Assignment of master's thesis

Title:	DPLL(MAPF): Integration of a SAT Solver and Multi-Agent Path Finding
Student:	Bc. Martin Čapek
Supervisor:	doc. RNDr. Pavel Surynek, Ph.D.
Study program:	Informatics
Branch / specialization:	Knowledge Engineering
Department:	Department of Applied Mathematics
Validity:	until the end of summer semester 2021/2022

Instructions

The task is to integrate an existing CDCL-based SAT solver and multi-agent path finding (MAPF) into a single compact solver. Contemporary decision procedures for a complex theory T often rely on the concept of DPLL(T) in which a SAT solver is integrated with a decision procedure for a conjunctive fragment of T. DPLL(T) can be used to employ a SAT solver in MAPF solving via regarding MAPF as a theory and implementing a decision procedure for its conjunctive fragment. Tasks for the student are as follows:

1. Study decision procedures for complex theories based on DPLL(T) and SAT-based algorithms for multi-agent path finding.

2. Suggest an integration of a SAT solver and MAPF via the DPLL(T) scheme. Investigate various parameters of the integration such as what constraints should be treated lazily and what constraint should be added in an eager way.

3. Implement the MAPF solver based on the DPLL(T) scheme using the suggested integration and evaluate it on relevant benchmarks.

–

[1] Robert Nieuwenhuis, Albert Oliveras, Cesare Tinelli: Solving SAT and SAT Modulo Theories: From an abstract Davis--Putnam--Logemann--Loveland procedure to DPLL(T). J. ACM 53(6): 937-977 (2006)

[2] Daniel Kroening, Ofer Strichman: Decision Procedures - An Algorithmic Point of View, Second Edition. Texts in Theoretical Computer Science. An EATCS Series, Springer 2016, ISBN 978-3-662-50496-3.

[3] Pavel Surynek: Unifying Search-based and Compilation-based Approaches to Multi-agent Path Finding through Satisfiability Modulo Theories. IJCAI 2019: 1177-1183

Electronically approved by Ing. Karel Klouda, Ph.D. on 6 January 2021 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

DPLL(MAPF): Integration of a SAT Solver and Multi-Agent Path Finding

Bc. Martin Čapek

Department of Applied Mathematics
Supervisor: doc. RNDr. Pavel Surynek, Ph.D.

May 6, 2021

Acknowledgements

I feel very grateful to have a supervisor who has guided me through my diploma thesis.

I would like to thank my family for their continuous support that has been shown during my college years.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague on May 6, 2021 . . . .

Czech Technical University in Prague
Faculty of Information Technology
© 2021 Martin Čapek. All rights reserved.
This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis

Čapek, Martin. *DPLL(MAPF): Integration of a SAT Solver and Multi-Agent Path Finding.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2021.

Abstrakt

Multiagentní hledání cest (MAPF) se často kopiluje do výrokové splnitelnosti (SAT) a je dále vyřešeno existujícím SAT řešičem.

V této práci jsme přišli s novým kompilačním schématem *DPLL(MAPF)*, který používá těsnou inegraci teorie MAPF a SAT řešiče. Vysvětlili jsme, že DPLL(MAPF) je dalším logickým krokem pro zlepšení řešičů MAPF používající líné kódování. Takovým řešičem je SMT-CBS, na který jsme se v této práci zaměřili.

Dále jsme navrhli novou stategii líného kódování - Eager chokepoints.

Impementovali jsme DPLL(MAPF) a podrobili testům. Vyšlo nám, že DPLL(MAPF) dokáže překonat SMT-CBS. Nakonec jme vyhodnotili strategii *Eager chokepoints*, která se ukázala jako nevhodná.

Klíčová slova multiagentní hledání cest (MAPF), kooperativní hledání cest, SAT, umělá inteligence, DPLL(MAPF), kompilace problému do výrokové splnitelnosti, snaživé kódování, líné kódování

Abstract

Multi-agent path finding (MAPF) is often compiled to Boolean satisfiability (SAT) and solved by existing SAT solvers.

We present in this paper a novel compilation scheme called DPLL(MAPF), which brings closer integration of SAT solver and MAPF theory. We show that DPLL(MAPF) is the next logical step in improving lazy encoding MAPF solvers. Such solver that we focus on is SMT-CBS.

We propose a new strategy of adding constrains more eagerly - Eager chokepoints.

We implemented DPLL(MAPF) and evaluated it. The results show that DPLL(MAPF) can outperform SMT-CBS and our strategy Eager chokepoints is not a favorable improvement.

Keywords multi-agent path finding (MAPF), cooperative pathfinding, propositional satisfiability (SAT), artificial intelligence, DPLL(MAPF), SAT-based compilation, eager encoding, lazy encoding

Contents

Introduction **1**

1 Backround and theory **3**
1.1 Multi-agent pathfinding 3
1.1.1 Cumulative objectives 5
1.1.2 MAPF methods 5
1.2 Propositional logic 5
1.3 CDCL 6
1.4 Automated planning 8
1.4.1 Classical planning 9
1.4.2 Planning example 10
1.4.3 SATPlan 12
1.5 Eager Encoding: MDD-SAT 14
1.5.1 Optimality and completeness 16
1.6 Constrains 17
1.6.1 Example 18

2 Novel methods **21**
2.1 Eager vs Lazy encoding 21
2.2 Lazy Encoding: SMT-CBS 22
2.3 DPLL(T) 25
2.4 SAT Solver + MAPF = DPLL(MAPF) 27
2.4.1 Adding constrains 29
2.4.2 Eagerly adding of chokepoints 31

3 Prototype **33**
3.1 reLOC 34
3.2 CPF format 35
3.3 encode_MAP 35
3.4 MAPF_handler 36

3.5 Solver 37

4 Experimental evaluation **39**
4.1 Graphs 39
4.2 Checking parameter 41
4.2.1 Uniform 42
4.2.2 Exponential 42
4.2.3 Size of formula 42
4.2.4 Another concepts 43
4.3 Results 43
4.3.1 Measurement description 43
4.3.2 Results and evaluation 44
4.3.3 Eager chokepoints evaluation 47

Conclusion **49**
Future works 50

Bibliography **51**

A Acronyms **57**

B Contents of enclosed CD **59**

List of Figures

0.1 Autonomous drive units and storage pods that contain products and can be moved by the drive units (left) and the layout of a warehouse system (right) [1] 1

1.1 An example of move forbidden by no-swap constrain 4
1.2 An example of move forbidden by following constrain 4
1.3 An example of MAPF consisting of three agents and its solution . 4
1.4 Example of implication graph 7
1.5 Example of a system, which includes planning [2] 9
1.6 Cargo transport as planning 11
1.7 Schema of SATPlan with optimization criterion 12
1.8 Example of MAPF with one agent 15
1.9 A TEG for MAPF from figure 1.8. 15
1.10 An MDD for TEG from figure 1.9. 16
1.11 Simple MAPF problem 18
1.12 MDDs for agents in figure 1.11 18

2.1 diagram of SMT-CBS 24
2.2 SMT-CBS in DPLL(T) perspective 25
2.3 The main components of DPLL(T) 27
2.4 diagram of DPLL(MAPF) 30
2.5 Part of MDD with two chokepoints 31

4.1 MAP file format and image of such map 40
4.2 `maze-32-32` 40
4.3 `empty` 41
4.4 `Warehouse-100-63` 41

List of Tables

4.1 Number of variables 42
4.2 Number of checks without finding collision 43
4.3 Empty 16 runtime results in milliseconds 44
4.4 Empty 32 runtime results in seconds 45
4.5 Maze runtime results in seconds 45
4.6 Warehouse runtime results in seconds 46
4.7 Sums of $T_{DPLL(MAPF)} - T_{SMT-CBS}$ in seconds and percentage improvement counted as $(T_{DPLL(MAPF)}/T_{SMT-CBS})-1$. Columns are sorted by row sum. Negative numbers are highlighted. 46
4.8 Sums of $T_{DPLL(MAPF)_{Choke}} - T_{SMT-CBS}$ in seconds. Last two rows are sum of column and percentage improvement counted as $(T_{DPLL(MAPF)_{Choke}}/T_{SMT-CBS}) - 1$ 47
4.9 Sums of $T_{DPLL(MAPF)} - T_{DPLL(MAPF)_{Choke}}$ in seconds and sum of columns. Positive numbers are highlighted. 47

Introduction

Multi-agent path finding (MAPF) has been well studied by researchers from artificial intelligence, robotics, theoretical computer science and operations research. The task of the (standard) MAPF is to find the paths for multiple agents in a given graph from their current vertices to their targets without colliding with other agents, while at the same time optimizing a cost function. The MAPF problem represents an abstraction for many important real-life tasks from warehouse logistics [3], computer games [4], or ship avoidance [5]. In order to reflect various aspects of real-life applications variants of MAPF have been introduced such as those considered *kinematic constraints* [6], *large agents* [7], or *deadlines* [8].

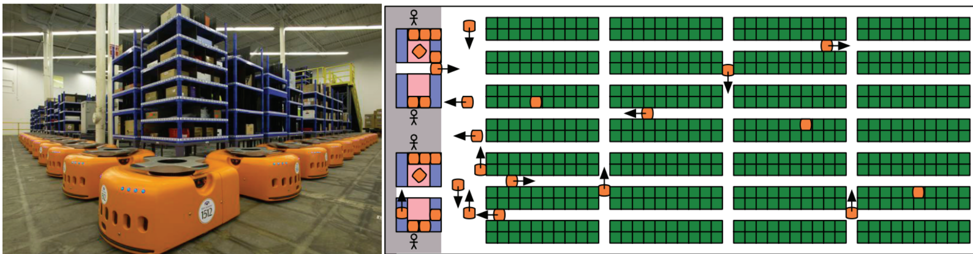


Figure 0.1: Autonomous drive units and storage pods that contain products and can be moved by the drive units (left) and the layout of a warehouse system (right) [1]

Let us see the automated warehouse on figure 0.1. In the warehouse, items need to be moved. In order to move X items, X robots (agents) are given a task to move to these items positions and that is a simple MAPF problem. When the first robot picks up an item, it needs to go to another position, or an inactive robot gets a command to move to some position, we have another instance of MAPF problem, and we need to recalculate it. Therefore, the real life problem of navigating robots in a warehouse is more dynamic and complex (considering rotation, different speed, ...). As we said earlier, the

standard MAPF is an abstraction of the problem, but can be used with some extensions.

In this work, we focus on developing a faster method for the standard formulation of the MAPF problem via converting it into a boolean satisfiability problem (SAT). We are targeting on improving the lazy approach by saving SAT solver from the calculation of infeasible solutions, which should lead to better performance.

Chapter 1

Backround and theory

In this chapter, we will go through the basic concepts and prerequisites required for understanding the later parts of this work.

First, we will introduce MAPF formally. Then we will look at propositional logic, which is crucial for translating MAPF to SAT. Then we will explain what is automated planning and how is SAT used in automated planning. After that we will have everything ready to show you how to encode MAPF to SAT instances and you will have an idea how to solve MAPF by eager encoding, which is our final goal in this chapter.

1.1 Multi-agent pathfinding

Multi-agent path finding (MAPF) [9, 10, 11, 12, 13, 14, 15] is the task of navigating agents from given starting positions to given individual goal positions. The task takes place in an undirected graph $G = (V, E)$. Agents from a set $A = \{a_1, a_2, ..., a_k\}$ are located in vertices[1] of G with at most one agent per vertex.

Valid movement of an agent is either to move to its adjacent vertex or stay at the same vertex.

Additionally, the movements of agents are instantaneous and are possible across edges assuming no other agent is entering the same target vertex. Agents are allowed to enter vertices being simultaneously vacated by other agents. However, the trivial case when a pair of agents swaps their positions across an edge is forbidden in the standard formulation of MAPF.

In other words, we can say we are using *no-swap constraint* (fig.1.1) and do not *following constrain* (fig.1.2).

There are more MAPF constrains like *edge conflict*, *cycle conflict* or *disappear at target*. They will not occur with our setting of MAPF and *disappear at target* is not standart, so we will omit them.

[1]often called nodes

(1) No-swap constrain forbids agents from swapping their position over a single edge. In other words, agents are not allowed to use the same edge at the same time. This constrain is demonstrated on figure 1.1 with two agents, labeled by green and red color.

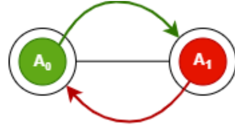


Figure 1.1: An example of move forbidden by no-swap constrain

(2) Following constrain forbids agents from moving to a vertex if it is not empty before the move action has begun, see figure 1.2.

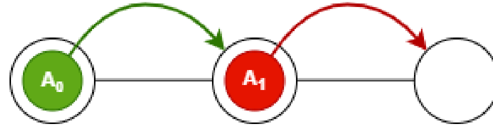
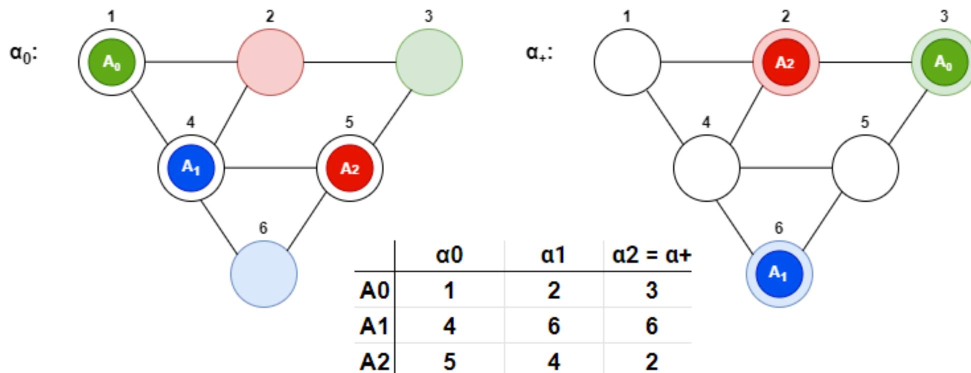


Figure 1.2: An example of move forbidden by following constrain

We usually denote the configuration of agents in the vertices of G at a discrete time step t as $\alpha_t : A \rightarrow V$. Non-conflicting movement transform configuration α_t *instantaneously* into the next configuration α_{t+1}. We do not consider what happens between t and $t+1$ in this discrete abstraction. Multiple agents can move at a time, hence the MAPF problem is inherently parallel.

The initial configuration of agents in vertices of the graph can be written as $\alpha_0 : A \rightarrow V$ and similarly the goal configuration as $\alpha_+ : A \rightarrow V$. The task of navigating agents hence can be expressed as a task of transforming the initial configuration of agents $\alpha_0 : A \rightarrow V$ into the goal configuration $\alpha_+ : A \rightarrow V$ via valid moves.



	α0	α1	α2 = α+
A0	1	2	3
A1	4	6	6
A2	5	4	2

Figure 1.3: An example of MAPF consisting of three agents and its solution

On figure 1.3 there is MAPF problem on the left with three agents and their destinations, on the right is the goal configuration (graph where agents are on their goal destinations), and in the middle bottom is a solution written as table.

1.1.1 Cumulative objectives

We often aim at minimizing the global cumulative cost of two commonly used functions:
(1) sum-of-costs (denoted ξ) is the summation, over all agents, of the number of time steps required to reach the goal location.
Formally, $\xi = \sum_{i=1}^{m} \xi(path(a_i))$, where $\xi(path(a_i))$ is an individual path cost of agent a_i from $\alpha_0(a_i)$ to $\alpha_+(a_i)$.
(2) makespan: (denoted μ) is the total time until the last agent reaches its destination (i.e., the maximum of the individual costs).

While finding a feasible solution of MAPF can be done in polynomial time [16, 17], finding an optimal solution with respect to either the makespan or the sum-of-costs is an NP-hard problem, because it is a generalized problem of $2^n - 1$ sliding puzzle. [18].

1.1.2 MAPF methods

There are many methods for solving MAPF. There are bounded suboptimal [19], suboptimal search [20] and optimal methods [21], such as answer set programming (ASP) [22], constraint satisfaction problem (CSP) [23] and Boolean satisfiability (SAT) [24].

Nevertheless, in this work we are focusing on reduction MAPF to satisfiability.

1.2 Propositional logic

Propositional logic is the simplest logic that we can use for the description of states and relations of a real/fictional world as an abstraction, and then we can use this abstraction as a working environment.

We will go through some definitions that will appear in this work.
Atomic formula is basic statement/proposition that can be assigned *True* or *False*, e.g. It is raining, I like you, $5 < 3$, ...
Formula can also be created as a combination of formulas connected by logical operators (AND, OR, IMPLIES, ...), e.g. It's raining and I like you.

We then use variables as shortcuts for atomic formula, e.g. $x \wedge y$, where $x =$ It's raining, $y =$ I like you
Literal is an atomic formula (proposition) or its negation, e.g. $x, \neg x$
Clause is a disjunction of literals, e.g. $y \vee \neg x$

A clause is true if at least one of the literals is true.
CNF (Conjunctive Normal Form) is a conjunction of one or more clauses, e.g. $(x \wedge y) \vee \neg x$, $x \vee y$, x

Any formula can be represented in CNF. CNF is used as a canonical representation of formulas in many algorithms.

The problem is when we rely on that the formula will be in CNF and it is not, so we have to convert it to CNF, because the conversion may lead to an exponential explosion of the formula. This might rapidly slow the algorithm. Luckily, many real world problems can be naturally represented as a formula in CNF.
Assignment means assigning truth values to formula, e.g. we have formula $y \vee \neg x$ and we will assign values to atomic formulas, such as $x = \mathit{1}$, $y = \mathit{0}$
Assignment A **satisfies** the formula P if and only if $A(P) = True$, e.g. assignment $x = \mathit{1}$, $y = \mathit{0}$ satisfies formula $y \vee \neg x$
SAT (Boolean Satisfiability Problem) is the problem of determining if there exists an interpretation that satisfies a given propositional formula (is there an assignment that makes it true?). Cook–Levin theorem says that SAT is NP-complete [25]. This means that any NP problem can be reduced to SAT and it has been a popular procedure since SAT solvers started improving.

If the formula is unsatisfiable, we denoted that as $UNSAT$. The well-known algorithm for solving SAT is CDCL.

1.3 CDCL

Conflict-driven clause learning (CDCL) is an algorithm for solving the SAT. CDCL was inspired by the older algorithm DPLL. CDCL outperformed DPLL and modern SAT solving algorithms are most often CDCL based.

Davis–Putnam–Logemann–Loveland (DPLL) is an algorithm for solving the CNF-SAT problem (deciding the satisfiability of the formula in CNF).

DPLL algorithm use *Backtracking*, *Unit propagation* and *Pure literal*:

Backtracking - algorithm starts by choosing a propositional variable, assigning it a truth value. By assigning a truth value, it splits the problem into two subproblems (we can image a binary tree, where node is a variable and two edges leading from it represent assigning True and False). Then it continuous on the subproblem with the same logic (splitting problem), if any solution was not found in this subproblem, we will try to solve the second subproblem (with opposite truth value).

Unit propagation - when a clause contains only a single literal, we know which truth value should be assigned to the related variable to satisfy this clause. We then propagate this variable we have just assigned to other clauses.

Pure literal - propositional variable with only one polarity (with or without logical negation $\neg$). Such variable can be assign truth value that literal

is true. Furthermore in CNF if one literal in clause is true, than the clause is true, so we can delete clauses with *pure literal*.

DPLL algorithm uses chronological backtracking (as we introduced), without learning. CDCL uses non-chronological backtrack ("*Backjumping*") with learning on *implication graph*.

Backjumping is form of backtracking. Backtracking always goes up one level in the search tree, while backjumping may go up more levels. This is allowed by knowledge from *implication graph*.

Implication graph is a directed acyclic graph where each vertex represents a variable assignment. When the assignment of one variable leads to the assignment of another variable, we will write that relation as edge. If a graph contains a variable assigned both true and false, we call it conflict. From the graph we can see roots of the conflict.

We will show you an example of an implication graph (see figure 1.4) constructed from the propositional formula defined below. In our implication graph, we will first assign X as true, then we will be applying unit propagation (each edge will have a number representing the clause). We will end with conflict.

We will construct an implication graph (see figure 1.4) as example. We will be using the CNF formula with clauses:

1. $X \vee W \vee Z$
2. $\neg X \vee Y \vee W$
3. $\neg X \vee \neg W$
4. $\neg Y \vee \neg Z$
5. $W \vee Z$

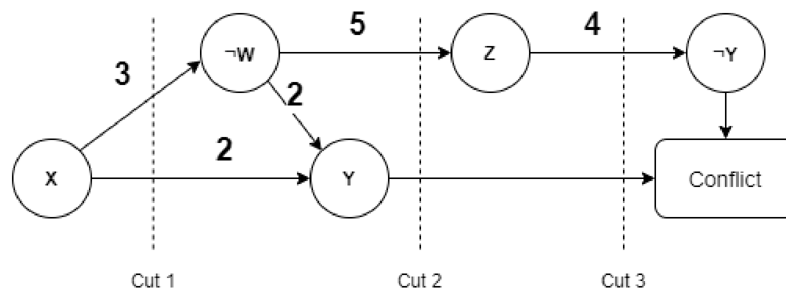


Figure 1.4: Example of implication graph

First, we assign variable X as true and create node 'X'. Then we propagate this literal - first clause is satisfied, second is changed $\neg X \vee Y \vee W \rightarrow \vee Y \vee W$, third is changed $\neg X \vee \neg W \rightarrow \neg W$. In the third clause, unit propagation is

triggered assigning W to be false. So we create a node '$\neg W$' and mark the edge with the number of clauses leading to this decision. In our case, the unit propagation will repeat until a conflict happens and that is our implication graph on figure 1.4.

When we reach a conflict in an implication graph, we can make cuts and learn from them. In figure 1.4 there are 3 cuts and from each we can get a conflicting clause:

- Cut 1: $\neg X$
- Cut 2: $\neg Y \vee W$
- Cut 3: $\neg Y \vee \neg Z$

We provide algorithm 1 which represents high-level overview of the Conflict-Driven Clause-Learning algorithm, where BCP() is backjumping. DECIDE() Chooses an unassigned variable and a truth value for it, if all variables are assigned, it returns false.

Moreover, a scheme of CDCL is provided as part of figure 2.2 in next chapter.

Algorithm 1: CDCL-SAT

```
1 CDCL (CNFformula)
2   while True do
3     while (BCP() = "conflict do
4       backtrack-level ← Analyze-Conflict()
5       if backtrack-level < 0 then
6         return "Unsatisfiable"
7       BackTrack(backtrack-level)
8     if ¬DECIDE() then
9       return "Satisfiable"
```

1.4 Automated planning

The world is in a certain state, but we would like it to be in another state. Thus, we will search for such a plan that will lead us through a sequence of actions to the desired goal state.

That was a very simple definition of planning. However, if we want AI to do the planning for us, we need to define the environment states and actions for the AI so it can make a plan (and we can use propositional logic for that).

Planning is often just a part of an agent or system, see figure 1.5.

Because we talk about planning in artificial intelligence, we can imagine a taxi as an agent (either an autonomous car or taxi driver who acts by our planning). Agent got the task to deliver passenger. First, the agent will plan the route to pick up passenger. Then the agent will act by this plan. The agent can create a plan more dynamically (on the way) or react to the environment (e.g. traffic jam, which could have been messaged from other agents). Or we can have some centralised system that will process all requests and create plans (navigation) to all taxis.

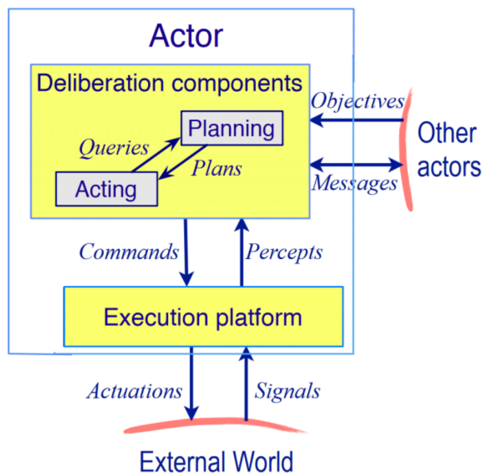


Figure 1.5: Example of a system, which includes planning [2]

Ideally, we would like to have a general planning-problem solver to solve such planning problems. However, the real word is too complex and we reduce the difficulty by simplifying assumptions. So the *automated planning* can be split into many categories based presumptions like:

- deterministic vs non-deterministic (e.g. some vehicles can be delayed by traffic or can have an accident)
- Can be actions apprehended as instantaneous?
- Can the current state be observed unambiguously?
- ...

Simple and popular planning classification is the *classical planning problem*.

1.4.1 Classical planning

Planning have:

- (representation of) goal to achieve
- (representation of) actions that can be performed
- (representation of) the environment

and have to generate a plan to achieve the goal.

Classical planning has these restrictions:

- Deterministic – each action has only one outcome
- Finite system – finitely many states, actions, events
- Static (no exogenous events) – no changes, but the agent's actions
- Attainment goals – existence of a set of goal states
- Sequential plans – a plan is linearly ordered as a sequence of actions
- Implicit time – no time duration, linear sequence of instantaneous states
- Off-line planning – first plan, then act

It is also always *fully observable*, because the initial state is known unambiguously, and all actions are deterministic, so we now exactly in what state we are after any sequence of actions.

To represent the planning problem we use a language based on propositional logic. Two best known languages are PDDL (Planning Domain Definition Language) and older STRIPS (Stanford Research Institute Problem Solver) [26]. These two languages are very similar and they uses operators.

Operator consists of:

- Name of operator
- List of all variables used in the operator
- A Precondition – a set of literals which must be established before the action is performed
- An Effect – a set of literals which are established after the action is performed

An instance of an operator is called action.

1.4.2 Planning example

The basic example of a planning problem is cargo transport, see fig.1.6. The initial state is some cargo C on place A and truck T on place B. The goal state is cargo on place D.

First, we need to define an environment:

At(x,a): object x (truck or cargo) is at place a

In(c,t): cargo c is in truck t

Then we need to define the initial state, goal state, and operators:

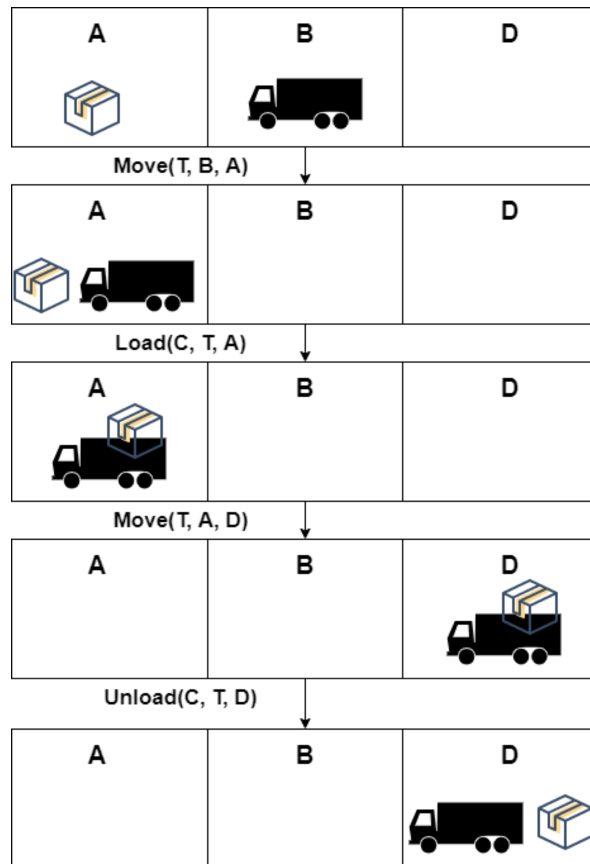


Figure 1.6: Cargo transport as planning

Initial state: $At(C, A) \wedge At(T, B) \wedge Place(A) \wedge Place(B) \wedge Cargo(C) \wedge Place(D) \wedge Truck(T)$
Goal state: *At(Cargo, C)*

Operator(Load(c, t, a))
precond: $At(c, a) \wedge At(t, a) \wedge Cargo(c) \wedge Truck(t) \wedge Place(a)$
effect: $\neg At(c, a) \wedge In(c, t)$
Operator(Unload(c, t, a))
precond: $In(c, t) \wedge At(t, a) \wedge Cargo(c) \wedge Truck(t) \wedge Place(a)$
effect: $At(c, a) \wedge \neg In(c, t)$
Operator(Move(t, from, to))
precond: $At(t, from) \wedge Truck(t) \wedge Place(from) \wedge Place(to)$
effect: $\neg At(t, from) \wedge At(t, to)$

A plan solution to the problem is an ordered set of actions:

[*Move(T, B, A), Load(C, T, A), Move(T, A, D), Unload(C, T, D)*]

Deciding whether any plan exists for a propositional STRIPS instance is PSPACE-complete. Various restrictions can be enforced to decide if a plan exists in polynomial time or at least make it an NP-complete problem [27].

1.4.3 SATPlan

Planning as satisfiability is a powerful approach in automated planning, proposed by Henry Kautz and Bart Selman in their SATPLAN system in 1992 [28]. Reduces the planning problem instance into an instance of classical propositional SAT problem, which is further solved by some SAT solver.

SATPlan is about satisfaction. We want any solution, not necessarily the cheapest or the shortest.

Bounded SATPlan is the question whether there exists a plan of a given length or less (we want an optimal solution).

In the bounded SATPlan, we have a planning problem P and a number n (representing optimization criterion). We have to define formula for *(P,n)* such that any satisfying truth assignment of the formula represent a solution to *(P,n)*. We say that this formula is satisfiable if it has a solution.

The schema of bounded SATPlan is captured on figure 1.7, where CNF is a propositional formula in conjunctive normal form of *(P,n)*.

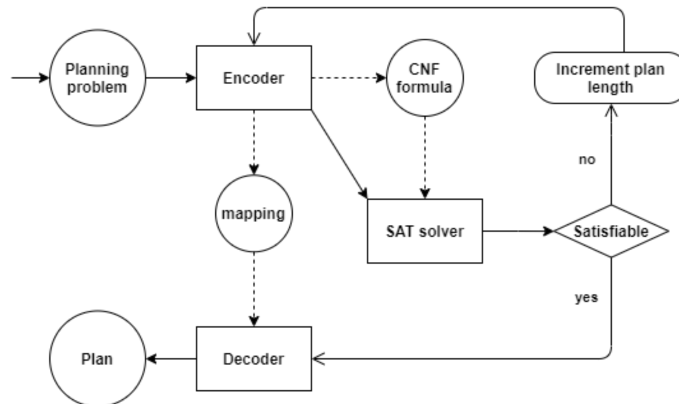


Figure 1.7: Schema of SATPlan with optimization criterion

We also provide algorithm 2, where T_{max} is the upper bound and the rest of the input is a planning problem.

We will show an example of encoding problem into a propositional formula. The problem will be: We have truck T on location A and want to move it to location B. The problem is a simplified problem from subsection 1.4.2 and we will use similar symbols.

Propositional formula for one time step:

Algorithm 2: Framework of bounded SATPlan

```
1 SATPlan (init, transition, goal, T_max)
2     for t = 0 to T_max do
3         cnf ← translate_to_SAT(init, transition, goal, t)
4         assignment ← SAT-solver(cnf)
5         if model is not null then
6             return extract_solution(assignment)
7     return failure
```

1. $At(T, A)^0$ - initial state
2. $At(T, B)^1$ - goal state
3. $At(T, A)^1 \Leftrightarrow (Move(T, B, A)^0 \vee At(T, A)^0)$ - successor state axiom
4. $At(T, B)^1 \Leftrightarrow (Move(T, A, B)^0 \vee At(T, B)^0)$ - successor state axiom
5. $Move(T, A, B)^0 \Rightarrow (At(T, A)^0 \wedge At(T, B)^1 \wedge \neg At(T, A)^1)$ - operator
6. $Move(T, B, A)^0 \Rightarrow (At(T, B)^0 \wedge At(T, A)^1 \wedge \neg At(T, B)^1)$ - operator
7. $(\neg Move(T, B, A)^0 \vee \neg Move(T, A, B)^0)$ - restriction for only one action

The upper index represents time. The numbered lines represent clauses (although they are not, but let us imagine we can transform them any time) and with the conjunction of lines we get CNF, which can be put into SAT solver.

The initial and goal states are marked blue, because they are true. Our goal is to find an assignment that all lines are true. In the next step, we propagate the initial and goal state. We will mark true literals with blue and false ones with red:

3. $At(T, A)^1 \Leftrightarrow (Move(T, B, A)^0 \vee At(T, A)^0)$
4. $At(T, B)^1 \Leftrightarrow (Move(T, A, B)^0 \vee At(T, B)^0)$
5. $Move(T, A, B)^0 \Rightarrow (At(T, A)^0 \wedge At(T, B)^1 \wedge \neg At(T, A)^1)$
6. $Move(T, B, A)^0 \Rightarrow (At(T, B)^0 \wedge At(T, A)^1 \wedge \neg At(T, B)^1)$
7. $(\neg Move(T, B, A)^0 \vee \neg Move(T, A, B)^0)$

To satisfy 4. line we have to mark $Move(T, B, A)^0$ as true. We will continue assigning/coloring until we have:

3. $At(T, A)^1 \Leftrightarrow (Move(T, B, A)^0 \vee At(T, A)^0)$
4. $At(T, B)^1 \Leftrightarrow (Move(T, A, B)^0 \vee At(T, B)^0)$

5. $Move(T, A, B)^0 \Rightarrow (At(T, A)^0 \wedge At(T, B)^1 \wedge \neg At(T, A)^1)$
6. $Move(T, B, A)^0 \Rightarrow (At(T, B)^0 \wedge At(T, A)^1 \wedge \neg At(T, B)^1)$
7. $(\neg Move(T, B, A)^0 \vee \neg Move(T, A, B)^0)$

We ended with all satisfied lines, thus the problem is solvable with this solution. We can get a solution as a set of actions if we select all true operators. In our case, it is $Move(T, B, A)^0$.

1.5 Eager Encoding: MDD-SAT

The idea behind MDD-SAT [29] is to construct a *complete Boolean model*, a propositional formula $\mathcal{F}(\xi)$ according to the following definition.

Definition 1 ***(complete model).*** *Propositional formula $\mathcal{F}(\xi)$ is a complete Boolean model for MAPF Σ iff $\mathcal{F}(\xi)$ is satisfiable $\Leftrightarrow$ Σ has a solution of sum-of-costs ξ.*

Being able to construct $\mathcal{F}(\xi)$ for solvable MAPF, one can obtain the optimal sum-of-costs by consulting the SAT solver with a series of queries about $\mathcal{F}(\xi_0)$, $\mathcal{F}(\xi_0 + 1)$, ... until a satisfiable formula is found, where ξ_0 is a lower bound on the sum-of-costs such as the sum of shortest paths of individual agents. This iterative scheme works due to the fact that satisfiability of $\mathcal{F}(\xi)$ is a non-decreasing function in parameter ξ.

The construction of $\mathcal{F}(\xi)$ must ensure that a valid MAPF solution can be extracted from its satisfying assignment. This is done by representing the configurations of agents at all relevant time steps before they reach their goals via propositional variables. We first make a *time expanded graph* (TEG) of the underlying graph G [24] for each agent, a directed acyclic graph obtained by copying vertices of G for all relevant time steps. A directed edge is introduced into TEG for each pair of nodes from consecutive copies corresponding to vertices that are connected in G. In addition to this, nodes from consecutive copies corresponding to identical vertices are connected by directed edges as well to represent wait actions. A directed path in TEG corresponds to an individual plan of the agent (sequence of its moves). The construction of TEG is shown in Figures 1.8 and 1.9.

It remains to ensure that satisfying assignments of $\mathcal{F}(\xi)$ correspond to non-conflicting paths. A Boolean variable $\mathcal{X}_u^t(a_i)$ is introduced for every node u_t from TEG (a node corresponding to $u \in V$ at time step t) for each agent agent $a_i \in A$; $\mathcal{X}_u^t(a_i)$ is *TRUE* iff agent a_i is located in u at time step t. Similarly, we introduce Boolean variables for edges denoted $\mathcal{E}_{u,v}^t(a_i)$, with analogous meaning; $\mathcal{E}_{u,v}^t(a_i)$ is *TRUE* iff agent a_i moves from u to v starting the move at time step t. Finally constraints are added so that truth assignments are restricted to those that correspond to valid solutions of a given MAPF. The

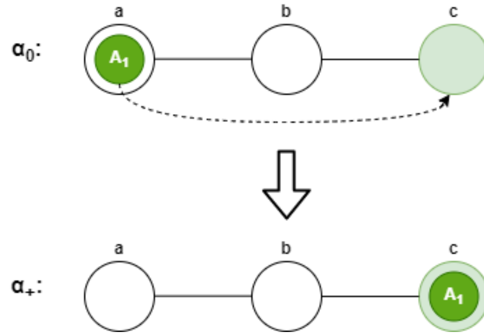


Figure 1.8: Example of MAPF with one agent

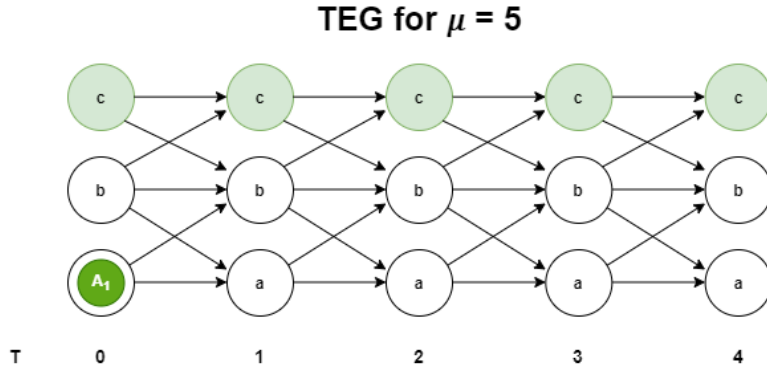


Figure 1.9: A TEG for MAPF from figure 1.8.

added constraints together ensure that $\mathcal{F}(\xi)$ is a *complete Boolean model* for given MAPF. We will closely look at the constrains in the next section.

The MDD-SAT solver implements an improvement over TEGs based on the observation that not all nodes in TEG can be reached under a given sum-of-costs ξ. The unreachable nodes can be pruned out from the TEG resulting in a directed acyclic graph called *multi-value decision diagram* (MDD) which has been introduced in the context of MAPF by search-based solvers [12, 13] (see Figure 1.10). Adoption of MDDs in MDD-SAT resulted in much smaller formulae.

Algorithm 3 shows the pseudo-code of MDD-SAT.

The SATPlan schema on figure 1.7 is also scheme for MDD-SAT workflow.

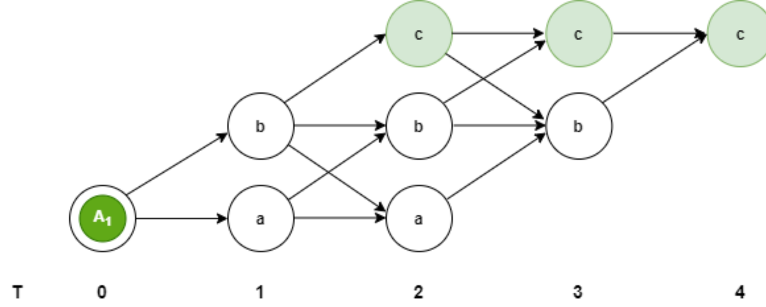


Figure 1.10: An MDD for TEG from figure 1.9.

Algorithm 3: Framework of MDD-SAT, an optimal SAT-based MAPF solver

```
1  MDD-SAT (Σ = (G = (V, E), A, α_0, α_+))
2      paths ← {shortest path from α_0(a_i) to α_+(a_i)|i = 1, 2, ..., k}
3      ξ ← Σ_{i=1}^{k} ξ(paths(a_i))
4      while TRUE do
5          F(ξ) ← encode-Complete(ξ, G, A, α_0, α_+)
6          assignment ← consult-SAT-Solver(F(ξ))
7          if assignment ≠ UNSAT then
8              paths ← extract-Solution(assignment)
9              return paths
10         ξ ← ξ + 1
```

1.5.1 Optimality and completeness

Definition 2 ***(Optimality)***. *An algorithm is optimal when it returns an optimal solution (with respect to a criterion).*

Definition 3 ***(Completeness)***. *An algorithm is complete if it*

- *always terminates*
- *returns valid solution when the input is valid*

Algorithm 3 clearly terminates for solvable MAPF instances as we start seeking a solution of $\xi = \xi_0$ and increment ξ to all possible values (algorithm using makespan μ would proceed the same way - initialize μ and incrementing it). So if we iterate over ξ values starting at minimum ξ_0, we must at some point obtain a solution and it will be optimal, because no previous solution was found $\rightarrow$ *optimality* ensured.

In the case of unsolvable instances, algorithm 3 would be iteratating forever. The unfeasibility of an MAPF instance can be checked separately by a polynomial-time complete sub-optimal algorithm such as push-and-rotate [30].

This ensures *completeness* and this technique will not worsen our time complexity.

1.6 Constrains

In the previous chapter, we learned how to construct variables from MAPF and told that we need to add constrains to them to have *complete model*.

We can formalize MAPF rules with four types of constrains and all of them can be easily interpreted in CNF:

1. **Edges**

   Agents can move in the graph through edges. We will look at example from figure 1.10. If an agent is at time 1 in vertex a (denoted as a_1) then in time 2 agent can be in a_2 or b_2 but can't be in c_2. We will write this constraint in propositional logic as $a_1 \Rightarrow (a_2 \vee b_2)$. This can be rewritten in clause: $\neg a_1 \vee a_2 \vee b_2$

2. **Collisions**

   More agents cannot occupy one vertex. In other words, the sum of agents for each vertex and time is less than or equal 1. Therefore, we disallow every pair: $\neg\chi_v^t(\alpha_i) \vee \neg\chi_v^t(\alpha_j)$; $\forall$ *i,j* $\in$ *Agents* $\wedge\, i \neq j$; $\forall v \in Vertices$; *for each* $t = 0 \ldots \mu$
   Prohibition of pair is clause so we create CNF by conjunction of these clauses.

3. **Agent can be in one vertex at one time**

   This might seem obvious to us, but we have to tell the program explicitly.

   We will do this same way as collisions - prohibition of pairs. By this, we allow an agent to be nowhere at one time. We can afford to do this, because Edge constraint will ensure that the agent will not disappear.

   We will look at the example from figure 1.10. Agent A_1 in time 2 cannot be simultaneously in vertices a, b and c. We can write this as CNF:
   $(\neg a_2 \vee \neg b_2) \wedge (\neg a_2 \vee \neg c_2) \wedge (\neg b_2 \vee \neg c_2)$
   Therefore assignment $a_2 = 0, b_2 = 0, c_2 = 0$ is valid for this constraint as we spoke few lines earlier.

4. **Swaps**

   In the standard MAPF we forbid swaps (see figure 1.1). We will do that by forbidding all four variables, that represent such swap - $\neg\chi_v^t(\alpha_i) \vee \neg\chi_w^t(\alpha_j) \vee \neg\chi_w^{t+1}(\alpha_i) \vee \neg\chi_v^{t+1}(\alpha_j)$; $\forall$ *i,j* $\in$ *Agents*$\wedge i \neq j$; $v, w \in Vertices$; $t = 0 \ldots \mu - 1$

1.6.1 Example

We will demonstrate how exactly are these constrains created using a simple MAPF example, see figure 1.11.

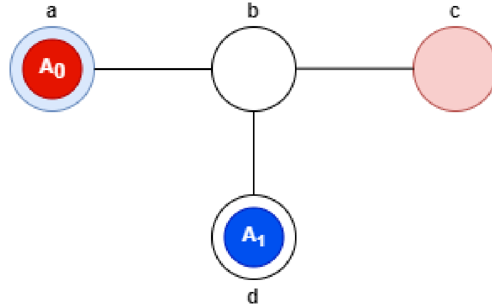


Figure 1.11: Simple MAPF problem

For each agent, we will construct MDD with $\mu = 2$, see figure 1.12. We can see that this makespan is insufficient for this problem (we need at least 3 time steps to solve it), thus this encoding will not be *complete model.* Algorithm 3 would proceed the same way, first it would create these MDDs and explicitly encoding, then try to solve it resulting in UNSAT, after that increasing μ to 3, make new encoding, which will be solvable.

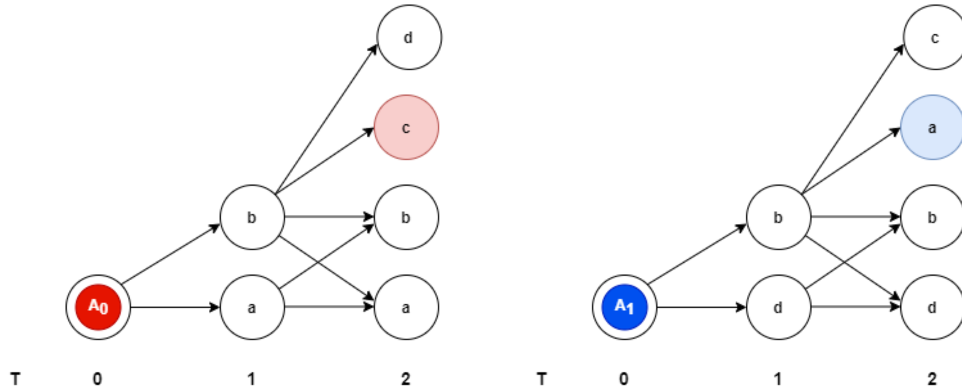


Figure 1.12: MDDs for agents in figure 1.11

The variables for MDDs in figure 1.12 can be written as: $a^0_0, a^1_0, a^2_0, b^1_0, b^2_0, c^2_0, d^2_0, d^0_1, d^1_1, d^2_1, b^1_1, b^2_1, a^2_1, c^2_1$, where letter represents vertex, index represents time and upper index represents agnet.

Then we use the variables to construct the constrains as clauses. These clauses will be used to create CNF formula by conjuncting them. We will show all clauses divided into categories:

- **Edges**

  We will divide clauses for each agent/MDD.

Agent 0	Agent 1
$\neg a_0^0 \vee a_0^1 \vee b_0^1$	$\neg d_1^0 \vee d_1^1 \vee b_1^1$
$\neg a_0^1 \vee a_0^2 \vee b_0^2$	$\neg d_1^1 \vee d_1^2 \vee b_1^2$
$\neg b_0^1 \vee a_0^2 \vee b_0^2 \vee c_0^2 \vee d_0^2$	$\neg b_1^1 \vee a_1^2 \vee b_1^2 \vee c_1^2 \vee d_1^2$

- **Collisions**

$$\neg b_0^1 \vee \neg b_1^1$$
$$\neg a_0^2 \vee \neg a_1^2$$
$$\neg b_0^2 \vee \neg b_1^2$$
$$\neg c_0^2 \vee \neg c_1^2$$
$$\neg d_0^2 \vee \neg d_1^2$$

- **Agent can be in one vertex at one time**

  We will divide clauses for each agent/MDD.

Agent 0	Agent 1
$\neg a_0^1 \vee \neg b_0^1$	$\neg d_1^1 \vee \neg b_1^1$
$\neg a_0^2 \vee \neg b_0^2$	$\neg d_1^2 \vee \neg b_1^2$
$\neg a_0^2 \vee \neg c_0^2$	$\neg d_1^2 \vee \neg c_1^2$
$\neg a_0^2 \vee \neg d_0^2$	$\neg d_1^2 \vee \neg a_1^2$
$\neg b_0^2 \vee \neg c_0^2$	$\neg b_1^2 \vee \neg c_1^2$
$\neg b_0^2 \vee \neg d_0^2$	$\neg b_1^2 \vee \neg a_1^2$
$\neg c_0^2 \vee \neg d_0^2$	$\neg c_1^2 \vee \neg a_1^2$

- **Swaps**

$$\neg b_0^1 \vee \neg d_1^1 \vee \neg b_1^2 \vee \neg d_0^2$$
$$\neg b_1^1 \vee \neg a_0^1 \vee \neg b_0^2 \vee \neg a_1^2$$

We constructed 29 clauses. This formula is unsatisfiable, because both agents would collide on vertex b if they have to get to their destination in 2 time steps. Therefore, we increase makespan μ to 3 and construct a new CNF formula, which contains 57 clauses (twice as much).

So we can imagine that the number of clauses scale quite well with a MAPF instance. Although it is not easy to define scaling because it depends on many corresponsive features, e.g. the number of agents, agent's paths, size and complexity of the graph,...

Chapter 2

Novel methods

2.1 Eager vs Lazy encoding

Compiling MAPF to other formalisms for which an off-the-shelf solver is available is a popular solving approach. Optimal solvers for MAPF based on the compilation to *constraint satisfaction problem* (CSP) [23], *answer set programming* (ASP) [22], integer programming (IP) [31], and Boolean satisfiability (SAT) [24] currently exist.

In this work, we focus on compilation of MAPF to SAT [32]. Contemporary techniques of MAPF compilation regard the SAT solver as an external tool having only limited interaction with the main MAPF solver. Often, a formula, called a *complete Boolean model*, encoding a question whether there exists a solution to input MAPF of a specified cost is constructed in a single-shot and consulted with the SAT solver. The task of the SAT solver is to determine the truth value assignment of all decision variables satisfying the formula or the answer that such assignment does not exist. This scheme has been used in MDD-SAT (see section 1.5), the first sum-of-costs optimal SAT-based MAPF solver [29].

The disadvantage of the SAT consultation scheme from MDD-SAT is twofold:
(1) The complete Boolean model must be fully specified so that the equivalence between the solvability of the input MAPF instance and the satisfiability of the Boolean model is established, which may result in a large formula.
(2) The SAT solver in this scheme acts as a black box for the main solver that has no way to interact with the SAT solver until it finishes.

The first disadvantage has been addressed in SMT-CBS [33] (will be properly explained in next section 2.2), a sum-of-costs optimal SAT-based solver that introduces the concept of *incomplete Boolean models*. Using the incomplete Boolean model, the input MAPF instance is not fully specified so only the implication between the solvability of the input MAPF and the Boolean model holds. Such a relaxation requires that the MAPF solution obtained from the

truth value assignment of the model is checked for consistency against MAPF movement rules (as those are not fully encoded in the model). If the rules are not violated, the solving process is finished. Otherwise the model needs to be refined by constraints that forbid the detected MAPF rule violations and consulted with the SAT solver again [2].

The benefit of using incomplete Boolean models is that often the solving process finishes with a small formula since a lot of constraints specifying the MAPF problem completely will not come into effect (for example we do not need to specify all collision avoidance constraints between agents in a sparse instance). A similar process has been adopted in the compilation of MAPF to IP [31].

However, the second disadvantage only becomes more apparent in SMT-CBS. The main MAPF solver must wait until the complete truth value assignment is found by the SAT solver. In SMT-CBS not every rule is encoded, thus SAT solver may make an early decision leading to MAPF rule violation. Such violation cannot be detected by SAT solver because it is not aware of the rules and the communication with the main solver that knows the absent rules. This disadvantage will be covered by DPLL(MAPF) approach (which will be introduced in further section 2.4). DPLL(MAPF) will be able to detect such early decision violating MAPF rules.

2.2 Lazy Encoding: SMT-CBS

An important innovation step from MDD-SAT is represented by SMT-CBS [33], an optimal SAT-based solver employing the idea of encoding MAPF as a Boolean formula **lazily**. The lazy encoding is formalized through the concept of *incomplete Boolean model* defined as follows.

Definition 4 ***(incomplete model).*** *Propositional formula $\mathcal{H}(\xi)$ is an incomplete Boolean model of MAPF Σ iff $\mathcal{H}(\xi)$ is satisfiable $\Leftarrow \Sigma$ has a solution of sum-of-costs ξ.*

In an incomplete Boolean model $H(\xi)$ we do not specify all constraints (see section 1.6) defining the movement rules of MAPF. We rely on being lucky to obtain a valid MAPF solution from an under-specified formulation. Hence, in the contract to MDD-SAT, we need to add a check that the solution extracted from the satisfying assignment of $\mathcal{H}(\xi)$ is consistent, that is, we need to ensure that agents do not jump, do not disappear, do not appear from nothing etc. since the correspondence between non-conflicting directed paths in MDDs and satisfying assignments of $H(\xi)$ is no longer preserved in the under-specified formulation. If the consistency check is successfully passed, we can return

[2] This process is analogous to conflict-based search (CBS) [13] where MAPF rule violations (conflicts between pairs of agents) are resolved via branching the search.

the valid MAPF solution extracted from the model, otherwise the incomplete model needs to be refined.

Algorithm 4: SMT-CBS algorithm for MAPF solving

```
1  SMT-CBS (Σ = (G = (V, E), A, α_0, α_+))
2      conflicts ← ∅
3      paths ← {shortest path from α_0(a_i) to α_+(a_i) | i = 1, 2, ..., k}
4      ξ ← Σ_{i=1}^{k} ξ(paths(a_i))
5      while TRUE do
6          (paths, conflicts) ← SMT-CBS-Fixed(conflicts, ξ, Σ)
7          if paths ≠ UNSAT then
8              return paths
9          ξ ← ξ + 1
10 SMT-CBS-Fixed(conflicts, ξ, Σ)
11     H(ξ) ← encode-Incomplete(conflicts, ξ, Σ)
12     while TRUE do
13         assignment ← consult-SAT-Solver(H(ξ))
14         if assignment ≠ UNSAT then
15             paths ← extract-Solution(assignment)
16             collisions ← check-Consistency(paths) /* via DECIDE_MAPF */
17             if collisions = ∅ then
18                 return (paths, conflicts)
19             for each (a_i, a_j, v, t) ∈ collisions do
20                 H(ξ) ← H(ξ) ∪ {¬X_v^t(a_i) ∨ ¬X_v^t(a_j)}
21                 conflicts ← conflicts ∪ {[(a_i, v, t), (a_j, v, t)]}
22         return (UNSAT,conflicts)
```

The pseudo-code of SMT-CBS algorithm is shown as Algorithm 4. The high-level loop that iterates sum-of-costs is the same as in MDD-SAT. The difference rests in low-level loop within the SMT-CBS-Fixed function that answers the existence of a solution of a specified sum-of-costs ξ in which the incomplete Boolean model is refined. Various strategies of refinement can be adopted. The SMT-CBS starts with $H(\xi)$ where only collision avoidance constraints are omitted. The constraints making agents to move along directed paths in MDDs are present. Hence, the consistency check consists in a check for collisions (lines 16-18). If a collision in a vertex is detected, say a collision between agents a_i and a_j in $v \in V$ at time step t, then the model is refined with collision avoidance constraints, in this case clause $\neg\mathcal{X}_v^t(a_i) \vee \neg\mathcal{X}_v^t(a_j)$ to $H(\xi)$ is added (line 20). Eventually $H(\xi)$ may converge towards the complete Boolean model, however often a solution is obtained much earlier.

Terminology alert: We introduced *collisions* as constrain in section 1.6. There is also introduced the *swap* constrain. In lazy encoding we see both swap and collisions contrains as collisions. We can look at swap as a collision on an edge.

We present the diagram of SMT-CBS on figure 2.1. It may seem com-

plicated, but we can look at the diagram as an extension of the scheme of SATPlan on figure 1.7 with addition of:

- **Separating conflicts** will produce *incomplete model* consisting of *Problem without conflicts* and *Collisions*. The *Encoder* takes these two separated parts for encoding.

- **Collisions** are encountered conflicts. They are empty at start and can be extended by action *Add conflicts*.

- **Paths** (potential plan) are decoded *assignment*. Because we are using *incomplete model*, the paths may violate some unencoded MAPF rules. The *Paths* can become *Plan* after successful *Consistency check*.

- **Consistency check** will find all *Conflicts* in *Paths* and give them all to *Add conflicts*. If none is found then our diagram returns *Plan* and ends.

- **Add conflicts** will add conflicts to *Collisions* that arose during *Consistency check*.

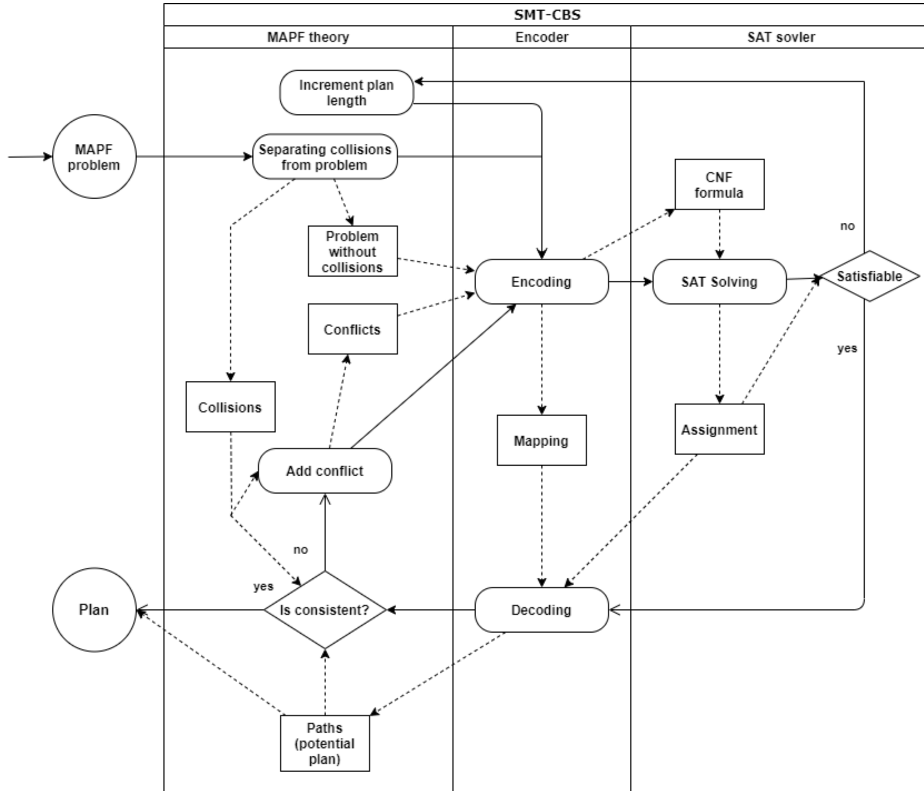


Figure 2.1: diagram of SMT-CBS

The optimality and completeness are the same as in MDD-SAT, see section 1.5.1. Completeness is not changed because a complete check is performed when the assignment is full.

2.3 DPLL(T)

DPLL(T) [34, 35] which is a framework for integrating the SAT solver with a decision procedure, usually denoted $DEDUCTION_T$, for the conjunctive fragment of some first-order theory T. The two components of DPLL(T) together form a decision procedure for general problems in theory T with arbitrary Boolean structure where the SAT solver component takes care of the Boolean structure and the $DEDUCTION_T$ component checks the consistency of assignments suggested by the SAT solver against axioms of T.

Figure 2.2 describes the workflow of using SAT solver with a decision procedure using lazy encoding (such as SMT-CBS). It is a very close diagram to DPLL(T), therefore we describe it and then show how to enhance it into DPLL(T).

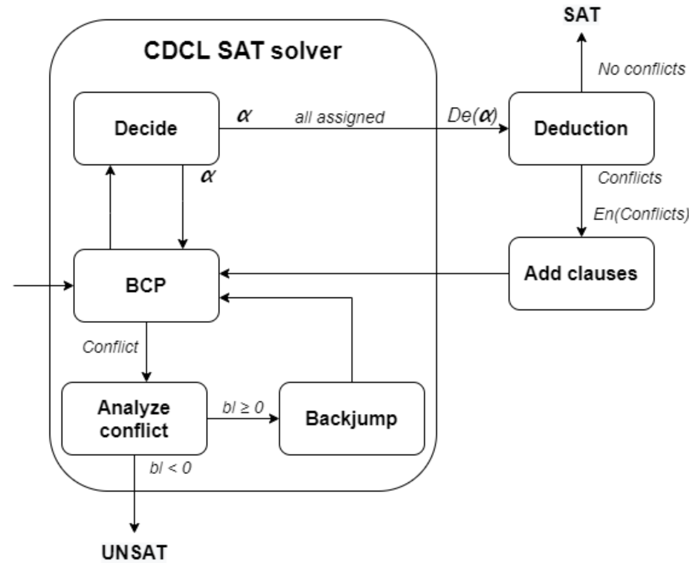


Figure 2.2: SMT-CBS in DPLL(T) perspective

First we will look at *CDCL*[3] *SAT solver* in figure 2.2. It is high-level overview of the Conflict-Driven Clause-Learning algorithm described in section 1.3. It consists of:

[3]The proper name of the algorithm hence should be CDCL(MAPF), but we follow the notation DPLL(T) used in the literature.

- **Decide** – Chooses an unassigned variable and a truth value for it. If all variables are assign, it gives an assignment to Deduction.
- α – is an assignment (either partial or full).
- **BCP** (Boolean Constraint Propagation) – Repeated application of the unit clause rule until either a conflict is encountered or there are no more implications.
- **Analyze conflict** – It is responsible for computing the backtracking level, detecting global unsatisfiability, and adding new constraints to the search in the form of new clauses.
- bl – is the backtracking level, i.e., the decision level to which the procedure backtracks.
- **Backjump** – Sets the current decision level to bl and erases assignments at decision levels larger than bl.

On top of SAT solver, there is another functionality that has the ability to reason about theory T:

- **Deduction** – decision procedure, which takes decoded assignment and returns conflicts with theory T.
- **Add cluses** – simply adds encoded conflicts.
- **De()** – decoding variables into a theory instance.
- **En()** – encoding theory instances into CNF formula.

This figure is SMT-CBS when we identify our theory T as MAPF. We will take further steps to improve it into DPLL(T)/DPLL(MAPF).

In section 2.1 we mentioned the disadvantage that this scheme has - No way to interact with SAT solver until it finishes. We can see that *deduction* is called after a full satisfying assignment is found. Thus, the time taken from making a mistake[4] to complete the assignment is wasted. In DPLL(T) we can call *deduction* before full assignment.

In figure 2.3 we have same functionalities as in previous figure 2.2. The most spotable is how they are connected. We can see that there is no more SAT solver, this is because the parts reasoning about the theory were integrated closer into SAT solver, thus, we could call it Theory solver instead of SAT solver. We can still imagine making a horizontal line separating **Add clauses** and **Deduction**, but the communication through this imaginary line would be different from the figure 2.2 and would not resemble communication with SAT solver.

[4]by mistake we mean assignment that violate MAPF rules

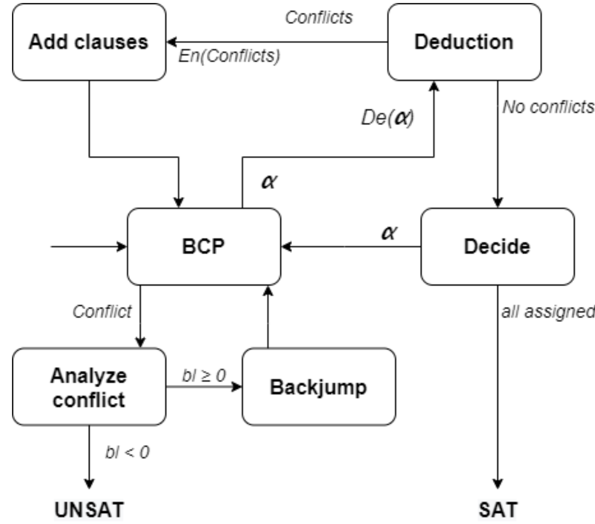


Figure 2.3: The main components of DPLL(T)

Deduction is invoked after no further implications can be made by BCP. Deduction finds conflicts and communicates them to the SAT solver part form of the propositional formula. Hence, in addition to implications, there are now also implications due to the theory T. This is called theory propagation. Thus, in addition to the normal Boolean constraint propagation (BCP) performed by the SAT solver, there is now also theory propagation.

2.4 SAT Solver + MAPF = DPLL(MAPF)

The SAT solver in the SMT-CBS framework is put in a position that it does not completely understand what is the problem being solved. It may happen that early variable assignments in the SAT solver's search are inconsistent with MAPF rules. However, the MAPF solution consistency check can be made only after the SAT solver assigns all variables. Hence, we suggest to make a further innovation step from SMT-CBS and to check consistency also in partial assignments of incomplete Boolean models. This step requires very close integration of the SAT solver and the MAPF theory.

The pseudo-code of the low-level search of DPLL(MAPF) that checks the existence of MAPF solution for fixed sum-of-costs ξ is shown as Algorithm 5 (it is based on the DPLL(T) pseudo-code from [36]). The algorithm follows the design of *conflict-directed clause learning* SAT solver [37, 38] into which MAPF consistency check is added when the SAT solver has a partial assignment of Boolean variables at hand (lines 19-20). After a collision is detected the incomplete Boolean model is refined with new collision avoidance clauses (lines 22 and 23) and backtracking based on the analysis of the *implication graph*

Algorithm 5: Framework of SAT-based MAPF solver

```
1  DPLL-MAPF-Fixed (conflicts, ξ, Σ)
2      H(ξ) ← encode-Incomplete(conflicts, ξ, Σ)
3      if propagate-Unit() = UNSAT then
4          return (UNSAT,conflicts)
5      while TRUE do
6          (x, v) ← assign-Variable()
7          if x = NULL then
8              paths ← extract-Solution(assignment)
9              return (paths, conflicts)
10         assignment ← assignment ∪ {x = v}
11         repeat
12             while propagate-Unit() = UNSAT do
13                 backtrackLevel ← analyze-Conflict()
14                 if backtrackLevel < 0 then
15                     return (UNSAT,conflicts)
16                 else
17                     back-Track(backtrackLevel)
18             paths ← extract-Partial-Solution(assignment)
19             collisions ← check-Consistency(paths) /* via DECIDE_MAPF */
20             for each (a_i, a_j, v, t) ∈ collisions do
21                 H(ξ) ← H(ξ) ∪ {¬X_v^t(a_i) ∨ ¬X_v^t(a_j)}
22                 conflicts ← conflicts ∪ {[(a_i, v, t), (a_j, v, t)]}
23         until collisions = ∅;
```

is initiated (lines 13-18). The backtracking phase adds a conflict clause that forbids repeating the conflicting assignment in the future. The high-level that increments the sum-of-costs is the same as in SMT-CBS.

On figure 2.4 we can see the diagram of DPLL(MAPF). This diagram is almost the same as diagram of SMT-BS on figure 2.1. We just added 3 components marked with a red border. Although we have swimlanes in diagram showing us into which category the components belongs, we saw in figure 2.3 that DPLL(T) requires tighter integration and final MAPF solver will have components from all three categories.

Red components description:

- **Is assignment complete**

  In DPLL(MAPF), SAT solver can produce *partial assignment.* In that case we want to check collisions of that assignment. If the assignment is complete, we continue the same way as SMT-CBS.

- **Decode partial assignment**

  This is basically the same action as *decoding.* This action can decode partial and full assignment, but *decode* is not supposed to decode the

partial assignment and thus it probably cannot do that. We can look at *decode partial assignment* as superset of *decode*.

- **Is consistent**

  This function appears two times in our diagram with the same name but different color. Although they do the same thing, the most evident distinguishability is the different workflow. Furthermore, the black function in MAPF theory might not be designed for the plan created by *partial assignment* (implementation detail).

These new components doesn't break the optimality and completeness of SMT-CBS. Thus, DPLL(MAPF) has these properties.

2.4.1 Adding constrains

The question is which constraints (see section 1.6) should be treated lazily. In SMT-CBS only collisions are treated lazily, but it doesn't mean that all lazy encodings have to do it the same way.

In our diagram (figure 2.4) we extended SMT-CBS, which means that we adopted the same strategy. This strategy is reasonable, but our approach does not necessarily has to follow the same strategy.

If we want to see how DPLL(MAPF) improved SMT-CBS method by comparing their runtimes we should keep the same strategy. But on top of that we could try another strategy.

If we want to change the SMT-CBS lazy encoding strategy there are two main paths to choose from:

1. Constrains that ware added lazily we can add eagerly.

2. Constrains that ware added eagerly we can add lazily.

If we would follow the first option, then we would omit some constraints that ensure that agents are not teleporting, disappearing, or copying themself. We think that it is better to leave this constrains in eager encoding.

This lead us to the second option. We have to add some constrains in an eager way. This means we have to pick some collisions that we encode at the initial encoding.

We came up with the idea of *Eagerly adding of chokepoints*.

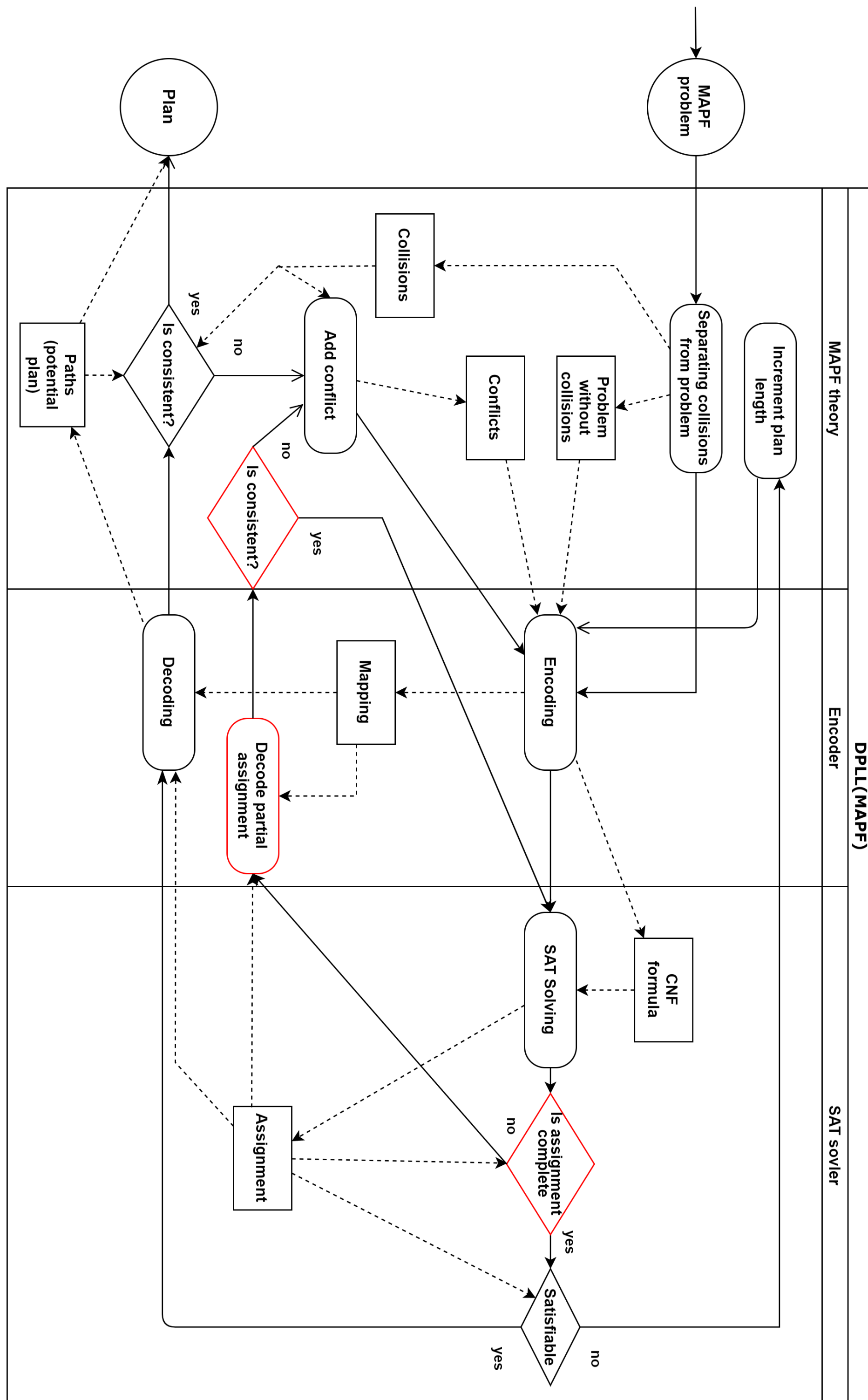


Figure 2.4: diagram of DPLL(MAPF)

2.4.2 Eagerly adding of chokepoints

Chokepoint is basically a narrow space, e.g. Battle of Thermopylae, it became famous because Spartans used the narrow space (chokepoint) as a defending position against their enemies who outnumbered them.

Chokepoint in our context will be a narrow passage in MDD.

We will define a chokepoint as time slice in MDD when there are no more than 2 vertices. Excluding the initial and final time where is only one vertex and no collisions are possible.

On figure 2.5 we have a part of MDD where are two chokepoints at time 1 and 3.

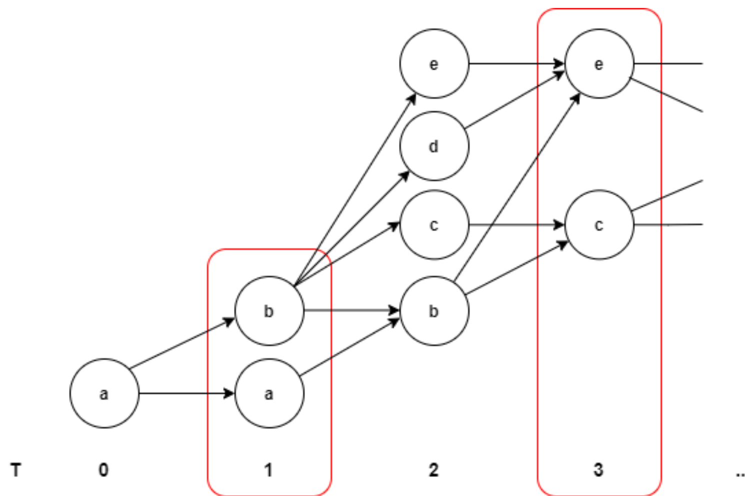


Figure 2.5: Part of MDD with two chokepoints

We propose to encode collisions in chokepoints at the beginning, because it is easy in to find chokepoints and generally collisions here should be more likely than in wide time slices.

Chapter 3

Prototype

First, we implemented SMT-CBS, because it is very similar to DPLL(MAPF) and then DPLL(MAPF) itself. Because they share most of the code, the evaluation of performance (comparing our implementation of SMT-CBS and DPLL(MAPF)) will not differ because of the implementation, but because of the algorithm's design.

The implementation was done in C++ language.

DPLL(MAPF)'s foundation of `main()` function is captured in listing 3.1. We are using makespan instead of sum-of-costs, because it was more comfortable for coding and both are equally good. First we go through inputs of `main()`:

- string Filename – path to MAPF instance (.cpf file, explained in section 3.2).
- int checking_parameter – number for checking parameter, see section 4.2.
- bool testing – False if we want a human readable output, True gives runtine in seconds as output.
- bool exp – True if we want our checking parameter grow exponentially, False for uniform, see section 4.2.

Listing 3.1: Simplified main(filename, checking_parameter, testing, exp)

```
1  MAPF_handler problem = MAPF_handler(filename); // load file
2  makespan = problem.get_shortest_path();
3  while (true)
4  {
5      Solver solver; // Glucose with modifications
6      problem.encode(solver, makespan);
7      if (solver.solve()) // compute valid solution
8      {
9          exit(0); // solved
```

```
10            break;
11        }
12        makespan++;
13    }
```

Listing 3.2: Simplified SMT-CBS main(filename)

```
1   MAPF_handler problem = MAPF_handler(filename); // load file
2   makespan = problem.get_shortest_path();
3   while (true)
4   {
5     Solver solver; // Glucose with add_clauses() function
6     problem.encode(solver, makespan);
7     if (solver.solve()) // may compute invalid solution
8     {
9       vector<vector<int>> collisions;
10      collisions = problem.check_collisions(solver.my_model)
11      if (collisions.empty())
12        exit(0); // solved
13      else
14        solver.add_caulses(collisions)
15      break;
16    }
17    makespan++;
18  }
```

We added listing 3.2, where is SMT-CBS `main()` in same simplified version as DPLL(MAPF) `main()`, to see comparison of these two implementations. The main difference is that in DPLL(MAPF) functions `check_collisions()` (listing 3.2, line 10) and `add_clauses()`(listing 3.2, line 14) are called inside `Solver::solve()` and `check_collisions()` is adjusted to check the incomplete assignment. These two functions are represented on diagram 2.4 as *Add conflict* and both *Is consistent*.

Before we continue, let us say that `vector<int>` is good representation of clause. Example: $(\neg x \vee \neg y \vee z) \rightarrow$ [-1,-2,3]
And `vector<vector<int>>` (listing 3.2, line 9) can represent CNF formula as a vector of clauses.

In the next sections we will explain classes, their methods, and more used in DPLL(MAPF) `main()`.

3.1 reLOC

Our program accepts file input in .cpf format. Because we used **reLOC** [39] for generating MAPF problems/instances.

reLOC is set of programs focused on MAPF and one of them can generate MAPF instances. It does that by taking map in .map file format (see section

4.1) and it generates and saves MAPF instance in .cpf format with specified number of agents. It produces only solvable problems.

3.2 CPF format

The CPF file format represents a graph with agents. The first line of this file is always "V =". Next lines define vertices. Vertex is represented as a pair of numbers $(v : -1)$, where v in number of vertex from 0 to $\#V - 1$ and the second number -1 is unused in our case. Each vertex continuous with the set of 3 numbers, these numbers represent the following:

1. number representing agent's starting position
2. number representing agent's destination
3. number is unused and will be the same as the previous number.

All agents are numbered from 1 , where 0 marks that there is no agent's starting nor goal position.

After vertices, the edges are defined starting with the line "E =". Each edge is represented as a pair of numbers $\{x, y\}$, meaning an edge from vertex x to vertex y. Each edge has it's weight, -1 if the graph is not weighted .

Listing 3.3: CPF format of simple MAPF problem on figure 1.11

```
V =
(0:-1)[1:2:2]
(1:-1)[0:0:0]
(2:-1)[0:1:1]
(3:-1)[2:0:0]
E =
{0,1} (-1)
{1,2} (-1)
{1,3} (-1)
```

3.3 encode_MAP

`encode_MAP` is class used by class `MAPF_handler` and it represents our encoding map. Such map was captured on figure 2.4 as Mapping.

Mapping is a bijection of vertices of MDDs (fig.1.10) to propositional variables.

It consist of `vector<encode_unit> map` and methods for accessing, creating, and printing variables. `encode_unit` is `struct` representing propositional variable. It consist of three variables `uint16_t time, agent, vertex`.

These variables are enough to represent specific vertex in MDD and propositional variable is represented by index in `map`. Furthermore `encode_unit` has implemented operators == and < for faster encoding.

The bijection looks like this: $[time, agent, vertex] \Leftrightarrow map\ index$

Written more like code:
propositional variable $x \Leftrightarrow [map[x].time, map[x].agent, map[x].vertex]$

3.4 MAPF_handler

`MAPF_handler` is class that takes care of part of MAPF theory. It loads the problem, does eager encoding, and creates encoding map.

It has 3 variables. `vector<vector<bool>> graph` and `vector<pair<uint16_t, uint16_t>> agents` are representation of MAPF instance and `encode_MAP map` is an encoding map.

There are 3 MAPF_handler's methods called from main():

1. Constructor `MAPF_handler(filename)` creates internal representation of the graph and agent's start and goal positions.

2. `get_shortest_path()` returns greatest of agent's shortest paths.

3. `encode(solver, makespan)` creates MDD for each agent. Then if some MDD has only one vertex v in some time t (excluding 0 and makespan) we know that in SAT solver this variable needs to be marked as true, because it is only possible place for agent to be in time t. So we can omit translating this variable and delete pair (v,t) from other agent's MDDs. This also leads to making less constraints. Finally it adds collisions to `Solver` and create an encoding map based on these MDDs.

In the listing 3.4 is part of `encode(solver, makespan)` that creates collisions and encoding map.

Listing 3.4: Creating collisions

```
1  vector<vector<set<uint16_t>>> MDDS; // MDDS[agent][time][vertex]
2  ... // code that creates MDDs
3  for (size_t t = 0; t < makespan; t++)
4  {
5    for (size_t a = 0; a < MDDS.size(); a++) // for each agent's MDD
6    {
7      map.add(t, a, MDDS[a][t]);
8      for (size_t i = 0; i < MDDS[a][t].size(); i++)
9        solver.newVar(); // Glucose internal method
10     solver.add_disallowing_pairs(map.encode_set(t, a, MDDS[a][t]));
11     solver.add_clauses(create_edges(t,a,MDDS[a][t-1],MDDS[a][t]));
12   }
13 }
```

map starts with empty map.map and it is being filled on line 7. map.add() gets one time slice of MDD (MDDS[a][t], we can image it as column of MDD on figure 1.10) and saves them in same order as new propositional variables are added into Solver by Solver::newVar() on line 9. This ensures compatibility between Solver's variables and encode_MAP map.

In section 1.6 we introduced 4 types of constrains - edges, agent can be in one vertex at one time, collisions, and swaps.

Constrain agent can be in one vertex at one time is encoded on line 10 by method Solver::add_disallowing_pairs(). We implemented this method and it is explained in the next section.

Edges are encoded on line 11. First method MAPF_handler::create_edges() is called, which creates CNF formula representing edge constraints and pass it to Solver::add_clauses(), which accepts and adds CNF formula to Solver's formula.

Collisions and swaps are not encoded directly in SMT-CBS and our DPLL(MAPF). In our DPLL(MAPF) we implemented these constrains into Solver.

3.5 Solver

As our SAT solver we used Glucose 4 SAT solver [40]. It has good performance, it is open source, written in C++ and quite readable. It is based on MiniSAT [41]. The main class in Glucose is Solver and we transformed it from SAT solver to MAPF solver.

We added 3 methods to Solver class:

- check_collisions(int size) - checks collisions and stores them in Solver.collisions. collisions are then added by add_clauses(). size means how many variables were assigned
- add_clauses(const vector<vector<int>> &v) - adds clauses to Solver's formula, uses Glucose internal methods.
- add_disallowing_pairs(const vector<int> &v) - creates clause $(\neg x \vee \neg y)$ for every pair in input vector, they all should be positive numbers. This method serve for easier adding clauses from method MAPF_handler::encode().

And changed method Solver::solve() from original (see listing 3.5) to look like listing 3.6.

Listing 3.5: Solver::solve() in original Glucose

```
1        budgetOff();
2        assumptions.clear();
3        return solve_() == l_True;
```

Listing 3.6: Solver::solve() in DPLL(MAPF)

```
1  lbool status = l_Reset;
2  while (status == l_Reset)
3  {
4      add_clauses(collisions);
5      collisions.clear();
6      budgetOff(); // Glucose internal method
7      status = solve_(); // Glucose method for SAT solving
8  }
9  return status == l_True; // l_True if SAT
```

We created `l_Reset` as new state for `lbool` so method `Solver::solve_()` can tell us if collisions was found during SAT solving. While cycle (line 2) keeps calling `Solver::solve_()` until there are no collisions.

Inside `solve_()` (line 7) we are calling `check_collisions()` (frequency is determined by checking parameter, see section 4.2). If collisions were found we exit `solve_()` with `l_Reset` status.

We did not manage to integrate our code with Glucose 4 on such level that DPLL(T) proposes. The Glucose 4 is very complex and it would take much more time to understand it completely.

We were not able to add clauses inside method `Solver::check_collisions()` or right after this method without jumping outside main loop of `solve_()` and when conflicts occur some reset needs to be done (optimally there would be no reset). Before `Solver::addClause_()` we have to call `Solver::cancelUntil(0)` and when we tried that inside main loop of `solve_()` it broke. Therefore, we don't know if a perfect implementation is possible with this SAT solver.

Nevertheless, the integration of our implementation is close enough to call it DPLL(MAPF).

Chapter 4

Experimental evaluation

Because our aim is to make a faster algorithm, we have to run some tests and evaluate them. Thus, the first step is the test preparation, which is split into two sections:

1. Creating instances for our evaluation – section 4.1.
2. Creating a set of checking parameters – section 4.2.

At the end, we present the results and its evaluation. The evaluation of the proposed strategy Eagerly adding of chokepoints (see section 2.4.2) is done in separate section 4.3.3.

In this chapter we used # as symbol for number, e.g. #agents means number of agents.

4.1 Graphs

We chose 3 graphs from site `movingai.com` [42] to create MAPF problems for our evaluation. All graphs are in .map format.

MAP format serves as a representation of graph as map. MAP file starts with type, in our case it will be "octile". Height and width are specified on the next two lines. On the 4th line is the keyword "map". After that, the map itself starts. The basic format that we are using has only two symbols:

- "`.`" represents a passable terrain.
- "`@`" represents an unpassable terrain/obstacle.

We created a simple example of MAP format on figure 4.1, where is also an image of such map.

On figures 4.2, 4.3 and 4.4 are images of the graphs with sizes in their names. Only Empty is without size, because we used this map twice with size 16x16 and 32x32. These images represent their .map format. Only the graph

```
type octile
height 4
width 7
map
....@@@
.@@.@..
..@....
@@@.@@@
```

Figure 4.1: MAP file format and image of such map

of the warehouse was shortened from 161x63 to 100x63, to ensure reasonable computational time of evaluation (the image is also edited).

We picked 3 completely different types of maps to ensure diversity for our tests and because our program is a general MAPF solver (it is not specified on one task, e.g. automated warehouse) and should perform well on all types of maps. Maps are quite small, this is because our program should perform well on smaller maps.

In the previous section, we mentioned that **reLOC** creates MAPF instances from .map file and specified number of agents. Basically it adds n agent's start and goal destinations on distinguishable passable terrain of map.

For each map, we created 14 instances with different number of agents from 2 to 30 by 2 (i.e. 2,4,6,...,30).

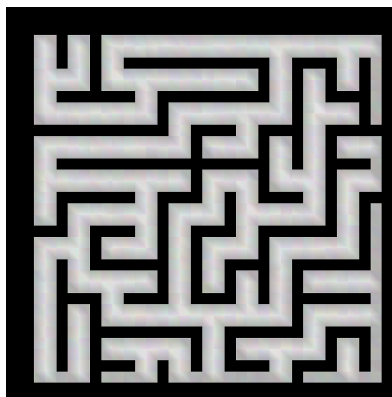


Figure 4.2: `maze-32-32`

The map Empty is a grid in graph theory.

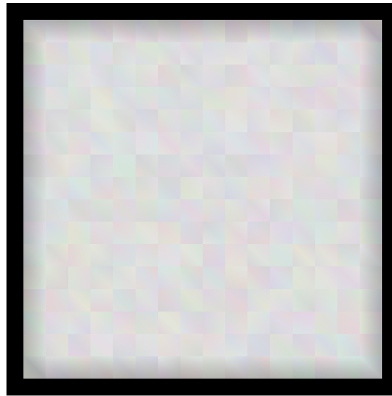


Figure 4.3: `empty`

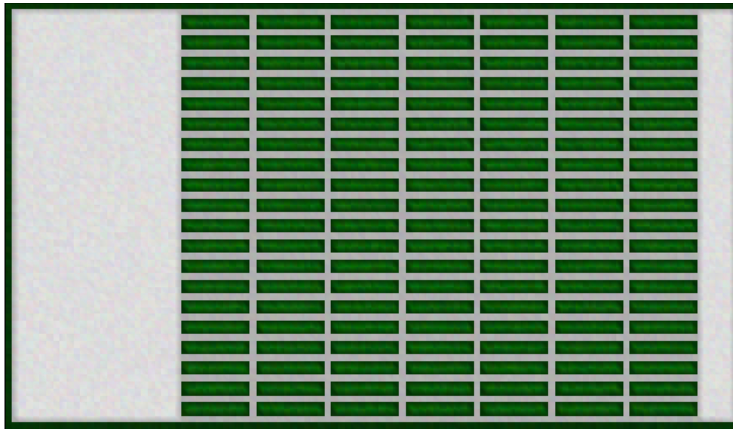


Figure 4.4: `Warehouse-100-63`

4.2 Checking parameter

During SAT solving we have to stop it and check if there are collisions in it's partial assignment. The question is when to stop it.

If we would follow the DPLL(T) diagram on figure 2.3 we should check collisions after each BCP, after each assigned variable. This would be too expensive, because we have to decode the assignment before checking and analyzing all possible conflicts.

What if we check collisions when half of the assignment is done and then at the end? That seems reasonable, but shouldn't we do checks more often, for example in 1/4, 1/2, 3/4 of assigned variables? From this thought, we created the uniform parameter.

4.2.1 Uniform

The task of SAT solver is to assign V variables. We suggest to check consistency after each $(\#V)/N$ variables are assigned, where N is our parameter. This means that check are done at most N times uniformly to the number of variables.

We suggest to try values {3, 5, 10} as uniform parameter in our testing.

4.2.2 Exponential

When lesser variables are assigned, then the encoding is cheaper and so is the consistency checking. We suggest to make more checks at the earlier phase of SAT solving and then less.

We suggest that the first check will be at some constant number of assigned variables and the next ones grow exponentially. The constant number was determined as 100 after a little experiment where we checked when collisions start to appear. Consistency check is done after $100 * N^x$ variables are assigned, where N is our parameter and x is number that start at 0 and increment after each check.

We suggest to try values {2, 1.6, 1.4} as exponential parameter.

4.2.3 Size of formula

We provide table 4.1 which captions the number of variables of some instances that we test. We can say that our instances will be translated to a formula

map	size	#agents	#variables
Empty	16x16	2	832
Empty	16x16	30	28 599
Empty	32x32	2	299
Empty	32x32	30	322 624
maze	32x32	2	2 838
maze	32x32	30	635 421
Warehouse	100x63	2	36 301
Warehouse	100x63	16	468 539

Table 4.1: Number of variables

with thousands of variables. We prepared table 4.2 that shows how many times the exponential parameter checks collisions, if no collisions are found. In this case, we were lucky and SMT-CBS would be faster, because it would not do any early checks of partial assignment.

Uniform parameter always does the same number of checks. However, the exponential number of checks grows with the number of variables.

parameter	1000	10 000	100 000
exp 2	5	8	11
exp 1.6	6	11	16
exp 1.4	8	15	22

Table 4.2: Number of checks without finding collision

This feature of the exponential parameter and the fact that most of the checks are in the early phase, thus cheap, lead us to the assumption that these parameters will achieve better results (or at least have a potential to do so).

4.2.4 Another concepts

If we imagine that we can check at any time and do as many checks as we want. This means that for one instance that is encoded to N number of variables we have $2^{N-1} - 1$ possible ways of checking. Because sum of all possible k combinations from N variables is $\sum_{k=1}^{N} \binom{N}{k} = 2^N - 1$ and we have to do check when final variable is assigned, thus the formula $2^{N-1} - 1$. By the way for 832 variables (which is our smallest instance) $2^{331} - 1 \approx 10^{250}$. Thus, trying all possible combinations is out of question because its merely impossibly computation.

We came up with two approaches, but more can be invented. We will see in the results how much will the performance vary with different parameters and if it is worth to work with the checking parameter.

Another idea is dynamically changing the checking parameter based on when and how many collisions were found, e.g. if we are using the parameter uniform 10 and the first collision is found on the 8th check and there is just 1 collision, so maybe next time we could check more lately. This dynamical checking seems promising, but it needs to be well explored. In this work, we focus on the experimental evaluation, which can tell us if DPLL(MAPF) is a good way of improving SMT-CBS. We think that our parameters will do the job.

Reminding the chosen parameters:

- Uniform – 3, 5, 10
- Exponential – 2, 1.6, 1.4

4.3 Results

4.3.1 Measurement description

Tests were done on a system with Core i7 CPU, 16 GB RAM, under Windows 10. Each instance was run 10 times and the presented results are averaged.

We will be calling maps from the previous section Empty 16, Empty 32, Maze and Warehouse. Only tests on Warehouses were shortened by the number of agents greater than 16, due to `Glucose::OutOfMemoryException` in larger instances and we decided to give up on these, because even if we managed to solve this exception, the runtime might be very large. We think that 8 instances of Warehouse are enough for our evaluation.

For testing, we wrote shell scripts which create .csv files filled with runtimes. Then we used python 3 with jupyter notebook[5] to average the results and for subtraction DPLL(MAPF) runtimes by SMT-CBS's to see the difference.

In the next sections, we will use letter T as shortcut for runtime.

4.3.2 Results and evaluation

In this section, we provide an evaluation of our implementation. Special strategy Eager chokepoints (section 2.4.2) will be discussed in the next section.

#agents	SMT-CBS	UNI 3	UNI 5	UNI 10	EXP 2	EXP 1.4	EXP 1.6
2	13.6	14.5	16.2	11.6	13.7	17.3	15.3
4	20.0	18.8	21.4	20.3	22.3	24.1	25.8
6	8.1	14.7	8.2	10.7	18.6	16.5	13.0
8	18.2	20.1	22.5	21.5	22.5	21.6	21.8
10	61.7	50.5	57.4	52.9	54.6	49.8	52.3
12	56.6	41.3	43.4	45.9	40.3	40.0	38.0
14	92.2	74.9	72.2	76.4	75.6	81.5	72.4
16	78.2	75.1	72.7	72.5	68.1	80.9	72.7
18	215.2	173.3	177.7	169.9	184.8	176.6	182.6
20	449.2	219.4	235.7	228.2	236.6	272.4	227.5
22	211.2	148.8	145.9	165.4	149.1	165.4	143.3
24	169.3	166.7	155.6	163.6	156.8	151.6	151.4
26	255.7	244.5	249.6	279.8	260.3	258.4	273.8
28	432.6	515.6	479.3	481.6	478.5	522.2	497.2
30	290.9	362.0	318.6	361.6	296.4	312.0	356.0
sum	2372.6	2140.1	2076.2	2162.0	2078.0	2190.5	2143.2

Table 4.3: Empty 16 runtime results in milliseconds

When we look at row *sum* in table 4.7, it is interesting that the uniform parameter's results are in order 10, 5, 3, meaning that fewer checks perform better. However the exponential parameters are sorted 2, 1.6, 1.4, which means more checks leads to better runtime. But this is only when we look at row *sum* and *Maze* others are slightly different.

[5]web application that allows you to interactively work with python

#agents	SMT-CBS	UNI 3	UNI 5	UNI 10	EXP 2	EXP 1.4	EXP 1.6
2	0.04	0.04	0.04	0.05	0.04	0.04	0.03
4	0.66	0.67	0.68	0.79	0.64	0.64	0.59
6	0.88	1.06	0.85	1.23	0.91	0.89	0.88
8	1.15	1.23	1.08	1.59	1.19	1.08	1.10
10	0.89	1.00	0.84	1.16	0.92	0.82	0.86
12	1.77	1.94	2.22	2.44	1.70	1.61	1.74
14	1.27	1.35	1.53	1.86	1.32	1.20	1.19
16	8.10	8.58	9.34	9.77	7.37	7.31	7.26
18	4.35	4.92	4.81	5.87	4.03	3.91	4.10
20	5.60	5.71	5.40	7.18	5.26	5.11	5.07
22	11.16	11.19	11.62	13.18	9.78	9.64	9.92
24	18.53	17.19	15.14	22.78	14.38	14.34	14.78
26	5.96	5.46	5.46	7.13	5.50	4.87	4.96
28	7.32	6.81	6.65	9.68	6.83	6.59	6.28
30	10.58	9.41	10.27	14.19	9.34	9.08	9.37
sum	78.25	76.55	75.92	98.88	69.21	67.13	68.12

Table 4.4: Empty 32 runtime results in seconds

#agents	SMT-CBS	UNI 3	UNI 5	UNI 10	EXP 2	EXP 1.4	EXP 1.6
2	0.17	0.22	0.41	0.19	0.19	0.19	0.17
4	0.15	0.15	0.17	0.15	0.13	0.14	0.14
6	0.90	0.95	1.14	1.08	0.86	0.83	0.80
8	1.01	0.96	1.25	1.11	1.06	0.97	0.91
10	0.99	0.97	1.10	1.09	1.02	0.88	0.91
12	1.07	1.07	1.45	1.36	1.23	1.06	1.04
14	2.66	2.69	3.24	3.80	3.12	2.59	2.51
16	3.77	3.40	4.63	5.62	3.70	2.85	2.98
18	6.39	6.70	8.82	8.52	7.99	6.46	6.74
20	10.84	10.54	20.18	30.64	10.41	12.04	10.99
22	5.78	7.99	10.38	17.40	8.47	7.82	8.44
24	11.40	12.44	13.71	36.39	11.98	9.45	14.43
26	9.63	15.68	12.29	32.56	11.77	11.62	9.69
28	14.97	20.66	18.16	68.12	27.88	19.32	16.67
30	27.72	29.26	49.14	84.27	36.00	25.97	29.86
sum	97.47	113.68	146.05	292.28	125.79	102.18	106.28

Table 4.5: Maze runtime results in seconds

#agents	SMT-CBS	UNI 3	UNI 5	UNI 10	EXP 2	EXP 1.4	EXP 1.6
2	6.05	6.03	5.88	6.00	6.08	5.71	5.62
4	4.88	4.92	4.79	4.80	4.86	4.74	4.51
6	8.22	8.20	8.12	7.75	7.93	7.55	7.85
8	27.54	27.03	26.91	27.74	28.31	26.33	26.56
10	61.09	48.74	44.95	51.29	56.36	51.40	51.08
12	49.24	50.92	47.94	48.31	55.98	49.71	49.43
14	57.63	65.01	51.22	50.83	62.61	52.60	51.20
16	33.98	42.29	35.36	38.32	44.15	33.64	34.38
sum	248.62	253.15	225.17	235.05	266.27	231.69	230.62

Table 4.6: Warehouse runtime results in seconds

	UNI 10	EXP 2	UNI 5	UNI 3	EXP 1.6	EXP 1.4
Empty 16	-0.21	-0.29	-0.30	-0.23	-0.18	-0.23
Empty 32	20.63	-9.05	-2.34	-1.71	-10.13	-11.12
Maze	194.82	28.32	48.58	16.21	8.81	4.72
Warehouse	-13.57	17.66	-23.44	4.53	-17.99	-16.92
sum	201.67	36.64	22.51	18.81	-19.49	-23.56
percentage	-32.09%	-7.91%	-5.01%	-4.22%	4.79%	5.84%

Table 4.7: Sums of $T_{DPLL(MAPF)} - T_{SMT-CBS}$ in seconds and percentage improvement counted as $(T_{DPLL(MAPF)}/T_{SMT-CBS}) - 1$. Columns are sorted by row sum. Negative numbers are highlighted.

Only in Maze DPLL(MAPF) was not able to outperform STM-CBS. Although in some instances of Maze some DPLL(MAPF) is doing better than SMT-CBS (e.g. #agents=30, SMT-CBS=27.7s, EXP 1.4=26s).

DPLL(MAPF) with uniform parameter 5 has bad performance on Maze (48 s slower than SMT-CBS), but does best in Warehouse (-23 s). This is very interesting, and it might be due to finding optimal parameter on Warehouse instances. If so, it means that checking parameter is very sensitive to different types of instances. Alike, the parameter UNI 10 has really bad runtime in Maze (194 s) but does pretty well in Warehouse (-13 s).

Over all we can say that DPLL(MAPF) runtime depends on checking parameter (improvement varies from -32.1% to 5.8%) and it differs on different MAPF instances. This means that we can fine-tune this parameter on specific instances or specific maps.

DPLL(MAPF)'s checking of partial assignment might save time or can be just a redundant computation. It depends on if there are mistakes in assignment. By this we explain the difference in the runtimes.

4.3.3 Eager chokepoints evaluation

We proposed different lazy encoding strategy in section 2.4.2, we call it Eager chokepoints. We changed our implementation of DPLL(MAPF) to use this encoding and we denote it as $DPLL(MAPF)_{Choke}$.

We ran the same test as in previous section. We will show the sumation of runtimes and comparison.

	UNI 3	UNI 5	UNI 10	EXP 2	EXP 1.4	EXP 1.6
Empty 16	0.11	0.06	0.23	0.14	0.15	0.26
Empty 32	-11.20	-8.97	-9.95	-9.15	-9.36	-7.71
Maze	137.45	142.30	162.24	128.25	137.13	119.43
Warehouse	-6.13	-11.32	-6.32	-12.21	-5.40	-2.36
sum	120.23	122.07	146.20	107.03	122.52	109.62
percentage	-21.98%	-22.24%	-25.52%	-20.05%	-22.31%	-20.44%

Table 4.8: Sums of $T_{DPLL(MAPF)_{Choke}} - T_{SMT-CBS}$ in seconds. Last two rows are sum of column and percentage improvement counted as $(T_{DPLL(MAPF)_{Choke}}/T_{SMT-CBS}) - 1$

In table 4.8 we can see in row *percentage* that this strategy is worse than SMT-CBS. The interesting thing is that all parameters have a very similar sum of runtimes, because DPLL(MAPF) results varied more, see table 4.7.

In table 4.9 we compared $DPLL(MAPF)_{Choke}$ to DPLL(MAPF).

	UNI 3	UNI 5	UNI 10	EXP 2	EXP 1.4	EXP 1.6
Empty 16	-0.35	-0.36	-0.44	-0.44	-0.33	-0.49
Empty 32	9.51	6.65	30.60	0.11	-1.76	-2.41
Maze	-121.24	-93.70	32.59	-99.91	-132.41	-110.62
Warehouse	10.65	-12.13	-7.26	29.87	-11.54	-15.63
sum	-101.43	-99.53	55.49	-70.36	-146.04	-129.15

Table 4.9: Sums of $T_{DPLL(MAPF)} - T_{DPLL(MAPF)_{Choke}}$ in seconds and sum of columns. Positive numbers are highlighted.

If we look at the rows, we can see that $DPLL(MAPF)_{Choke}$ does better in Empty 32 and Warehouse, comparable in Empty 16 and significantly worse in Maze. We think that in Maze, there were a lot of chokepoints to encode and it led to worse runtime.

We can see significant changes in row Maze. This is because in Maze there were more chokepoints than in other maps. However, this strategy improves some runetimes (highlighted). That means that we added eagerly some collisions before they were encountered and it saved some time. Therefore, the developing of a better lazy encoding strategy might lead to better performance.

However, it is very hard to determine which constraints should be treated eagerly, because as we can see in the results, if you unnecessarily eagerly encode some constrains it will slow the algorithm.

Overall, this encoding strategy is worse than one used in SMT-CBS. It only helps one parameter *UNI 10*.

Conclusion

The first chapter gives us the background for understanding the rest of our work.

In the second chapter, we provided an overview of lazy encoding followed by state-of-the-art SMT-CBS method. We then explained DPLL(T) and how we can look at SMT-CBS from DPLL(T) perspective. Eventually, the DPLL(MAPF) was introduced as MAPF solver using DPLL(T).

We implemented DPLL(MAPF) and described our implementation in section 3. During implementation we had to decide when to check collisions, so we created a checking parameter (see section 4.2).

Then we created and evaluated tests.

In Experimental evaluation we found out that DPLL(MAPF) is sensitive to checking parameter and it might be possible to fine-tune it. This option might be welcome for someone who is willing to spend time to fine-tune in order to get better results. On the other hand, DPLL(MAPF) becomes slightly more complicated to use than SMT-CBS, where you don't have to specify the checking parameter. Nevertheless, in our Experimental evaluation we tried 6 different parameters and 2 of them outperformed SMT-CBS, so finding a quite good parameter might be easy.

DPLL(MAPF) with checking parameter EXP 1.4 has 5.84% improvement compared to SMT-CBS in our benchmark.

We conclude that DPLL(MAPF) method is a favorable way of extending lazy encoding.

We also proposed Eager chokepoints (see section 2.4.2). The results showed this method to be worse than the one used in both SMT-CBS and our DPLL(MAPF) implementation.

Future works

We see further work in exploring the checking parameter. We mentioned that there are many ways of checking in section 4.2.4

We mentioned in section 3.5 that our implementation is almost perfect, but we can imagine even better synergy with SAT solver. Because of the complexity of the SAT solver and because it does not natively support the operations we need, we think that functions that we implemented might have been even better implemented. We think that especially the author of SAT solver can quite easily transform his own SAT solver into MAPF solver via DPLL(MAPF) and some manual how to do that might be created.

Bibliography

[1] Wurman, P. R.; D'Andrea, R.; et al. Coordinating Hundreds of Cooperative, Autonomous Vehicles in Warehouses. *AI Magazine*, volume 29, no. 1, Mar. 2008: p. 9, doi:10.1609/aimag.v29i1.2082. Available from: `https://ojs.aaai.org/index.php/aimagazine/article/view/2082`

[2] Ghallab, M. *Automated planning and acting*. New York, NY: Cambridge University Press, 2016, ISBN 9781107037274.

[3] Basile, F.; Chiacchio, P.; et al. A Hybrid Model of Complex Automated Warehouse Systems - Part I: Modeling and Simulation. *IEEE Trans. Automation Science and Engineering*, volume 9, no. 4, 2012: pp. 640–653.

[4] Wender, S.; Watson, I. D. Combining Case-Based Reasoning and Reinforcement Learning for Unit Navigation in Real-Time Strategy Game AI. In *ICCBR*, *LNCS*, volume 8765, Springer, 2014, pp. 511–525.

[5] Kim, D.-G.; Hirayama, K.; et al. Collision avoidance in multiple-ship situations by distributed local search. *Journal of Advanced Computational Intelligence and Intelligent Informatics*, volume 18, 09 2014: pp. 839–848.

[6] Hönig, W.; Kumar, T. K. S.; et al. Summary: Multi-Agent Path Finding with Kinematic Constraints. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017*, 2017, pp. 4869–4873.

[7] Li, J.; Surynek, P.; et al. Multi-Agent Path Finding for Large Agents. In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019*, AAAI Press, 2019, pp. 7627–7634.

[8] Ma, H.; Wagner, G.; et al. Multi-Agent Path Finding with Deadlines. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden.*, 2018, pp. 417–423.

[9] Kornhauser, D.; Miller, G. L.; et al. Coordinating Pebble Motion on Graphs, the Diameter of Permutation Groups, and Applications. In *25th Annual Symposium on Foundations of Computer Science, FOCS 1984*, IEEE Computer Society, 1984, pp. 241–250.

[10] Ryan, M. R. K. Exploiting Subgraph Structure in Multi-Robot Path Planning. *J. Artif. Intell. Res.*, volume 31, 2008: pp. 497–542.

[11] Silver, D. Cooperative Pathfinding. In *Proceedings of the First Artificial Intelligence and Interactive Digital Entertainment Conference, June 1-5, 2005, Marina del Rey, California, USA*, AAAI Press, 2005, pp. 117–122.

[12] Sharon, G.; Stern, R.; et al. The increasing cost tree search for optimal multi-agent pathfinding. *Artif. Intell.*, volume 195, 2013: pp. 470–495.

[13] Sharon, G.; Stern, R.; et al. Conflict-based search for optimal multi-agent pathfinding. *Artif. Intell.*, volume 219, 2015: pp. 40–66.

[14] Surynek, P. A novel approach to path planning for multiple robots in bi-connected graphs. In *2009 IEEE International Conference on Robotics and Automation, ICRA 2009, Kobe, Japan, May 12-17, 2009*, IEEE, 2009, pp. 3613–3619.

[15] Wang, K. C.; Botea, A. MAPP: a Scalable Multi-Agent Path Planning Algorithm with Tractability and Completeness Guarantees. *J. Artif. Intell. Res.*, volume 42, 2011: pp. 55–90.

[16] Luna, R.; Bekris, K. E. Push and Swap: Fast Cooperative Path-Finding with Completeness Guarantees. In *IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence*, IJCAI/AAAI, 2011, pp. 294–300.

[17] de Wilde, B.; ter Mors, A.; et al. Push and rotate: cooperative multi-agent path planning. In *International conference on Autonomous Agents and Multi-Agent Systems, AAMAS 2013*, IFAAMAS, 2013, pp. 87–94.

[18] Ratner, D.; Warmuth, M. K. Finding a Shortest Solution for the N x N Extension of the 15-PUZZLE Is Intractable. In *Proceedings of the 5th National Conference on Artificial Intelligence, AAAI 1986*, Morgan Kaufmann, 1986, pp. 168–172.

[19] Surynek, P. Reduced Time-Expansion Graphs and Goal Decomposition for Solving Cooperative Path Finding Sub-Optimally. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, edited by Q. Yang; M. J. Wooldridge, AAAI Press, 2015, ISBN 978-1-57735-738-4, pp. 1916–1922. Available from: `http://ijcai.org/Abstract/15/272`

[20] Barer, M.; Sharon, G.; et al. Suboptimal Variants of the Conflict-Based Search Algorithm for the Multi-Agent Pathfinding Problem. In *SOCS*, 2014.

[21] J. Yu, S. M. L. Planning optimal paths for multiple robots on graphs. *IEEE International Conference on Robotics and Automation*, 2013: pp. 3612–3617.

[22] Bogatarkan, A.; Erdem, E. Explanation Generation for Multi-Modal Multi-Agent Path Finding with Optimal Resource Utilization using Answer Set Programming. *Theory Pract. Log. Program.*, volume 20, no. 6, 2020: pp. 974–989.

[23] Ryan, M. Constraint-Based Multi-agent Path Planning. In *AI 2008: Advances in Artificial Intelligence, 21st Australasian Joint Conference on Artificial Intelligence, Proceedings*, *Lecture Notes in Computer Science*, volume 5360, Springer, 2008, pp. 116–127.

[24] Surynek, P. Time-expanded graph-based propositional encodings for makespan-optimal solving of cooperative path finding problems. *Ann. Math. Artif. Intell.*, volume 81, no. 3-4, 2017: pp. 329–375.

[25] Wikipedia contributors. Cook–Levin theorem — Wikipedia, The Free Encyclopedia. `https://en.wikipedia.org/w/index.php?title=Cook%E2%80%93Levin_theorem&oldid=994190506`, 2020, [Online; accessed 2-March-2021].

[26] Wikipedia contributors. Action language — Wikipedia, The Free Encyclopedia. `https://en.wikipedia.org/w/index.php?title=Action_language&oldid=968002360`, 2020, [Online; accessed 2-March-2021].

[27] Bylander, T. The Computational Complexity of Propositional STRIPS Planning. *Artificial Intelligence*, volume 69, 1994: pp. 165–204.

[28] Kautz, H. A.; Selman, B. Planning as Satisfiability. In *ECAI*, edited by B. Neumann, John Wiley and Sons, 1992, ISBN 9780471936084, pp. 359–363. Available from: `http://dblp.uni-trier.de/db/conf/ecai/ecai92.html#KautzS92`

[29] Surynek, P.; Felner, A.; et al. Efficient SAT Approach to Multi-Agent Path Finding Under the Sum of Costs Objective. In *ECAI 2016 - 22nd European Conference on Artificial Intelligence*, *Frontiers in Artificial Intelligence and Applications*, volume 285, IOS Press, 2016, pp. 810–818.

[30] DeWilde, B.; Mors, A.; et al. Push and Rotate: a Complete Multi-agent Pathfinding Algorithm. *Journal of Artificial Intelligence Research*, volume 51, 10 2014: pp. 443–492, doi:10.1613/jair.4447.

[31] Lam, E.; Bodic, P. L.; et al. Branch-and-Cut-and-Price for Multi-Agent Pathfinding. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019*, ijcai.org, 2019, pp. 1289–1296.

[32] Biere, A.; Biere, A.; et al. *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009, ISBN 1586039296, 9781586039295.

[33] Surynek, P. Unifying Search-based and Compilation-based Approaches to Multi-agent Path Finding through Satisfiability Modulo Theories. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019*, ijcai.org, 2019, pp. 1177–1183.

[34] Nieuwenhuis, R.; Oliveras, A.; et al. Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *J. ACM*, volume 53, no. 6, 2006: pp. 937–977, doi: 10.1145/1217856.1217859. Available from: `https://doi.org/10.1145/1217856.1217859`

[35] Katz, G.; Barrett, C. W.; et al. Lazy proofs for DPLL(T)-based SMT solvers. In *2016 Formal Methods in Computer-Aided Design, FMCAD 2016*, IEEE, 2016, pp. 93–100.

[36] Kroening, D.; Strichman, O. *Decision Procedures - An Algorithmic Point of View, Second Edition*. Texts in Theoretical Computer Science. An EATCS Series, Springer, 2016.

[37] Silva, J. P. M.; Sakallah, K. A. GRASP - a new search algorithm for satisfiability. In *Proceedings of the 1996 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 1996*, IEEE Computer Society / ACM, 1996, pp. 220–227.

[38] Silva, J. P. M.; Sakallah, K. A. Conflict Analysis in Search Algorithms for Satisfiability. In *Eigth International Conference on Tools with Artificial Intelligence, ICTAI 1996*, IEEE Computer Society, 1996, pp. 467–469.

[39] Surynek, P. Compact Representations of Cooperative Path-Finding as SAT Based on Matchings in Bipartite Graphs. In *2014 IEEE 26th International Conference on Tools with Artificial Intelligence (ICTAI)*, Los Alamitos, CA, USA: IEEE Computer Society, nov 2014, ISSN 1082-3409, pp. 875–882, doi:10.1109/ICTAI.2014.134. Available from: `https://doi.ieeecomputersociety.org/10.1109/ICTAI.2014.134`

[40] Audemard, G.; Simon, L. On the Glucose SAT Solver. *Int. J. Artif. Intell. Tools*, volume 27, no. 1, 2018: pp. 1840001:1–1840001:25.

[41] Een, N.; Mishchenko, A.; et al. A Single-Instance Incremental SAT Formulation of Proof- and Counterexample-Based Abstraction. 2010, 1008.2021.

[42] Sturtevant, N. R. Benchmarks for Grid-Based Pathfinding. *IEEE Trans. Comput. Intell. AI Games*, volume 4, no. 2, 2012: pp. 144–148.

Appendix A

Acronyms

MAPF Multi-agent pathfinding

SAT Boolean satisfiability problem

CDCL Conflict-driven clause learning

TEG Time expanded graph

MDD Multi-value Decision Diagram

CBS Conflict Based Search

SMT-CBS Satisfiability modulo theories conflict based search

Appendix B

Contents of enclosed CD

```
readme.txt ......................... the file with CD contents description
src ..................................... the directory of source codes
  dpll_mapf .......................... source codes of DPLL(MAPF)
  smt_cbs ................................ source codes of SMT-CBS
  measurement ....... the directory with results and jupyter notebooks
  thesis .............. the directory of LaTeX source codes of the thesis
text ......................................... the thesis text directory
  thesis.pdf ......................... the thesis text in PDF format
```