TÉCNICO LISBOA

Multi-Robot Hybrid Motion Planning Using Satisfiability Methods

Rui Pedro Caramujo de Sá

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

Supervisors: Prof. Rodrigo Martins de Matos Ventura
Prof. Mikoláš Janota

Examination Committee

Chairperson: Prof. João Fernando Cardoso Silva Sequeira
Supervisor: Prof. Rodrigo Martins de Matos Ventura
Member of the Committee: Dr. António José dos Reis Morgado

October 2021

Declaration

I declare that this document is an original work of my own authorship and that it fulfils all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Para a minha Mãe.

Acknowledgments

I would like to thank my family for always supporting and believing in me, without them I would not be where I am today.

I would also like to acknowledge my dissertation supervisors Prof. Rodrigo Ventura and Prof. Mikoláš Janota for their knowledge and support and for giving me this idea to explore.

Also, to my friends who were there when I needed them most, a most sincere thank you.

Last, but certainly not least, I would like to particularly thank my mother, for raising me to be the person I am today, for always being proud of me, and for always having some good advice at hand.

Abstract

Automation is an ever-growing need, especially when it comes to space exploration. In this thesis, the problem of planning for the stowage of cargo by a fleet of robots, in micro-gravity conditions, is tackled. This requires working with a hybrid and non-linear environment, since deliberation on both discrete and continuous-time characteristics of the system have to be made. Most techniques focus on the discretization of the continuous part, or the separation of both discrete and continuous and are not capable of handling non-linearity. The methods presented here consider all of them at the same time by using hybrid automata to describe the system and then, dReach ([1]), a tool which encodes the hybrid δ-reachability problem into a logical language, so that a Boolean Satisfiability (SAT), more specifically SAT Modulo Theory (SMT) solver, dReal ([2]), can find a model which validates it (a solution). dReal is the tool which enables taking into account the non-linearity of the system. This work, inspired by the advancements in SMT technology and tools which widen the range of problems they can handle, serves as a proof of concept, showing the potential for the use of the aforementioned techs and tools to solve these types of non-linear hybrid problems, which are historically very complex and hard to solve by conventional planning methods.

Keywords

Hybrid Planning; Non-Linear Hybrid Systems; Satisfiability Modulo Theories; δ-reachability; Hybrid Automata; dReach

Resumo

A automatização de processos é uma necessidade cada vez mais presente na sociedade, principalmente no que toca a exploração espacial. Nesta dissertação são apresentados métodos que permitem o planeamento de uma frota de robôs para a realização de descargas (incluindo arrumação) de carregamentos em micro-gravidade. Para tal, é necessário considerar um modelo híbrido não-linear para o sistema, visto que este apresenta características discretas e continuas em relação ao tempo. Enquanto que a maioria das técnicas existentes se focam em métodos de discretização ou na separação das deliberações continuas e discretas, o trabalho aqui apresentado utiliza autómatos híbridos para representar o sistema, recorrendo posteriormente a uma ferramenta, dReach, que codifica o problema de δ-*reachability* correspondente em linguagem lógica de modo a que um *solver*, dReal, consiga encontrar um modelo que valide (solucione) o problema. Assim, a deliberação ocorre tendo em conta ambas as propriedades continuas e discretas do sistema ao mesmo tempo, diferenciando-se das outras técnicas. O uso da ferramenta e *solver* dReach e dReal permite a consideração das não-linearidades do sistema. Os métodos aqui retratados, inspirados pelos mais recentes avanços nas tecnologias de SMT e também nas ferramentas que aumentam o espectro de problemas que estas podem resolver, servem como prova de conceito evidenciando o potencial destas técnicas para a resolução de problemas de planeamento híbrido em sistemas não lineares, os quais são, historicamente, extremamente complexos de resolver por meios convencionais de planeamento automático.

Palavras Chave

Planeamento Híbrido; Sistemas Híbridos Não-Lineares; *Satisfiablity Modulo Theories*; δ-*reachability*; Autómatos Híbridos; dReach

Contents

List of Figures

List of Tables

List of Algorithms

Acronyms

AI	Artificial Intelligence
DPLL	Davis–Putnam–Logemann–Loveland algorithm
HTN	Hierarchical Task Network
ICP	Interval Constraint Propagation
IVP	Initial Value Problem
ODE	Ordinary Differential Equation
SAHTN	State-Abstracted Hierarchical Task Network
SAT	Boolean Satisfiability
sat	Satisafiable
SMT	SAT Modulo Theory
TMP	Task and Motion Planning
unsat	Unsatisfiable

1

Introduction

Space exploration has been a human interest since we first looked up. However, for humans to be able to survive in the vast emptiness that is the universe, we need some infrastructure to sustain a habitable environment. A question is posed, who will build and maintain those infrastructures?

The answer may lie in robotics. For that, some level of intelligence and autonomy is needed. In this thesis, a method for motion planning for the transportation of cargo is presented, using hybrid planning in combination with satisfiability methods.

The scenario considered, for the problem this thesis is trying to solve, is a space station, where a fleet of robots is tasked with the unloading of supplies and other packages that arrive. The fleet should take the cargo from its initial position to a final one, previously designated and, in some assumed way (bar codes for each piece of cargo, which, when scanned, give this information, for example), communicated to the robots. The only available information is the map of the station, the previously mentioned positions and the maximum number of robots available.

Now, it is important to take into account that these operations will be subject to micro or even zero-gravity conditions. As such, part of the process of solving this problem was to decide how the robots would transport the cargo. This was actually the first decision that was made in the effort to find a solution.

Two options were considered:

- each robot attaches itself to the cargo and uses its propellers to guide the movement, as in Figure 1.1;
- each robot attaches itself to the cargo and an anchor point in the wall and pulls the cargo towards that anchor point, as in Figure 1.2.

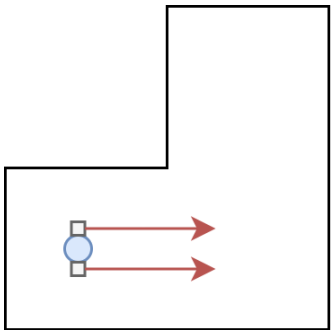

Figure 1.1: Example of mode of transportation where the robots' (squares in the figure) propellers are used to move the cargo (circle)

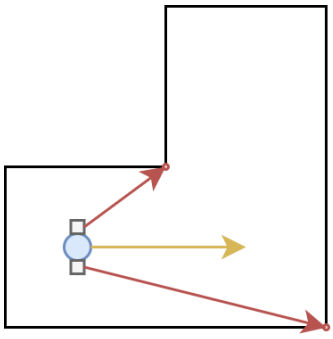

Figure 1.2: Example of mode of transportation where anchor points (red points in the Figure) are used by the robots (squares) to move the cargo (circle)

Due to simplicity of operation and control, the second option was the one considered as means of transportation.

In fact, the robot may not even need to attach itself to the anchor point. In theory, the force exerted by the recalling of the grappling hook will have the same direction and orientation as the one exerted by the robot if it were to pull the cargo while moving towards an anchor point, only differing in magnitude (which is directly controllable). As such this formulation can be directly used in a scenario where a fleet of robots attaches themselves to the cargo and each of them then travels to a specific anchor point, using methods from control theory (said anchor point would be the reference goal position for the robot, as such the system would always force the robot to move toward it). As a note, consider a cargo being transported by a fleet of three robots, r_1, r_2 and r_3, and a plan in which there exists a mode q_i. If mode q_i uses less than three anchor points, then, the number of robots which do not have an available anchor point should stop all actions until the plan reaches a stage in which they are needed again, but continue to be attached to the cargo, such as to make the transitions between modes as fast and seamless as possible.

With this in mind, the robot should be able to plan which anchor points to connect to, in what order and how much time it should remain connected to that point.

Since finding a solution to this problem involves dealing with continuous change to the cargo's coordinates, as well as discrete change of the anchor points a robot is connected to, a hybrid model is needed, which considers both types of changes.

After creating a model, a satisfiability modulo theory solver is used to create a plan.

In Chapter 2, a theoretical background is given, in which every previous knowledge needed in order to understand the methods developed is given, it gives a brief overview of the current landscape for solving problems of the same type, while the approach developed in this Thesis is explained in full detail in Chapter 3. Some results are analysed in Chapter 4. Finally, Chapter 5 provides a few concluding remarks as well as some suggested future work.

2

Background and Related Work

Contents

This chapter explains the planning and modelling tools used in solving the motion planning problem. Then, it gives a brief overview of the other techniques used to tackle similar issues.

2.1 Boolean Satisfiability Problems and SMT

In order to obtain a plan, a model of the system is encoded as a Boolean Satisfiability (SAT) and then a solver is used to find the solution to the SAT problem. A plan is, then, obtained by decoding that solution.

2.1.1 SAT

In propositional logic the zeroth elements are the ***literals***. A literal is a Boolean variable, x, in the true case, and its negation, $\neg x$, in the false.

The disjunction of literals makes a ***clause***, e.g. $x \vee y$, and a conjunction of clauses compounds a ***formula***, e.g. $f = (x \vee y) \wedge (\neg x \vee z \vee k)$

A Boolean satisfiability problem is the problem of deciding whether a certain formula, f, has a satisfying ***interpretation***.

An interpretation is a model which gives value to the Boolean variables in a formula. For instance, the interpretation $m = x \wedge \neg y \wedge z \wedge k$ validates the formula f, since its value assignment validates both clauses that compound f.

2.1.2 SAT Modulo Theory (SMT)

In a lot of Artificial Intelligence (AI) applications, a more expressive language is often needed, one that allows to check satisfiability to some background theory. Sometimes a theory cannot even be fully captured by propositional logic.

Let there be a first-order ***signature***, describing all non-logic symbols of first-order language, $\mathcal{L}$, and a theory $\mathcal{T}$. $\mathcal{L}$ extends the formulas used in conventional SAT problems adding the ability to encode more types of conditions(these are called $\mathcal{L}$-formulas), the theory $\mathcal{T}$ is used to verify each interpretation.

SAT modulo theory (SMT) is the name of the field which is concerned with the satisfiability of $\mathcal{L}$-formulas with respect to $\mathcal{T}$, i.e., an SMT problem tries to answer the question of whether there is an interpretation of the $\mathcal{L}$-formulas that is true in $\mathcal{T}$, being Satisafiable (sat) in this case and Unsatisfiable (unsat) otherwise.

An SMT solver is a technology which can give a solution to an SMT problem, it outputs whether a given set of formulas is sat or unsat. The solver used here on out is named ***dReal***.

2.2 dReal

dReal [2] is a solver that uses δ***-complete decision procedures***, that is, it uses decision making algorithms which can correctly solve δ***-SMT problems***.

Given a formula, φ, and any positive rational number, δ, a δ-SMT problem tries to decide upon two possible choices. Either φ is unsat, or its δ***-weakening***, the numerical relaxation of φ according to δ (for instance, the δ-relaxation of $x = 0$ is $|x| \leq \delta$), is δ-sat. This relaxation has significant effects on the decidability of a formula. For instance, an SMT formula extending real arithmetic by sine becomes undecidable, when its δ-SMT counterpart is decidable even when considering theories with exponentiation, trigonometric functions and, Lipschitz-continuous ordinary differential equations.

In the context of a planning problem, when dReal evaluates a formula encoding the problem as `unsat`, then there is no solution. When it is δ-`sat`, it is possible for dReal to provide a "witness", that is, a set of values for each variable which can be used as a valid solution to the problem, thus, a plan can be derived from such "witnesses".

2.3 Hybrid Model

The context of this problem forces the use of a model which takes into account both discrete changes, when the robot changes the anchor point it is connected to, for instance, and continuous changes, like the evolving position of the cargo through the path from the initial to the final point. This type of model is called a ***Hybrid Model***.

This model integrates a discrete state space in which the continuous changes are represented for each state by continuous variables, and changes to the discrete state variables correspond to the discontinuous changes of the problem.

A hybrid model is formally described by the ***Hybrid Automaton***

2.3.1 Hybrid Automaton

According to [3], a **hybrid automaton** is a structure which contains

- $\mathbf{X} = \{x_1, x_2, ..., x_n\}$, a finite set of continuous variables;
- and Q, a finite directed graph, called ***Control Graph***, with each node being called a ***Control Mode*** and each edge a ***Control Switch***.

Each **control mode**, q, is composed of

- a set of invariant conditions, ***inv***$_q$, over the variables of $\mathbf{X}$, which must be true when in mode q;

- a set of flow conditions, ***flow***$_q$, in the form of differential equations over the variables of $\mathbf{X}$ and their first derivatives, $\dot{\mathbf{X}}$, which represent the dynamics of each variable in $\mathbf{X}$ when in mode q;

- and a set of initial values of $\mathbf{X}$ in q, ***init***$_q$.

Each **control switch**, jump$_{q\to q'}$, is associated to a pair of control modes, q and q', and a set of jump conditions over the variables of X, which, when true, allow switching from q to q'.

Note that each jump$_{q\to q'} \neq$ jump$_{q'\to q}$, that is, the jump condition from mode q to mode q' may differ from its reciprocate, from q' to q. Additionally, the existence of jump$_{q\to q'}$ does not imply the same of jump$_{q'\to q}$.

All in all, a hybrid automaton can be defined as a structure

$$H = \langle \mathbf{X}, Q, \text{init}, \text{inv}, \text{flow}, \text{jump} \rangle \tag{2.1}$$

where init, inv and flow contain init$_q$, inv$_q$ and flow$_q$ for each mode q and jump contains every jump$_{q\to q'}$ from mode q to mode q'.

According to this definition, discrete change is modeled by the control switches of the automaton, that is, each control switch corresponds to a discrete change in the system, and continuous change is modeled by the flow conditions associated to each control mode.

With this tool, one can represent a problem as an automaton, where the control graph corresponds to the state-space, which represents every possible discrete scenario of the problem, with respect to the possible values and combinations between each discrete state variable, and within each control mode (state) the continuous variables evolve according to certain conditions.

2.3.2 $\mathcal{L}_{\mathcal{F}}$-Representation of Hybrid Automata

$\mathcal{L}_{\mathcal{F}}$ is the first-order signature of the set $\mathcal{F}$, of Type 2 computable functions, over $\mathbb{R}$. In [4] a representation of hybrid automata in $\mathcal{L}_{\mathcal{F}}$ is defined.

The work in [4] also gives the definition of bounded $\mathcal{L}_{\mathcal{F}}$-formulae ($\Sigma_1$-sentences in $\mathcal{L}_{\mathcal{F}}$). It is defined as

$$\varphi : Q^{I_1} x_1 \ldots Q^{I_n} x_n . \psi(x_1, \ldots, x_n) \tag{2.2}$$

where Q can be either existential or universal quantifiers ($\exists$ and $\forall$). This definition means that ψ is true for at least one or all (depending on the used quantifier) x_i, $1 \leq i \leq n$. For instance, the sentence $\forall^{I_i} x_i . \phi$, in which ϕ is any applicable logic formula, corresponds to $\forall x_i . (x_i \in I_i \implies \phi)$.

A hybrid automaton, such as the one presented before, in $\mathcal{L}_{\mathcal{F}}$-representation is

$$H = \langle \mathbf{X}, Q, \{flow_q(\mathbf{x}, \mathbf{y}, t) : q \in Q\}, \{inv_q(\mathbf{x}) : q \in Q\}, \\ \{jump_{q \to q'}(\mathbf{x}, \mathbf{y}) : q, q' \in Q\}, \{init_q(\mathbf{x}, \mathbf{y}, t) : q \in Q\}\rangle \quad (2.3)$$

in which $\mathbf{X}$ and Q are as before and the remaining elements are finite sets of quantifier-free $\mathcal{L}_{\mathcal{F}}$-formulas. For any hybrid automaton, H, $X(H)$, $Q(H)$, flow(H), inv(H), jump(H) and init(H) denote its components.

2.3.3 Hybrid Time Domains and Trajectories

Consider $m \in \mathbb{N} \cup \{+\infty\}$, an increasing sequence in $\mathbb{R}^+$, $\{t_i\}_{i=0}^m$, with $t_0 = 0$ and $t'_i = t_{i+1}$. A hybrid time domain is a subset of $\mathbb{N} \times \mathbb{R}$ of the form

$$T_m = \{(i, t) : i < m \text{ and } t \in [t_i, t'_i] \text{ or } [t_i, +\infty)\}. \quad (2.4)$$

Given an Euclidean space $X \subseteq \mathbb{R}^n$ and a hybrid time domain T_m, a hybrid trajectory is a continuous mapping $\xi : T_m \to X$, its time domain ($T_m$of ξ) can be written as $T(\xi)$. Note that the hybrid time domain moves in two ways, i changes discretely, these are referred to as steps, while t changes continuously and only increases within a step, returning to zero when i changes. In this problem's context, i will be the automaton's modes' ids and t will be the time spent in each mode.

Linking hybrid automata and trajectories requires the use of a labelling function $\sigma_{\xi,H}(i)$, which maps to each trajectory step i an appropriate discrete mode in H, also making sure every corresponding condition is satisfied. The trajectory of hybrid automata are, thus, defined by the existence of such labelling function.

For a hybrid domain T_m and hybrid automaton H, a hybrid trajectory $\xi : T_m \to \mathbf{X}(H)$ is said to be a trajectory of H of discrete depth m, or $\xi \in [\![H]\!]$ ($[\![H]\!]$ being the trajectory space associated to H), if there exists $\sigma_{\xi,H}(i) : \mathbb{N} \to Q(H)$ such that:

- For some $q \in Q(H)$, $\sigma_{\xi,H}(0) = q$ and $\mathbb{R}_{\mathcal{F}} \vDash init_q(\xi(0, 0))$, this guarantees the initial conditions of the first mode in the trajectory are satisfied;
- For any $(i, t) \in T_m$, $\mathbb{R}_{\mathcal{F}} \vDash inv_{q_i}(\xi(i, t))$, where $q_i = \sigma_{\xi,H}(i)$, this guarantees the invariant conditions are satisfied;
- For any $(0, t) \in T_m$, $\mathbb{R}_{\mathcal{F}} \vDash flow_{q_0}(\xi(0, 0), \xi(0, t), t)$, where $q_0 = \sigma_{\xi,H}(0)$, this guarantees the flow conditions of the first mode in the trajectory are satisfied;

- For any $i = k + 1$, where $0 < k + 1 < m$, $\mathbb{R}_{\mathcal{F}} \vDash \mathsf{flow}_{q_{k+1}}(\xi(k+1, t_{k+1}), \xi(k+1, t), (t - t_{k+1}))$, and $\mathbb{R}_{\mathcal{F}} \vDash \mathsf{jump}_{q_k \to g_{k+1}}(\xi(k, t'_k), \xi(k+1, t_{k+1}))$ where $q_{k+1} = \sigma_{\xi,H}(k+1)$ and $q_k = \sigma_{\xi,H}(k)$, guaranteeing the flow conditions of mode $k+1$ in the trajectory are satisfied, as well as the jump conditions from mode k to mode $k+1$.

From the definition one can derive that in each mode the dynamics of the continuous variables behave according to the ones defined in flow_q. Also, the actual duration of the k-th mode is $(t - t_k)$ and switching between two modes, $q \to q'$ requires the update of $\xi(k+1, t_{k+1})$ (the initial value of the continuous variables in q') to the exit value $\xi(k, t'_k)$ (the final value of the continuous variables in q) following the jump conditions.

It is very important for the understanding of trajectories for hybrid automaton that there is a vital difference between jump and inv conditions. The latter, when broken, specify when H ***must*** switch to another mode, the former, when valid, indicate whether H ***may*** switch to another mode, hence these require different logical encodings.

The $\mathcal{L}_{\mathcal{F}}$-representation has no more restrictions on the formulas that can be used for its description other that that the flow predicates should define continuous trajectories over time. More formally, a ***well-defined flow predicate*** is such that, for all tuples $(\mathbf{a}, \mathbf{b}, \tau) \in X(H) \times X(H) \times \mathbb{R}^{\geq 0}$ for which $\mathbb{R} \vDash \mathsf{flow}(\mathbf{a}, \mathbf{b}, \tau)$, there exists a continuous function $\eta : [0, \tau] \to X$ such that $\eta(0) = \mathbf{a}$, $\eta(\tau) = \mathbf{b}$ and for all $t' \in [0, \tau]$, $\mathbb{R} \vDash \mathsf{flow}(\mathbf{a}, \eta(t), t)$. H is said to be well-defined if every one of its flow predicates are also well-defined.

Flows that are defined using differential equations, differential inclusions and explicit continuous mapping all satisfy this condition, thus, the remaining discussion of hybrid automata assume their well-definedness.

2.3.4 δ-weakening of Hybrid Automata

For any non-negative rational number δ, The δ-weakening of a $\mathcal{L}_{\mathcal{F}}$-representation of hybrid automaton H, defined as in 2.3.2, is obtained by δ-weakening every formula in H, introducing δ-relaxations in H's formulas. It is written as

$$H^{\delta} = \langle \mathbf{X}, Q, \mathsf{flow}^{\delta}, \mathsf{jump}^{\delta}, \mathsf{inv}^{\delta}, \mathsf{init}^{\delta} \rangle. \tag{2.5}$$

It is worth noting that δ-relaxations are a purely syntactic notion, as they are defined on the description of H, instead of a semantic one, defined on the trajectories. The former corresponding to over-approximations of H in the trajectory space, $[\![H]\!]$. Also, for any H and $\delta \in \mathbb{Q}^{+} \cup \{0\}$, $[\![H]\!] \subseteq [\![H^{\delta}]\!]$.

2.3.5 Reachability in Hybrid systems

Consider an n-dimensional hybrid automaton H and a subset of its state space $U \subseteq Q(H) \times \mathbf{X}(H)$. If there is a trajectory $\xi \in [\![H]\!]$ such that $\exists (i,t) \in T(\xi)$ satisfying $(\sigma_{\xi,H}(i), \xi(i,t)) \in U$ (for each step i in ξ, there should be a label $\sigma_{\xi,H}(i)$ which maps that step into the appropriate discrete mode in H), then it is said that U is reachable by H.

When restricting the continuous time duration to a bounded interval and the number of discrete switches to a finite number, one gets a ***bounded reachability problem for hybrid systems***. Considering, now, that $X(H)$ is a bounded subset of $\mathbb{R}^n$ and there is some $k \in \mathbb{N}$ and $M \in \mathbb{R}^{\geq 0}$, the (k, M)-bounded reachability problem asks for the same requirements as the unbounded version, with the addition of the conditions that $i \leq k$ and $t = \sum_{i=0}^{k} t_i$ where $t_i \leq M$, i.e., the trajectory should have, at most, k steps and its total duration (the sum of the duration of all the steps) should be no more than M.

2.4 Planning as Model Checking

In the case of reachability goals, one can use pre-existing hybrid model checkers to solve a planning problem.

For this, dReach [1] is used. It is a tool for bounded reachability analysis of nonlinear hybrid systems.

2.4.1 dReach

dReach solves bounded reachability problems of hybrid systems by encoding them as first-order logic formulas over the real numbers. It, then, uses an SMT solver, dReal (Section 2.2), to find a solution to the resulting formula. Specifically, it provides a δ-complete reachability analysis, which reduces model checking problems to δ-decision problems of formulas over the reals.

Standard bounded reachability problems for simple hybrid systems are highly undecidable [5], thus, dReach decides the δ-reachability of hybrid systems instead. This means the analysis of the hybrid system tolerates numerical errors within the bounds specified by the user via an arbitrary positive rational number, δ. For this, it considers the δ***-bounded overapproximations*** of both the hybrid system H and unsafe region `unsafe`

Bounded δ-reachability

Consider a hybrid system defined as in Section 2.3, $H = \langle X, Q, \text{flow}, \text{jump}, \text{inv}, \text{init} \rangle$, with Q being its control graph flow, jump, inv, init its flow, jump, invariant and initial conditions, respectively. Given a

chosen non-negative rational error bound, δ, the δ-perturbation of H is defined as

$$H^{\delta} = \langle X, Q, \varphi^{\delta}_{flow}, \varphi^{\delta}_{jump}, \varphi^{\delta}_{inv}, \varphi^{\delta}_{init} \rangle \tag{2.6}$$

where φ_{flow}, φ_{jump}, φ_{inv} and φ_{init} are the Σ_1-sentences that encode the conditions in flow, jump, inv and init, and $\varphi^{\delta}_{...}$ their corresponding δ-weakened formulae.

Now, let there be a bound in the number of steps of a plan, $n \in \mathbb{N}$, and an upper bound on time duration, $T \in \mathbb{R}^{+}$. `unsafe` is defined by a first-order formula and denotes a subset of the state space of H. A bounded δ-reachability problem reaches one the two conclusions:

- `safe` - H cannot reach `unsafe` in n steps within time T;
- δ-`unsafe` - H^{δ} can reach `unsafe`$^{\delta}$ in n steps and within time T, here `unsafe`$^{\delta}$ is the δ-weakening of `unsafe`.

Note that, when the tool reaches the first option, it is certain that H does not reach the unsafe region. In case of δ-`unsafe` as an answer, there exists a δ-bounded perturbation of the system which can render it unsafe. This last answer discovers robustness problems in the system, which should be regarded as unsafe.

If the `unsafe` region is made to be the goal region of the cargo, then, dReach is able to decide if that region (more specifically, its δ-weakening) is reachable by the hybrid automaton H when starting at a specific initial point, if so, the returned "witness" can be used to generate a motion plan from that initial point to the final region.

The framework defined in [4], where the decidability of bounded δ-reachability is proven to be possible for a wide range of nonlinear hybrid systems, provides the formal correctness guaranties of dReach.

δ-complete Analysis for Bounded Reachability

The first step of this analysis is to encode bounded reachability in $\mathcal{L}_{\mathcal{F}}$.

Let there be a Hybrid automaton H, $\texttt{unsafe} = \{\texttt{unsafe}_q : q \in Q(H)\}$ is the $\mathcal{L}_{\mathcal{F}}$-representation of an unsafe region in the state space of H. Additionally, $[\![\texttt{unsafe}]\!] = \bigcup_{q \in Q(H)} [\![\texttt{unsafe}_q]\!] \times \{q\}$.

As to ensure that a specific mode is selected at a particular step, a set of auxiliary formulas are needed. The use of which will later be explained when the full encodings of bounded reachability are defined.

Consider $Q(H) = \{q_1, \ldots, q_n\}$, a set of modes, then assign a Boolean variable for any $q \in Q$ and

$i \in \mathbb{N}$, written as b_q^i. It is now possible to define `enforce` formulas

$$\begin{aligned} \texttt{enforce}_Q(q,i) &= b_q^i \wedge \bigwedge_{p \in Q(H)\setminus\{q\}} \neg b_p^i \\ \texttt{enforce}_Q(q,q',i) &= b_q^i \wedge b_{q'}^{i+1} \wedge \bigwedge_{p \in Q(H)\setminus\{q\}} \neg b_p^i \wedge \bigwedge_{p' \in Q(H)\setminus\{q'\}} \neg b_{p'}^{i+1} \end{aligned} \tag{2.7}$$

The former represents the assignment of mode q to the i-th step, it assures only mode q is active in step i by forcing b_q^i and negating the Boolean variables of every other mode for step i. The latter is a representation of a control switch, it assures only mode q is active at step i and only mode q' is active at step $i+1$, it does this by replicating the firstly described `enforce` formula and duplicating it for both (q,i) and $(q',i+1)$. When context is clear, the subscript $_Q$ of the `enforce` formula is removed.

First, the encoding of a k***-step reachability problem*** for a hybrid automaton with no invariants, the simplest case, is explained.

A hybrid automaton H is said to be ***invariant-free*** if, for every $q \in Q(H)$, $inv_q(H) = T$ (True). The formula which checks whether an unsafe region is reachable in exactly k steps for this type of hybrid system is as follows

$$\begin{aligned} &\exists^{\mathbf{X}}\mathbf{x}_0 \exists^{\mathbf{X}}\mathbf{x}_0^t \ldots \exists^{\mathbf{X}}\mathbf{x}_k \exists^{\mathbf{X}}\mathbf{x}_k^t \exists^{[o,M]} t_0 \ldots \exists^{[o,M]} t_k. \\ &\bigvee_{q \in Q(H)} \big(\mathsf{init}_q(\mathbf{x}_0) \wedge \mathsf{flow}_q(\mathbf{x}_0, \mathbf{x}_0^t, t_0) \wedge \mathsf{enforce}(q,0)\big) \\ \wedge \bigwedge_{i=0}^{k-1} &\left(\bigvee_{q,q' \in Q(H)} \big(\mathsf{jump}_{q\to q'}(\mathbf{x}_i^t, \mathbf{x}_{i+1}) \wedge \mathsf{enforce}_q(q,q',i) \wedge \mathsf{flow}_{q'}(\mathbf{x}_{i+1}, \mathbf{x}_{i+1}^t, t_{i+1}) \wedge \mathsf{enforce}_{q'}(q', i+1)\big) \right) \\ &\wedge \bigvee_{q \in Q(H)} \mathsf{unsafe}_q(\mathbf{x}_k^t). \end{aligned} \tag{2.8}$$

This is the $\mathcal{L}_{\mathcal{F}}$-formula $\mathsf{Reach}_{H,U}(k,M)$, where U is a subset of the state space of H and it is represented by `unsafe` in the formula.

The trajectories begin with some initial mode, q in which $\mathsf{init}_q(\mathbf{x}_0)$ is satisfied. At each switch from q to q', it resets the continuous variables according to $\mathsf{jump}_{q\to q'}(\mathbf{x}_k^t, \mathbf{x}_{k+1})$ and $\mathsf{flow}_{q'}(\mathbf{x}_{i+1}, \mathbf{x}_{i+1}^t, t_{i+1})$ ensures that a continuous flow from $\mathbf{x}_{i+1}$ to $\mathbf{x}_{i+1}^t$ after time t is made according to the flow conditions of the $(i+1)$-th mode.

The next step is to ensure, in case of hybrid automata with invariant conditions, that those conditions hold. For this, there is a need to use universal quantifiers over time, because of this, allowing non-deterministic flows complicates the situation to a point where no solution was found by [4]. As such, the

following encoding is applicable to systems with non-trivial invariants and deterministic flow.

$$
\begin{aligned}
&\exists^{\mathbf{X}}\mathbf{x}_0 \exists^{\mathbf{X}}\mathbf{x}_0^t \ldots \exists^{\mathbf{X}}\mathbf{x}_k \exists^{\mathbf{X}}\mathbf{x}_k^t \exists^{[o,M]} t_0 \ldots \exists^{[o,M]} t_k. \\
&\bigvee_{q\in Q(H)} \Big(\mathsf{init}_q(\mathbf{x}_0 \wedge \mathsf{flow}_q(\mathbf{x}_0, \mathbf{x}_0^t, t_0) \wedge \mathsf{enforce}(q,0) \wedge \forall^{[0,t_0]} t \forall^{\mathbf{X}} \mathbf{x} (\mathsf{flow}_q(\mathbf{x}_0, \mathbf{x}, t) \implies \mathsf{inv}_q(\mathbf{x})) \Big) \\
&\quad \wedge \bigwedge_{i=0}^{k-1} \Big(\bigvee_{q,q'\in Q(H)} (\mathsf{jump}_{q\to q'}(\mathbf{x}_i^t, \mathbf{x}_{i+1}) \wedge \mathsf{enforce}_q(q,q',i) \wedge \mathsf{flow}_{q'}(\mathbf{x}_{i+1}, \mathbf{x}_{i+1}^t, t_{i+1}) \\
&\qquad \wedge \mathsf{enforce}(q', i+1) \wedge \forall^{[0,t_{i+1}]} t \forall^{\mathbf{X}} \mathbf{x} (\mathsf{flow}_{q'}(\mathbf{x}_{i+1}, \mathbf{x}, t) \implies \mathsf{inv}_{q'}(\mathbf{x}))) \Big) \\
&\quad \wedge \bigvee_{q\in Q(H)} (\mathsf{unsafe}_q(\mathbf{x}_k^t) \wedge \mathsf{enforce}(q,k)).
\end{aligned}
\tag{2.9}
$$

Every time point between the initial and final time point in a flow, $t \in [0, t_{i+1}]$, must lead to values of the continuous variables $\mathbf{x}$ which satisfy the invariant conditions $\mathsf{inv}_q(\mathbf{x})$. This is expressed in the extra universal quantifier for each continuous flow.

In the case of systems with both invariants and non-deterministic flows, i.e., for some $q \in Q$, there exist $\mathbf{a}_0, \mathbf{a}_t, \mathbf{a}_t' \in \mathbb{R}^n$ and $t \in \mathbb{R}$ such that $\mathbf{a}_t \neq \mathbf{a}_t'$ and $\mathbb{R} \vDash \mathsf{flow}_q(\mathbf{a}_0, \mathbf{a}_t, t)$ and $\mathbb{R} \vDash \mathsf{flow}_q(\mathbf{a}_0, \mathbf{a}_t', t)$, there is multiple possible values for the continuous variables at the same time point. This leads to multiple possible trajectories, and only one that satisfies the invariants on all time points is searched for, leading to the need to quantify over a trajectory, writing $\exists \xi \forall t. \mathsf{inv}(\xi(t))$. This is a second-order logic quantification, in [4] it is conjectured that it cannot be reduced to first-order expression.

Until this point, the invariant conditions were assumed to hold for all possible trajectories in the case of non-deterministic flow, this is now explicitly defined as ***Strictly-Imposed Invariants***.

A hybrid automaton H has strictly-imposed mode invariants if, for an arbitrary starting point $\mathbf{a}$ in mode q, satisfying $\mathsf{inv}_q(a)$, and any $\mathbf{b}, \mathbf{b}' \in \mathbf{X}(H)$ such that $\mathsf{flow}_q(\mathbf{a}, \mathbf{b}, \tau)$ and $\mathsf{flow}_q(\mathbf{a}, \mathbf{b}', \tau)$ are both true at the same time point, $\tau \in \mathbb{R}$, then $\mathsf{inv}_q(\mathbf{b})$ is true if, and only if, $\mathsf{inv}_q(\mathbf{b}')$ is also true. $\mathsf{flow}_q(\mathbf{x}, \mathbf{y}, t)$ and $\mathsf{inv}_q(\mathbf{x})$ are the flow and invariant conditions in mode q of H.

This being the case, all flows are required to satisfy the same invariants for a "witness" trajectory of bounded reachability. Meaning it is still possible to use the last presented encoding which requires that all flows satisfy the invariants. When this condition applies, it is still possible to use first-order encoding for reachability in the presence of non-deterministic flows.

It is now possible to define the δ-complete analysis problem. For a hybrid system H and a subset of its state space U, represented by the $\mathcal{L}_{\mathcal{F}}$-formula $\mathtt{unsafe}$, the δ-complete analysis for (k, M)-bounded reachability problem, with $k \in \mathbb{N}$ and $M \in \mathbb{R}^+$, has one of the following outputs:

- (k, M)-safety, where H does not reach $[\![\mathtt{unsafe}]\!]$ within the (k, M)-bound;
- δ-unsafety, where H^δ reaches $[\![\mathtt{unsafe}^\delta]\!]$ within the (k, M)-bound.

From this definition, one can conclude that, being $\text{Reach}_{H,U}(k, M)$ the $\mathcal{L}_{\mathcal{F}}$-formula encoding the (k, M)-bounded reachability of H with respect to U, then, $\mathbb{R} \vDash (\text{Reach}_{H,U}(k, M))^{\delta}$ if, and only if, $\exists \xi \in [\![H^{\delta}]\!]$ such that, for some $(k, t) \in T(\xi)$, $0 \leq t \leq M$, $(\xi(k, t), \sigma_{\xi}(k)) \in [\![\texttt{unsafe}^{\delta}]\!]$, U is reachable by U if and only if there is a trajectory ξ which can be mapped to modes in H.

Now, to find said trajectory, one only needs to solve the δ-decision problem (see 2.2) of $\text{Reach}_{H,U}(i, M)$ for $0 \leq i \leq k$. Following this line of thought, $\text{Reach}_{H,U}(i, M)$ is either false for all i, or it is δ-true for some number of steps.

In the former case, there is no trajectory $\xi \in [\![H]\!]$ that can reach U within i, and, consequently, the system is safe within the (k, M)-bound. In the latter, for some $i \leq k$, $\text{Reach}^{\delta}_{H,U}(i, M)$ is true, meaning that $\exists \xi \in [\![H^{\delta}]\!]$ that can reach the region represented by $\texttt{unsafe}^{\delta}$ in i steps, that is, within the (k, M)-bound.

This proves the decidability of the δ-complete analysis for bounded reachability problems for general $\mathcal{L}_{\mathcal{F}}$-representable hybrid systems.

dReach System Description

In Figure 2.1, taken from [1], the architecture of dReach is represented. It is composed of a bounded model-checking module and an SMT solver, dReal. An input hybrid system is translated into a first-order logic formula by the Encoder module, then, dReal, the SMT solver, solves the encoded δ-reachability problem, using the framework exposed in Section 2.2.

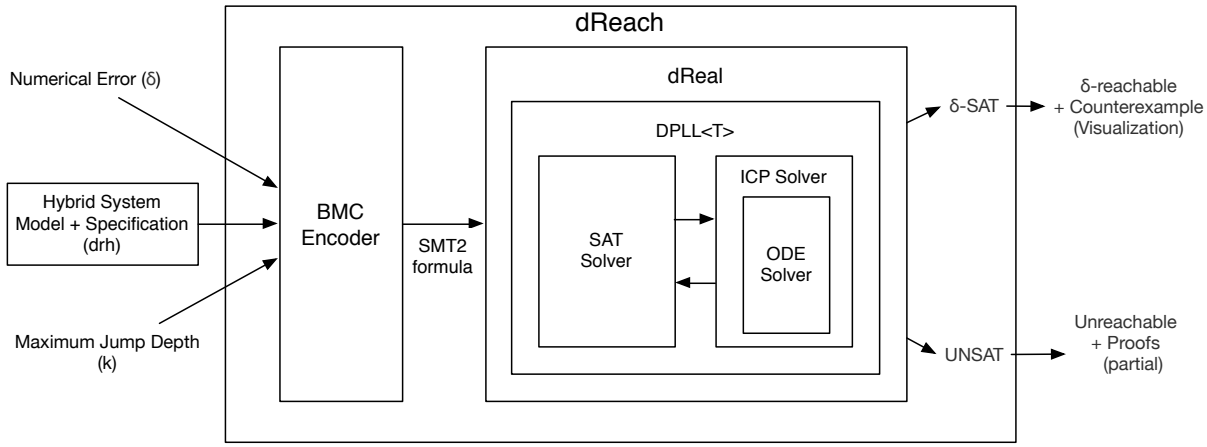

Figure 2.1: dReach architecture, as in [1].

To use dReach, a user must provide an input file, which specifies the hybrid system as well as the reachability properties in question and some time bounds on each mode's flow, and two other parameters, a bound on the number of mode changes and a numerical error bound δ. The tool takes these and a logical encoding is generated and iteratively solved by dReal, when the answer is δ-`reachable` (δ-`sat` output from dReal), a counterexample and its visualization are generated.

Differential equations and universal quantifiers generated by mode invariants are encoded as an extension of the SMT-LIB standard as it is defined in [6].

A command, `define-ode`, extends the language by assigning a name to a group of Ordinary Differential Equation (ODE)s. For instance, considering $\frac{dx}{dt} = v$ and $\frac{dv}{dt} = -x^2$, the command assigns the name flow_1 to these equations as follows

```
(define-ode flow_1 ((= d/dt[x] v) (= d/dt[v] (- 0 (^ x 2))))).
```

Integration is also encoded as to view the solution of the system of differential equations as a constraint between the initial and end state variables as well as time duration of that state. It is written as follows,

```
(= [x_t_1 ... x_t_n] (integral 0 t [x_0_1 ... x_0_n] flow_i)),
```

to represent the equation $\mathbf{x} = \mathbf{x}_0 + \int_0^t \text{flow}_i(\mathbf{x}(s))ds$. $\mathbf{x}(s)$ as a function is inferred by the solver.

Mode invariants need $\exists\forall^t$-formulas as defined in [7], these are a restricted form of formulas with quantifiers where the universal quantifiers are limited to the time variables. A new command to encode these formulas is introduced, `forall_t`. With a time bound $[0, time_i]$ and mode invariant f at mode q, the encoding is as follows

```
(forall_t q [0 time_i] f).
```

dReach usage

The input file, whose extension is `.drh`, needs five sections to define the hybrid system and reachability properties.

- Macro definitions - the user may define macros which can be used in the following sections;
- Variable declarations - The user specifies, for each continuous variable, its domain in a real interval, as well as a special declaration for $time$ variable which specifies the upper bound on mode duration;
- Mode definitions - each mode is defined by a unique id, its mode invariants, flow equations and jump conditions;
- Initial and goal conditions - the initial conditions and final goal conditions are defined by the user.

To run dReach, the following command needs to be used.

```
dReach <options> <drh file>
```

In `<options>` the user may define the k bound on steps (`-k <N>`) the default being 3, and one may also define a lower and upper bound `-l <N>` and `-u <N>` on steps.

2.5 Related Work

In this section a few writings are introduced which explore similar problems or apply similar techniques.

2.5.1 Task and Motion Planning

Task and Motion Planning (TMP) problems are ones in which, for a robot to generate a plan, it must be able to decide on tasks (discrete decisions) and motion (continuous decisions). They are, by definition, hybrid planning problems.

In [8] the authors solve a planning problem for a hybrid system. They consider a problem in which a mobile robot with arms has to pick up bottles from various positions and move them to their goal regions, a pick-and-place domain. The robot can move its base, torso, arm of gripper to a target joint configuration as its primitive actions.

Since the robot should be able to reason about discrete actions as well as continuous configurations, this is a hybrid problem. In order to handle this, the authors use sampling to discretize the configuration space and then use a Hierarchical Task Network (HTN) to find a solution. For this they present an algorithm, the State-Abstracted Hierarchical Task Network (SAHTN) algorithm, which, given a description of the domain and a function which specifies the relevant state-variables for doing an action from some state, outputs a hierarchical optimal solution.

The work in [9] also tackles planning problems which involve both discrete and continuous decisions. An algorithm is presented which uses incremental SMT solvers (SMT solver which can perform repeated satisfiability checks while constraints are pushed and popped from a stack maintained by the solver) to generate task plans and then a sampling-based motion planner, RRT-Connect, to generate a motion plan according to the previously generated task plan. If the motion planner fails to find a solution for a given task plan, then new task constraints are added to the task planner in order to produce a new task plan, if, however, the task planner fails to produce a new task plan, the main algorithm increments the task planning step horizon (maximum number of steps in the task plan) and the motion planning sampling horizon (timeout of the motion plan) and starts over without the added constraints to the task planner.

As a final example, another approach for solving TMP problems is presented in [10], named Robosynth. This approach receives as input a scene description, plan outline and a set of requirements.

A scene description is composed of a domain, where the user describes the information which remains unchanged throughout every instance of the problem, and a scene, the opposite, information which may change between instances. A plan outline consists of a program in which the user gives a high-level description of what a successful plan should look like. Lastly, the user should provide Robosynth with a goal and a set of invariants which must hold for the entirety of the planned paths.

Robosynth uses a variation of a manipulation graph, called a placement graph, as a sort of database of possible, feasible paths, then it uses an SMT solver to find a solution to the logical representation of the problem, querying the placement graph whenever it decides on a path as to verify it, thus solving both discrete and continuous parts of the problem.

2.5.2 Planning with Hybrid Automata

The work in [11] introduces a way of representing a hybrid planning problem using hybrid automata as well as its subsequent logical encoding as to allow an SMT solver to find a solution. This is the work which is closest to this thesis, and it is also the first of the presented works which takes into account the possibility of the presence of non-linearity in the system.

3

Approach

Contents

The problem this thesis is trying to solve is how to use SMT methods to generate a motion plan. This work, thus, uses an SMT solver only as a tool. It aims to create a model of the motion planning problem which can be used as a decidable SMT problem. Most importantly, methods to automate this whole modelling process are created, the goal being, given the simplest input possible, be able to reach a solution. The inputs considered here are listed bellow, accompanied by Figure 3.1 which gives an example of the first four items.

- A map;
- Its anchor points;
- An initial point;
- A final point;
- The pulling speed;
- The maximum number of robots alowed to be used at any given moment.

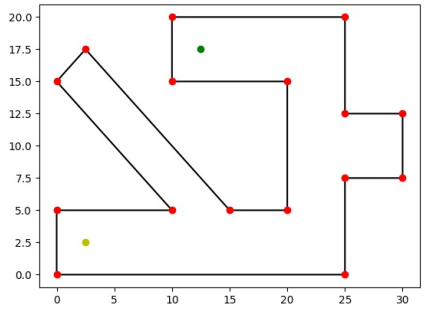

Figure 3.1: Example of input map, anchor points (represented by the red dots), initial point (yellow dot in the figure) and final point (green point).

The map would always be the same, except if some change occurred, in which case a centralised controller would update the map in every robot. The same applies to the connection points available. The initial point would be the robot's position, which could be sensed by the robot itself. Maximum pulling speed would be controlled internally by each robot, according to its condition and capabilities. Given this, a robot would already have most of the problem's information, only the final point and number of robots would have to be passed on, for each piece of cargo.

To find a solution to this problem, it must be described as a hybrid automaton. In this chapter the theoretical model and algorithms to generate the input file for dReach are explained.

3.1 First Iteration

The first step in creating a hybrid automaton is to define the variables. The only continuous variables considered at this stage of development are the coordinates of the cargo's position (x_c, y_c). Defining the discrete variables implies defining the different possible modes of the system. Since these modes serve to describe the dynamics of the continuous variables, defining them is simply a case of finding the different ways these vary. Given that the method of transportation considered in this thesis is through the attachment of each active robot to an anchor point followed by the pulling of the cargo to each of

those anchor points, there should be a mode for each possible combination of anchor points which can be used to pull the cargo. Additionally, the map is divided into convex regions with associated anchor points inside of which the cargo is able to be pulled by any of those anchors.

This can be done in at least two ways:

- the map can be divided into convex regions, without taking into account the anchor points, and only after having these computed regions are the anchor points associated to the ones they are inside of;
- the map can be divided according to the visibility regions of each anchor point. The cargo is guaranteed to be reachable by an anchor point if it is inside the visibility region of that anchor point. Overlaps in the regions correspond to areas where the cargo can be pulled by more than one anchor.

The first option was the one used due to its simplicity in processing the input data.

As an example, consider the simple problem presented in Figure 3.2, where only one robot intervenes in the transportation of the cargo. The map was divided into two overlapping convex regions with one anchor point each and only two modes were considered.

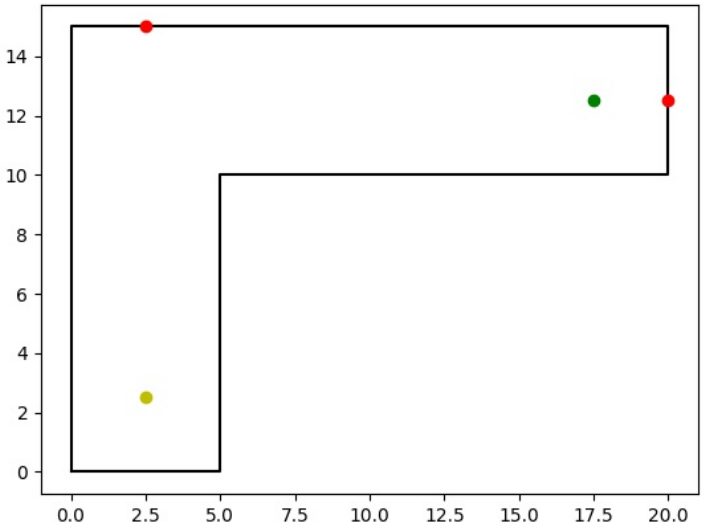

Figure 3.2: Simplest problem, the first to be considered in the implementation. The yellow and green points correspond to the initial and final points, respectively. The red dots are the available anchor points.

Let $a_1 = (2.5, 15)$ and $a_2 = (20, 12.5)$ be the two anchor points present in Figure 3.2 and q_1, q_2 the two modes corresponding, respectively, to each point.

The next step is to define the various conditions associated with each mode.

3.1.1 Flow Conditions

The method of transportation chosen for consideration in this thesis (see Chapter 1) can be represented by a vector, v_i, that "pulls" from the cargo to the corresponding anchor point at a certain speed w (Figure 3.3 illustrates this with vectors for the case where two robots are "pulling" the cargo). The flow equations for a mode where r robots are each connected to a distinct anchor point a_i, $1 \leq i \leq r$, can thus be written as follows.

$$\begin{cases} \dot{x} = w \sum_{i=1}^{r} \frac{x_{v_{a_i}}}{||v_{a_i}||} = w \sum_{i=1}^{r} \frac{x_{a_i} - x}{\sqrt{(x_{a_i} - x)^2 + (y_{a_i} - y)^2}} \\ \dot{y} = w \sum_{i=1}^{r} \frac{y_{v_{a_i}}}{||v_{a_i}||} = w \sum_{i=1}^{r} \frac{y_{a_i} - y}{\sqrt{(x_{a_i} - x)^2 + (y_{a_i} - y)^2}} \end{cases} \tag{3.1}$$

That is, the sum of every vector generated by each acting robot's pull. With $v_{a_i} = (x_{v_{a_i}}, y_{v_{a_i}}) = (x_{a_i} - x, y_{a_i} - y)$ being the vector generated by the pulling of the cargo in position (x, y) towards the anchor point $a_i = (x_{a_i}, y_{a_i})$.

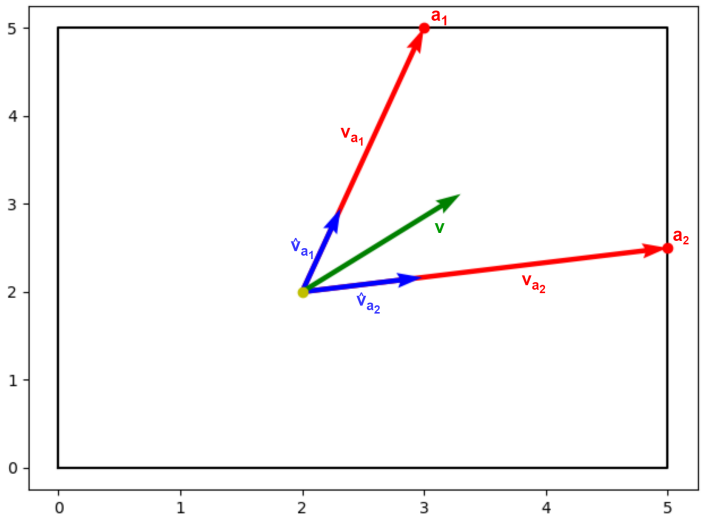

Figure 3.3: Example of set of vectors which represent the force generated by the pulling motion of the two active anchor points (red dots) and cargo position represented by the yellow dot. The red vectors represent the ones that "pull" the cargo, in blue their unit vectors, in green is the vector resulting from the addition of the two (blue) unit vectors.

A visual representation is given in Figure 3.3, where red vectors represent the pull of each robot, v_{a_i} attached to each anchor point a_i (red dots), blue vectors the corresponding unit vector $\hat{v_i}$, and in green the resulting motion vector v.

As an example, again using the problem in figure 3.2, for a mode corresponding to the robot being pulled upwards towards the red dot in position $(2.5, 15)$ the flow conditions are as follows.

$$\begin{cases} \dot{x} = w\frac{2.5-x}{\sqrt{(2.5-x)^2+(15-y)^2}} \\ \dot{y} = w\frac{15-y}{\sqrt{(2.5-x)^2+(15-y)^2}} \end{cases} \tag{3.2}$$

It is noteworthy that assuming the same fixed pulling speed, w, for every robot limits the degrees of freedom of this method.

After defining the flow in each mode, it is now time to define the invariants, what should be *required* to be true in each mode.

3.1.2 Invariants

Taking into account that each mode is defined by the set of anchor points to which the active robot (or set of active robots) pulling the cargo is (are) connected, the invariants should represent the regions in space where the cargo is reachable by each anchor point. The simplest solution to this was to firstly divide the map into convex regions. Inside this region, the cargo is reachable by any anchor point, as long as this anchor is also inside the region.

After the division (see Section 3.2 for a detailed description of the methods used) each connection point is associated to the region (or regions) it is in.

It is worth mentioning that an initial idea was to use path finding algorithms in a graph representing the divided map, to generate a ”reference” path. Toward this end, each region had a set of two points, which defined the middle line of said region. This was the reference used and originally the invariants were going to be just said line for each mode’s corresponding region. However, this was decided to be too great a trivialization of the satisfiability methods and the invariants were decided to be such as to define the inside of a mode’s region, as exemplified by equations 3.3 and 3.4.

The invariant creation process is now exemplified using the problem in Figure 3.2. Figure 3.4 shows the decomposition of the map, then, Equations 3.3 and 3.4 represent the invariant conditions for mode q_1 associated with the blue region in Figure 3.4 and mode q_2 associated with the other region in the same Figure.

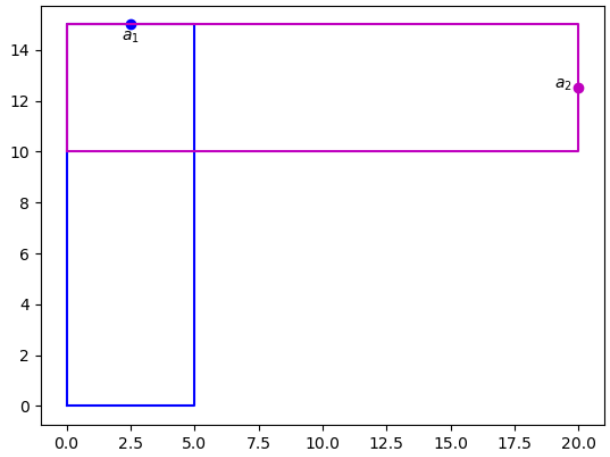

Figure 3.4: Decomposition of the map in Figure 3.2 into two regions, c_1 in blue and c_2 in magenta, the anchor points correspond to the regions of the same colour.

$$\text{inv}_{q_1} : \begin{cases} x > 0.0 \\ y < 15.0 \\ x < 5.0 \\ y > 0.0 \end{cases} \tag{3.3}$$

$$\text{inv}_{q_2} : \begin{cases} x > 0.0 \\ y < 15.0 \\ x < 20.0 \\ y > 10.0 \end{cases} \tag{3.4}$$

Notice how the conditions are derived from the equations which define the border lines of each region.

The final step in defining the hybrid automaton to describe the problem is to define the possible transitions between modes, the jumps.

3.1.3 Jump Conditions

Since this decomposition of the map consists of overlapping regions, modes q_i from a region c_i can only switch to modes q_j from region c_j, $i \neq j$, when the cargo (x, y) is in both c_i and c_j.

Jumping from one mode to another can only be done if the preconditions are all true and there can be effects of such transition on the continuous variables of the problem. As mentioned, the preconditions to any jumps which require moving regions is for the cargo to be in the overlap shared by those regions. In case the modes are of the same region and only the connection point changes, there are no preconditions, because the region is convex, it is always possible for this lastly mentioned type of transitions to happen. The effects on the continuous variables, cargo coordinates (x, y), are non-existent at switching time (beside the change in flow equations which naturally occurs when switching modes), the

initial values of the next mode for these variables are passed in the effects of a jump to that mode.

In the case of the simple example in Figure 3.2, equations 3.5 and 3.6 represent the jump conditions for both the switch $q_1 \rightarrow q_2$ and $q_2 \rightarrow q_1$. The preconditions of $\text{jump}_{q_i \rightarrow q_j}$ are represented by $\text{pre}_{q_i \rightarrow q_j}$ represent and its effects (through which the updating of the continuous variables from mode q_i to q_j is done) by $\text{eff}_{q_i \rightarrow q_j}$.

$$\text{jump}_{q_1 \rightarrow q_2} : \begin{array}{r} \text{pre}_{q_1 \rightarrow q_2} : \begin{cases} x > 0.0 \\ y < 15.0 \\ x < 20.0 \\ y > 10.0 \end{cases} \\ \text{eff}_{q_1 \rightarrow q_2} : \begin{cases} x_2^0 = x_1^t \\ y_2^0 = y_1^t \end{cases} \end{array} \tag{3.5}$$

$$\text{jump}_{q_2 \rightarrow q_1} : \begin{array}{r} \text{pre}_{q_2 \rightarrow q_1} : \begin{cases} x > 0.0 \\ y < 15.0 \\ x < 5.0 \\ y > 0.0 \end{cases} \\ \text{eff}_{q_2 \rightarrow q_1} : \begin{cases} x_1^0 = x_2^t \\ y_1^0 = y_2^t \end{cases} \end{array} \tag{3.6}$$

Where x_i^t and y_i^t are the final values of (x, y) of mode i. Notice how the preconditions of the switch $q_i \rightarrow q_j$, $\text{pre}_{q_i \rightarrow q_j}$, coincide with the invariant conditions of mode q_j, translating the first paragraph of this subsection into logical conditions.

3.1.4 Initial and Final conditions

The `.drh` file, which encodes the problem to the dReach input language, requires the initial and final modes to be explicitly indicated. Due to this, two modes were created to act only as initial and final modes. Otherwise, there would have to be an additional step in this thesis' process in which the modes with associated regions which contained the initial point would be considered as initial modes.

These modes have invariants which coincide with the problem's initial and final conditions and null flow conditions (the cargo should be stopped in these modes). The initial mode contains jumps to every mode associated with the region the initial point is in, the final mode contains no jumps since it will always be the last step in the plan.

The initial and final conditions must be defined in the `.drh` file as follows.

```
1 init:
2 @{} conditions;
3
4 goal:
5 @{} conditions;
```

Where inside the brackets should be the id numbers of the initial and final mode, respectively (the modes are, naturally, identified by an id number within the automaton's control graph), and to their right the initial and final conditions, respectively, which must be true inside each correspondinhg mode.

The problem described in figure 3.2 has the initial point $p_i = (2.5, 2.5)$ and the final point $p_f = (17.5, 12.5)$, thus, the initial and final conditions are simply $\{(x = 2.5) \wedge (y = 2.5)\}$ and $\{(x = 17.5) \wedge (y = 12.5)\}$.

As such, two modes are introduced, q_i the initial, and q_f the final. They are defined in Equations 3.7 and 3.8.

$$q_i : \begin{cases} \text{inv}_i : \begin{cases} x = 2.5 \\ y = 2.5 \end{cases} \\ \text{flow}_i : \begin{cases} \dot{x} = 0 \\ \dot{y} = 0 \end{cases} \\ \text{jump}_{q_i \to q_1} : \begin{cases} \text{pre}_{q_i \to q_1} : \begin{cases} x > 0.0 \\ y < 15.0 \\ x < 5.0 \\ y > 0.0 \end{cases} \\ \text{eff}_{q_i \to q_1} : \begin{cases} x_1^0 = x_i^t \\ y_1^0 = y_i^t \end{cases} \end{cases} \end{cases} \tag{3.7}$$

$$q_f : \begin{cases} \text{inv}_f : \begin{cases} x = 17.5 \\ y = 12.5 \end{cases} \\ \text{flow}_f : \begin{cases} \dot{x} = 0 \\ \dot{y} = 0 \end{cases} \\ \text{jump}_{q_i \to q_1} : \end{cases} \tag{3.8}$$

As already mentioned, the final mode does not have any actual jump condition and the initial mode only connects to mode q_1 because the initial point is only inside region 1. It is worth mentioning that the modes to which the initial mode connects do not connect back to it.

Finally, a jump condition from mode q_2 to mode q_f was added to the model (equation 3.9).

$$\text{jump}_{q_2 \to q_f} : \begin{cases} \text{pre}_{q_2 \to q_f} : \begin{cases} x = 17.5 \\ y = 12.5 \end{cases} \\ \text{eff}_{q_2 \to q_f} : \begin{cases} x_f^0 = x_2^t \\ y_f^0 = y_2^t \end{cases} \end{cases} \tag{3.9}$$

In the end the `.drh` file will encode the initial and final in the following way.

```
1 init:
2 @1  (and (x = 2.5) (y = 2.5));
3
4 goal:
5 @2  (and (x = 17.5) (y = 12.5));
```

3.1.5 .drh file and dReach

After creating everything needed to define the hybrid automaton which encodes the problem, this information needs to be fed to dReach, through a `.drh` file. This file contains every mode in the control graph as well as initial and final conditions among other trivial definitions.

In Appendix A.1, it is possible to see the .drh file generated for the problem in Figure 3.2.

There are, however, a few differences between the description in the `.drh` file and the model presented in this section. Firstly, the identification of the modes is different, modes 1 and 2 in the file correspond to modes q_i and q_f in the model, and modes 3 and 4 to q_1 and q_2.

Secondly, the flow equations, as presented, could not be handled by dReach in an acceptable amount of time when considering the full problem (it takes 13 minutes to solve a simple problem with a two-step solution and is not able to reach one for a more complex problem with a 4 step solution when left running overnight, which is not acceptable for this type of decision making processes). Due to this, two methods were devised (both discussed in Section 3.2), the first one involves using an approximation for the flow conditions of each mode, the other one is to split the full problem into smaller, more manageable problems.

For the simple problem in Figure 3.2, an approximation of the actual flow was given in place of the actual original equations.

In this case, since it was known that mode q_1 only moves in the positive direction of variable y and q_2 in the positive direction of x, their equivalents in the `.drh` file have the equations in 3.10 and 3.11 as flow conditions.

$$\text{flow}_{3(q_1)} : \begin{cases} \dot{x} = 0 \\ \dot{y} = 2 \end{cases} \tag{3.10}$$

$$\text{flow}_{4(q_2)} : \begin{cases} \dot{x} = 2 \\ \dot{y} = 0 \end{cases} \tag{3.11}$$

The value of 2 was an arbitrary choice, in the next section the method used to handle this approximation is explored more in depth.

Lastly, since the solution is trivial and known (the system should follow $q_i(1) \rightarrow q_1(3) \rightarrow q_2(4) \rightarrow q_f(2)$ as a sequence of modes, in parenthesis are the equivalent mode ids in the `.drh` file), for simplification, the switch $q_2 \rightarrow q_1$ was omitted from the description file.

The description in the `.drh` file (see A.1) leads to the solution represented in Figure 3.5.

This first simple problem, model and the corresponding `.drh` definition served to explore the dReach tool and to decide how best to describe any problem given a map, a set of anchor points and an initial and final point. After this, algorithms which modelled the problem and generated its `.drh` file were created.

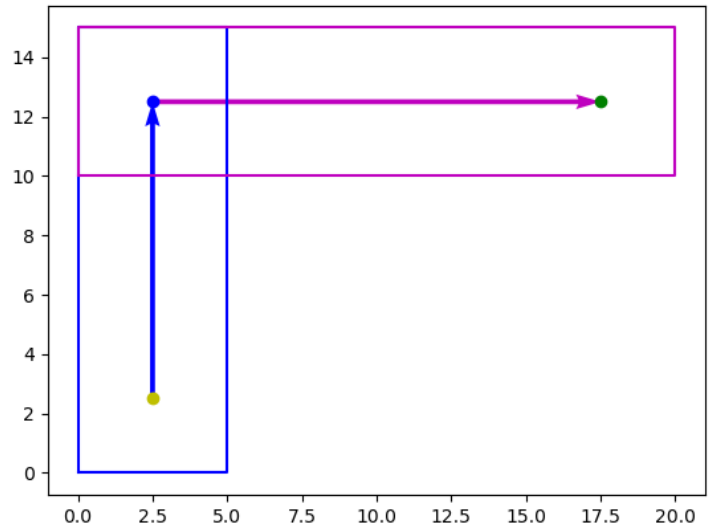

Figure 3.5: Representation of the solution of the simplest problem, described in section 3.1. The blue region and arrow correspond to mode q_1 and the magenta ones to mode q_2. The yellow and green points correspond to the initial and final points, respectively.

3.2 Automation

The main focus of this thesis is to find a path plan given a map description, including anchor points, and an initial and final points. It is not very desirable for the process described in the previous section to be made by hand. For that reason, a few methods, which automate the process of modelling the problem and generating the respective `.drh` file, were created and are described in this section.

3.2.1 Input File

The first step is to create a file containing the problem description, which is then read and its information used in the remaining processes. This file should end in `.prb`

Its structure is fairly simple. The first line is where the value of w is defined, the speed at which each robot is pulling the cargo to the anchor point to which it is connected. In the following lines, the bounds on each continuous variable are written, as well as the maximum time the system can spend in a given mode.

Afterwards, the maximum number of robots to be used in each mode is defined, followed by the initial and final points for the problem.

The map is represented by a polygon, as such, a block containing its vertices, in clockwise order, is written. Finally, there is a block where the coordinates of the anchor points are given.

An example of this file can be seen in Appendix A.2. Notice how the vertices of the polygon coincide with the anchor points. This is explained in the next subsection, when the convex decomposition of the map is delved into.

3.2.2 Convex Decomposition

After reading the input file, the first step of the modelling process is to decompose the map into convex polygons. The method used here is the one briefly described in [12] and a flowchart of the implementation done here is presented in Appendix B.

The map's convex decomposition is done by recurrently calling an algorithm which receives a simple polygon with no holes, P, and tries to find a convex polygon within P, represented by a list of vertex $v_1, \ldots, v_n$ in clockwise order.

Note that for a polygon of size n, when referring to its possible indexes, the reader should consider the remainder of the division by n, for instance, index i should be read as $i\%n$, $\%$ representing the remainder. This allows looping through the list of vertices which represents a polygon even when the starting point doesn't coincide with the one of said list.

Algorithm 3.1 has the corresponding pseudocode, with Algorithm 3.2 being an auxiliary function used in the process.

Starting at a given index, s, of the polygon, given as an input along with $P = \{v_1, \ldots, v_n\}$, the algorithm appends vertices v_s and v_{s+1} into a list L, initialized as empty, which holds the vertices of the convex polygon. After L consists of two vertices, the algorithm only adds a vertex v_i after checking three angles. These are ang(v_i, l_1, l_2), ang(l_k, v_i, l_1) and ang(l_{k-1}, l_k, v_i), where l_i is the vertex at position i of L, and ang(a, b, c) is the angle from vector $\vec{ab}$ to $\vec{bc}$ in the counter-clockwise direction. Since the vertex are in clockwise order in L (because they are so in P), the three angles checked give the internal angle in v_i and each of its two adjacent neighbors, as such, if any of these are greater than 180, the addition of v_i fails and the algorithm goes to the next phase. If this is not the case, v_i is successfully added to L and removed from $P \setminus L$ (which was initialized as a copy of P), the remaining polygon when L is subtracted from P.

Afterwards, in case L has more than two vertices, the algorithm checks its validity. First it checks if there is any notch, v, of $P \setminus L$ in R, the smallest rectangle containing L with sides parallel to the x and y axis. A vertex of a polygon is a notch if its internal angle is greater than 180. In case this is the situation, it then checks if that notch is inside L itself. This being the case, L needs to be changed, otherwise L is valid and the algorithm can move on to the next phase.

Correcting L is a simple case of removing its last vertex, l_k, as well as every vertex contained in the half-plane defined by l_1 and l_k containing the notch that triggered the correction, v.

This verification and correction step is done until either L converges to a valid convex polygon (i.e.,

Algorithm 3.1: MP3

```
Data: P - simple polygon with no holes
s - starting index
Result: L - extracted convex polygon
P\L - P without L
the index (in P\L) of the last element in L
if P is convex then
    return L = [], P\L = P and 0;
end
Add P(s) and P(s+1) to L;
Initialise i at s+2;
repeat
    Add P(i) to L;
    increment i;
until P(i) creates a notch in L;
Remove lastly added P(i) from L
repeat
    Check_L(L);
until L returns unchanged;
Initialise i at s-1;
repeat
    Add P(i) to L;
    decrease i by 1;
until P(i) creates a notch in L;
Remove lastly added P(i) from L;
repeat
    Check_L(L);
until L returns unchanged;
if L is of length 2 or any vertex of L is a notch then
else
    return L = [], P\L = P and 0;
end
return L, P\L, and i;
```

L remains unchanged after this step) or L becomes composed by less than three vertices (stops being a polygon).

The algorithm now starts adding vertices to $L = \{l_1, \ldots, l_k\}$ once again, except this time it starts adding in counter-clockwise order. Let v_s be the equivalent in $P \setminus L$ of l_1, $\{v_{s-1}, vs-2, \ldots\}$ is the sequence of vertices whose addition will be considered. When adding a new vertex, the internal angles it creates in itself, l_k and the lastly added vertex (l_1 if no new vertex has been added in this stage yet) are tested as in the first vertex addition stage. Note that if a, b, c are in clockwise order, using ang(a, b, c) always gives the internal angle at vertex b.

Another verification is done in the same conditions as previously mentioned. Finally, if L is composed of more than two vertices and if either its first or last vertex is a notch of P, then L is accepted as a convex component of P.

When iterated, this algorithm is capable of decomposing a simple polygon P with no holes into a set

Algorithm 3.2: Check_L

```
Data: L - extracted convex polygon
P\L - P without L
Result: changed - boolean variable True if L changed during this algorithm and False otherwise
if any notch of P\L in R (the smallest possible rectangle which contains L) then
    if any notch of P\L in L then
        remove last vertex from L;
        remove vertices from L that are in the half-plane created by the first and last vertices of L
         which also contains the notch which triggered the condition;
        return changed = True
    else
        return changed = False
    end
else
    return changed = False
end
```

D of convex polygons. It starts by receiving P and the index of the first element in P and outputting a convex polygon L, which is promptly added to D, $P \setminus L$ and the index, in $P \setminus L$, of the last element of L, f. At each subsequent iteration, it takes $P \setminus L$ as a polygon to divide and f as the starting index, repeating this until L is returned empty, meaning that $P \setminus L$ was already convex at the end of the previous iteration and it is then added to D, completing the decomposition.

As a result of this decomposition, the original map is transformed into a set of non-overlapping convex polygons, which are connected through shared borders. This forces the need to change the preconditions of jumps between modes. Also, this decomposition gives way to the possibility of regions not having any anchor point inside them. This can be solved either by inserting an anchor point in each vertex or by computing the visibility polygon for each available anchor point (if a point a is in the visibility area of another point b in a polygon then the line defined by those points is completely inside the polygon) and checking if the region is inside that visibility polygon, associating them with each other if that is the case. The first option was the one considered in this thesis as it was not a focal problem at this point. These methods should find a solution, if the anchor points disposition allows for one, that can be guaranteed with every anchor point being in the vertices of the map.

The convex decomposition of the simplest problem map (see Section 3.1) is depicted in Figure 3.6.

After decomposition, the map is now composed by a set of non-overlapping, convex polygons.

3.2.3 Polygon Extension

After analysing the results of implementing the divide and conquer method (subsection 3.2.5) on top of a map consisting of non-overlapping polygons (this analysis can be found in Chapter 4) it was found that this does not lead to a working program.

To solve this, an algorithm which extends each polygon into its neighbours, while maintaining their

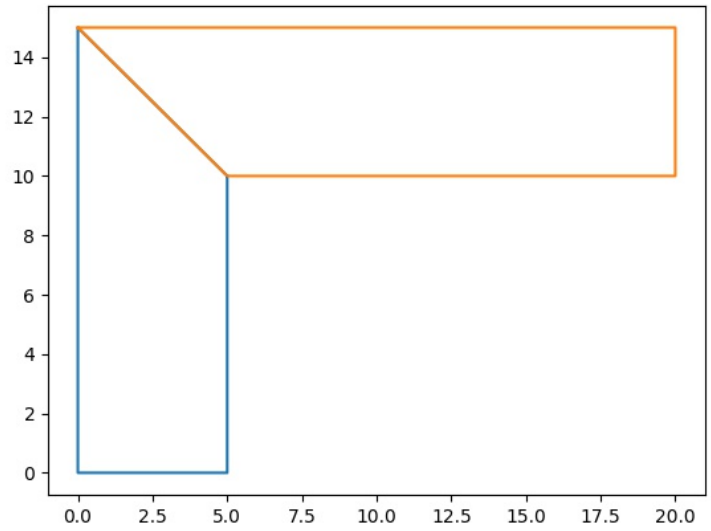

Figure 3.6: Convex decomposition of simplest problem map

convexity was created.

For a given map, which is represented by a list of its vertices' coordinates in a clockwise order, there will be a set of convex polygons which together form the polygon which corresponds to this map.

Consider the map in Figure 3.7 and its given convex decomposition in Figure 3.8. The method described henceforth is going to expand polygon A into polygon B and vice-versa.

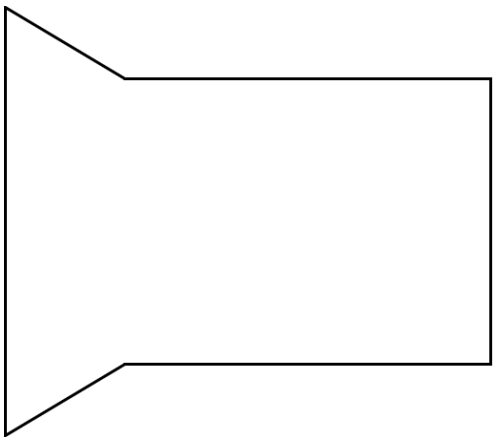

Figure 3.7: Example of map for Polygon Extension algorithm.

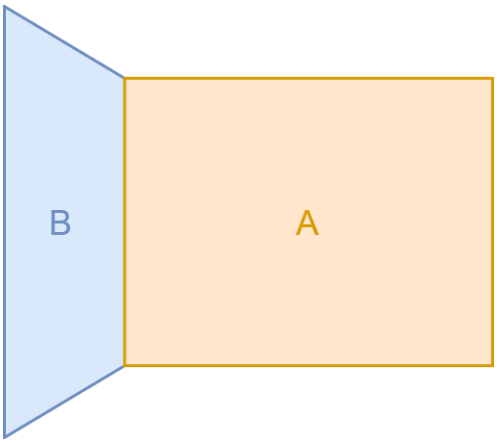

Figure 3.8: Decomposition of map if Figure 3.7.

In order to do this, the algorithm considers the two vertex points which define the border line between these polygons. When computing the expansion of polygon A into polygon B, pictured in Figure 3.9, the border lines neighbouring the one being expanded (the one between the polygons A and B) are extended

into in the direction of polygon B. In Figure 3.9 these would be the lines $\overline{{}^{A}P_n^{A}P_1}$ and $\overline{{}^{A}P_3^{A}P_2}$. Then, there are three possible cases:

- The intersection point of $\overline{{}^{A}P_n^{A}P_1}$ with $\overline{{}^{A}P_3^{A}P_2}$, ${}^{A}P_i$ exists and is **inside** polygon B;
- ${}^{A}P_i$ exists and is **outside** polygon B;
- ${}^{A}P_i$ does not exist.

In the first case, ${}^{A}P_i$, is added to polygon A in between ${}^{A}P_1$ and ${}^{A}P_2$, in the correct, clockwise order.

In both of the other two cases, the intersection points of $\overline{{}^{A}P_n^{A}P_1}$ and $\overline{{}^{A}P_3^{A}P_2}$ with polygon B's border are computed, ${}^{A}P_i, 1$ and ${}^{A}P_i, 2$ respectively. Then, they are added to a copy of polygon B, in the correct, clockwise order. Finally, starting from ${}^{A}P_i, 1$, each point, in clockwise order, which belongs to polygon B, is added to polygon A, between ${}^{A}P_1$ and ${}^{A}P_2$, in the correct, clockwise order.

In Figure 3.9, one can see that it is the last case which happens when extending polygon A into B, while in the inverse extension, polygon B into A, it is the first.

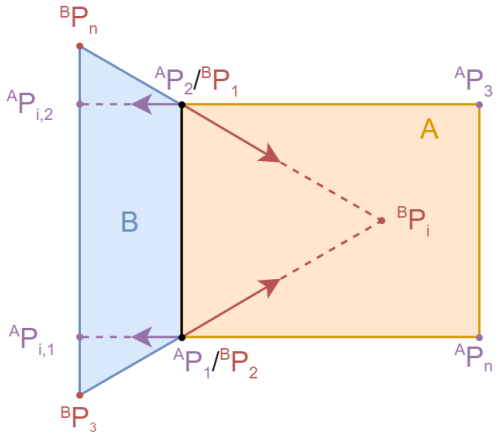

Figure 3.9: Example of Polygon Extension algorithm.

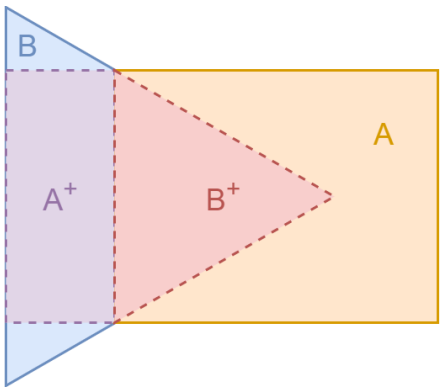

Figure 3.10: Overlapping zones resulting from the algorithm in3.9.

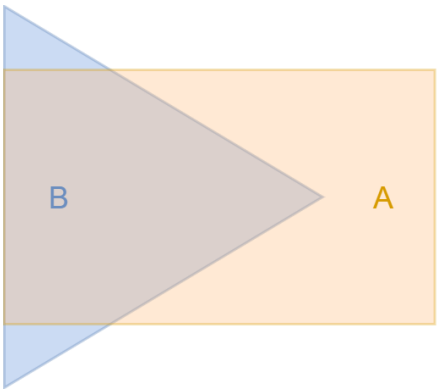

Figure 3.11: Final convex decomposition for the map in 3.7.

Figure 3.10 shows the computed overlap zones to be added to each polygon and Figure 3.11 shows the final convex decomposition for the map in Figure 3.7, after expanding the polygons in the initial decomposition.

Consider the slightly altered map, represented in Figure 3.12 already decomposed, and the application of the extension algorithm to its composing polygons, A and B, also presented in Figure 3.12. When expanding polygon B into A, an example of the second case can be witnessed, in which the intersection point between lines $\overline{{}^{B}P_{n}^{B}P_{1}}$ and $\overline{{}^{B}P_{3}^{B}P_{2}}$, ${}^{B}P_{i}$, is outside of polygon A.

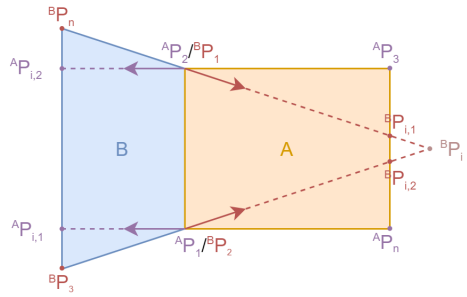

Figure 3.12: Example of Polygon Extension algorithm.

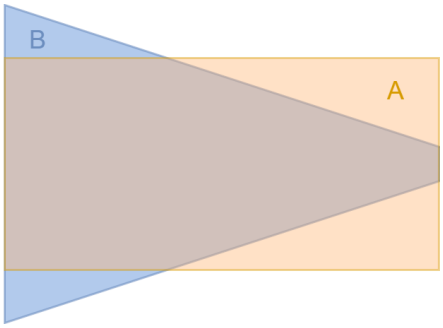

Figure 3.13: Final convex decomposition resulting from the algorithm application in Figure 3.12

This is the last step in the transformation of the original map into a set of convex, overlapping polygons. This transformation is very important to the model, for it enables the division of the problem into smaller, simpler sub-problems, a crucial step in its resolution, further explored in subsection 3.2.5.

The result of extending the polygons in Figure 3.6 is depicted in Figure 3.14.

3.2.4 Generating Modes

Now that the map is divided into overlapping, convex regions, it is time to start generating the hybrid automaton which will define the problem.

Firstly, the initial and final modes are generated, q_1 being the former and q_2 the latter. They have null flows, the invariants of q_1 lock (x, y) in the initial position and in q_2 they define a box centered in the final point by adding and subtracting a value (2.5 was used in this thesis). For instance, for a final point $(5, 5)$, its invariant conditions would be $\{(x > 2.5), (x < 7.5), (y > 2.5), (y < 7.5)\}$. The jump conditions of these modes are only generated after every other mode is first defined.

Next, for every region, the corresponding modes are generated according to the maximum number of robots which can be used and to the available anchor points. For instance, a region with 4 anchor points in a problem with a maximum allowance of two robots, would generate 4 modes for the cases where only one robot is active (one for each anchor point) and another 6 for the cases where two robots

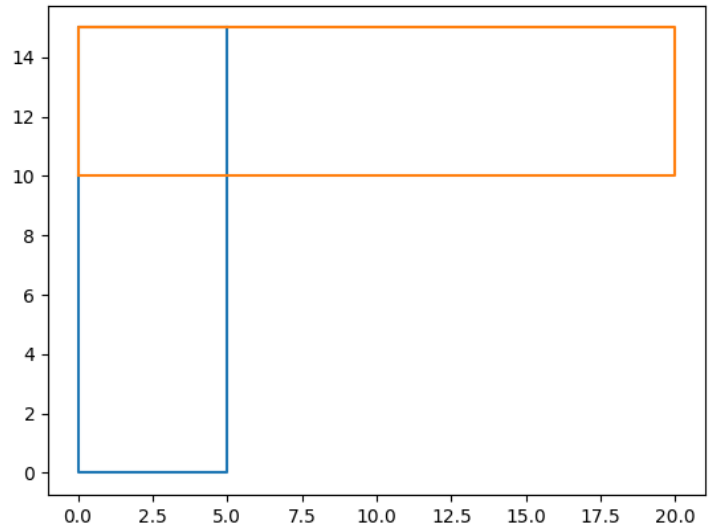

Figure 3.14: Result of the extension of polygons in Figure 3.6.

are being used (there are 6 possible combinations of 4 anchor points when grouped into pairs).

The invariants of each one of these modes are the defining inequations of the corresponding region.

Consider again the example in Figure 3.14, modes in the blue region would have invariant conditions defined as follows.

$$\text{inv} : \begin{cases} x > 0 \\ x < 5 \\ y > 0 \\ y < 15 \end{cases} \tag{3.12}$$

Notice how the conditions derived from actual walls of the map do not allow for (x, y) to actually be in said walls, however, the last of the conditions is derived from the line which separates both regions, as such, (x, y) should (and, in fact needs to) be able to reach that line, otherwise the system would not allow (x, y) to travel from a region to another.

After realising that the flow conditions, as they were defined in Section 3.1, could not be handled by dReach in an acceptable amount of time when considering the full problem (it takes 13 minutes to solve a simple problem with a two-step solution and is not able to reach one for a more complex problem with a 4 step solution when left running overnight), an initial fix, which consisted in devising an approximation for the flow equations of each mode, was tried.

The idea behind the approximation is to take the initial vector of each mode (the vector which results from the sum of all the vectors applied to the cargo by the pulling motion exerted by each robot) and use it as the direction for the entirety mode.

As such, a new set of continuous variables was created, v_x and v_y, which represent the x and y

components of the resulting vector v, the one which results from the sum of every unit vector created by each active robot. These new variables are computed when a switch occurs. When switching from mode q to mode q', supposing q' has anchor points $a_i = (x_{a_i}, y_{a_i})$ and $a_j = (x_{a_j}, y_{a_j})$ active, v'_x and v'_y are computed in the following way.

$$\begin{cases} v'_x = w \sum_{k=i}^{j} \frac{x_{a_k} - x}{\sqrt{(x_{a_k}-x)^2+(y_{a_k}-y)^2}} = \frac{x_{a_i} - x^t}{\sqrt{(x_{a_i}-x^t)^2+(y_{a_i}-y^t)^2}} + \frac{x_{a_j} - x^t}{\sqrt{(x_{a_j}-x^t)^2+(y_{a_j}-y^t)^2}} \\ v'_y = w \sum_{k=i}^{j} \frac{y_{a_k} - y}{\sqrt{(x_{a_k}-x)^2+(y_{a_k}-y)^2}} = \frac{y_{a_i} - y^t}{\sqrt{(x_{a_i}-x^t)^2+(y_{a_i}-y^t)^2}} + \frac{y_{a_j} - y^t}{\sqrt{(x_{a_j}-x^t)^2+(y_{a_j}-y^t)^2}} \end{cases} \tag{3.13}$$

Where x^t and y^t are the values of (x, y) in mode q at the time of switching. The flow of these new variables is null, as to maintain their value throughout the mode.

This computation gives an approximation of the motion caused by the pull from each anchor point based on the initial vector which describes the movement of the cargo at the beginning of each mode.

With these new variables, the flow in each mode can be described as in Equations 3.14.

$$\text{flow}_i : \begin{cases} \dot{v_x} = 0 \\ \dot{v_y} = 0 \\ \dot{x} = w \cdot v_x \\ \dot{y} = w \cdot v_y \end{cases} \tag{3.14}$$

A few effects are also added to the jumps between modes. These conditions force the update of v_x and v_y according to the active anchor points, $a_i = (x_{a_i}, y_{a_i})$, and the position of the cargo at the time of switching, (x_c, y_c).

$$\begin{cases} v^0_{x_2} = \frac{x_{a_i} - x^t_c}{\sqrt{(x_{a_i}-x^t_c)^2+(y_{a_i}-y^t_c)^2}} \\ v^0_{y_2} = \frac{y_{a_i} - y^t_c}{\sqrt{(x_{a_i}-x^t_c)^2+(y_{a_i}-y^t_c)^2}} \end{cases} \tag{3.15}$$

These values of v_x and v_y then stay constant, defining the values for the flow conditions throughout the entirety of the mode, only being re-calculated when switching to another mode again.

Referring once more to the problem in Figure 3.2, for instance, conditions in Equations 3.16 and 3.17 should be added to $\text{eff}_{q_1 \to q_2}$ and $\text{eff}_{q_2 \to q_1}$ in the jumps expressed in Equations 3.5 and 3.6. It is these values computed upon entrance in new mode which define the value of the flow conditions for the duration of the step.

$$\begin{cases} v^0_{x_2} = \frac{20 - x^t_1}{\sqrt{(20-x^t_1)^2+(12.5-y^t_1)^2}} \\ v^0_{y_2} = \frac{12.5 - y^t_1}{\sqrt{(20-x^t_1)^2+(12.5-y^t_1)^2}} \end{cases} \tag{3.16}$$

$$\begin{cases} v^0_{x_1} = \frac{2.5 - x^t_2}{\sqrt{(2.5-x^t_2)^2+(15-y^t_2)^2}} \\ v^0_{y_1} = \frac{15 - y^t_2}{\sqrt{(2.5-x^t_2)^2+(15-y^t_2)^2}} \end{cases} \tag{3.17}$$

There is, however, an error associated with this approximation, it is explained in Chapter 4. To solve this, one can try to enhance the approximation method used in the following way.

Each mode q_i can be divided into two sub-modes, q_i^0 and q_i^1, which have a new variable, d representing the duration of each $q_i^m,\ m = 0, 1$ (this is achieved by setting $\dot{d} = 1$ as the flow conditions for d). Then, a limit is added to the value of d, by inserting $d \leq d_max$ to the set of invariants of the new modes. A jump condition is also needed, corresponding to the breaking of this limit, which enforces switching from $q_i^{0(1)}$ to $q_i^{1(0)}$. These alterations enforce the update of the initial motion vector, discretizing the handling of the flow conditions and reducing the approximation error.

For instance, suppose a mode q_i is defined in `.drh` by the following sets of conditions. inv_{q_i}, flow_{q_i} and $\text{jump}_{q_i \to q_j}$, q_i's invariants, flow and jump conditions. The new modes q_i^0 and q_i^1 would be such that

$$\begin{aligned} \text{inv}_{q_i^0} = \text{inv}_{q_i^1} &= \text{inv}_{q_i} \cup \{d \leq d_{max}\} \\ \text{flow}_{q_i^0} = \text{flow}_{q_i^1} &= \text{flow}_{q_i} \cup \{\dot{d} = 1\} \\ \text{jump}_{q_i^0 \to q_j^0} &= \text{jump}_{q_i \to q_j} \cup \text{jump}_{q_i^0 \to q_i^1} \\ \text{jump}_{q_i^1 \to q_j^0} &= \text{jump}_{q_i \to q_j} \cup \text{jump}_{q_i^1 \to q_i^0} \end{aligned} \tag{3.18}$$

Where each jump in $\text{jump}_{q_i^{0/1} \to q_j^0}$ has the same preconditions as its equivalent in $\text{jump}_{q_i \to q_j}$ and their effects are extended from the latter's, adding the restart of the duration variable d ($d_{j^0}^0 = 0$). There is also a jump from q_i^0 to q_i^1 and vice-versa, which have as preconditions $d = d_{max}$, this creates a loop inside mode q_i which persists until the cargo reaches a position in which the solver decides to switch to a different mode $q_{j \neq i}$.

This is exemplified by figures 3.15 and 3.16, the former being an Input/Output (I/O) representation of mode q_i and the latter an I/O representation of q_i^0 and q_i^1.

Figure 3.15: I/O representation of mode q_i.

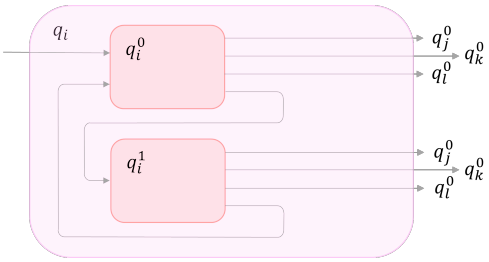

Figure 3.16: I/O representation of modes q_i^0 and q_i^1, the discretization of mode q_i.

Since the effects of switching between the sub-modes of q_i are the same as the effects of jumps from every mode to q_i (in Chapter 3 it is explained how the effects of entering a mode are what actually enforces the theoretical flow conditions of that mode, due to the approximation which the method being explained is trying to improve), forcing one of these switches is equivalent to recomputing the initial vector in which the approximation is based. With this is mind, it is fair to say that forcing a switch like this to happen every d_{max} seconds, would also force an update of the initial motion vector used in the flow approximation, thus discretizing its processing.

Although this seems to be an acceptable solution for the approximation problem, this way of modelling the problem may, in fact, lead to an increase in the solving time taken by dReach, since the number of steps needed to find a solution to the generated reachability problems increases as d_{max} decreases. Therefore, in order to get a more accurate approximation one must sacrifice the time needed to solve the problem as a whole.

Three scenarios were tested, one in which the mode refreshes every 1 second, another in which the mode refreshes every 0.5 seconds and the last in which it refreshes every 0.1 seconds. These generated the graphs in Figures 3.17, 3.18 and 3.19, respectively.

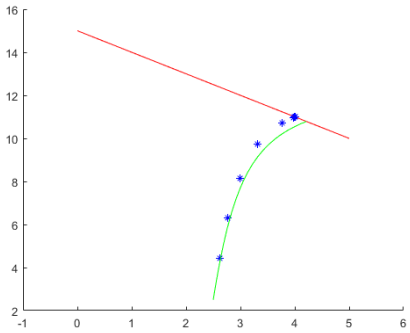

Figure 3.17: Graph of recursive method test with 1 second of refresh time.

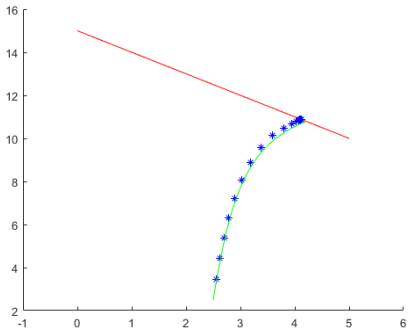

Figure 3.18: Graph of recursive method test with 0.5 seconds of refresh time.

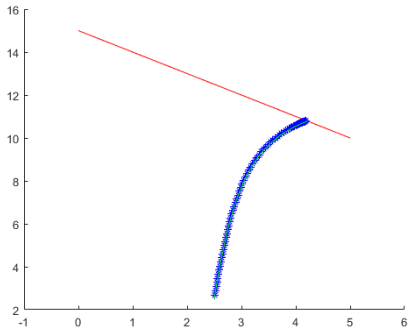

Figure 3.19: Graph of recursive method test with 0.1 seconds of refresh time.

All three of these tests tried to approximate the path analysed in Figure 4.5.

The first, whose graph is in Figure 3.17, was done with a refresh time of 1 second. That is, each second the vector is recalculated and used as the new motion direction. This refresh frequency needs about 10 steps to get to the target line, depicted in red. It is expected that the more time passes, the greater the error between the computed approximation, depicted by the blue stars, and the true path, the green line, becomes. In the end this error becomes too great to be negligible.

In the remaining tests, the results are better when it comes to the error, although the steps needed to

reach the goal line largely increase, with the test in Figure 3.18, with refresh rate of 0.5 seconds, taking about 20 steps to reach the goal and the one in Figure 3.19 about 100.

This does not indicate a good method. On one hand, the number of steps needed by each refresh rate would lead to a huge increase of computing time by the solver. Even in the best case of these three scenarios, where the number of steps is one, the solver already struggles with more than five or six steps, hence this situation would only be worse, not to mention 20 or even 100 steps. On the other hand, even if the first scenario was feasible in terms of steps, the error it generates would only propagate further and lead to unreliable plans, when there would be any.

Due to these restraints, focus was given instead to extending the polygons in the map's convex decomposition in order to correct the previously mentioned issue with the combination of the "divide and conquer" method (see Section 3.2) with the non-overlapping polygons resulting from the convex decomposition of the map (see Section 3.2). This resulted in the polygon extension method explained in subsection 3.2.3

To this end, the flow conditions only have this approximation when the regions used are not extended and do not overlap, otherwise they are as defined in Section 3.1.

The only thing left to do is to connect the modes. For a mode q, every other mode q' is checked. If their corresponding regions are the same, they are automatically connected, with preconditions for the switch being generated as they were in Section 3.1. If they are in different regions, their overlap zones are used to generate the jump conditions, (x, y) should be in this overlap zone in order to jump from one mode to another. For instance, considering q_i and q_j the modes associated with the blue and orange regions in Figure 3.14, respectively, the preconditions of the switch $q_i \rightarrow q_j$, are $\text{pre}_{q_i \rightarrow q_j} : \{x < 5 \wedge y > -x + 15 \wedge y < 15\}$, which equates to the area added, after extending it, to the blue polygon from Figure 3.6, the overlap zone.

If the initial point (x_i, y_i) is in the region of mode q, then the initial mode q_1 is connected to it and a jump condition, whose precondition is $x = x_i \wedge y = y_i$ and effects are generated as previously explained, is created. Note that no mode can switch to q_1.

Additionally, in case of the final point $p_f = (x_f, y_f)$ being in the region of q, mode q is connected to mode q_2 according to a jump whose preconditions are for (x, y) to be in a box centred in p_f, $\text{pre}_{q \rightarrow q_f}$: $x = x_f \wedge y = y_f$, and effects as before except for v_x and v_y which are nullified, $v_{x_2}^0 = 0 \wedge v_{y_2}^0 = 0$ (only for modes in which the flow conditions use the approximation). Note that q_2, the final mode, does not connect to any other mode, the system always stops here.

Now that every mode is generated and connected between each other and the initial condition and goals are defined, a `.drh` file can be generated and dReach can analyse the reachability problem for the hybrid system. This is done automatically with a script which generates the `.drh` file according to the specifications given in Section 2.4.

3.2.5 Divide and Conquer

As mentioned in Section 3.1, the other method used to tackle the solving time issue is to split the full problem into smaller, simpler sub-problems.

In order to do this, a graph is created, using the set of overlapping regions which result from the convex decomposition of the map followed by the expansion of the derived polygons, where each node is a region and each edge connects two nodes if it is possible to travel from one region to the other. Then, a path finding algorithm is used to find a sequence of nodes from the initial region, corresponding to the one in which the initial point is located, to the final, corresponding to the region in which the final point is located. This sequence is used to divide the problem into the smaller problems, which are then solved sequentially and solutions concatenated to get the motion plan of the full problem.

To create a sub-problem, each region is considered individually, along with anchor points in it located, and the same methods as before are used to generate the modes of the hybrid automaton. A `.drh` file is created and passed to dReach, in the very same way as previously mentioned, the solution is then used to define the next sub-problem's initial point. If no solution is found, then the path (from the initial to the final region) being currently evaluated is considered unfeasible and the next one is evaluated. If no other path exists, then the algorithm informs the user it was not able to find a solution to the problem.

The only mode which is defined differently from what has already been presented is the final mode, which, for the first $n-1$ sub-problems in a path with n steps, instead of being defined by a box centred on the final point, is defined by the overlap zone between the regions it is traversing, the transitions zone (the n-th sub-problem's final mode is the whole problem's final mode, thus, defined as before).

Consider a region i, with $0 < i < n$, a sub-problem p_i's final mode is defined by the overlap zone between region i and region $i+1$, i.e., a solution to ${}^{s}p_i$ is one which takes (x, y) from the initial point (given by the solution of the previous sub-problem p_{i-1}) to this zone, the actual point in the zone where the cargo ends up is given in the model computed by dReach, in case there is a solution. This final point becomes the initial point of the next sub-problem (p_{i+1}).

As an example consider the problem represented in Figure 3.20, and its convex decomposition (after polygon extension) in Figure 3.22.

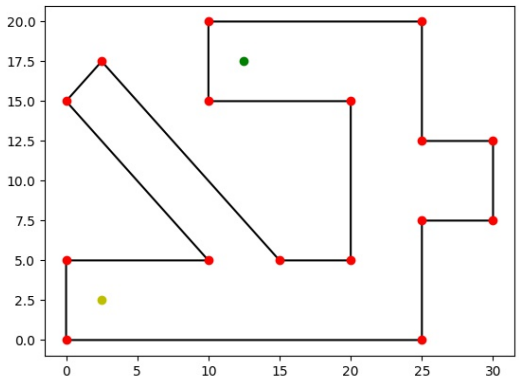

Figure 3.20: Visual representation of the more complex problem. Black lines represent the walls of the map, red dots the available anchor points, yellow and green dots the initial and final points, respectively.

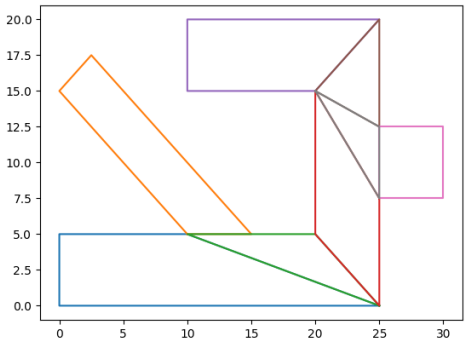

Figure 3.21: Convex decomposition of the complex map in Figure 3.20

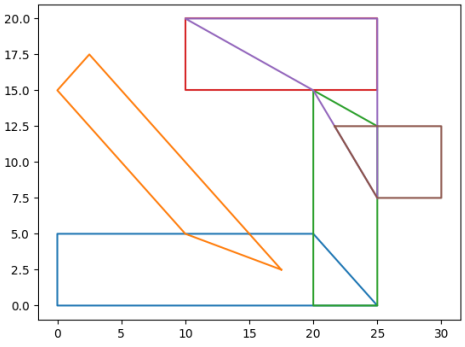

Figure 3.22: Convex decomposition of the complex map in Figure 3.20 with expansion of resulting polygons.

A sub-problem which aimed to carry a cargo from the initial point, which is inside the blue polygon in Figure 3.22, into the region defined by the green polygon in the same Figure would have as final condition the overlapping area between these two polygons, written in the `.drh` file as follows.

```
1 goal:
2 @2  ( and (y >= 0.0+ 0.2) and (x >= 20+ 0.2) and (y <= -1.0 * x + 25.0- 0.2));
```

Note that the conditions are derived from the equations which define the overlap area's border lines, with the inclusion of a constant term to guarantee the cargo is inside the next region in the path, and not on any of the actual borders. This avoids issues which arise from the fact that dReach checks for δ-decidability, that is, there is always a δ-sized error, which is not compatible with the rigidity of a

condition such as a point being in a border line (if the error leaves the cargo on the wrong side of the border, the next sub-problem will not be satisfiable because the initial conditions invalidate the invariants of every mode, the cargo would be outside of the region). This issue is the reason behind the necessity of polygon expansion after the convex decomposition of the map is computed. Overlap areas between the regions were needed in order to relax the goal conditions of each sub-problem.

The concatenation of every sub-problem's computed plan, with the same order in which the regions appear in the region path, leads to a plan which takes the cargo from the initial point to the final region.

4

Results and Analysis

The final methods were tested with two files, `simple.prb` and `complex.prb`, the former describes the problem in Figure 4.1 and the latter the one in Figure 3.20. The first tests were done still with the approximation to verify whether such approximation would be a feasible idea. In both problem files, the maximum number of robots is two.

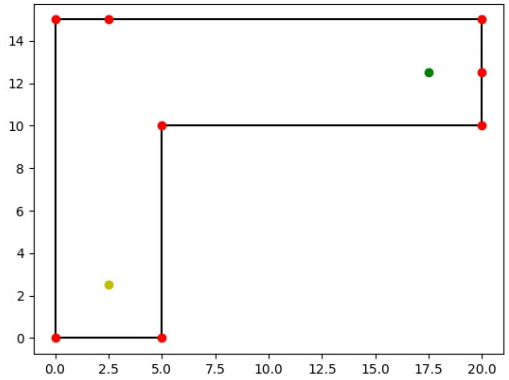

Figure 4.1: Problem described in `simple.prb`, the map is represented by the black lines, initial and final points are the yellow and green dots, respectively, and the red dots give the location of the anchor points.

The problem in `simple.prb` is solvable by the methods described in Chapter 3 when using the flow conditions approximation, and the solution is depicted in Figure 4.2, along with a table with the time duration of each step in Table 4.1. However, when testing file `complex.prb`, dReach could only find solutions where the cargo is pulled towards only one anchor point and passes from anchor to anchor until reaching the final box (the goal is a 5 by 5 box around the final point), this is depicted in Figure 4.3 and Table 4.2 has the time duration of each step.

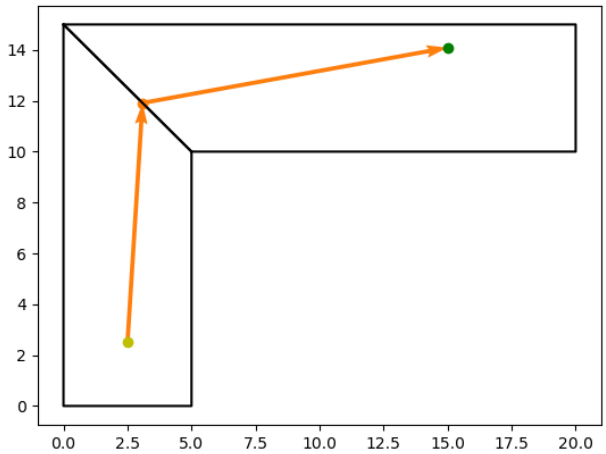

Figure 4.2: Solution path of problem `simple.prb`.

q_i	t_i	$Connections$
q_1	4.880s	$(0, 15) \wedge (5, 10)$
q_2	12.111s	$(20, 15)$

Table 4.1: Table with the time duration of each mode. q_i is the mode at step i of the solution of `simple.prb`

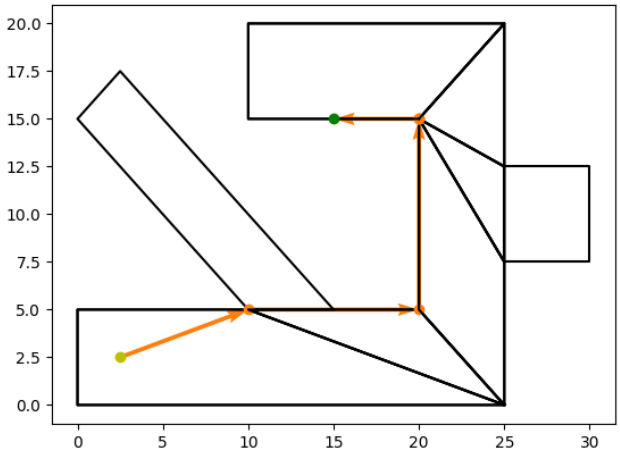

Figure 4.3: Solution path of problem `complex.prb` when the cargo is allow to get as close as it needs to the walls.

q_i	t_i	$Connections$
q_1	7.906s	$(10, 5)$
q_2	10.0s	$(20, 5)$
q_3	10.0s	$(20, 15)$
q_4	0.0s	$(20, 15)$
q_5	0.0s	$(20, 15)$
q_6	5.001s	$(15, 15)$

Table 4.2: Table with the time duration of each mode. q_i is the mode at step i of the solution of `complex.prb`.

Unfortunately, when the conditions were modified to force the distance from the cargo to the map borders to be greater than 0.1 (see end of Section 3.2), the methods here described were not able to solve this problem. Upon inspection of a proof file generated by dReal (when specified in its running options) whenever it gives an `unsat` result, it was found that the reason this search fails is because the solver does not accept the initial conditions of the sub-problem. Note that the fact that the result was `unsat` only means that a solution was not found, this is not wrong necessarily, but in this case there was a solution for the problem and the result from the solver should have been δ-`sat`, accompanied by a model.

Specifically, the issue is that the sub-problem corresponding to the blue corridor in Figure 4.4 ends in the point represented by the orange dot in the same figure, which should belong to the border line,

with an error smaller than δ (due to the nature of the solver, see end of subsection 3.2.5). The next sub-problem, associated with the corridor in magenta, uses this end point as its initial point. dReach does not accept this initial point, because, due to this δ-sized error, its value conflicts with the constraint that represents the border line between both corridors ($y \geq -\frac{1}{3}x + \frac{25}{3}$). Since this issue occurs in the initial conditions, the problem is invalid from the start and the solver does not find a solution.

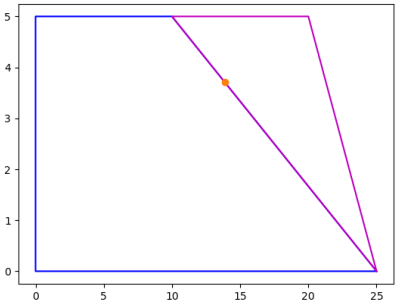

Figure 4.4: Figure to help describe the issue in solving problem `complex.prb`.

Note that, according to Table 4.2, two of the modes have null duration. This is because the cargo is at the corner point shared by the corridors from q_3, q_4, q_5 and q_6. The system has to go through q_4 and q_5 to get to q_6 from q_3 but when going to q_4 it positions the cargo in the point which unites all border lines it has to travel through, thus, when going from q_4 to q_5, it is already in the border line so no time is needed, the same happening from q_5 to q_6.

Additionally, there is an error associated to the approximation to the flow conditions mentioned in subsection 3.2.4 and used in the aforementioned tests. Figure 4.5 illustrates this error for the first mode of the solution depicted in Figure 4.2. It is possible (and expected) to see that in the beginning the two trajectories are acceptably approximate, but they start diverging and the approximation error becomes too great to be ignored.

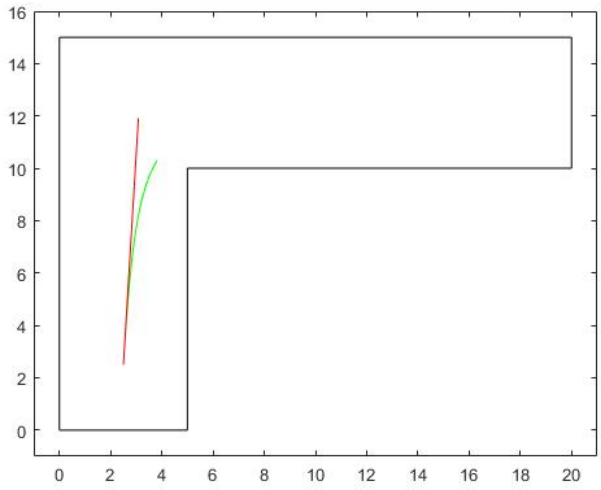

Figure 4.5: Difference between approximated trajectory (red) and real trajectory (green) for mode q_1 of the solution in Figure 4.2.

To solve this issue, an enhancement of the approximation method was analysed, refer to Chapter 3 for that analysis.

When this extension was applied to the simple problem in Figure 4.1, a solution was obtained both when solving the whole problem and when using divide and conquer, resulting in the solutions depicted in Figures 4.6 and 4.7 with information about the duration of each mode as well as which anchor points are being used presented in Tables 4.3 and 4.4, respectively.

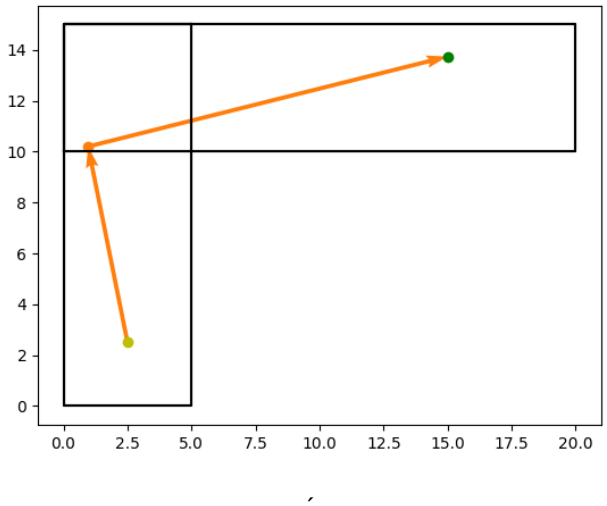

Figure 4.6: Solution when using divide and conquer **without** extended polygons on simple map.

q_i	t_i	$Connections$
q_1	1.571s	$(0.0, 15.0)$
q_2	2.896s	$(20.0, 15.0)$

Table 4.3: Table with the time duration of each mode, as well as connection points used. q_i is the mode at step i of the solution of `simple.prb` depicted in Figure 4.6

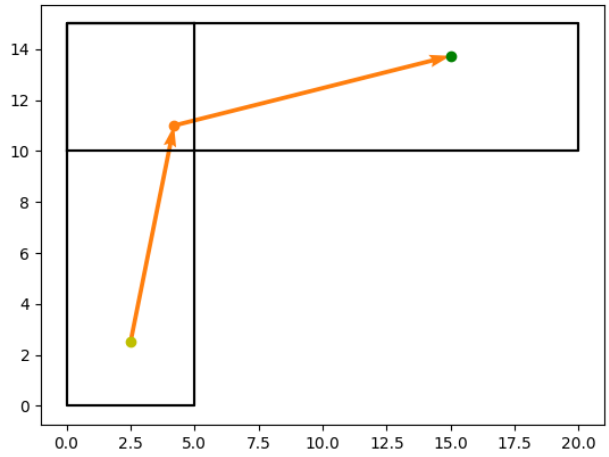

Figure 4.7: Solution when using divide and conquer **with** extended polygons on complex map.

q_i	t_i	$Connections$
q_1	1.734s	$(5.0, 15.0)$
q_2	2.228s	$(20.0, 15.0)$

Table 4.4: Table with the time duration of each mode, as well as connection points used. q_i is the mode at step i of the solution of `simple.prb` depicted in Figure 4.7

This problem was possible to be solved by the two methods, with and without ”divide and conquer”, taking approximately 13 minutes and 8 seconds (788 seconds) when solving the full problem at once and approximately 5 seconds when using the ”divide and conquer” method. This points to a decrease of approximately 99.27% in the resolution time.

In the case of the complex problem in Figure 3.20, extending the polygons generated by the convex decomposition of its map resulted in the set of polygons in Figure 3.22. Figure 4.9 depicts the solution of the problem as computed using the combination of divide and conquer and extended polygons, with information about the duration of each mode as well as which anchor points are being used presented in Table 4.5.

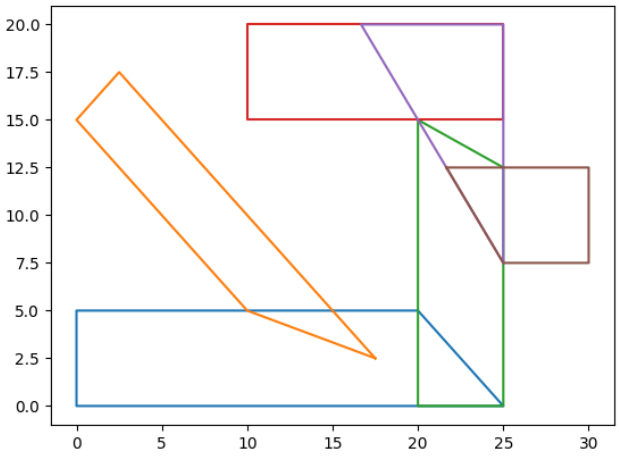

Figure 4.8: Fixed convex decomposition and polygon extension of the map from Figure 3.20.

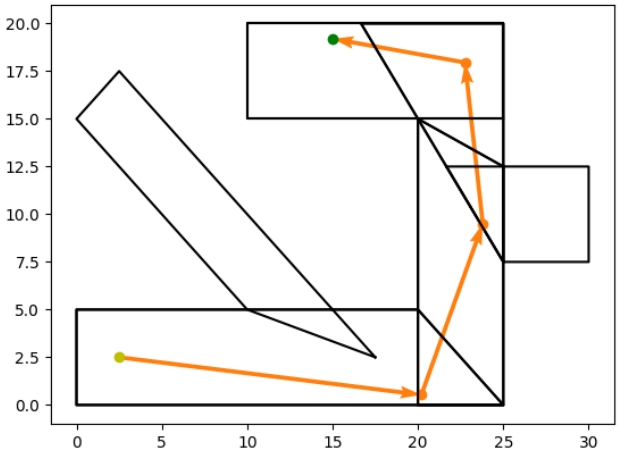

Figure 4.9: Solution for the problem from Figure 3.20, computed using the ”divide and conquer” in combination with overlapping convex polygons.

q_i	t_i	$Connections$
q_1	17.810s	$(25.0, 0.0)$
q_2	9.664s	$(25.0, 12.5)$
q_3	8.515s	$(22.5, 20.0)$
q_4	7.853s	$(10.0, 20.0)$

Table 4.5: Table with the time duration of each mode, as well as connection points used. q_i is the mode at step i of the solution of `complex.prb` depicted in Figure 3.20

This final combination of methods is finally able to solve the complex problem, taking approximately

7 seconds to do it.

5

Conclusion

Contents

This chapter includes a few final notes, then the contribution of this thesis towards this field is given, followed by some recommendations for future work to enhance the capabilities of the methods here presented.

This dissertation tries to start the development of methods for enabling a fleet of robots to successfully plan the transportation of cargo, from its drop off point to its designated place. It hypothesises that this can be achieved using a hybrid representation and a δ-reachability problem solver, dReach, both of which are explained in Chapter 2. As such, the main problem can be seen as a motion planning problem where the object whose motion is being planned is the cargo, the robots are the means through which the cargo moves, thus, deciding how the cargo moves implies deciding what each robot should do (to which anchor points each should be connected) in order to force that to happen. This problem formulation is thoroughly described in Chapter 3.

5.1 Contribution

The problem of motion planning for micro-gravity conditions is, at the time of writing this thesis, relatively unexplored. As such, the work here presented is meant to be seen as closer to a proof of concept than to an actual solution for how to handle these types of problems.

The results presented and analysed in Chapter 4 show the current tools, when allied with the right models, may be ready for these types of challenges. Particularly, the combination of the ”divide and conquer” method (see Section 3.2) in combination with convex decomposition of the map with overlapping of polygons shows much promise to anyone who cares to follow this work, having been able to reduce the solving time very significantly already, in some cases this being critical to the decidability of the problem.

However, the reader should take note of the simplicity of the model, it does not consider such things as inertia, nor obstacle handling, both very important in this field of research. This is a start, but there is still a long way to go.

Further work is needed, both in the model and solver sides, before this type of planners can be used. As such continuing to try and use methods like the ones presented here and refine them to become more robust and consider a wider range of scenarios would be effort well spent in this steep uphill hike for knowledge.

5.2 Future Work

There are a lot of other spins to solving this problem which can be derived from the work in this thesis and can potentially refine it.

Firstly, the decomposition of the map in regions could be made in a different way, in which the visibility polygons for each anchor point are computed and the map is divided into zones according to which anchor points are visible where, or even according to combinations of anchor points, given the maximum number of robots. For instance, a map of a problem in which a maximum of 2 robots may be used, can be divided into each visibility polygon, for modes with 1 robot, and every intersection between every pair of polygons, for modes in which two anchor points are used. The divisions will be overlapping and this may cause an explosion in the state space of the hybrid automaton but the methods in Section 3.2 should be able to counter the effects of this explosion.

This actually solves an issue, where there is an overlap zone between two corridors but there is not any connection point capable of pulling the cargo into this zone, in fact, an example of this can be observed in Figure 3.22, if a cargo was to be transported from the orange corridor into the blue corridor, it would fail to do that, as the whole of its overlap area is in the middle of the blue corridor. There can only be anchor points in the walls of the map, so there would not be any anchor point able to pull a cargo from the orange corridor into its overlapping zone with the blue corridor, which would also belong to the orange corridor. Due to this issue, these methods may not be complete and there may be solutions which are ignored due to the limitations of the model in what regards to fully expressing the problem. This creates a need to continue to develop the model.

In this specific case, this happens due to the fact that, when expanding the yellow polygon in Figure 3.21, it is expanded into the green one in the same Figure. Since this green one ends up being ”swallowed” by the expansion of the blue polygon, the former should be replaced by the latter when expanding the yellow polygon. This would result in the yellow polygon extending all the way to the map border, acquiring a bit of that border as its own. Then it would only be a case of making sure that that piece of border had anchor points.

That, in its place could be solved by having anchor points placed at certain intervals in the map walls, or planning the anchor point positions after having the full map, and having put it through the whole transformation. And still, there would be situations where this would not be avoidable and would have to be handled in another way. By having the map be divided into visibility polygons of the anchor points this issue could be avoided altogether. Of course this would inevitably come with its own troubles, but it is always worth investigating, there is always something to learn.

The solver could be improved in order to better handle these kinds of flow equations, reducing the amount of time it take to solve a full problem without requiring the divide and conquer method.

There is also a heuristics which could be used to filter through the possible modes, enabled by using the ”divide and conquer” method. Since a sub-problem takes place in a small, convex polygon, the path from the initial point to the goal area, will most likely already start with a direction pointing, to a certain angle, to such area. Thus, the initial vectors of movement for each mode can be checked and the modes

filtered based on this condition.

Additionally, the robots could save each solution and compare each new problem with its database. This could save time in a situation where some initial part of the plan for the new problem coincides with a part of an old problem. This part of the old plan can be “reused” in the new plan, and the decision making process for those steps skipped, saving a portion of time.

Finally, subsequent models should work in an effort to recreate the real conditions the system will be subject to, something this work does not take into account, since it is an early sketch of a possible tool. As such, it is crucial that some things are taken in consideration, the flow equations should account for mass, drag, inertia and other properties of motion in micro-gravity and the model should evolve to three-dimensional space. The mechanics and logistics of switching between anchor points, whichever way it is decided to happen, as well as how much time this switch takes are also important aspects to consider.

Bibliography

[1] S. Kong, S. Gao, W. Chen, and E. M. Clarke, "dReach: δ-Reachability Analysis for Hybrid Systems," in *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, ser. Lecture Notes in Computer Science, C. Baier and C. Tinelli, Eds., vol. 9035. Springer, 2015, pp. 200–205.

[2] S. Gao, S. Kong, and E. M. Clarke, "dReal: An SMT Solver for Nonlinear Theories over the Reals," in *Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings*, ser. Lecture Notes in Computer Science, M. P. Bonacina, Ed., vol. 7898. Springer, 2013, pp. 208–214.

[3] M. Ghallab, D. S. Nau, and P. Traverso, *Automated Planning and Acting*. Cambridge University Press, 2016.

[4] S. Gao, S. Kong, W. Chen, and E. M. Clarke, "δ-Complete Analysis for Bounded Reachability of Hybrid Systems," *CoRR*, vol. abs / 1404.7171, 2014.

[5] R. Alur, C. Courcoubetis, T. A. Henzinger, and P. . H. Ho, "Hybrid Automata: An Algorithmic Approach to the Specification and Verification of Hybrid Systems," in *Hybrid Systems*, ser. Lecture Notes in Computer Science, R. L. Grossman, A. Nerode, A. P. Ravn, and H. Rischel, Eds., vol. 736. Springer, 1992, pp. 209–229.

[6] C. Barrett, A. Stump, and C. Tinelli, "The SMT-LIB Standard: Version 2.0," in *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*, A. Gupta and D. Kroening, Eds., 2010.

[7] S. Gao, S. Kong, and E. M. Clarke, "Satisfiability modulo ODEs," in *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*. IEEE, 2013, pp. 105–112.

[8] J. A. Wolfe, B. Marthi, and S. J. Russell, "Combined Task and Motion Planning for Mobile Manipulation," in *Proceedings of the 20th International Conference on Automated Planning and Scheduling,*

ICAPS 2010, Toronto, Ontario, Canada, May 12-16, 2010, R. I. Brafman, H. Geffner, J. Hoffmann, and H. A. Kautz, Eds. AAAI, 2010, pp. 254–258.

[9] N. T. Dantam, Z. K. Kingston, S. Chaudhuri, and L. E. Kavraki, "Incremental Task and Motion Planning: A Constraint-Based Approach," in *Robotics: Science and Systems XII, University of Michigan, Ann Arbor, Michigan, USA, June 18 - June 22, 2016*, D. Hsu, N. M. Amato, S. Berman, and S. A. Jacobs, Eds., 2016.

[10] S. Nedunuri, S. Prabhu, M. Moll, S. Chaudhuri, and L. E. Kavraki, "SMT-based synthesis of integrated task and motion plans from plan outlines," in *2014 IEEE International Conference on Robotics and Automation, ICRA 2014, Hong Kong, China, May 31 - June 7, 2014*. IEEE, 2014, pp. 655–662.

[11] D. Bryce, S. Gao, D. J. Musliner, and R. P. Goldman, "SMT-Based Nonlinear PDDL+ Planning," in *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA*, B. Bonet and S. Koenig, Eds. AAAI Press, 2015, pp. 3247–3253.

[12] J. Fernández, B. Tóth, L. Cánovas, and B. Pelegrín, "A Practical Algorithm for Decomposing Polygonal Domains Into Convex Polygons by Diagonals," *Top*, vol. 16, no. 2, p. 367–387, 2008.

A

Problem Files

A.1 Simplest problem

The following `simple_case.drh` file was generated to describe the problem modeled in 3.1.

```
#define w 1.0
[0.0, 20.0] x;
[0.0, 15.0] y;
[0, 100.0] time;

{ mode 1;

        invt:
                (x = 2.5);
                (y = 2.5);
        flow:
                d/dt[x] = 0;
                d/dt[y] = 0;
        jump:
                (and (x > 0.0) (and (y < 15.0) (and (x < 5.0) (y > 0.0)))) ==> @3 (and (x' = x) (y' = y));
}
{ mode 2;
```

```
18
19          invt:
20                  (x = 17.5);
21                  (y = 12.5);
22          flow:
23                  d/dt[x] = 0;
24                  d/dt[y] = 0;
25          jump:
26  }
27  { mode 3;
28
29          invt:
30                  (x > 0.0);
31                  (y < 15.0);
32                  (x < 5.0);
33                  (y > 0.0);
34          flow:
35                  d/dt[x] = 0.0;
36                  d/dt[y] = 2.0;
37          jump:
38                  (and (x > 0.0) (and (y < 15.0) (and (x < 20.0) (y > 10.0)))) ==> @4 (and (x' = x) (y' = y));
39  }
40  { mode 4;
41
42          invt:
43                  (x > 0.0);
44                  (y < 15.0);
45                  (x < 20.0);
46                  (y > 10.0);
47          flow:
48                  d/dt[x] = 2.0;
49                  d/dt[y] = 0.0;
50          jump:
51                  (and (x = 17.5) (y = 12.5)) ==> @2 (and (x' = x) (y' = y));
52  }
53  init:
54  @1          (and (x = 2.5) (y = 2.5));
55
56  goal:
57  @2          (and (x = 17.5) (y = 12.5));
58
```

A.2 Complex Problem

A more complex problem was created as to test the robustness of this method. Figure 3.20 represents this problem.

The input `.prb` file which describes this problem is as follows.

```
1  w 1
2  x [0, 30]
```

```
y [0, 20]
vx [0, 30]
vy [0, 20]
tmax 1000
n_bots 1
init (2.5, 2.5)
final (12.5, 17.5)

Polygon {
        (0, 5)
        (10, 5)
        (0, 15)
        (2.5, 17.5)
        (15,5)
        (20, 5)
        (20, 15)
        (10,15)
        (10,20)
        (25,20)
        (25, 12.5)
        (30,12.5)
        (30,7.5)
        (25,7.5)
        (25,0)
        (0,0)
}
Connection Points {
        (0, 5)
        (10, 5)
        (0, 15)
        (2.5, 17.5)
        (15,5)
        (20, 5)
        (20, 15)
        (10,15)
        (10,20)
        (25,20)
        (25, 12.5)
        (30,12.5)
        (30,7.5)
        (25,7.5)
        (25,0)
        (0,0)
}
END
```

B

Flowcharts

B.1 convex decomposition

The convex decomposition algorithm can be described by the following flowcharts.

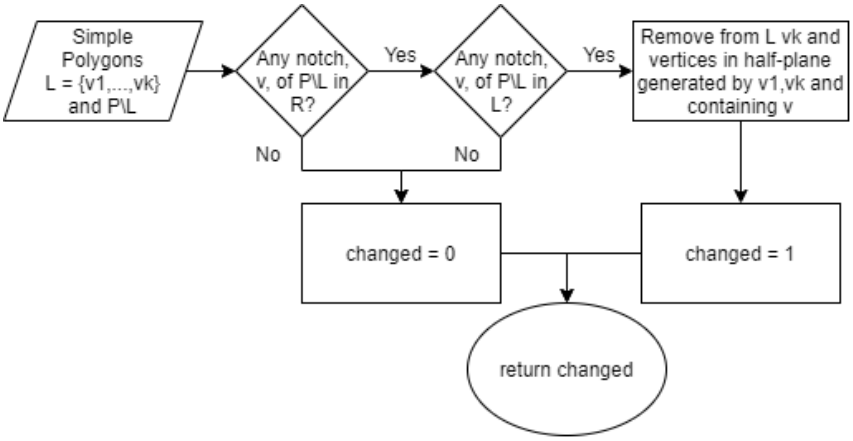

Figure B.1: Flowchart for the algorithm that checks the validity of L and corrects it when needed.

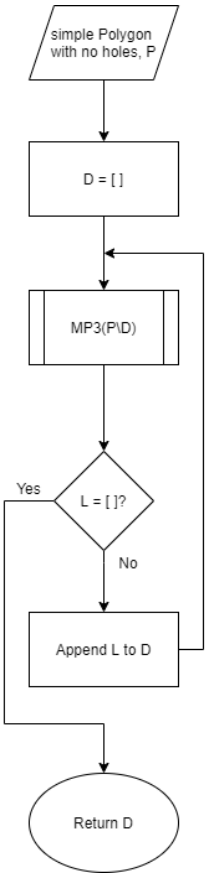

Figure B.2: Flowchart of the convex decomposition algorithm.

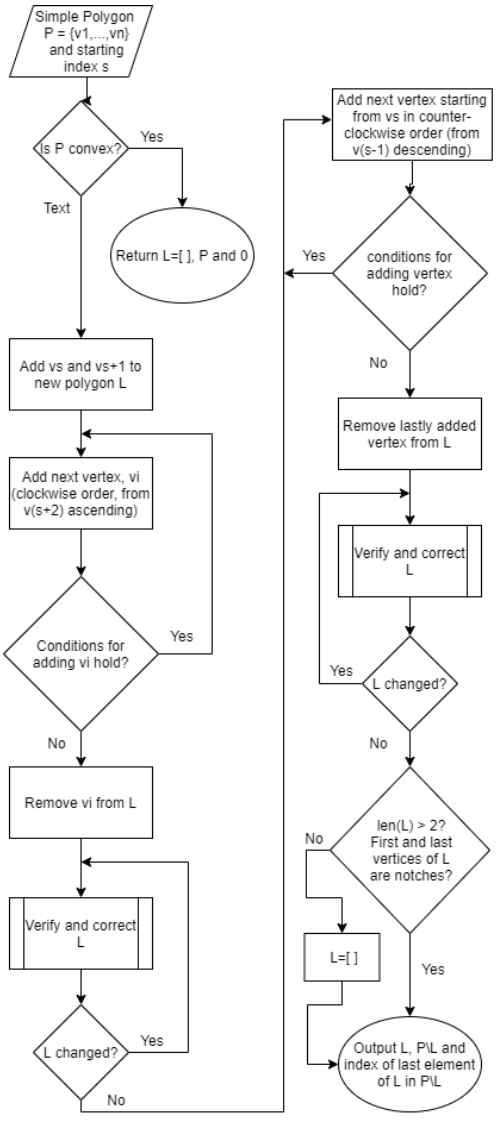

Figure B.3: Flowchart of the MP# algorithm.