Modeling and Solving Problems Using Propositional Logic and SAT Solvers

Auteur : Aldeghi, Florian
Promoteur(s) : Fontaine, Pascal
Faculté : Faculté des Sciences appliquées
Diplôme : Master : ingénieur civil en informatique, à finalité spécialisée en "management"
Année académique : 2022-2023
URI/URL : http://hdl.handle.net/2268.2/17699

University of Liège - Faculty of Applied Sciences

Modeling and Solving Problems Using Propositional Logic and SAT Solvers

Dissertation carried out in partial fulfillment of the requirements for the degree of Master of Civil Engineering in Computer Science

by Aldeghi Florian

Academic supervisor :
Prof. Pascal Fontaine

Academic year 2022-2023

Abstract

This research focuses on problem-solving using logical models, aiming to efficiently solve a variety of well-known puzzles and games that are known for their challenging nature. The goal is to find ways to model the problem and formulate constraints in order to reduce the computation time required to find a solution.

The study expands beyond single-player puzzles and also explores two-player games. By analyzing and proposing novel solutions for different puzzle scenarios, this research explores a part of propositional logic applied to the resolution of games and puzzles.

Specifically, the study explores the application of logical model to the Snake Cube puzzle, demonstrating its effectiveness. Furthermore, the efforts have enabled addressing larger puzzle instances (4x4x4 snake cube) by exploiting the symmetry properties of the problem. Additionally, the research investigates different approaches to solving the 2x2 Rubik's Cube, with the objective of achieving reasonable solution times for the 3x3 Rubik's Cube. This work experiment problem-solving techniques within the domain of logic-based models and lay the foundation for further improvements and applications.

An analysis of the rules and scenarios of the game of Othello has been conducted to propose a model that allows the simulation of a game's progression. Othello is a strategic game that necessitates careful planning and decision-making. The research findings revealed the minimum number of moves required to reach the end of a game for different board size, which is nine for boards of size 5 to 8. An extension of the model using quantified Boolean formulae provides a straight path towards solving the game. While the game has already been solved for a 6x6 board, the model presented here serves as a scalable foundation. With improvements in writing constraints in future research, the finale objective is to solve the full-scale 8x8 game.

Keywords: SAT solver, Boolean variables, clauses, model, computation time, satisfiability, efficiency, quantifier.

Acknowledgements

First, I would like to express my gratitude to Prof. Pascal Fontaine, my supervisor, who took the time to work with me in finding a subject that aligned with my interests, who followed up my progress regularly and made sure I was on the right path, and who provided me with valuable guidance throughout this research, greatly contributing to its successful completion.

I extend my appreciation to ChatGPT, an AI language model by OpenAI, for its invaluable assistance and insightful suggestions during the development of this thesis. Its contributions have greatly enhanced the quality and clarity of my work.

I would also like to acknowledge the use of the website 'draw.io', which was instrumental in creating all the images used in this work.

Thank you to my girlfriend for being with me throughout the entire year, consistently checking in on my progress and providing encouragement in my work. Simply put, thank you for being there. Your support has meant everything to me.

Finally, I would like to express my heartfelt gratitude to my family and friends for being there for me and following the progress of my work, all while believing in me.

Contents

1 Introduction

In today's world, puzzles, two-player games, and brain teasers and games are found everywhere and are fascinating people from all walks of life. The Rubik's Cube, for instance, is a puzzle that has challenged countless individuals, leaving them contemplating over its solution. This research is driven by the motivation to explore and develop effective methods for solving such problems using logical modeling techniques. Furthermore, it investigates the field of model checking, which involves the successive generation of possibilities until a buggy trace is found, although without the aid of logical reasoning. This buggy trace is, in Section 5, synonymous with a found solution. To tackle logical models, the SAT solver will be a useful tool, as it can reason over the proposed models and provide viable solutions.

The field of logic includes various approaches that allow the modeling of different problems with varying degrees of expressiveness. Among these logical frameworks, such as first-order logic and propositional logic, see e.g. [2] or [18]. We specifically focus on the use of propositional logic. It serves as the fundamental logic system for our study. Additionally, the field of logic finds application in various domains, including data verification in databases and proof construction in the theory of computation, which were covered in the courses I have taken. To this end, SAT solvers, which come in different types and employ various algorithms and parameters, play a crucial role. While some SAT solvers are customized to specific domains or problems to achieve high efficiency, others, like the *sat4j* solver [3] used in this work, offer a more general approach.

Within the domain of puzzles and related fields of interest, considerable research has already been conducted. Notably, solutions have been presented for games such as Connect Four, where it has been proven that the first player always possesses a winning strategy, see [16]. However, despite the vast array of games and puzzles that have been analyzed, there remain numerous unexplored challenges awaiting logical modeling and solution discovery.

The personal motivation to engage in this research originates from a strong interest in puzzles and a persistent quest for solutions. Whether it is the desire to outsmart an opponent or simply to find the most efficient path to puzzle resolution, the satisfaction derived from solving problems using logical reasoning is immeasurable. For instance, after numerous rounds of playing Connect Four with my partner, I have often wondered how to develop a winning strategy. The domain of problem-solving through logical modeling offers a wide scope that poses countless intriguing questions, and this research aims to contribute answers to some of these questions.

Multiple objectives are at the heart of this research. Firstly, the aim is to address unresolved problems by finding solutions for puzzles and games that have not been previously explored in the context of logical modeling. Secondly, for problems where solutions already exist, the focus shifts towards improving computation times and identifying key factors that influence the efficiency of the solutions. By analyzing existing approaches and exploring innovative modeling techniques , the research aims to improve existing solutions.

Additionally, a crucial objective is to expand the repertoire of solved games that can be used in future projects for logic courses. By presenting elegant or efficient solutions, this research belongs to the domain of game-solving techniques and provides a valuable resource for game enthusiasts, researchers, and developers. The goal is to not only demonstrate the effectiveness of logical modeling but also inspire further investigations and applications in

solving various puzzles and games.

We establish a roadmap to guide our study and achieve our objectives. It begins with an in-depth exploration of the Snake Cube puzzle, which serves as an excellent starting point due to the absence of an existing logical model. This initial stage allows for the formulation of innovative ideas and the development of constraint modeling techniques. The main goal is to identify ways to enhance our initial model to extend it to handle larger instances of the snake-cube while maintaining reasonable computation times.

Making use of the knowledge and experience acquired from the Snake Cube puzzle, the investigation then shifts its focus to the Rubik's Cube, a well-known puzzle with a myriad of existing solver models available online. By examining different approaches, algorithms, and solution methodologies, the research aims to identify an effective strategy for solving the Rubik's Cube with a propositional logic model by first addressing the problem with model checking. Furthermore, this step provides lesser-known ways of modeling constraints, such as the 2-product encoding, see [6], which will prove instrumental in the subsequent phase of the study.

After exploring various models and approaches for constructing constraints, the work concludes with an exploration of Othello, a complex two-player game requiring strategic planning and decision-making. This final stage will use the practical insights gained from the Snake Cube and Rubik's Cube analyses to a more complex problem domain. The constraints in Othello are more complex to model, but they are based on the generalized methodologies developed in the previous stages. The objective of our model is to determine the minimum number of moves required for each grid size to reach the end of the game, which can be useful in creating clauses to determine the winner.

2 Propositional Logic

Propositional logic, also known as *statement logic*, is a branch of logic that aims to connect *propositions* using *logical connectives*. By combining propositions, more *complex propositions* are formed. The terms *statement* and *proposition* can be used interchangeably. An *atomic proposition* represents an indivisible unit and is typically represented by a *Boolean variable*. The combination of atomic propositions results in a larger *proposition*. Propositional logic is sometimes referred to as *zeroth-order logic* and serves as the foundation for higher-level logics, such as *first-order logic*, which involves non-logical objects. It can be seen as the fundamental level of logic from which higher levels are built upon.

A proposition is a statement or evaluation that can be either true or false. In the case of atomic propositions, they are represented by Boolean variables that can have a truth value of either true or false. For instance, the propositions "I have a house" and "My house has four facades" can be represented by the Boolean variables 'p' and 'q'. These variables can take on the values true or false.

By combining multiple atomic propositions using logical connectives, also known as logical operators, we can create composite propositions that can also be evaluated as true or false based on the values of the atomic propositions and the logical connectives used. For example, we can form the composite proposition "I have a house and my house has four facades". The truth value of the entire proposition is determined by the individual truth values of its component atomic propositions and the logical connective 'and'.

The logical connectives in propositional logic include the conjunction '*and*', disjunction '*or*', negation '*not*', implication '*if ... then ...*', and equivalence '*if and only if*'. These connectives are represented by the symbols '$\wedge$' (conjunction), '$\vee$' (disjunction), '$\neg$' (negation), '$\rightarrow$' (implication), and '$\leftrightarrow$' (equivalence) respectively.

A *sentence*, in the context of propositional logic, refers to a complete statement or proposition that can be true or false (it is equivalent to a composite proposition). It is composed of atomic propositions and combinations of logical connectives. For example, these logical connectives allow us to create the sentence "If I have a house and my house has four facades, then I have a four-sided villa". This can be rewritten as "(I have a house $\wedge$ My house has four facades) $\rightarrow$ I have a four-sided villa". Using Boolean variables p, q, and r respectively, we can express it as $(p \wedge q) \rightarrow r$. If variables p and q are assigned the value true, we can conclude that r is also true if the sentence is true. Conversely, if r is known to be true, the sentence must also be true.

However, a sentence can be false depending on the combination of logical connectives and propositions. For instance, in the sentence "I have a house and I do not have a house", represented as $p \wedge \neg p$, the sentence is false.

We encounter two scenarios: either a composite proposition is evaluated to true, allowing us to derive the values of certain atomic propositions based on others, or we derive the value of the entire composite proposition based on the values of the atomic propositions within it. In the latter case, where we seek the value of a composite proposition, we can employ a *truth table*. A *truth table* presents all possible combinations of values for the atomic propositions on the left side and the corresponding value for the composite proposition on the right side. In the following discussion, we will use *variables* instead of atomic propositions, and the term *proposition* will encompass both atomic and composite propositions. For example, the logical operator '$\wedge$', which is a binary operator that denotes

conjunction, is true only when both propositions it connects are true. Its truth table is as follows:

α	β	$\alpha \wedge \beta$
0	0	0
0	1	0
1	0	0
1	1	1

Each of the logical operators has its own truth table as presented in the tables below. Once all the variables have been assigned values, the truth values of the proposition can be determined thanks to tables. If there are no parentheses, the order of precedence is to first evaluate negations, then conjunctions, followed by disjunctions, implications, and finally equivalences.

α	β	$\alpha \vee \beta$
0	0	0
0	1	1
1	0	1
1	1	1

α	$\neg\alpha$
0	1
1	0

α	β	$\alpha \rightarrow \beta$
0	0	1
0	1	1
1	0	0
1	1	1

α	β	$\alpha \leftrightarrow \beta$
0	0	1
0	1	0
1	0	0
1	1	1

- The logical operator '$\vee$' is a binary operator that is true if at least one of the two propositions is true.
- The logical operator '$\neg$' operator is a unary operator that is true if the proposition is false.
- The logical operator '$\rightarrow$' is a binary operator that means that is the left proposition is true, the right proposition must be true. Therefore, it is true is the left proposition is false or if the right proposition is true.
- The logical operator '$\leftrightarrow$' is a binary operator that means that the two propositions must have the same value. It is true if both propositions are false or if both propositions are true

We can use *true* and *false* as equivalents of 1 and 0 to denote the value of a proposition. The are represented by 'T ' for true and 'F ' for false. The operators '$\wedge$' and '$\vee$' are both associative and commutative. T is dominant in a disjunction and absorbed in a conjunction while F is dominant in a conjunction and absorbed in a disjunction. Both of them respect the law of implication $\alpha \vee \alpha \equiv \alpha$ and $\alpha \wedge \alpha \equiv \alpha$.

When multiple operators are used to form a more complex proposition, we can proceed by following the priority rules. For instance, the proposition $\alpha \leftrightarrow \neg\beta \rightarrow \gamma \wedge \delta$ can be rewritten as $\alpha \leftrightarrow ((\neg\beta) \rightarrow (\gamma \wedge \delta))$. For $\alpha = \mathrm{T}$, $\beta = \mathrm{T}$, $\gamma = \mathrm{F}$, and $\delta = \mathrm{T}$, we can substitute these values into the proposition:

$$\begin{aligned} \mathrm{T} \leftrightarrow ((\neg \mathrm{T}) \rightarrow (\mathrm{F} \wedge \mathrm{T})) &\equiv \mathrm{T} \leftrightarrow (\mathrm{F} \rightarrow \mathrm{F}) \\ &\equiv \mathrm{T} \leftrightarrow \mathrm{T} \\ &\equiv \mathrm{T} \end{aligned}$$

In this case, the proposition is true. However, if β is set to false, the implication becomes false, resulting in $\mathrm{T} \leftrightarrow \mathrm{F}$, which makes the proposition false. For all propositions constructed using the five logical operators, we can create a truth table to determine the value of the proposition for all possible combinations of Boolean variables. The challenge arises when dealing with complex propositions that involve a large number of variables, as the number of possible values increases exponentially with the number of variables. Specifically, it becomes 2^n, if n is the number of variables. Consequently, as the proposition becomes more complex and involves numerous variables, the truth table becomes extremely large.

Another useful task in propositional logic is the application of logical equivalences to reduce the complexity of propositions. Logical equivalences allow us to construct different propositions using the same variables but with different logical operators, while ensuring that both propositions have the same value for every combination of variable values. By employing these equivalences, we can simplify complex propositions. Here are some commonly used logical equivalences:

1. Double Negation: $\neg\neg\alpha \leftrightarrow \alpha$
2. De Morgan's Laws: $\neg(\alpha \vee \beta) \leftrightarrow \neg\alpha \wedge \neg\beta$
3. De Morgan's Laws: $\neg(\alpha \wedge \beta) \leftrightarrow \neg\alpha \vee \neg\beta$
4. Implication: $\alpha \rightarrow \beta \leftrightarrow \neg\alpha \vee \beta$
5. Biconditional: $(\alpha \leftrightarrow \beta) \leftrightarrow (\alpha \rightarrow \beta) \wedge (\beta \rightarrow \alpha)$
6. Distribution: $(\alpha \wedge \beta) \vee \gamma \leftrightarrow (\alpha \vee \gamma) \wedge (\beta \vee \gamma)$
7. Distribution: $(\alpha \vee \beta) \wedge \gamma \leftrightarrow (\alpha \wedge \gamma) \vee (\beta \wedge \gamma)$

A logical proposition that is always true is called a *tautology*. For example, $\alpha \vee \neg\alpha$ is a tautology. The truth table of the proposition is always true. All the logical equivalences presented above are tautological propositions. We can prove that the following proposition is a tautology using these logical equivalences:

$$\begin{aligned}
\neg(\alpha \vee \beta) \rightarrow (\alpha \wedge \beta \rightarrow \neg\alpha \vee \neg\beta) &\equiv \neg\alpha \wedge \neg\beta \rightarrow \neg(\alpha \wedge \beta) \vee \neg\alpha \vee \neg\beta \\
&\equiv \neg(\neg\alpha \wedge \neg\beta) \vee \neg\alpha \vee \neg\beta \vee \neg\alpha \vee \neg\beta \\
&\equiv \alpha \vee \beta \vee \neg\alpha \vee \neg\beta \\
&\equiv \mathrm{T}
\end{aligned}$$

The inverse of a *tautology* is a *contradiction* (or self-contradiction). A tautology is a logical proposition that is always true, regardless of the truth values of the propositional variables involved. On the other hand, a contradiction is a proposition that is always false, irrespective of the truth values assigned to the propositional variables. A proposition that is neither self-contradictory nor tautological is called a *contingent* proposition.

When solving problems, one of the objectives is to determine whether a proposition or a set of propositions consists entirely of tautologies, has some instances that make the set of propositions true, or has no instances that make all the propositions true simultaneously. An *instance* refers to assigning a truth value to each variable in the set. If there are

instances that make the set true, it is considered *satisfiable*, whereas if there are no instances to make it true, it is *unsatisfiable*.

To achieve this objective, one approach is to construct the truth table of the proposition or, if we have a set, for each propositions in the set. By doing so, we can observe the different values that the proposition can take. However, as the number of Boolean variables in the set increases, constructing the truth table becomes impractical. For instance, in Section 8, we encounter a scenario with more than 10 000 Boolean variables, resulting in a truth table size of 2^{10000}, which is infeasible in practice.

In the subsequent sections of the work, the goal is to determine the satisfiability of the set of propositions that represents the problem and, if it is satisfiable, to find a truth value assignment to the variables that yields a solution. If multiple propositions within the same set are logically equivalent, they can be simplified by retaining only one of them in the set.

As solving a problem with truth tables is impractical, we need to use algorithms to determine the satisfiability of a set of propositions. This can be accomplished by finding a variable assignment that satisfies all the propositions or by inferring that the set consists of tautologies even if it does not often happen.

When searching for a satisfiable instance, we can either manually infer new propositions to determine the variable assignments or use a SAT solver. For instance, if our set includes the proposition $p \wedge q$, we can infer p and q. An *inference* is a logical consequence of a proposition, indicating that for all instances that make the initial proposition true, the inferred proposition will also be true. It is written as $\{p \wedge q\} \vdash p$ and $\{p \wedge q\} \vdash q$. There are various methods to derive new propositions. Another example is the set $\{p, p \rightarrow q\} \vdash q$.

During the process of reasoning over all propositions in the set, multiple inferences can be made. Whenever a proposition contains only one variable, we can determine whether that variable must be true or false. Each newly inferred proposition becomes part of the set. If we manage to infer two contradictory propositions, it demonstrates that the set is unsatisfiable. For instance, with the set $\{p \rightarrow q,\ p \vee q,\ \neg q\}$ we can infer:

$$\{p \vee q,\ \neg q\} \vdash p$$
$$\{p \rightarrow q,\ p\} \vdash q$$
$$\{\neg q,\ q\} \text{ is a contradiction}$$

However, in the set $\{p \rightarrow q,\ p \vee \neg q,\ q\}$ we can infer:

$$\{p \vee \neg q,\ q\} \vdash p$$
$$\{p \rightarrow q,\ p\} \vdash q$$
$$\{p,\ q\}$$

Unlike logical equivalence, a logical consequence does not permit the removal of the initial proposition from the set. In the last example, the final set is a logical consequence of all the other propositions in the set. Furthermore, assigning a true value to both variables makes the initial set true. Thus, this set is satisfiable with the instance $p = \mathrm{T},\ q = \mathrm{T}$.

References for this section include [18] and [10]. These resources provide numerous examples demonstrating how deduction and inference can be performed.

2.1 Conjunctive Normal Form

This section is based on the knowledge acquired during the course [14].

A set of propositions can be modified using logical equivalences established in the previous section to respect a given form. The two main forms are the disjunctive normal form (DNF) and the conjunctive normal form (CNF). In this work, the useful form is the conjunctive normal form. Actually, it is useful in the application of propositional logic, such as satisfiability testing, and logic-based problem solving.

A Conjunctive Normal Form (CNF) is a form used in propositional logic to represent complex logical propositions in a standardized way. CNF is constructed as a conjunction of clauses, where each clause is a disjunction of literals. A CNF is a conjunction of clauses. In a CNF, each clause represents a separate proposition, and these clauses are combined using '$\wedge$' to form the overall expression. For instance, if the set is $\{p \vee q,\ \neg p \rightarrow q,\ p \rightarrow \neg q\}$, it represents the complex proposition $(p \vee q) \wedge (\neg p \rightarrow q) \wedge (p \rightarrow \neg q)$. However, this set is not in CNF because the propositions of the set are not clauses.

Within each clause, we have a disjunction of literals. A *literal* is a Boolean variable and a sign. The literal is said to be negative if negated and positive otherwise. These literals represent atomic proposition that can be true or false. For the variable 'p', the literal is either 'p' or '$\neg p$'.

The combination of conjunctions and disjunctions in CNF allows us to express complex logical relationships between propositions. By representing them in CNF, we can easily analyze and manipulate them using well-defined rules and algorithms and identify contradictions or redundancies.

The transformation of a logical proposition into CNF involves a series of steps. These steps ensure that the resulting CNF expression consists of a conjunction of clauses, where each clause is a disjunction of literals.

1. Equivalence Elimination: The first step is to transform all equivalences into two implications. $\alpha \leftrightarrow \beta$ becomes $(\alpha \rightarrow \beta) \wedge (\beta \rightarrow \alpha)$.

2. Implication Elimination: The second step is to eliminate all implications into a disjunction. $\alpha \rightarrow \beta$ becomes $\neg\alpha \vee \beta$.

3. Application of De Morgan's Laws: The second step usually adds a negation between the left hand side proposition. This law allows to put the negation into the proposition if it was not an atomic proposition. $\neg(\alpha \vee \beta)$ and $\neg(\alpha \wedge \beta)$ become $\neg\alpha \wedge \neg\beta$ and $\neg\alpha \vee \neg\beta$

4. Distribution of Disjunction over Conjunction: At this step, only conjunctions and disjunctions remain. Disjunctions must be distributed to have the priority on them and have a disjunction of literals. $(\alpha \wedge \beta) \vee (\gamma \wedge \delta)$ becomes $(\alpha \vee \gamma) \wedge (\alpha \vee \delta) \wedge (\beta \vee \gamma) \wedge (\beta \vee \delta)$.

5. Simplification: This last step consists in removing redundancies, double negations and obtain a conjunction of disjunctions of literals.

Another method to transform a set a propositions in CNF is using *Tseitin transformation* which uses auxiliary variables (more details in [19]).

In the following example, each step is useful to obtain the initial proposition in CNF:

$$\begin{aligned}
\alpha \leftrightarrow (\beta \vee \gamma) &\equiv (\alpha \rightarrow \beta \vee \gamma) \wedge (\beta \vee \gamma \rightarrow \alpha) \\
&\equiv (\neg\alpha \vee \beta \vee \gamma) \wedge (\neg(\beta \vee \gamma) \vee \alpha) \\
&\equiv (\neg\alpha \vee \beta \vee \gamma) \wedge ((\neg\beta \wedge \neg\gamma) \vee \alpha) \\
&\equiv (\neg\alpha \vee \beta \vee \gamma) \wedge (\neg\beta \vee \alpha) \wedge (\neg\gamma \vee \alpha)
\end{aligned}$$

There is no need of simplifications because the last proposition is already in CNF.

For this work, all constraints describing a problem or a puzzle are first written in a readable form using implications and equivalence. This transformation into CNF will be useful to solve the problems using a SAT solver.

A disjunctive normal form (DNF) is a disjunction of cubes where a cube is a conjunction of formulas, that is a formula of the form $(l_1 \wedge \ldots \wedge l_n) \vee \ldots \vee (l_i \wedge \ldots \wedge l_j)$. As DNF will not be used in this work, further details are left the reader on the website [21].

2.2 SAT Solvers

SAT solvers, or Satisfiability solvers, are powerful computer algorithms used to decide the satisfiability of propositional logic formulas [29] (see also [14] which is a course that contributed to the following explanations). These solvers employ sophisticated search algorithms and logical inference techniques to efficiently explore the vast solution space of possible truth assignments without constructing the whole truth table.

The main objective of a SAT solver is to determine if a given propositional logic formula is satisfiable or unsatisfiable. If the SAT solver finds the problem satisfiable, it outputs a satisfiable solution with a truth value assignment for each Boolean variable present in the propositions. Otherwise, the SAT solver returns "unsatisfiable", proving that no such assignment exists.

SAT solvers typically operate on formulas represented in CNF, which provides a standardized format for logical propositions that facilitates efficient processing. These solvers can effectively tackle large-scale problems, handling formulas with thousands or even millions of variables and clauses. In fact, in this work, we encountered scenarios, such as the game of Othello, where modeling a complete game required over 10,000 Boolean variables.

The input of a SAT solver is a CNF file written in the DIMACS CNF file format. This format is supported by almost all SAT solvers. The format is described as follow:

- lines starting with a 'c' are comments
- the first line is of the form '$p\ cnf\ n\ m$ where 'n' is the number of Boolean variables and 'm' the number of clauses
- clauses are encoded by lines terminated by a 0. For the clause $(l_1 \vee l_2 \vee l_3)$, the corresponding encoding is '$v_1\ v_2\ v_3\ 0$' where v_i is the sign and the value associated to the literal l_i. If the value associated to the literal l_i is i, and only l_2 is negated. The line for this clause is '1 -2 3 0'. The 0 represents the end of a clause.

The DIMACS CNF file for the set $(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_2 \vee x_3)$ is:

```
c this is some comment
p cnf 3 2
1 -2 0
-1 -2 3 0
```

The output gives the satisfiability of the problem. An output to the previous example could be:

```
c clasp version 3.3.3
c Reading from stdin
c Solving...
c Answer: 1
v 1 -2 3
s SATISFIABLE
c other comments
```

In this work, the constraints of the models described in subsequent sections are implemented using CNF and solved using the *sat4j* solver [3]. Sat4j is a solver that uses the Conflict-Driven Clause Learning (CDCL) algorithm and allows for tuning certain features to adapt to specific problems [4]. Further details about these features and their configuration can be found in the documentation [3].

2.3 Conflict-Driven Clause Learning algorithm

The Conflict-Driven Clause Learning (CDCL) algorithm is an algorithm used to solve satisfiability problems and is used by some SAT solvers, as the one used in this work. The objective of the algorithm is to determine a truth value assignment to Boolean variables of the CNF formulas that make them satisfiable or prove that the problem is unsatisfiable.

The main steps of the CDCL algorithm are performed iteratively, and can be roughly described as follows:

1. Unit Propagation: During unit propagation, truth values are assigned to variables appearing in unit clauses, and these assignments are propagated to other clauses in the formula. This propagation process has two effects on non-unit clauses:

   - Removal: If a non-unit clause becomes satisfied (at least one literals evaluate to true), it can be removed from the working set of clauses since it is no longer relevant to the satisfiability of the formula. In practice, the clause is not really removed.
   - Reduction: If a literal in a non-unit clause evaluates to false, that literal is removed from the clause. In practice, the literal is not really removed.

   For example, in the set $(\neg x_1 \vee \neg x_2),\ (\neg x_1),\ (x_1 \vee x_2)$, with the unit clause $(\neg x_1)$, we assign the value true to the literal $\neg x_1$. Therefore, the value variable x_1 is false and

this assignment propagates to the other clauses. The first clause becomes satisfied and can be removed, while the third clause is reduced to (x_2), which in turn indicates that x_2 must be true.

2. Decision: If no unit propagation can be performed and no conflict is detected, the CDCL algorithm proceeds with making a decision to assign a truth value to a variable at a specific decision level. The decision level is the number of decisions made so far. This decision is typically guided by heuristics that aim to facilitate the search for a satisfying assignment. The heuristic can determine whether the algorithm should choose a positive or negative literal, as well as which specific literal to select. Once the decision is made, the assigned truth value is propagated to other clauses in a similar manner as unit propagation.

3. Conflict Analysis: If a conflict is detected during the propagation in the first two steps, the algorithm performs an analysis to identify the cause of the conflict. A conflict occurs when a subset of clauses, typically two clauses, becomes unsatisfiable. For instance, consider the clauses $(x_1 \vee x_2)$ and $(x_1 \vee \neg x_2)$, and let's assume that the literal x_1 is decided. Through propagation, we obtain both x_2 and $\neg x_2$. This subset of clauses is unsatisfiable, indicating the presence of a conflict. The algorithm learns from the cause of the conflict by incorporating a new clause into the working set.

4. Backtracking: The algorithm backtracks to a previous decision level, undoing variable assignments and conflict analysis, in order to search for a different assignment. The decision level at which backtracking occurs is chosen based on the analysis of the conflict. When a conflict is detected, the algorithm examines the learned clause and identifies the decision level associated with the conflicting literals. The decision level is then adjusted to backtrack to a previous state where the conflict can be resolved. Backtracking allows the algorithm to explore alternative paths and continue the search for a satisfying assignment.

5. Repeat: Steps 1-4 are repeated until a satisfying assignment is found or it is determined that the problem is unsatisfiable.

More details about conflict analysis, learned clauses and backtracking can be found in [20].

The following example illustrates how the algorithm works on a simple set containing four clauses:

$$
\begin{aligned}
&(1)\quad x_1 \vee x_2\\
&(2)\quad \neg x_1 \vee x_2\\
&(3)\quad \neg x_1 \vee \neg x_2 \vee x_3\\
&(4)\quad \neg x_1 \vee \neg x_2 \vee \neg x_3
\end{aligned}
$$

The heuristic used in this example decides positive literals by increasing order. There is no unit propagation or conflict, a decision needs to be made. The heuristic decides the literal x_1 which is propagated in clauses (1) and (2). The working set becomes:

$$
\begin{aligned}
&(2)\quad x_2\\
&(3)\quad \neg x_2 \vee x_3\\
&(4)\quad \neg x_2 \vee \neg x_3
\end{aligned}
$$

The clause (2) becomes a unit clause through the propagation of x_1. The literal x_2 is assigned true with the unit propagation and propagated in clauses (3) and (4). The working set becomes:

$$
\begin{aligned}
&(2)\ \ x_2 \\
&(3)\ \ x_3 \\
&(4)\ \ \neg x_3
\end{aligned}
$$

Clauses (3) and (4) are two conflicting clauses. As only one decision has been done so far, the only analysis that can be done is that variable x_1 must be false. We add the learned clause $(\neg x_1)$ to the working set and backtrack to the first decision:

$$
\begin{aligned}
&(1)\ \ x_1 \vee x_2 \\
&(2)\ \ \neg x_1 \vee x_2 \\
&(3)\ \ \neg x_1 \vee \neg x_2 \vee x_3 \\
&(4)\ \ \neg x_1 \vee \neg x_2 \vee \neg x_3 \\
&(5)\ \ \neg x_1
\end{aligned}
$$

Clause (5) is a unit clause and is propagated into all the other clauses, clause (1) is reduced while clauses (2), (3), (4) are removed:

$$
\begin{aligned}
&(1)\ \ x_2 \\
&(5)\ \ \neg x_1
\end{aligned}
$$

The algorithm reaches the end and find a satisfiable solution. As x_3 is no more in the working set, its value can be set to true or false. The algorithm could return the solution $x_1 = \mathrm{F},\ x_2 = \mathrm{T},\ x_3 = \mathrm{T}$.

3 Solving the Snake Cube with Propositional Logic

The snake cube is a puzzle which consists in the construction of a cube with a chain of blocks. In this problem, blocks are cubic elements of the chain, there are in white and orange in the left part of Figure 1. Cells are the locations in the final cube in 3D (right part of Figure 1.) Each block is either aligned with the precedent and the next block in the chain or it makes an angle with the latter two. Blocks that make an angle are called *corner blocks*. The objective of the puzzle is to find the good orientation of each corner block to form the final cube. In the basic problem, this is a chain with 27 blocks to form a cube of size 3x3x3 as on the Figure 1.

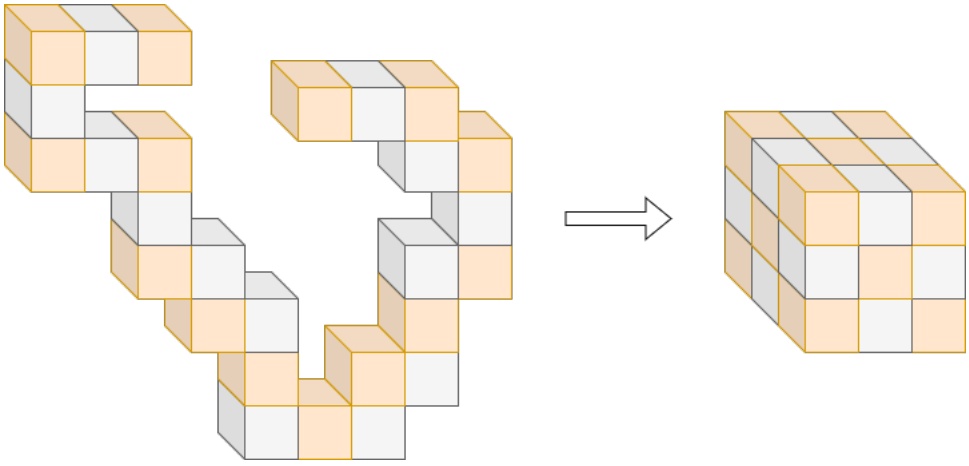

Figure 1: snake cube: initial chain to final cube

Of course, the problem can be adapted to cubes of different size like 4x4x4 or even to some chains that do not contain a cubic number of block, to form a rectangular solid e.g. of size 3x4x4. The last two cases are also analysed in this section.
For the cubes of size 1 and 2, there is, for each, only one possible initial chain and the resolution is trivial. In larger dimensions however, this puzzle becomes non-trivial and can keep someone busy for some time. Our motivation to study this puzzle is also because there exist various potential approaches to solve it. We here present the first attempt (as far as we know) to solve it with propositional logic.

3.1 Modeling

We consider two natural ways to model the problem. For a chain with a cubic number of block $n \geq 8$ to form a final cube of size m with $m^3 = n$, the two model styles are as follows.

- The chain is ordered with values from 1 to n for each block. Cells of the cube are identified by Cartesian coordinates, top left front cell is $1, 1, 1$. j increases to the left, k down and l back. For each cell, we create n Boolean variables to state if the block i is in the cell j, k, l. Therefore there are $m^3 \times n = n^2$ Boolean variables:

$$\forall i \in [1, n],\ \forall j, k, l \in [1, m]^3 \quad p_{i,j,k,l} = \begin{cases} 1 & \text{if block } i \text{ is in cell } j, k, l \\ 0 & \text{otherwise.} \end{cases}$$

- Four Boolean variables per corner block are used to indicate the position of the next block relative to it. It means that if the block i is a corner block, there are four Boolean variables $p_{i,1}, p_{i,2}, p_{i,3}, p_{i,4}$ and the one that is true indicates respectively if block $i+1$ is above, below, to the left, or to the right of block i. There are at most $(n-2) \times 4$ Boolean variables since first and last blocks are not corner blocks.

  $$\forall i \in [1,n] \text{ s.t. } i \text{ is a corner block, } \forall j \in [1,4],\ p_{i,j} = \begin{cases} 1 & \text{if block } i \text{ has orientation } j \\ 0 & \text{otherwise.} \end{cases}$$

  The statement "i has orientation j" indicates the position of block $i+1$ in relation to block i. If block i has orientation 1, block $i+1$ is positioned above block i. The same applies to the other three orientations, as explained earlier.

In the second case, there is much less Boolean variables and the problem seems to be solved more quickly. However, modeling constraints is more complex and it is difficult to express the problem in a convenient way. In fact, this modeling style uses the least amount of useful information to solve the problem. As I did not find how to model constraints with the second model style, the following analyses are conducted based on the first style.

Therefore, we will solve the snake cube with the first modeling style that requires n^2 Boolean variables. The chain will thus be represented by a binary string where 1s indicate corner blocks and 0s indicate aligned blocks. For convenience, the first and last blocks of the chain are 0s but it does not really matter.

3.2 Constraints

According to the modeling style, we must write the further constraints on the Boolean variables for the solver to find a correct solution. There are two rules to describe a correct solution of this puzzle: we must respect the structure of the initial chain and all cells of the final cube must be filled with one and only one block of the chain. Each of these two rules can be represented respectively thanks to three and two sets of constraints.
Therefore, the problem can be entirely defined by five sets of constraints. The use of the symbol '$\forall$' in the constraints and its similarity with the symbol '$\wedge$' is explained in Section 8.2.

The first three sets impose the structure of the chain:

- Each pair of consecutive blocks $\{(1,2),(2,3),\ldots,(n-1,n)\}$ must be in two neighboring cells in the final cube:

  $$\forall j,k,l \in [1,m]^3,\ \forall i \in [1,n-1]\ \ p_{i,j,k,l} \Rightarrow (p_{i+1,j-1,k,l} \vee p_{i+1,j+1,k,l} \vee \ldots \vee p_{i+1,j,k,l+1})$$

  In the disjunction on the right side of the implication in the above formula, Boolean variables that do not have indices within the bounds are simply removed. The disjunction contains at most six Boolean variables, corresponding to the six neighboring cells. Those clauses can be rewritten in conjunctive normal form using the logical equivalence $\alpha \Rightarrow \beta \longleftrightarrow \neg\alpha \vee \beta$. It creates $n \times (n-1)$ clauses.

- For all blocks that are not corner blocks (0s in the binary string), the previous and next blocks must be in cells of a same row with exactly one cell between them. The first and last block of the chain are not considered here since they do not have respectively previous and next blocks:

$$\forall i \in [2, n-1] \text{ s.t. block } i \text{ is not a corner block, } \forall j, k, l \in [1, m]^3 :$$

$$p_{i-1,j,k,l} \Rightarrow (p_{i+1,j-2,k,l} \vee p_{i+1,j+2,k,l} \vee \ldots \vee p_{i+1,j,k,l+2})$$

  As in the previous constraint, the right side of the implication contains at most six Boolean variables, as those with unbound indices are removed from the disjunction.

- For all blocks that are corner blocks (1s in the binary string), the previous and next block must be diagonally adjacent cells:

$$\forall i \in [2, n-1] \text{ s.t. block } i \text{ is a corner block, } \forall j, k, l \in [1, m]^3 :$$

$$p_{i-1,j,k,l} \Rightarrow (p_{i+1,j-1,k-1,l} \vee p_{i+1,j+1,k-1,l} \vee \ldots \vee p_{i+1,j,k+1,l+1})$$

  The same remark regarding the number of variables in the right side of the implication applies to this constraint as well, as mentioned in the two previous constraints. In this case, there are at most twelve Boolean variables.

These last two sets of constraints create together $(n-2) \times n$ clauses. With those three sets, the chain structure is well respected. We have an illustration of those three sets in 2D in Figure 2.

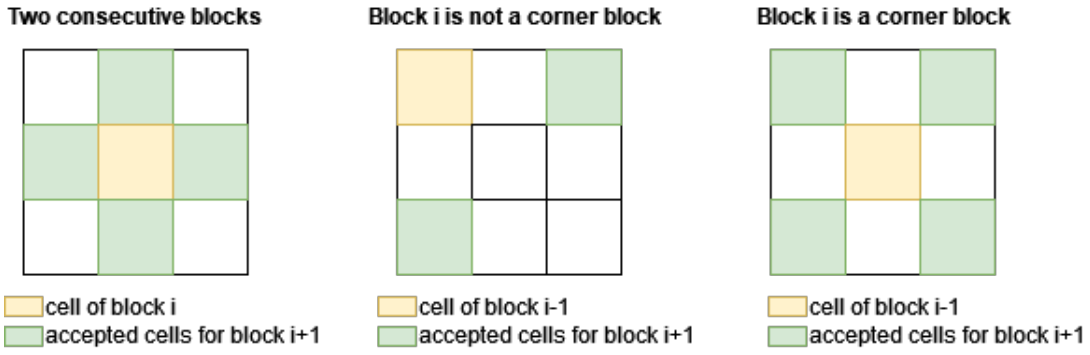

Figure 2: illustration of constraints for the chain structure

To enforce the rule that each cell in the cube contains exactly one unique block, there are multiple possibilities to describe it, each requiring two sets of constraints. The objective is to ensure that no cell contains more than one block and that each block is present in exactly one cell. This can be achieved by formulating constraints that cover both aspects:

1. Each block appears in at least one cell and there is at most one block per cell
2. Each block appears in at most one cell and there is at least one block per cell
3. There is at least one block per cell and at most one block per cell

The third set of constraints does not explicitly specify that different blocks must be present in each cell. However, due to the constraints imposed by the chain structure, where each block must be adjacent to the next one, it implicitly ensures that all blocks are present in the cube. There are three possible ways to represent this, each requiring the same number of clauses. These representations involve an "At Least One" (ALO) constraint and an "At Most One" (AMO) constraint on a set of Boolean variables, where the set size remains constant. A clear explanation of the ALO and AMO constraints is provided in Section 6.2.2.

Since the number of clauses is the same for each pair of constraints, we will arbitrarily choose the first approach for the remaining discussion:

- Each block appears in at least one cell:

$$\forall i \in [1, n] \quad p_{i,1,1,1} \vee p_{i,1,1,2} \vee \ldots \vee p_{i,m,m,m}$$

  There are m^3 Boolean variables per clause and n clauses.

- At most one block per cell, or in other words, there cannot be two different blocks in the same cell:

$$\forall j, k, l \in [1, m]^3, \ \forall i \in [1, n-1], \ \forall h \in [i+1, n] \quad \neg p_{i,j,k,l} \vee \neg p_{h,j,k,l}$$

  An AMO constraint uses $n \times \sum_{i=1}^{n-1} i$ clauses. However, this number of clauses can be reduced using techniques presented in Section 6.2.2.

The total number of clauses is $n \times (n-1) + (n-2) \times n + n + n \times (n-1) \times \frac{n}{2} = n \times (\frac{n^2}{2} + \frac{3n}{2} - 2)$ and the total number of Boolean variables is n^2.

3.3 Improvement with Symmetries

Now we have a good model and a set of clauses that perfectly describe the problem, The objective is to find how to improve the set of clauses so that the SAT solver goes faster. In fact, it is possible to add new clauses that will add new restrictions for the SAT solver to converge faster to the solution (or find that the constraints are unsatisfiable.)

An interesting observation in this problem is the presence of symmetries in the snake cube. Therefore we can add a set of clauses to prevent redundancy in the search of the solution. In fact, several instances of the solution are the same but with a rotation or a symmetry of the cube. For instance, if we place the first block of the snake in one corner of the cube, it does not matter which corner we choose, all the instances we can create with the first block in a corner will be similar. The only difference between these instances will be a rotation or symmetry of the cube. All the instances we can create with the first block in the corner $1, 1, 1$ will be identical to those we can create with the first block in the corner $1, 1, n$, and the same applies to all other corners, whether these instances are satisfiable or not.

What is called an *instance* in this section is assigning each block of the chain to a different cell of the cube. By definition, an instance satisfies all clauses that describe the second rule "all cells of the final cube must be filled in with a different block of the chain". However, an instance does not necessarily satisfy the first rule on the chain structure.

Thus, the objective is to find an instance that satisfies this first rule.
Two instances are similar if one is the image of the other by a rotation or a symmetry of the cube.

Thanks to those new clauses, we can prevent the solver to test similar instances and waste time. Still in the previous example, if putting the first block in the corner $1, 1, 1$ does not allow to find a satisfiable solution, it is useless to try to put the first block in another corner since from another corner, all possible instances will be similar instances to the one made by being in the corner $1, 1, 1$. So, we can add a clause that says that block 1 can not be in one of the seven other corners than the $1, 1, 1$.

In order to detect when there are symmetries and so similar instances, we introduce the notion of *configuration*. A configuration is like an instance but where only the first i blocks are assigned to a cell, $i \in [0, n]$. Configuration 1 is when only the first block is assigned to a cell (there exists n configuration 1), and configuration n is an instance (there exists $n!$ instances).

Recursively, starting from configuration 0, we look if there is one or more symmetry axis in the cube. If a symmetry axis is detected, we incorporate a clause that imposes a constraint during the transition from configuration i to $i+1$. This constraint ensures that the next block can only be positioned on a specific side of the symmetry axis, precisely in a neighboring cell relative to the one currently occupied by block i.

If a symmetry axis intersect an empty cell and that this cell is adjacent to the one containing block i, this cell is accepted for block $i + 1$. If there are several symmetry axis for one configuration i, the next block must be in one of the sections created by the axis. An example is illustrated in Figure 3 in a 2D representation. We are currently in configuration 1 and examining the possible configurations 2.

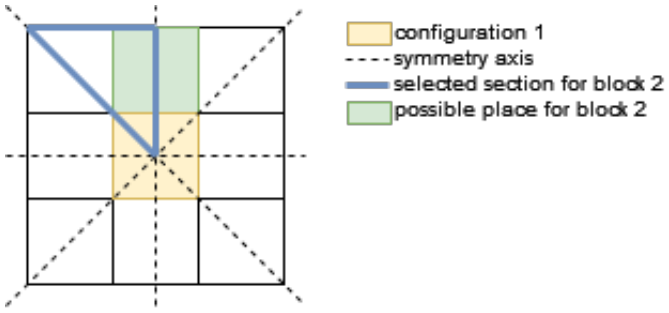

Figure 3: Illustration of symmetries in 2D representation

In Figure 3, only one configuration 2 is possible from configuration 1.

To be consistent, we start in configuration 0 where no block is placed. We just have an empty cube. There are nine symmetry axis. In the case of a 3x3x3 cube, the first block be positioned in only four different cells. We mentioned that block $i + 1$ should be in an adjacent cell of the one containing block i but at configuration 0, as no block is placed, we just take all cells from one section described by symmetry axis to put the first block. Those four cells are shown in Figure 4. The central cell $2, 2, 2$ is green but hidden by the other cells.

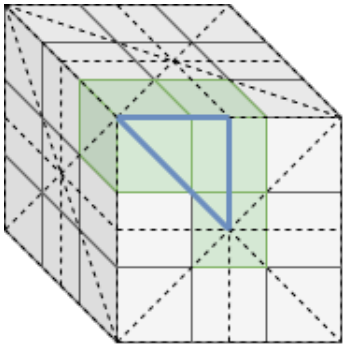

Figure 4: possible configuration 1 from configuration 0 in 3x3x3 cube

The symmetries of configuration 0 to 1 add the clause: $p_{1,1,1,1} \vee p_{1,2,1,1} \vee p_{1,2,2,1} \vee p_{1,2,2,2}$. We obtain four possible configurations for configuration 1. We define $\texttt{Conf}_i$ as the set of all possible configurations i, and $\texttt{conf}_{i,j}$ as the j^{th} configuration within $\texttt{Conf}_i$. For example, $\texttt{Conf}_1$ contains the four configurations $\{p_{1,1,1,1}, p_{1,1,1,2}, p_{1,1,2,2}, p_{1,2,2,2}\}$, where each Boolean variable represents a configuration within this set.

For all the configurations $conf_{i,j}$ in the set $\texttt{Conf}_i$ that exhibits symmetries, we introduce the clause $\texttt{conf}_{i,j} \Rightarrow x \vee \ldots \vee z$, where $x, \ldots, z$ represent the Boolean variables that corresponds to the accepted cells for block $i+1$. This clause ensures that if $conf_{i,j}$ is chosen, the next block $(i+1)$ can only be placed in the specified cells $x, \ldots, z$. Recursively, we go through all possible configurations by incrementing i.

Using this method will create a tree with all configurations with configuration 0 as the root node. The nodes at depth n are the set of all instances that respect the rule "each block is in a cell next to the one of the previous block", removing all similar instances. Due to the large size of the tree, it is possible to truncate it after a certain number of iterations. This is because after several iterations, the number of symmetries decreases significantly, leading to a diminishing number of new and relevant clauses.

In $\texttt{Conf}_0$, there is only one configuration which is the null configuration and this null configuration can be represented by the $True$ predicate. For the 3x3x3 cube, as determined earlier, there are four configurations in $\texttt{Conf}_1$. For example, the configuration $conf_{1,1} = p_{1,1,1,1}$ has three symmetries. The section made by those symmetries contains only one cell of the cube. Therefore, the associated constraint is $p_{1,1,1,1} \Rightarrow p_{2,2,1,1} \longleftrightarrow \neg p_{1,1,1,1} \vee p_{2,2,1,1}$. As $\texttt{conf}_{1,1}$ accepts only one cell for the next block, it adds only one configuration $p_{1,1,1,1} \wedge p_{2,1,1,2}$ in $\texttt{Conf}_2$.

In the code used for analyzing the complexity and computation time of the SAT solver, we include clauses generated by configurations up to $\texttt{Conf}_3$ for the 3x3x3 cube and $\texttt{Conf}_4$ for the 4x4x4 cube. However, for cases such as the 3x3x4 cube which involve a rectangular shape, the code does not include symmetry clauses. All those clauses are general clauses that are adapted to all chains.

3.4 Resolution

Once the set of clauses is prepared, it can be passed to the SAT solver for further processing. However, before delving into the computational aspects, it is important to consider the features of the SAT solver in order to design a solver that is well-suited to the problem at

hand. In this section, we use the sat4j solver [3]. Many of its features are described in [4]. These features provide valuable insights into the customization options available with the sat4j solver.

Among the various features offered by the sat4j solver, one feature of particular importance to us is the *phase selection strategy*. This strategy plays a crucial role in determining how Boolean variables are evaluated during the solving process. When the solver encounters a point where it needs to make a decision on a literal, the phase selection strategy guides the solver in choosing whether to assign a positive or negative value to that literal. This decision can have a significant impact on the solver's performance and efficiency in finding a solution.

In order to adapt the solver to our problem, we need to examine how this feature affects the resolution process. The objective of the symmetry clauses we have added is to provide a means of constructing the cube step by step from configuration 0 to a satisfiable instance. Therefore, we want to decide positive literal because they will be propagated in other clauses. In a way, we want to follow the configuration tree with a sort of depth first search (DFS), DFS is explained in Section 4.2.

Furthermore, in line with the construction approach, we must initially determine the placement of the first blocks. This implies that clauses related to configuration 0 should be prioritized, followed by subsequent configurations. Regrettably, none of the features in sat4j enable us to establish a preference order for clauses. Nevertheless, despite the inability to prioritize these clauses, they still serve a valuable purpose. Hence, in this scenario, we employ the *positive literal selection strategy*.

Another strategy that can be of interest is determining negative literals. This is because, in the case of a 3x3x3 cube, there are only 27 positive literals and 702 negative literals at the conclusion of a satisfiable solution. Consequently, making a decision on a negative literal has a higher probability of being a favorable choice. However, since we have an AMO constraint that create a lot of clauses with two negative Boolean variables. Deciding on a negative literal will remove all the clauses where this literal appears for the AMO constraint but will not make a unit propagation on these clauses. Therefore, there will be fewer variables that will take on a value after a negative decision than after a positive decision. In this case, we use *negative literal selection strategy*.Taking a strategy that mixes positive and negative decision is never the best for our problem.

In the following Table 1, we have the computation time in seconds of the SAT solver for different size of cube, with and without symmetry clauses, and with different decision strategies. The set of symmetry clauses was not tested on a non cubic problem.

	With symmetry clauses		Without symmetry clauses	
Decision strategy	Positive literal	Negative literal	Positive literal	Negative literal
3x3x3 sat	0.11	0.06	0.2	0.08
3x3x3 unsat	0.78	0.89	2.34	4.52
3x3x4 sat	/	/	33.3	0.42
4x4x4 sat	No result	671	No result	231

Table 1: computation time in second of sat4j solver with the "at most one block per cell" constraint

The general tendency that we observe is that adding symmetry clauses is beneficial for the speed of the solver. Moreover, the negative literal decision strategy seems to perform better than the positive one except in the case of an unsatisfiable problem.

The use of symmetry clauses in a SAT solver has a clear advantage when it comes to determining the unsatisfiability of a problem. These clauses help the solver avoid redundant computations for similar problem instances, resulting in faster detection of unsatisfiability. In fact, the solver can infer more clauses and find more quickly two or more contradictory clauses. However, when it comes to satisfiable problems, drawing significant conclusions becomes challenging. The way the SAT solver proceeds resembles an examination of all possible instances. Therefore, the time required to find a satisfiable solution is highly dependent on the specific problem and cannot be generalized.

However, there are some insights we can gain when considering the impact of symmetry clauses on the distribution of satisfiable and unsatisfiable instances. Let's consider a hypothetical scenario where, by removing all similar instances, we have only one satisfiable instance. Initially, there are a large number of unsatisfiable instances compared to satisfiable instances. The inclusion of symmetry clauses reduces the total number of satisfiable instances to one as it removes all similar instances. While the proportion between the number of satisfiable and unsatisfiable instances remains the same, the elimination of similar instances with symmetry clauses reduce the search space.

Let's use arbitrary numbers to illustrate the example. Initially, we have $x = 100$ satisfiable instances and $y = 9900$ unsatisfiable instances. After applying symmetry clauses, we have $x = 1$ satisfiable instance and $y = 99$ unsatisfiable instances. In both cases, the probability of finding a satisfiable instance is $\frac{1}{100}$. However, the expected time required to find the first satisfiable instance is lower when using symmetry clauses. As the SAT solver generates a new clause each time there is a conflict with the CDCL algorithm (see Section 2.3), the solver do not try the same instance twice. The numbers used are not realistic but are solely employed for the purpose of illustrating the maintained proportion between satisfiable and unsatisfiable instances in a simpler manner.

To conclude, in theory, the symmetry clauses should help but since the way the SAT solver works has some randomness, it is not always the case in practice. It explains why the computation time for solving the 4x4x4 cube can be greater with symmetry clauses.

An interesting point that was noticed afterward is that changing the way to express the second rule "exactly one block per cell and each block in a cell", with the second proposition of Section 3.2 can have an impact on the SAT solver efficiency. Indeed, with the constraint "at most one block per cell", when we decide a positive literal, we know that the cell is 'locked' but the block can still be in another cell and lock another cell. It is when all cells will be locked that we can see that the constraint at least one cell per block is unsatisfied.

If we use the constraint "at most one cell per block", once a positive literal is decided, the block is 'locked' but the cell stays 'unlocked'. Thanks to the constraints on the chain structure, the two previous and two next blocks can not be in the same cell.

With this implementation, we obtain the results of Table 2:

	With symmetry clauses		Without symmetry clauses	
Decision strategy	Positive literal	Negative literal	Positive literal	Negative literal
3x3x3 sat	0.28	0.31	0.16	0.25
3x3x3 unsat	1.21	2.22	5	12.2
4x4x4 sat	17	60	101	145

Table 2: computation time in second of sat4j solver with the "at most one cell per block" constraint

The results indicate that the inclusion of symmetry clauses has a beneficial effect on the algorithm. This aligns with the original objective of the change. Therefore, it can be concluded that the *positive literal decision strategy* is the preferable approach.

3.5 Future Works

There are still several directions for improvement to achieve better efficiency. With the following suggestions, I propose that future research focuses on their implementation.

- We can exploit the fact that by construction, as we can see in Figure 1, we know that an 'even' block can not be close to an 'odd' block in the cube. 'Even' blocks are in white and 'odd' blocks are in orange in the figure.
- We can improve symmetry clauses by constructing them directly with the chain structure because for now, when we compute $Conf_{i+1}$ with $Conf_i$, we do not take into account if block i is a corner block or not.
- As discussed in the previous section, there are two ways to describe the second rule, each with its disadvantages. Maybe using the two representations can be positive even if it increases the number of clauses. In order to reduce the number of clauses, we can use the 2 product encoding for AMO constraints as explained in Section 6.2.2.
- We can reduce the number of Boolean variables by using a binary encoding (see Section 6.2.2) for cells location. It means that instead of having n Boolean variables to represent n locations of cells, we can use $\lceil \log_2(n) \rceil$ Boolean variables. However, the expression of clauses is much more difficult.
- We could use another SAT solver or even customize a problem specific SAT solver that allows to give priority on clauses and Boolean variables to choose when there is a decision to do.

4 Model Checking

Model checking is a technique used to verify whether a state space of a model adheres to a given set of specifications and rules. It is particularly useful for ensuring liveness requirements and preventing livelocks in concurrent programs, see [11]. The model and its specifications are described using a specific language that can be processed and verified by the model checker.

To verify the model, the model checker tool systematically explores the state space of the model to identify any buggy traces. The model itself can consist of multiple components, each capable of performing various actions. The state space can be represented as a directed graph, where nodes represent states and edges represent actions. Specifications are encoded into the model, including states that need to be avoided, such as system crashes. The model checker checks whether these specified states exist within the state space of the model. If such states are found, the model checker returns a buggy trace, which is a sequence of actions that led to the undesired state, highlighting what needs to be avoided, see [35].

A major challenge in model checking is the problem of combinatorial explosion of states. When exploring the state space, the model checker can encounter an enormous number of states, which can make the verification of complex systems nearly impossible. To address this issue, techniques such as model reduction or simplification of specifications and properties are employed to reduce the size of the state space. By applying these techniques, the model checker can focus on a more manageable subset of states, enabling more efficient and feasible verification, as discussed in [35].

In this study, the application of model checking focuses on a simple model-checking problem and consists in a verification of whether a formula in propositional logic is satisfiable given the rules of the problem. The acceptable states of the problem are encoded as undesired states, and the rules and moves of the problem are represented as actions. The initial state of the problem is encoded as the root node of the model checker. By traversing the state space of the problem through the execution of actions, the model checker aims to identify if an undesired state is reached. In such cases, the model checker stops and returns the corresponding buggy trace. While the model checker perceives this as an error, for us, it represents the solution to the problem at hand.

To summarize model checking can be characterised by five mains steps:

1. System Modeling;
2. Specification of Properties;
3. Exhaustive State Space Exploration;
4. Property Verification;
5. Handling Combinatorial Explosion;

Another challenge in model checking arises when the state space is either infinite or bounded but contains cycles. Without proper handling of cycles, the model checker may get trapped in an infinite loop and fail to terminate. To address this issue, two primary solutions have been developed.

- The first approach is to set a limit on either the number of states visited or the depth of the search (see Section 4.2). By doing so, the model checker explores only a portion of the state space, potentially missing out on a valid solution. However, this method helps prevent the model checker from getting stuck in infinite loops and allows for a termination condition to be reached.
- The second approach is to prevent cycles in the state space. One naive way to achieve this is by storing all the previously visited states in memory and ensuring that the model checker does not revisit the same state. However, this method can become memory-intensive and slow, especially when dealing with a large number of reachable states, such as in the case of a 3x3 Rubik's cube.

For the Rubik's cube example, we used the *spin* model checker [17]. The spin model checker requires a *Promela* file as input, which contains the model's details. It explores the state space using either a depth first search (DFS) or a breadth first search (BFS), as described in Section 4.2. The tool also provides options to adjust memory management and set a depth limit for the search. In the case of the Rubik's cube problem, a depth limit is necessary to find a solution using DFS.

4.1 Promela Model

In the Promela file, the model is described with several components. These components are processes encoded as *proctype* and run concurrently. Each process has a set of actions that can be performed. These actions are expressed as *guarded command sequence*. To write a Promela model, we need to follow the steps:

1. Identify processes: We start by identifying the processes required in the system. Each process represents a component that interacts with other processes.
2. Define process behavior: We can specify the behavior of each process using a *proctype* declaration. Within a process, we define actions, state variables, and transitions between different states. Actions can include sending and receiving messages, updating variables, or executing specific operations.
3. Declare global variables: We declare the global variables used by multiple processes. These variables represent the shared state of the system and can be accessed and modified by different processes.
4. Model process interactions: In cases where processes need to communicate, we specify how it will be done. This includes communication, synchronization, and coordination mechanisms such as message passing or shared variables.
5. Define system properties: We specify the properties or assertions that need to be verified in the model. We use *assert* statements to define these properties. If an assertion is not satisfied, it results in an error. The acceptable states of the problem are defined as negated assertions to stop the algorithm when reached.
6. Use a model checker: The Promela model is send to a model checker such as *spin* that is used in Section 5.

An introduction to the Promela language can be found in the course [31]. It is important to note that using numerous processes can make the model checking process less efficient.

4.2 Depth First Search and Breadth First Search

The model checker is used to explore the state space of a system or a problem. This state space can be represented as a directed graph, where nodes represent states and edges represent actions between states. For instance, in the case of a Rubik's cube, each state of the cube corresponds to a node, and the movements of the cube are represented by edges in the graph. The two main algorithms used to traverse a graph are Depth First Search (DFS) and Breadth First Search (BFS). This section is based on the course [15].

DFS is an algorithm used to traverse trees or graphs, directed or not. In the case of a tree, the algorithm starts from the root node. For a graph, it can start either from a node representing the initial state or, if none is specified, from an arbitrary node. Assuming the graph is depicted with the starting node at the top and other nodes positioned lower based on their distance from the root node, DFS follows the principle of "depth before breadth". This means it explores the deepest nodes in a branch before visiting neighboring nodes.

DFS is implemented using a Last-In-First-Out (LIFO) queue [9] to traverse the tree or graph vertically downwards from the starting node. It backtracks whenever it encounters a node with no unvisited neighbors. In the case of a graph, which can have cycles, DFS labels visited nodes to avoid revisiting them. The LIFO queue can be seen as a vertical stack where nodes are stacked on top of each other.

The algorithm starts by placing the starting node in the queue. To visit a node, we remove it from the queue and add all its unvisited neighboring nodes to the LIFO queue. The last added node is positioned at the top of the queue. At each step, the algorithm visits and removes the node at the top of the queue, which corresponds to the most recently added node. It then adds any unvisited nodes accessible from this current node to the queue. This process continues iteratively until the entire graph is traversed. When no more nodes or unvisited nodes can be reached from the current node, the algorithm backtracks to the node at the top of the queue (which belongs to a different branch in the case of a tree). If a node is accessible from multiple other nodes and appears multiple times in the queue, it is only visited once due to the labeling mechanism. When a labeled node is at the top of the queue, it is simply removed. The search terminates when the queue becomes empty or when a specific target node is found.

In the case of a search tree with multiple acceptable states, which correspond to accepted nodes, the search can:

- Terminate upon finding an acceptable state. However, this state may not necessarily be the accepted node with the shortest path from the starting node.
- Terminate upon reaching the end of the search tree, indicating a finite search tree.
- Terminate after visiting all branches up to a maximum depth defined as the search limit.
- Never terminate if there is no limit and the search tree is infinite. In this case, it is possible to continuously explore the same branch, going deeper and deeper.

The time complexity of the DFS algorithm is $\mathcal{O}(|V|+|E|)$, where $|V|$ represents the number of vertices and $|E|$ represents the number of edges in the graph. The space complexity is $\mathcal{O}(|V|)$ in the worst case.

BFS is an algorithm used to traverse trees or graphs, directed or not. In the case of a tree, the algorithm starts from the root node. For a graph, it can start either from a node representing the initial state or, if none is specified, from an arbitrary node. Assuming the graph is depicted with the starting node at the top and other nodes positioned lower based on their distance from the root node, BFS follows the principle of "breadth before depth". This means it explores all nodes at the current level before proceeding to the next level in the graph.

BFS is implemented using a First-In-First-Out (FIFO) queue [9] to traverse the tree or graph horizontally across the levels. It systematically explores all neighboring nodes before moving on to deeper levels or branches. In the case of a graph, where cycles can exist, BFS uses a labeling mechanism to mark visited nodes and avoid revisiting them. The FIFO queue can be visualized as a horizontal line where nodes are added to the back of the queue and removed from the front.

The algorithm begins by placing the starting node in the queue. To visit a node, we remove it from the front of the queue and add all its unvisited neighboring nodes to the FIFO queue. At each step, the algorithm visits and removes the node at the front of the queue, which corresponds to the node that has been in the queue for the longest time. Iteratively, the algorithm explores all nodes in the current level before proceeding to the next level. The BFS algorithm continues this process until the queue becomes empty, indicating that all nodes have been visited, or until a specific target node is found. If a node is reachable from multiple paths, it will be added to the queue multiple times, but it will only be visited once due to the labeling mechanism. When a labeled node is at the front of the queue, it is simply removed. The BFS algorithm ensures that all nodes at a given level are visited before moving on to the next level.

In the case of a search tree with multiple acceptable states, which correspond to accepted nodes, the search can:

- Terminate upon finding an acceptable state. It is guaranteed to be the accepted node with the shortest path from the starting node.
- Terminate upon reaching the end of the search tree, indicating a finite search tree.
- Terminate after visiting all branches up to a maximum depth defined as the search limit.
- Never terminate if there is no limit and the search tree is infinite.

The time complexity of BFS is $\mathcal{O}(|V|+|E|)$ and the space complexity is $\mathcal{O}(|V|)$.

We will see with the Rubik's cube example that both searches have their advantages and inconvenient. In the case of a tree that is very deep but not wide, the DFS algorithm will require more memory compared to BFS. This is because the DFS stack stores all the nodes in a branch along with their neighboring nodes. On the other hand, the BFS algorithm only needs to store at most the nodes of a certain level and the nodes of the next level in memory. Therefore, in a shallow but wide tree, BFS will consume more memory. Figure 5 illustrates both types of searches.

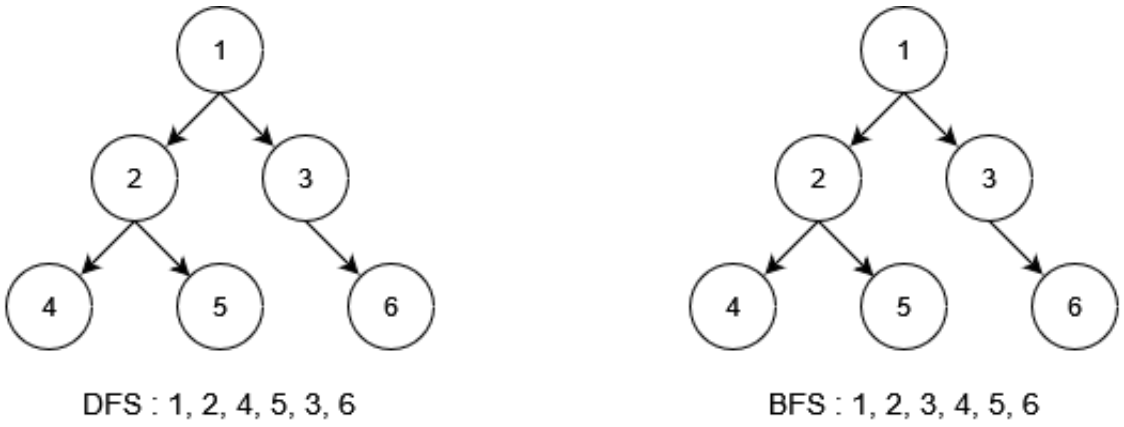

Figure 5: DFS and BFS in a finite tree

The evolution of the LIFO queue during the DFS is as follows:

initial queue : 1
node 1 visited: 3 2
node 2 visited: 3 5 4
node 4 visited: 3 5
node 5 visited: 3
node 3 visited: 6
node 6 visited: $\emptyset$

The evolution of the FIFO queue during the BFS is as follows:

initial queue : 1
node 1 visited: 2 3
node 2 visited: 3 4 5
node 3 visited: 4 5
node 4 visited: 5 6
node 5 visited: 6
node 6 visited: $\emptyset$

5 Solving the Rubik's Cube with Model Checking

The Rubik's cube is a puzzle which consists in a cube with each face divided in several facelets. The cube is formed by small cubes which can each move somehow independently. The objective is to rotate faces of the cube in order to have each face of a uniform color. There are six faces and six different colors. In a nxn cube, there are n^2 facelets by face and n^2 facelets of each color.

There exist many variants of the Rubik's cube that are no more cube. In this section, we will look into the 2x2 cube and how to solve it with a model checker. Other ways to solve Rubik's cube exist such as the Fridrich Method (see [13]) but still no method uses the model checker to get there.

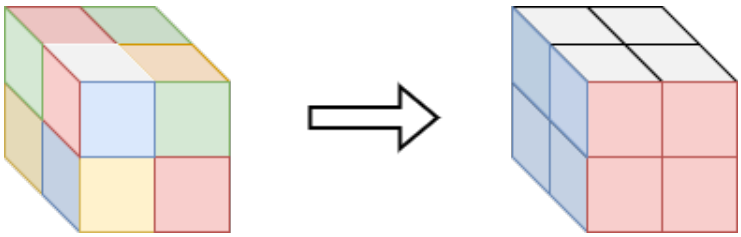

Figure 6: 2x2 Rubik's cube: from a random initial configuration to the acceptable state

Indeed, the 2x2 cube is similar to the well-known 3x3 cube, but with fewer possible movements and, consequently, fewer possible states. This is why the analysis of model checking on the Rubik's cube can be conducted more effectively using the 2x2 Rubik's cube as a basis.

5.1 Modeling

The cube is composed of six faces, each of them ordered from 1 to 6 as shown in Figure 7. In the figure, face 1 corresponds to the top face and the face 2 correspond to the front face. The remaining faces are deduced from these first two.

Each face has four facelets ordered from 1 to 4. A *configuration* of the cube associates a color to each facelet of each face. A valid configuration is a configuration that allows, with some moves, to reach an acceptable configuration.

The difference between a state and a configuration is that several configurations correspond to the same state. Actually, any rotation of the whole cube keeps the cube in the same state. For example, on Figure 6, if the red face goes up, the white face goes left and the blue face goes front, the state is unchanged, but the configuration is different. As there are 24 different rotations of the whole cube that correspond to the same state, it means that there are 24 configurations but one state that are acceptable. Opposite colors are always in pairs in an acceptable configuration : white-yellow, red-orange, blue-green. This explains why there are not 6! acceptable configurations.

The second aspect in the modeling to solve the cube is to describe authorized movements. A *movement* is the description of how a face can rotate and a *move* is the realisation of a movement. In a first approach, looking at the cube in front, there are 18 possible rotations. Each face can rotate 90° clockwise, counterclockwise or a half turn.

Three movements per face and six faces gives 18 movements. For a 2x2 cube, among those 18 rotations, some are redundant.

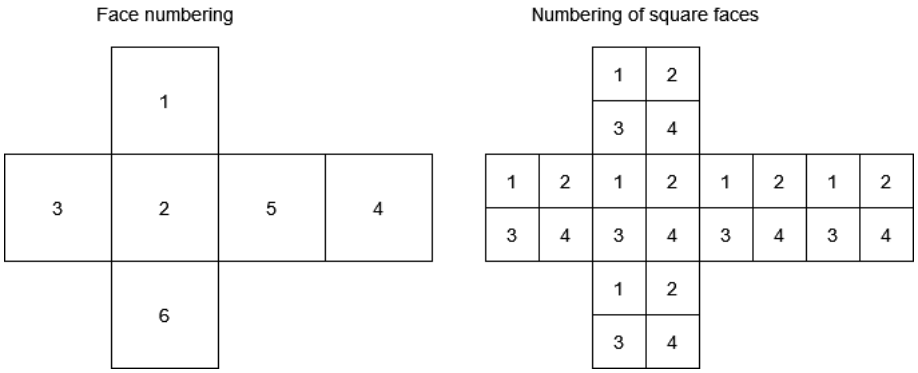

Figure 7: faces and facelets numbering

Indeed, based on the definitions of states and configurations mentioned earlier, it becomes evident that all movements in the Rubik's cube occur in pairs. For example, a clockwise rotation of the left face results in the same state as a clockwise rotation of the right face. The only distinction is a rotation of the entire cube, resulting in two different configurations but ultimately representing the same state. By exploiting these symmetries, we can eliminate nine redundant movements. One has the flexibility to decide whether a 180° rotation is counted as one or two moves. In the case where it is counted as two moves, these half-turns need to be subtracted from the total count of possible moves. There are still three ways to model the movements:

- Three movements: this is the simplest way to model the cube, with only left, up and front clockwise rotation. In fact, two or three clockwise rotations of the same face leads to the same configuration as a half turn or a counterclockwise rotation of this same face.
- Six movements: starting from the 18 initial and removing the six half turns, there are still 12 movements. By excluding the six symmetries of these pairwise movements, we are left with the left, up and front clock and counterclockwise rotations. In this case, the optimal number of moves to reach the acceptable state is less or equal to the case with three movements (equal if the optimal solution only needs clockwise rotations.)
- Nine movements: in fact, to have the optimal solution in terms of number of moves, nine movements are needed. We must add the possibility to do a rotation of 180° in one move, which adds three movements to the six ones just described. This model only remove the nine symmetries from 18 initial movements.

In those three cases, all reachable configurations starting from one initial configuration are the same. The case with three movements is the minimum number of movements required to reach all the states of the 2x2 cube. As defined above, the model style with six movements is an intermediary between the two other model style so we will analyse the two other in the next section.

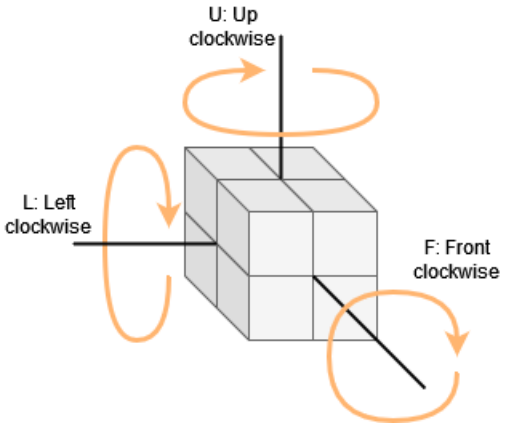

Figure 8: Three movements model style of the cube

5.2 States and Configuration

Once all the movements are described as well as the acceptable configurations, we can launch the model checker and analyse the results. However, we must reflect more on the 2x2 Rubik's cube before running the model checker.

In fact, it is possible to prevent some cycles to happen. For example, in the three cases described above, executing four times the same movement will always lead to the same configuration. With a naive storage approach, it is not possible to efficiently prevent all cycles. In fact, in order to ensure that we never revisit the same configuration, we would need to store all previously visited configurations. With a naive storage method, we would require 9 bytes for storing one configuration (24×3bits). As there are more than 88 million configurations, looking at each of them before making a movement becomes very expensive. In future research, another way to prevent all cycles would be to identify all sequences of moves that keep the cube in the same configuration. Preventing these move sequences would be more efficient as it would only require keeping track of the last few moves.

Therefore, if the model checker use a depth first search (DFS) defined is Section 4.2, it requires a limit in the depth. Otherwise it will stay in the same branch going deeper and deeper indefinitely because the same configuration can occur twice or more and there will always be one more authorized movement.

The 2x2 Rubik's Cube is composed of 8 small cubes, which are all corners. We will refer to these small cubes as *corners*. It has 3 674 160 states which correspond to $7! \times 3^6$, see [26]. Intuitively, it seems to be $8! \times 3^8$ because, if we take the 8 corners and we need to reconstruct the cube, the first corner can be placed in one of the 8 positions. Then, the second corner can be placed in one of the remaining 7 positions, and so on until the last one. This corresponds to the mathematical concept of permutations of the 8 corners, which can be counted as $8 \times \ldots \times 1$ and is denoted as 8! (read as "8 factorial"). Additionally, each corner can have 3 orientations. Since each cube can be oriented in one of the 3 ways, this does $3 \times \ldots \times 3 = 3^8$ possibilities. Thanks to the definition of a configuration, the number $8! \times 3^8$ looks to be the number of configurations. However it is not the number of valid configurations.

Actually, there are certain configurations that are unreachable in a solvable 2x2 Rubik's cube. When seven corners have their orientations determined, the orientation of the last corner becomes fixed with only one possible configuration, since corner rotations occur in pairs. This reduces the total number of configurations to $8! \times 3^7$. Thus, we now have the accurate count of configurations. As we have defined earlier, there are 24 times more configurations than states which gives $\frac{8!\times 3^7}{24} = \frac{8!\times 3^7}{8\times 3} = 7! \times 3^6$.

More formally, it comes from the fact that to prevent rotations of the whole cube, we need to fix the first corner at a fixed place with a fixed orientation and all the other corners move in relation to it. It finally gives a permutation of seven corners and an orientation for six of them, see also [26].

The number of states in the 2x2 cube, which amounts to "only" 3 974 160 possible states, is significantly lower compared to the 43 252 003 274 489 856 000 states of the 3x3 cube. This relatively smaller number of states makes it feasible for a computer to handle. However, the question remains as to how much the depth of the search can be limited.

5.3 Resolution

The implementation of the model follows the explanation provided in Section 4. It involves one process with three or nine guarded command sequences, each corresponding to a specific movement. After each guarded command sequence, the model checker verifies if an acceptable configuration has been reached. If this is the case, the model checker terminates and returns the buggy trace, which effectively represents the solution to solve the cube.

The depth limit for Rubik's cube resolution is called the God's number (demonstration in [5]). It is the maximum number of moves to go from one state to any other possible state trying to use the fewest movements as possible. If we construct a graph where vertices are the states and edges are moves, the God's number would be the diameter of the graph. This number depends on the definition of movements. For the 2x2 Rubik's cube, the God's number is 14 when we consider only 90° rotations (six movements model) and 11 when we allow 180° rotations (nine movements model).

This God's number allows to put an upper limit for the depth of the search. If the starting state of the algorithm is always a possible state, it is useless to have a limit in depth for the BFS because the algorithm will always find the optimal solution which is less or equal to the God's number of moves. Otherwise, if non feasible states are allowed as starting state, we also need to put the limit for the BFS. It will detect when the Rubik's cube is feasible or not.

The God's number for the model with three movements is unknown but since each movement of the nine movements model can be done in at most three moves of the three movements model, the God's number is at most $3 \times 11 = 33$.

With this limit on the depth of the search, we can run the algorithm to see which one between depth first search (DFS) and breadth first search (BFS), see Section 4.2 is best. Computation times presents in Table 3 is calculated using different limits of depth on an unfeasible starting state. By taking an unfeasible starting state, we can see the time for each search to go through the whole search tree without finding any buggy trace. Therefore, it represents the time required to determine that there is no solution.

	max 9 moves		max 10 moves		max 11 moves	
	search time	memory used	search time	memory used	search time	memory used
DFS	0.11s	132Mb	0.16s	136Mb	0.72s	166Mb
BFS	8.14s	1723Mb	40.2s	4618Mb	no result	not enough memory

Table 3: Computation time and memory used by the two search algorithms

One thing to mention about the memory used is that there are 128Mb used for the hash table required by the algorithm. The model checker used in this section is the *spin* model checker [17]. The operation of a model checker is explained in Section 4. As the results show, the DFS is much faster. Actually, the memory used by the DFS is lower and much less time is wasted in managing this memory as explained in Section 4.2.

The problem with DFS is that with a limit of 11 moves, we are not sure to have the optimal solution in terms of the minimal number of moves (most of the time, DFS gives a solution where the number of moves is worth the limit). To obtain the optimal solution, there are two main ideas:

- Use BFS. In this case, we always find the optimal solution in one search but it is possible to have to wait for a while if the optimal solution is 11 moves.
- Use several times the DFS. In this case, there are again different strategies to reduce the search time but in any case, it will be better in terms of time than BFS. In fact, even by running several times the DFS with a limit going from 1 to the optimal number of moves, it is faster than running once the BFS.

Let's consider a practical approach for ordering the limit to execute the DFS. Previous searches have exhaustively explored all possible initial states to determine the optimal number of moves required for each of them [5]. The results are summarized in the following table, which displays the number of states at different distances from the solved state.

Distance from start	number of states
0	1
1	9
2	54
3	321
4	1847
5	9992
6	50136
7	227536
8	870072
9	1887748
10	623800
11	2644
total	3674160

Table 4: Number of states for each distance from start

The median number of moves is 9 and the average number is 8.75 moves. By taking the computation time to do a DFS for each distance, we found the best way to have, on average, the smallest computation time to find the optimal solution. First, we must run a DFS with a limit of nine moves. Then, if no solution is found, we increase the limit by one until a solution is found. At the limit of 11, a solution should be found if the initial state is a valid state. Otherwise, we decrease the limit by one until no solution is found. The optimal solution is therefore the last solution found.

Details of this method depends on the computation time of each distance from start and number of states that are at this distance. Therefore, this method is specific to the nine movements model. The average computation time of this method is less than a quarter of a second to find the optimal solution if the initial state is valid.

The results of the model with three movements cannot be directly compared to those of the model with nine movements. Although both models can find the optimal number of movements by decreasing the limit, the optimal solution in one model is not necessarily equal to the optimal solution in the other model. In the model with three movements, the optimal solution will always be greater than or equal to the optimal solution with nine movements. Since these optimal solutions are different, the two models cannot be compared based on this aspect.

With the model style with three movements, the same move can not be done more than three times in a row, otherwise we find ourselves in the same configuration. It will allow to reduce the size of the search tree in width. A DFS simulation with a maximum of 33 moves computes the time to traverse the search tree, which includes every possible state at least once, as previously mentioned. When discussing the *search tree*, it is more appropriate to refer to states rather than configurations. This is because, in all cases, only movements of the top, front, and left faces are considered, and the bottom corner at the right back remains unchanged. As a result, there are no two configurations that represent the same state within the search tree.

The computation time to solve the Rubik's cube with the three movements model is 5s and the memory used is 402Mb. Even if we said that the two models could not be compared, this model is not better performing than the one with nine movements. Moreover, there is no way to find the optimal solution, where the optimal solution is the one that uses the least face rotations and where a 180° rotation has the same weight as a 90° face rotation. Actually, to find the optimal solution in this sense, we should specify in our model that three clockwise rotations of the same face count as only one move but this leads to the six or nine-movement model.

All those results are theoretically computed because we go through the whole search tree of an initial state which is not valid. In practice, if we only have valid initial states, the solution is found even faster because we only explore a subset of the search tree.

One last thing to remember is that those results are computed with a model checker. As for a Rubik's cube 3x3, the number of different movements is higher and God's number is equal to 20 (for a model that accept 180° rotation), the computation time increases a lot and it becomes difficult to find a solution with a model checker. There exist solvers that provide a solution in a few seconds for the 3x3 Rubik's cube but they use other methods or they use hard coding of ready-made techniques that are known to solve Rubik's cube. At this time, I do not see any improvement that could be useful to improve the computation time and allow us to solve the 3x3 Rubik's cube with a model checker in a reasonable time.

6 Solving the Rubik's Cube with Propositional Logic

The resolution of the Rubik's Cube using a SAT solver is interesting for two main reasons. Firstly, it allows us to explore the possibility of solving the 2x2 cube more efficiently compared to using a model checker, despite the fact that the model checker is already very fast. It raises the question of whether we can extend this approach to solve the 3x3 cube, which currently remains a challenge for the model checker. Secondly, it is interesting pedagogically to present various classes of problems that can be modeled with propositional logic and solved with a SAT solver.

Actually, in Section 3, we used the SAT solver to solve a static problem. This is a problem where we are looking for a satisfiable instance of the problem and where there are no steps. Another popular example of a static problem is the Sudoku problem, where the goal is to fill in the empty cells with numbers that satisfy the game's rules.

In this section, the problem is dynamic. It means that the problem evolves step by step and we can talk about states in the problem. In dynamic problems, we are looking for a sequence of actions leading to an acceptable state. Another dynamic problem is Solitaire, where the objective is to find a series of movements to eliminate each ball from the board. These problems are shown in Figure 9.

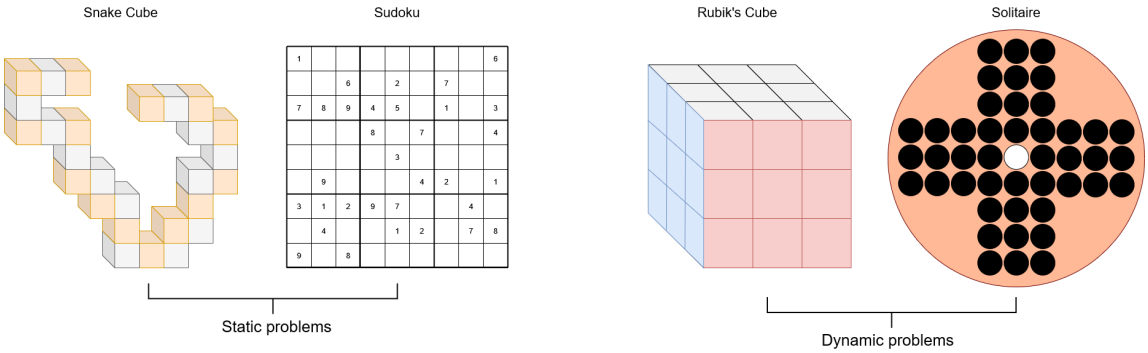

Figure 9: Static vs dynamic games

A dynamic problem can be seen as a static problem where the instance that we find is a series of actions, one for each step. As an illustration, in the case of Sudoku, the instance involves the task of assigning a number from 1 to 9 (in a 9x9 Sudoku) to each empty cell. Conversely, in Solitaire, the instance is assigning a single action for each step ranging from 1 to n. A step of a dynamic problem comprises choosing an action and updating the state.

To deal with a dynamic problem as a static problem, it is essential to have a finite number of steps. Without such a limitation, determining the required number of Boolean variables becomes impossible. Since each step requires at least one action, and the actions necessitates Boolean variables, an unbounded number of steps would imply an unbounded number of Boolean variables. Consequently, it can not be handled in propositional logic as a unique formula, and solved with one call to a SAT solver.

For the Rubik's cube, the number of steps is not known precisely, but it is limited by the God's number (as discussed in Section 5.3). On the other hand, for the Solitaire problem, the number of steps is known exactly and corresponds to the number of balls minus one. While dynamic problems can be addressed using a model checker, their efficiency is not

always guaranteed. In certain cases, using propositional logic to tackle these problems can offer improved performance.

6.1 Modeling

The modeling of a dynamic problem can be divided into two components: the representation of states and the representation of actions (or also called movements in the case of the Rubik's cube). As discussed earlier (Section 5.3), there is a limit on the number of steps in the problem. In this section, a state is equal to a configuration in Section 5.2, as it represents a picture of the problem at some point.

Actions and states are linked. We start from an initial given state zero, and at each step one action $a_i, i \geq 1$ is chosen according to the state s_{i-1}. Implicitly, an action also depends on the previous actions. Once the action a_i is chosen, the state s_{i-1} is updated to obtain the state s_i. The state $s_i, i \geq 1$ only depends on the state s_{i-1} and the action a_i. This link between states and actions is called a Markov Decision Process.

In order to model the Rubik's cube, we drew inspiration from the work of *Jingchao Chen* [7], who presented a SAT solver-based approach for solving the 3x3 cube. However, it's important to note that the computation time of the SAT solver increases exponentially with respect to the number of steps. To overcome this limitation and ensure the solver can handle configurations that require up to 20 steps to solve, he proposed a SAT encoding of the problem and used a problem-specific solver. The algorithm of the specific solver is capable of generating the necessary 20 actions when required, thereby enabling the solution of all possible initial states of the 3x3 Rubik's cube.

We extended this method to the 2x2 cube by preserving the same model style as described in Section 5.1, which includes nine movements, six faces, and 24 facelets. In this adaptation, an action corresponds to a movement of the top, left, or front face. The main objective is to discover a solution that enhances the encoding approach to further improve efficiency while using a general SAT solver.

According to *Jingchao Chen*'s model style, we use one Boolean variable by possible action for each step, that is, nine Boolean variables for the nine movements, and three Boolean variables to specify the moving face. It allows to have a better encoding of constraints. For states, we use a binary encoding for the color of each facelet, which means that only three Boolean variables are needed instead of six. More explanation on the binary encoding is provided in Section 6.2.3. A state is therefore defined by 24 times three Boolean variables and an action by 12 Boolean variables.

As explained in Section 5.3, moving the same face twice in a row is useless since it can be achieved with a single action. This means that an action depends not only on the previous state but also on the previous action. Therefore, we have a Partially Observable Markov Decision Process (POMDP) for the 2x2 Rubik's cube, as shown in Figure 10.

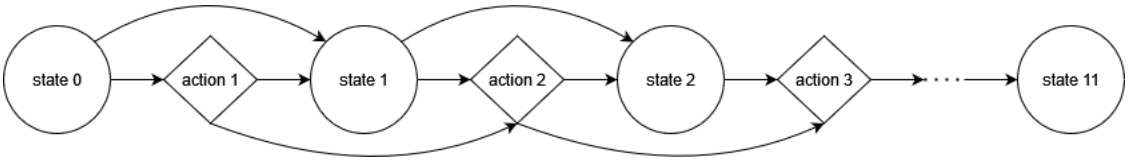

Figure 10: Simplified Partially Observable Markov Decision Process of Rubik's cube 2x2.

For a convenient SAT encoding, we also need a Boolean variable to ensure that an acceptable state is reached. It adds one Boolean variable per state which is equal to 1 if the state is an acceptable state. At least one state must be an acceptable state.
At the end, since there is a limit of 11 actions, we have $11 \times 12 = 132$ Boolean variables for actions and $12 \times (24 \times 3 + 1) = 876$ Boolean variables for states. Here is the list of Boolean variables:

1. $c(i, j, k, l)$ is the l^{th} color bit of facelet j of face i at state k;
2. $a(m, s) = 1$ if step s corresponds to action m (among the nine movements);
3. $f(n, s) = 1$ if face n (among top, left and front face) at step s is moved;
4. $t(k) = 1$ if the state k is an acceptable state.

With $i \in$ [1,6], $j \in$ [1,4], $k \in$ [0,11], $l \in$ [1,3], $m \in$ [1,9], $n \in$ [1,3], $s \in$ [1,11].
The set M containing values of m can be divided in three subsets $M_1 := \{1, 2, 3\}, M_2 := \{4, 5, 6\}$ and $M_3 := \{7, 8, 9\}$. Each subset of action corresponds to one moving face. The face $n = 1$ corresponds to moving the top face and is associated to the subset of actions M_1 which represents moving the top face of 90° clockwise, 180° or 90° counterclockwise. For $n = 2$ and 3, it is the same for the left face and the front face. The set $N := \{1, 2, 3\}$.

In the binary encoding of colors, one reads the three Boolean variables as the bits of a binary number. Therefore, we can have a number from 0 to 7. Each number from 0 to 5 represents a color while 6 and 7 are forbidden. 0 corresponds to orange, 1 to red, 2 to green, 3 to blue, 4 to yellow and 5 to white in our encoding.

6.2 Constraints

The SAT encoding of the Rubik's cube involves five sets of constraints, each of which can be further decomposed into smaller subsets. The use of the symbol '$\forall$' in the constraints and its similarity with the symbol '$\wedge$' is explained in Section 8.2, and the explanations of AMO and ALO constraints is in Section 6.2.2.

- The initial state must be respected. We obtain this thanks to:

$$\forall i \in [1, 6],\ \forall j \in [1, 4],\ \forall l \in [1, 3]\ \ \text{IS}(c(i, j, 0, l))$$

  where, for each i, j, k, l, $\text{IS}(c(i, j, k, l)) = c(i, j, k, l)$ if the l^{th} bit color of facelet j of face i is 1 at the initial state, otherwise $\text{IS}(c(i, j, k, l)) = \neg c(i, j, k, l)$.

- Exactly one action must be chosen at each step with the corresponding moving face. The moving face is described with:

$$\forall s \in [1, 11]\ \ \text{ALO}\ f(N, s) \wedge \text{AMO}\ f(N, s)$$

  Moreover, for each moving face, at most one action can be done

$$\forall n \in [1, 3],\ \forall s \in [1, 11]\ \ \text{AMO}\ a(M_n, s)$$

For the face chosen to move, at least one corresponding action should be true.

$$\forall s \in [1,11],\ \forall n \in [1,3]\ \ f(n,s) \Rightarrow \mathsf{ALO}\ a(M_n,s)$$

Finally, for the faces that are not chosen to move, no corresponding action can be done.

$$\forall s \in [1,11],\ \forall n \in [1,3],\ \forall m \in M_n\ \ \neg f(n,s) \Rightarrow \neg a(m,s)$$

- Once an action is chosen, the next state must correspond to the previous state updated by this action.

  Here, the fact to have the moving face and the action is useful. The moving face allows to know which face does not move and so which facelets do not move.

  $$\forall s \in [1,11],\ \forall n \in [1,3],\ \forall l \in [1,3],\ \forall (i,j) \in \mathrm{Opp}_n\ \ f(n,s) \Rightarrow c(i,j,s,l) = c(i,j,s-1,l)$$

  where the set Opp_n consists of pairs (i,j) that describe the 12 facelets opposite to face n. These facelets are preserved between the two steps. Once the positions of these 12 facelets are updated, the specific action is applied to update the positions of the remaining 12 facelets that move.

  $$\forall s \in [1,11],\ \forall m \in [1,9],\ \forall l \in [1,3],\ \forall (i,j) \notin \mathrm{Opp}_n\ \ a(m,s) \Rightarrow c(i,j,s,l) = A_m(c(i,j,s,l))$$

  where A_m is the update function corresponding to action a_m.

- There must be exactly one state that is an acceptable state. While multiple states could potentially meet the criteria for acceptability, specifying exactly one state simplifies the search once an acceptable state is found. We cannot stop after finding an acceptable state because the number of Boolean variables is predefined. As there are a certain number of variables based on the allowed number of moves, we cannot stop after finding an acceptable state; we must assign values to all the remaining variables as well.

  $$\mathsf{AMO}\ t(S)\ \mathit{and}\ \mathsf{ALO}\ t(S)$$

  With S, the set of all states.

  The definition of an acceptable state also constitutes a constraint:

  $$\forall k \in [0,11],\ \forall j \in [1,3],\ \forall i \in [1,6],\ \forall l \in [1,3]\ \ t(k) \Rightarrow c(i,j,k,l) = c(i,4,k,l)$$

  It means that each of the three first facelets of each face has the same color (same three color Boolean variables) as the fourth facelet of the same face if a state is an acceptable state.

- Consecutive repetitions of the same moving face are not allowed:

  $$\forall s \in [1,10],\ \forall n \in [1,3]\ \ f(n,s) \Rightarrow \neg f(n,s+1)$$

For the third constraint about the update of states, an example of the function A_1 is given here below. It corresponds to the action a_1 (moving face f_1) which corresponds to a 90° clockwise rotation of the top face. This function attributes for each state Boolean variable, an other state Boolean variable of the previous step. The details for the remaining eight functions are provided in Appendix A.

Facelets of top face: $\forall l \in [1,3],\ \forall s \in [1,11]\ A_1(c(1,1,s,l)) = c(1,3,s-1,l),\ A_1(c(1,2,s,l)) = c(1,1,s-1,l),\ A_1(c(1,4,s,l)) = c(1,2,s-1,l),\ A_1(c(1,3,s,l)) = c(1,4,s-1,l)$.

Facelets of side faces: $\forall l \in [1,3],\ \forall s \in [1,11],\ \forall j \in [1,2]\ \ A_1(c(2,j,s,l)) = c(5,j,s-1,l)$ and $\forall i \in [3,5]\ \ A_1(c(i,j,s,l)) = c(i-1,j,s-1,l)$.

These are the changes for the 12 moving facelets during a movement of the top face. Each facelet is assigned the value of another facelet from the previous step.
The Figure 11 shows facelets belonging to Opp_1

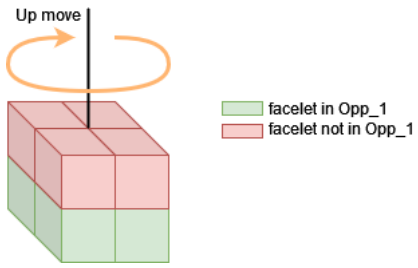

Figure 11: Set Opp_1

In the last example with action A_1, we have a move of the top face that keeps green facelets unchanged and modifies red facelets.

6.2.1 Transformation of Constraints in CNF

The constraints described in the previous section are not in conjunctive normal form (CNF) and cannot be directly solved by a SAT solver. It is necessary to apply some logical transformations to rewrite these formulas into CNF and obtain the corresponding clauses. Each quantifier '$\forall$' in the constraints ranges on the whole line even if there other logical symbols or functions. For each value of the variable in the '$\forall$', it creates a new clause. An equal '=' symbol corresponds to the equivalence '$\Leftrightarrow$' symbol. It means that the two variables must have the same values.

All other symbols are described in the Section 2.1. By following these steps, all constraints that represent the encoding of the Rubik's cube. It only remains to convert the AMO and ALO constraints into CNF.

6.2.2 AMO and ALO Constraints

ALO means *at least one* and AMO means *at most one*. Those constraints apply to a set of Boolean variables among which at least or at most one must be true. When combining the two constraints on the same set of Boolean variables, we have an *exactly one* constraint.

When we use a uppercase letter within the parentheses, it represents a set of Boolean variables. If the set represented by the uppercase letter is not already defined, it represents the set that the corresponding lowercase letter can take. For example, the set S contains the values that s can take, which are [1, 11]. The constraint AMO t(S) is therefore on the set $\{t(1), \ldots, t(11)\}$.

The simplest constraint to express is the ALO constraint, which is a disjunction of positive literals. For each Boolean variable in the set, there is a corresponding literal. If the set contains Boolean variables α, β, and γ, the clause representing the ALO constraint is $\alpha \vee \beta \vee \gamma$. In other words, at least one of the Boolean variables in the set must be true. Thus, the ALO constraint is described by a single clause.

The AMO constraint requires multiple clauses to be expressed. It other words, it means that for any pair of Boolean variables in the set, at least one of them must be false. For each possible pairs of Boolean variables (α, β), we introduce the clause $\neg\alpha \vee \neg\beta$. If the set contains Boolean variables (α, β, γ), we have the following three clauses: $\neg\alpha \vee \neg\beta$, $\neg\alpha \vee \neg\gamma$, and $\neg\beta \vee \neg\gamma$. The AMO constraint is described by $(n-1) \times \frac{n}{2}$ clauses, where n is the number of Boolean variables in the set. As the number of Boolean variables in the set increases, the number of clauses increases still more.

At this state, we are able to express all constraints in CNF. The number of clauses is determined by the sum of the number of clauses required for each constraint. For each constraint, the number of clauses can be computed by multiplying all elements of the constraint. In this computation, an 'equal' constraint creates 2 clauses, an AMO constraint creates $(n-1) \times \frac{n}{2}$, and a '$\forall$' multiplies the number of clauses by e, where e is the number of elements in the '$\forall$'. By performing this computation for each constraint, we obtain the total number of clauses.

An interesting point described by *Jingchao Chen* and presented in [6] is a logical equivalence to reduce the number of clauses required to write an AMO constraint. It increases the number of Boolean variables to reduce the number of clauses. The idea is to generate two new sets of Boolean variables to represent the initial set of variables. By applying an AMO constraint to each of the new sets and assigning one Boolean variable from the initial set to each pair consisting of one Boolean variable from each of the two new sets, we can effectively express the AMO constraint on the initial set. The two new sets can be seen as rows and columns of a two-dimensional grid. The variable from the initial set located at the intersection of the two Boolean variables that are true, if there is one, is the only one permitted to be true. This arrangement ensures that only one variable from the initial set can be true at a time, therefore enforcing the AMO constraint.

This method called the 2-product encoding is illustrated in Figure 12.

P \ Q	q_1	q_2	q_3
p_1	x_1	x_2	x_3
p_2	x_4	x_5	x_6
p_3	x_7	x_8	x_9
p_4	x_10		

Figure 12: Illustration of the 2-product encoding principle

The two axis of the grid are the sets P and Q where P has $\lceil\sqrt{n}\rceil$ Boolean variables p_i and Q has $\lceil\frac{n}{|P|}\rceil$ Boolean variables q_j. If the initial set of Boolean variables is X and contains $\{x_1, \ldots, x_n\}$, each Boolean variable x_k corresponds to a pair (p_i, q_j). To have

AMO of X, we do AMO of P, AMO of Q and a we add a constraint to tell that x_k can be true only if both variables of its pair are true:

$\mathsf{AMO}(X) \equiv \mathsf{AMO}(P) \wedge \mathsf{AMO}(Q) \wedge \forall k \in X,\ \forall(i,j)$ corresponding to k $(\neg x_k \vee p_i) \wedge (\neg x_k \vee q_j)$

The AMO encoding requires $\frac{n^2}{2} - \frac{n}{2}$ clauses while the 2-products encoding for AMO constraint needs $2n+(3|P|-4)+(3|Q|-4)$ clauses and at most $4\sqrt{n}$ auxiliary Boolean variables.

In the last example in Figure 12, the initial set contains ten Boolean variables and the two new sets P and Q contain respectively four and three Boolean variables.

The paper says that this encoding is useful starting with AMO constraints with more than 20 Boolean variables. For a huge number of Boolean variables in the set, the 2-product encoding can be done recursively for the two new AMO constraints. Another way to proceed is the n-product encoding, the idea is the same but using n new sets of auxiliary Boolean variables.

6.2.3 Binary Encoding

In this section, we will introduce an alternative approach to eliminate the need for AMO and ALO constraints. This method employs a binary encoding for the variables involved in the constraint. For instance, if we have a set of seven variables with an 'exactly one' constraint on them, we can either use seven Boolean variables with AMO and ALO constraints on them, or we can use a binary encoding with $\lceil \log(\#var) \rceil$ Boolean variables to represent the variables.

Similar to the encoding of colors discussed in Section 6.1, we can use a binary representation for variables. With the example of seven variables, we would need $\lceil \log(7) \rceil = 3$ Boolean variables, denoted as p_1, p_2, p_3. A variable i is true if the binary number formed by the Boolean variables is $i - 1$. For instance, if the Boolean variables take the values 0 0 0, it corresponds to variable one being true, while 0 0 1 represents variable two being true, and so on until 1 1 0 for variable seven. Since this binary encoding represents only a single number, the AMO constraint is automatically satisfied.

However, the ALO constraint is not directly satisfied. For example, if we the three Boolean variable are 1 1 1, it does not correspond to any variable because there is no variable eight. To ensure that the ALO constraint is verified, we must prevent the appearance of numbers that do not correspond to any variable. This can be achieved by introducing a clause for each forbidden number. In the given case, the clause would be $\neg(p_1 \wedge p_2 \wedge p_3)$, which can be equivalently expressed as $\neg p_1 \vee \neg p_2 \vee \neg p_3$.

Therefore, the maximum number of clauses required is at most equal to the number of forbidden values. The number of forbidden values can be calculated as $2^{\lceil \log(n) \rceil} - n$, where n is the number of variables. However, in some cases, it is possible to prevent the forbidden values using fewer clauses than the actual number of forbidden values.
Let's consider another example where values from 0 to 8 are allowed, and we need to prevent values from 9 to 15. In this scenario, we have four Boolean variables p_1, p_2, p_3, p_4 for encoding. With only the three clauses $\neg p_1 \vee \neg p_2$, $\neg p_1 \vee \neg p_3$, and $\neg p_1 \vee \neg p_4$, we can successfully prevent the forbidden values while allowing the remaining values.

If we have only an ALO constraint, the binary encoding cannot be used and becomes useless as it will allow only one constraint to be true. However, if we have only an AMO

constraint, we can employ this encoding as long as the Boolean variables can represent at least one more number than the total number of variables. In fact, if all variables are false, we require the Boolean variables to be able to represent a number that does not correspond to any variable. We would need $\lceil \log(n+1) \rceil$ Boolean variables.

In a dynamic problem involving states and actions, the state at step s is connected to the state at step $s-1$ through the action taken at step s. In such cases, if states use a binary encoding with an AMO constraint, there is no need to create clauses for forbidden values in the states because the action chosen will conduct directly to a permitted state if the model is well-done.
For instance, we can consider the Rubik's cube as it is the subject of this section. The initial state can be represented using positive unit clauses and a valid color is given to each facelet using binary encoding. When the first action is chosen, it explicitly determines the next state. It means that there is no need to prevent color Boolean variables to be 1 1 0 or 1 1 1 because it becomes impossible to obtain these forbidden values in this scenario.

6.3 Results

Initially, the model is implemented as described in Section 6.2, without using the 2-product encoding for the AMO constraints. Actually, the number of Boolean variables involved in the sets on which the AMO constraints are applied is lower than 20. Therefore, we have 11 244 clauses (72 for the initial state + 67 for exactly one final state + 1296 for final state checking + 275 for the all chosen moves + 9504 for the update of a move + 30 to prevent the same moving face twice in row) and 1008 Boolean variables (12 states $\times$ 24 facelets $\times$ 3 color bits + 12 states $\times$ 1 accepted state checker + 11 actions $\times$ 12 movements).

Taking into account the fact that the SAT solver is slow for unsatisfiable problems (more than a minute), we will focus on generating only solvable initial state to assess the computation time. We randomly generate 12 solvable initial states and measure the average computation time of the SAT solver for these cases. If the goal is to determine whether an initial state is solvable or not, the average computation time needs to be calculated differently. The twelve initial states are listed in Appendix B.

In contrast to the approach described in Section 3.4, for these results, we do not modify the SAT solver's parameters and keep the default configuration. This decision was made as we did not conduct any analysis on the impact of SAT solver's parameters on the computation time in this section.

To calculate the average computation time, we take the average of the means on five attempts for each initial state above. We obtain an average computation time of 1.94 second to solve the cube. Moreover, only three initial states take more than two seconds to be solved, the variance is 5.27 s^2.

In the second step, we will use binary encoding for representing the action and the moving face. The aim is to enhance the model and reduce the number of Boolean variables and clauses required. Specifically, instead of using 12 Boolean variables to determine the action and the moving face at step s, we only need two Boolean variables for the moving face and four Boolean variables for the action.

As we require the four Boolean variables p_1, p_2, p_3, p_4 for the nine different actions, only binary values 0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111 and 1000 are allowed for these

four Boolean variables. To prevent the occurrence of the seven other values, the following constraints should be applied: $\neg p_1 \vee (\neg p_2 \wedge \neg p_3 \wedge \neg p_4)$. In CNF, three clauses are required: $\neg p_1 \vee \neg p_2$, $\neg p_1 \vee \neg p_3$ and $\neg p_1 \vee \neg p_4$.
To choose the moving face among the three ones, we need the two Boolean variables p_5, p_6 where only 00,01,10 are allowed. Therefore we add the clause $\neg p_5 \vee \neg p_6$.

In terms of the overall number of Boolean variables and clauses, the reduction achieved may not be significant since the majority of the clauses are used to describe the transition from one state to another, but the objective is to see if it can have a positive impact on the computation time. Once this enhancement is implemented, the performance is evaluated on the same sample used in the first model. Although the results may not be highly accurate (not enough data used), they provide a general understanding of the performance of this new model.

The disadvantage when using the method of binary encoding is to retrieve the information. Actually, when using one Boolean variable for one variable, the information is clear. For instance, if the Boolean variable that represents action 1 at step 1 is 1, we know directly the action at the first step. With the binary encoding, since the variable for the action at step 1 is divided in four Boolean variables. We need to retrieve these four Boolean variables to know the action chosen. On one hand, we search within a group of Boolean variables for the one that is positive, while on the other hand, we search for all the Boolean variable groups to obtain their values. However, these are merely practical details that do not affect the efficiency of the SAT solver.

The model with binary encoding is, on average across the sample of initial state, significantly worse. Although the size of the sample does not allow for a precise calculation of the average computation time, it still provides a good estimate of the order of magnitude to expect. It becomes evident that the first method is better performing.
In fact, we obtain a computation time of 22.29 seconds to solve the cube. Some initial states take only a few seconds to be solved, while others can take nearly a minute. The variance in solving times with this model is remarkably high with a value of 365 s^2.

These results are disappointing as we had hoped to improve *Jingchao Chen*'s model. However, it does prove that reducing the number of clauses and Boolean variables in a model solved with a SAT solver is not always synonymous with improved computation time. This highlights the complexity of improving SAT-based models and emphasizes the need for careful analysis and experimentation to achieve desired performance improvements.

In conclusion, to solve the Rubik's Cube using a SAT solver, those two models with SAT-encoding do not guarantee finding the optimal solution in terms of the minimum number of moves. To achieve the optimal solution, a strategy similar to Section 5.3 needs to be employed, when we search the optimal solution with a DFS. Actually, to reduce the maximum number of moves used to solve the cube, we only need to modify the clause "at least one" acceptable state among eleven in "at least one" acceptable state among x where x is the maximum number of moves used. We can reduce the number of clauses and Boolean variables by removing all clauses and Boolean variables that describe Boolean variables after step x.

The average computation time to determine the impossibility of solving a cube within the maximum number of allowed moves is approximately 1.2 seconds for eight moves, 5 seconds for nine moves, and 25 seconds for ten moves. Therefore, it is recommended to initially search for a solution within eight moves. If a solution is found, the number of

allowed moves can be progressively decreased until no solution is found, indicating the optimal solution. Conversely, if no solution is found within eight moves, the number of allowed moves can be increased until a solution is obtained.

6.4 Future Works

To enhance the proposed model by *Jingchao Chen*, which was adapted for the 2x2 Rubik's cube, we can explore an alternative approach to represent states and their updates. When examining the number of clauses in our model, we find that 9 504 out of 11 244 clauses are associated with states updates following a move. Currently, we employ $24 \times 3 = 72$ Boolean variables to store a state, and with nine movements, this results in $72 \times 9 \times 2 = 1296$ clauses to express a single step (since each 'equals' operation requires two clauses). However, by using Boolean variables for the type of movement, we reduce the number of clauses to 864 for one step.

As mentioned in Section 5.2, the 2x2 Rubik's cube has a total of 3 674 160 states, which can be represented using $\lceil \log(3674160) \rceil = 22$ bits. However, our current model uses 72 Boolean variables to store a state, which may not be the most efficient representation. Therefore, exploring alternative representations of the cube could potentially improve performance if we are able to significantly reduce the number of clauses. The main challenge would lie in finding a way to express the state update through actions using clauses. This would require careful consideration and experimentation to find an effective approach.

Another analysis that could be performed using this model is to verify the God's number. In order to do this, we would need to remove constraints on the initial state, and search for an initial state that cannot be solved within a number of moves corresponding to the God's number. If this problem is unsatisfiable, we can try again by reducing the number of moves by one. If, in this case, the problem is satisfiable, it would prove the God's number through the model.

7 Two-player Games Problem

The process of solving a two-player game involves determining whether one of the players has a winning strategy regardless of the other player's moves. The objective is to identify if a player can achieve victory no matter what strategy their opponent adopts. If the answer is affirmative, then for each move made by the player in question, a list of moves that maintain their winning position is generated. Depending on the opponent's move, a new list is computed for the subsequent move.

For example, when playing connect-4, the solver [32] can provide guidance to the first player, indicating the optimal move to secure a win. Similarly, the solver may offer advice to the second player, suggesting strategies that prolong the game as much as possible, aiming to delay the opponent's victory.

In addition to the task of solving a two-player game, there are other valuable analyses that can be performed to deepen our understanding of gameplay dynamics. One such analysis involves exploring various game scenarios and observing the outcomes that arise from different player strategies. By considering multiple hypothetical scenarios, we can gain insights into the strengths and weaknesses of different approaches and anticipate the potential consequences of specific moves.

When dealing with such problems, a systematic approach involves several key steps. Firstly, it is essential to determine the initial state of the game. Next, the winning conditions need to be identified, clarifying what constitutes a successful outcome for each player. To formalize the rules of the game, constraints are employed to show the authorized moves, interactions, and restrictions within the game environment. Constraints are also required to model how the game progresses and evolves based on the actions taken by the players. A crucial step is to determine all indicator variables that will allow to model an entire game. In order to solve the game, we will use quantifiers.

7.1 Quantified Boolean Formula

In logic and mathematics, quantifiers allow to express statements involving variables. A quantifier specifies the scope of a variable and determines whether the statement applies to all elements (universal quantification) or at least one element (existential quantification) within a given domain. A Quantified Boolean Formula (QBF) combines these quantifiers with Boolean logic, enabling us to represent and reason about complex problems.

A QBF follows a specific structure: QB, where Q is a sequence of quantifiers $Q_1 v_1 \dots Q_n v_n$. Each quantifier Q_i, which can be either an existential ($\exists$) or universal ($\forall$) quantifier, binds a variable v_i. Importantly, each variable appears at most once in the sequence Q. The formula B represents a Boolean expression in CNF (see Section 2.1).

Universals and existentials in a QBF are the variables quantified by universal and existential quantifiers, respectively. A QBF is determined by the truth values of its subformulas. For a QBF of the form $\exists v_1 Q_2 v_2 \dots Q_n v_n B$, it is true if the QBF $Q_2 v_2 \dots Q_n v_n B$ is true either for $v_1 = \mathrm{T}$ or $v_1 = \mathrm{F}$. Similarly, for a QBF of the form $\forall v_1 Q_2 v_2 \dots Q_n v_n B$, it is true only if the QBF $Q_2 v_2 \dots Q_n v_n B$ is true for both $v_1 = \mathrm{T}$ and $v_1 = \mathrm{F}$.

By employing quantifiers and Boolean logic, QBF provides a means of expressing and analyzing a wide range of problems. They enable us to reason about the existence and

universality of solutions, explore different scenarios, and determine the satisfiability or validity of complex statements. The next sections will delve further into the application of QBF to solve the two-player game of Othello.

In the context of a two-player game, the player for whom we seek victory (typically the first player) is quantified existentially, while the other player is quantified universally. This formulation allows us to focus on the existence of a winning strategy for the first player. By alternating the moves of the first and second players and alternating the existential and universal quantifiers, we formulate the problem as "Does there exist a move for the first player such that for every move of the second player, there exists a move for the first player...?" This can be seen as searching for a satisfiable solution to this problem, where satisfying assignments represent winning strategies for the first player. The indicator variables required to establish the state of the game are quantified existentially following the move that corresponds to that particular state.

8 Playing Othello with Propositional Logic

Othello is a strategic game played by two players on a green square grid board with a size of $n \times n$, where $n \geq 4$. Each player possesses tokens with a black and a white face. During their respective turns, alternatively, players aim to strategically place their tokens on the board. The objective is to have the most tokens of their own color at the end of the game. Typically, the game concludes when the entire grid is filled with $n \times n$ tokens. However, there are situations where the game ends prematurely if neither player can make a valid move. A player who cannot make a move at its turn must pass its turn, while a player who can make a move at its turn must do so as it is mandatory.

The game begins with two tokens of each color placed in a cross formation at the center of the board, as depicted in the left portion of Figure 13. The right portion of Figure 13 illustrates the conclusion of the game, indicating here a victory for the white player.
The player with the black tokens takes the first turn. To place a token on the board, it must be played in an adjacent cell to an opponent's token. Additionally, the played token must form a line with one or more opponent's tokens, with the line ending in a token of the player's color. When a token is played, all opponent's tokens that form a line enclosed by the played token and another token of the same player already on the board are flipped to the player's color. This process is illustrated in the middle of Figure 13, where the valid cells for a black token are marked with black circles and the red circle indicates the placement of the black token. The lines can be horizontal, vertical, or oblique. If multiple lines are enclosed by the played token, all of those lines are flipped. The rules of the game are described in [30]

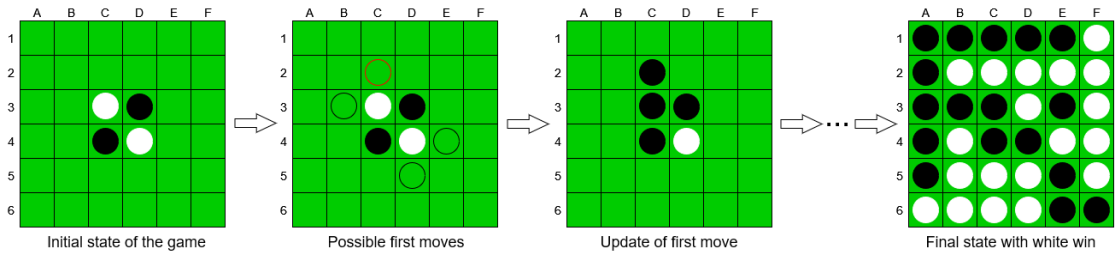

Figure 13: From initial to end state of a Othello game

In this section, illustrations will be presented using a 6x6 board, but the description of variables and constraints will be general for an $n \times n$ board. In propositional logic, the total number of Boolean variables must be determined at the start of the game. Regardless of the specific case, the objective is to progress from state 0 to state S, where S represents the number of tokens required to fill the board. In cases where the game ends before the board is fully occupied, we still proceed until state S without making any further moves.

The game of Othello for a 6x6 board has already been solved, with the white player emerging as the winner. This solution was achieved through the application of machine learning techniques and other mechanisms, as detailed in the reference [28]. The objective of this section is to present a well-designed SAT encoding of Othello and enabling to further researcher to solve it using quantified Boolean formulae (QBF). This encoding can serve as inspiration for researchers tackling other two-player games. Additionally, it can be adapted to accommodate boards of different sizes and facilitate various analyses. For instance, with the first model of constraint presented in the two following sections, we can enforce a player

to make a move by adding a unit clause or ensure that a specific game state occurs. Since no previous results were found in this context, the encoding allows us to determine the minimum number of moves required to reach the end of the game. This idea motivated me to approach the problem of solving Othello using SAT encoding and to answer this question about the minimum number of move.

8.1 Modeling

We will encode the game of Othello as a dynamic problem since states evolve with actions, such as the Rubik's cube problem of Section 5 and 6. As a base of the encoding, we require two kinds of main variables, one for the states and one for the actions. With these two types of variables, we can represent the entire game by describing each step with the move made and the corresponding state. However, the number of clauses would be very large and composed of a significant number of literals, making the encoding unreadable. One source of inspiration for constructing this model is the SAT encoding of the game Connect-4, described in the paper [16].

In our encoding, we will use 15 types of variables, out of which 12 are paired, representing the black and white players. The distinction between them lies in the variable naming convention, where variables for black tokens start with a "b" and variables for white tokens start with a "w". The term *iff* means "if and only if". Traditionally, the columns of the board are identified using letters, while the rows are identified using numbers. However, to simplify the encoding of constraints, we will also encode the columns using numbers.

First of all, we will use the variable $black_{s,c,r}$ (respectively $white_{s,c,r}$) which is *true* iff the cell (c, r) contains a black (or white) token at state s. It allows to keep the state of game. Since playing on a square board of size n, with the initial state $s = 0$, we have $(S + 1)n^2$ variables for each color where S is the total number of moves to fill the board. Last state is state $s = S$. Not surprisingly, another variable is $bmove_{s,c,r}$ (respectively $wmove_{s,c,r}$) which is *true* iff a black (or white) token is played in cell (c, r) at step s. We introduce Sn^2 Boolean variables.

We will add to these kind of variables eleven complementary variables that are only there to make the understanding of the game easiest and the encoding more readable. The variables related to the actions (*bmove* and *white*) are the key variables while all the others (including *black* and *white*) are called *indicator variables*.

- $occupied_{s,c,r}$ is *true* iff the cell (c, r) contains a black or white token at state s. It introduces $(S + 1)n^2$ variables.
- $bline_{s,c_1,r_1,c_2,r_2}$ (respectively $wline_{s,c_1,r_1,c_2,r_2}$) is set to *true* iff there exists a vertical, horizontal, or diagonal line of at least three cells, starting from (c_1, r_1) and ending at (c_2, r_2), in state s that allows a black (or white) token to be placed at (c_1, r_1) in the next move. In such a line, the first cell is unoccupied, the last cell contains a black (or white) token, and all the intermediate cells contain white (or black) tokens. In a 6x6 board, this introduces $S \times 360$ variables for each color, as there are 360 possible lines.
- $bvalid_{s,c,r}$ (respectively $wvalid_{s,c,r}$) is *true* iff the cell (c, r) is a valid cell for a black (or white) token at state s. It introduces Sn^2 variables for each color.

- $bimpo_s$ (respectively $wimpo_s$) is set to $true$ iff there are no valid cells for placing a black (or white) token at state s. It introduces S variables for each color.
- $turn_s$ is $true$ if a black token will be played at move $s+1$, and it is $false$ if a white token will be played at move $s+1$. If the game is over at state s, $turn_s$ is $true$ if the last token played was black, and 0 otherwise. It introduces S variables.
- $bflip_{s,c,r}$ (respectively $wflip_{s,c,r}$) is $true$ if the token in cell (c,r) is flipped to black (or white) after move s. It is $true$ if a black (or white) token is played in cell (c,r) or if cell (c,r) is at the end of a line of white (or black) tokens that were flipped at step s. Otherwise, $bflip_{s,c,r}$ (respectively $wflip_{s,c,r}$) is $false$. It introduces Sn^2 variables for each color.
- $endgame_s$ is 1 iff no move can be done at step $s+1$. It introduces S variables.

Figure 14 provides an illustration of these indicator variables. The indices 's', 'c', and 'r' represent the states, columns, and rows, respectively. 'c' and 'r' range from 1 to n, while 's' ranges from 0 to S. It should be noted that certain Boolean variables do not have the state $s = 0$ or $s = S$.

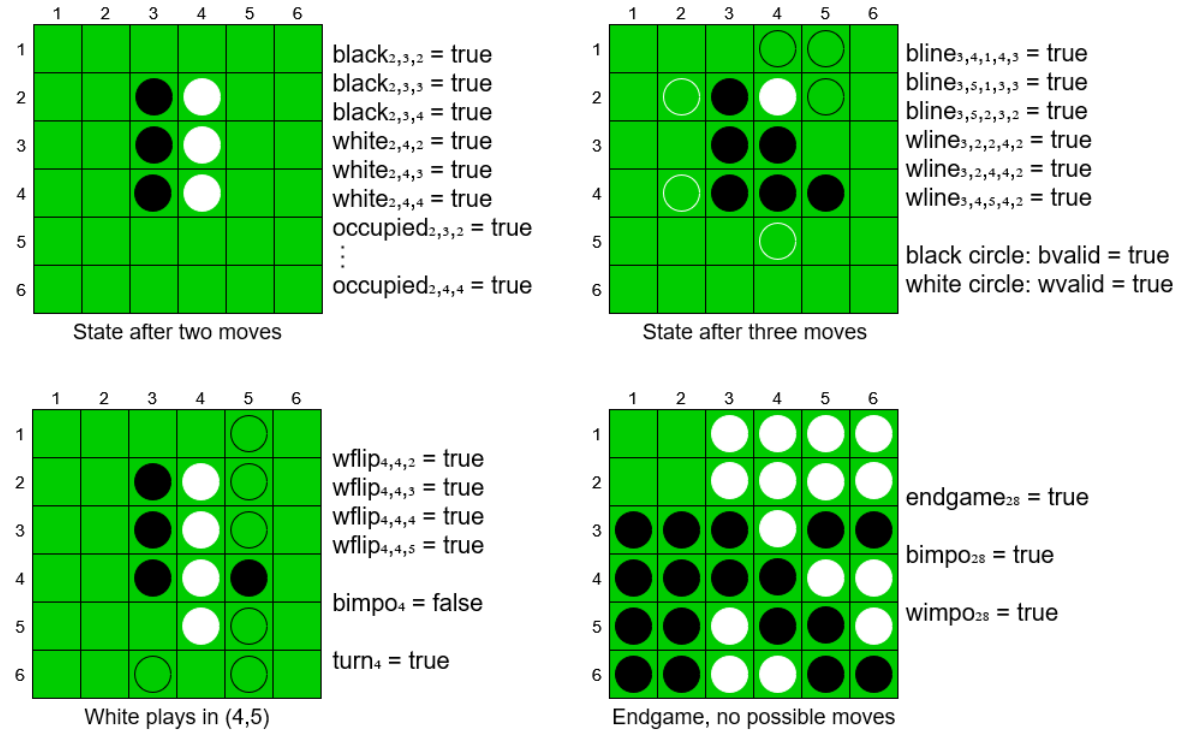

Figure 14: Illustration of indicator variables

All of these indicator variables are interconnected. The variable *occupied* depends only on *black* and *white* variables and is used to check if a *bline* or *wline* starts with an empty cell. In turn, these indicators are used to determine if a cell is valid and available for a black or white token to be played. The variables *bimpo* and *wimpo* indicate if a player is unable to make a move or if the game has reached its end. The variable *turn* represents which player must do the next move and is determined based on the previous turn and the availability of a valid move for the player. To track the evolution of black and white

tokens, the variables *bflip* and *wflip* are used, which are determined by the lines from the previous state and the chosen move. Finally, the *endgame* indicator signifies that no more moves can be made.

The Figure 15 shows the relation between all kinds of variables from state 0 to state S. The expression of constraints will be based on this schema.

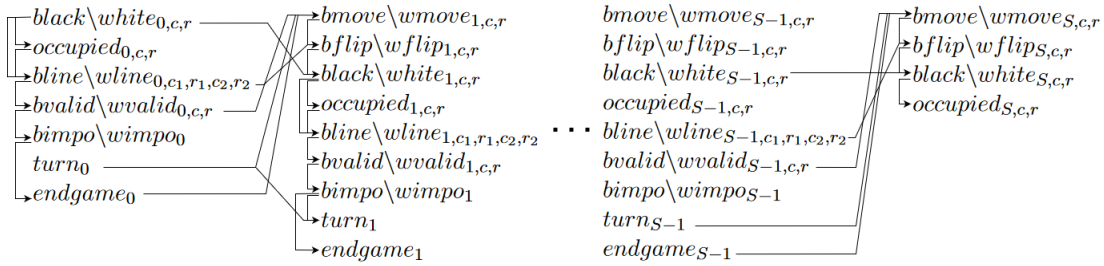

Figure 15: Relation between variables from initial state to end state

To initialize the game, certain variables need to be declared. Since for boards with an even size, the initial state of the actual game is symmetric for the first player (black), the first move does not have any influence on the game. Thus, in our model, the initial state has five tokens on the board, and the total number of cells remaining to be filled is $S = n^2 - 5$. For boards of size five and seven, the initial state of our model is modified by describing only four tokens and giving turn to the black player. The initial state of the game is the state $s = 0$, as shown in Figure 16, and is declared using unit clauses.

$$\forall c, r \in [1, n]^2 \begin{cases} white_{0,c,r} & \text{if } c = r = \lfloor \frac{n}{2} \rfloor + 1 \\ \neg white_{0,c,r} & \text{otherwise} \end{cases}$$

$$\forall c, r \in [1, n]^2 \begin{cases} black_{0,c,r} & \text{if } c = \lfloor \frac{n}{2} \rfloor \text{ and } r = \lfloor \frac{n}{2} \rfloor - 1 \\ black_{0,c,r} & \text{if } c = \lfloor \frac{n}{2} \rfloor \text{ and } r = \lfloor \frac{n}{2} \rfloor \\ black_{0,c,r} & \text{if } c = \lfloor \frac{n}{2} \rfloor \text{ and } r = \lfloor \frac{n}{2} \rfloor + 1 \\ black_{0,c,r} & \text{if } c = \lfloor \frac{n}{2} \rfloor + 1 \text{ and } r = \lfloor \frac{n}{2} \rfloor \\ \neg black_{0,c,r} & \text{otherwise} \end{cases}$$

$$\neg turn_0$$

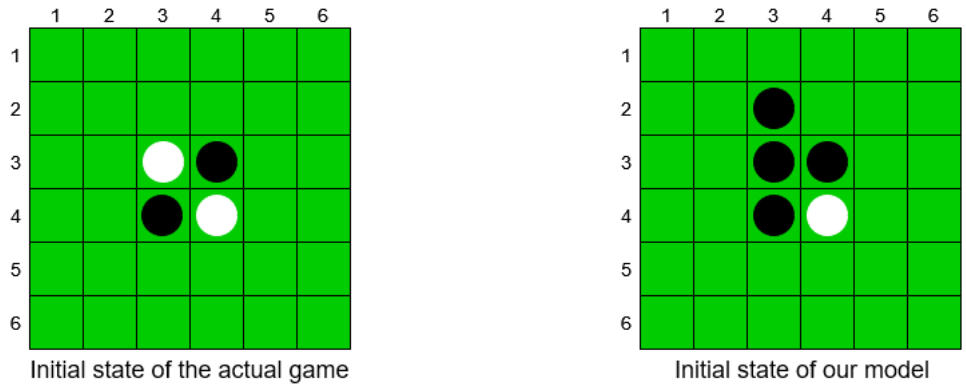

Figure 16: Initial state of the actual game and of our model

Once we have all our Boolean variables for our model, we must link them through constraints to have a complete game from state 0 to state S. So far, for a 6x6 game board, we have the following quantities: $n = 6$, $S = n^2 - 5 = 31$, and a total of 32 596 Boolean variables.

8.2 Constraints

For the expression of constraints, we will use the seven symbols '$\forall$', '$\wedge$', '$\bigwedge$', '$\vee$', '$\bigvee$', '$\Rightarrow$' and '$\Leftrightarrow$'. It is important to have a clear understanding of these symbols to facilitate the comprehension and ensure the correctness of the constraints.

- '$\vee$' and '$\wedge$' are binary operators representing logical disjunction and conjunction, respectively. '$\wedge$' has higher priority than '$\vee$' when evaluating statements. They are used to combine literals and/or formulas. For example, the constraints $p_1 \wedge p_2 \vee p_3$ and $p_2 \vee p_3 \wedge p_1$ both represent $(p_1 \wedge p_2) \vee p_3$.
- '$\forall$' is a quantifier that applies to the entire formula on its right, using a set of values for indices. In fact, this quantifier is a syntactic sugar used to represent a large conjunction. It creates a new clause for each value of the indices, substituting the indices in the formula. For example, the constraint $\forall i \in [2, 4], p_1 \vee p_i$ represents $p_1 \vee p_2$, $p_1 \vee p_3$, and $p_1 \vee p_4$.
- '$\bigwedge$' and '$\bigvee$' are operators that apply to literals directly on their right or, if followed by parentheses, to a set of literals within those parentheses. Unlike '$\forall$', they do not create new clauses but create a new conjunction or disjunction within the formula. For example, the constraint $\bigvee_{i=1}^{3}(p_i \vee \bigwedge_{j=4}^{6} p_j)$ represents $(p_1 \vee (p_4 \wedge p_5 \wedge p_6)) \vee (p_2 \vee (p_4 \wedge p_5 \wedge p_6)) \vee (p_3 \vee (p_4 \wedge p_5 \wedge p_6))$. '$\forall$' is similar to '$\bigwedge$' followed by parentheses applied to the entire formula.
- '$\Rightarrow$' and '$\Leftrightarrow$' are binary operators. '$\Rightarrow$' has higher priority than '$\Leftrightarrow$' but lower priority than '$\vee$' and '$\wedge$'. For example, the constraint $p_1 \wedge p_2 \Leftrightarrow p_3 \Rightarrow p_4 \vee p_5$ represents $(p_1 \wedge p_2) \Leftrightarrow (p_3 \Rightarrow (p_4 \vee p_5))$.

These symbols and their rules of precedence are important for understanding and correctly representing the constraints in a logical and concise manner. A reminder for the unary operator '$\neg$' that has the priority over all previous operators as long as there are no parentheses. The explanations for the AMO and ALO constraints are in Section 6.2.2.

The most intuitive way to explain and describe constraints is to use the logical operators '$\Rightarrow$' and '$\Leftrightarrow$'. For instance, when discussing the act of making a move, the most accurate description would be to say that playing a black token in a particular cell implies flipping certain tokens. However, these operators are not allowed in CNF.

To ensure readability and correctness, we will follow a three-step process when expressing the constraints. First, we will explain them using natural language. Then, we will translate them into logical formulas. Finally, we will transform these formulas into CNF to respect the requirements of SAT solvers.

We have 15 types of Boolean variables, which leads to 15 sets of constraints that describe the relationships between each variable and the others. Most of the variables are

grouped into pairs based on the distinguishing characters "black" and "white", resulting in nine main sets of constraints. For brevity, we will provide the description and initial representation for the black variable only, and the CNF expression will be shown for both the black and white variables, separated by a horizontal line. The variables that are not useful in the last state $s = S$ are described for all s until $s = S - 1$.

1. The first constraint concerns the *occupied* variable and states that a cell is occupied iff a black or a white token is present in the cell.

$$\forall s \in [0, S],\ \forall c, r \in [1, n]^2 \quad black_{s,c,r} \vee white_{s,c,r} \Leftrightarrow occupied_{s,c,r}$$

   In CNF:

$$\begin{aligned} \forall s \in [0, S],\ \forall c, r \in [1, n]^2 \quad & black_{s,c,r} \vee white_{s,c,r} \vee \neg occupied_{s,c,r} \\ & \neg black_{s,c,r} \vee occupied_{s,c,r} \\ & \neg white_{s,c,r} \vee occupied_{s,c,r} \end{aligned}$$

2. In this constraint, we will update the token color present in cells to have *black* and *white* variables. If there is a black flip, the cell contains a black token. If there is a white flip, there is no black token. Otherwise, there is a black token if there was one in the previous step.

$$\begin{aligned} \forall s \in [1, S],\ \forall c, r \in [1, n]^2 \quad & \neg wflip_{s,c,r} \wedge \neg bflip_{s,c,r} \Rightarrow (black_{s,c,r} \Leftrightarrow black_{s-1,c,r}) \\ & bflip_{s,c,r} \Rightarrow black_{s,c,r} \\ & wflip_{s,c,r} \Rightarrow \neg black_{s,c,r} \end{aligned}$$

   In CNF:

$$\begin{aligned} \forall s \in [1, S],\ \forall c, r \in [1, n]^2 \quad & wflip_{s,c,r} \vee bflip_{s,c,r} \vee \neg black_{s,c,r} \vee black_{s-1,c,r} \\ & wflip_{s,c,r} \vee bflip_{s,c,r} \vee black_{s,c,r} \vee \neg black_{s-1,c,r} \\ & \neg bflip_{s,c,r} \vee black_{s,c,r} \\ & \neg wflip_{s,c,r} \vee \neg black_{s,c,r} \end{aligned}$$

$$\begin{aligned} \forall s \in [1, S],\ \forall c, r \in [1, n]^2 \quad & bflip_{s,c,r} \vee wflip_{s,c,r} \vee \neg white_{s,c,r} \vee white_{s-1,c,r} \\ & bflip_{s,c,r} \vee wflip_{s,c,r} \vee white_{s,c,r} \vee \neg white_{s-1,c,r} \\ & \neg wflip_{s,c,r} \vee white_{s,c,r} \\ & \neg bflip_{s,c,r} \vee \neg white_{s,c,r} \end{aligned}$$

3. The following constraints will illustrate the variables *bimpo* and *wimpo*. A move for the black player is impossible iff all cells of the board are not valid for a black token.

$$\forall s \in [0, S-1] \quad \bigwedge_{c=1,r=1}^{n,n} \neg bvalid_{s,c,r} \Leftrightarrow bimpo_s$$

   In CNF:

$$\begin{aligned} \forall s \in [0, S-1] \quad & \bigvee_{c=1,r=1}^{n,n} bvalid_{s,c,r} \vee bimpo_s \\ & \forall c, r \in [1, n]^2 \quad \neg bvalid_{s,c,r} \vee \neg bimpo_s \end{aligned}$$

$$\forall s \in [0, S-1] \quad \bigvee_{c=1,r=1}^{n,n} wvalid_{s,c,r} \vee wimpo_s$$
$$\forall c, r \in [1, n]^2 \quad \neg wvalid_{s,c,r} \vee \neg wimpo_s$$

4. We will now introduce the variable *turn*. *turn* is *true* iff it is the black player's turn. It is the black player's turn iff it was the white player's turn in the previous move and the black player is able to make a move, or if it was the black player's turn in the previous move and the white player is not able to make a move.

$$\forall s \in [1, S-1] \quad \neg turn_{s-1} \wedge \neg bimpo_s \Rightarrow turn_s$$
$$turn_{s-1} \wedge wimpo_s \Rightarrow turn_s$$

   In CNF:

$$\forall s \in [1, S-1] \quad turn_{s-1} \vee bimpo_s \vee turn_s$$
$$\neg turn_{s-1} \vee \neg wimpo_s \vee turn_s$$
$$\neg turn_{s-1} \vee wimpo_s \vee \neg turn_s$$
$$turn_{s-1} \vee \neg bimpo_s \vee \neg turn_s$$

5. This constraint specifies the value of the variable *endgame*. This is the end of the game iff no move can be done.

$$\forall s \in [0, S-1] \quad bimpo_s \wedge wimpo_s \Leftrightarrow endgame_s$$

   In CNF:

$$\forall s \in [0, S-1] \quad \neg bimpo_s \vee \neg wimpo_s \vee endgame_s$$
$$bimpo_s \vee \neg endgame_s$$
$$wimpo_s \vee \neg endgame_s$$

6. The constraints for the Boolean variables *bmove* and *wmove* involve additional steps. In each step, there can be at most one black move. If it is the black turn ($turn_s$ is *true*) and the game is not over, there must be at least one black move (exactly one, but expressing it requires additional constraints, and the AMO constraint is already expressed). Furthermore, a black move can only be made if the cell is valid for black. Otherwise, if it is not black's turn or if it is the end of the game, no black move can be made.

$$\forall s \in [1, S] \quad turn_{s-1} \wedge \neg endgame_{s-1} \Rightarrow \mathsf{ALO}\ bmove_{s,C,R}$$
$$\mathsf{AMO}\ bmove_{s,C,R}$$
$$\forall c, r \in [1, n]^2 \quad \neg turn_{s-1} \vee endgame_{s-1} \Rightarrow \neg bmove_{s,c,r}$$
$$\neg bvalid_{s-1,c,r} \Rightarrow \neg bmove_{s,c,r}$$

In CNF:

$$\forall s \in [1,S] \quad \neg turn_{s-1} \vee endgame_{s-1} \vee \bigvee_{c=1,r=1}^{n,n} bmove_{s,c,r}$$

$$\bigwedge_{\substack{c_1,r_1,c_2,r_2 \in [1,n]^4 \\ (c_1,r_1)\neq(c_2,r_2)}} \neg bmove_{s,c_1,r_1} \vee \neg bmove_{s,c_2,r_2}$$

$$\forall c,r \in [1,n]^2 \quad turn_{s-1} \vee \neg bmove_{s,c,r}$$

$$\neg endgame_{s-1} \vee \neg bmove_{s,c,r}$$

$$bvalid_{s-1,c,r} \vee \neg bmove_{s,c,r}$$

$$\forall s \in [1,S] \quad turn_{s-1} \vee endgame_{s-1} \vee \bigvee_{c=1,r=1}^{n,n} wmove_{s,c,r}$$

$$\bigwedge_{\substack{c_1,r_1,c_2,r_2 \in [1,n]^4 \\ (c_1,r_1)\neq(c_2,r_2)}} \neg wmove_{s,c_1,r_1} \vee \neg wmove_{s,c_2,r_2}$$

$$\forall c,r \in [1,n]^2 \quad \neg turn_{s-1} \vee \neg wmove_{s,c,r}$$

$$\neg endgame_{s-1} \vee \neg wmove_{s,c,r}$$

$$wvalid_{s-1,c,r} \vee \neg wmove_{s,c,r}$$

For the three last constraints, we will introduce four objects that are necessary to represent the constraints in a readable form. Two of these objects are illustrated in Figure 17, where a line starts with a point and finishes with a bar. The lines are at least length three because there must be at least one cell of the opposite color between the empty cell and the cell of our color. In the right illustration, the blue cells represent those that belong to set *Inlineof(2,3)*.

- A set *Lines* contains all horizontal, vertical and oblique lines of at least three cells that are on the board. In a 6x6 board, there are 360 lines. A *line* in *Lines* is represented by the series of cells where $line_1$ denotes the coordinates of the first cell and $line_l$ denotes the coordinates of the last cell.
- A dictionary *Lineof* where, for each coordinates (c,r), Lineof(c,r) represents the set of all lines that start at cell (c,r). In a 6x6 board, corner cells have the highest number of lines, with a total of 12 lines.
- A dictionary *Linewith* where, for each coordinates (c,r), Linewith(c,r) represents the set of all lines which contain (c,r). In a 6x6 board, cells of the center are in the highest number of line, in a total of 66 lines.
- A dictionary *Inlineof* where, for each coordinates (c,r), Inlineof(c,r) represents the set of all coordinates (c',r') which are in a horizontal, vertical or oblique line of (c,r). In a 6x6 board, cells of the center have the highest number of coordinates, with a total of 20 coordinates.

The total length of all the combined lines is 1400 cells. This result is useful for calculating the total number of clauses needed to represent the constraints.

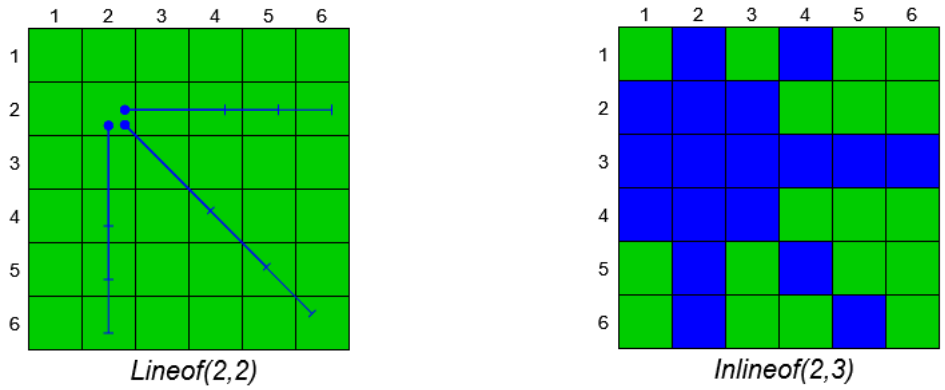

Figure 17: Illustration of sets for Othello constraint

7. The next constraint will give a value to the variables *bline* and *wline*. A line is a black line iff the first cell is unoccupied, the last cell contains a black token and all other are white.

$$\forall s \in [0, S-1],\ \forall line \in Lines\ \ \neg occupied_{s,line_1} \bigwedge_{i=2}^{l} white_{s,line_i} \wedge black_{s,line_l} \Leftrightarrow bline_{s,line_1,line_l}$$

In CNF:

$$\begin{aligned}
\forall s \in [0, S-1],\ \forall line \in Lines\ \ & occupied_{s,line_1} \bigvee_{i=2}^{l-1} \neg white_{s,line_i} \vee \neg black_{s,line_l} \vee bline_{s,line_1,line_l} \\
& \neg occupied_{s,line_1} \vee \neg bline_{s,line_1,line_l} \\
& black_{s,line_l} \vee \neg bline_{s,line_1,line_l} \\
& \forall i \in [2, l-1]\ \ white_{s,line_i} \vee \neg bline_{s,line_1,line_l}
\end{aligned}$$

$$\begin{aligned}
\forall s \in [0, S-1],\ \forall line \in Lines\ \ & occupied_{s,line_1} \bigvee_{i=2}^{l-1} \neg black_{s,line_i} \vee \neg white_{s,line_l} \vee wline_{s,line_1,line_l} \\
& \neg occupied_{s,line_1} \vee \neg wline_{s,line_1,line_l} \\
& white_{s,line_l} \vee \neg wline_{s,line_1,line_l} \\
& \forall i \in [2, l-1]\ \ black_{s,line_i} \vee \neg wline_{s,line_1,line_l}
\end{aligned}$$

8. Before playing a token, we need to know if the cell is valid thanks to the variables *bvalid* and *wvalid*. A cell is valid for a black token iff there exists a black line starting from this cell.

$$\forall s \in [0, S-1],\ \forall c, r \in [1, n]^2 \bigvee_{line \in Lineof(c,r)} bline_{s,line_1,line_l} \Leftrightarrow bvalid_{s,c,r}$$

In CNF:

$$\begin{aligned}
\forall s \in [0, S-1],\ \forall c, r \in [1, n]^2 & \bigvee_{line \in Lineof(c,r)} bline_{s,line_1,line_l} \vee \neg bvalid_{s,c,r} \\
& \forall line \in Lineof(c,r)\ \ \neg bline_{s line_1,line_l} \vee bvalid_{s,c,r}
\end{aligned}$$

$$\forall s \in [0, S-1],\ \forall c, r \in [1, n]^2 \quad \bigvee_{line \in Lineof(c,r)} wline_{s,line_1,line_l} \vee \neg wvalid_{s,c,r}$$
$$\forall line \in Lineof(c,r)\ \ \neg wline_{s_line_1,line_l} \vee wvalid_{s,c,r}$$

9. Finally, the last set of constraints describes the variables *bflip* and *wflip*. If a cell is not located on the same row, column, or diagonal as the cell where a black token is played, there is no black flip in that cell. Additionally, if a white move is made, it implies that no black flips occur. In the case of a black move, all tokens that are in the black lines originating from the move's cell will be black flipped. Subsequently, any cells that lie on the same row, column, or diagonal as the cell where a black token was played but are not part of any black line originating from the move's cell will remain unflipped. Finally, if the game is over, no flips are performed.

$$\forall s \in [1, S],\ \forall c, r \in [1, n]^2 \quad bmove_{s,c,r} \Rightarrow \bigwedge_{\substack{c_1=1,r_1=1 \\ (c_1,r_1) \notin Inlineof(c,r)}}^{n,n} \neg bflip_{s,c_1,r_1}$$
$$wmove_{s,c,r} \Rightarrow \bigwedge_{c_1=1,r_1=1}^{n,n} \neg bflip_{s,c_1,r_1}$$
$$\forall line \in Lineof(c,r)\ \ bmove_{s,c,r} \wedge bline_{s-1,line_1,line_l} \Rightarrow \bigwedge_{i=1}^{l} bflip_{s,line_i}$$
$$\forall (c_1, r_1) \in Inlineof(c,r)\ \ bmove_{s,c,r} \bigwedge_{\substack{line \in Linewith(c_1,r_1) \\ \in Lineof(c,r)}} \neg bline_{s-1,line_1,line_l} \Rightarrow \neg bflip_{s,c_1,r_1}$$
$$endgame_{s-1} \Rightarrow \neg bflip_{s,c,r}$$

In CNF:

$$\forall s \in [1, S],\ \forall c, r \in [1, n]^2 \quad \forall c_1, r_1 \begin{cases} \in [1, n]^2 \\ \notin Inlineof(c,r) \end{cases} \quad \neg bmove_{s,c,r} \vee \neg bflip_{s,c_1,r_1}$$
$$\forall c_1, r_1 \in [1, n]^2\ \ \neg wmove_{s,c,r} \vee \neg bflip_{s,c_1,r_1}$$
$$\forall line \in Lineof(c,r), \forall i \in [1, l]\ \neg bmove_{s,c,r} \vee \neg bline_{s-1,line_1,line_l} \vee bflip_{s,line_i}$$
$$\forall (c_1, r_1) \in Inlineof(c,r)\ \ \neg bmove_{s,c,r} \bigvee_{\substack{line \in Linewith(c_1,r_1) \\ \in Lineof(c,r)}} bline_{s-1,line_1,line_l} \vee \neg bflip_{s,c_1,r_1}$$
$$\neg endgame_{s-1} \vee \neg bflip_{s,c,r}$$

$$\forall s \in [1, S],\ \forall c, r \in [1, n]^2 \quad \forall c_1, r_1 \begin{cases} \in [1, n]^2 \\ \notin Inlineof(c,r) \end{cases} \quad \neg wmove_{s,c,r} \vee \neg wflip_{s,c_1,r_1}$$
$$\forall c_1, r_1 \in [1, n]^2\ \ \neg bmove_{s,c,r} \vee \neg wflip_{s,c_1,r_1}$$
$$\forall line \in Lineof(c,r), \forall i \in [1, l]\ \neg wmove_{s,c,r} \vee \neg wline_{s-1,line_1,line_l} \vee wflip_{s,line_i}$$
$$\forall (c_1, r_1) \in Inlineof(c,r)\ \ \neg bmove_{s,c,r} \bigvee_{\substack{line \in Linewith(c_1,r_1) \\ \in Lineof(c,r)}} wline_{s-1,line_1,line_l} \vee \neg wflip_{s,c_1,r_1}$$
$$\neg endgame_{s-1} \vee \neg wflip_{s,c,r}$$

All of these constraints accurately describe the rules and scenarios of the game. The total number of clauses is obtained by summing the clauses from each constraint (444117)

along with the initial unit clauses (73). This results in a total of 444 190 clauses for a 6x6 board, as shown in Table 5, using 32 596 Boolean variables.

constraint	1	2	3	4	5	6	7	8	9	total
clauses	3456	8928	2294	120	93	45818	109120	24552	249736	444117

Table 5: Number of clauses per set of constraints

This total number of clauses is calculated for a 6x6 board with a total number of lines of 360, and an average length per line of $\frac{1400}{360}$ cells.

8.2.1 Reduction of Clauses and Variables

There are several possibilities to reduce the number of clauses and Boolean variables. One idea is to eliminate unnecessary variables. Specifically, we can remove variables corresponding to the five cells where tokens are initially placed, as there will be no moves on these cells. Additionally, we do not need variables to indicate the occupancy and validity of these cells, as we already know the answers (*occupied* is *true* and *valid* is $false$). Consequently, we can eliminate all lines starting from these five cells. Furthermore, due to the construction of the game, we know that the four corner cells will never be flipped.

Removing all those Boolean variables reduces the total by 4314. Another approach to reducing variables is to observe that certain cells, such as those on the sides, cannot be flipped in the initial moves. Additionally, based on the game's construction, some variables only become relevant after a certain number of moves. Taking these factors into account can further reduce the overall number of Boolean variables needed.

Removing variables will indeed lead to a reduction in the number of clauses. Eliminating 53 out of 360 lines will significantly impact the last three constraints, which generate the highest number of clauses. Additionally, another approach to reducing the number of clauses is by using the 2-product encoding for the "at most one" AMO constraint in the 7^{th} constraint, as described in Section 12. This encoding technique can help further minimize the clause count. Another possible encoding is the binary encoding for black and white moves, as described in Section 6.2.3. This encoding is particularly useful when there is at most one move made per step. However, it was also demonstrated in the same section that a binary encoding does not always provide efficiency benefits.

A last helpful modification is to include the endgame variable in all clauses related to the *bvalid*, *bline*, *bimpo* and *turn* variables of the next step (respectively the same variables for the white). This ensures that when the game is over, all clauses pertaining to subsequent moves are automatically true and no move is made. Our encoding already propagates the end of the game through the connection between variables but explicitly including "$\ldots \vee endgame_s$" in those clauses for state $s+1$ provides a direct termination of the game without further reasoning of the SAT solver. In that case, we need to add the constraint $endgame_s \Rightarrow endgame_{s+1}$ and $\neg endgame_0$ to propagate the end of the game at each step.

Implementing these changes will indeed lead to a significant reduction in the overall number of clauses and Boolean variables, potentially improving efficiency. However, it is not the only approach to enhance efficiency. Adding additional clauses to establish

relationships between variables can greatly assist the SAT solver in its task. For instance, we can add clauses to say that if a cell is occupied at state s, it will be occupied for all state from s to S. While our current encoding serves as a solid foundation for solving the Othello game and conducting various analyses, there are still numerous directions for further improvement.

8.3 Quantifiers and Directives to Solve Othello

With the model established in the previous two sections, we are already capable of performing analysis and reasoning on the game but not yet of solving it. For example, to determine the minimum number of moves required to end the game, we can add the unit clause $endgame_s$ with a specific value for s to see if the solver can find a valid game sequence that ends at state s. If the answer is satisfiable, we can gradually decrease s to find the minimum number of moves needed to finish the game. On the other hand, if the problem is unsatisfiable, we can increase s until we find a valid game sequence. For this particular objective, the last modification proposed in the previous section is very useful, as it allows us to eliminate all clauses beyond state s since *endgame* will be decided to true directly through the unit clause.

We analyzed these questions for the five different board sizes ranging from four to eight. Table 6 provides the minimum number of moves required to reach the end of the game starting from an initial state with four tokens. Since odd-sized boards start with four tokens, we use the same initial state for all five sizes to facilitate comparison in the table.

board size	4x4	5x5	6x6	7x7	8x8
number of moves	6	9	9	9	9

Table 6: The minimum number of moves required to reach an endgame state

These minimum numbers of moves is a scenario where all tokens on the board are black.

Another aspect we can explore with our model is finding optimal strategies or analyzing different scenarios. By introducing additional constraints or modifying existing ones, we can fine-tune the behavior of the game solver and study various aspects of the game. For example, we can impose constraints on the positioning of black or white tokens, simulate specific game situations, or study the impact of different move sequences on the outcome of the game. These modifications will provide valuable insights into the strategic elements of Othello and help us understand the game dynamics more comprehensively.

In order to determine the winner of a game, we need to introduce a new type of Boolean variable. By using the variables *bwin*, *wwin*, and *draw*, we can determine the outcome of a game, as discussed in the previous paragraph. To do this, we need to add clauses that define the winner of a game. Since the victory goes to the player with the most tokens of its color, we need to find a way to count tokens using clauses. Here, we will explain a preliminary approach to express the victory of the white player.

In the case where the board is filled at the end of the game, white wins if at least $\lceil \frac{n^2}{2} + 1 \rceil$ white tokens are present. For a 6x6 board, this would be at least 19 tokens. Our idea is to consider all possible combinations of 19 cells and check if the *white* variable is true for these 19 cells at the final state. If one of the combinations satisfies this condition,

white wins. The same approach can be applied to determine if black wins. Detecting a draw involves considering all possible partitions of the cells into two groups of 18. If we find a partition where all black variables for cells in group 1 are true and all white variables for cells in group 2 are true, it is a draw.
To identify the white win, we create the following constraint. Let C be the set of all combinations of 19 pairs (i, j) chosen from $[1, 6]^2$. $\forall c \in C \ \bigwedge_{(i,j)\in c} white_{S,i,j} \Rightarrow wwin$.

However, if the board is not fully filled at the end of the game, white can win even without having more than 18 white tokens. In this case, we need to add additional victory conditions. If one cell is empty, white only needs 18 tokens to win. In this scenario, finding a combination of 18 white cells and 1 empty cell results in a victory for white. The same approach should be applied to all possible scenarios. For example, if we find a combination with 11 white cells and 15 empty cells, it is also a victory for white because black cannot have more than 10 tokens. To determine the maximum number of empty cells that we can have when a game is over, we need to know the minimum number of moves required to finish a game that are calculated in Table 6. It gives us the possible number of empty cells.

It is important to note that using combinations and permutations on large sets leads to an enormous number of clauses. Therefore, this approach is only a preliminary one and is intended to be improved in future research.

We will now introduce quantifiers and follow the directives explained in Section 7.1. The first player must be an existential player, while the second player must be a universal player. This means that we are interested in determining if the first player has a winning strategy. We want to check if there always exists a sequence of moves for the first player such that for any possible sequence of moves for the second player, the first player can win. For even-sized boards, we look for a winning strategy for the white player, while for odd-sized boards, we look for a winning strategy for the black player.

To achieve this, we add the unit clause *wwin* to the set of clauses, representing the desired win condition for the first player (white). The objective of the first player is to make the problem satisfiable in order to win, while the objective of the second player is to make the problem unsatisfiable to prevent the first player from winning. In fact, if the problem is satisfiable, the clause *wwin* is true.

Since each move in Othello has an influence on the next move, we need to establish an order for quantifying the variables. We can start by quantifying the first move made by the first player, followed by the quantification of the second move made by the second player, and so on until the end of the game.

$$
\begin{aligned}
&\exists\{wmove_{1,1,1}, wmove_{1,1,2}, \ldots, wmove_{1,n,n}\}\\
&\forall\{bmove_{2,1,1}, bmove_{2,1,2}, \ldots, bmove_{2,n,n}\}\\
&\exists\{wmove_{3,1,1}, wmove_{3,1,2}, \ldots, wmove_{3,n,n}\}\\
&\vdots\\
&\exists\{wmove_{S,1,1}, wmove_{S,1,2}, \ldots, wmove_{S,n,n}\}
\end{aligned}
$$

One challenge arises when a player cannot play and needs to pass its turn as it becomes unclear which player will make the even and odd moves. To account for the fact that both

players can potentially make a move on each turn, we can employ a different approach. We include all players in all moves, always starting with the first player (although the order itself is not important). We introduce quantifiers such as "there exists a first move for player 1, such that for every first move of player 2, there exists a second move for player 1", and so on until the end of the game. By using the variable *turn*, we can allow both players the opportunity to play, but only the player whose turn it is will actually make the move.

$$
\begin{aligned}
&\exists\{wmove_{1,1,1}, wmove_{1,1,2}, \ldots, wmove_{1,n,n}\}\\
&\forall\{bmove_{1,1,1}, bmove_{1,1,2}, \ldots, bmove_{1,n,n}\}\\
&\exists\{wmove_{2,1,1}, wmove_{2,1,2}, \ldots, wmove_{2,n,n}\}\\
&\vdots\\
&\exists\{wmove_{S,1,1}, wmove_{S,1,2}, \ldots, wmove_{S,n,n}\}\\
&\forall\{bmove_{S,1,1}, bmove_{S,1,2}, \ldots, bmove_{S,n,n}\}
\end{aligned}
$$

The second problem is that, currently, the second player has always a winning strategy. In fact, for example, it can simply make the "at most one move per step" constraint false. With its universal quantifier, it can set two different *bmove* variables to true at the same step, making the problem unsatisfiable and winning the game. To counter this, we introduce a new Boolean variable *bcheat*, which is true if the second player (black) cheats. This variable will be added to all the clauses related to *bmove*. Therefore, if the black player tries to cheat by making a clause false by not following the rules, the first player (white) can set *bcheat* to true and win the game. For instance, instead of having $\neg bmove_{2,1,1} \vee \neg bmove_{2,1,2}$, we will have $\neg bmove_{2,1,1} \vee \neg bmove_{2,1,2} \vee bcheat_2$. If the second player wants to cheat by placing two tokens in cell $(1,1)$ and $(1,2)$ at the second step, the first player can make the problem satisfiable by setting $bcheat_2$ to true.

Now, the problem is that the first player has always a winning strategy by accusing the second player of cheating. Thus, we need to ensure that *bcheat* is true if and only if the second player cheats. To continue with the same example, we add the clause $bmove_{1,1,1} \vee bmove_{1,1,2} \vee \neg bcheat_1$. In this way, if the second player does not cheat and the first player declares $bcheat_1$ to be true, all the clause that ensure that *bcheat* is false only if a rule is broken become false. Therefore, the problem is unsatisfiable and the second player wins. By using this new variable *bcheat*, set in an if-and-only-if manner in all the rules related to the black player's moves, it becomes impossible to cheat without facing defeat. It gives, for each of the five constraints that define *wmove* (constraint 6 in Section 8.2), the following equivalence: $\neg$(constraint) $\Leftrightarrow bcheat_s$.

Finally, to enforce a win for the white player when black cheats, we add the following clause: $\forall s \in [1, S] \quad wwin \vee \neg bcheat_s$.

Now that we have ensured that both players will adhere to the rules (accounting for all scenarios where the universal quantifier would enable the second player to cheat), we can proceed by introducing existential quantifiers for the indicator variables. The variables *bcheat* and *wwin* are included as part of these indicators. The final representation of the quantified Boolean formulae is as follows:

$\exists\{black_{0,C,N}, white_{0,C,N}, occupied_{0,C,N}, \ldots, endgame_0, wwin\}$
$\exists\{wmove_{1,1,1}, wmove_{1,1,2}, \ldots, wmove_{1,n,n}\}$
$\forall\{bmove_{1,1,1}, bmove_{1,1,2}, \ldots, bmove_{1,n,n}\}$
$\exists\{$Indicator variables for step 1$\}$
$\exists\{wmove_{2,1,1}, wmove_{2,1,2}, \ldots, wmove_{2,n,n}\}$
$\vdots$
$\exists\{wmove_{S,1,1}, wmove_{S,1,2}, \ldots, wmove_{S,n,n}\}$
$\forall\{bmove_{S,1,1}, bmove_{S,1,2}, \ldots, bmove_{S,n,n}\}$
$\exists\{$Indicator variables for step S$\}$

In summary, we have developed an initial model consisting of variables and constraints that accurately define the game of Othello and its rules. This model enables analysis of various gameplay scenarios and to find the minimum number of moves to reach an endgame state. To determine the outcome of a game, we introduced Boolean variables to represent the winner and presented an initial approach to evaluate it. Additionally, in order to solve the Othello game, we introduced quantifiers and cheat variables to systematically explore all possible scenarios.

8.4 Future Works

To further expand on this work, an objective for the researchers is to implement these constraints using the proposed enhancements. The first objective starting from the proposed model is to reformulate, if possible, the constraints that use the higher number of clauses. Then, we can use the minimum number of moves required to set a lower bound on the number of tokens needed to win, thereby reducing the number of clauses required to define the variable *wwin*. The proposed version should be refined to make it more accessible. Finally, the main goal is to use the QBF approach with a suitable solver to solve the game of Othello on a 6x6 board, and potentially extend it to solve the 7x7 and 8x8 board if feasible.

9 Conclusion

In conclusion, this research project has successfully achieved its primary objectives, which had two main components. Firstly, it aimed to extend the application of propositional logic to games and problems that had not been previously approached in this manner, thereby introducing new directions of exploration. By doing so, it has opened up new possibilities for student research projects and offered fresh perspectives for students to delve into. Additionally, it aimed to improve problem-solving techniques in terms of computation times, with the primary goal of effectively managing and solving larger problem instances.

To accomplish these objectives, the project began by exploring various ways to model the problems, with a particular focus on propositional logic, in order to establish functional models. Subsequently, key factors influencing the efficiency and ease of search for the SAT solver were investigated. Different models were examined to assess the impact of constraint construction on overall efficiency. Notably, the project used the SAT solver *sat4j*, which operates on constraints in Conjunctive Normal Form (CNF).

In the domain of two-player games, Quantified Boolean Formulas (QBF) were introduced to allow for alternating moves between the players. With an existential and an universal player, we proposed a way of solving the game as depicted in Section 8.3. In the context of model checking, the evaluation primarily focused on breadth-first-search (BFS) and depth-first-search (DFS) techniques.

For the Snake Cube problem, we found out the significant benefit of incorporating problem symmetry into the constraints, resulting in improved efficiency and the successful resolution of a 4x4x4 cube. In the case of the Rubik's Cube, limitations were identified with the model checker, preventing the handling of the 3x3x3 cube. By adapting a propositional logic model to the 2x2x2 cube, it was observed that reducing the number of Boolean variables and clauses did not always lead to improved efficiency. However, this still demonstrates different ways of thinking when it comes to establishing constraints. Regarding the Othello game, the project primarily focused on providing an initial approach where we defined a model allowing us to determine the minimum number of moves to reach an endgame state. Moreover, there are guidelines to develop in order to solve the game in future research.

Overall, this work has explored the domain covered by propositional logic, opening directions for further investigation. The proposed ideas for further research are detailed in the "Future Works" sections of each respective chapter.

Finally, this project has not only allowed for personal growth and the expansion of knowledge in this field, but it also aims to inspire and engage new students and researchers in this fascinating and ever-evolving domain.

References

[1] aurelien. *Les Rubiks cube*. 2013. URL: http://rubikscube-aurelien.over-blog.com/l-algorithme-de-dieu.html.

[2] Jon Barwise. "An Introduction to First-Order Logic". In: *Handbook of Mathematical Logic*. Ed. by Jon Barwise. Studies in Logic and the Foundations of Mathematics. Amsterdam, NL: North-Holland, 1977. ISBN: 978-0-444-86388-1.

[3] Daniel Le Berre and Anne Parrain. "The Sat4j library, release 2.2". In: *J. Satisf. Boolean Model. Comput.* 7.2-3 (2010), pp. 59–6. DOI: 10.3233/sat190075. URL: https://doi.org/10.3233/sat190075.

[4] Daniel Le Berre and Stéphanie Roussel. "Sat4j 2.3.2: on the fly solver configuration System Description". In: *J. Satisf. Boolean Model. Comput.* 8.3/4 (2014), pp. 197–202. DOI: 10.3233/sat190098. URL: https://doi.org/10.3233/sat190098.

[5] Jerry Bryan. *Algorithme de Dieu pour le cube de poche 2x2x2*. Dec. 1993. URL: http://www.math.rwth-aachen.de/~Martin.Schoenert/Cube-Lovers/Jerry_Bryan__God's_Algorithm_for_the_2x2x2_Pocket_Cube.html.

[6] Jingchao Chen. "A New SAT Encoding of the At-Most-One Constraint". Unpublished. 2011.

[7] Jingchao Chen. "Solving Rubik's Cube Using SAT Solvers". In: *CoRR* abs/1105.1436 (2011). arXiv: 1105.1436. URL: http://arxiv.org/abs/1105.1436.

[8] Christinne Choppy. *Spécification des Systèmes Complexes (SDSC)*. Institut Galilée. Nov. 2022. URL: https://lipn.fr/~rodriguez/teach/sdsc/2015-16/.

[9] code_r. *FIFO vs LIFO approach in Programming*. GeeksforGeeks. URL: https://www.geeksforgeeks.org/fifo-vs-lifo-approach-in-programming/.

[10] Wikipedia contributors. *Propositional calculus - Wikipedia*. https://en.wikipedia.org/wiki/Propositional_calculus. 2023-05-31. 2023.

[11] *Deadlock, Livelock and Starvation*. Baeldung. Apr. 2021. URL: https://www.baeldung.com/cs/deadlock-livelock-starvation.

[12] Lux Edixhoven. "Attacking the n-puzzle using SAT solvers". MA thesis. Leiden Institute of Advanced Computer Science (LIACS), July 2016.

[13] Denes Ferenc. *Different Rubik's Cube Solving Methods*. https://ruwix.com/the-rubiks-cube/different-rubiks-cube-solving-methods/. Accessed on 2023-06-01.

[14] Pascal Fontaine. *Logics for Computer Science*. 2022.

[15] Pascal Fontaine. *Structure de données et algorithme*. 2020.

[16] Ian P Gent and Andrew G D Rowley. "Encoding Connect-4 using Quantified Boolean Formulae". In: *Unknown* (Aug. 2003).

[17] Gerard J. Holzmann. *Spin Model Checker*. ACM Digital Library. URL: https://spinroot.com/spin/whatispin.html (visited on 11/2022).

[18] Internet encyclopedia of philosophy contributors James Fieser Bradley Dowden. *Propositional Logic | Internet Encyclopedia of Philosophy*. https://iep.utm.edu/propositional-logic-sentential-logic/. 2023-05-31.

[19] Tommi Junttila. *CNF Translations*. https://users.aalto.fi/~tjunttil/2020-DP-AUT/notes-sat/cnf2.html. 2023-05-31. 2020.

[20] Tommi Junttila. *Conflict-driven clause learning (CDCL) SAT solvers*. https://users.aalto.fi/~tjunttil/2020-DP-AUT/notes-sat/cdcl.html. 2023-05-31. 2020.

[21] Tommi Junttila. *DNF — The Disjunctive Normal Form*. https://users.aalto.fi/~tjunttil/2020-DP-AUT/notes-sat/dnf.html. 2023-05-31. 2020.

[22] K1ntus. *Othello-solver*. GitHub. Jan. 2023. URL: https://github.com/K1ntus/Othello-Solver.

[23] Chunxiao Li et al. "Towards a Complexity-Theoretic Understanding of Restarts in SAT Solvers". In: *Theory and Applications of Satisfiability Testing – SAT 2020*. Ed. by Luca Pulina and Martina Seidl. Cham: Springer International Publishing, 2020, pp. 233–249. ISBN: 978-3-030-51825-7.

[24] nimble-code. *Exécutables précompilés de Spin*. https://spinroot.com/spin/Bin/. Nov 2022. spinroot.

[25] OW2 Consortium. *Index of /sat4j*. https://release.ow2.org/sat4j/. OW2 Consortium, Nov. 2022.

[26] PHILIPPE PICART. *The Rubik's Cube*. Nov. 2022. URL: http://trucsmaths.free.fr/rubik.htm.

[27] Knot Pipatsrisawat and Adnan Darwiche. "A Lightweight Component Caching Scheme for Satisfiability Solvers". In: *Theory and Applications of Satisfiability Testing - SAT 2007, 10th International Conference, Lisbon, Portugal, May 28-31, 2007, Proceedings*. Ed. by João Marques-Silva and Karem A. Sakallah. Vol. 4501. Lecture Notes in Computer Science. Springer, 2007, pp. 294–299. DOI: 10.1007/978-3-540-72788-0_28. URL: https://doi.org/10.1007/978-3-540-72788-0%5C_28.

[28] F. Pittner. *Solving the 6x6 normal othello game*. July 2006. URL: http://www.tothello.com/html/solving_the_6x6_normal.html.

[29] Simon Prince. *Tutorial #9: SAT Solvers I: Introduction and applications*. Borealis AI. Nov. 2020. URL: https://www.borealisai.com/research-blogs/tutorial-9-sat-solvers-i-introduction-and-applications/.

[30] *Règles du jeu d'Othello/Reversi*. Fédération Française d'Othello. URL: https://www.ffothello.org/othello/regles-du-jeu/.

[31] César Rodríguez. *Introduction à l'outil Spin et au langage Promela*. Institut Galilée. Nov. 2022. URL: https://lipn.fr/~rodriguez/teach/sdsc/2015-16/files/01-cours.pdf.

[32] *Solveur de puissance 4*. Game Solver. URL: https://connect4.gamesolver.org/.

[33] Zhouheng (Jeffrey) Sun. *How to Model a Rubik's Cube and Build a Solver*. Dec. 2020. URL: https://observablehq.com/@onionhoney/how-to-model-a-rubiks-cube.

[34] *Tcl/Tk Software*. https://www.tcl.tk/software/tcltk/bindist.html. Tcl Developer Xchange site, Nov. 2022.

[35] Wikipedia contributors. *Model Checking*. https://en.wikipedia.org/wiki/Model_checking. May 30, 2023. 2023.

A Action function of the Rubik's Cube

The actions are applicable for all values of s in the range [1, 11] and all values of l in the range [1, 3].

Action	Facelets of rotating face	Facelets of side faces
A_1: 90° clockwise top face	$A_1(c(1,1,s,l)) = c(1,3,s-1,l)$ $A_1(c(1,2,s,l)) = c(1,1,s-1,l)$ $A_1(c(1,4,s,l)) = c(1,2,s-1,l)$ $A_1(c(1,3,s,l)) = c(1,4,s-1,l)$	$A_1(c(2,1,s,l)) = c(5,1,s-1,l)$ $A_1(c(2,2,s,l)) = c(5,2,s-1,l)$ $A_1(c(3,1,s,l)) = c(2,1,s-1,l)$ $A_1(c(3,2,s,l)) = c(2,2,s-1,l)$ $A_1(c(4,1,s,l)) = c(1,1,s-1,l)$ $A_1(c(4,2,s,l)) = c(1,2,s-1,l)$ $A_1(c(5,1,s,l)) = c(1,1,s-1,l)$ $A_1(c(5,2,s,l)) = c(1,2,s-1,l)$
A_2: 180° top face	$A_2(c(1,1,s,l)) = c(1,4,s-1,l)$ $A_2(c(1,4,s,l)) = c(1,1,s-1,l)$ $A_2(c(1,2,s,l)) = c(1,3,s-1,l)$ $A_2(c(1,3,s,l)) = c(1,2,s-1,l)$	$A_2(c(2,1,s,l)) = c(4,1,s-1,l)$ $A_2(c(2,2,s,l)) = c(4,2,s-1,l)$ $A_2(c(3,1,s,l)) = c(5,1,s-1,l)$ $A_2(c(3,2,s,l)) = c(5,2,s-1,l)$ $A_2(c(4,1,s,l)) = c(2,1,s-1,l)$ $A_2(c(4,2,s,l)) = c(2,2,s-1,l)$ $A_2(c(5,1,s,l)) = c(3,1,s-1,l)$ $A_2(c(5,2,s,l)) = c(3,2,s-1,l)$
A_3: 90° counterclockwise top face	$A_3(c(1,1,s,l)) = c(1,2,s-1,l)$ $A_3(c(1,2,s,l)) = c(1,4,s-1,l)$ $A_3(c(1,4,s,l)) = c(1,3,s-1,l)$ $A_3(c(1,3,s,l)) = c(1,1,s-1,l)$	$A_3(c(2,1,s,l)) = c(3,1,s-1,l)$ $A_3(c(2,2,s,l)) = c(3,2,s-1,l)$ $A_3(c(3,1,s,l)) = c(4,1,s-1,l)$ $A_3(c(3,2,s,l)) = c(4,2,s-1,l)$ $A_3(c(4,1,s,l)) = c(5,1,s-1,l)$ $A_3(c(4,2,s,l)) = c(5,2,s-1,l)$ $A_3(c(5,1,s,l)) = c(2,1,s-1,l)$ $A_3(c(5,2,s,l)) = c(2,2,s-1,l)$
A_4: 90° clockwise left face	$A_4(c(3,1,s,l)) = c(3,3,s-1,l)$ $A_4(c(3,2,s,l)) = c(3,1,s-1,l)$ $A_4(c(3,4,s,l)) = c(3,2,s-1,l)$ $A_4(c(3,3,s,l)) = c(3,4,s-1,l)$	$A_4(c(1,1,s,l)) = c(4,4,s-1,l)$ $A_4(c(1,3,s,l)) = c(4,2,s-1,l)$ $A_4(c(2,1,s,l)) = c(1,1,s-1,l)$ $A_4(c(2,3,s,l)) = c(1,3,s-1,l)$ $A_4(c(6,1,s,l)) = c(2,1,s-1,l)$ $A_4(c(6,3,s,l)) = c(2,3,s-1,l)$ $A_4(c(4,4,s,l)) = c(6,1,s-1,l)$ $A_4(c(4,2,s,l)) = c(6,3,s-1,l)$

A_5: 180° left face	$A_5(c(3,1,s,l)) = c(3,4,s-1,l)$ $A_5(c(3,4,s,l)) = c(3,1,s-1,l)$ $A_5(c(3,2,s,l)) = c(3,3,s-1,l)$ $A_5(c(3,3,s,l)) = c(3,2,s-1,l)$	$A_5(c(1,1,s,l)) = c(6,1,s-1,l)$ $A_5(c(1,3,s,l)) = c(6,3,s-1,l)$ $A_5(c(6,1,s,l)) = c(1,1,s-1,l)$ $A_5(c(6,3,s,l)) = c(1,3,s-1,l)$ $A_5(c(2,1,s,l)) = c(4,4,s-1,l)$ $A_5(c(2,3,s,l)) = c(4,2,s-1,l)$ $A_5(c(4,4,s,l)) = c(2,1,s-1,l)$ $A_5(c(4,2,s,l)) = c(2,3,s-1,l)$
A_6: 90° counterclockwise left face	$A_6(c(3,1,s,l)) = c(3,2,s-1,l)$ $A_6(c(3,2,s,l)) = c(3,4,s-1,l)$ $A_6(c(3,4,s,l)) = c(3,3,s-1,l)$ $A_6(c(3,3,s,l)) = c(3,1,s-1,l)$	$A_6(c(1,1,s,l)) = c(2,1,s-1,l)$ $A_6(c(1,3,s,l)) = c(2,3,s-1,l)$ $A_6(c(2,1,s,l)) = c(6,1,s-1,l)$ $A_6(c(2,3,s,l)) = c(6,3,s-1,l)$ $A_6(c(6,1,s,l)) = c(4,4,s-1,l)$ $A_6(c(6,3,s,l)) = c(4,2,s-1,l)$ $A_6(c(4,4,s,l)) = c(1,1,s-1,l)$ $A_6(c(4,2,s,l)) = c(1,3,s-1,l)$
A_7: 90° clockwise front face	$A_7(c(2,1,s,l)) = c(2,3,s-1,l)$ $A_7(c(2,2,s,l)) = c(2,1,s-1,l)$ $A_7(c(2,4,s,l)) = c(2,2,s-1,l)$ $A_7(c(2,3,s,l)) = c(2,4,s-1,l)$	$A_7(c(1,3,s,l)) = c(3,4,s-1,l)$ $A_7(c(1,4,s,l)) = c(3,2,s-1,l)$ $A_7(c(3,4,s,l)) = c(6,2,s-1,l)$ $A_7(c(3,2,s,l)) = c(6,1,s-1,l)$ $A_7(c(6,2,s,l)) = c(5,1,s-1,l)$ $A_7(c(6,1,s,l)) = c(5,3,s-1,l)$ $A_7(c(5,1,s,l)) = c(1,3,s-1,l)$ $A_7(c(5,3,s,l)) = c(1,4,s-1,l)$
A_8: 180° front face	$A_8(c(2,1,s,l)) = c(2,4,s-1,l)$ $A_8(c(2,4,s,l)) = c(2,1,s-1,l)$ $A_8(c(2,2,s,l)) = c(2,3,s-1,l)$ $A_8(c(2,3,s,l)) = c(2,2,s-1,l)$	$A_8(c(1,3,s,l)) = c(6,4,s-1,l)$ $A_8(c(1,4,s,l)) = c(6,2,s-1,l)$ $A_8(c(6,2,s,l)) = c(1,3,s-1,l)$ $A_8(c(6,1,s,l)) = c(1,4,s-1,l)$ $A_8(c(3,4,s,l)) = c(5,1,s-1,l)$ $A_8(c(3,2,s,l)) = c(5,3,s-1,l)$ $A_8(c(5,1,s,l)) = c(3,4,s-1,l)$ $A_8(c(5,3,s,l)) = c(3,2,s-1,l)$
A_9: 90° counterclockwise front face	$A_9(c(2,1,s,l)) = c(2,2,s-1,l)$ $A_9(c(2,2,s,l)) = c(2,4,s-1,l)$ $A_9(c(2,4,s,l)) = c(2,3,s-1,l)$ $A_9(c(2,3,s,l)) = c(2,1,s-1,l)$	$A_9(c(1,3,s,l)) = c(5,1,s-1,l)$ $A_9(c(1,4,s,l)) = c(5,3,s-1,l)$ $A_9(c(3,4,s,l)) = c(1,3,s-1,l)$ $A_9(c(3,2,s,l)) = c(1,4,s-1,l)$ $A_9(c(6,2,s,l)) = c(3,4,s-1,l)$ $A_9(c(6,1,s,l)) = c(3,2,s-1,l)$ $A_9(c(5,1,s,l)) = c(6,2,s-1,l)$ $A_9(c(5,3,s,l)) = c(6,1,s-1,l)$

B Initial States for Testing the Rubik's Cube Models

Twelve initial states are used to compute the average computation time of the SAT solver. The first digit corresponds to the color of facelet one of face one and so on for each facelet of each face, as shown in Figure 7.

1. 510250321251523343144400
2. 255025405011310431442332
3. 002443015035425513423211
4. 035521414041532230312405
5. 223014511500543321302445
6. 515502002314504213432341
7. 105124334020223134055451
8. 040431155535320120342142
9. 353044020122340421555113
10. 540033023554111243022514
11. 441354131325234501522000
12. 225330535140114054422130