasgupta, Christos H. Papadimitriou, and Umesh V. Vazirani. *Algorithms*. McGraw-Hill, 2008. ISBN: 978-0-07-352340-8.

[8] Peter Jeavons, David Cohen, and Marc Gyssens. "Closure properties of constraints". In: *J. ACM* 44.4 (July 1997), 527â548. ISSN: 0004-5411. DOI: 10.1145/263867.263489. URL: https://doi.org/10.1145/263867.263489.

[9] Peter G. Jeavons and Martin C. Cooper. "Tractable constraints on ordered domains". In: *Artificial Intelligence* 79.2 (1995), pp. 327–339. ISSN: 0004-3702. DOI: https://doi.org/10.1016/0004-3702(95)00107-7. URL: https://www.sciencedirect.com/science/article/pii/0004370295001077.

[10] Melven R. Krom. "The Decision Problem for a Class of First-Order Formulas in Which all Disjunctions are Binary". In: *Zeitschrift fÃ¼r Mathematische Logik und Grundlagen der Mathematik* 13.1â2 (1967), 15â20. DOI: 10.1002/malq.19670130104.

[11] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[12] C.H. Papadimitriou. "On selecting a satisfying truth assignment". In: *[1991] Proceedings 32nd Annual Symposium of Foundations of Computer Science*. 1991, pp. 163–169. DOI: 10.1109/SFCS.1991.185365.

[13] Robert Tarjan. "Depth-First Search and Linear Graph Algorithms". In: *SIAM Journal on Computing* 1.2 (1972), pp. 146–160. DOI: 10.1137/0201010. eprint: https://doi.org/10.1137/0201010. URL: https://doi.org/10.1137/0201010.

[14] E. Tsang. *Foundations of Constraint Satisfaction*. Computation in cognitive science. Academic Press, 1993. ISBN: 9780127016108. URL: https://books.google.es/books?id=TnxQAAAAMAAJ.

[15] Yuanlin Zhang, Roland H. C. Yap, and Joxan Jaffar. "Functional elimination and 0/1/All constraints". In: *Proceedings of the Sixteenth National Conference on Artificial Intelligence and the Eleventh Innovative Applications of Artificial Intelligence Conference Innovative Applications of Artificial Intelligence*. AAAI '99/IAAI '99. Orlando, Florida, USA: American Association for Artificial Intelligence, 1999, 175â180. ISBN: 0262511061.