Learning to make mixed-integer robot motion planning easy

Tanguy Lewko

Semester project

Under the supervision of:
Dr. Tony A. Wood, Tingting Ni, Prof. Maryam Kamgarpour

20/01/2023

sycamore lab
SYSTEMS CONTROL AND MULTIAGENT OPTIMIZATION RESEARCH

École Polytechnique Fédérale de Lausanne (EPFL)
1015 Lausanne, Switzerland

Abstract

Robot motion planning involves finding a feasible path for a robot to follow while satisfying a set of constraints and optimizing an objective function. This problem is critical for enabling robots to navigate and perform tasks in real-world environments. However, it can become challenging, especially when the environment is cluttered or uncertain, or when the task requirements are complex.

A range of approaches have been developed to tackle the robot motion planning problem. Mixed Integer Programming (MIP) is one of them. MIP is an optimization technique consisting of optimising an objective function subject to constraints, where some of the variables are integer valued. This framework is well-suited for finding feasible paths for robots because it allows to incorporate logical constraints such as avoiding obstacles and also continuous constraints such as robot's dynamics. However, it can be computationally intensive and may not scale well to large or complex environments.

To address these limitations, we propose a two-pronged approach. First, we explore the use of machine learning techniques to learn a part of the solution, in order to reduce the size of the MIP problem and improve its scalability. We condlude that state-of-the-art approaches are not directly applicable in a realistic robot motion planning setup. Second, we adopt a receding-horizon approach, in which the global problem is broken down into a series of local problems, each of which can be solved more efficiently. Machine learning techniques are used to learn the solutions of these local problems.

We demonstrate the effectiveness of this novel approach through a series of experiments on a range of robot motion planning problems. Our results show that our approach can significantly reduce the computation time of the MIP solution, and make it more scalable to large and complex environments.

Contents

Symbol table

Latex Symbol	Description
p	position vector $[m]$
p_0, p_T	initial and final position of the robot $[m]$
x, y	x and y coordinates of the robot $[m]$
v	velocity vector $[m/s]$
v_0, v_T	initial and final velocities of the robot $[m/s]$
a	acceleration vector $[m/s^2]$
T	Horizon
τ	Step time $[s]$
A	matrix of coefficients of the variables in linear constraints
b	vector of constants of linear constraints
M	big M value
δ	vector of binaries in big-M constraints
N	number of obstacles
K_i	number of edges of obstacle i
I	identity matrix
θ	optimization parameters
n_δ	number of binary variables
G	Good-Turing estimator
$I(\theta)$	optimal integer total strategy
$S_i(\theta)$	individual strategy related to obstacle i
$S(\theta)$	reconstructed total strategy
n_{probs}	number of problems
n_{evals}	number of binary strategies evaluated
T_M	set of tight big-M constraints
C_{vis}	cost-to-go from a vertex to the goal point
x_{vis}	waypoint robot is aiming in a receding horizon framework
K_v	number of obstacle vertices
K_{cand}	number of waypoint candidates
v_j	coordinates of vertex j $[m]$
d_j	distance from vertex j to goal $[m]$
x_{end}	last planned point of the trajectory
β	vector of binaries for waypoint choice
S	segment joining two vertices
$\mathbf{O}$	domain of all points within obstacles

Chapter 1

Introduction

Mixed integer programming (MIP) problems are a type of optimization problem that involves both continuous and discrete variables. Mixed Integer Programming is a well-suited method for trajectory optimization, as it allows incorporating logical constraints such as avoiding obstacles and also continuous constraints such as robot's dynamics. However, MIPs are not used a lot in real-world robotics due to slow computations times (typically from seconds to minutes) and non embeddable algorithms (best solvers rely on complex multithreaded implementations) [5].

There are several reasons why solving MIP problems quickly is important for robotic motion planning. First, faster solutions can enable robots to react more quickly to changes in their environment, such as obstacles or changes in their goal. This can help to improve the safety and reliability of the robot's actions. Second, faster solutions can enable robots to make more efficient use of their resources, such as battery power or computational resources. This can help to extend the operational lifetime of the robot and improve its overall performance. Finally, faster solutions can enable robots to solve more complex motion planning problems, which can expand the capabilities of the robot and allow it to tackle a wider range of tasks.

In Chapter 2, we discuss different works which introduced methods to make mixed integer motion planning problems easier to solve. In Chapter 3, we define the optimization problem we try to solve. Then, in Chapter 4, we implement and discuss a state-of-the-art method using machine learning to reduce the online complexity of the optimization problem. In Chapter 5, we develop a receding horizon algorithm which ensures that the robot will reach the goal and reduces problem's complexity. Finally, in Chapter 6, we combine the learning approach and the receding horizon approach, which helps us to extend the MIP framework to larger problems, both in number of obstacles and horizon length.

Chapter 2

Related work

This chapter helps to contextualize the contribution of this work within the existing body of knowledge. We divide this review in three parts. The first one relates to the use of MIP to solve motion planning problems. The second one resumes the different machine learning approaches used to reduce the computation time. Finally we also investigate the existing iterative algorithms which address the issues of MIP coping with large scale models.

2.1 MIP for motion planning problems

In [8], the authors evaluated the state-of-the-art for MIP-based motion planning. This work gives a good overview of the field, and resumes how to model with MIP task assignment, path planning and obstacle/collision avoidance. The authors note that there is a tradeoff between conservativeness and complexity in non-convex modeling representation. They also point out the fact that only small size problems can be solved in real-time, and that it is important to formulate the problems in a compact way.

Work [9] formulates the trajectory optimization of an aircraft in a MIP form, including the dynamics model, collision avoidance and multiple waypoint extension.

2.2 Machine learning approaches

In [4], a data-driven algorithm is developed to quickly find solutions for Mixed Integer Convex Programs (MICPs). The algorithm, named CoCo, consists of a two-stage approach. During the offline phase, a neural network classifier is trained to map problems parameters to the integer solution (the optimal assignement of integer variables). Online, a forward pass on the classifier allows to obtain the integer solution of the MICP and then to solve it as a convex optimization problem. They showed that CoCo finds nearly optimal solutions to motion planning MICPs with 1 to 2 orders of magnitude solution speedup compared to other approaches.

2.3 Iterative MIP algorithms

In [7], the authors consider trajectory-generation problems with obstacle avoidance requirements and reduce the computational effort required to solve them by developing an iterative algorithm. An approach for autonomous fixed wings performing large scale maneuvers is described in [1]. It designs nearly minimum time planar trajectories to a goal, with no fly zones and vehicle's maximum speed and turning rate. MIP is used over a receding planning horizon both to reduce the computational effort and introduce feedback. A method to avoid entrapment behind obstacles is also developed and the approach has been shown to work on large trajectory optimization problems, which is not the case of fixed horizon controllers.

[10] builds up on this approach and apply it for a truck use case in which only nearby obstacles can be detected and shows that this method can control a real vehicle in an uncertain environment. A control architecture is developed, allowing the online planner to compensate for obstacle discovery beyond the execution horizon while low-level feedback rejects vehicle disturbance within that horizon.

Chapter 3

Mixed Integer Programming (MIP) problem for path planning

3.1 Dynamics

The dynamics are considered to be linear and time invariant. We start by considering simple planar dynamics to solve the problem : single and double integrator dynamics.

By discretizing single integrator dynamics with a sample time τ, we obtain the following state equations, where input v is velocity, t denotes a time-step, and x and y describe the robot's position:

$$\begin{bmatrix} x_{t+1} \\ y_{t+1} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_t \\ y_t \end{bmatrix} + \begin{bmatrix} \tau & 0 \\ 0 & \tau \end{bmatrix} \begin{bmatrix} v_{x_t} \\ v_{y_t} \end{bmatrix} \tag{3.1}$$

Single integrator dynamics are a simple and intuitive model for the motion of a point mass. They are easy to analyze and can be used as a starting point for more complex dynamics models. However, they have some limitations, such as the inability to model acceleration or deceleration directly. In many cases, double integrator dynamics, which include a state for acceleration, are a more appropriate model for the motion of a point mass.

For double integrator dynamics we have :

$$\begin{bmatrix} x_{t+1} \\ y_{t+1} \\ v_{xt+1} \\ v_{y_{t+1}} \end{bmatrix} = \begin{bmatrix} 1 & 0 & \tau & 0 \\ 0 & 1 & 0 & \tau \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_t \\ y_t \\ v_{xt} \\ v_{y_t} \end{bmatrix} + \begin{bmatrix} \tau^2 & 0 \\ 0 & \tau^2 \\ \tau & 0 \\ 0 & \tau \end{bmatrix} \begin{bmatrix} a_{x_t} \\ a_{y_t} \end{bmatrix} \tag{3.2}$$

where a_x and a_y are input accelerations.

3.2 Constraints

The map used to evaluate the implementation is a rectangle measuring 3.5 meters in width and 1.6 meters in height. It contains 22 obstacles, which are all rectangles measuring 5 centimeters in width and varying in length. These obstacles can be tilted. This laboratory map is displayed on Figure 3.1, where the robot, a JetBot powered by Jetson Nano, is circled in red. The robot is shown on Figure 3.2.

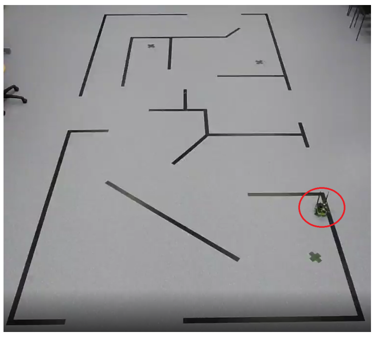


Figure 3.1: Laboratory map

Figure 3.2: Jetbot

Obstacles are represented as lines between two points that require dilating by the sum of half the obstacle's width (2.5 centimeters) and the robot's radius (20 centimeters). As depicted in Figure 3.3, the dilatation of these lines is illustrated using red lines and the dilated obstacles are represented by grey rectangles. The largest rectangle in the figure represents the limit of the map. The method used to dilate the obstacles is conservative, as the corners of the obstacles are not rounded. Additionally, we do not merge multiple obstacles if they intersect; instead, we model them as separate entities. This approach has the advantage of leaving the obstacles as convex polygons, but it may prevent us from achieving truly optimal solutions, such as in the shortest path problem. Constraints are more easily formulated with polygonal obstacles, as we can define the constraint by half planes.

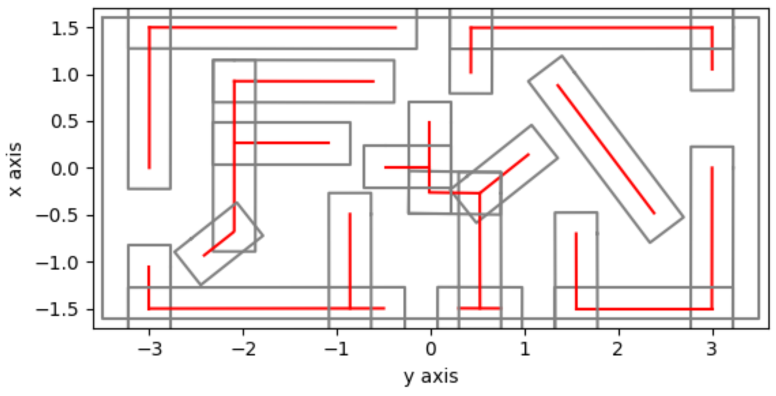


Figure 3.3: Dilatation of the obstacles

In order to determine the linear constraints for staying within the dilated obstacles, which are convex polygons, the corresponding vertices were utilized as references. Although the result is the same, deriving the constraints from the vertices of the dilated polygons is more straightforward than defining the half planes directly. These constraints can be represented in matrix form for a single obstacle as $Cp \leq d$, where p is the position vector of the robot, C is a $K \times 2$ matrix, with K being the number of edges of the obstacle, and d is a vector of size K. This equation gives us a set of K constraints that must be simultaneously respected in order to stay within an obstacle. The inside of the obstacle is defined by K half planes. To stay outside of the polygons, the equations for the opposite half planes are defined in matrix form as $Ap \leq b$, with $A = -C$ and $b = -d$. Only one of these constraints needs to be effective in order to stay outside of a polygonal obstacle.

To incorporate the fact that only one constraint needs to be respected, we employ the big-M method as suggested by Bertsimas et al. in their work [2]. This method results in the following constraints for each obstacle i, where N is the number of obstacles, K_i is the number of edges of obstacle i, δ_i is a vector of K_i binary variables, and M is a number chosen arbitrarily high.

$$A_i p \leq b_i + M\delta_i \qquad i = 1, ...N \quad t = 0, ..., T-1 \tag{3.3}$$

$$\sum_{k=0}^{K_i} \delta_{i_k} \leq K_i - 1 \qquad i = 1, ..., N \quad t = 0, ..., T-1 \tag{3.4}$$

$$\delta_{i_k} \in \{0, 1\} \qquad k = 1, .., K_i \quad i = 1, ..., N \quad t = 0, ..., T-1 \tag{3.5}$$

The constraint for staying on the outside half plane defined by edge k is enforced by the equation $A_{i_k} p \leq b_{i_k}$, where A_{i_k} is the k-th line of matrix A_i, and b_{ik} is the k-th component of vector b_i. The vector of K_i binary variables δ_i encodes the fact that this constraint can either be enforced or violated, as long as one of the constraints for obstacle i is enforced. This means that one component of δ_i must be zero. If δ_{i_k} is equal to one, the constraint is relaxed, and if it is equal to zero, it is enforced.

Other constraints need to be specified such as :
- Input is bounded : $u_t \in [\underline{u}, \overline{u}] \qquad t = 0, ..., T-2$
- Position and speed are bounded : $p_t \in [\underline{p}, \overline{p}], \quad v_t \in [\underline{v}, \overline{v}] \qquad t = 0, ..., T-1$
- Initial state of the robot : $p_0 = p^{init}, \quad v_0 = v^{init}$
- Ending state of the robot : $p_T = p^{end}, \quad v_T = v^{end}$

3.3 Objective function

The objective function, or cost, can be any convex quadratic function of inputs and states. We list here a few objective functions we used for experiments :

- **Shortest path for single integrator dynamics**
  Minimizing the length of the path requires minimizing the sum of the squared distances traveled between all time steps.

$$\text{minimize} \sum_{t=1}^{T-1} (x_t - x_{t-1})^2 + (y_t - y_{t-1})^2 \tag{3.6}$$

As for single integrator dynamics we have $v_{x_t}\tau = (x_t - x_{t-1})$ and $v_{y_t}\tau = (y_t - y_{t-1})$ where τ is the time step, equation 3.6 is equivalent to:

$$\text{minimize} \sum_{t=0}^{T-2} v_{x_t}^2 + v_{y_t}^2 \tag{3.7}$$

- **Minimizing fuel consumption [6]**

$$\text{minimize} \sum_{t=0}^{T-2} (|a_{x_t}| + |a_{y_t}|) \tag{3.8}$$

- **Minimizing distance to a desired position while expending minimal control effort for double integrator dynamics**

$$\text{minimize} \sum_{t=0}^{T-1} (x_t - x_{des})^2 + (y_t - y_{des})^2 + \gamma(a_{x_t}^2 + a_{y_t}^2) \tag{3.9}$$

3.4 Optimization problem

The motion planning problem for shortest path and single integrator dynamics can be written as :

$$\begin{array}{llrrr}
\text{minimize} & \sum_{t=0}^{T-2} v_{x_t}^2 + v_{y_t}^2 & & & \\
\text{subject to} & p_{t+1} = Ip_t + \tau I v_t & & & t = 0, ..., T-2 \\
& A_i p \leq b_i + M\delta_i & & i = 1, ..., N & t = 0, ..., T-1 \\
& \sum_{k=1}^{K_i} \delta_{i_k} \leq K_i - 1 & & i = 1, ..., N & t = 0, ..., T-1 \\
& \delta_{i_k} \in \{0, 1\} & k = 1, .., K_i & i = 1, ..., N & t = 0, ..., T-1 \\
& p_0 = p^{init}, \quad v_0 = v^{init} & & & \\
& p_T = p^{end}, \quad v_T = v^{end} & & & \\
& p_t \in [\underline{p}, \overline{p}], \quad v_t \in [\underline{v}, \overline{v}] & & & t = 0, ..., T-1
\end{array} \tag{3.10}$$

where I is the identity matrix, and other variables have been defined earlier. We will use this problem formulation for testing our methods.

We use Gurobi optimizer to find the solution of this optimization problem using branch-and-bound algorithm. A resulting trajectory can be seen in Figure 3.4. The objective function to minimize and the system dynamics can easily be changed to suit our needs.

As an example, in figure 3.4, we defined the horizon as T = 36, the time step $\tau = 0.6s$, we constrained x and y velocities between 0.2 and -0.2m/s and Gurobi took 0.03 seconds to solve the problem. In this optimization problem, we have 72 quadratic objective terms, 216 continuous variables, and 3168 binary variables. Same horizon with double integrator dynamics increases that time by two orders of magnitude for the same problem. This time may vary a lot depending on the problem, as MIP is a NP-hard problem.

Number of integers (binaries) in the motion planning problem is equal to the numbers of obstacles times the number of edges of the obstacle times the horizon. For T = 36, the number of binaries is then 22(obstacles) × 4(number of edges) × 36(time steps) = 3168(binaries). Increasing the horizon is very costly because each time step increases the number of integers by 88.

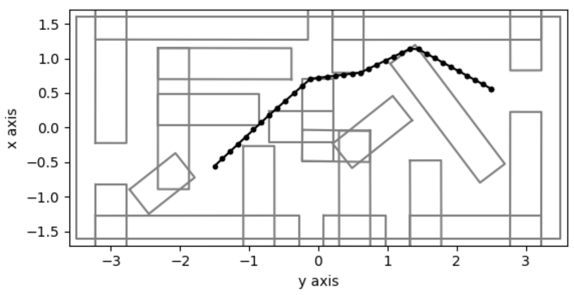


Figure 3.4: MIP problem 3.10 solved with Gurobi

3.5 Application to online motion planning

The application of MIP to online motion planning is usually not possible, as the optimization problems typically must be solved within a few milliseconds in order to control the system dynamics [2]. Indeed, the condition for real-time motion planning is that the inputs are frequent enough to control the system dynamics. That is why we intend to review the existing solutions and try new ones to be able to solve the MIP problems for motion planning much faster.

Our approach will be as follows. As the computational bottleneck is due to the exponential increase of binary variables with the size of the problem -which is defined by $T \times N \times K$, where T is the horizon, N is the number of obstacles and K is the number of edges per polygonal obstacle-, we will try to learn the optimal binary sequences for our problem using a neural network. To do so, we will sample space to solve the problem for a lot of starting and store the optimal solutions obtained (positions, inputs of the system, binary sequence, time to solve, cost).

Chapter 4

Learning Mixed Integer Convex Optimization (MICO) for motion planning

4.1 Learning

Our MIP problem is defined as :

$$\begin{array}{ll} \text{minimize} & f_0(x,\delta;\theta) \\ \text{subject to} & f_i(x,\delta;\theta) \leq 0, \quad i = 1,...,m_f \\ & \delta \in \{0,1\}^{n_\delta} \end{array} \tag{4.1}$$

where δ are binaries associated to logical constraints, x is a continuous variable -robot's state- and θ are the parameters of the problem. In our case, as the obstacles do not change in position or in size, the only parameters we consider in our problem are initial position and goal position.

Total integer strategy $I(\theta)$ is a tuple $(\delta^*, T(\theta))$ where T is the set of active inequality constraint. The total integer strategy is what we want to learn offline, it corresponds to the optimal binary assignment in the optimization problem 3.10.

To estimate the probability of finding new unseen strategy, we can use the Good-Turing estimator [3]. The Good-Turing estimator is defined as :

$$G = N_1/n_{obs} \tag{4.2}$$

where N_1 is the number of distinct strategies that appeared exactly one and n_{obs} is the total number of problems. This estimator can then be used to bound the probability of encountering a parameter θ corresponding to an unseen strategy $s(\theta)$ satisfies with confidence at least $1 - \beta$:

$$P(s(\theta) \notin S(\theta_{n_{obs}}) \leq G + c\sqrt{(1/n_{obs})ln(3/\beta)} \tag{4.3}$$

where G is the Good-Turing estimator and $c = (2\sqrt{2} + \sqrt{3})$.

We can use this bound to determine if we gathered enough data, i.e if :

$$G + c\sqrt{(1/n_{obs})ln(3/\beta)} \leq \epsilon \tag{4.4}$$

where ϵ it a threshold we define, the lower the better. If we encounter a new strategy not found in the training data, even if we have an ideal classifier, we could still not be able to solve the problem since our classifier didn't learn that this class exists.

Once an optimal integer strategy $I(\theta)$ has been predicted with a forward pass on a neural network, we can obtain optimal solution of the original problem by solving a much simpler (convex) problem, in which the optimization is no longer made on the binary variables:

$$\begin{array}{ll} \text{minimize} & f_0(x, \delta; \theta) \\ \text{subject to} & f_i(x, \delta^*; \theta) \leq 0, \quad i \in T(\theta) \end{array} \tag{4.5}$$

To solve this problem, we solve a set of linear equations defined by KKT conditions. To resume, we want to learn the mapping between parameters and a corresponding integer strategy. This is a multiclass classification problem over dataset $D = \{(\theta_i, I_i)\}_{i=1}^{T}$ with T samples of different parameters. The worst case possible number of strategies is 2^{KNT} with K the number of edges per obstacle, N the number of obstacles and T the horizon.

The offline steps of the algorithm are described in Algorithm 1. First step is to gather data, by solving many MIP problems with varying parameters (e.g initial position) and recording binaries assignment for the obstacles avoidance task for all obstacles at all time steps. These complete binaries assignments are referred as *total strategies*. Then, we add a new label each time a new total strategy is found. Finally, we train a neural network to minimize the cross-entropy loss via stochastic gradient descent. We refer to this approach of using total strategies for the classification task as the *naive approach*.

Once we have evaluated the output of the neural network for parameter θ, solving this problem is much faster (by 1 to 2 orders of magnitude) and can be done online. The online step of the algorithm is described in Algorithm 2.

4.2 Data generation

We start by collecting data by solving a control problem for several values of key parameters. We choose single integrator dynamics and the map used is the laboratory map shown in Figure 3.3. The horizon is set to $T = 30$, the step time is $\tau = 0.6s$, and we use the shortest path objective function 3.7. Norm of the speed is limited to 0.5 m/s. In our case, we decide to vary the following parameters : initial position and goal position. Other parameters such as the position of obstacle are kept constant. We save the optimal integer assignment for each solution. We solve the problem n_{probs} times to generate the data.

Parameters that vary are then:

- x_0 : Initial positions, matrix of size $n_{probs} \times 2$
- x_e : Ending positions, matrix of size $n_{probs} \times 2$

The solutions to the problem we solve are:

Algorithm 1 Offline algorithm

```
Require: Training data $\{\theta_i\}_{i=1,...,T}$
 1: Initialize strategy dictionary $S \leftarrow \{\}$, train set $D \leftarrow \{\}$
 2: $k \leftarrow 0$
 3: for each $\theta_i$ do
 4:     Solve $P(\theta)$        ▷ Solving the optimization problem for one set of parameters
 5:     Construct optimal strategy $S^*$
 6:     if $S^*$ not in $S$ then
 7:         Add $S^*$ to $S$, assign class k
 8:         $k \leftarrow k + 1$
 9:     end if
10:     Assign binary sequence $\delta_k$ of strategy class $S^*$
11:     Add $(\theta_i, \delta_k)$ to $D$
12: end for
13: Train neural network to minimize cross-entropy loss via stochastic gradient descent
14: return neural network weights
```

Algorithm 2 Online algorithm

```
Require: Problem parameters $\{\theta\}$, strategy dictionary $S$, trained neural network
 1: Compute class scores with forward pass on the Neural Network
 2: Identify top $n_{evals}$ scoring strategies in $S$
 3: for $j = 1, ..., n_{evals}$ do
 4:     if $P(\theta)$ is feasible for strategy $S^{(j)}$ then
 5:         return feasible solution $(x^*, \delta^*)$
 6:     end if
 7: end for
 8: return failure
```

- X : All positions, matrix of size $n_{probs} \times 2 \times T$, with T the horizon. We chose an horizon of 30, with a maximum norm of speed of 0.5 m/s. Smaller horizon leads to much of the problems being infeasible. Higher speed norm leads to robot going through the obstacles.
- U : Inputs (velocities), matrix of size $n_{probs} \times 2 \times T - 1$
- Y : Binaries , matrix of size $n_{probs} \times 4 * N \times T$, with N the number of obstacles.

We also keep track of the solving times and the costs of the MIP problems we solve :

- costs: vector of size n_{probs}
- solve_times: vector of size n_{probs}

We solve the problems using Gurobi, storing solutions for each of the n_{probs} in X, U and Y defined earlier as well as the time required to solve and the objective function value.

4.3 Training

1000 problems were solved using Gurobi. These problems took 13.82 hours to solve. We display the computation times in Figure 4.1. The median time to solve these problems is of 5.4s, but the mean time is 49.78s. The time distribution is heavy tailed, as some problems took a huge amount of time to solve. The maximum time to solve a problem was of 11660s, which is 2160 times the median computation time. It is hard to determine why some particular problems take as much time, but it comes from the NP-hardness of the problem. The high computational time variability also makes it hard to use solvers such as Gurobi to plan online, as some tree search may be much longer than others.

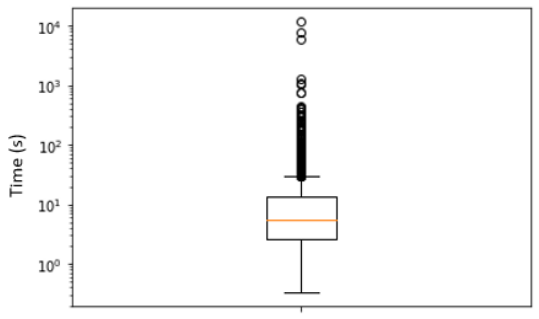


Figure 4.1: Computational times of 1000 problems solved using Gurobi

The data was divided into two sets. 90% of the data was used for training and 10% of the data for testing.

We chose a neural net architecture as follows : 3 linear layers with ReLU activation function and ADAM optimizer [5]. There are 4 input features (x and y coordinates of starting and ending position) and the number of output features is equal to the number of different binary strategies found during the training. The loss function used was the cross-entropy loss function.

Our problem is a multi-class classification problem where input are starting and ending positions of the robot and a class is defined by a sequence of binaries. The input dimension is equal to four (x and y coordinates of starting and ending positions)

For 900 different choices of training parameters, we get 900 different strategies and so 900 different classes, which indicates we would need much more data to do the classification task with this approach. This is discussed in the next paragraph.

4.4 The need for pruning

The previous approach cannot handle more than toy example. Indeed, we saw that in order to make the optimization problem feasible and not allow the robot to go through the obstacles, we need an horizon of 30. This makes the number of binaries equal to 22 (numbers of obstacles) x 4 (number of binaries to represent the big-M constraints) x 30 (horizon) = 2640. There are 2^{2640} possible combinations of binaries with such obstacles and horizon, and thus a huge amount of strategies (different sequences of binaries solution to the optimization problem). We call the approach of listing all the possible combinations of binaries for all obstacles and time steps our naive approach. To encounter all of the possible strategies, and thus be able to classify, we would need to collect an infinite amount of data, which is not feasible. This implies that we need to prune strategies to reduce the number of binaries.

4.5 Pruning strategies with CoCo (Combinatorial Offline Convex Online)

Our previous Mixed Integer Convex Programming problem can be written as :

$$\begin{array}{ll} \text{minimize} & f_0(x,\delta;\theta) \\ \text{subject to} & f_i(x,\delta;\theta) \leq 0, \quad i = 1,...,m_f \\ & \delta \in \{0,1\}^{n_\delta} \end{array} \tag{4.6}$$

where x is a vector of continuous variables describing the state (x and y coordinates for single integrator dynamics), δ are binary variables associated to the obstacles, θ are the problem parameters (for example starting and ending position), f_i are constraints that must be respected.

The simplest way to define integer strategies is to define a new *total strategy* for all different sequences of optimal integer solutions. However, this approach leads to an exponential number of strategies in the size of the problem.

We can exploit the repetitive structure of the obstace-avoidance task to reduce the number of strategies. The state-of the art approach for solving MICP online with learning is CoCo (Combinatorial Offline, Convex Online) framework [5]. In this framework, the authors use task-specific strategies and prune redundant ones to significantly improve scalability compared to the naive approach used for learning in Section 4.1. With the naive approach, there can exist multiple optimal integer strategies, meaning multiple correct labels from the same problem parameters θ. To cope with this issue, we use the fact that an integer strategy can be uniquely defined by considering the set of tight big-M constraints.

In CoCo framework, a logical strategy $S(\theta)$ is defined as a tuple $(\delta^*, T_M(\theta))$, where $\delta^*(\theta)$ is a particular integer solution and $T_M(\theta)$ is the set of tight big-M constraint :

$$T_M(\theta) = \{i | g_i(x^*; \theta) \leq a_i(\theta)\delta_i \Leftrightarrow \delta_i = \delta_i^*\} \tag{4.7}$$

A tight big-M constraint is a big-M constraint that can only be satisfied with the binary value from the solution δ^*.

Let us translate this definition to the way we defined a big-M constraint for an obstacle in Equation 3.3 where : $A_{i_k}x \leq b_{i_k} + M\delta_{i_k}$ with A_{i_k} the kth line of A_i and b_{ik} the kth component of b_i enforce the constraint for being on the outside half plane defined by edge k. This big-M constraint is tight if and only if $A_{i_k}x > b_{i_k}$. If this inequality holds, the only way to make the big-M constraint satisfied is to have δ_{i_k} equal to one. On the contrary if $A_{i_k}x \leq b_{i_k}$, we can set δ_{i_k} to zero or one without violating the big-M constraint. The constraint satisfaction does not depend on the value of δ_{i_k} and so the constraint is not tight.

Considering tight big-M constraints help us reduce the number of possible strategies. Indeed, strategies with same tight constraints are equivalent, as they differ only on values of binaries that can have either value 0 or 1 without impacting the problem. We show this on Figure 4.2, where we detail the constraints associated to being on the four half planes defined by the obstacle, as defined in Section 3.2. In this Figure, the binary values δ_1, δ_3 and δ_4 are required to have a value of 1 so that their associated constraint of staying in the corresponding half-plane remain true while the robot, represented as a blue circle, on the top half-plane defined by the obstacle. However, the constraint $A_{21}x_1 + a_{22}x_2 \leq b_2 + d_2M$ is satisfied no matter the value of d_2 since $A_{21}x_1 + a_{22}x_2 \leq b_2$ is true, with the given robot position. This means that the only non tight constraint is this one.

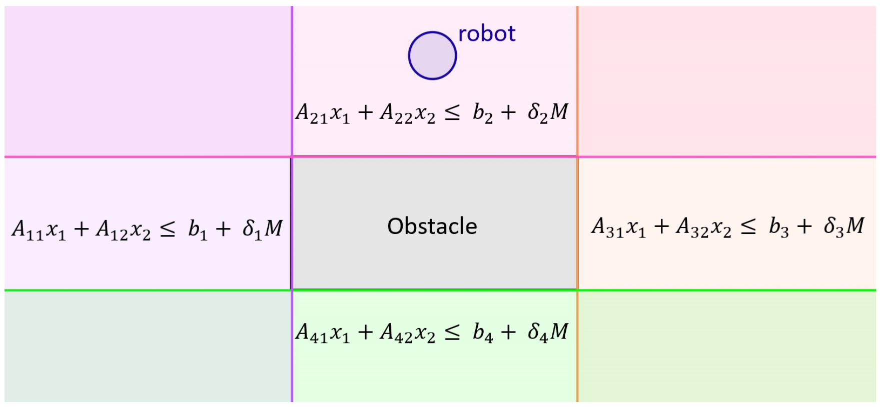

robot
$A_{21}x_1 + A_{22}x_2 \leq b_2 + \delta_2 M$
$A_{11}x_1 + A_{12}x_2 \leq b_1 + \delta_1 M$
Obstacle
$A_{31}x_1 + A_{32}x_2 \leq b_3 + \delta_3 M$
$A_{41}x_1 + A_{42}x_2 \leq b_4 + \delta_4 M$

Figure 4.2: Tight constraint illustration. Robot's position is marked by the blue circle. The only non tight constraint is the one on the top as binary variable d_2 can take value 0 or 1 without it being violated.

An additional way to reduce the number of possible strategies is to consider smaller binary sequences associated to each specific obstacle.

CoCo framework uses separability principle to subdivide a *total strategy* $S(\theta)$ into *individual strategies* $S_1(\theta)$ ... $S_N(\theta)$ where N is the number of obstacles. Indeed, the value of a binary δ_j can be determined solely from the logical value of the binaries referring to constraints emanating from the same obstacle. The advantages of defining new individual strategies is to reduce the total number of strategies as there exist much less possible combinations of binaries compared to total strategies. Also, many of these individual strategies are repeated, due to the repetitive structure of the obstacle avoidance task, so we can prune them.

This new strategy decomposition has been shown to greatly help the strategy classification, as the number of possible classes becomes much lower. A single classifier can be trained for all sub-formulas of identical structure, which artificially augments the size of the training dataset. At runtime, the set of individual strategies S_i must be recombined to create the total strategy used to solve the convex optimization problem. The classifier must now produce N correct predictions to reconstruct the total strategy. Additional steps in the offline algorithm are to detect tight constraints and decompose the total strategies into individual strategies. In the online algorithm, we now predict individual strategies before aggregating them into a total strategy and solving the convex optimization problem using the total strategy.

The decrease of strategies in CoCo approach is directly linked to the number of obstacles. With one obstacle, the only gain is due to the fact that we consider only tight constraints. For example, with one obstacle and for an horizon of 9, where we sampled 70 000 pairs of starting and ending points in the safe area, we get 506 strategies using CoCo and 618 strategies using the naive approach. With three obstacles, we discover 665 using CoCo and 19819 using the naive approach. The number of strategies encountered with CoCo framework does not explode with the number of obstacles, but it still increases with the length of the horizon.

The number of constraints and variables increases polyniomally with the length of the horizon, but the worst case computational cost increases exponentially. For this reason, it is interesting to try to reduce the length of the horizon. This length is constrained by the fact that with a too short horizon (for which we must increase the step time to still reach the objective), the robot will go through the obstacles since it is only constrained to be outside of the obstacles at each step time. This phenomenon is showcased on Figure 4.3.

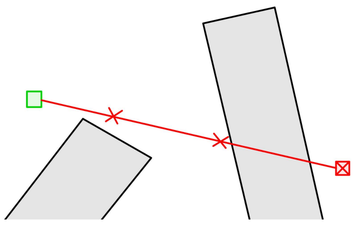


Figure 4.3: Robot trajectory going through obstacle for a too large step time. The robot is constrained to be outside of obstacles at each step time marked by a red cross, but not in between.

Receding horizon approaches can reduce problem complexity by breaking down the global problem into a series of local problems with shorter horizon. In a receding horizon approach, the robot tries to reach a waypoint which is in a path to the target, and not directly the target. If the waypoint of the trajectory is not reachable, the robot might end up entrapped in a dead end when trying to reach the goal point (showcased in Chapter 5). We thus need a heuristic to avoid such cases, this heuristic will be introduced in the next chapter. Another advantage of using receding horizon control is the possibility to incorporate feedback on the control to account for disturbances and modelling errors.

Chapter 5

Receding horizon mixed integer programming for motion planning

To further reduce the computational effort, we consider applying MIP over a receding horizon since computation time to solve a MIP problem grows non linearly with the problem size. With this approach, MIP is used to plan short trajectories toward a goal. Special care must be given to the definition of the objective function and constraints to avoid entrapment behind obstacles, which can prevent from reaching the goal. Entrapment happens when the optimal action in the receding horizon time does not contribute anymore to reach the overall goal. Bellingham *et al.* introduce a cost function for accounting for decisions beyond the planning horizon by estimating the time to reach the goal from the plan's end point [1]. They estimate this time using a graph representation of the environment. Their approach is shown to avoid entrapment and lead to near-optimal performance when trying to achieve minimum time.

5.1 Phase 1: Computation of the cost-to-go

To adapt the problem to shorter horizon times while avoiding entrapment, the first step of the algorithm is to compute a visibility graph $G = (V, E)$ .Vertices V of this graph are every polygonal obstacles vertices, and the ending point of the trajectory. All visible vertices are connected by edges E, meaning two vertices are connected if and only if the straight line joining the two vertices does not cross an obstacle. The output of the algorithm computing the visibility graph is a dictionary with the vertex's number as key, and tuples containing connected vertices with their distance from the key vertex as values. The complexity of the algorithm to compute the visibility graph is $O(n^2)$, with n the number of nodes, which is equal to the number of the obstacles plus one. The visibility graph for our map is represented in Figure 5.1.

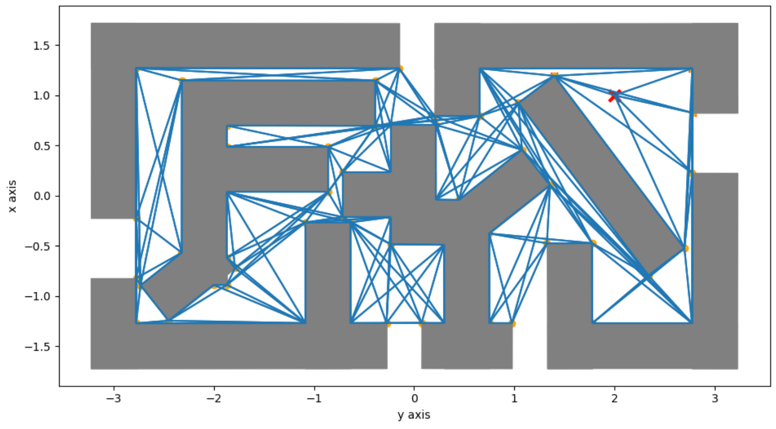


Figure 5.1: Visibility graph for lab map and one goal point

We then use Dijkstra single source shortest path algorithm to compute the shortest distance from all nodes of the graph to the ending point node. The shortest distance from a node to the goal point obtained by applying Dijkstra to the visibility graph is the optimal shortest path (in distance). Indeed, any shortest path between two points among a set S of disjoint polygonal obstacles is a polygonal path whose inner vertices are vertices of S.

5.2 Phase 2: Defining the receding horizon MIP problem to solve

5.2.1 Objective function and constraints

Phase 1 of the algorithm is computed only once and the gathered information is used in phase 2 to define the heuristic for global guidance. We define the new objective function that uses prior knowledge of the cost-to-go from each vertices of the obstacles to the goal in the following way:

$$\text{minimize} \quad ||x_{vis} - x_{end}||_2^2 + C_{vis} \tag{5.1}$$

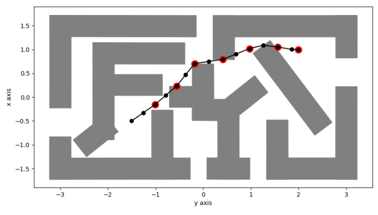
where x_{vis} is the position of one of the vertices of the visibility graph which is an optimization variable. We seek to minimize the distance from the last planned point of the trajectory to the chosen vertex x_{vis} plus the distance from x_{vis} to the target vertex. We also introduce some more variables compared to optimization problem 3.10. We introduce two new continuous variables which are the x and y coordinates of x_{vis} and a binary variable β_j for each vertex of the obstacles. The binary variables are used to encode the choice of a vertex position x_{vis}.

In addition to changing the objective function and adding these variables, we need to add some additional constraints :

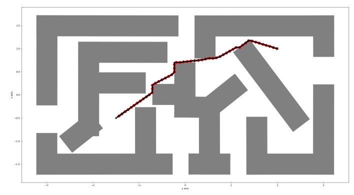
$$
\begin{aligned}
x_{vis} &= \sum_{j=0}^{K_v-1} \beta_j p_j \\
C_{vis} &= \sum_{j=0}^{K_v-1} \beta_j d_j \\
&\sum_{j=0}^{K_v-1} \beta_j = 1 \\
&\beta_j \in \{0,1\} \qquad j = 0,..,K_v-1
\end{aligned}
\tag{5.2}
$$

In this set of constraints, v_j is the position of vertex j and d_j is the distance from this vertex to the goal. K_v is the number of vertices. Exactly one component of β must be equal to one because we want to select only one vertex for x_{vis}.

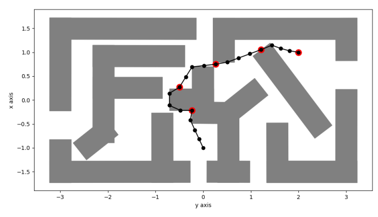
At this point, the controlled robot can still be entrapped in some situations, because it might select as x_{vis} a vertex that is behind an obstacle and not reachable. This effect can be visible in Figure 5.2d. The entrapment on the robot happens when it tries to reach vertices which are not visible and getting closer to them leads towards obstacles. Entrapment is more likely to happen when the time step and the horizon are small (see Figures 5.2c and 5.2d) .



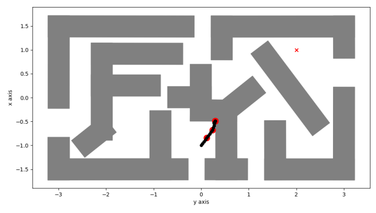
(a) τ = 0.5s, T = 3



(b) τ = 0.08s, T = 3



(c) τ = 0.5s, T = 5



(d) τ = 0.1s, T = 5

Figure 5.2: Receding horizon planning problem solved with different times steps and horizons. Showcasing entrapment problem in (d). Black points are planned positions and red points are position at last time step of the horizon.

5.2.2 Avoiding the entrapment problem

We need to add one key constraint to avoid entrapment. This constraint is making x_{vis} visible from the ending point of the horizon. Visibility constraints are non linear because they involve making sure every point along a line is not in the interior of the obstacles. For two points $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$, the segment S joining those two points can be written as :

$$S = \begin{bmatrix} tx_1 + (1-t)x_2 \\ ty_1 + (1-t)y_2 \end{bmatrix}, \quad t \in [0,1] \tag{5.3}$$

and the visibility constraint between p_1 and p_2 is :

$$(x, y) \notin \mathbf{O}, \quad \forall (x, y) \in S \tag{5.4}$$

where **O** is the domain of all points that are within the polygonal obstacles.

These constraints can be approximated by interpolating along the line of sight and making sure each point is not in the interior of obstacles, as suggested in [1]. However, doing that adds m constraint with m the number of points interpolated, and for each of these m constraints it adds 4 times the number of obstacles binaries (88 in our case). In term of the number of binaries, when checking for each point along the line if it is in an obstacle, we end up adding as much binaries as if we add one time step to the horizon. Thus, this approach is not as effective as we would like.

To estimate the number of points we need to sample along the trajectory, we can first compute the distance between the starting point and the ending point of the trajectory, and then compute how much points we need based on the minimum width of obstacles. The minimum number of points to sample along the trajectory is equal to $\frac{||p_{start} - p_{end}||_2}{min\_width} - 1$ which can reach a high number of binaries (27 for our map for the largest distance between start and end point).

To cope with this issue, we introduce a new method to deal with the selection of the right vertices. In this new approach, we precompute from the initial point the set of vertices that could be visible at the end of the horizon length. First, we compute the set of vertices that are visible from the initial point. This is our initial set of candidates. Then, we compute the distance between the starting position and each of the visible vertices. If this distance is shorter than the length of the planning horizon, we add successors to the point until one point in the successors exceeds the planing horizon. We are using iterative deepening to search the candidates in the visibility graph. Note that only the first computation is needed during the planning, as if the map is static, we can already have the visibility graph of obstacles with distances stored. The set of candidates for a given problem is depicted in Figure 5.3.

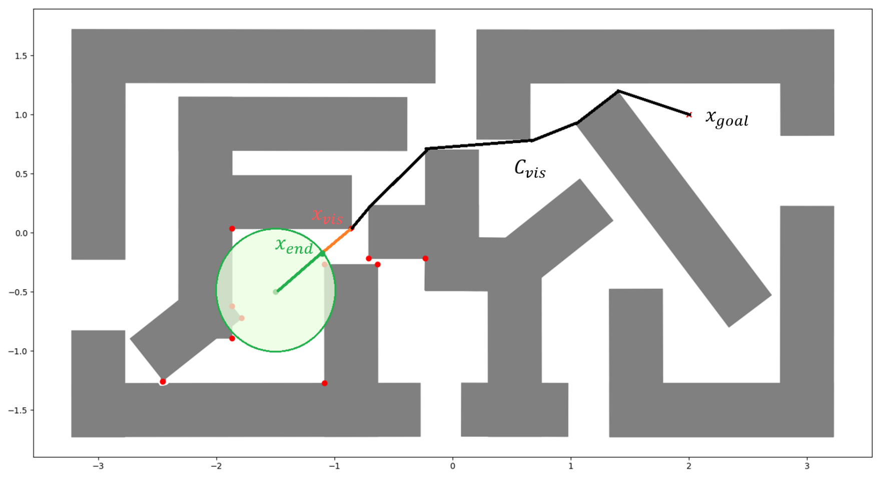


Figure 5.3: Set of candidate point (in red) for an horizon of $T = 3$, a step time of $\tau = 0.5$s and a maximum speed of 0.5 m/s. Green point and red cross are respectively start and goal points. The interior of the green circle denotes the points reachable at the end of the horizon. The point x_{vis} is the point that will be chosen as a waypoint by the optimization, the length minimized by the objective function being the sum of the length of the orange and black segments.

This pre-computation has multiple advantages. First, it allows to reduce the number of binaries used in the formulation of the MIP problem, but most importantly it also allows to not introduce the visibility constraint in the MIP. The formulation of the optimization problem for our receding horizon approach is then:

$$
\begin{array}{lllll}
\text{minimize} & ||x_{vis} - x_{end}||_2^2 + C_{vis} & & & \\
\text{subject to} & p_{t+1} = Ip_t + \tau I p_t & & & t = 0, ..., T-2 \\
& A_i p \leq b_i + M\delta_i & & i = 1, ..., N & t = 0, ..., T-1 \\
& \sum_{k=1}^{K_i} \delta_{i_k} \leq K_i - 1 & & i = 1, ..., N & t = 0, ..., T-1 \\
& \delta_{i_k} \in \{0, 1\} & k = 1, .., K_i & i = 1, ..., N & t = 0, ..., T-1 \\
& p_{vis} = \sum_{j=0}^{K_{cand}-1} \beta_j v_j & & & \\
& C_{vis} = \sum_{j=0}^{K_{cand}-1} \beta_j d_j & & & \\
& \sum_{j=0}^{K_{cand}-1} \beta_j = 1 & & & \\
& \beta_j \in \{0, 1\} & & & j = 1, .., K_{cand} \\
& p_0 = p^{init}, \quad v_0 = v^{init} & & & \\
& p_t \in [\underline{p}, \overline{p}] & & & \\
& v_x^2 + v_y^2 \leq 0.5^2 & & & t = 0, ..., T-1
\end{array}
\tag{5.5}
$$

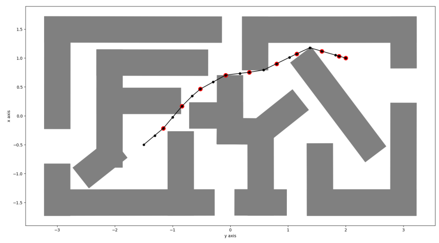
where K_{cand} refers to the number of candidate vertices.

On Figure 7, we showcase the results of this method on the same problems solved in Figure 5.2. The problem of entrapment is now solved and the robot can reach any point from anywhere in the map, as long as the starting point and goal point are outside of the obstacles.

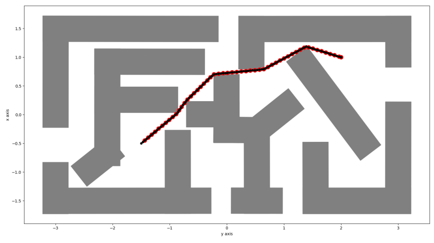
Note that the receding horizon approach adopted does not ensure feasibility for every system dynamics. Indeed, if the system has a limited turning rate (example of such systems include cars, or airplanes), the algorithm may bring the system in a place where it can't avoid a collision. The current algorithm could be adapted for such systems using a generalization of Dubin's path concept in phase 1 of the algorithm. The idea would be to place circles in the vertices of obstacles to define if the robot could pass at critical points. Another approach, implemented in [11] is to constrain the last point of the trajectory to end in a safe state, in which a vehicle can safely remain for an indefinite period of time. For a fixed-wing UAV in 2D, this safe state is defined by the smallest circle the airplane can make. This approach gives the ability to avoid crash in cases in which the system has only a local map of the environment, but the system can still get into a dead end if the dead end is in a safe state.

From now on, we can think of two ways of further reducing the computational cost of using receding horizon MIP for path planning.

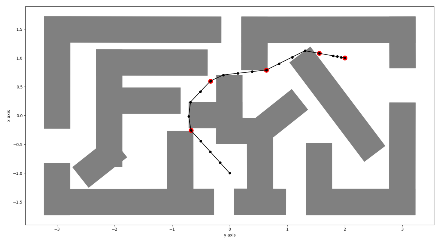
The first one is to exploit the pre-computation before the MIP to reduce the number of binaries used for obstacle avoidance. Indeed we can consider only the binaries associated with the obstacles that we can reach in the horizon time, as all the others are not used. This can be done by considering only the



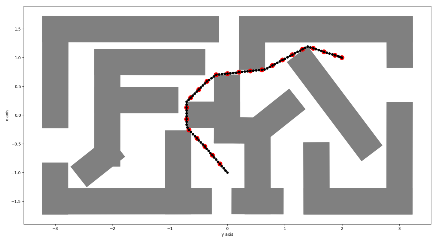
(a) τ = 0.5s, T = 3



(b) τ = 0.08s, T = 3



(c) τ = 0.5s, T = 5



(d) τ = 0.1s, T = 5

Figure 5.4: Receding horizon planning problem solved with different times steps and horizons. There is no more entrapment problem thanks to the computation of candidates for x_{vis}. Black points are planned positions and red points are position at last time step of the receding horizon.

obstacles which have vertices in the set of candidates for x_{vis} or simply which the robot could reach at maximum speed towards those obstacles.

The second one is to use learning to learn two things : the point the robot aims for x_{vis} and strategies of binaries. Learning the point to reach x_{vis} allows to reduce the number of binaries introduced by the number of candidates there was for x_{vis}, as we introduced one binary for each candidate. It also allows to not compute visible points from the start point and searching successors in the graph. In Chapter 6, we use learning on the local problems produced by our receding horizon so that algorithm can scale better to large problems, both in horizon length and number of obstacles.

Chapter 6

Learning and reconstructing strategies for local problems

In this chapter, we first define the problem that needs to be solved in paragraph 6.1 and how we generate the data for the learning part in paragraph 6.2. Then, introduce our method for learning the waypoint selection part in paragraph 6.3. We also explain how we learn the binary sequences associated to the constraint of staying outside of obstacles in paragraph 6.4. In Section 6.5, we show results of our current approach and how it can be improved. In Section 6.6, we fix the problems of the previous approach by considering only nearby obstacles when reconstructing the aggregated strategy, and evaluate the new algorithm. Finally, we gathered some pros and cons about our algorithm and discussed the effect of the different parameters in sections 6.7 and 6.8.

6.1 Problem statement

Following from our receding horizon approach, the local problem to solve is :

$$
\begin{array}{llllr}
\text{minimize} & ||x_{vis} - x_{end}||_2^2 + C_{vis} & & & \\
\text{subject to} & p_{t+1} = Ip_t + \tau I p_t & & & t = 0, ..., T-2 \\
& A_i p \leq b_i + M\delta_i & & i = 1, ..., N & t = 0, ..., T-1 \\
& \sum_{k=1}^{K_i} \delta_{i_k} \leq K_i - 1 & & i = 1, ..., N & t = 0, ..., T-1 \\
& \delta_{i_k} \in \{0, 1\} & k = 1, .., K_i & i = 1, ..., N & t = 0, ..., T-1 \\
& p_{vis} = \sum_{j=0}^{K_{cand}-1} \beta_j v_j & & & \\
& C_{vis} = \sum_{j=0}^{K_{cand}-1} \beta_j d_j & & & \\
& \sum_{j=0}^{K_{cand}-1} \beta_j = 1 & & & \\
& \beta_j \in \{0, 1\} & & & j = 1, .., K_{cand} \\
& p_0 = p^{init}, \quad v_0 = v^{init} & & & \\
& p_t \in [\underline{p}, \overline{p}] & & & \\
& v_x^2 + v_y^2 \leq 0.5^2 & & & t = 0, ..., T-1
\end{array}
\tag{6.1}
$$

To plan the robot's trajectory, we solve this problem recursively until the distance of the robot to the end point is close to zero. At each replanning step, the starting position of the robot is a point of the trajectory that was planned at the previous time-step.

To reduce the computational cost of solving this problem, we want to learn the waypoint x_{vis} that the robot is aiming for. x_{vis} is either one vertex of the polygonal obstacles or the ending point. We also need to learn the sequences of binaries associated to the constraint to stay outside every obstacle at every step time, which are variables δ_i of the optimization problem.

If we can predict the optimal assignment of these variables, the resulting problem is now a convex optimization problem that can be written as:

$$
\begin{array}{llll}
\text{minimize} & ||x_{vis} - x_{end}||_2^2 & & \\
\text{subject to} & p_{t+1} = Ip_t + \tau I p_t & & t = 0, ..., T-2 \\
& A_i p \leq b_i + M\delta_i & i = 1, ..., N & t = 0, ..., T-1 \\
& p_0 = p^{init}, \quad v_0 = v^{init} & & \\
& p_t \in [\underline{p}, \overline{p}] & & \\
& v_x^2 + v_y^2 \leq 0.5^2 & & t = 0, ..., T-1
\end{array}
\tag{6.2}
$$

where the binaries δ_i and and the point x_{vis} are no more variables of the optimization problem, and are instead predicted using the starting position p^{init} as a feature.

6.2 Data generation

We recorded the data for 200000 problems with a horizon of 3, a step time of 0.5 seconds, saving the solution of each receding horizon problem. The map in which the problems are solved is the lab map displayed in Figure 3.3 which is a rectangle with a 7 meters length and 3.2 meters width in which there are 22 obstacles. The parameters of the problem are the initial position coordinates, the goal point being fixed.

We use 90% of the data for the training set and 10% of the data for the test set.

6.3 Learning the waypoint

Since we don't want to compute candidate points at each time-step because it would require computing distances and doing a graph search, and because it would add more binaries in the optimization problem (one per candidate point), we try supervised learning to learn waypoints x_{vis} that the robot should try to reach in parallel with the binary strategies.

To do so, we convert the set of vertices to class numbers, where the class number is associated to x and y coordinates of the vertex using a mapping function $f : \mathbb{N} \rightarrow \mathbb{R}^2$.

The input feature vector is of dimension 2 (x and y coordinates of the robot) and an output of dimension 21 where 21 is the number of different vertices that were assigned to x_{vis} during training for different starting positions.

We used a neural network with 3 layers, and 50 neurons per hidden layer. We used the cross entropy loss and stochastic gradient descent optimizer with learning rate ranging from 0.1 to 0.04 which we decreased during the training. The training loss is displayed on Figure. We achieve an accuracy of 97% on the test set which is composed of 20 000 problems.

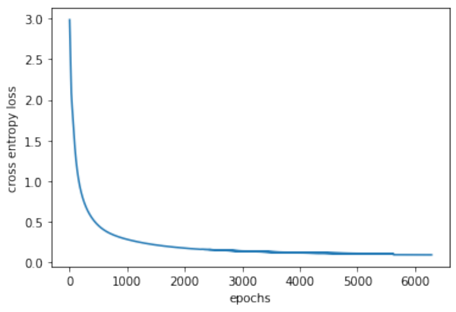


Figure 6.1: Training loss

6.4 Learning strategies

Since we can solve the problem in a receding-horizon fashion, we have mitigated the problem of the number of binaries exploding with the horizon length. Still, we want to use the CoCo framework to reduce the number of strategies that can be discovered.

We construct the strategies of the training set by separating the global binary sequences (the entire binaries assignment for all edges of obstacles and all time steps) into multiple smaller binary sequences linked to each obstacle, still for every time step in the horizon length. With the naive approach, which considered the entire sequence of binaries, we had one binary sequence of 22 obstacles $\times$ 4 binaries per obstacle $\times$ horizon length. Now we have 22 binary sequences of 4 binaries per obstacle $\times$ horizon length. Here, we exploit the structure of the problem in which values of binaries only depend of other binary assignments within each obstacle and it allows us to greatly reduce the number of different binaries combinations that can be encountered.

Also, we detect active constraint to give a unique key to each strategy. This allows to not have multiple binary sequences solutions for the same problem. Two binary sequences solutions are equivalent if they have the same set of active constraints. Active constraints are detected for a each constraint and at each time step.

Now, we use a neural network to learn to predict the strategies for each obstacle. The input features are : x and y initial coordinates positions of the robot and a one-hot encoded vector representing which obstacle this strategy is linked to. The obstacle to which a strategy belongs is categorical data which have no ordered meaning (obstacle number 4 has no reason to have a smaller number than obstacle 8, and obstacle 8 is not two times obstacle 4), thus it should not be passed as integer parameters. One Hot Encoding (OHE) is a good solution to represent the variable indicating the obstacle to which belongs the strategy as it avoid these pitfalls.

As an example, a one-hot vector (0,0,1,0, ..., 0) indicates that the strategy belongs to the third obstacle.

By separating strategies into obstacles, we multiplied the initial number of feature vectors by the number of obstacles.

To learn the binary strategies associated to each obstacle, we used a neural network and input dimension of 24 (x and y coordinates and size 22 one hot encoded vector), and three linear layers with 256 neurons in the hidden layers and an output dimension of 31 where 31 is the number of different strategies found in the training data.

We use stochastic gradient descent with a learning rate of 0.1. We use minibatch gradient descent to train faster and use less RAM. The batch size we use is 32. The training loss, smoothed with a moving average on the 100 previous samples, is represented in Figure 6.2. The accuracy on the test set is 0.98.

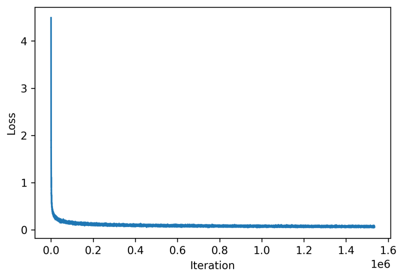


Figure 6.2: Training loss for strategy prediction

6.5 Results and motivation to consider only closest obstacles

During the online phase of the algorithm, we first predict the waypoint and the individual binary strategies for each of the obstacles. We have, as an output of the neural network, the probabilities p_i associated to each individual strategy $S_i(\theta)$ where $i = 1, ..., N$. Each predicted individual strategy needs to be correct for us to ensure a feasible aggreagted total strategy $S(\theta) = S_1(\theta)$ ... $S_N(\theta)$.

To increase the robustness of our algorithm, we consider the most likely strategy combinations from our neural network.

If the problem is not feasible, we try to solve it with the another strategy likely to be the right one. This approach led to 100% feasibility with a smaller problem with 3 obstacles, while testing a maximum of 7 total strategies, but as the number of obstacles grows, the amount of likely combinations of individual strategies increases and it becomes time consuming to test more of them. The average computation time on this smaller problem is less than three milliseconds, and it reduces the time required to solve the receding horizon problem optimization by two orders of magnitude. Note that only changing one individual strategy leads to a different total strategy $S(\theta)$. The reconstitution of the total strategy using CoCo approach becomes harder as the number of obstacle grows. Indeed, with three obstacles, if our neural network has an accuracy of 0.9 for individual strategies, the percentage of correct predictions for the total aggregated strategy becomes $0.9^3 = 0.72$, for 22 obstacles, it is equal to $0.9^{22} = 0.01$ which means the CoCo approach is much harder to use here. There is an exponential relationship between the accuracy of the prediction of the total strategy and the number of obstacles. To overcome this problem, we need to ensure that our model accuracy is as good as possible, and also try to change in priority the individual strategies that have the smallest scores predicted by our neural network. We expect this to not be enough and envision constructing strategies only for obstacles which are close to the robot.

With our full problem with 22 obstacles, using only the most likely individual strategies predicted by the neural network, only 2.5% of the optimization problems remain feasible. If we try to solve for all the problems using the individual strategies with highest scores, except one for which we use the second most likely prediction, we get 33% feasible problems, but testing those problems is time-consuming, and make us lose the computational time benefits of CoCo approach.

This first approach of keeping the 22 obstacles is impractical. While we can increase the amount of feasible problems by testing successively different strategies, the accuracy of our predictions remains low, and the more strategies we test, the more time it takes. Before having reached 30% of feasible problems, we already reach solving times similar to the ones without the learning approach.

To overcome the difficulty of predicting correctly the strategies for all obstacles, we consider predicting strategies only for obstacles which are in a certain radius of the robot, thus reducing the complexity of reconstructing the optimal total strategy $S(\theta)$. Indeed, because we use a receding horizon approach, it is useless to consider the constraint of being outside of an obstacle the robot cannot reach. We now detail the results of this second approach.

6.6 Results when aggregating strategies for proximal obstacles only

Computation time calculation

To ensure a reliable computation time comparison. We use the same hardware and software environment, the same input data, and average the computation times of the problems. We use a processor Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz, 2208 MHz, 6 hearts, 12 logical processors.

Feasibility and solving times

While keeping only the obstacles which are in the reachable radius by the robot, we get 92% of feasible problems. Fetching the nearest obstacles has a computational complexity which linearly increases with the number of obstacles. For 22 obstacles, this operation takes a mean time of t = 3.7e-4s (std = 5.5e-4s). The mean number of obstacles considered for our example problem is 2.98 obstacles. We show the distribution on Figure 6.5 along with the part of feasible problems, which are problems which have a solution.

The mean time for solving the initial local MIP problems of the test set (Figure 6.3 was 0.05s (std = 5.9e-3s). The forward pass on the neural network takes a mean time of 6.5e-4 s (std = 8.1e-4s), and solving the convex optimization problem after having predicted the binaries and the waypoint takes 5.3e-3s (std = 4.3e-3s), as shown on Figure 6.4. We recall that global MIP problem was taking an average of 5 seconds, with some problems taking hours to solve due to NP-hardness.

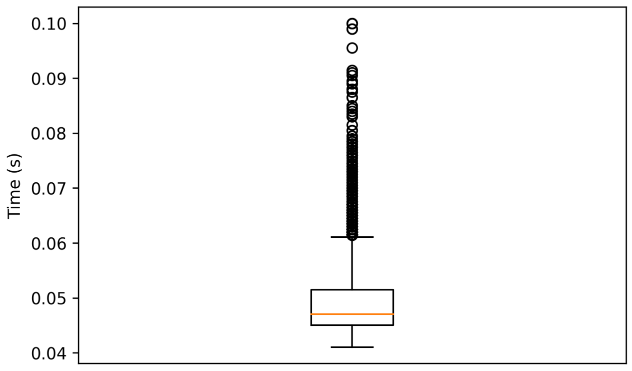


Figure 6.3: Time to solve the local MIP problems (T = 3)

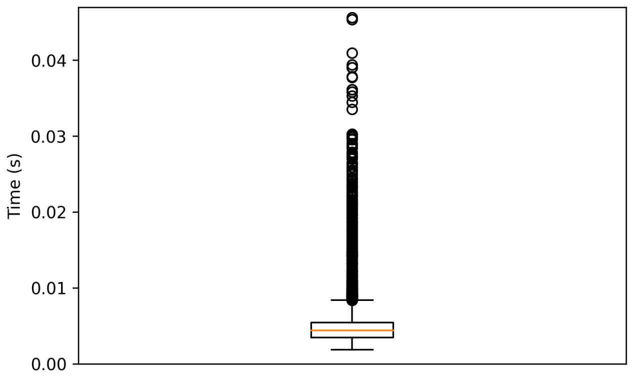


Figure 6.4: Time to solve the local convex optimization problem (T = 3)

Impact of the number of nearby obstacles on performance

As we mentioned before, the number of obstacles considered has a great impact on the prediction of the optimal total strategy as the more obstacles are considered, the smaller the accuracy of the global prediction is.

We observed that the percentage of feasible problems indeed decreased when more obstacles are added, the fraction of feasible problems for two obstacles is the one for one obstacle to the power of two. Overall, the percentage of feasible problems is 92 %.

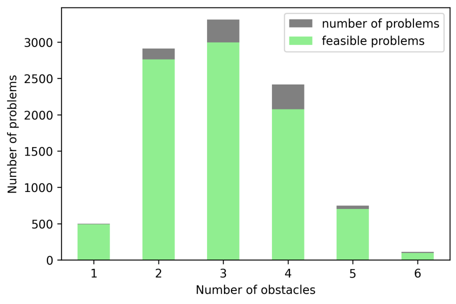


Figure 6.5: Part of feasible problems with respect to the number of nearby obstacles ($T = 3$, $\tau = 0.5$)

Evaluating the cost difference with MIP

To evaluate briefly the difference between the problems solved with MIP and the one solved using CoCo and the receding horizon approach, we compute the mean difference in distance to the waypoint the robot should try to reach, which is less than 2 millimeters, with a standard deviation of 8 millimeters.

6.7 Pros and Cons of the approach

We resume here the pros and the cons of the previously developed approach.
Pros of our approach are :

1. The receding horizon framework allows to solve problems with very large horizon without additional cost.
2. Generating the data to train the neural network is less computationally intensive as we generate data for local problems of smaller horizon length and not for the full problem.
3. We can exploit the fact that we use a receding horizon to reduce the problem's complexity by aggregating strategies for the nearby obstacles only, and this combines effectively with CoCo framework, which separates the total strategy prediction for each obstacle.
4. The solving time is reduced both by considering a receding horizon, and also applying CoCo approach.
5. The proposed waypoint method can be used with a variety of cost functions, as long as we can define a time invariant heuristic adapted to the task.

The cons are :

1. The achieved cost is sub-optimal because the receding horizon approach optimizes on segments of the trajectory using a heuristic.
2. The proposed method is only useful if one task needs to be repetitively solved a large amount of times.

6.8 Effect of the parameters

We discuss here the qualitative effect of different critical parameters of the problem, in the offline and online parts of the algorithm.

- **Horizon** T: The bigger the (receding) horizon, the longer the problem takes to solve when generating the data as much more variables need to be assigned a value. Online, the local problems will also be more complicated to solve as more binary predictions will need to be correct simultaneously as we will consider more nearby obstacles and for more time steps.

  On Figure 6.6, we show the path for a local problem where the horizon is 2 on the left and 5 on the right. For the local problem with horizon $T = 2$, the only obstacle considered online for the avoidance constraint will be obstacle 1, meaning a single strategy, which consist of the correct binary variables assignment for the obstacle for two time steps, must be correctly predicted. On the right, for the horizon $T = 5$, three obstacles are considered, because the robot can now reach all of them. This means we need three correct strategy predictions, and each strategy is harder to predict because it consist of binary variable assignments for five time steps.

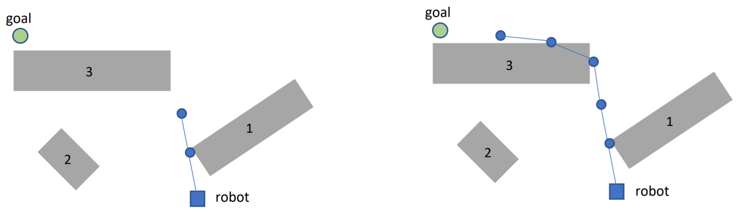


Figure 6.6: Illustration of a local problem solved with horizon $T = 2$ (left) and $T = 5$ (right)

- **Time step** τ: The size of the time step does not impact a lot data gathering as it does not reduce the number of variables of the MIP problem. The smaller the time step, the quicker the problem takes to solve as less obstacles will be considered for prediction and online solving. The main gain is in accuracy of the problem. However, if the time step is smaller, more problems need to be solved to allow the robot to navigate between its position and the goal. If the time step is too big, the robot can move through obstacles.

  In Figure 6.7, we show a local problem solved with two different time step duration. For the one on the left, which has a small time step, the online prediction of strategy is done only for obstacle 1, whereas considering a larger time step (right), the strategies are predicted for all of the obstacles shown, as the robot can go farer between two steps.

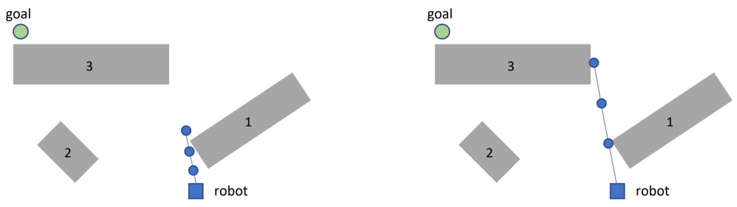


Figure 6.7: Local problem solved with small time step (left) and large time step (right)

- **Number of obstacles** n_{obs}: The more obstacles there are in the map, the harder it is to train the neural network and the longer it takes to gather enough data. Also, it is more likely that the online algorithm should take into account more nearby obstacles, if the map size is constant, leading to a smaller amount of feasible and optimal problems. The strategy prediction does not increase significantly the time taken to solve the problem when we consider more obstacles. The time it takes to solve the convex optimization problem increases because we add more constraints.

Figure 6.8 shows two local problems with same configuration and which have the same solution, but with maps differing in their number of obstacles. Problem on the right is harder to solve because it is longer to gather data, and it requires more correct predictions in the online algorithm as we need to predict strategies for obstacles 4, 5, and 6.

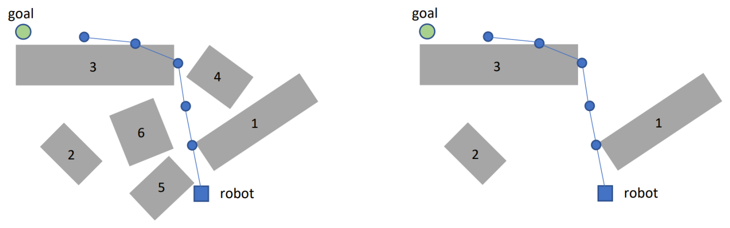


Figure 6.8: Local problem solved with a dense (left) and a sparse field of obstacles (right)

- **Size of the map**: The size of the map has no big impact on the solution of the problem, because of the receding horizon approach. A bigger map however requires to gather more data to cover a wide range of initial positions However, problem size can increase a lot when joining one side of the map from the opposite size.

Chapter 7

Implementation on a robotic testbed

To demonstrate our approach, we used ROS to send the path computed to the robot to follow. As we are not using the real robot dynamics, we used Model Predictive Control (MPC) for the control. Note this controller was not developed as part of this project.

To ensure that the robot's trajectory is always feasible, if the problem is infeasible, we use an approach inspired from Model Predictive Control (MPC), which consist on solving for multiple steps, but only implementing the first one. If the new sub-problem is not feasible or solution cannot be found in the given time, we can then just continue using the second step of the previous solution. This method makes the algorithm very robust. Figure 7.1 illustrates the solution of one problem solved with our approach, where each green point is the first step of a local problem and blue rectangles represent obstacles.

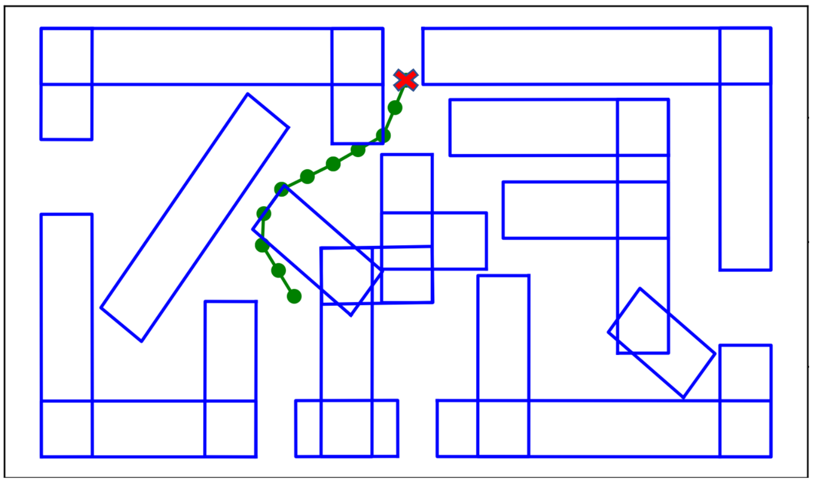


Figure 7.1: Full problem solved iteratively

Chapter 8

Conclusion

In this work, we first started by researching the state of the art techniques to solve the robot motion planning problem using Mixed Integer Programming. While these techniques are powerful, they do not scale well to large or complex environments, for which there are many obstacles and computing a long path. We developed a two steps approach to address these limitations.

The first phase of the algorithm is done offline. During this phase we compute a visibility graph connecting the vertices of the obstacles and the target point. We then solve Dijkstra single source shortest path algorithm to obtain the distance from every vertex to the target point. This allows to formulate the MIP problem as a receding horizon problem, in which the global planning problem is broken down a series of local problems, which can be solved more efficiently. We then gather data for the receding horizon problem and learn how to predict a waypoint the robot should try to reach each time the local problem is solved, along with the binaries constraining the robot to stay outside of the obstacles.

During the online phase of the algorithm, we solve a standard convex optimization problem, using the predicted waypoint and binaries. We only consider constraints associated to nearby obstacles, which enable us to gain in accuracy.

Finally, we demonstrated the effectiveness of our approach on a range of robot motion planning problems. Our results showed that our approach can significantly reduce the computation time and make it more scalable to large and complex environments with long horizons and many obstacles.

There are still some open questions like the choice of the local problem horizon length and overlap between two problem, and the consideration of different cost functions and dynamics.

Thanks

I would like to express my sincere gratitude to Dr. Tony Wood and Tingting Ni for their valuable advice and engaging discussions throughout the semester. Their expertise and guidance have been valuable in helping me to understand and approach the subject matter in a more comprehensive way.

I would also like to thank Prof. Maryam Kamgarpour for her insights and for welcoming me into her lab. Her interest for the subject and her willingness to share her knowledge have been instrumental in my learning experience.

Bibliography

[1] J. Bellingham, A. Richards, and J.P. How. Receding Horizon Control of Autonomous Aerial Vehicles. In *Proceedings of the 2002 American Control Conference (IEEE Cat. No.CH37301)*, volume 5, pages 3741–3746 vol.5, May 2002. ISSN: 0743-1619.

[2] Dimitris Bertsimas and Bartolomeo Stellato. Online Mixed-Integer Optimization in Milliseconds, March 2021. arXiv:1907.02206 [cs, math].

[3] Dimitris Bertsimas and Bartolomeo Stellato. The voice of optimization. *Machine Learning*, 110(2):249–277, February 2021.

[4] A. Cauligi, P. Culbertson, E. Schmerling, M. Schwager, B. Stellato, and M. Pavone. CoCo: Online Mixed-Integer Control via Supervised Learning, July 2021. arXiv:2107.08143 [cs].

[5] A. Cauligi, P. Culbertson, B. Stellato, D. Bertsimas, M. Schwager, and M. Pavone. Learning Mixed-Integer Convex Optimization Strategies for Robot Planning and Control, April 2022. arXiv:2004.03736 [cs].

[6] Chaw-Bing Chang. On the minimum fuel control problem for discrete systems. *Proceedings of the IEEE Conference on Decision and Control*, December 1975.

[7] M.G. Earl and R. D'Andrea. Iterative MILP methods for vehicle-control problems. *IEEE Transactions on Robotics*, 21(6):1158–1167, December 2005. Conference Name: IEEE Transactions on Robotics.

[8] Daniel Ioan, Ionela Prodan, Sorin Olaru, Florin Stoican, and Silviu-Iulian Niculescu. Mixed-integer programming in motion planning. *Annual Reviews in Control*, 51:65–87, January 2021.

[9] A. Richards and J.P. How. Aircraft trajectory planning with collision avoidance using mixed integer linear programming. In *Proceedings of the 2002 American Control Conference (IEEE Cat. No.CH37301)*, volume 3, pages 1936–1941 vol.3, May 2002. ISSN: 0743-1619.

[10] Arthur Richards, Yoshiaki Kuwata, and Jonathan How. Experimental Demonstrations of Real-time MILP Control. November 2003.

[11] T. Schouwenaars, J. How, and E. Feron. Receding horizon path planning with implicit safety guarantees. In *Proceedings of the 2004 American Control Conference*, volume 6, pages 5576–5581 vol.6, June 2004. ISSN: 0743-1619.