University of Brasília – UnB
Faculty of Science and Engineering Technologies – FCTE
Software Engineering

Optimisation of SAT-Based Planning via Hybrid and Prioritised Asynchronous Binary Search Strategies: An extension of Madagascar Planner

Author: Bruno Campos Ribeiro, Igor e Silva Penha
Supervisor: Prof. Dr. Bruno Cesar Ribas

Brasília, DF
2025

Bruno Campos Ribeiro, Igor e Silva Penha

Optimisation of SAT-Based Planning via Hybrid and Prioritised Asynchronous Binary Search Strategies: An extension of Madagascar Planner

Monograph submitted to the undergraduate degree in Software Engineering at the University of Brasília, as a partial requirement for obtaining the Bachelor's degree in Software Engineering.

University of Brasília – UnB

Faculty of Science and Engineering Technologies – FCTE

Supervisor: Prof. Dr. Bruno Cesar Ribas

Brasília, DF

2025

Abstract

This undergraduate dissertation investigates the Planning as Satisfiability (SAT) paradigm, a prominent approach in automated planning in which classical planning problems are translated into propositional logic formulae and solved by SAT solvers. A comprehensive review of the theoretical foundations of automated planning and propositional satisfiability is presented, followed by an in-depth analysis of major SAT-based planners, including the SATPLAN family, Blackbox and Madagascar. Emphasis is placed on the evolution of encoding strategies, the use of planning graphs, and the impact of mutex propagation on solver performance. Based on this theoretical framework, the present work introduces the design, implementation, and evaluation of a set of search algorithms integrated into the Madagascar planning system. These algorithms encompass sequential, heuristic, exponential, backward, and asynchronous parallel strategies for determining the optimal planning horizon under the SAT-based formulation. The primary objective is to reduce both the number of calls to the SAT solver and the time required to obtain the smallest feasible planning horizon. An extensive experimental evaluation demonstrates that the implemented strategies improve the efficiency of SAT-based planning.

Key-words: Artificial Intellingence, Automated Planning, Satisfiability, SAT Solver, Propositional Logic, Planning as SAT, PDDL, Madagascar.

List of Figures

Figure 1 – Aristotle's square of opposition. 13

List of Tables

Table 1 – Evolution of SATPLAN Systems 30
Table 2 – Search Algorithm Modes Integrated into Madagascar 41
Table 3 – Benchmark Results for Proposed Algorithms 58
Table 4 – Performance comparison using the PAR-2 metric (60-second timeout) . 67

List of abbreviations and acronyms

AST	Abstract Syntax Tree
AI	Artificial Intelligence
CDCL	Conflict-Driven Clause Learning
CNF	Conjunctive Normal Form
CPU	Central Processing Unit
CSP	Constraint Satisfaction Problems
IPC	International Planning Competition
KB	Knowledge Base
PDDL	Planning Domain Definition Language
SAT	Satisfiability
SCC	Strongly Connected Components
UNSAT	Unsatisfiability

Contents

1 INTRODUCTION . . . 9

2 THEORETICAL FRAMEWORK . . . 11

2.1 Principles of Automated Planning . . . 11

2.2 Fundamental Logic of Satisfiability . . . 12

2.3 The Concept of Satisfiability . . . 13

2.3.1 Solving Satisfiability Problems . . . 14

2.4 The Language of Planning Problems . . . 14

2.4.1 Representation of States . . . 15

2.4.2 Representation of Goals . . . 15

2.4.3 Representation of actions . . . 15

2.4.4 The Planning Domain Definition Language (PDDL) . . . 16

2.5 Planning with propositional logic . . . 16

2.5.1 Describing planning problems in propositional logic . . . 17

2.5.2 Planning Approaches . . . 19

2.6 Planning as SAT . . . 21

2.6.1 SatPlan 1992 . . . 21

2.6.2 SatPlan 1996 . . . 21

2.6.2.1 Linear Encoding . . . 22

2.6.2.2 Operator Splitting e Explanatory Frame Axioms . . . 23

2.6.2.3 Parallel and Planning Graph-Based Encodings . . . 24

2.6.2.4 Lifted Causal Encodings . . . 25

2.6.3 SatPlan 1999 - BlackBox . . . 26

2.6.3.1 Blackbox Architecture . . . 26

2.6.3.2 Improvements over SATPLAN96 . . . 26

2.6.4 SatPlan 2004 . . . 28

2.6.4.1 SatPlan04 Architecture . . . 28

2.6.4.2 Encoding Styles . . . 28

2.6.5 SatPlan 2006 . . . 29

2.6.5.1 Improvements over SATPLAN04 . . . 29

2.6.6 MADAGASCAR - Parallel Encodings of Classical Planning as Satisfiability . 30

2.6.6.1 Step Semantics . . . 30

2.6.6.2 Process Semantics . . . 31

2.6.6.3 1-Linearization Semantics . . . 31

2.6.6.4 Algorithm S: Sequential Evaluation . . . 32

2.6.6.5 Algorithm A: Parallel Evaluation with n Processes . . . 32
2.6.6.6 Algorithm B: Geometric CPU Time Distribution . . . 33
2.6.7 Efficient Encoding of Cost Optimal Delete-Free Planning as SAT . . . 33
2.6.7.1 Causal Representations of Relaxed Plans . . . 34
2.6.7.2 SAT Encoding . . . 34
2.6.7.3 Cost Propagation and Minimization . . . 35

3 AN EXTENSION OF MADAGASCAR PLANNER . . . 37
3.1 Architecture of Madagascar . . . 37
3.1.1 Input and Parsing Module . . . 37
3.1.2 Preprocessing and Grounding Module . . . 37
3.1.3 Translation and Search Strategy Module . . . 38
3.1.4 SAT Solver Module . . . 38
3.1.5 Data Structures and Utilities . . . 39
3.2 Implementation of the Search Strategies . . . 39
3.2.1 Command-Line Interface . . . 41
3.2.2 Adapting the Solver Interface: The `computeonestepBB` Function . . . 42
3.2.3 Implemented Search Algorithms . . . 43
3.2.3.1 Sequential Binary Search (`runalgorithmBB`) . . . 43
3.2.3.2 Binary Search with Dynamic Probing (`runalgorithmBBD`) . . . 45
3.2.3.3 Asynchronous k-ary Binary Search via Round Robin (`runalgorithmBBA`) . . . 47
3.2.3.4 Asynchronous Prioritized k-ary Search (`runalgorithmBBB`) . . . 49
3.2.3.5 Hybrid Strategies (`runalgorithmBBDA` and `runalgorithmBBDB`) . . . 51
3.2.3.6 Exponential Search (`runalgorithmExp`) . . . 52
3.2.3.7 Backward Search Optimized by Action Counting (`runalgorithmBBD_LR`) . . . 53

4 EMPIRICAL EVALUATION . . . 55

5 CONCLUSION . . . 59
5.1 Future Work . . . 60

Postscript: Methodological Notes and Lessons Learned . . . 62
The Genesis and Early Inspiration Underpinning the Concept . . . 62
From Planning to Plan: A Review of the Development Process . . . 63
Critical Readings . . . 64
Overview of Publications and Contributions . . . 65
bni: A PDDL Parser, REPL and Validate Tool . . . 65
Mojified Pac-Man . . . 66
Experimental Study with PluSAT . . . 66

REFERENCES 68

1 Introduction

Automated planning (HASLUM et al., 2019) is a well-established branch of Artificial Intelligence (AI), concerned with the development of systems capable of generating sequences of actions that transition an environment from an initial state to a desired goal. Over the years, the field has evolved significantly, incorporating techniques from logic, search algorithms, and satisfiability solving. Among the most influential modeling standards is the Planning Domain Definition Language (PDDL) (MCDERMOTT et al., 1998), which offers a formal syntax for specifying planning problems in a domain-independent manner.

Among the different approaches to automated planning, the Planning as Satisfiability paradigm has gained particular relevance. In this approach, planning problems are translated into Boolean formulas, usually in conjunctive normal form (CNF), and solved using Boolean satisfiability (SAT) solvers. This translation allows planning to benefit directly from the advances achieved by modern SAT solvers. Classical systems such as SATPLAN and Blackbox demonstrated the practical viability of this methodology by competitive results in international planning competitions.

Similarly, Madagascar (RINTANEN; HELJANKO; NIEMELÄ, 2006) emerges as an enhanced extension of SATPLAN, building upon advances in CNF problem encodings, incremental solving techniques, and in the interaction with modern SAT solvers. Its architecture is designed to generate and solve a sequence of SAT instances corresponding to increasing plan lengths, efficiently refining the planning horizon until a valid plan is found. These improvements provide high performance across a wide range of domains, establishing Madagascar as a reference planner within the Planning as Satisfiability paradigm.

Despite its efficiency, several limitations remain in the current version of Madagascar. In particular, the planner lacks native support for cost-aware plan generation, exhibits redundancy in certain parallel encodings, and relies primarily on sequential strategies for evaluating candidate planning horizons. Moreover, the identification of optimal plans often requires the exhaustive evaluation of multiple unsatisfiable instances, which can be computationally expensive.

To address these challenges, the main goal of this work is to extend the Madagascar planner with efficient binary search based strategies for identifying optimal planning horizons in SAT-based planning. By replacing purely sequential evaluation with more efficient sequential, parallel, asynchronous, and hybrid binary search strategies, the planner can more rapidly converge to optimal or near-optimal solutions. These strategies aim to reduce the computational effort required to determine the minimal satisfiable horizon

while preserving the soundness and completeness.

2 Theoretical Framework

This chapter presents the theoretical foundation necessary for understanding the relationship between SAT and automated planning. First, the main concepts and principles underlying SAT are introduced from the perspective of propositional logic, including its formal definition and its significance in computational complexity theory. Next, automated planning is explored, discussing its relevance in artificial intelligence and how planning problems may be represented and solved through SAT formulations. By establishing this theoretical basis, the study aims to contextualise the proposed research within the broader literature on automated reasoning and problem-solving.

2.1 Principles of Automated Planning

In AI planning, the integration of knowledge representation methods and problem-solving techniques (BENCH-CAPON, 2014) plays a crucial role. The process begins with the appropriate representation of the problem, which is essential for the effective application of AI methods such as heuristic search. Often, it is necessary to develop efficient heuristics to optimise the solution search process.

A fundamental aspect of AI planning is the introduction of a problem description language, specifically the Planning Domain Definition Language (MCDERMOTT, 2000). PDDL provides an formal representation (HASLUM et al., 2019) for expressing state transformation problems. Planning systems, or "planners", use this formal modelling as input. They employ specific problem-solving algorithms, such as heuristic search methods or techniques based on SAT solving (JEROSLOW; WANG, 1990), to find valid solutions to the planning task. For the planning algorithm designer, the challenge lies in transforming this modelling into a search space or logical reasoning problem, in addition to developing effective heuristics for efficient problem-solving. In "An Introduction to the Planning Domain Definition Language" (HASLUM et al., 2019), Planning is defined as:

> "AI planning can be described as the study of computational models and methods of creating, analysing, managing, and executing plans."

The majority of planners in mainstream use today are characterised by domain independence, which enables them to be applied to any problem modelled in a specified description language. Although not all planners are capable of solving all types of problems, those that are not referred to as domain-specific planners. Nowadays, the PDDL is widely used and is particularly relevant in international planning competitions, such

as the International Planning Competition (IPC) (IPC, 2023). The study of PDDL encompasses not only its syntax and semantics, but also its extensions, which allow for the representation of different types of planning problems.

AI planning heavily relies on the analysis of action consequences and the efficient identification of a course of action leading to a desirable outcome. In this context, automated planning focuses particularly on the task of finding a plan. Here, the formal description of the problem provides a valuable predictive model, allowing for the deduction of possible actions and their respective results. Thus, the AI planning process not only seeks to predict the consequences of actions but also promotes the optimisation of formulated plans to achieve pre-established objectives.

2.2 Fundamental Logic of Satisfiability

Logic as a science was developed in Athens by Aristotle between 384 and 322 B.C. In his work "Organon" (OWEN et al., 1899), he established foundational principles of logic that prevailed for nearly 2000 years. He devised syllogistic or categorical logic, characterised by simple subject-predicate sentence structures. Aristotle and his disciples regarded language as a medium to express ideas reflecting entities and their properties in the external world. They perceived that concepts are combine in the mind to create subject-predicate propositions, akin to being conscious of two concepts — the subject S and the predicate P — simultaneously. Aristotle identified four ways to capture these propositions in relation to a certain "world" w, also knows as Aristotle's square of opposition Figure 1, based on universal or particular, positive or negative relationships: every S is P, no S is P, some S is P, and some S is not P. These were categorised as A (universal affirmative), E (universal negative), I (particular affirmative), and O (particular negative) propositions, each with defined truth conditions.

$$A: \quad \text{Every } S \text{ is } P \text{ is true in } w \iff \forall x(S(x,w) \rightarrow P(x,w))$$

$$E: \quad \text{No } S \text{ is } P \text{ is true in } w \iff \neg\exists x(S(x,w) \land P(x,w))$$

$$I: \quad \text{Some } S \text{ is } P \text{ is true in } w \iff \exists T \forall x((S(x,w) \land P(x,w)) \rightarrow (P(x,w) \land T(x,w)))$$

$$O: \quad \text{Some } S \text{ is not } P \text{ is true in } w \iff \neg\forall x\,(S(x,w) \rightarrow P(x,w))$$

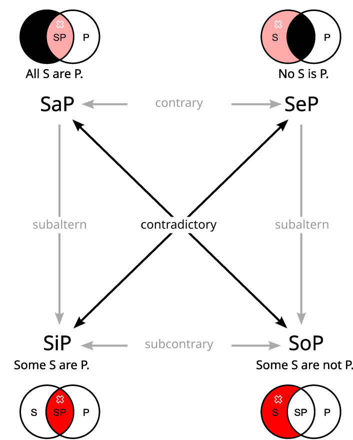

Figure 1 – Aristotle's square of opposition.

George Boole (1815-1864) significantly advanced logic through the development of Boolean algebra (MORETTI, 2012), a structure denoted as $\langle B, \vee, \wedge, \neg, 0, 1\rangle$, where $\vee$ and $\wedge$ are binary operations, and $\neg$ is a unary operation that keeps B closed, with 0 and 1 belonging to B. His key innovation was creating an algebraic notation for fundamental set properties, aiming to provide a broader theory of term logic, encompassing traditional syllogistic logic. Boole was among the earliest to use symbolic language, employing term variables s, t, u, v, w, x, y, z *etc.* to represent sets and introducing standard Boolean operations such as complementation, union, and intersection. He established standard laws of Boolean algebra like association, commutation, and distribution, as well as properties of complementation and the universal and empty sets. Additionally, Boole encoded Aristotle's four categorical propositions using symbolic representation.

2.3 The Concept of Satisfiability

The concept of satisfiability applies to both individual formulas and theories, which are collections of formulas. A theory is considered satisfiable if there is an interpretation that makes every formula true, and is valid if every formula is true in all interpretations (BIERE; HEULE; MAAREN, 2009). This notion is important in evaluating the consistency of a theory, and in first-order logic, it is connected to Gödel's completeness theorem (HENKIN, 1950). The opposite notions of satisfiability and truth are unsatisfiability and untruth, respectively, and they relate to each other in a way similar to Aristotle's square of opposition.

In propositional logic, determining whether a formula is satisfiable is a decidable problem, known as the (SAT) problem. However, for first-order logic sentences, determining satisfiability is generally undecidable. In fields such as universal algebra, equational theory, and automated theorem proving, methods such as term rewriting, congruence

closure, and unification are used to address satisfiability (BAADER, 2003). Whether a particular theory is decidable generally depends on whether it is variable-free and on other conditions.

2.3.1 Solving Satisfiability Problems

When it comes to solvers, a SAT solver is a computer programme which aims to resolve Boolean satisfiability problems (SAT formulas). Basically, it takes a Boolean formula as input, such as $(x_0 \vee x_1) \wedge (x_0 \vee \neg x_1)$, and outputs whether the formula is satisfiable, meaning that there are possible values of x_0 and x_1 which make the formula true, or unsatisfiable, meaning that there are no such values of x_0 and x_1, which make the formula true.

The formula received is in CNF. The CNF is a standardised way of writing Boolean formulas in logic, where the formula is composed of a conjunction (AND, denoted as $\wedge$) of one or more clauses, and each clause is a disjunction (OR, denoted as $\vee$) of literals (a variable or its negation). For example, $(x_0 \vee x_1) \wedge (x_0 \vee \neg x_1)$ is in CNF.

The decision-making challenge in propositional logic, often known as the (SAT) problem, is famously NP-complete (COOK, 2023). While NP-completeness suggests (assuming the reasonable hypothesis that $P \neq NP$) that any comprehensive SAT algorithm operates in exponential time in the worst-case scenario, SAT solvers frequently perform better than this worst-case prediction. Numerous algorithms have been developed over the years to tackle SAT problems. In this paper, we will examine some of these algorithms in the following sections.

2.4 The Language of Planning Problems

The formalism used to express planning problems enables automated reasoning about actions and their consequences. By employing logical languages, a diverse range of real-world domains can be translated into representations suitable for computational planning. Classical approaches describe states and goals using logical literals, and define actions in terms of their preconditions and effects. While restrictions imposed by early formalisms such as STRIPS[1] promote computational efficiency, practical applications often require greater expressiveness, which has led to extensions allowing for richer state and action descriptions. A detailed account of the languages and formalisms used in the representation of classical planning can be found in the Chapter 11, Planning, in the book "Artificial Intelligence: A Modern Approach" (RUSSELL; NORVIG, 2016).

[1] STRIPS stands for STanford Research Institute Problem Solver.

2.4.1 Representation of States

Planners decompose the world into logical conditions and represent a state as conjunction of positive literals. We will consider propositional literals; for example, $Poor \wedge Unknown$ might represent the state of a hapless agent. We will also use first-order literals; for example, $At(Plane_1, Accra) \wedge At(Plane_2, Brasília)$, to represent a state in the package delivery problem.

Literals in first-order state descriptions must be **ground** and **function-free**. A literal is said to be ground if it contains no variables, that is, all its arguments are constants denoting concrete objects of the domain. This restriction ensures that states describe only fully instantiated facts rather than schematic or generic conditions. Moreover, literals must be function-free, meaning that no function symbols are allowed in their arguments; consequently, terms such as nested or composed expressions are excluded, and only constant symbols may occur. This guarantees a finite set of possible ground atoms and, therefore, a finite state space. An *atom* is a formula of the form $(P(t_1, \ldots, t_n))$, where (P) is an (n)-ary predicate symbol and $(t_1, \ldots, t_n)$ are terms; atoms constitute the most basic assertions about the domain and contain no logical connectives or quantifiers. A literal is either an atom or its negation. As a result, literals such as $At(x, y)$ or $At(Father(Fred), Brasília)$ are not allowed. The **closed-world assumption** is used, meaning that any conditions that are not mentioned in a state are assumed false.

2.4.2 Representation of Goals

A goal is a partially specified state, represented as a conjunction of positive ground literals, such as $Rich \wedge Famous$ or $At(P_2, Tahiti)$. A propositional state s **satisfies** a goal g if s contains all the atoms in g (and possibly others). For example, the state $Rich \wedge Famous \wedge Miserable$ satisfies the goal $Rich \wedge Famous$.

2.4.3 Representation of actions

An action is specified in terms of the preconditions that must hold before it can be executed and the effect that ensue when it is executed. For example, an action for flying a plane from one location to another is:

$$
\begin{aligned}
&Action(Fly(p, from, to),\\
&\quad PRECOND : At(p, from) \wedge Plane(p) \wedge Airport(from) \wedge Airport(to)\\
&\quad EFFECT : \neg At(p, from) \wedge At(p, to)
\end{aligned}
$$

2.4.4 The Planning Domain Definition Language (PDDL)

To facilitate interoperability and benchmarking within the automated planning community, the PDDL (MCDERMOTT et al., 1998) was developed as a unifying formalism. First introduced for the inaugural International Planning Systems Competition in 1998 (MCDERMOTT, 2000), PDDL represented a collective endeavour led by leading researchers to provide a common framework for the specification of planning domains and problems. Since its inception, PDDL has undergone multiple revisions, with subsequent editions of the language introducing additional features aimed at accommodating a broader range of planning challenges, such as numeric reasoning and temporal constraints. These enhancements have often been motivated by the evolving requirements of the International Planning Competition (IPC, 1998) itself.

Although PDDL aspires to function as a shared modelling language, it should not be mistaken for an official standard. Its role is best seen as a community-driven approach to enabling the exchange and reuse of planning systems and models. The lack of a definitive and completely unambiguous specification, particularly given the variation in language versions and stylistic differences in supporting documentation, means that the interpretation of certain constructs can differ between planning systems. In practice, most planners implement only a subset of features from specific versions, sometimes interpreting ambiguous elements according to their own conventions.

Despite these variations, there exists a broadly recognised core within PDDL that is widely understood and utilised across the community. Discourse and scholarship generally centre on this consensus core, incorporating key aspects from various versions whilst eschewing features that are poorly supported or ambiguously defined. It is common practice to highlight areas where the semantics or syntax may be open to interpretation, ensuring that users of planning technology are aware of potential discrepancies between implementations.

2.5 Planning with propositional logic

The Planning with propositional logic approach is based on testing the satisfiability of a logical sentence rather than proving a theorem. We are looking for models of propositional sentences that look like this:

$$\textit{initial state} \wedge \textit{all possible actions descriptions} \wedge \textit{goal state}.$$

The statement incorporates proposition symbols for every possible action occurrence. A model that satisfies the statement assigns *true* to actions that are part of a valid plan and $false$ to those that are not. An assignment corresponding to an invalid plan is

not considered a model, as it contradicts the assertion that the goal is achievable. If the planning problem is unsolvable, the statement is regarded as unsatisfiable.

2.5.1 Describing planning problems in propositional logic

The process of translating $STRIPS$ problems into propositional logic exemplifies the knowledge representation cycle. This process commences with the establishment of a reasonable set of axioms, followed by the identification of any unintended models that may emerge, leading to the refinement of these axioms.

To illustrate this process, (RUSSELL; NORVIG, 2016) proposes a basic air transport problem. In the initial state (time 0), plane P_1 is situated at SFO, while plane P_2 is located at JFK. The objective is to reposition P_1 at JFK and P_2 at SFO, effectively facilitating a swap between the two aircraft. Distinct proposition symbols for each time step are required, with superscripts employed to denote the specific time. The initial state can be formally represented as

$$At(P_1, SFO)^0 \wedge At(P_2, JFK)^0.$$

Given that propositional logic does not operate under a closed-world assumption, it is imperative to specify propositions that are not *true* in the initial state. In instances where certain propositions remain unknown, they may be left unspecified, thereby adopting an **open-world assumption**. In this particular example, the following propositions are specified:

$$\neg At(P_1, JFK)^0 \wedge \neg At(P_2, SFO)^0.$$

The goal must be linked to a specific time step. As the number of steps required to achieve the goal is not known in advance, it is feasible to initially assert that the goal is true at time $T = 0$, represented as $At(P_1, JFK)^0 \wedge At(P_2, SFO)^0$. If this assertion fails, the process is repeated for $T = 1$, and so forth, until the minimum feasible plan length is determined. For each value of T, the knowledge base will encompass only the sentences relevant to the time steps from 0 to T. To ensure the end of the algorithm, an arbitrary upper limit, $T_{\max}$, is established. This algorithm is illustrated in the following *pseudo-code* on Listing 2.1.

Listing 2.1 – SATPLAN pseudo-code algorithm

```
1 function SATPLAN(problem, T_max) returns solution or failure
2     inputs: problem, a planning problem
3     T_max, an upper limit for plan length
4     for T = 0 to T_max do
5         cnf, mapping <- TRANSLATE-TO-SAT(problem, T)
6         assignment <- SAT-SOLVER(cnf)
7         if assignment is not null then
8             return EXTRACT-SOLUTION(assignment , mapping)
9     return failure
```

The next challenge is to encode action descriptions within propositional logic. The most straightforward method is to assign a unique proposition symbol for each action occurrence; for example, $Fly(P_1, SFO, JFK)^0$ is true if plane P_1 flies from SFO to JFK at time 0. Propositional versions of the successor-state axioms are employed to represent these actions.

For instance, the relationship can be expressed as follows:

$At(P_1, JFK)^1 \Leftrightarrow (At(P_1, JFK)^0 \wedge \neg(Fly(P_1, JFK, SFO)^0 \wedge At(P_1, JFK)^0)) \vee (Fly(P_1, SFO, JFK)^0 \wedge At(P_1, SFO)^0)$.

This indicates that plane P_1 will be at JFK at time 1 if it was at JFK at time 0 and did not fly away, or if it was at SFO at time 0 and flew to JFK. It is necessary to formulate one such axiom for each plane, airport, and time step. Additionally, the inclusion of each new airport introduces further travel options to and from a given airport, thereby increasing the number of disjuncts on the right-hand side of each axiom.

With these axioms in place, we can run the satisfiability algorithm to find a plan. There ought to be a plan that achieves the goal at time $T = 1$, namely, the plan in which the two planes swap places. Now, suppose the knowledge base (KB) is

$\textit{initial state} \wedge \textit{successor-state axioms} \wedge \textit{goal}^1$,

which asserts that the goal is true at time $T = 1$. You can check that the assignment in which

$Fly(P_1, SFO, JFK)^0$ and $Fly(P_2, JFK, SFO)^0$

are true and all other action symbols are false constitutes a model of the KB. While this is a valid model, there may be other potential models that the satisfiability algorithm could yield. However, not all of these alternative models represent satisfactory plans. For instance, consider a rather trivial plan defined by the action symbols

$Fly(P_1, SFO, JFK)^0$ and $Fly(P_1, JFK, SFO)^0$ and $Fly(P_2, JFK, SFO)^0$.

This plan is unreasonable because plane P_1 departs from SFO, making the action $Fly(P_1, JFK, SFO)^0$ impossible. To prevent the creation of plans with invalid actions, we need to incorporate precondition axioms that require the preconditions to be met for an action to occur. For instance, we need to establish that:

$Fly(P_1, JFK, SFO)^0 \Rightarrow At(P_1, JFK)^0$.

Since $At(P_1, JFK)^0$ is indicated as false in the initial state, this axiom guarantees that $Fly(P_1, JFK, SFO)^0$ is also set to false in every model. By adding these precondition axioms, there is only one model that fulfills all the axioms when the goal is to be achieved at time 1. This model involves plane P_1 flying to JFK and plane P_2 flying to SFO. It is important to note that this solution consists of two parallel actions.

Introducing a third airport, LAX, leads to unexpected outcomes, as each plane now has two possible actions per state. Running the satisfiability algorithm reveals that existing axioms permit a plane to reach two destinations simultaneously, creating unrealistic solutions. To prevent this, additional axioms, such as action exclusion axioms preventing concurrent actions, must be added. For example, we can enforce complete exclusion by including all possible axioms of the form

$$\neg(Fly(P_2, JFK, SFO)^0 \wedge Fly(P_2, JFK, LAX)^0).$$

These axioms guarantee that no two actions can be performed simultaneously, thereby removing all invalid plans. However, they also require every plan to be totally ordered, resulting in a loss of flexibility as partially ordered plans are no longer possible. Additionally, this may increase the number of time steps in the plan and thus lengthen computation time.

Exclusion axioms can sometimes appear rather blunt. Rather than stating that a plane cannot fly to two airports simultaneously, we might instead simply require that no object can be in two places at once:

$$\forall p, x, y, t \quad x \neq y \quad \Rightarrow \quad \neg(At(p, x)^t \wedge At(p, y)^t) \ .$$

2.5.2 Planning Approaches

Planning is currently a highly active research area in AI, largely because it brings together the key concepts of search and logic discussed earlier. In essence, a planning system can be seen as either searching for a solution or as constructing a logical proof that a solution exists (FIKES; NILSSON, 1971). The interplay between these two fields has led to dramatic enhancements in efficiency over the past decade, resulting in planners being widely adopted in industry. However, it remains unclear which methods are most effective for particular types of problems, and it is likely that future approaches will surpass those we have today.

Planning is primarily about managing the challenges of combinatorial explosion. In a domain with p primitive propositions, there are 2^p possible states. As the complexity of the domain grows, so can p, making things even more daunting. When objects in the domain have various properties (such as *Location* or *Colour*) and take part in relations (like *At*, *On*, or *Between*), the number of possible states increases dramatically. For example, with d objects and ternary relations, there are 2^{d^3} potential states. This makes it appear, at least in the worst case, that planning is an insurmountable task.

Despite such a gloomy outlook, the divide-and-conquer strategy can be highly effective. When a problem can be fully broken down into independent sub-problems, this method can result in exponential gains in efficiency. However, the presence of negative interactions between actions often disrupts this decomposability. Partial-order planners

address these issues by explicitly representing causal links, but every conflict that arises must be resolved by deciding the sequence of actions, leading to a rapid increase in possible configurations.

In contrast, GRAPHPLAN[2] sidesteps explicit choices during graph construction by using mutex links to indicate conflicts, rather than immediately deciding how to address them. SATPLAN[2] also manages these mutual exclusions but encodes them in general CNF form rather than through a dedicated structure. The effectiveness of this approach largely depends on the capability of the underlying SAT solver.

There are several strategies for managing combinatorial explosion. Techniques developed for controlling backtracking in constraint satisfaction problems (CSPs), such as dependency-directed backtracking, are equally applicable to planning. For instance, extracting a solution from a planning graph can be cast as a Boolean CSP, where each variable represents whether a particular action takes place at a specific time. These CSPs can be tackled with algorithms like min-conflicts (MINTON et al., 1992).

A similar approach, used in the BLACKBOX[2] system, involves translating the planning graph into a CNF formula and then using a SAT solver to extract a plan. This method generally performs better than SATPLAN likely because the planning graph has already filtered out many impossible states and actions. It also tends to outperform GRAPHPLAN, most probably because SAT-based methods such as WALKSAT (KAUTZ; SELMAN; MCALLESTER, 2004) offer more flexibility than the more rigid backtracking search used by GRAPHPLAN.

Planning frameworks like GRAPHPLAN, SATPLAN, and BLACKBOX have significantly advanced the discipline of automated planning. Not only have these planners enhanced the overall efficiency and capability of planning systems, but they have also contributed to a deeper understanding of the representational and combinatorial complexities inherent in this area. Nevertheless, these approaches are fundamentally based on propositional logic, which places certain restrictions on the variety of domains they can adequately model. For the field to progress further, it appears increasingly necessary to adopt first-order representations and algorithms, despite the ongoing utility of structures such as planning graphs for generating valuable heuristics.

In the next chapter we will provide a more comprehensive exploration of automated planning algorithms and their main approaches—ranging from the earliest and most established techniques to the specific method that forms the core focus of this paper. This progression will clarify how the field arrived at its current state and highlight the rationale behind the chosen approach for this study.

[2] GRAPHPLAN, SATPLAN and BLACKBOX will be discussed in the following section.

2.6 Planning as SAT

The Planning as Satisfiability approach aims to solve classical deterministic planning problems by converting them into instances of the (SAT) problem. This method stands out by establishing a bridge between two well-established areas of artificial intelligence: automated planning and SAT formula solving.

2.6.1 SatPlan 1992

In the seminal work of Kautz and Selman (KAUTZ; SELMAN et al., 1992) proposed the construction of a propositional formula that encodes both the initial state and the goal, as well as the state transitions allowed by actions. The proposed formulation is composed of four main components: (i) variable representing the facts that are true at each time step, (ii) the actions executed, (iii) the preconditions and effects of the actions, and (iv) the mutual exclusion (mutex) constraints. These mutex constraints help maintain the consistency of the plan by preventing, for instance, the simultaneous execution of actions.

A key parameter in this formulation is the time horizon t, which defines the maximum number of discrete time steps considered when searching for a valid plan. Each time step represents a point at which an event can take place. Thus, setting a time horizon t implies exploring whether a plan of exactly t steps, or fewer, under specific encoding strategies, can achieve the transition from the initial state to the goal.

In SAT-based planning, this horizon determines the structure of the propositional encoding: variables and clauses must be generated for all facts and actions at each level from time 0 to time t. If no satisfying assignment is found, the horizon is incremented, and a new SAT instance is generated for the extended time window. In this way, the planning problem is effectively reduced to a (SAT) problem.

The algorithm operates iteratively, incrementing the time horizon and generating a new SAT formula at each step. An efficient SAT solver algorithm, such as DPLL, is employed to assess the satisfiability of each formula. If a satisfying assignment is found, it corresponds directly to a valid sequence of actions—that is, a feasible plan. If not, the process continues with an expanded horizon and a new formula, repeating until either a solution is found or it is concluded that the problem is unsolvable within the imposed constraints.

2.6.2 SatPlan 1996

The article by Kautz, McAllester, and Selman (KAUTZ et al., 1996) deepens the Planning as Satisfiability approach by introducing, analyzing, and comparing various

propositional encodings for STRIPS-style planning problems. The main focus of the work lies in the efficiency and conciseness of reductions from planning instances to propositional logic formulas, aiming for improved SAT solver performance.

2.6.2.1 Linear Encoding

Linear encoding is based on the temporal indexing of fluents and actions. Each action is represented by a proposition indicating its execution at a specific time step, while fluents are indexed per state layer. In this setting, fluents are formalised as state variables whose values may change over time as a consequence of action execution.

Formally, let $F = \{f_1, \ldots, f_n\}$ be a finite set of fluents, where each fluent

$$f_i : S \to \{\mathsf{true}, \mathsf{false}\}$$

is a Boolean-valued function over the set of world states S. A state $s \in S$ is completely characterised by the subset of fluents that hold in s, that is, $s \subseteq F$. This view of fluents originates in logical formalisations of action and change, where fluents represent properties of the world whose truth values depend on the current situation or state (MCCARTHY; HAYES, 1981; FIKES; NILSSON, 1971).

Actions are defined by their preconditions and effects over fluents. Given an action a, with preconditions $\mathrm{Pre}(a) \subseteq F$, positive effects $\mathrm{Add}(a) \subseteq F$, and negative effects $\mathrm{Del}(a) \subseteq F$, the application of a in a state s produces a successor state

$$s' = (s \setminus \mathrm{Del}(a)) \cup \mathrm{Add}(a),$$

provided that $\mathrm{Pre}(a) \subseteq s$. Hence, fluents are precisely the components of the state whose truth values may be modified by actions (GHALLAB; NAU; TRAVERSO, 2004).

Under a linear or temporal encoding, fluents are further interpreted as functions of discrete time,

$$f_i : \mathbb{N} \to \{\mathsf{true}, \mathsf{false}\},$$

where $f_i(t)$ denotes the truth value of fluent f_i at time step t. This temporal interpretation underlies propositional and constraint-based encodings of planning, in which each pair (f_i, t) corresponds to a distinct variable representing the value of a fluent at a given state layer (BLUM; FURST, 1997; KAUTZ; SELMAN, 1996).

The resulting formula includes clauses that (i) Ensure the correspondence between actions and their effects; (ii) Implement classical frame axioms, preserving unchanged fluents; (iii) Enforce the exclusivity of actions at each time step.

To illustrate, consider a simple scenario with a planning horizon $T = 1$ and the action `MOVE(A,B)`, which moves block `A` onto block `B`. We introduce variables such as

`MOVE(A,B,0)` for the action at time 0, and `ON(A,B,1)` to indicate the resulting state at time 1.

The encoding includes clauses that:

- assert the initial state, e.g., `CLEAR(B,0)` and `ON(A,Table,0)`;
- enforce preconditions, such as `CLEAR(B,0)` and `ON(A,Table,0)` for `MOVE(A,B,0)`;
- define effects, like `MOVE(A,B,0)` implying `ON(A,B,1)` and ¬`ON(A,Table,1)`;
- preserve unaffected fluents via frame axioms;
- restrict the plan to one action per time step.

However, the number of variables and clauses grows rapidly with the domain size and operator arity. To mitigate this growth, the authors propose two extensions: operator splitting and explanatory frame axioms.

2.6.2.2 Operator Splitting e Explanatory Frame Axioms

Operator splitting decomposes actions of arity k into k unary sub-actions, reducing the combinatorial complexity of arguments. Instead of encoding `MOVE(A,B,C,t)` directly, we split it into unary components `SRC(A,t)`, `OBJ(B,t)`, `DST(C,t)`. Then, the action `MOVE(A,B,C)` at time t is represented by:

$$\texttt{SRC(A,t)} \land \texttt{OBJ(B,t)} \land \texttt{DST(C,t)}$$

Explanatory frame axioms, on the other hand, state that changes in fluents occur only if caused by some explicit action, thereby reducing the number of clauses compared to classical frame axioms. Suppose at time t, `CLEAR(B)` is true, but at time $t+1$, it is false. The explanatory frame axiom ensures that this change must be caused by some action that deletes `CLEAR(B)` at time t.

$$\neg\texttt{CLEAR(B,}t{+}1) \land \texttt{CLEAR(B,}t) \rightarrow \bigvee_{a\in\mathcal{A}_{\text{del}}} \texttt{a}(t)$$

Where $\mathcal{A}_{\text{del}} = \{a \mid \texttt{CLEAR(B)} \in \text{Del}(a)\}$ is the set of actions that delete `CLEAR(B)`.

If `CLEAR(B)` becomes false, then at least one action that deletes it must have occurred at that time. Otherwise, the fluent must persist true by default. If no such action occurs at time t, then the change is not explained, and thus:

$$\neg\left(\bigvee_{a\in\mathcal{A}_{\text{del}}} \texttt{a}(t)\right) \rightarrow \texttt{CLEAR(B,}t{+}1)$$

2.6.2.3 Parallel and Planning Graph-Based Encodings

The propose is a planning graph-based encoding that allows for the parallel execution of actions. In this representation, conflicting actions are made mutually exclusive through additional clauses, and each fluent is supported by actions that maintain or change it.

Let us consider two actions occurring at time step t, `MOVE(A, Table, B, t)`, Move block `A` from the table onto block `B` and `MOVE(C, Table, D, t)`, Move block `C` from the table onto block `D`. Assume that `A`, `B`, `C`, and `D` are all distinct blocks, and that these actions do not interfere (e.g., they access disjoint resources). The SAT encoding introduces the following types of clauses:

- **Action Preconditions:** An action implies its preconditions at the previous layer:

$$\texttt{MOVE(A,Table,B,t)} \rightarrow \texttt{CLEAR(B,t)} \land \texttt{ON(A,Table,t)} \land \texttt{CLEAR(A,t)}$$

- **Action Effects:** The fluent is true at time $t + 1$ if it was added by an action or maintained:

$$\texttt{ON(A,B,t+1)} \rightarrow \texttt{MOVE(A,Table,B,t)} \lor \texttt{MAINTAIN(ON(A,B),t)}$$

- **Mutex Constraints:** Actions that conflict are made mutually exclusive. For instance, if `MOVE(A,Table,B,t)` deletes a fluent required by `MOVE(A,B,C,t)`, they cannot co-occur:

$$\texttt{MOVE(A,Table,B,t)} \rightarrow \neg\texttt{MOVE(A,B,C,t)}$$

- **Persistence (No-Op):** For fluents that remain unchanged, a `MAINTAIN` action is used:

$$\texttt{MAINTAIN(ON(A,B),t)} \rightarrow \texttt{ON(A,B,t)} \land \texttt{ON(A,B,t+1)}$$

In this encoding, planning proceeds through layers (levels), alternating between fluent layers and action layers. Each level encodes a moment in time, and the SAT solver is responsible for selecting a consistent set of actions that achieve the goal fluents at the final layer, obeying mutex constraints and causal support.

The advantage of this encoding lies in the potential reduction of the temporal depth of the SAT instance, since multiple independent actions can be applied simultaneously. However, in the worst case, this encoding is not necessarily more compact than the linear one with explanatory axioms.

2.6.2.4 Lifted Causal Encodings

The encoding is based on lifted causal plans, derived from the approach by McAllester and Rosenblitt (MCALLESTER; ROSENBLATT, 1991). This encoding avoids the full enumeration of operators in the domain by handling actions symbolically (lifted), using variables that are instantiated during the search for a solution.

Each step in the plan is represented by a parameterized copy of an operator. The encoding introduces causal links that connect a producer action to a consumer of a fluent, along with ordering constraints to ensure the validity of those links in the presence of actions that could threaten them (known as deleting threats).

To illustrate consider a simple planning problem in the *Blocks World* domain. The goal is to achieve the fluent `ON(A,B)` starting from an initial state where both blocks A and B are on the table, i.e., `ON(A,Table)` and `ON(B,Table)`. Suppose the plan involves the following symbolic steps:

- s_1: a step executing `MOVE(A, Table, B)`, which *adds* the fluent `ON(A,B)`.
- s_2: a later step (e.g., the goal step) that *requires* `ON(A,B)` as a precondition.

A *causal link* is established from s_1 to s_2:

$$s_1 \xrightarrow{\texttt{ON(A,B)}} s_2$$

- s_3: a potential threat such as `MOVE(C, B, Table)`, which *deletes* `ON(A,B)` if $C = A$.

Since s_3 could invalidate the effect of s_1 before s_2 is able to use it, it represents a *threat* to the causal link. To preserve the correctness of the plan, one of the following ordering constraints must be enforced:

$$s_3 \prec s_1 \quad \text{or} \quad s_2 \prec s_3$$

These constraints guarantee that s_3 cannot intervene between s_1 and s_2, thereby protecting the causal link.

This approach has a higher asymptotic complexity compared to earlier formulations: the number of variables is $\mathcal{O}(n^2)$ and the number of literals is $\mathcal{O}(n^3)$, where n is proportional to the number of plan steps. Moreover, there is no explicit need for frame axioms, as the persistence of fluents is inherently ensured by the structure of the causal links.

2.6.3 SatPlan 1999 - BlackBox

The evolution of the Planning as Satisfiability approach, initiated by Kautz and Selman (1992, 1996), culminated in the development of the Blackbox (KAUTZ; SELMAN, 1999). This system proposed a unification between propositional encodings of planning problems and the plan graph structure introduced by Blum and Furst (BLUM; FURST, 1997). The goal was to combine the structural efficiency of Graphplan with the solving power of modern SAT solvers.

2.6.3.1 Blackbox Architecture

Blackbox operates through five main phases:

1. It converts the STRIPS problem description into a plan graph with a given horizon T.
2. During graph expansion, it computes mutexes (mutual exclusions) using dedicated algorithms.
3. It translates the graph into a CNF formula, encoding mutual exclusions as binary negative clauses.
4. It applies general propositional simplification algorithms (e.g., the failed literal rule).
5. It solves the resulting formula using SAT solvers. If no solution is found, the time horizon T is incremented and the process restarts.

This iterative cycle enables the incremental application of propositional encodings while retaining the structural compactness of Graphplan. Compared to direct STRIPS to CNF translation, the plan graph leads to fewer variables. Meanwhile, SAT solvers, both systematic and stochastic, can benefit from this compactness and from pruning the search space using mutex constraints.

2.6.3.2 Improvements over SATPLAN96

The CNF encoding resulting from the plan graph is significantly smaller than those produced by direct STRIPS translations. By including only essential mutexes (e.g., conflicts between preconditions and effects), the system drastically reduces the number of clauses, especially in sequential domains such as blocks world.

Blackbox applies propositional inference techniques to simplify the CNF before solving, including:

- **Unit Propagation** – inference based on unit clauses, example 1.

- **Failed Literal Rule** – testing the implications of assigning a literal and checking for inconsistency, example 2.
- **Binary Failed Literal Rule** – inference over literal pairs, example 3.

Example 1 (Unit Propagation)**.** *Consider the CNF formula:*

$$\Phi = (A) \wedge (\neg A \vee B) \wedge (\neg B \vee C) \wedge (\neg C)$$

Applying unit propagation step by step:

1. (A) *implies* $A =$ *True;*
2. *From* $(\neg A \vee B) \Rightarrow B =$ *True;*
3. *From* $(\neg B \vee C) \Rightarrow C =$ *True;*
4. *Contradicts* $(\neg C) \Rightarrow$ ***conflict***.

Conclusion: *The formula is unsatisfiable under the assignment* $A = True$.

Example 2 (Failed Literal Rule)**.** *Let the formula be:*

$$\Phi = (\neg A \vee B) \wedge (\neg B \vee C) \wedge (\neg C)$$

To apply the failed literal rule, we temporarily assign $A =$ *True and propagate:*

1. $A = True \Rightarrow B = True$*;*
2. $B = True \Rightarrow C = True$*;*
3. *Contradiction with* $(\neg C)$.

Result: *Since assuming* $A = True$ *leads to a conflict, we deduce that* $A = False$.

Example 3 (Binary Failed Literal Rule)**.** *Consider the formula:*

$$\Phi = (A \vee B) \wedge (\neg A \vee C) \wedge (\neg B \vee C) \wedge (\neg C)$$

Test the literal pair $A = True, B = True$*:*

1. *From the second and third clauses, deduce* $C =$ *True;*
2. *This contradicts* $(\neg C)$.

Result: *The pair* (A, B) *is invalid. We can safely add the following clause to exclude this combination from future consideration:*

$$\neg A \vee \neg B$$

These inference rules have proven especially effective in sequential domains, where binary techniques alone can determine up to 100% of variable assignments. In addition to logical inference, Blackbox also incorporates randomization and restart strategies. After a fixed number of backtracks, the solver resets and restarts the search with a randomized variable ordering. This approach significantly reduces average solving time and mitigates the heavy-tail behavior typical of systematic search algorithms.

2.6.4 SatPlan 2004

This new version introduces improvements in both the encoding of planning problems and the efficiency of the SAT solving process, while maintaining compatibility with domains expressed in the STRIPS subset of the PDDL language.

2.6.4.1 SatPlan04 Architecture

Like Blackbox, SATPLAN04 follows a multi-phase pipeline:

1. Constructs a Graphplan-style *plan graph* up to a depth k.
2. Translates the planning graph into a propositional formula in CNF.
3. Applies a SAT solver to attempt to satisfy the formula.
4. If the formula is unsatisfiable or a timeout occurs, k is incremented and the process restarts.
5. Otherwise, the satisfying assignment is translated into a valid plan.
6. A post-processing step removes redundant actions from the solution.

This final step is critical, as the encoding does not guarantee that all activated actions are necessary to achieve the goal. Filtering out actions improves both the quality and the readability of the resulting plan.

2.6.4.2 Encoding Styles

Based on the classes of clauses (i) actions imply their preconditions; (ii) Facts imply disjunctions of actions that could have produced them in the previous step (including

`noops`); (iii) Actions imply disjunctions of actions that establish each of their preconditions; (iv) Actions with conflicting preconditions or effects are mutually exclusive; (v) Mutexes inferred via propagation through the planning graph. SATPLAN04 implements four distinct propositional encoding styles:

- **Action-based:** based on regressive inference over action preconditions (iii), (iv) and (v).
- **Graphplan-based:** rooted in the classical Graphplan structure, encoding both direct and indirect causal relationships (i), (ii), (iv) and (v).
- **Skinny Action-based and Skinny Graphplan-based:** optimized variants that include only mutex clauses inferred through constraint propagation, exclude the necessity of (v).

The skinny encodings include only essential clauses, which reduces the total number of variables and clauses in the resulting CNF. Empirically, the skinny action-based encoding proved to be particularly effective in large-scale domains where efficient memory usage is crucial.

2.6.5 SatPlan 2006

The updated version, *SATPLAN-2006*, introduces strategic improvements aimed at balancing the expressiveness of propositional encodings with practical scalability for solving complex planning problems.

As with its predecessors, SATPLAN-2006 seeks to find plans of minimal parallel length, allowing multiple non-interfering actions to be executed simultaneously at each time step. Its primary innovation lies in the way mutexes are handled and encoded, as well as in the direct representation of fluents as Boolean variables—enhancing both memory efficiency and performance.

SATPLAN-2006 maintains the resolution structure of SATPLAN-2004 and Blackbox: it constructs a planning graph up to level k, propagates mutexes among actions and fluents during graph expansion, generates a CNF formula, and invokes an external SAT solver. If no solution is found, k is incremented and the process repeats.

2.6.5.1 Improvements over SATPLAN04

The main motivation behind the changes introduced in SATPLAN-2006 was to mitigate the exponential growth of mutex clauses observed in SATPLAN-2004. While SATPLAN-2004 avoided mutex propagation entirely during graph construction to conserve memory, SATPLAN-2006 adopts a more balanced and selective approach:

Table 1 – Evolution of SATPLAN Systems

Feature	**Blackbox**	**SATPLAN04**	**SATPLAN06**
Mutexes in plan graph	Yes (actions + fluents)	Not used	Yes (only fluents)
Propositional variables	Actions only	Actions only	Actions + Fluents
Compact encoding	Partial	Yes (skinny variants)	Yes (optimized)
Solver modularity	Partial	Yes	Yes
PDDL support	STRIPS	STRIPS + types	STRIPS + types

- **Mutex propagation is enabled**, but only a *subset of mutual exclusions* is encoded.
- **Only mutexes between fluents** are translated into binary negative clauses; action mutexes are omitted.
- **Both actions and fluents** are represented as propositional variables, unlike earlier versions that modeled only actions.

This selective encoding significantly reduces the number of clauses generated, improving scalability without compromising the ability to solve planning problems.

The differences among Blackbox, SATPLAN-2004, and SATPLAN-2006 are summarized in Table 1.

2.6.6 MADAGASCAR - Parallel Encodings of Classical Planning as Satisfiability

Rintanen, Heljanko, and Niemelä (RINTANEN; HELJANKO; NIEMELÄ, 2004) investigates different semantics for parallel plans in the context of classical planning as satisfiability. The authors revisit the traditional notion of parallelism, known as *step semantics*, and propose two alternative variations: process semantics and 1-linearization semantics. These proposals are accompanied by propositional encodings designed to reduce formula complexity and improve practical performance.

2.6.6.1 Step Semantics

The step semantics, used in works such as Kautz and Selman (1996), allows the simultaneous execution of operators provided that:

- Their preconditions are satisfied in the current state;
- Their effects do not conflict with one another;
- The execution order does not affect the final state.

This semantics is implemented through constraints that prevent interference between operators. The propositional encoding defines variables for fluents and actions at each time step, with axioms modeling action preconditions, effects, and fluent persistence.

2.6.6.2 Process Semantics

Process semantics is a variation of step semantics with an additional restriction: an action must be executed at the earliest possible time step, according to step semantics criteria. In other words, actions should not be delayed if they can be advanced without causing conflicts.

The propositional encoding introduces clauses to prevent unnecessary delays. If an action o is scheduled for execution at time $t+1$, the formula requires that there exists at least one action o_i at time t whose presence prevents o from being executed earlier:

$$o_{t+1} \rightarrow (o_1^t \vee o_2^t \vee \ldots \vee o_n^t)$$

The experiments showed that this semantics did not yield consistent improvements in solving time. The increased number of clauses introduced by the added constraints may hinder the SAT solver's performance.

2.6.6.3 1-Linearization Semantics

1-linearization semantics is less restrictive than step semantics. It allows the simultaneous execution of operators as long as there exists at least one total ordering of those actions that results in a valid sequential plan. Unlike step semantics, it does not require that all possible orderings be valid.

To ensure that concurrently applied actions are linearizable, the authors introduce **disabling graphs**. In these graphs, an edge between two actions indicates that one may invalidate the other if executed first. The presence of cycles in such graphs indicates that the concurrent execution of those actions is not feasible.

Two encoding methods are proposed:

- **Cubic encoding** ($O(n^3)$): Uses auxiliary variables to detect the absence of cycles. While more expressive, this encoding can generate large formulas in domains with many operators.
- **Fixed ordering**: Establishes an arbitrary order among actions and forbids the concurrent execution of those that would violate this order. This encoding is lighter and achieved good practical performance in experimental evaluations.

Rintanen (RINTANEN, 2004) proposes new strategies for evaluating sequences of propositional formulas in the context of SAT-based planning. The traditional strategy consists of evaluating these formulas sequentially in order to find the shortest plan length that yields a satisfiable formula.

Although the sequential strategy ensures plans with minimal length, it may result in high execution times, particularly when proving the unsatisfiability of shorter formulas is computationally expensive. Rintanen's key observation is that, in many cases, evaluating a satisfiable formula corresponding to a slightly longer plan may be significantly faster than proving the unsatisfiability of previous shorter formulas. This is due to the increasing complexity of formulas as the plan length grows and to the fact that less constrained satisfiable formulas are often easier to solve.

2.6.6.4 Algorithm S: Sequential Evaluation

The traditional method (Algorithm S) evaluates formulas in ascending order of plan length, starting from ϕ_0, and proceeds until the first satisfiable formula is found. This method guarantees the shortest possible plan, but may require the complete evaluation of several unsatisfiable formulas before reaching a solution.

2.6.6.5 Algorithm A: Parallel Evaluation with n Processes

Algorithm A distributes the evaluation of formulas among n concurrent processes. Each process evaluates a formula ϕ_i until its satisfiability is determined. When a process finishes, it is assigned the next unevaluated formula. The algorithm terminates as soon as one satisfiable formula is found. The algorithm A exhibits the following characteristics:

- Allows concurrent evaluation of formulas with different plan lengths.
- Avoids full evaluation of formulas that may later be solved by other processes.
- The choice of parameter n directly affects performance, requiring a balance between parallelism and resource usage.
- Algorithm A, with n processes, has a lower bound on performance improvement of $1/n$ compared to Algorithm S.

In some cases, multiple processes may end up evaluating satisfiable formulas beyond the first one. The performance gain depends on the distribution of evaluation costs across the sequence.

2.6.6.6 Algorithm B: Geometric CPU Time Distribution

Algorithm B proposes an interleaved distribution of CPU time across formulas, without requiring a fixed number of processes. Each formula ϕ_i receives a fraction of time proportional to a geometric factor γ^i, where $\gamma \in (0, 1)$. The algorithm B exhibits the following characteristics:

- Enables evaluation of many formulas in parallel with decreasing priority.
- Avoids the need to select a fixed value of n, as required in Algorithm A.
- More adaptable in scenarios where the optimal plan length is unknown.
- Algorithm B, with parameter γ, has a performance gain lower bound of $1 - \gamma$ compared to Algorithm S.

This algorithm can begin evaluating promising formulas without waiting for the full resolution of earlier ones, potentially leading to significant savings in total solving time.

2.6.7 Efficient Encoding of Cost Optimal Delete-Free Planning as SAT

The work of Rankooh and Rintanen (RANKOOH; RINTANEN, 2022) proposes a novel encoding for solving delete-free planning problems with minimum cost using propositional satisfiability. The core idea is to represent relaxed plans through partial causal functions that associate propositions with actions. This encoding was implemented in the *Madagascar* planner and compared with linear and integer programming approaches.

Delete-free problems are those in which actions only add propositions to the state without removing any. Every STRIPS problem can be relaxed into a delete-free problem, defining the h^+ heuristic, widely used as an admissible estimate of optimal cost.

Example 4. *Consider a planning problem with the following elements:*

- *Propositions:* $P = \{p, q\}$*;*
- *Actions:*
  - a_1*: preconditions* $\emptyset$*, effects* $\{p\}$*, cost* 1*;*
  - a_2*: preconditions* $\{p\}$*, effects* $\{q\}$*, cost* 2*.*
- *Initial state:* $I = \emptyset$*;*
- *Goal:* $G = \{q\}$*.*

In this problem, the only way to achieve goal q is by first making p true via a_1, followed by executing a_2.

- *The corresponding plan is $[a_1, a_2]$;*
- *The total cost of the plan is $1 + 2 = 3$.*

2.6.7.1 Causal Representations of Relaxed Plans

The main theoretical structure used is the partial causal function $f : P \setminus I \to A$, which maps propositions to actions that add them. This function must satisfy:

- Every proposition in the plan must be added by some action;
- The preconditions of each action must be either in the initial state or caused by other actions;
- The derived causal graph G_f must be acyclic.

Based on this formalism, minimizing the total cost of actions in the range of f is equivalent to computing h^+. The relaxed plan can be extracted via a topological ordering of the acyclic graph G_f.

Returning to Example 4, we have $f(p) = a_1$ and $f(q) = a_2$. Action a_1 adds p, which satisfies the precondition of a_2, which in turn adds q. The causal graph G_f contains two actions and an edge from a_1 to a_2. This graph is acyclic, and the resulting plan is $[a_1, a_2]$.

2.6.7.2 SAT Encoding

The SAT encoding introduces propositional variables to represent:

- Whether a proposition p has an associated action in f;
- Whether an action a adds p and $f(p) = a$.

The formulas encode the constraints for f to be a valid partial function, maintaining dependency relations between propositions and ensuring goal coverage.

For proposition q, the encoding sets $f(q) = a_2$; hence, action a_2 must be present, and p (a precondition of a_2) must be available.

This dependency is translated into SAT clauses that ensure a valid causal chain up to the goal proposition q.

To ensure that the causal graph G_f is acyclic, the authors examine two methods:

- **Transitive closure:** encodes paths and directly detects cycles.
- **Vertex elimination:** uses a node ordering to simulate vertex removal and prevent cycles, based on directed elimination width.

If the causal graph contains actions $a_1 \to a_2 \to a_3 \to a_1$, a cycle is present, rendering f invalid. The vertex elimination method tries to define an order $a_1 \prec a_2 \prec a_3$ and ensures that no edge violates this ordering.

2.6.7.3 Cost Propagation and Minimization

To determine the precise value of the h^+ heuristic, i.e., the minimum cost of a plan devoid of deletions, a SAT encoding approach is employed, incorporating constraints on the overall cost of causal actions. The search proceeds incrementally, using a parameter u as an upper limit on the total cost. The SAT formulation is satisfiable only if there exists a causal function f such that the sum of the costs associated with its domain actions does not exceed u.

The goal is to find the smallest value of u such that the corresponding SAT formula is satisfiable. This equates to identifying an acyclic subset of actions sufficient to achieve all goals from the initial state with minimum cost.

Propagation via Propositions (P-SAT)

In this approach, each proposition p receives a variable representing the minimum cost required to make it true, based on the causal function f.

The propagation rule is:

$$\text{cost}(p) = \min_{a \in \text{add}(p)} \left[\text{cost}(a) + \max_{q \in \text{pre}(a)} \text{cost}(q) \right]$$

This rule indicates that the cost to reach p is computed by considering all actions that produce p and summing the cost of each with the maximum cost of their preconditions. Thus, the cost propagation proceeds as:

$$\text{cost}(p) = 1$$

$$\text{cost}(q) = 2 + \text{cost}(p) = 3$$

This propagation is implemented as a set of SAT clauses enforcing the cost variable of q to reflect the accumulated dependencies.

Propagation via Actions (A-SAT)

Here, control is applied directly to the actions included in the domain of f. Each time an action is selected, its cost is added to a global total cost variable.

The upper bound u is enforced by adding a clause that blocks assignments exceeding it:

$$\sum_{a \in A} \text{cost}(a) \cdot \mathbf{1}_{a \in \text{dom}(f)} \leq u$$

The SAT encoding uses binary variables for each action and a counter system for summing costs, similar to pseudo-Boolean constraints encoded in CNF.

Combining Strategies

The AP-SAT version combines both propagation methods, the local estimates from proposition-based propagation and the direct control from action-based propagation. This hybrid approach improves robustness across domains, as the effectiveness of each method depends on the problem's structure and the relative number of propositions and actions.

Minimum Cost Search

The value of u is determined using binary search:

1. Start with interval $[u_{\min}, u_{\max}]$;
2. At each iteration, a new formula is built with $u = \lfloor (u_{\min} + u_{\max})/2 \rfloor$;
3. If the formula is satisfiable, try a lower value; otherwise, increase the limit;
4. The process ends when the smallest satisfiable u is found.

In the earlier example, the optimal cost is known to be 3. If we start with $u_{\max} = 5$ and $u_{\min} = 0$, the first test with $u = 2$ is unsatisfiable, forcing an increase. When $u = 3$, the formula is satisfiable, and the process terminates with the optimal value.

3 An Extension of Madagascar Planner

This chapter describes the architectural and algorithmic modifications applied to the Madagascar planner to support our horizon-search strategies. We begin by detailing the extensions to the command-line interface, followed by modifications to the SAT solver interaction layer, and finally the design and implementation of the new sequential, heuristic, and parallel search strategies.

3.1 Architecture of Madagascar

The Madagascar planner is fully implemented in *C*, and its architecture is organized into five main modules: (i) Input and Parsing, (ii) Preprocessing and Grounding, (iii) Translation to SAT and Search Strategy, (iv) Internal SAT Solver, and (v) Data Structures and Utilities. Each module plays an essential role in the processing pipeline.

3.1.1 Input and Parsing Module

This module is responsible for reading the domain and problem files in PDDL and converting them into intermediate structures manipulable by the system. The syntactic analysis is implemented through the files `parser.y` (Yacc/Bison) (AHO et al., 2006b) and `lexer.lex` (Lex/Flex) (AHO et al., 2006a), which define, respectively, the grammar and lexical analyzer used to interpret the input. After this stage, the files `asyntax.c` and `asyntax.h` define the Abstract Syntax Tree (AST), which represents the planning problem.

3.1.2 Preprocessing and Grounding Module

Before the SAT translation, the planner performs the *grounding* process, in which parameterized predicates are instantiated as atomic propositions. Additionally, the module executes invariant detection, which is fundamental for reducing the search space and producing more compact SAT encodings.

The main components of this module include:

- `ground.c` / `ground.h`: Implement the grounding process by instantiating actions and predicates present in the problem.
- `invariants.c` / `invariants.h`: Detect domain invariants, i.e., properties that remain constant or vary in restricted ways throughout execution, contributing to important structural optimizations.

- `scc.c` / `scc.h`: Implement algorithms for identifying Strongly Connected Components (SCCs) (TARJAN, 1972), used for dependency analysis among variables during preprocessing.

3.1.3 Translation and Search Strategy Module

This module constitutes the conceptual core of Madagascar, as it is responsible for converting the instantiated planning problem into a Boolean formula in CNF, as well as defining the search strategy with respect to the temporal horizon.

The file `translate2sat.c` and header `translate2sat.h` implements the central encoding logic, mapping states, actions, and transitions into Boolean variables and CNF clauses. This module also defines incremental search algorithms over the time horizon, such as `runalgorithmA` and `runalgorithmB`.

Complementarily, the file `dimacs.h` provides macros and auxiliary structures for generating and manipulating instances in the DIMACS format, the standard input format for SAT solvers.

3.1.4 SAT Solver Module

Madagascar integrates its own SAT solver, avoiding communication overhead with external solvers and allowing it to operate directly over the clause database produced by the translation process. This solver follows the Conflict-Driven Clause Learning (CDCL) paradigm (MARQUES-SILVA; SAKALLAH, 2002; BIERE; HEULE; MAAREN, 2009) and incorporates heuristic techniques tailored to planning problems.

The files that compose this module include:

- `clausedb.c` / `clausedb.h`: Manage the clause database, including insertion, removal, and cleaning of learned clauses.
- `clausesets.c` / `clausesets.h`: Implement abstractions for manipulating clause sets.
- `learn2.c`: Implements clause-learning mechanisms used by the CDCL algorithm.
- `heuristics2.c`: Defines the solver's decision heuristics, including variants of VSIDS (MOSKEWICZ et al., 2001) and planning-specific heuristics employed in the Mp and MpC planners, both Mp and MpC are two different variants of these planning-specific heuristics, both originating from the Madagascar Planner.
- `interface.h`: Provides the interface between the planning module and the internal SAT solver.

3.1.5 Data Structures and Utilities

The operation of the main modules is facilitated by a set of mechanisms and tools specifically designed to address the requirements of the planner. Among the most significant are:

- `tables.c` / `tables.h`: Maintain symbol tables and mappings used throughout the translation process.
- `intsets.c` / `intsets.h` and `ordintsets.c` / `ordintsets.h`: Implement integer sets, widely used to represent literals, clauses, and auxiliary sets.
- `main.c` / `main.h`: Define the system entry point, including command-line argument processing (such as options `-F`, `-T`, and `-G`) and the invocation of routines from `translate2sat.c`.
- `printplan.c` / `printplan.h`: Responsible for formatting and displaying the plan returned by the SAT solver when a solution is found.

3.2 Implementation of the Search Strategies

Before introducing the proposed strategies, it is useful to understand the motivation for applying a binary search over horizon lengths. Madagascar exposes several runtime parameters that allow users to guide the exploration:

- `-S` (step size for horizon lengths, default `-S 5`),
- `-P` (search algorithm, default E-steps `-P 2`),
- `-F` (initial horizon length, default `-F 0`),
- `-T` (final horizon length, default `-T 3000`), and
- `-M` (use a maximum of n processes, default `-M 20`).

For instance, running the sequential algorithm (`-P 0`) with step size `-S 1` and a single worker (`-M 1`) ensures the discovery of optimal plans in sequential domains. However, this strategy necessitates a linear $\mathcal{O}(n)$ number of calls to the SAT solver, corresponding to exploring each step in the horizon range sequentially.

From this observation arises the proposal to steer the horizon exploration using the `-F` and `-T` flags. In this setting, we may begin with an arbitrary intermediate horizon length h. If this horizon is shown to be SAT, we know that smaller horizons may also be

SAT. Thus, the search horizon may be reduced by half ($h/2$), following the principle of binary search (SEDGEWICK; WAYNE, 2011).

Conversely, if the horizon is found to be UNSAT, we may conclude that all shorter horizons are UNSAT as well. In this case, the search must therefore be directed towards potentially satisfiable horizons above the current value. Accordingly, if a satisfying horizon has not yet been discovered, the search limit is doubled ($2h$). However, if a satisfying horizon has already been found (thus establishing an upper bound), the next candidate horizon is set to the midpoint between the current UNSAT value and the lowest known SAT value.

The complete procedure is summarised in Listing 3.1, which presents the *pseudo-code* for the binary search applied over the horizon lengths using the `-F` and `-T` flags.

Listing 3.1 – satiator.sh pseudo-code algorithm

```
1  h=50
2  lastsat=0
3  lastunsat=0
4  while true; do
5    (( lastunsat+1 == lastsat )) && break
6    ./madagascar -P 0 -F $h -T $h -Q -o $PLAN $DOMAIN $PROBLEM > /dev/null
7    if [[ -f $PLAN ]]; then
8      lastsat=$h
9      rm $PLAN
10     h=$(((h+lastunsat)/2))
11     echo SAT = $lastsat
12   else
13     lastunsat=$h
14     (( lastsat == 0 )) && h=$((2*h))
15     (( lastsat != 0 )) && h=$(((h+lastsat)/2))
16     echo UNSAT = $lastunsat
17   fi
18   echo "h=$h; ls=$lastsat; lu=$lastunsat"
19 done
20 echo OUT $lastsat
```

The Listing 3.1 thus served as a conceptual starting point. It allowed us to empirically observe the gains achieved by replacing the default linear exploration of horizons used in the standard execution modes of Madagascar, with a guided strategy capable of quickly identifying the smallest satisfiable horizon. However, being an external solution, the satiator inherently suffers from structural limitations, such as the lack of internal instance reuse, the absence of memory lifecycle integration, and the impossibility of employing parallel or heuristic-driven search techniques.

These limitations motivated the development of the techniques presented in the following subsections. Unlike the satiator, the algorithms described next were fully integrated into Madagascar. These changes were implemented primarily in two source files: `main.c`, responsible for the user interface and configuration, and `translate2sat.c`, which concentrates the search logic and SAT encoding.

3.2.1 Command-Line Interface

To enable the selection and parametrization of the new search strategies, the `main` function, located in `main.c`, was extended. The command line option `-G` was added to support multiple operational modes, also allowing the passing of additional arguments.

A new parsing mechanism for the call `-G <mode> [k]` was implemented, where:

- **`mode` (0–7)**: selects the specific binary or parallel search strategy;
- **`k`**: defines the level of parallelism (number of *workers*), required in parallel modes (2 to 5).

Table 2 presents a summary of the new execution modes integrated into the planner:

Table 2 – Search Algorithm Modes Integrated into Madagascar

Mode (-G)	Internal Algorithm	Description
0	`runalgorithmBB`	Standard binary search (sequential).
1	`runalgorithmBBD`	Binary search with dynamic probing.
2	`runalgorithmBBA`	Asynchronous k-ary search using *round-robin* scheduling. Requires parameter `k`.
3	`runalgorithmBBB`	Prioritized asynchronous k-ary search. Requires parameter `k`.
4	`runalgorithmBBDA`	Hybrid strategy combining dynamic probing and k-ary *round-robin* search. Requires parameter `k`.
5	`runalgorithmBBDB`	Hybrid strategy combining dynamic probing and prioritized k-ary search. Requires parameter `k`.
6	`runalgorithmExp`	Exponential horizon expansion followed by standard binary search.
7	`runalgorithmBBD_LR`	Dynamic probing combined with reverse linear search.

In the header file `main.h`, the following global variable was added:

```
extern int paramBB;
```

This variable stores the value of `k`, making it accessible to the search module. Furthermore, the variables `planSemantics` were adjusted to ensure that the new algorithms preserve sequential semantics, preventing multiple actions within the same horizon, and `outputTimeStep`, whose default value was updated to 20 in order to accelerate the identification of satisfiable horizons in strategies based on dynamic probing.

3.2.2 Adapting the Solver Interface: The `computeonestepBB` Function

The original `computeonestep` function in Madagascar was designed under the assumption of *Linear Search*. In this model, horizons are evaluated sequentially as T, $T + K$, $T + 2K, \ldots$, where K is a predefined constant. Thus, whenever a horizon T is determined to be UNSAT, its instance becomes irrelevant and must be immediately freed to allow construction of the next horizon.

However, this assumption is incompatible with Binary Search strategies, in which the evaluation order is non-monotonic. In this scenario, the algorithm may test horizons such as $T = 50$, then $T = 25$, and subsequently $T = 37$, breaking the sequential discard logic.

To support these new requirements, the `computeonestepBB` function was developed as a control-oriented wrapper over the underlying CDCL engine. The first modification involved revising its return semantics. In the original version, the function followed a binary scheme in which a return value of 1 indicated a SAT solution and 0 indicated either UNSAT or an incomplete search.

In the new design of `computeonestepBB`, this behavior was extended with the introduction of a third return state. The function now distinguishes explicitly between UNSAT, SAT, and cases where the solver reaches a timeout or conflict limit, the latter being reported through a dedicated “incomplete” state.

Another structural modification was the adoption of an inversion of control model for memory management. In the previous approach, the original function automatically freed unsatisfiable instances through calls to `freeinstance`. This coupled resolution logic to memory disposal.

In contrast, binary search cannot assume that previously processed horizons are necessarily smaller, making such behavior unsafe. Premature disposal could destroy instances still needed later, generating inconsistencies in the algorithm.

Thus, the new `computeonestepBB` function becomes purely computational: it only reports the solver state and delegates memory management to higher-level algorithms (`runalgorithmBBDA`, `runalgorithmBBDB`, etc.), which possess global knowledge of the search interval [*low*, *high*].

By removing all implicit assumptions about the order in which horizons are processed, the function now behaves as a stateless and side-effect-free component, ensuring that no instance is prematurely destroyed or altered. This redesign not only avoids inconsistencies when navigating the search space in a non-monotonic fashion but also provides a more flexible and extensible interface for integrating additional search strategies.

Listing 3.2 and Listing 3.3 presents a simplified version of the implementation of

`computeonestep` and `conmputeonestepBB`, respectively.

Listing 3.2 – Structure of (`computeonestep`).

```
1  // Original Logic (Coupled and Synchronous)
2  int computeonestep(int i) {
3      solve = solver_engine(i);
4      if (solve == SAT) {
5          return 1;
6      }
7
8      if (solve == UNSAT) {
9          for (j = 0; j < i; j++) {
10             if (seqs[j].sati->value == -1) { /* Formulas that must be UNSAT
                   */
11                 freeinstance(seqs[j].sati);
12             }
13         }
14         freeinstance(seqs[i].sati);
15     }
16     return 0;
17 }
```

Listing 3.3 – Structure of (`computeonestepBB`).

```
1  // Proposed Logic (Decoupled and State-Oriented)
2  int computeonestepBB(int i) {
3      solve = solver_engine(i);
4
5      // Only reports the state. Does not modify the environment.
6      if (solve == UNSAT) return 0;
7      if (solve == SAT)   return 1;
8
9      return -1; // Incomplete state: timeout or conflict limit
10 }
```

3.2.3 Implemented Search Algorithms

Unlike the original approach, which performed a linear iteration over the time horizon, the new routines manage the lifecycle of SAT instances in a non-monotonic manner. This structural change enables the exploration of navigation strategies in the search space, especially in scenarios where horizons are evaluated outside the natural sequential order.

Below, we detail the operational logic of the developed algorithms.

3.2.3.1 Sequential Binary Search (`runalgorithmBB`)

The `runalgorithmBB` function implements the classical binary search strategy (SEDGEWICK; WAYNE, 2011) over the time interval $[F, T]$, where F denotes the initial horizon (`firstTimePoint`) and T the final horizon (`lastTimePoint`). This strategy was conceived as a comparative *baseline*, providing the theoretical guarantee of identifying the optimal horizon with an asymptotic number of SAT solver calls equal to $\mathcal{O}(\log_2(T - F))$,

a substantial improvement over the linear complexity $\mathcal{O}(T - F)$ of the original sequential algorithms (A and B).

Regarding its operation, the algorithm maintains two indices, `low` and `high`, delimiting the current search interval. At each iteration, a central pivot is selected,

$$M = \left\lfloor \frac{low + high}{2} \right\rfloor,$$

for which a SAT instance is constructed and solved.

In the SAT case, the existence of a plan at M establishes this value as a valid upper bound. The algorithm registers M as the best solution (`best_solution`), updates the upper bound to $high = M - 1$, and discards the corresponding instance. In the UNSAT case, the absence of a plan at M implies that no horizon $t < M$ can contain a solution, then the lower bound is updated to $low = M + 1$, and the instance is discarded.

Listing 3.4 presents a simplified version of the implementation. Notably, the routine employs an internal `while` loop that forces synchronous completion of the solver call (`computeonestepBB`), a crucial requirement to avoid inconsistencies between different horizons.

Listing 3.4 – Structure of the Sequential Binary Search (`runalgorithmBB`).

```
1  void runalgorithmBB() {
2      int low = firstTimePoint;
3      int high = lastTimePoint;
4      int best_solution = -1;
5
6      while (low <= high) {
7          int mid = low + (high - low) / 2;
8
9          // avoid testing non-optimal horizons
10         if (best_solution != -1 && mid >= best_solution) {
11             high = mid - 1;
12             continue;
13         }
14
15         // Instantiate and solve synchronously
16         startlength(current_idx, mid);
17
18         while (seqs[current_idx].sati->value == -1) {
19             // In sequential BB, solver completion is enforced
20             computeonestepBB(current_idx, 0);
21         }
22
23         if (seqs[current_idx].sati->value == 1) { // SAT
24             best_solution = mid;
25             high = mid - 1;
26         } else { // UNSAT
27             low = mid + 1;
28         }
29     }
30 }
```

Although it exhibits excellent performance in terms of the number of solver queries, the `runalgorithmBB` routine faces practical limitations in the context of automated planning. In particular, the computational cost of demonstrating UNSAT or even SAT for large horizons (near the upper limit $T_{\max}$) can be extremely high.

This behavior motivates the development of the heuristic and parallel strategies presented in the following subsections, which aim to mitigate the impact of difficult cases through dynamic probing, parallelism, and prioritization policies.

3.2.3.2 Binary Search with Dynamic Probing (`runalgorithmBBD`)

The `runalgorithmBBD` strategy was designed to overcome a central limitation of traditional binary search. When an arbitrary midpoint is selected as the initial pivot (e.g., $M = 750$ in $[0, 1500]$), pure binary search may direct the solver toward an unnecessarily difficult instance. Since the cost of proving the satisfiability grows exponentially with the structural complexity of the SAT instance, this behavior can lead to substantial delays. To mitigate this issue, the BBD approach organizes execution into two complementary phases.

In Phase 1, the objective is not to immediately determine the optimal makespan, but rather to narrow the interval quickly and safely. To this end, a scheduling heuristic inspired by Madagascar's original Algorithm B is employed, based on accumulated effort with exponential decay.

The initial probe maintains several concurrent instances in a form of "simulated parallelism", alternating execution within a single thread. Priority is always given to the instance with the smallest horizon ($H_{\min}$). Larger horizons ($H > H_{\min}$) may progress only if their accumulated effort E_H satisfies:

$$E_H < E_{H_{\min}} \cdot r^{(H-H_{\min})} \tag{3.1}$$

where r (`paramB`) is the decay factor, with $0 < r < 1$. This mechanism ensures that distant horizons are explored gradually, without hindering progress on the most promising (smaller) instances.

During the probe, two control variables are updated dynamically:

- `dynamic_low`: the largest horizon already confirmed as UNSAT, initially F;
- `high_bound`: the first horizon confirmed as SAT, initially undefined.

A reactive *pruning* mechanism is also used: whenever the probe detects that an instance with horizon H is UNSAT, not only is `dynamic_low` updated to $H + 1$, but all still-active instances with horizon $H' < H$ are immediately aborted, as they must also be UNSAT due to the logical monotonicity of the planning problem.

Phase 1 terminates as soon as the first SAT result appears, setting the provisional upper bound `high_bound`. All remaining instances are discarded, and the algorithm proceeds to the next phase.

In Phase 2, a classical binary search is executed, but now restricted to the refined interval [`dynamic_low`, `high_bound` − 1], eliminating the possibility of prematurely testing very large (and computationally expensive) horizons.

Listing 3.5 shows the general structure of the implementation, highlighting the flow between phases and the pruning mechanism.

Listing 3.5 – Structure of the Binary Search with Dynamic Probing (`runalgorithmBBD`).

```
void runalgorithmBBD(double r) {
    // --- PHASE 1: Probe ---
    do {
        // Heuristic scheduling based on Algorithm B
        int result = computeonestepBB(i, 0);

        if (result == 1) { // SAT found
            best_solution = i;
            goto phase_2_start; // Immediate transition to phase 2
        }
        else if (result == 0) { // UNSAT found
            int unsat_H = seqs[i].horizon;

            // Update dynamic lower bound
            if (unsat_H + 1 > dynamic_low) {
                dynamic_low = unsat_H + 1;
            }

            // Pruning: remove instances below unsat_H
            prune_instances_below(unsat_H);
        }
    } while (actives > 0);

phase_2_start:
    // --- PHASE 2: Restricted Binary Search ---
    int low  = dynamic_low;
    int high = best_solution - 1;

    // Conventional binary search on the reduced interval
    run_binary_search(low, high);
}
```

The `runalgorithmBBD` strategy therefore represents a significant improvement over traditional binary search. By deliberately avoiding potentially very difficult horizons during the initial stage, the algorithm substantially reduces the risk of encountering the typical “worst case” in SAT-based planning. Nevertheless, Phase 2 remains sequential, testing one horizon at a time.

3.2.3.3 Asynchronous k-ary Binary Search via Round Robin (`runalgorithmBBA`)

The `runalgorithmBBA` strategy represents the transition to a search model that, similarly to the `runalgorithmA` and `runalgorithmB` approaches, exploits a set of concurrent workers to more effectively handle makespans of high structural complexity. Whereas the previous strategies operated essentially with a single execution “front”, BBA simultaneously manages a pool of k workers, where k is user-configurable.

The main goal of this approach is to explore multiple regions of the search space $[low, high]$ in parallel, increasing the probability of quickly discovering either satisfiable or unsatisfiable horizons to shrink the search interval and escape complex regions of the search domain.

The fundamental innovation of BBA lies in its dynamic worker-allocation policy. Instead of dividing the initial interval into k fixed subintervals, as in a static k-ary search, BBA employs an adaptive mechanism based on the current “gaps” between the horizons under evaluation.

At each control iteration, the algorithm performs the following steps:

1. Collects all active horizons in the worker pool, together with the global bounds *low* and *high*.
2. Sorts these values to form the time-ordered sequence

$$S = \{low, h_1, h_2, \ldots, h_n, high\}.$$

3. Computes the distances $d_i = h_{i+1} - h_i$ between adjacent elements.
4. Selects the largest gap and spawns a new worker at the midpoint of that interval.

This heuristic ensures that computational effort is directed toward the least explored regions of the domain, preventing excessive concentration of tests in already well-investigated areas and avoiding the need to stop or reset workers already in progress. Since the maximum number of workers is capped at 50, the cost of sorting remains minimal.

Once allocated, workers follow a *Round Robin* (TANENBAUM; BOS, 2015) scheduling policy. At each cycle, the algorithm iterates over the pool and grants each active instance a uniform processing slice, a single call to `computeonestepBB`. This policy promotes fairness among tested horizons, ensuring that a particularly difficult instance cannot monopolize computational resources preventing starvation.

The efficiency of BBA critically depends on its ability to immediately react to results produced by the workers. Since each instance operates on a different makespan, its completion may invalidate the need to continue executing other instances.

The pruning logic is as follows:

- **SAT event at H:** If a worker finds a plan at horizon H, then H becomes a valid upper bound. The algorithm updates $high = H - 1$, and all instances with horizons $h > H$ are immediately aborted, since any solution found by them would be necessarily suboptimal.

- **UNSAT event at H:** If a worker proves that horizon H is UNSAT, all smaller horizons must also be unsatisfiable. Thus, the algorithm updates $low = H + 1$, and aborts all workers with horizons $h < H$.

Listing 3.6 shows the general structure of the main control loop, including the scheduling policy and the asynchronous pruning mechanism.

Listing 3.6 – Pruning and Scheduling Logic in BBA (`runalgorithmBBA`).

```
1  void runalgorithmBBA(int k) {
2      // Initialization of the worker pool
3      // ...
4
5      while (low <= high) {
6          // --- Phase 1: Gap Filling ---
7          // (Detect gaps, sort active horizons, and spawn new workers)
8
9          // --- Phase 2: Round Robin and Pruning ---
10         for (int i = 0; i < k; i++) {
11             if (!is_active(i)) continue;
12
13             int result = computeonestepBB(i, 0);
14             int H = active_horizons[i];
15
16             if (result == 1) { // SAT
17                 if (H < best_solution) best_solution = H;
18                 high = H - 1;
19
20                 for (int j = 0; j < k; j++) {
21                     if (active_horizons[j] > high)
22                         free_worker(j);
23                 }
24             }
25             else if (result == 0) { // UNSAT
26                 if (H + 1 > low) low = H + 1;
27
28                 for (int j = 0; j < k; j++) {
29                     if (active_horizons[j] < low)
30                         free_worker(j);
31                 }
32             }
33         }
34     }
35 }
```

Although the *Round Robin* strategy guarantees fairness, it does not take into account important domain characteristics, such as the fact that proving SAT at short horizons is typically much faster than proving SAT at larger horizons. Consequently, uniform

time distribution may waste resources on long-horizon instances that would eventually be pruned if shorter instances were given priority.

Another relevant detail is that the sorting performed to detect gaps cannot effect the physical position of workers in the vector, since this would break the correspondence between each position and its associated SAT instance. Thus, the pruning process must scan the entire vector to correctly identify which workers should be released.

3.2.3.4 Asynchronous Prioritized k-ary Search (`runalgorithmBBB`)

Although the BBA algorithm guarantees fairness in the distribution of CPU time across workers, such a policy may be suboptimal in the context of SAT-based planning. In particular, the *Round Robin* scheduler may waste computational cycles by slowly advancing instances that will later be discarded due to a SAT or UNSAT discovery at another horizon.

The `runalgorithmBBB` algorithm ("B-style" Binary Search) preserves both the worker-pool architecture and the *Gap Filling* mechanism of BBA, but replaces the scheduler with a priority policy based on accumulated effort. This policy derives from Madagascar's "Algorithm B" and introduces, in a controlled manner, the possibility of deliberate starvation for less promising instances.

The core idea is to grant priority to instances whose horizon is closest to the center of the search interval, as such horizons tend to produce the strongest global pruning effects. Distant horizons receive CPU time only when the effort of the central instance has grown sufficiently to justify exploration of more remote regions.

At each control cycle, the algorithm identifies the active instance closest to the midpoint of the current horizon interval called `first_active`, whose horizon is denoted by H_{min}. This instance receives absolute priority and always executes a call to `computeonestepBB`.

For any other instance i, with horizon $H_i > H_{min}$, permission to execute depends on its accumulated computational effort (E_i) relative to a dynamic threshold defined by Equation 3.2:

$$Threshold_i = E_{min} \cdot r^{\Delta_i} + 0.5 \tag{3.2}$$

where:

- E_{min} is the accumulated effort of the `first_active` instance;
- r is the decay factor ($0 < r < 1$), configurable via parameter `paramB`;
- Δ_i is the normalized distance: $\Delta_i = (H_i - H_{min})/$`step`.

This equation induces an exponential time-allocation profile: the farther the horizon, the smaller the threshold, and therefore the lower the probability of execution. As a result, the algorithm concentrates computational power around mid-range horizons, maximizing the speed at which new global pruning conditions are discovered.

Listing 3.7 shows the structure of the main loop, highlighting the execution decision conditioned on the effort threshold:

Listing 3.7 – Prioritized Scheduling Policy (`runalgorithmBBB`).

```
void runalgorithmBBB(int k) {
    // ... (Initialization and spawning identical to BBA) ...

    // Identify the anchor (instance with the closest-to-mid horizon)
    int first_idx = find_mid_horizon_index();
    int H_first = active_horizons[first_idx];
    int effort_first = seqs[first_idx].effort;

    // Iterate over the pool, deciding who receives CPU
    for (int i = 0; i < k; i++) {
        if (!is_active(i)) continue;

        bool should_run = false;

        if (i == first_idx) {
            // The central instance always runs
            should_run = true;
        } else {
            // Compute threshold for more distant instances
            int dist = abs(active_horizons[i] - H_first) / step;
            float threshold = effort_first * power(paramB, dist) + 0.5;

            if (seqs[active_seq_indices[i]].effort < threshold) {
                should_run = true;
            }
        }

        if (should_run) {
            // Execute one step and apply pruning (identical to BBA)
            computeonestepBB(active_seq_indices[i], 0);
        } else {
            // Instance deliberately left without CPU in this round
            continue;
        }
        // ... (Asynchronous Pruning logic) ...
    }
}
```

The BBB strategy combines the parallel exploration of BBA with the pruning focus of AlgorithmB. Where BBA distributes resources uniformly, BBB allocates them according to the heuristic relevance of each horizon. Thus, the most informative horizons typically those near the center receive priority, accelerating the shrinkage of the global interval and avoiding unnecessary effort on instances likely to be pruned.

The result is an approach that preserves the benefits of asynchronous parallelism while directing computational effort more intelligently, thereby maximizing the impact of pruning on the search interval.

3.2.3.5 Hybrid Strategies (`runalgorithmBBDA` and `runalgorithmBBDB`)

The hybrid strategies were conceived from the observation that pure parallel search (BBA/BBB), although efficient, may incur a high initial cost when the search interval $[F, T]$ is very large. In such scenarios, launching multiple workers over a broad and poorly informative region tends to waste resources on horizons far from the optimal solution, whose elimination may require substantial time.

To mitigate this limitation, the functions `runalgorithmBBDA` and `runalgorithmBBDB` adopt a two-stage architecture, combining the lightweight and safe nature of a sequential probe with the throughput of parallel search.

Execution begins with Phase 1, which applies the probing logic of `runalgorithmBBD`. The aim of this stage is not to determine the optimal plan but to quickly delimit the promising region of the search space. During probing, the lower bound (`dynamic_low`) is updated whenever a horizon is proven UNSAT, while the first SAT horizon found defines the initial upper bound (`high_bound`). As soon as the first SAT result is detected, all probe instances are immediately discarded, concluding Phase 1.

The resulting interval [`dynamic_low`, `high_bound` $- 1$] is typically much smaller than the original interval, focusing the search exactly where there is evidence of a transition between impossibility and existence of a plan. With this drastically reduced space, the algorithm transitions to Phase 2, instantiating a pool of k workers that will operate exclusively on the refined region.

The distinction between the two hybrid variants lies in the scheduling policy applied during Phase 2:

- **`runalgorithmBBDA`**: Uses the *Round Robin* policy identical to BBA. It is suitable for scenarios in which the variability of difficulty among adjacent horizons inside the refined interval is low or unpredictable.
- **`runalgorithmBBDB`**: Employs the Effort-Based Prioritized policy, as in BBB. This variant is advantageous when one assumes that the central horizons of the interval tend to offer the best opportunities for pruning, thus justifying a larger allocation of computational effort.

At this point, the search becomes substantially more focused, as the algorithm no longer explores irrelevant horizons. The Listing 3.8 below provides a high-level view of how the hybrid controller orchestrates the probe and parallel phases.

Listing 3.8 – Control flow of the Hybrid Strategies.

```
void runalgorithmHybrid(int k, int mode) {
    // --- Phase 1: Probe ---
    // Execute the probing heuristic until the first SAT is found
    // ...
    // Result: Reduced interval [dynamic_low, high_bound]

    if (best_solution == -1) return; // No feasible plan found

    // --- Phase 2: K-ary Search on the Refined Interval ---
    printf("Transitioning to Phase 2 in interval [\%d, \%d]\n",
           dynamic_low, high_bound);

    // Initialize a pool of k workers for the refined interval
    spawn_workers_in_gaps(k, dynamic_low, high_bound);

    while (low <= high) {
        // Dynamic gap allocation
        // ...

        // Scheduling according to the selected policy
        if (mode == BBDA) {
            execute_round_robin_step();
        } else { // BBDB
            execute_prioritized_step();
        }

        // Asynchronous pruning (common to both variants)
        apply_pruning();
    }
}
```

The hybrid approach thus seeks to combine the best of both worlds: Phase 1 avoids waste by exploring only what is necessary to locate the SAT/UNSAT transition region, while Phase 2 applies concentrated parallelism to the refined interval, substantially accelerating convergence toward the optimal horizon.

3.2.3.6 Exponential Search (`runalgorithmExp`)

Another approach implemented to mitigate the problem of an unknown or arbitrarily defined upper bound for the planning horizon (T_{max}), often set by the user (e.g., $T = 3000$), is the Exponential Search strategy (BENTLEY; YAO, 1976). This approach rapidly identifies a promising region of the search space before applying exact refinement.

The algorithm is structured into two main phases:

1. Expansion Phase (Discovery): The search space is explored in exponential steps, testing horizons at powers of two ($H_k = 2^k$). For each tested horizon, the following rules apply:

- If the result at H_k is UNSAT, it follows that $H_{opt} > H_k$. The lower bound is updated to $low = H_k + 1$, and the search proceeds to H_{k+1}.
- If the result at H_k is SAT, then the optimal solution is guaranteed to lie within the interval $[2^{k-1} + 1, 2^k]$. The upper bound is therefore set to $high = H_k - 1$, and the expansion phase terminates.

2. Refinement Phase (Binary Search): Once the interval $[low, high]$ is determined, a classical binary search is executed to exactly locate the optimal horizon H_{opt}.

This strategy exhibits $\mathcal{O}(\log i)$ complexity, where i denotes the position of the first satisfiable horizon. It is therefore particularly well suited for domains in which solutions are expected to be short relative to the theoretical maximum bound, avoiding the cost of linear probing over excessively large horizons.

3.2.3.7 Backward Search Optimized by Action Counting (`runalgorithmBBD_LR`)

The algorithm combines the heuristic probing capability of Algorithm B with an optimized backward search, in which the cardinality of the extracted plan is used to perform decisive jumps in the optimality verification process.

In the first probing stage, the exponential effort-decay heuristic is applied exactly as in `runalgorithmBBD`, with the sole objective of finding the first satisfiable horizon as quickly as possible.

In the second search stage, once a satisfiable solution is found at horizon H_{found}, the algorithm analyzes the returned plan and extracts the total number of actions actually executed, denoted by $N_{actions}$. In strictly sequential domains, executing N actions in fewer than N time steps is impossible. Therefore, whenever $N_{actions} < H_{found}$, a frequent situation in plans containing idle steps (*noops*), an exhaustive linear verification of $H_{found} - 1, H_{found} - 2, \dots$ becomes unnecessary.

The next candidate horizon is directly defined as:

$$H_{next} = \min(H_{current} - 1, N_{actions}) \tag{3.3}$$

The outcome of this test determines the subsequent control flow:

- SAT: A new shorter plan is found. The action count $N_{actions}$ is updated, and a new backward jump is performed.
- UNSAT: The impossibility of further plan compression is formally established. The last satisfiable horizon found is then certified as the global optimum.

This strategy is particularly effective in domains where, during the probing phase, solvers tend to produce valid but sparse plans with a large number of idle steps. The action-count-based jump allows pruning large regions of the search space that would otherwise be redundantly checked by a naive linear backward search, yielding substantial reductions in the total number of solver calls.

4 Empirical Evaluation

To assess the actual impact of the techniques proposed in this work, we carried out a series of systematic experiments on domains widely used within the community. This chapter outlines the adopted experimental design, the computational environment employed, and the set of selected benchmarks, establishing the basis for the analysis of the results that will be presented subsequently.

The evaluation of the proposed changes was conducted through a sequence of benchmark experiments to validate the performance, robustness, and correctness of the modifications introduced by this work. Each newly proposed algorithm was tested on the following domains from the International Planning Competition:

- grid-round-2-strips and gripper-round-1-strips (IPC, 1998);
- blocks-strips-[un]typed and logistics-strips-typed (IPC, 2000).

All experiments were conducted on a high-performance machine equipped with two Intel Xeon E5-2680 v3 CPUs (12 cores/24 threads each) and 768 GB of RAM. Each instance was executed on a single thread, totalling 24 simultaneous instances per round. It was ensured that only instances from the same domain and using the same version of the tested algorithm were executed together.

Table 3 provides a summary of the results obtained in the experiments, including the ratio between the number of optimal plans found and the total number of processed instances, as well as the PAR-2 scores for each proposed algorithm. Each execution was limited to a maximum timeout of 900 seconds per instance and the PAR-2 score of a solver is defined as the sum of all runtimes for solved instances + 2 × timeout for unsolved instances. It is important to mention that all algorithms that require a number `k` of workers, such as `-G [2345]`, were run using two workers.

The quantitative results presented in Table 3 provide significant insights into the behavior of the proposed search strategies compared to the sequential baseline. A critical analysis of the data reveals that the performance of the new algorithms is not uniform across all scenarios but is strongly correlated with the structural characteristics of each planning domain and the internal dynamics of the SAT solver.

One of the most notable observations is the disparity in performance gains across different domains. The effectiveness of the proposed binary and hybrid search strategies is intrinsically linked to how well the domain translates into the boolean satisfiability formalism.

In the `blocks-strips` domains (both typed and untyped), which typically favor SAT-based approaches, the proposed algorithms demonstrated substantial improvements. For instance, in `blocks-strips-typed`, the sequential baseline (`-S 1 -P 0 -M 1`) solved 49 instances with a PAR-2 score of 96,511. In contrast, the Hybrid Prioritized strategy (`-G 5`) solved 57 instances, reducing the PAR-2 score to 85,000. The Backward Search strategy (`-G 7`) achieved the best overall performance in this domain, solving 58 instances with a PAR-2 of 83,644. This confirms that in favorable domains, the ability to "jump" over large portions of the horizon allows the planner to locate the optimal solution much faster than a linear increment strategy.

Conversely, in domains that are historically challenging for SAT-based planning, such as `gripper` and `logistics`, the performance of the proposed algorithms tends to converge towards the traditional sequential approach. In `logistics-strips-typed`, all internal algorithms (`-G 0` to `-G 7`) solved approximately 7 out of 84 instances, with PAR-2 scores hovering around 139,000, statistically indistinguishable from the baseline. Similarly, in `gripper-round-1`, the number of solved instances remained constant at 2/20 across all methods. In such adverse scenarios, the bottleneck lies in the difficulty of solving any individual instance within the horizon, rather than in the orchestration of the search steps. Consequently, the overhead of managing parallel workers or probing heuristic bounds yields diminishing returns.

A specific phenomenon was observed when comparing the embedded Exponential Search strategy (`-G 6`) with the external script-based approach (`satiator.sh`). Although both share the same geometric expansion logic ($H_k = 2^k$), their runtime behaviors differ due to memory management.

In the `blocks-strips-untyped` domain, the external script `satiator.sh` solved 57 instances, while the embedded version (`-G 6`) solved 56. While the difference in solved instances is marginal, it highlights a trade-off. The embedded implementation maintains a persistent clause database. While this theoretically allows for the reuse of learned clauses between iterations, in practice, it was observed that this accumulation can be detrimental in certain domains. The addition of new clauses from larger horizons rapidly increases the memory footprint and the propagation overhead of the solver. As a result, the solver becomes "heavier" with each iteration, slowing down the resolution of subsequent steps.

In contrast, the external script approach forces a complete restart of the planner process for each horizon check. This "fresh start" clears all learned clauses and heuristics. The experimental data suggests that, for domains like `blocks`, the benefit of operating with a clean, lightweight solver instance can outweigh the cost of re-learning constraints, making the external approach paradoxically more efficient in specific cases where the clause database becomes polluted with non-reusable constraints.

The experiments also corroborate the hypothesis that resource prioritization is

superior to fairness in this context, particularly when combined with dynamic probing.

In the `grid-round-2` domain, the pure parallel strategies without probing failed significantly: `-G 2` (Round Robin) and `-G 3` (Prioritized) solved 0 instances. However, when combined with the probing phase in the hybrid strategies, the difference in scheduling policies became apparent. The Hybrid Round Robin (`-G 4`) solved only 1 instance, whereas the Hybrid Prioritized (`-G 5`) solved 2 instances, matching the baseline but with a better PAR-2 score than the pure binary search (`-G 0`) which failed to solve any instance.

Furthermore, in the `blocks` domains, the Hybrid strategies (`-G 4` and `-G 5`) significantly outperformed the pure parallel strategies (`-G 2` and `-G 3`). For `blocks-strips-typed`, pure parallel methods solved 39 instances , while hybrid methods solved 57. This validates the premise that the initial probing phase is crucial to restrict the search space, and that within the refined interval, allocating CPU time to shorter horizons (via prioritization) effectively prunes the search space, cancelling the execution of more expensive, longer-horizon instances that would otherwise consume resources in a fair scheduling policy.

Table 3 – Benchmark Results for Proposed Algorithms

Domain	Algorithm	Plan-Instance Ratio	PAR-2
grid-round-2-strips	-S 1 -P 0 -M 1	2/5	5772.178
	satiator.sh	2/5	5732.104
	seq-opt-fdss-2023	**4/5**	**1817.200**
	-G 0	0/5	9000.000
	-G 1	2/5	6002.600
	-G 2	0/5	9000.000
	-G 3	0/5	9000.000
	-G 4	1/5	7204.111
	-G 5	2/5	5750.134
	-G 6	2/5	5987.446
	-G 7	1/5	7205.613
gripper-round-1-strips	-S 1 -P 0 -M 1	2/20	32448.070
	satiator.sh	2/20	32485.127
	seq-opt-fdss-2023	**7/20**	**23605.586**
	-G 0	2/20	32701.895
	-G 1	2/20	32597.754
	-G 2	2/20	32439.138
	-G 3	2/20	32460.409
	-G 4	2/20	32447.253
	-G 5	2/20	32451.064
	-G 6	2/20	32530.130
	-G 7	2/20	32488.666
blocks-strips-typed	-S 1 -P 0 -M 1	49/102	96511.270
	satiator.sh	57/102	85178.072
	seq-opt-fdss-2023	27/102	135062.568
	-G 0	29/102	131561.935
	-G 1	54/102	88959.443
	-G 2	39/102	114542.013
	-G 3	39/102	114962.467
	-G 4	57/102	85545.348
	-G 5	57/102	85000.753
	-G 6	56/102	86720.154
	-G 7	**58/102**	**83644.530**
blocks-strips-untyped	-S 1 -P 0 -M 1	49/102	96534.202
	satiator.sh	57/102	85167.863
	seq-opt-fdss-2023	30/102	129667.952
	-G 0	29/102	131546.337
	-G 1	54/102	89063.733
	-G 2	39/102	114636.459
	-G 3	39/102	115057.899
	-G 4	57/102	85709.563
	-G 5	57/102	85007.709
	-G 6	56/102	86714.424
	-G 7	**58/102**	**83691.801**
logistics-strips-typed	-S 1 -P 0 -M 1	7/84	138914.126
	satiator.sh	7/84	138842.632
	seq-opt-fdss-2023	**25/84**	**107046.951**
	-G 0	7/84	139287.072
	-G 1	6/84	140475.883
	-G 2	7/84	139106.690
	-G 3	7/84	138969.211
	-G 4	7/84	138889.545
	-G 5	7/84	138981.162
	-G 6	7/84	139346.089
	-G 7	7/84	139442.794

5 Conclusion

The present work addressed the problem of efficiency in determining the optimal temporal horizon in automated planning based on SAT. The investigation was motivated by the inherent limitations of sequential linear search strategies, which frequently impose a prohibitive computational cost by requiring proofs of unsatisfiability for all horizons below the optimum. Consequently, we presented and demonstrated a general, sound, and complete approach for obtaining optimal plans through different binary-search strategies applied to horizon lengths in SAT-based planners.

The main contribution of this research lies in the architectural and algorithmic extension of the Madagascar planner. Unlike the original approach, which assumed monotonicity in the execution flow, this work restructured the planner's orchestration core to support non-monotonic, asynchronous, and parallel search behaviour. The implementation of the `computeonestepBB` function was a key milestone in this process, introducing an inversion-of-control model for memory management and a trivalent state-returning mechanism (SAT, UNSAT, INCOMPLETE), thereby enabling the development of sophisticated search strategies without direct coupling to the CDCL inference engine.

At the algorithmic level, eight new search strategies were designed and integrated. The Dynamic-Probe Binary Search strategy (`runalgorithmBBD`) demonstrated the importance of avoiding the "worst case" of pure binary search through an initial heuristic bounding phase. Moving towards parallelism, the asynchronous strategies (`runalgorithmBBA` and `runalgorithmBBB`) explored the concept of a worker pool with dynamic gap allocation. The comparison between Round-Robin scheduling and Effort-based Prioritisation revealed that fairness in CPU allocation may be less efficient than strategic prioritisation in scenarios with asymmetric proof costs.

Furthermore, a relevant observation emerged from the comparison between the embedded Exponential Search strategy (`runalgorithmExp`) and the external script-based approach (`satiator.sh`). Although both employ geometric expansion of the horizon, we found that monolithic integration does not always translate into performance gains. The incremental addition of clauses to the solver's clause database, inherent to the embedded approach, may degrade performance due to increased CNF complexity and higher memory demand. This phenomenon was particularly evident in the *blocks world* domain, where the overhead of managing accumulated clauses offset the benefits of the search strategy, suggesting that, in certain cases, a clean solver restart (as performed by the external script) may be advantageous.

The empirical evaluation also revealed a significant performance disparity depen-

dent on the structural characteristics of each domain. In domains that are intrinsically unfavourable to SAT encodings, the proposed strategies performed similarly to the traditional sequential linear search, indicating that the bottleneck in such cases resides in the efficiency of the SAT encoding itself rather than in the orchestration of the search. Conversely, in domains well-suited to SAT encoding, the parallel and hybrid strategies developed in this work yielded substantial improvements in the number of optimal plans found, validating the effectiveness of intelligent orchestration when the underlying solver can operate efficiently.

The empirical evaluation also revealed a significant performance disparity dependent on the structural characteristics of each domain. In domains that are intrinsically unfavourable to SAT encodings, the proposed strategies performed similarly to traditional sequential linear search, indicating that the bottleneck in such cases resides in the efficiency of the SAT encoding itself rather than in the orchestration of the search. Conversely, in domains well-suited to SAT encoding, the parallel and hybrid strategies developed in this work yielded substantial improvements in the number of optimal plans found, validating the effectiveness of intelligent orchestration when the underlying solver operates efficiently.

Ultimately, the hybrid strategies (`runalgorithmBBDA` and `runalgorithmBBDB`), together with the optimised backward search (`runalgorithmBBD_LR`), synthesised the observed benefits while mitigating resource wastage. In summary, the results indicate that transitioning from linear searches to divide-and-conquer strategies, combined with asynchronous parallelism, provides robustness and accelerates convergence to the optimal plan.

5.1 Future Work

Despite the advances achieved with the proposed strategies, the complexity of SAT-based planning offers several opportunities for further investigation and extension. We suggest the following directions for future work:

- **Empirical Analysis and Parameter Tuning:** Conduct systematic systematic studies and sensitivity analyses to determine optimised configurations for the parameters k (number of workers) and r (decay factor) across different domain classes. The objective is to identify robust static values or simple configuration heuristics that maximize performance without the need to introduce the overhead and complexity of machine learning models.
- **Expansion of Parallel Models:** Investigate new parallel architectures to overcome the hardware limitations of a single machine. This includes enhancing shared

memory usage via OpenMP to optimise local thread communication, as well as implementing distributed parallelism using MPI (Message Passing Interface), allowing the worker pool to scale across multiple nodes in a computing cluster.

- **Integration with Modern SAT Solvers:** Generalise the `computeonestepBB` interface to decouple the planner from its native inference engine. Future experiments should evaluate the performance impact of coupling the proposed search orchestration with state-of-the-art SAT solvers, such as Kissat, Glucose, or CaDiCaL, verifying whether the specific decision heuristics of these solvers offer advantages in hard planning instances.

- **Support for Action Costs and Non-Sequential Metrics:** Extend the search algorithms, currently focused on *makespan* optimisation (number of temporal steps), to support domains with variable action costs. Adapting search strategies to minimize the sum of action costs, rather than just the temporal horizon, would significantly broaden the planner's applicability to resource optimisation problems and complex temporal domains.

Postscript: Methodological Notes and Lessons Learned

The Genesis and Early Inspiration Underpinning the Concept

The conception of this Final Year Project arose from a practical concern encountered during the use of automated planners. When working with Madagascar to solve more complex instances, a mismatch became evident between the sophistication of its inference engine and the simplicity of its temporal search strategy.

We observed that, in order to guarantee the optimality of a plan (the smallest possible *makespan*), the tool required a strictly sequential execution, testing time horizons incrementally ($T = 1, T = 2, \ldots, T = N$). This linear search approach, with complexity $O(N)$ in calls to the SAT solver, proved prohibitive in problems where the optimal solution lay at distant horizons. The planner wasted immense computational resources proving the unsatisfiability of dozens or even hundreds of intermediate steps, when intuitively we knew that the answer could be found much more quickly through temporal "jumps".

The central idea therefore arose from a fundamental algorithmic question: if binary search is the standard solution for finding elements in ordered spaces, could we reduce the number of calls to the planner? The question then became whether using binary search would actually be worthwhile, since although it reduces the number of solver calls, those calls may themselves be more complex.

Initially, this hypothesis was explored externally to the planner's source code. The creation of simple scripts (such as the `satiator.sh` prototype), which orchestrated successive calls to Madagascar by restarting the process for each newly tested horizon, served as a proof of concept. These preliminary tests showed that, although the overhead of restarting the planner was high, the reduction in the total number of solver calls compensated for the cost in large-scale instances.

However, it soon became clear that, in order to validate the hypothesis that binary search could redefine the performance of SAT-based planning, it would be necessary to "open the box" and integrate this logic directly into the system's core, transforming an external orchestration into a native and optimised functionality.

From Planning to Plan: A Review of the Development Process

The main entry barrier was understanding the lifecycle of SAT instances. Unlike a theoretical approach in which the problem is simply translated and solved, Madagascar operated in an incremental and destructive manner. The discovery that the system was designed to immediately discard any horizon that proved unsatisfiable (UNSAT) was a key moment. We realised that it would not be enough to write a new search algorithm on top of the existing code; it would be necessary to understand and modify the system architecture so that it could support the non-monotonicity of a binary search.

The development process can be described as a continuous exercise in reverse engineering followed by careful refactoring. The initial idea of implementing a binary search seemed, in theory, simple: one merely had to test the midpoint of the interval. In practice, however, it proved to be far more subtle.

The function `computeonestep`, in its original version, behaved as an "all or nothing" mechanism: it either found a plan or destroyed the instance to free memory. To enable the strategies proposed in this work, it was necessary to rewrite this logic, creating `computeonestepBB`. The development of this function required fine-grained control over memory management, ensuring that the solver state could be preserved ("frozen") or discarded on demand, depending on the strategic decision of the higher-level algorithm, rather than automatically.

The implementation of parallel strategies introduced a new layer of complexity. Dealing with race conditions and ensuring that the worker pool operated without corrupting the planner's global data structures was one of the most challenging stages. There were numerous debugging sessions to resolve segmentation faults caused by attempts to access instances that had been incorrectly pruned by other processes.

However, it was precisely in overcoming these errors that understanding of the system was consolidated. Seeing the `BBA` algorithm run for the first time, with multiple horizons being tested simultaneously and observing asynchronous pruning, where the discovery of a plan at $T = 50$ instantly cancelled searches at $T = 80$ and $T = 90$, marked a turning point. Development ceased to be merely about coding search algorithms and became an in-depth study of how to orchestrate computational resources efficiently in high-complexity problems.

Beyond the technical obstacles of implementation, the project went through periods marked by a scarcity of new approaches and by stagnation in preliminary results. The unpredictable nature of SAT solvers often made it difficult to diagnose whether a new search strategy was inherently flawed or simply required fine-tuning. On several occasions, implementations that seemed theoretically promising, such as early versions of parallel search, failed to deliver significant performance gains in initial tests, sometimes

performing worse than the original sequential algorithm.

In those moments, perseverance became the most important aspect of the research. It was necessary to resist the frustration of not seeing immediate improvements and to continue the investigation, refactoring the code, re-evaluating race conditions and adjusting prioritisation heuristics. The continuity of the work, even in the absence of immediate empirical validation, was what ultimately made it possible to overcome performance bottlenecks and to advance with the development of new implementations.

Critical Readings

Throughout the development of this work, a review of several sources was conducted, covering both theoretical foundations and recent advances in the area of SAT-based planning. Among the core references, the (BIERE; HEULE; MAAREN, 2009) stands out for providing a comprehensive synthesis of the main techniques and applications of logical satisfiability.

To deepen the understanding of efficient encoding strategies in planning, works such as (RANKOOH; RINTANEN, 2022) and (KAUTZ et al., 1996) were analysed, as they discuss different approaches to translating planning problems into propositional formulae.

In the context of practical applications, the works of (KAUTZ; SELMAN et al., 1992) deserve particular attention, as well as the lecture slides by Leliane Nunes de Barros (<https://www.ime.usp.br/~leliane/IAcurso2006/slides/Aula20-satplan-2006.pdf>), which present in a didactic manner the concepts and methodologies for using SAT in planning. Additionally, studies such as (KAUTZ; SELMAN; HOFFMANN, 2006) and (KAUTZ; SELMAN, 1999) provide different perspectives and extensions on the topic.

The theoretical foundation also included the study of Chapters 7 and 11 of the book by (RUSSELL; NORVIG, 2016), which address essential concepts of planning and automated reasoning. Furthermore, the book by (HASLUM et al., 2019) was used as the main reference with respect to the PDDL language.

In the context of satisfiability solving, materials detailing the structure and operation of SAT solvers were examined, including the specification of the DIMACS format (DIMACS Challenge, 1993) and advances in optimisation techniques. In this regard, the works (MOSKEWICZ et al., 2001), (ZHANG; STICKELY, 1996), and (FREDRIKSON,) are particularly noteworthy. Specific materials on *MiniSAT* were also analysed, including lecture slides available at <http://minisat.se/downloads/escar05.pdf> and the article by (EÉN; SÖRENSSON, 2003), which discusses the architecture and extensibility of this solver.

Finally, the literature review included studies addressing current challenges and strategies in satisfiability problem solving, with particular emphasis on advances in search heuristics, learning mechanisms, and optimisation techniques, such as (ZHANG, 2003) and (FICHTE; HECHER; SZEIDER, 2020). Works discussing the importance of domain modelling in automated planning were also considered, including (VALLATI et al., 2021) and (HOOKER, 2007).

These references constitute a solid basis for understanding and theoretically grounding the topic addressed in this work.

Overview of Publications and Contributions

During the course of this work, we had the opportunity to contribute to parallel projects outside this work itself. These initiatives, although not directly related to the core content of the monograph, are situated within the same thematic chapter of automated planning and have enabled a broader understanding of the field. As a result of these collaborations, we highlight the publication of two papers, which address different aspects of this research area. This section provides a comprehensive overview of important achievements and key milestones accomplished during our studies and research for the monograph. The following presents a brief description of these parallel works, emphasising their objectives and relevant contributions.

bni: A PDDL Parser, REPL and Validate Tool

One of the first works undertaken was published in *Encontro Nacional de Inteligência Artificial e Computacional* (ENIAC) 22nd edition (ENIAC; SBBD, 2025) which is part of the 35th Brazilian Conference on Intelligent Systems (BRACIS) (BRACIS; SBC, 2025), titled "*bni*: A PDDL to C compiler with integrated REPL for interactive testing" (RIBEIRO; PENHA; RIBAS, 2025). This paper presents the implementation of a modular parser and an interactive REPL PDDL, entirely developed in the C programming language. The proposed solution enables the translation of PDDL domain and problem specifications into efficient, manipulable C data structures, fostering transparency, performance, and portability. Complementing the parser, the integrated REPL facilitates incremental testing and interactive debugging of planning domains and problem instances, supporting a cycle of real-time action validation and state exploration. The system also incorporates an integrated validation functionality. This feature enables both incremental and *post hoc* verification of action sequences and complete plans, ensuring that they adhere to the defined domain and problem specifications. The validate capability of the tool allows users to detect logical inconsistencies and confirm goal achievement efficiently, even in large-scale instances. Compared to established tools such as VAL, the proposed

solution offers improved flexibility and performance for plan validation, particularly in scenarios involving extensive and complex plans. This work represents a novel contribution to the field of automated planning, bridging the gap between high-level modelling languages and low-level systems programming.

Mojified Pac-Man

The second work to be highlighted is "Mojified Pacman: A Deterministic and Fully Observable Variant for PDDL Modeling Competitions" (RIBAS et al., 2025), published in the 35th International Coference on Automated Planning and Scheduling (ICAPS) journal (ICAPS, 2025). In this work, we address a challenging planning problem inspired by the classic arcade game Pac-Man, reimagined as a deterministic and fully observable turn-based variant. The objective is to eliminate adversaries and navigate a grid environment enriched with dynamic elements such as teleportation portals, ice tiles, and collectable fruits that affect agent capabilities. The game's mechanics involve asynchronous movement, conditional interactions, and spatial reasoning, making it a complex but natural candidate for PDDL modelling. While the problem can be expressed using standard PDDL features, its intricacies reveal modelling challenges that impact plan optimality and solver performance. We propose a benchmark composed of procedurally generated maps with varying combinations of terrain features and difficulty levels, classified into three competition tracks: agile, satisficing, and optimal. This work offers a new, expressive domain for evaluating the capabilities of planning systems and raises important questions about the trade-offs between modelling precision and solving efficiency.

Experimental Study with PluSAT

The last work was an experimental study conducted using **PluSAT** (CHAVES, 2022), a simple and extensible SAT solver designed for educational purposes by then-student Felipe Borges. The objective of the project was to understand the internal functioning of SAT solvers and to evaluate how specific techniques can directly influence the performance of such systems.

The project involved the development of a new *plugin* for PluSAT, modifying three main functions of the DPLL algorithm: `PreProcessing`, `Decide`, and `BCP` (Boolean Constraint Propagation). The `resolveConflict` function was kept in its original form, using standard chronological backtracking.

The main modifications implemented were:

- **Use of watched literals (Two-Watched Literals)**: a technique used to optimize the propagation process by reducing the number of clauses that need to be inspected for each assignment;

- **Jeroslow-Wang (JW) heuristic**: a variable selection method based on the frequency and size of clauses in which literals occur, favoring more promising choices for resolution.

During the preprocessing phase, all clauses were converted to a structure using two watched literals, allowing for faster identification of unit clauses. At the same time, the JW heuristic was implemented, assigning scores to literals based on the weighted frequency of their occurrence. The `Decide` function was modified to choose the unassigned literal with the highest score. The `BCP` function was rewritten to exploit the watched literals structure, significantly optimizing the propagation process.

The evaluation was based on 8,000 random formulas from the *Uniform Random-3-SAT* class, sourced from the SATLIB repository (HOOS; STüTZLE, n.d.). These formulas were executed using the original PluSAT implementation, with a time limit of three minutes per instance. Subsequently, the 100 formulas with the longest resolution time were selected to form a new test set.

The evaluation metric adopted was **PAR-2** (*Penalized Average Runtime*), which penalizes unresolved instances within the time limit by adding twice the maximum allowed time. For this experiment, the time limit was set at 60 seconds.

Solver	Solved formulas	PAR-2
Original PluSAT	75	4198.50
Improved PluSAT	100	**5.57**
Clasp (external reference)	100	2.53

Table 4 – Performance comparison using the PAR-2 metric (60-second timeout)

The Table 4 demonstrate a significant performance improvement with the implemented changes, allowing the modified version of PluSAT to solve all 100 benchmark formulas with a considerably lower penalized average runtime. Its performance approached that of well-known solvers such as Clasp (GEBSER et al., 2007), confirming that techniques like efficient propagation and decision heuristics have a direct and measurable impact on SAT solver performance.

References

AHO, A. V. et al. Lexical analysis. In: ______. *Compilers: Principles, Techniques, and Tools*. 2. ed. Boston, MA: Pearson / Addison-Wesley, 2006. cap. 3, p. 109–190. Chapter 3. Cited on page 37.

AHO, A. V. et al. Syntax analysis. In: ______. *Compilers: Principles, Techniques, and Tools*. 2. ed. Boston, MA: Pearson / Addison-Wesley, 2006. cap. 4, p. 191–302. Chapter 4. Cited on page 37.

BAADER, F. *The description logic handbook: Theory, implementation and applications*. [S.l.]: Cambridge university press, 2003. Cited on page 14.

BENCH-CAPON, T. J. *Knowledge representation: An approach to artificial intelligence*. [S.l.]: Elsevier, 2014. v. 32. Cited on page 11.

BENTLEY, J. L.; YAO, A. C.-C. An almost optimal algorithm for unbounded searching. *Information processing letters*, SLAC National Accelerator Lab., Menlo Park, CA (United States), v. 5, n. SLAC-PUB-1679, 1976. Cited on page 52.

BIERE, A.; HEULE, M.; MAAREN, H. van. *Handbook of satisfiability*. [S.l.]: IOS press, 2009. v. 185. Cited 3 times on pages 13, 38, and 64.

BLUM, A. L.; FURST, M. L. Fast planning through planning graph analysis. *Artificial intelligence*, Elsevier, v. 90, n. 1-2, p. 281–300, 1997. Cited 2 times on pages 22 and 26.

BRACIS; SBC. *35th Brazilian Conference on Intelligent Systems*. 2025. BRACIS. Disponível em: <https://bracis.sbc.org.br/2025/>. Cited on page 65.

CHAVES, F. B. d. S. *PluSAT: um resolvedor SAT modular*. 41 p. Dissertação (Trabalho de Conclusão de Curso (Bacharelado em Engenharia de Software)) — Universidade de Brasília, Brasília, 2022. Il. Cited on page 66.

COOK, S. A. The complexity of theorem-proving procedures. In: *Logic, automata, and computational complexity: The works of Stephen A. Cook*. [S.l.: s.n.], 2023. p. 143–152. Cited on page 14.

DIMACS Challenge. *Satisfiability Suggested Format*. [S.l.], 1993. Last revision: May 8, 1993. Disponível em: <https://www.cs.ubc.ca/~ajh/courses/cpsc513/assign-cmbcmp-sat/satformat.pdf>. Cited on page 64.

EÉN, N.; SÖRENSSON, N. An extensible sat-solver. In: SPRINGER. *International conference on theory and applications of satisfiability testing*. [S.l.], 2003. p. 502–518. Cited on page 64.

ENIAC; SBBD. *22nd Encontro Nacional de Inteligência Artificial e Computacional*. 2025. ENIAC. Disponível em: <https://bracis.sbc.org.br/2025/eniac/>. Cited on page 65.

FICHTE, J. K.; HECHER, M.; SZEIDER, S. A time leap challenge for sat-solving. In: SPRINGER. *International Conference on Principles and Practice of Constraint Programming*. [S.l.], 2020. p. 267–285. Cited on page 65.

FIKES, R. E.; NILSSON, N. J. Strips: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, Elsevier, v. 2, n. 3-4, p. 189–208, 1971. Cited 2 times on pages 19 and 22.

FREDRIKSON, M. Lecture notes on optimized sat-solving techniques. Cited on page 64.

GEBSER, M. et al. clasp: A conflict-driven answer set solver. In: SPRINGER. *Logic Programming and Nonmonotonic Reasoning: 9th International Conference, LPNMR 2007, Tempe, AZ, USA, May 15-17, 2007. Proceedings 9*. [S.l.], 2007. p. 260–265. Cited on page 67.

GHALLAB, M.; NAU, D.; TRAVERSO, P. *Automated Planning: theory and practice*. [S.l.]: Elsevier, 2004. Cited on page 22.

HASLUM, P. et al. *An introduction to the planning domain definition language*. [S.l.]: Springer, 2019. v. 13. Cited 3 times on pages 9, 11, and 64.

HENKIN, L. Completeness in the theory of types. *Journal of Symbolic Logic*, v. 15, n. 2, p. 81–91, 1950. Cited on page 13.

HOOKER, J. N. Good and bad futures for constraint programming (and operations research). *Constraint Programming Letters*, v. 1, p. 21–32, 2007. Submitted May 2007; Published November 2007. Disponível em: <http://constraintprogramming.org/letters/Papers/v1/hooker.pdf>. Cited on page 65.

HOOS, H. H.; STüTZLE, T. *SATLIB: A Library for SAT Research*. n.d. Accessed: 2025-07-07. Disponível em: <https://www.cs.ubc.ca/~hoos/SATLIB/index-ubc.html>. Cited on page 67.

ICAPS. *International Conference on Automated Planning and Scheduling (ICAPS)*. 2025. ICAPS. Disponível em: <https://icaps25.icaps-conference.org>. Cited on page 66.

IPC. *The 1st International Planning Competition, 1998*. 1998. IPC. Disponível em: <https://ipc98.icaps-conference.org/>. Cited 2 times on pages 16 and 55.

IPC. *The 2nd International Planning Competition, 2000*. 2000. IPC. Disponível em: <https://ipc00.icaps-conference.org/>. Cited on page 55.

IPC. *International Planning Competition (IPC)*. 2023. IPC. Disponível em: <https://ipc2023-classical.github.io/>. Cited on page 12.

JEROSLOW, R. G.; WANG, J. Solving propositional satisfiability problems. *Annals of mathematics and Artificial Intelligence*, Springer, v. 1, n. 1, p. 167–187, 1990. Cited on page 11.

KAUTZ, H. et al. Encoding plans in propositional logic. *KR*, v. 96, p. 374–384, 1996. Cited 2 times on pages 21 and 64.

KAUTZ, H.; SELMAN, B. Pushing the envelope: Planning, propositional logic, and stochastic search. In: *Proceedings of the national conference on artificial intelligence*. [S.l.: s.n.], 1996. p. 1194–1201. Cited on page 22.

KAUTZ, H.; SELMAN, B. Unifying sat-based and graph-based planning. In: *IJCAI*. [S.l.: s.n.], 1999. v. 99, p. 318–325. Cited 2 times on pages 26 and 64.

KAUTZ, H.; SELMAN, B.; HOFFMANN, J. Satplan: Planning as satisfiability. In: *5th international planning competition*. [S.l.: s.n.], 2006. v. 20, n. 49, p. 156. Cited on page 64.

KAUTZ, H.; SELMAN, B.; MCALLESTER, D. Walksat in the 2004 sat competition. In: *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing*. [S.l.: s.n.], 2004. Cited on page 20.

KAUTZ, H. A.; SELMAN, B. et al. Planning as satisfiability. In: CITESEER. *ECAI*. [S.l.], 1992. v. 92, p. 359–363. Cited 2 times on pages 21 and 64.

MARQUES-SILVA, J. P.; SAKALLAH, K. A. Grasp: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, IEEE, v. 48, n. 5, p. 506–521, 2002. Cited on page 38.

MCALLESTER, D.; ROSENBLATT, D. Systematic nonlinear planning. 1991. Cited on page 25.

MCCARTHY, J.; HAYES, P. J. Some philosophical problems from the standpoint of artificial intelligence. In: *Readings in artificial intelligence*. [S.l.]: Elsevier, 1981. p. 431–450. Cited on page 22.

MCDERMOTT, D. et al. Pddl - the planning domain definition language. In: *Proceedings of the 4th International Conference on Artificial Intelligence Planning Systems (AIPS)*. [s.n.], 1998. Disponível em: <https://www.cs.yale.edu/homes/dvm/papers/pddl-bnf.pdf>. Cited 2 times on pages 9 and 16.

MCDERMOTT, D. M. The 1998 ai planning systems competition. *AI magazine*, v. 21, n. 2, p. 35–35, 2000. Cited 2 times on pages 11 and 16.

MINTON, S. et al. Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems. *Artificial intelligence*, Elsevier, v. 58, n. 1-3, p. 161–205, 1992. Cited on page 20.

MORETTI, P. George boole and boolean algebra. Università degli Studi Roma Tre, 2012. Cited on page 13.

MOSKEWICZ, M. W. et al. Chaff: Engineering an efficient sat solver. In: *Proceedings of the 38th annual Design Automation Conference*. [S.l.: s.n.], 2001. p. 530–535. Cited 2 times on pages 38 and 64.

OWEN, O. F. et al. *The Organon, or Logical Treatises, of Aristotle*. [S.l.]: Geo. Bell, 1899. v. 11. Cited on page 12.

RANKOOH, M. F.; RINTANEN, J. Efficient encoding of cost optimal delete-free planning as sat. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. [S.l.: s.n.], 2022. v. 36, n. 9, p. 9910–9917. Cited 2 times on pages 33 and 64.

RIBAS, B. et al. Mojified pacman: A deterministic and fully observable variant for pddl modeling competitions. In: *Proceedings of the 35th ICAPS Workshop on Knowledge Engineering for Planning and Scheduling (KEPS 2025)*. [s.n.], 2025. Disponível em: <https://icaps25.icaps-conference.org/program/workshops/keps-papers/pacman.pdf>. Cited on page 66.

RIBEIRO, B.; PENHA, I.; RIBAS, B. bni: A pddl to c compiler with integrated repl for interactive testing. In: *Proceedings of the 22nd National Meeting on Artificial and Computational Intelligence*. Porto Alegre, RS, Brasil: SBC, 2025. p. 1785–1796. ISSN 2763-9061. Disponível em: <https://sol.sbc.org.br/index.php/eniac/article/view/38853>. Cited on page 65.

RINTANEN, J. Evaluation strategies for planning as satisfiability. In: *ECAI*. [S.l.: s.n.], 2004. v. 16, p. 682. Cited on page 32.

RINTANEN, J.; HELJANKO, K.; NIEMELÄ, I. Parallel encodings of classical planning as satisfiability. In: SPRINGER. *European Workshop on Logics in Artificial Intelligence*. [S.l.], 2004. p. 307–319. Cited on page 30.

RINTANEN, J.; HELJANKO, K.; NIEMELÄ, I. Planning as satisfiability: parallel plans and algorithms for plan search. *Artificial Intelligence*, Elsevier, v. 170, n. 12-13, p. 1031–1080, 2006. Cited on page 9.

RUSSELL, S. J.; NORVIG, P. *Artificial Intelligence: A Modern Approach*. [S.l.]: pearson, 2016. Cited 3 times on pages 14, 17, and 64.

SEDGEWICK, R.; WAYNE, K. *Algorithms*. [S.l.]: Addison-wesley professional, 2011. Cited 2 times on pages 40 and 43.

TANENBAUM, A. S.; BOS, H. *Modern operating systems*. [S.l.]: Pearson Education, Inc., 2015. Cited on page 47.

TARJAN, R. Depth-first search and linear graph algorithms. *SIAM journal on computing*, SIAM, v. 1, n. 2, p. 146–160, 1972. Cited on page 38.

VALLATI, M. et al. On the importance of domain model configuration for automated planning engines. *Journal of Automated Reasoning*, Springer, v. 65, n. 6, p. 727–773, 2021. Cited on page 65.

ZHANG, H.; STICKELY, M. E. An e cient algorithm for unit propagation. *Proc. of AI-MATH*, v. 96, 1996. Cited on page 64.

ZHANG, L. *Searching for truth: techniques for satisfiability of boolean formulas*. Tese (Doutorado) — Princeton University, 2003. Cited on page 65.