KU LEUVEN

FACULTEIT INGENIEURSWETENSCHAPPEN

Scalable Multirotor UAV Trajectory Planning using Mixed-Integer Linear Programming

Jorik De Waen

Thesis voorgedragen tot het behalen van de graad van Master of Science in de ingenieurswetenschappen: computerwetenschappen, hoofdoptie Artificiële intelligentie

Promotor:
Prof. dr. T. Holvoet

Assessoren:
Dr. M. Cruz Torres
Dr. B. Bogaerts

Begeleiders:
H. T. Dinh
Dr. M. Cruz Torres

Academiejaar 2016 – 2017

Preface

TODO: preface

Jorik De Waen

Contents

Abstract

TODO: english abstract

Samenvatting

TODO: dutch abstract

List of Figures

List of Tables

1

Introduction

As a consequence of ever-increasing automation in our daily lives, more and more machines have to interact with and unpredictable environment and other actors within that environment. One of the sectors that seems like it will change dramatically in the near future is the transportation industry. Autonomous cars are actually starting to appear on public roads, autonomous truck convoys are being tested and several large retail distributors are investing heavily into delivering orders with Unmanned Areal Vehicles (UAV) instead of by courier.

Multirotor UAVs, often called quadrocopters when they have four propellers, are rising quickly in popularity. This is due to a variety of factors. Multirotos UAVs are cheaper to build than helicopters and they are much more agile than airplanes. The continuous improvements in battery technology allow them to use electric motors, compared to the gasoline motors which were ubiquitous in model aircraft just a few years ago. Finally, advances in mobile computing and sensors (caused by the smartphone industry) allowed "fly-by-wire" technology as a standard. With fly-by-wire, a computer interprets the commands by a human controller while keeping the vehicle stabilized. This allows even laymen to fly a multirotor UAV with a minimal amount of practice even though multirotor UAVs are inherently unstable. This is in stark contrast with the complex interactions between forces and torques on helicopters which take extensive training to master.

The combination of the rising availability of multirotor UAVs and the perpetually increasing automation spark the imagination of many. Media outlets often present a utopic (distopic?) future with swarms of UAVs delivering products to consumers only minutes after they have been ordered. In the industry there is also an increasing amount of interest in autonomous UAVs because they could carry out inspections on large structures which are hard to reach for humans.

However, there are still many challenges that prevent the proliferation of autonomous UAVs. These include, legislative uncertainties, coordination of the increased air traffic, trajectory planning and more. This thesis focuses on the trajectory planning problem, and more specifically on trajectory planning in large and complex environments like cities.

1.1 Problem Statement

Typically, trajectory planning for UAVs is done "online" and in the short term. This means that the calculations happen in realtime while the UAV is flying and that the trajectory is only planned for the near future. This works well in open spaces with a limited amount of obstacles, but cannot be applied to complex environments like cities.

The goal of this thesis is to develop a long-term trajectory planning algorithm for multirotor UAVs which can scale to large and complex environments like cities. The algorithm is developed for offline trajectory planning, which means that the trajectory is planned before the flight of the UAV. Long-term offline planners are complimentary to short-term online planners. Long-term plans ensure that a safe trajectory to the goal exists and can guide the short-term planner in the right direction.

Mixed-Integer Linear Programming (MILP) is the most common technique used in short-term trajectory planners. It allows for a large amount of flexibility when modeling the trajectory planning problem. From this model, the optimal trajectory can be found by using widely-available solvers. However, performance challenges prevent this technique from being applied beyond short-term planning. This thesis aims to preserve the flexibility of MILP trajectory planning while making the technique scalable enough to solve planning problems on the scale of cities.

1.2 Contribution

In the past, long-term plans for UAVs have usually been built by humans. Research into long-term trajectory planning for UAVs has been limited. I have not been able to find previous work which has focused entirely on the scalability aspect of MILP trajectory planning. This thesis presents an approach which can scale to large and complex environments. While there is plenty of room for improvement, this new approach is the main contribution of this thesis.

The experiments performed to test the algorithm also point to some ways the approach can be adapted to improve the performance and quality of the generated trajectory. Because of the lack of earlier research into the scalability aspect of trajectory planning, this thesis contributes new insights into the challenges and opportunities associated with building a scalable MILP trajectory planning system.

1.3 Structure of the Thesis

Section 1.4 summarizes the previous work that has been done in the field, presenting similar approaches as well as some alternatives.

The previous work in the field shows a common design to modeling the path planning problem as a MILP problem. This design forms the foundation of the MILP model used in this thesis. Chapter 2 details how this MILP model is constructed.

Chapter 3 identifies the performance limitations which come along with this MILP approach. It also proposes an algorithm which preprocesses the trajectory planning problem to circumvent those performance limitations.

Chapter 4 presents several extensions to the algorithm to further improve the results. It also shows the graphical visualization tool which was developed to analyze the execution of the algorithm.

Chapter 5 presents a series of experiments which test the algorithm using a variety of environments and combinations between parameter values.

Chapter 6 discusses the results of the experiments and assesses whether or not the goal of this thesis has been reached. It also highlights some of the issues that are still present in the algorithm. Furthermore, it proposes several extensions and changes which could be added in the future to further improve the algorithm.

Finally, Chapter 7 summarizes the main elements of this thesis and forms a final conclusion.

1.4 Literature Review

Schouwenaars et al. [12] were the first to demonstrate that MILP could be applied to path planning problems. They used discrete time steps to model time with a vehicle moving through 2D space, just like the approach we present in this paper. The basic formulations of constraints we present in this paper are the same as in the work of Schouwenaars et al. To limit the computation complexity, they also presented a receding horizon technique so the problem can be solved in multiple steps. However, this technique was essentially blind and could easily get stuck behind obstacles. Bellingham[1] recognized that issue and proposed a method to prevent the path from get stuck behind obstacles, even when using a receding horizon.

Flores[7] and Deits et al[5] do not use discretized time, but model continuous curves instead. This not possible using linear functions alone. They use Mixed Integer Programming (MIP) with functions of a higher order to achieve this. The work by Deits et al. is especially relevant to this paper, since they also use convex regions to limit (or in their work: completely eliminate) the need to model obstacles directly.

Several papers [6, 8, 3, 10] show how the output from algorithms like these can be translated to control input for an actual physical vehicle. This demonstrates that, when they properly model a vehicle, these path planners need minimal post-processing to control a vehicle. Of course that does assume these planners can run in real time to deal with errors that inevitably will grow over time. Culligan [4] provides an approach built with real time operation in mind. Their approach only finds a suitable path for the next few seconds of flight and updates that path constantly.TODO: explain better

More work has been done on modeling specific kinds of constraints or goal functions. For instance, Chaudhry et al. [2] formulated an approach to minimize

radar visibility for drones in hostile airspace. However, none of these have really attempted to make navigating through a complex environment like a city feasible. The approach by Deits et al. [5] could work, but did not really explore the effects of longer paths on their algorithm.

TODO:PRM and RERT

1.5 Assumptions

TODO

2

Modeling Trajectory Planning as a MILP problem

2.1 Introduction

This section covers how a trajectory planning problem can be represented using Mixed Integer Linear Programming. This is a relatively simple model based on the work by Bellingham [1]. This model by itself is sufficient to solve the trajectory planning problem, but its poor scalability prevents it from being used on any but the most basic scenarios.

Section 2.2 starts out with a basic overview of what MILP is, highlighting its strengths and weaknesses. Section 2.3 to 2.6 describe how I modeled the trajectory planning problem.

2.2 Overview of MILP

2.2.1 Mathematical Programming

Mixed integer linear programming is and extension of linear programming, which is in turn a form of mathematical programming. In mathematical programming, problems are represented as models. Each model has some amount of unknown decision variables. Models are solved by assigning values to those decision variables. To accurately represent most problems, a model also needs constraints. Constraints are mathematical equations which determine the allowed values for each decision variable. When all decision variables have values which are allowed by the constraint, the constraint is called "satisfied".
In mathematical programming, solvers are used to assign values to variables in a model. The goal of a solver is to assign those values while ensuring that all constraints are satisfied. For many problems (and their corresponding models), this is extremely challenging. Only one solution may exist among a near infinite amount of options. In many cases, the goal is not to find just any solution to the problem, but to find the best possible solution. The quality of a solution is determined by an objective function. The solver aims to find the solution with the highest (or lowest) score based on the objective function, while also keeping all constraints satisfied. When

an objective function is present, these problems are called optimization problems.

2.2.2 Mixed Integer Linear Programming

Solving these general mathematical models turns out to be extremely hard (TODO: cite). By limiting how a problem can be expressed in a model, it becomes much easier to build a solver which can efficiently solve those models. This is where Linear Programming (LP) comes in. In LP, all constraints must be linear equations. The objective function must also be a linear function. The decision variables must be real values.
Under those limitations, LP problems can be solved quickly and reliably (TODO: cite). However, the kinds of problems that can be modeled using LP is very limited. It is impossible to model "OR" ($\vee$) relations, conditionals ($\rightarrow$) and many other logical relations. It also prevents the use of common mathematical functions like the absolute value, among others (TODO: cite?).
Many of those limitations can be resolved by allowing some (or all) of the decision variables to be integers [11]. While variables could take on integer variables in LP, there is no way to limit the domain of a variable so it can only take on integer values. This extension is called Mixed-Integer Linear Programming (MILP). LP alone is not sufficient to model a trajectory planning problem, so MILP is required.
The addition of integer variables makes MILP much harder to solve than LP. MILP belongs to a class of problems called "NP-Complete". Problems which belong to this class tend to quickly become too difficult to solve in an acceptable amount of time when the size of the problem is increased. For MILP, the worst case difficulty grows exponentially with the amount of integer variables.

2.2.3 Arguments for MILP

The poor worst case performance is the main disadvantage of MILP, but it also has advantages over other approaches.
The main advantage is flexibility. Mathematical Programming is a form of declarative programming. In imperative programming, the programmer needs to describe step-by-step how the algorithm should solve a specific kind of problem. Even a small change to the problem may cause large changes to how the algorithm works.
In declarative programming, the programmer does not tell the computer exactly how to solve the problem. A general solver does the heavy lifting and finds the solution. This means that when the problem changes slightly, the model only needs to be updated to reflect that small change.
Performance could actually be an advantage. Planning a good trajectory is not a trivial problem, so performance is likely to be a concern for any competing algorithm. MILP solvers have been used in other industries for decades, so a lot of work has gone in to making those solvers fast and memory-efficient. The performance of MILP approach could be comparable to, or even exceed, the performance of other algorithms.

2.3 Time and UAV state

The trajectory planning problem can be represented using discrete time steps. The state of the UAV (position, velocity, etc.) is defined at each time step. Time steps are spaced out regularly, with Δt between them. The progression of time is modeled by Equation 2.1 and 2.2. $time_n$ represents the amount of time that has passed at a given time step n. Equation 2.1 initializes the time at the first time step to be zero. Equation 2.2 progresses time by Δt at each time step. In total, there are N time steps.

$$time_0 = 0 \tag{2.1}$$

$$time_{n+1} = time_n + \Delta t, \quad 0 \leq n < N - 1 \tag{2.2}$$

The position of the UAV at time step n is represented by p_n. Equation 2.3 initializes the position in the first time step to the starting position $\boldsymbol{p}_{start}$. Equation 2.4 updates the position of the UAV in the next time step, based on the velocity $\boldsymbol{v}_n$ in the current time step and time step size Δt. When a variable is in **bold**, it represents a vector instead of a scalar value.

$$\boldsymbol{p}_0 = \boldsymbol{p}_{start} \tag{2.3}$$

$$\boldsymbol{p}_{n+1} = \boldsymbol{p}_n + \Delta t * \boldsymbol{v}_n \quad 0 \leq n < N - 1 \tag{2.4}$$

The velocity of the UAV is modeled the same way as the position. Equation 2.5 initialized the velocity in the first time step, and Equation 2.6 updates it at each time step based on the acceleration $\boldsymbol{a}_i$. Other derivatives can be modeled like this as well.

$$\boldsymbol{v}_0 = \boldsymbol{v}_{start} \tag{2.5}$$

$$\boldsymbol{v}_{n+1} = \boldsymbol{v}_n + \Delta t * \boldsymbol{a}_n \quad 0 \leq n < N - 1 \tag{2.6}$$

2.4 Objective function

The objective for the solver is to minimize the amount of time steps before the UAV reaches the goal position. Equation 2.7 describes this. $done_n$ is a boolean variable which is *true* when the UAV has reached the goal on or before time step n. A boolean variable is an integer variable which can only be 0 (false) or 1 (true).

$$minimize \quad -\sum_{n=0}^{n<N} done_n \tag{2.7}$$

The value of $done_{n+1}$ at time step $n+1$ is true if either $done_n$ is true, or a state transition $cdone_{n+1}$ occurs. $cdone_{n+1}$ is true whenever all conditions for reaching the goal have been satisfied. This is expressed in Equation 2.8.

$$done_{n+1} = done_n \vee cdone_{n+1}, \quad 0 \leq n < N - 1 \tag{2.8}$$

This formulation is nearly correct, but fails to address two important assumptions. The first assumption is that the UAV has not reached its goal at the start of the trajectory. $done_0$ is not constrained by Equation **??**, so a valid (and optimal) solution would be setting $done_0$ to *true*. Equation 2.9 resolves this issue by forcing $done_0$ to be false.

$$done_0 = 0 \tag{2.9}$$

Similarly, no constraints actually force the UAV to reach its goal. The value of the objective function would be poor, but the solution would be valid. Equation 2.10 forces $done_{N-1}$ to be true at the last time step. In combination with Equation 2.8 and 2.9, this ensures that the UAV must reach its goal eventually for the solution to be valid.

$$done_{N-1} = 1 \tag{2.10}$$

To model the state transition, represented by $cdone_n$, Lamport's [9] state transition axiom method was used. This method separates the value of the state ($done_n$) from the requirements for transition ($cdone_n$). Equation 2.8 is the first part: It expresses that the state changes when a transition event occurs. It is not concerned with what leads to that state transition.
The second part is Equation 2.11: It describes the requirements for a state transition to occur. In this case, the UAV must be at its goal position (represented by $cdone_{p,n}$) and not be done already at time step n.

$$cdone_n = cdone_{p,n} \wedge \neg done_n \quad 0 \leq n < N \tag{2.11}$$

The equations discussed so far are all in a general form which apply to a UAV moving through a world with an arbitrary amount of spatial dimensions. For this thesis, the assumption is that the UAV moves through a 2D world. Starting from Equation 2.12, some equations only apply to 2D worlds for simplicity.
Equation 2.12 defined when the UAV is considered to be at the goal position. x_n and y_n are the UAV's 2D coordinates at time step n, while x_{goal} and y_{goal} are the coordinates of the goal position. The UAV has reached the goal if each coordinate is at most ϵp away from the goal's matching coordinate

$$cdone_{p,n} = |x_n - x_{goal}| < \epsilon_p \wedge |y_n - y_{goal}| < \epsilon_p, \quad 0 \leq n < N \tag{2.12}$$

Adding additional requirements for the state transition is easy. As an example, we may wish to ensure that the UAV stops at its goal. Equation 2.11 can be extended with a $cdone_{v,n}$ requirement as shown in 2.13. $cdone_{v,n}$ is defined by Equation 2.14, where vx_n and vy_n are the components of the UAV's velocity vector which must be smaller than some ϵ_v.

$$cdone_n = cdone_{p,n} \wedge cdone_{v,n} \wedge \neg done_n \quad 0 \leq n < N \tag{2.13}$$

$$cdone_{v,n} = |vx_n| < \epsilon_v \wedge |vy_n| < \epsilon_v, \quad 0 \leq n < N \tag{2.14}$$

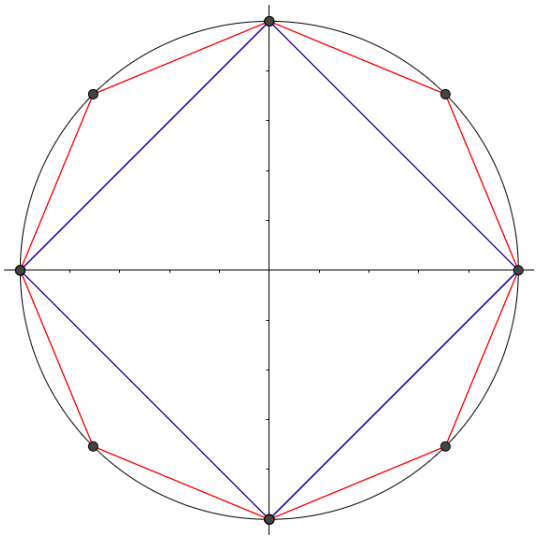


Figure 2.1: If the velocity is limited to a finite value, the velocity vector must lie within the circle centered on the origin with the radius equal to that value. This is represented by the black circle. This circle cannot be approximated in MILP, but it can be approximated using several linear constraints. The blue square shows the approximation using 4 linear constraints. The red polygon uses 8 linear constraints. As more constraints are used, the approximation gets closer and closer to the circle.

2.5 Vehicle state limits

Like any vehicle, a UAV has a certain maximum velocity. The model should contain constraints which prevent the UAV from moving faster than the maximum velocity. The magnitude of the velocity is the 2-norm of the velocity vector (which is Pythagoras' theorem for a 2D vector). This cannot be represented as a linear constraint because the components of the velocity vector need to be squared. However, the 2-norm can be approximated using multiple linear constraints.
Figure 2.1 visualizes how this works for the 2D case. All velocity vectors with a magnitude smaller than the maximum velocity are on the inside of the black circle with radius v_{max}. This cannot be expressed in MILP, but a regular polygon which fits inside the circle can be expressed. As the amount of vertices N_{vertex} making up the polygon increases, the approximation gets more and more accurate. The coordinates of the vertices of the polygon, $q_{x,i}$ and $q_{y,i}$ are defined by Equations 2.15-2.17 in counter-clockwise order.

$$\theta = \frac{2\pi}{N_{points}} \tag{2.15}$$

$$q_{x,i} = v_{max} * cos(\theta i), \quad 0 \leq i < N_{points} \tag{2.16}$$

$$q_{y,i} = v_{max} * sin(\theta i), \quad 0 \leq i < N_{points} \tag{2.17}$$

Like before, the equations are limited to the 2D case, but can be extended to 3D as well. Each edge of the polygon defines a line with slope a_i and intercept b_i, as determined by Equations 2.18-2.20.

$$\Delta q_{x,i} = q_{x,i} - q_{x,i-1}, \quad \Delta q_{y,i} = q_{y,i} - q_{y,i-1} \tag{2.18}$$

$$a_i = \frac{\Delta q_{y,i}}{\Delta q_{x,i}} \quad 0 \leq i < N_{vertex} \tag{2.19}$$

$$b_i = q_{y,i} - a_i q_{x,i} \quad 0 \leq i < N_{vertex} \tag{2.20}$$

Finally, the constraints can be constructed using the slope and intercepts of each line segment. For the edges on the "top" half of the polygon, the velocity vector must stay *below* those edges. This is expressed by Equation 2.21. Similarly, for the bottom half, the velocity vector must stay *above* those edges as expressed by Equation 2.22. If N_{vertex} is odd, the left-most edge of the polygon is vertical. Equation 2.23 handles this special case: the velocity vector must stay on the right of the left-most edge. TODO: add figure to explain better.

$$vy_n \leq a_i vx_n + b_i, \qquad \Delta q_{x,i} < 0, \quad 0 \leq n < N \tag{2.21}$$

$$vy_n \geq a_i vx_n + b_i, \qquad \Delta q_{x,i} > 0, \quad 0 \leq n < N \tag{2.22}$$

$$vx_n \geq q_{x,i}, \qquad \Delta q_{x,i} = 0, \quad 0 \leq n < N \tag{2.23}$$

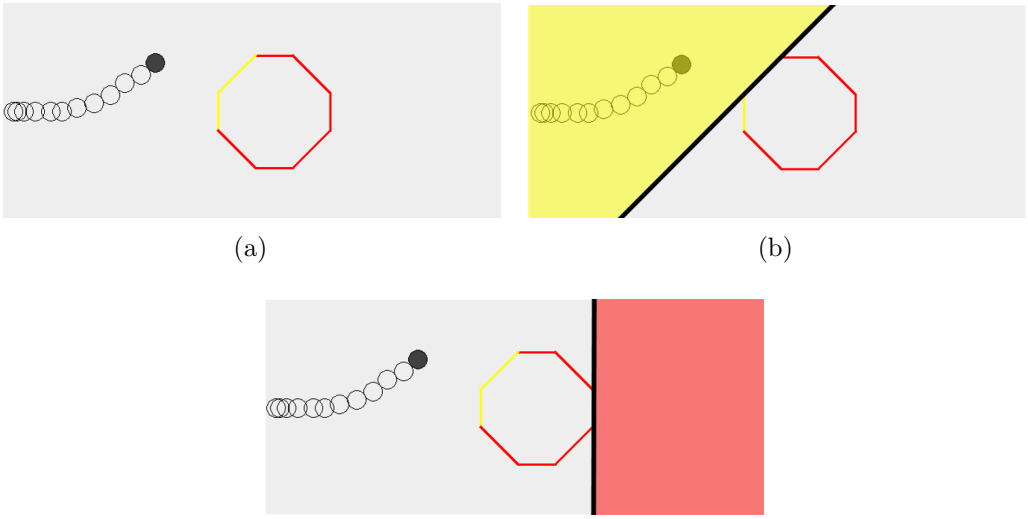
The acceleration and other vector properties of the vehicle can be limited in the same way.

2.6 Obstacle avoidance

The last element of the model is obstacle avoidance. Obstacles are regions in the world where the UAV is not allowed to be. This may be due to a physical obstacle, but it could also be a no-fly zone determined by law or other concerns.
Assuming that each obstacle is a 2D convex polygon, the constraints for obstacle avoidance are similar to those of the vehicle state limits in Section TODO:ref. However, the big difference is that the UAV must stay on the outside of the polygon instead of the inside.

2.6.1 Importance of Convexity

The maximum velocity is modeled without using any integer variables. This is possible because the allowed region for the velocity vector is a convex shape. A shape is convex if a line drawn between any two points inside that shape is also fully inside the shape. This is demonstrated in Figure 2.3a
This property does not hold for obstacle avoidance. If the UAV is on one side of an obstacle, it cannot move in a straight line to the other side of that obstacle. Because of that obstacle, the shape formed by all the positions where the UAV is allowed to



(a) (b)

(c)

Figure 2.2: A visual representation of how obstacle avoidance works. The top image shows the vehicle's current position as the filled circle, with its path in previous time steps as hollow circles. The color of the edges of the obstacle represent whether or not the vehicle is in the safe zone for that edge. An edge is yellow if the vehicle is in the safe zone, and red otherwise. The middle image shows the safe zone defined by a yellow edge in yellow. Note how the vehicle is on one side of the black line and the obstacle is entirely on the other side. The bottom image shows an edge for which the vehicle is not in the safe zone (represented in red this time). As long as the vehicle is in the safe zone of at least one edge, it cannot collide with the obstacle.

convex nonconvex

(a) TODO:convex (b) TODO:nonconvex

be is no longer convex. This can be seen in Figure 2.3b.
To express non-convex constraints, integer variables are needed. This causes the difference between what can be modeled in Linear Programming versus Mixed-Integer Linear Programming: non-convex constraints can be expressed in MILP, while this is not possible in LP.

2.6.2 Big M Method

Like with the maximum velocity in Section 2.5, the edges of the obstacle are used to construct the constraints. Each edge defines a line, as determined by Equations 2.18-2.20. If the obstacle is convex, the obstacle will entirely on one side of that line. If the UAV is on the other side, the one without the obstacle, it cannot collide with that obstacle. This region can be consider the safe region defined by that edge. As long as the UAV is in the safe region of at least one edge, no collisions can occur. Figure 2.2 demonstrates this visually.
A popular way to model this is the "Big M" method TODO:cite. For each edge, a

constraint is constructed which forces the UAV to be in the safe region for that edge. However, the UAV must only be in the safe region of at least one edge. This means that it must be possible to "turn off" constraints.
For each edge of an obstacle, a boolean *slack* variable is used to represent whether or the constraint for that edge is enabled. If this *slack* variable is *true* (equal to 1), the constraint is *disabled*. This is where the "Big M" comes in: the *slack* variable is multiplied by a very large number M on one side of the inequality constraint for that edge. M must be chosen to be large enough to ensure that ,when *slack* is *true*, the inequality is always satisfied.
In Equations 2.24-2.27 below, $q_{x,i}$ and $q_{y,i}$ are the 2D coordinates of vertex i the obstacle. The slope a_i and intercept b_i of edge i are calculated as in Equations 2.18-2.20. For every time step n:

$$y_n + M * slack_{i,n} \geq a_i x_n + b_i, \quad \Delta q_{x,i} < 0 \tag{2.24}$$
$$y_n - M * slack_{i,n} \leq a_i x_n + b_i, \quad \Delta q_{x,i} > 0 \tag{2.25}$$
$$x_n - M * slack_{i,n} \leq q_{x,i}, \quad \Delta q_{y,i} < 0, \quad \Delta q_{x,i} = 0 \tag{2.26}$$
$$x_n + M * slack_{i,n} \geq q_{y,i}, \quad \Delta q_{y,i} > 0, \quad \Delta q_{x,i} = 0 \tag{2.27}$$

Like in Equations 2.21 and 2.22, edges on the top and bottom of the obstacle are treated differently. Equation 2.24 covers the top edges. The UAV must be above the edge. If the UAV is not above the edge (when y_n is too small), enabling $slack_{i,n}$ will add M on the left-hand side of the inequality and ensure that it is always larger than the right-hand side. Equation 2.25 does the same for edges on the bottom. The inequality is flipped around so that the UAV must be below the edge This also means that M must be subtracted from the left-hand side of the inequality to ensure the constraint is always satisfied.
Vertical edges are also possible. A vertical edge may occur on the left or right side of the polygon, as covered in Equations 2.26 and 2.27 respectively.
Finally Equation 2.28 ensures that at least one of the slack variables must be false at each time step.

$$\neg \bigwedge_i slack_{i,n} \quad 0 \leq n \leq N \tag{2.28}$$

Equations 2.24 - 2.27 assume that the UAV has no physical size. The position of the UAV is the position of its center of mass. Because a UAV has certain physical dimensions, it will collide with an obstacle before its center of mass reaches the obstacle. The vehicle's shape is approximated as a circle with radius r. The vertical offset v_i required to move the line with slope a_i from the center of the circle to the edge is calculated by Equation 2.29. A geometric proof is given in Figure 2.4.

$$v_i = r\sqrt{1 + a_i^2} \tag{2.29}$$

With the vertical offset calculated, Equations 2.24 - 2.27 can be extended to Equations 2.30-2.31 which take the UAV size into account.

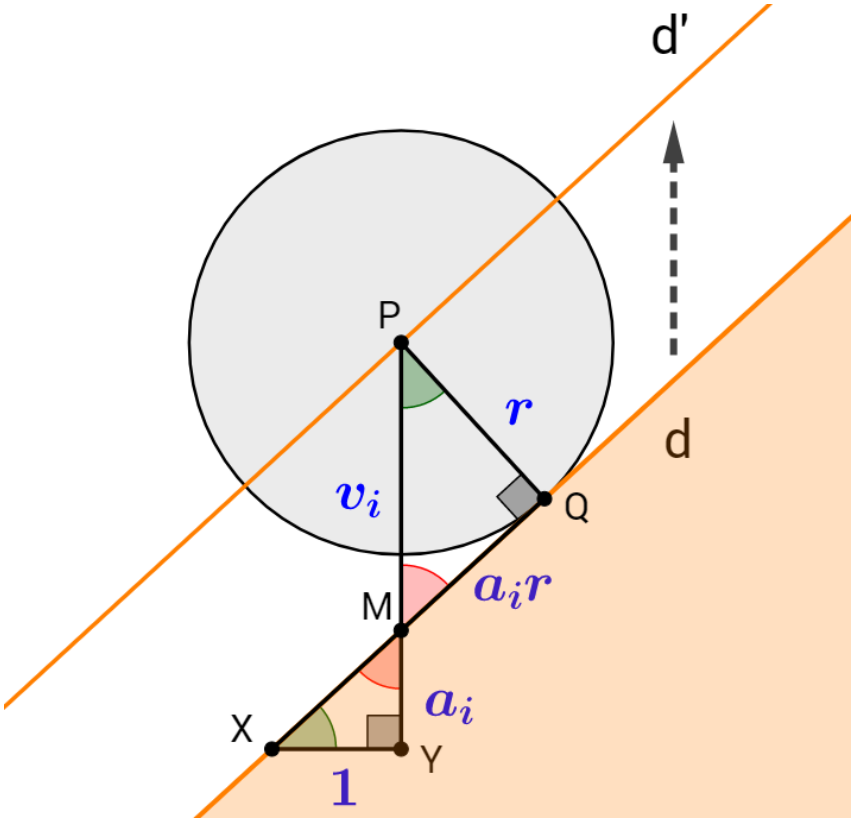


Figure 2.4: A geometric proof for the vehicle offset formula in Equation 2.29. Measures are marked in blue. P is the center of mass of the UAV. d is the line defined by edge i of an obstacle. The obstacle is the orange region. We wish to translate d vertically to d' such that when the UAV's center of mass P touches d', the circle with radius r (representing the UAV's shape, colored in grey) touches d in Q. If d' is used for the constraint instead of d, the UAV cannot collide with the obstacle. The difference between the intercepts of d and d' is $v_i = |PM|$. The slope of d' is a_i, so if XY is horizontal and $|XY| = 1$, then $|MY| = a_i$. Corner M is the same in both triangle PQM and XYM, as marked in red. Corner Q and Y are both right angles. This means PQM and XYM are similar triangles. As a result: if $|PQ| = r$, then $|QM| = a_i r$. Using Pythagoras' theorem: $v_i = \sqrt{r^2 + a_i^2 r^2} = r\sqrt{1 + a_i^2}$.

$$y_n + M * slack_{i,n} \quad - \quad v_i \quad \geq \quad a_i x_n + b_i, \qquad\qquad \Delta q_{x,i} < 0 \tag{2.30}$$

$$y_n - M * slack_{i,n} \quad + \quad v_i \quad \leq \quad a_i x_n + b_i, \qquad\qquad \Delta q_{x,i} > 0 \tag{2.31}$$

$$x_n - M * slack_{i,n} \quad + \quad r \quad \leq \quad q_{x,i}, \qquad \Delta q_{y,i} < 0, \quad \Delta q_{x,i} = 0 \tag{2.32}$$

$$x_n + M * slack_{i,n} \quad - \quad r \quad \geq \quad q_{y,i}, \qquad \Delta q_{y,i} > 0, \quad \Delta q_{x,i} = 0 \tag{2.33}$$

3

Segmentation of the MILP problem

3.1 Algorithm approach

TODO: small experiment to show bad performance

The MILP model described in section 2 is sufficient to solve the trajectory planning problem for short flights with few obstacles. However, it scales poorly when the duration of the flight or the amount of obstacles is increased. Mixed-Integer programming belongs to the "NP-Complete" class of problems [**?**]. Assuming there are n boolean variables, the worst case complexity is $O(2^n)$. An boolean variable is needed for every edge of every polygon, for every time step. By reducing both the amount of time steps needed and obstacles that need to be modeled, the execution time can be reduced dramatically.

The key insight that allows my algorithm to scale well beyond what's usually possible is that the path trajectory not need to be solved all at once. If the trajectory planning problem can be split into many different subproblems, each subproblem becomes easier to solve. The solution for each subproblem is a small part of the final trajectory. By solving theses subproblems sequentially, the final trajectory can gradually be constructed.

While diving the problem into subproblems does make things much easier to solve, it also has an important down side: Finding the fastest trajectory can no longer be guaranteed. Smaller subproblems make it easier to find a solution, but fundamentally the problem of finding the optimal trajectory is still just as hard. The necessary trade-off for better performance is that the optimal trajectory will likely not be found. Luckily, the optimal trajectory is often not required in navigation. A reasonably good trajectory will do.

3.1.1 The importance of convexity

While the worst case time needed to solve a MILP problem increases exponentially with the amount of integer variables, this is not the most useful way to measure the difficulty of a problem. Modern solvers are heavily optimized and are able to solve certain problems with many integer variables much faster than others. The key difference is the convexity of the solution space. Just like a circle is the solution space for "all points a certain distance away from the center point", the constraints used to model the trajectory planning problem form some geometric shape with a

dimension for every variable.
When only linear constraints with real values are used, the solution space will always be convex. It is this convexity that makes a standard linear program easy to solve. When integer variables are introduced, it is possible to construct solution spaces which are not convex. As the solution space becomes less and less convex, the problem becomes harder to solve. Integer variables can be seen as a tool which allows non-convex solution spaces to be modeled. When trying to improve the difficulty of a problem, the actual goal is making the problem more convex (or smaller, which always helps). Reducing the amount of integer variables is only a side effect.
This insight is critical when determining how to divide the trajectory problem into smaller subproblems, and which obstacles are important for each subproblem.

3.1.2 General Algorithm Outline

Algorithm 1 General outline

```
1: T ← {}                                          ▷ The list of solved subtrajectories
2: path ← Theta*(scenario)
3: events ← FindTurnEvents(path)
4: segments ← GenSegments(path, events)
5: for each segment ∈ segments do
6:     UpdateStartState(segment)
7:     GenSafeRegion(scenario, segment)
8:     GenSubMILP(scenario, segment)
9:     T ← T ∪ { SolveSubMILP }
10: end for
11: result ←MergeTrajectories(T)
```

Algorithm 1 shows the general outline of the algorithm. It consists of two phases. The first phase gathers information about the trajectory planning problem. A Theta* path planning algorithm is used to find an initial path (line 2). From this initial path, turn events are generated (line 3). These turn events mark where the trajectory will have to turn.
The seconds stage builds and solves "segments". Each segment represents a single sub-trajectory. The segments contain the information needed to build the corresponding MILP subproblem, which can in turn be solved by the solver. First, each segment is generated (line 3) from the turns found in the previous step. Before the MILP sub-problem can be generated and solved (line 8-9), a heuristic selects several obstacles to be modeled in the problem. A genetic algorithm generates a safe region which is allowed to overlap those selected obstacles only (line 7). To ensure a seamless transition between two consecutive sub-trajectories, the starting state for the UAV in the MILP current segment is updated to match the final state of the UAV in the previous segment (line 6). Finally, the sub-trajectories are merged together to form the final trajectory (line 11).

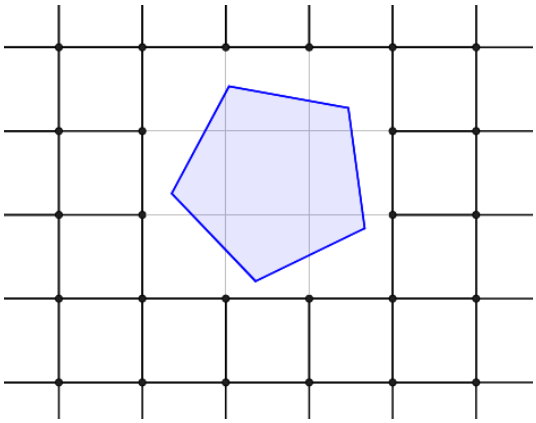


Figure 3.1: An example of how a grid is used to build the graph for the path finding algorithm. Each point is a node on the graph. If two points are connected by a line, their nodes in the graph also are connected by an edge. Diagonal edges are not shown here for clarity.

3.2 Finding the initial path

The first step in Algorithm 1 is finding the Theta* path (line 2). This path will be used to divide the trajectory planning problem into segments. The MILP-problem generated from each of those segments needs an intermediate goal to get the UAV closer to the final goal position. These intermediate goals will be determined by the Theta* path.
This path is not only useful to guide the trajectory towards the goal. It is also lets the algorithm determine where the turns will be in the trajectory. reason!!!

3.2.1 A*

Theta* is a variant of the A* path planning algorithm. In A*, the world is represented as a graph. Each possible position is represented as a node, with edges between nodes if one position can be reached from the other. The distance between connected positions is represented as a weight or cost on each edge.
Planning a path consists of picking a start and goal node. The A* algorithm will traverse the edges between nodes, keeping track of which edges it traversed to reach a certain node. When the algorithm reaches the goal node, the nodes visited to reach that goal node are the path from the start node to the goal .
In this case, the world the UAV travels through is a continuous (2D) world. The graph for A* star is generated by overlaying a grid on the world. Nodes are placed at each intersection of the grid, as long as they are not inside obstacles. Each node is also connected to its neighbors by moving horizontally, vertically or diagonally along the grid. Figure **??** shows an example of this.

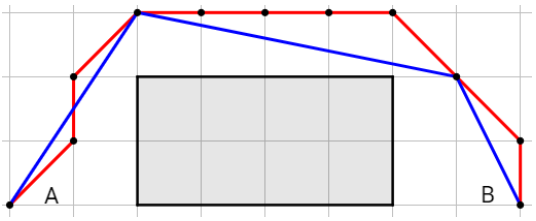


Figure 3.2: The red line shows a typical A* path, compared to the path found by Theta*. The gray rectangle is an obstacle.

3.2.2 Reason for Theta*

A* finds the shortest path through the graph. However, this graph is only an approximate representation of the actual continuous world. An A* path will only travel along the edges of the graph. This means that the path can only travel horizontally, vertically and possibly diagonally. If the shortest path between two points is at another angle, the A* path will contain zig-zags or detours because it is limited to traveling along the grid.
Theta* solves this problem. It is nearly identical to A*, but it allows the path to travel at arbitrary angles. It still traverses the graph using the edges between nodes, but does not restrict the path to only following those edges.

3.2.3 Theta* implementation

TODO: paraphrased! CITE!! Algorithm 2 shows how Theta* is implemented. It uses the following elements:

- the g-value $g(s)$ is the length of the shortest path between the start node and s.
- a function $c(s, s')$ which returns the distance between node s and s'.
- a heuristic function $h(s)$, which approximates the path distance left before the goal position s_{goal} is reached. The straight line distance between s and the s_{goal} is used, such that $h(s) = c(s, s_{goal})$. An admissible heuristic function must always be an underestimation of the actual path distance to the goal, which is ensured by using the straight line distance.
- a function $parent(s)$ which returns the node before s in the path. When the parent of a node is $null$, it is either not part of the path or the first node of the path.
- a priority queue *queue*. This is a queue of nodes to expand next. Each node s is added with a value x using the *queue*.Insert(s,x) method. If s is already in *queue*, its value is updated to x. The *queue*.Pop() method removes and returns the node s with the lowest value x.
- a set *expanded* which contains all nodes which have already been expanded.

Algorithm 2 Theta* Implementation

```
1: function Theta*(scenario)
2:     g(v_start) ← 0
3:     parent(v_start) ← null
4:     queue ← ∅
5:     queue.Insert(v_start, g(v_start) + h(v_start))
6:     expanded ← ∅
7:     while queue ≠ ∅ do
8:         s ← queue.Pop()
9:         if s = v_goal then
10:            return s.GetPath()
11:        end if
12:        expanded ← expanded ∪ {s}
13:        for each s' ∈ GenerateNeighbors(s) do
14:            if s' ∉ expanded then
15:                UpdateVertex(s,s')
16:            end if
17:        end for
18:    end while
19:    return "no path found"
20: end function
21: function UpdateVertex(s,s')
22:    if LineOfSight(parent(s),s') then
23:        s_parent ← parent(s)
24:    else
25:        s_parent ← s
26:    end if
27:    if g(s_prev) + c(s_prev, s') < g(s') then
28:        g(s') ← g(s_parent) + c(s_parent, s')
29:        parent(s') ← s_parent
30:        queue.Insert(s',g(s') + h(s'))
31:    end if
32: end function
```

- a function GenerateNeighbors(s) which generates and returns the neighbors of node s. These are the neighboring positions on the grid around s

Initialization When the algorithm initializes, the g-value of the start node v_{start} is set to zero and its parent is set to *null* (line 2-3). The priority queue *queue* is initialized, and v_{start} is added to it with value $g(v_{start}) + h(v_{start})$. The value attached to a node v in the priority queue is the shortest possible length of a path going from the start v_{start} to the goal v_{goal}, while going through v. The g-value is the length of the path between v_{start} and v, while $h(s)$ is an estimate of the length of the path between v and v_{goal}.

Main loop *queue*.Pop() always removes the node with s the lowest value (line 8). This means that as more nodes get added to *queue*, the algorithm will always explore the "most promising leads" first. If $s = v_{goal}$, the goal has been reached and the path is returned (line 9-11). Otherwise, s is added to *expanded* (line 12). This prevents s from being added to *queue* again. Line 13 generates every neighbor s' of s according to the grid and obstacles, as seen in Figure **??**, "expanding" node s. If the neighbor s' has not been expanded yet, UpdateVertex(s,s') is called (line 14-16).

UpdateVertex So far, the algorithm is identical to A*. The only difference between A* and Theta* are lines 22-26. Line 22 checks if the parent of s, which is the node before s in the path, can be connected in a straight line to s'. Going from $parent(s)$ to s' directly is always shorter than going from $parent(s)$ to s and from s to s' due to the triangle inequality. If $parent(s)$ and s' are in line of sight, the path should be constructed from $parent(s)$ to s', skipping s. Otherwise, the path goes through s, which is the behavior of A*. The choice of which node should be parent of s' is stored as s_{parent}
Line 27 checks if the path through s_{parent} to s' is the shortest path to s' found so far. If this is not the case, a shorter path to s' exists so the path through s_{parent} can safely be ignored. Otherwise, the g-value of s' is updated to the length of the path to s_{parent}, $g(s_{parent})$ plus the distance between s_{parent} and s', $c(s_{parent}, s')$ (line 28). The shortest path to s' is updated by setting $parent(s')$ to s_{parent} (line 29). Finally, s' is added to *queue* to be expanded further.

3.2.4 Performance improvements

By introducing the line of sight check, Theta* is considerably slower than A* in large worlds with many obstacles. The goal of this thesis is to improve scalability of trajectory planning to large and complex worlds, so the preprocessing phase must also scale well.
To help Theta* scale, the world is divided into rectangular sectors. When the world is first loaded, the algorithm determines in which sector(s) each obstacle is placed, creating an index mapping sectors to obstacles.
When the line of sight check is executed, the algorithm determines which sectors the line crosses. This is usually just one or two sectors. By using the index, only the

obstacles in those sectors need to be considered for the line of sight check. This is much faster than having to check every obstacle in the entire world every time.

3.3 Detecting turn events

According to Hypothesis **??**, the degree of non-convexity around the trajectory is responsible for the poor performance of MILP trajectory planning problem. This local degree of non-convexity around the trajectory is the amount of distinct convex shapes the trajectory needs to pass through to reach the goal position, such that every point in the trajectory lies within at least one shape.
Within a single convex shape, by definition, it is always possible to move in a straight line from one side of the shape to the other. As a consequence, if multiple convex shapes are needed, the trajectory needs to make a turn. If the turn is not needed, that implies the trajectory can go in a straight line, which means that only a single convex shape is needed.
Because of this, the turns in the trajectory and the degree of non-convexity are directly related. Every turn in the trajectory is a manifestation of a transition between two or more convex shapes, and thus contributes to the degree of non-convexity of the entire trajectory.
Solving the trajectory in smaller segments reduces the degree of non-convexity in each segment, making them easier to solve. If Hypothesis **??** is true, minimizing the amount of turns in each segment will improve performance even more.
While the Theta* path is only a rough approximation of the trajectory, it does have turns in roughly the same places as the trajectory will have. Finding those turns allows the algorithm build segments such that the amount of turns in each segment is minimal.
Because Theta* is used to generate the initial path, finding the turns is easy. When two nodes are in each other's line of sight, it is possible to construct a convex shape around those points. When they are not within line of sight, which is when Theta* keeps the previous node in the path, this is not possible. Using the same reasoning as above, the nodes in the Theta* path must always coincide with turns in both the Theta* path and the trajectory.
While every node in the Theta* path (except the start and goal) coincide with a turn, they can be close together. When two or more consecutive nodes turn in the same direction (clockwise or counter-clockwise) and are close enough together, they can be considered to represent a single turn. The algorithm groups those nodes together in a turn event. Each turn event contains one or more nodes. A turn event predicts the existence of a turn in the MILP trajectory based on the Theta* path, bridging the gap between them.

3.3.1 Algorithm Implementation

Algorithm 3 processes the Theta* path to generate turn events. Two factors determine whether or not nodes are grouped together into events:

- The turn direction: a node v in the path can either turn the path clockwise or counter-clockwise, as determined by TURNDIR(v) Two nodes with a different turn direction cannot be in the same turn event
- The maximum distance between two nodes in a turn event: Nodes which are too far from each other should not be merged. This distance Δmax is based on a turn tolerance parameter multiplied by the UAV's maximum acceleration distance (MAD).

Maximum Acceleration Distance With a_{max} and v_{max} respectively the UAV's maximum acceleration and velocity, the time needed to reach the maximum velocity is $t_{max} = v_{max}/a_{max}$. The maximum acceleration distance in the distance traveled in that time, which is $t_{max}^2/2 = v_{max}^2/2a_{max}$. The MAD is used in several places in the algorithm as an approximation of the distance in which the UAV can recover from a maneuver. In $1 * MAD$, the UAV can accelerate to any velocity from zero, or it can stop from any velocity. In $2 * MAD$, the UAV can transition from any velocity vector to any other velocity vector. It is an approximation of the distance at which the presence or absence of an event can influence the UAV.
In this case, the MAD is relevant because turns require the velocity vector of the UAV to change from before the turn to after the turn. If two nodes have a distance of more than $2 * MAD$ between them, the turns at each node can be taken independently as distinct turns. TODO: fig!

In the Theta* path, all but the first and last nodes are turns in the path. The second node is always the first node in a new turn event (line 5). Subsequent nodes in the path which are not too far away from the previous node (line 9) and turn in the same direction (line 12) are added to the current turn event (line 15). The maximum distance between nodes in the same turn is Δmax. Once a node is found which does not belong in the event, the event is stored (line 18), a new event is created for that node (line 5) and the process repeats until no more nodes are left.

3.4 Generating path segments

Once the turn events are found, the next step is generating the segments. Each segment defines a single MILP problem whose solution is a small part of the desired trajectory. As argued above, each turn event should be solved in a separate MILP problem to keep the solve times low.
Algorithm 4 calculates the boundaries of the segments, based on the Theta* path and the turn events. For the best performance, these segments should be as small as possible. However, performance is not the only factor. Stability of the algorithm is also important. Each segment needs to be large enough so the UAV can safely approach and exit each turn. This can be guaranteed by ensuring a segment always starts at least the maximum acceleration distance (MAD) before the turn event. The distance between the start of the segment and the turn event in that segment is called the expansion distance. When the expansion distance is least $1 * MAD$, the

Algorithm 3 Finding Turn Events

```
1:  function FindTurnEvents(path)
2:      Δmax ← max. acc. distance * turn tolerance
3:      events ← {}                          ▷ The list of turn events found so far
4:      i ← 1                                ▷ Skip the start node, it can't be a turn
5:      while i < |path| − 1 do              ▷ Skip the goal node
6:          event ← {path(i)}                ▷ Start new turn event
7:          turnDir ← TurnDir(path(i))
8:          i ← i + 1
9:          while i < |path| − 1 do
10:             if ||path(i − 1) − path(i)|| > Δmax then
11:                 break                    ▷ Node is too far from previous
12:             end if
13:             if TurnDir(path(i)) ≠ turnDir then
14:                 break                    ▷ Node turns in wrong direction
15:             end if
16:             event ← event ∪ {path(i)}    ▷ Add to event
17:             i ← i + 1
18:         end while
19:         events ← events ∪ {event}
20:     end while
21:     return events
22: end function
```

Figure 3.3: The left image shows the results after the Theta* algorithm has executed. The blue shapes are obstacles, while the gray line is the Theta* path. The right image shows the results after the path is segmented. Extra nodes have been added to the Theta* path, as marked by the green circles. These circles depict the transitions between segments.

UAV can come to a complete stop before it reaches the turn. If the UAV can safely come to a stop, it can also slow down to an appropriate speed to safely navigate the turn. Extending the segment beyond the end of the turn event also ensures that the

Figure 3.4: The left image shows the result after the genetic algorithm has executed. The obstacles in red have been selected to be modeled in the MILP problem. The dark grey shape is the convex allowed region generated by the genetic algortihm. Note how it does not overlap with any of the blue obstacles. The right image shows the final result. The trail of circles show the path of the vehicle up to the current time step, which is represented by the filled circles. The red and yellow colors depict the same information as in figure 2.2

UAV must have completely navigated the turn by the end of the segment. Ensuring that the UAV can always safely navigate a turn means that the MILP solver can always find a feasible trajectory for that segment. Because of this, ensuring safety also ensures stability.
Increasing the approach margin beyond the minimum required for stability can improve the speed of the trajectory. An approach margin of $1 * MAD$ is considered safe because the UAV can still "slam on the breaks" to slow down in time for the turn. However, a larger expansion distance gives the UAV more time and space to maneuver so it can efficiently navigate the turn. The ratio between the expansion distance and the MAD is called the approach margin: $expansiondistance = approachmargin * MAD$. Since the MAD is determined by the UAV's properties, the approach margin is used as the parameter which controls the expansion distance.

3.4.1 Implementation

To generate the segments, Algorithm 4 considers each turn event in turn (line 5-6). $lastEnd$ is always the end of the last segment that has been generated. It is initialized with the start position of the UAV (line 4). The algorithm considers turn events, but consecutive turn events may have a large distance between them. In those cases, additional segments (straight, without turns) may need to generated to "catch up" with the start of the turn events. When straight segments need to be inserted, the $catchUp$ flag is set to $true$. $catchUp$ starts as $true$ to catch up from the start position of the UAV to the first turn event (line 3).

First, consider the case in which there is no need to catch up to generate the segment for turn event i. The start of the segment is the end of the last segment,

Algorithm 4 Generating the segments

```
 1: function GenSegments(path, events)
 2:     segments ← {}
 3:     catchUp ← true
 4:     lastEnd ← path(0)
 5:     for i ← 0, |events| − 1 do
 6:         event ← events(i)
 7:         if catchUp then
 8:             ExpandBackwards(event.start)
 9:             AddSegments(lastEnd,event.start)
10:             lastEnd ← event.start
11:         end if
12:         nextEvent ← events(i + 1)
13:         if nextEvent.start is close to event.end then
14:             mid ← middle between event & nextEvent
15:             AddSegments(lastEnd,mid)
16:             lastEnd ← mid
17:             catchUp ← false
18:         else
19:             ExpandForwards(event.end)
20:             AddSegments(lastEnd,event.end)
21:             lastEnd ← event.end
22:             catchUp ← true
23:         end if
24:     end for
25:     AddSegments(lastEnd,path(|path| − 1))
26:     return segments
27: end function
```

lastEnd. The desired end of the segment is found by expanding the end of the turn event forwards along the path by a distance equal to the expansion distance. However, This may not be possible or desirable if the next turn event $(i + 1)$ is too close. Two events are considered close to each other if they are separated by less than three[1] times the expansion distance (line 13).
When the current turn event and the next are too close, the middle *mid* between those turn events is used as the boundary between the segments. In this case, the current segment is generated from *lastEnd* to *mid* (line 14-16). The next turn event is nearby, so there is no need to catch up (line 17).
When there is plenty of distance between the current turn event and the next, this is not necessary. The end of the turn event *event.end* can be expanded forwards along the path by the full expansion distance (line 19-21). There is some distance between

[1] Requiring three (instead of two) times the expansion distance as separation between turn events ensures that the segment between those turns is also at least as long as the expansion distance. This prevents some issues that can occur with very short segments.

this turn event and the next, so catching up will be required before the next event (line 22).

Before each turn event is considered, the algorithm checks if it needs to catch up first (line 7). If so, the start of the turn event is expanded backwards along the path by the desired expansion distance (line 8). One or more straight segments are added between the end of the last segment *lastEnd* and the (backwards expanded) start of the current event *event.start* (line 9-10). There are no turns in those straight segments, so their size can be kept small without the risk for stability issues.

3.4.2 Amount of time steps

The amount of time steps in the MILP problem needs to be determined ahead of time. The length of the Theta* path is used to estimate how much time is needed for the UAV to reach the end of each segment. To ensure that there are always enough time steps, a conservative estimate is used. This estimate assumes that the UAV starts the segment while stopped. Afterwards, the UAV accelerates towards the turn in the segment and comes to a stop at the turn. Finally, the UAV accelerates again from the turn and stops again at the end of the segment. The time needed to complete those actions is multiplied by a multiplier parameter.

3.5 Generating the active region for each segment

The last step of preprocessing determines which obstacles will be modeled in the MILP problem for each segment. Segmentation already reduces both the amount of time steps needed and the nonconvexity in each segment. However, modeling a large amount of obstacles still reduces the performance to unacceptable levels.
Not every obstacle needs to be modeled in the MILP subproblems to avoid collisions. Only the obstacles in the neighborhood around each segment are relevant.
Furthermore, not all obstacles in the neighborhood actually need to be modeled either. There is a fundamental difference between the obstacles on the inside of the turn and those on the outside. Obstacles on the inside of a turn are the "cause" for that turn. They cause the non-convexity around the trajectory. Without those obstacles, the UAV could move in a straight line in a convex neighborhood. Obstacle avoidance for the inner obstacles must reduce convexity
The same is not true for obstacles on the outside of the turn. Without those obstacles the optimal trajectory may have a slightly different shape, but the turn would still be there. As a result, obstacle avoidance for the outer obstacles does not necessarily have to reduce convexity.

Figure 3.5 shows this difference. The UAV cannot collide with obstacles on the outside of the turn as long as it stays inside the colored convex polygon. As long as this convex polygon does not overlap any of the obstacles on the outside of the corner and the UAV is constrained to stay within the polygon, those obstacles do not need to be modeled in the MILP problem. Only the few obstacles on the inside

Figure 3.5: genetic convexity example

of the turn will be modeled.

The obstacles to be modeled in the MILP problem are selected by calculating the convex hull of several important points of the segment. The algorithm use These are the start position, goal position and all nodes on the Theta* path between them. (TODO: more detail?). Any obstacle which overlaps this convex hull is on the inside of the turn and will be modeled in the MILP problem. In some cases, obstacles on the outside of the turn can be very restrictive. Due to the inherent randomness of the genetic algorithm, modeling those restrictive obstacle in the MILP problem as well improves stability.
The convex hull can be considered a safe region. If the UAV stays inside this region, it cannot collide with obstacles since any obstacle that overlaps with the safe region is modeled in the MILP problem. However, this safe region restricts the movements of the UAV more than necessary.
To make the safe region less restrictive, we use a genetic algorithm which attempts to grow it. TODO: FIG.

3.5.1 Implementation of the genetic algorithm

A genetic algorithm is an algorithm which evolves solutions using a process inspired by natural selection in biological evolution. Genetic algorithms typically use a population of individuals which compete with each other. Each individual has a genome consisting of chromosomes, which in turn consist of individual genes. The genome determines the traits of each individual, also called the phenotype. The structure and quantity of the genes depends on the kind of problem that the genetic algorithm should solve.
Like in biology, the individuals can produce offspring. This offspring can be a crossover between multiple parent individuals, or a mutated version of a single parent. An important part of evolution is the concept of "survival of the fittest". A genetic algorithm improves its population by letting individuals compete. The losing individuals get eliminated from the population, while those who remain get the opportunity to create offspring. This competition is based on a fitness function. Each individual represents a possible solution for a problem. The fitness function scores the individuals based on how well they solved the problem.

Algorithm 5 shows the implementation of the genetic algorithm. In this implementation, each individual in the population represents a single legal polygon. A legal polygon is convex, does not self-intersect, can only overlap with the selected obstacles and contains every node in the Theta* path for that specific segment. The latter requirement prevents the polygon from drifting off. Each individual has a single chromosome, and each chromosome has a varying number of genes. Each gene represents a vertex of the polygon.
The only operator is a mutator (line 4). Contrary to how mutators usually work, the

Algorithm 5 Genetic Algorithm

```
1: function GenSafeRegion(scenario, segment)
2:     pop ← SeedPopulation
3:     for i ← 0, N_gens do
4:         pop ← pop∪ Mutate(pop)
5:         Evaluate(pop)
6:         pop ← Select(pop)
7:     end for
8:     return BestIndividual(pop)
9: end function
10: function Mutate(pop)
11:     for each individual ∈ pop do
12:         add vertex with prob. P(add vertex)
13:         OR remove vertex with prob. P(remove vertex)
14:         for each gene ∈ individual.chromosome do
15:             randomly nudge vertex
16:             if new polygon is legal then
17:                 update polygon
18:             else
19:                 try again at most N_attempts times
20:             end if
21:         end for
22:     end for
23:     return BestIndividual(pop)
24: end function
```

mutation does not change the original individual. This means that the every individual can be mutated in every generation, since there is no risk of losing information. This mutator can add or remove vertices of the polygon by adding or removing genes (lines 12-13), but only if the amount of genes stays between a specified minimum and maximum. The mutator attempts to "nudge" every vertex/gene to a random position inside a circle around the current position (line 15). If the resulting polygon is not legal, it retries a limited number of times (line 16-19).
Tournament selection is used as the selector, with the fitness function being the surface area of the polygon (line 5-6). Fig. **??** Shows the obstacles modeled in the MILP problem in yellow and red, as well as the polygon generated by the genetic algorithm in dark gray.

4

Extensions

4.1 Solver-specific improvements

The main principles that drive the algorithm work regardless of which solver is used. However, the implementation can be improved by using more specific features of the solver. For this thesis, IBM CPLEX is used as the MILP solver. This is one of the fastest, proprietary solvers available on the market.

4.1.1 Indicator constraints

The Big M method to turn constraints on or off is functional, but there are some issues with it. M has to be big enough so it can always "overpower" the rest of the inequality. If M is too low, incorrect behavior may occur. However, if M is very large, CPLEX may have numerical difficulties or even find incorrect results [1].
Indicator constraints are a solution for this problem. The goal of the large M is to overpower the inequality so the constraint can be turned on or off based on a boolean variable. Indicator constraints allow constraints to be turned on or off, based on other constraints. They provide a direct way to model an "if/then" relation. Equation 4.1 is a modified version of the obstacle avoidance constraints 2.30 - 2.33. If $slack_{i,n}$ is not true, then the matching constraint on the right side must be true.

$$\neg slack_{i,n} \rightarrow \begin{cases} y_n - v_i & \geq & a_i x_n + b_i, & \Delta q_{x,i} < 0 & \\ y_n + v_i & \leq & a_i x_n + b_i, & \Delta q_{x,i} > 0 & \\ x_n + r & \leq & q_{x,i}, & \Delta q_{y,i} < 0, & \Delta q_{x,i} = 0 \\ x_n - r & \geq & q_{y,i}, & \Delta q_{y,i} > 0, & \Delta q_{x,i} = 0 \end{cases} \tag{4.1}$$

4.1.2 Max time

When the MILP problem is sufficiently difficult to solve, it may take a long time before the solver can find the optimal solution. To ensure an upper limit on computation time, CPLEX accepts a maximum solve time parameter. When the maximum time has expired, it will return the best solution found so far.
In the experiments, the maximum solve time was typically 120 seconds per segment.

[1] `http://www-01.ibm.com/support/docview.wss?uid=swg21400084`

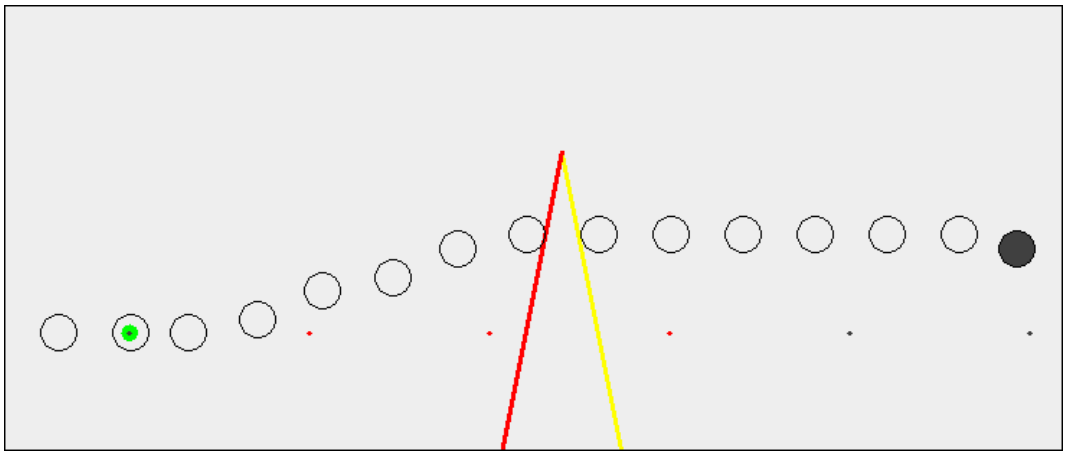


Figure 4.1: An example of corner cutting

The goal is for every segment to be relatively easy to solve, so if no solution can be found in two minutes it counts as a failure.

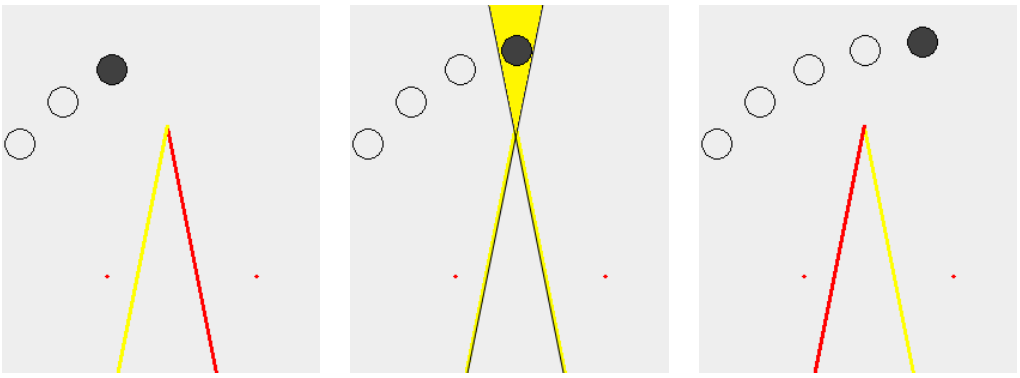
4.1.3 Max delta

During testing it became clear that the solver often spends a relatively long time trying to improve trajectories which are already nearly optimal, or optimal but not yet proven to be optimal. CPLEX provides a maximum delta parameter. The delta is the difference between the best solution found so far and the upper bound for the optimal solution. If the delta is below this maximum delta, the solvers stop executing and returns the best result. When this value is small, this can reduce some of the execution time while barely changing the quality of the trajectory.

4.2 Corner cutting

The MILP problem uses discrete time steps to model the changes in the UAV's state over time. An issue with this approach is that constraints are only enforced at those specific time steps. This allows the UAV to cut corners or even move through obstacles entirely if the vehicle is moving fast enough. Figure 4.1 shows an example of this. Each time step on its own is a valid position, but a collision is ignored between the time steps.
Using the indicator constraint notation, Equation 4.2 and 4.3 prevent collisions with obstacle o at time step n. Each edge of each obstacle has an associated *slack* variable, which determines whether or not the UAV is on the safe side for that edge.

$$\neg\, slack_{i,n} \rightarrow \begin{cases} y_n - v_i & \geq & a_i x_n + b_i, & \Delta q_{x,i} < 0 & \\ y_n + v_i & \leq & a_i x_n + b_i, & \Delta q_{x,i} > 0 & \\ x_n + r & \leq & q_{x,i}, & \Delta q_{y,i} < 0, & \Delta q_{x,i} = 0 \\ x_n - r & \geq & q_{y,i}, & \Delta q_{y,i} > 0, & \Delta q_{x,i} = 0 \end{cases} \tag{4.2}$$



(a) (b) (c)

Figure 4.2: These images are three consecutive time steps which demonstrate how the corner cutting prevention works. In 4.2a, the UAV is in the safe region of the left edge (which is indicated by the yellow color). In 4.2c, the UAV is in the safe region for the right edge, but not the left edge. To ensure that the UAV does not cut the corner, the UAV must enter the safe region of the right edge before it exits the safe region of the left edge. The intersection between those two safe regions is the inverted yellow triangle in 4.2b. If the UAV spends at least one time step in that yellow, it cannot cut the corner.

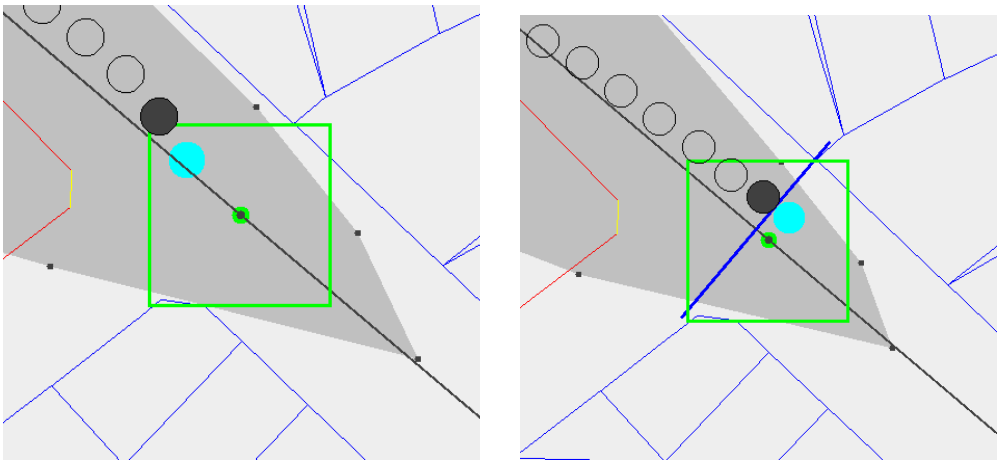
$$\neg \bigwedge_i slack_{i,n} \quad 0 \leq n \leq N \tag{4.3}$$

Richards and Turnbull[**?**] proposed a method which prevents corner cutting. In their method, the UAV is consider on the safe side of an edge only if that is true for two consecutive time steps. This is visualized in Figure 4.2. They apply the same constraints again, but this time on the position of the UAV in the last time step:

$$\neg\ slack_{i,n} \rightarrow \begin{cases} y_{n-1} - v_i & \geq & a_i x_{n-1} + b_i, & \Delta q_{x,i} < 0 \\ y_{n-1} + v_i & \leq & a_i x_{n-1} + b_i, & \Delta q_{x,i} > 0 \\ x_{n-1} + r & \leq & q_{x,i}, & \Delta q_{y,i} < 0, \quad \Delta q_{x,i} = 0 \\ x_{n-1} - r & \geq & q_{y,i}, & \Delta q_{y,i} > 0, \quad \Delta q_{x,i} = 0 \end{cases} \tag{4.4}$$

4.2.1 Goal conditions

Forcing the UAV to exactly reach each intermediate goal position is overly restrictive. Because the goal conditions are only check for each time step, the arrival at the goal position must line up with a time step. If this is not the case, the UAV could be right before the goal position on one time step, and already have passed the goal position the by the next time step. The result is that the UAV will slow down near the end of each segment so it can exactly line up with the goal position for that segment. This effect gets worse as the velocity increases.



(a) (b)

Figure 4.3: 4.3a shows the UAV right before a reaching its goal. The blue circle shows the position of the UAV at the segment transition. The goal position is the green dot, while the square around it show the tolerance region where the goal is considered to be reached. Note how the segment transition happens earlier because of the tolerance region. 4.3b shows the same, only this time a (blue) finish line is added. This time, the UAV needs to cross the finish line so it so the segment transition is roughly as far along the path as planned.

By allowing some difference between the goal position and the position of the UAV, this behavior can be resolved. However, this tolerance on the UAV's position means that the goal condition tends to be satisfied before the UAV has actually reached the goal. This is visible in Figure 4.3a. By adding a finish line perpendicular to the path at the goal position and forcing the UAV to cross this line, this early finish can be counteracted. The tolerance makes the trajectory less restricted, while the finish line ensures that the segment transitions happen at the right time. This can be seen in Figure 4.3b.

4.3 Stability Improvements

4.3.1 Maximum goal velocity

When two corners are close to each other, it may not be possible to expand each corner outwards by the full expansion distance. In that case, the middle between those corners is chosen as the transition between the the segments for each corner. This ensures that both corners get a fair share of the space between them. However, it breaks the assumption behind the corner expansion. If the velocity after the first corner is high, due to the reduced approach distance, the UAV may not be able to stop in time for the corner. In this situation, no solution will be found in the second segment. Figure 4.4 shows a situation when this may be necessary.
As a solution for this, the UAVs velocity at the goal of the first segment can be

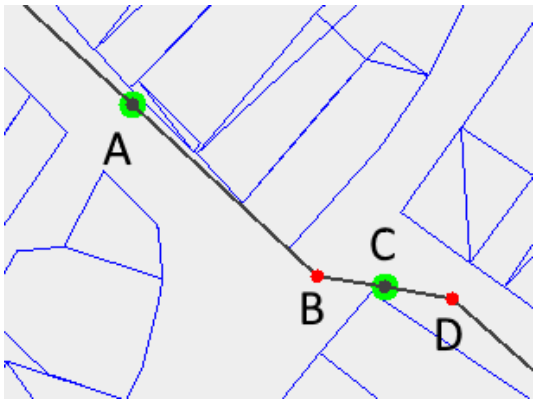


Figure 4.4: A visual demonstration of when a maximum goal velocity is used. Points B and D are individual turn events. The segment for event B starts at A, with $|AB|$ being the desired expansion distance for the segment. However, because D is so close to B, the end of the segment C cannot be placed at the desired expansion distance from B. Instead, C is placed in the middle between B and D, such that $|BC| = |CD|$. The first segment solves the trajectory from A to C past turn event B, the second segment starts as C, past D and onwards. The goal is to ensure that the UAV can still safely stop at D when it starts the second segment at C. This is done by limiting the maximum velocity of the UAV when it reaches the goal C in the first segment.

limited so it can stop in time for the corner in the next segment. The maximum distance at the goal of the segment is v'_{max}, given the actual expansion distance $dist$ in Equation 4.5.

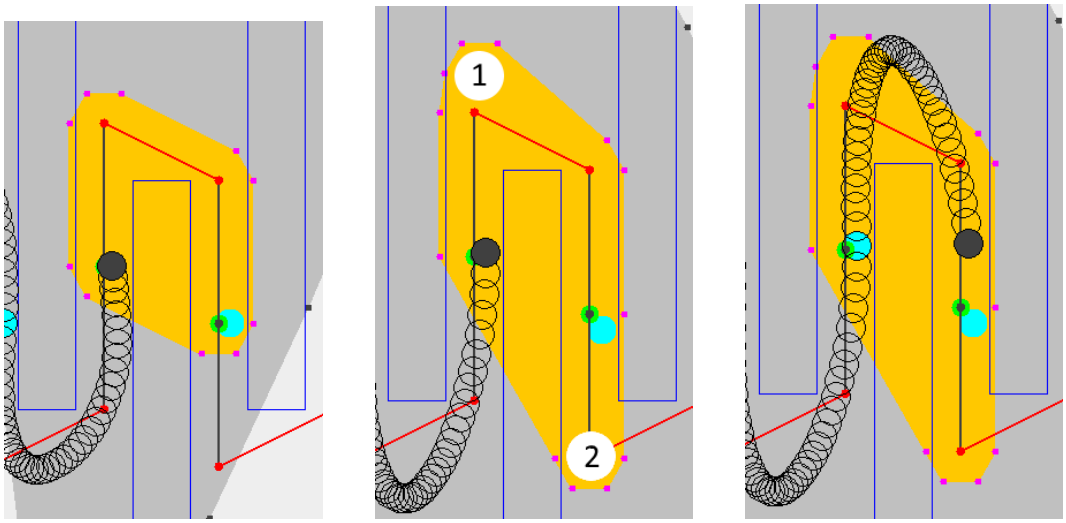
$$v'_{max} = \sqrt{2 * dist * a_{max}} \tag{4.5}$$

4.3.2 Initial Safe Region

The genetic algorithm is relatively simple and makes no attempt to construct the safe region such that the UAV can always stay inside it. This is problematic when the UAV has a long maximum acceleration distance. This problem presents itself in two ways.
The first issue arises at the start of the segment. The maximum goal velocity from section 4.3.1 limits the UAV's velocity, but it does not determine which way the vector is pointed. The maximum goal velocity only ensures safety when the velocity vector is pointed along the path. If the velocity vector is not pointed entirely along the path, the UAV may not be able to stay within the safe region generated by the genetic algorithm. This can be seen in Figure 4.5. The solution to this issue is including the "stop point" in the initial set of points used as the safe region. This stop point is the position where the UAV would come to a halt if it starts decelerates as at the start of the segment. This depends on the initial velocity vector for that segment.

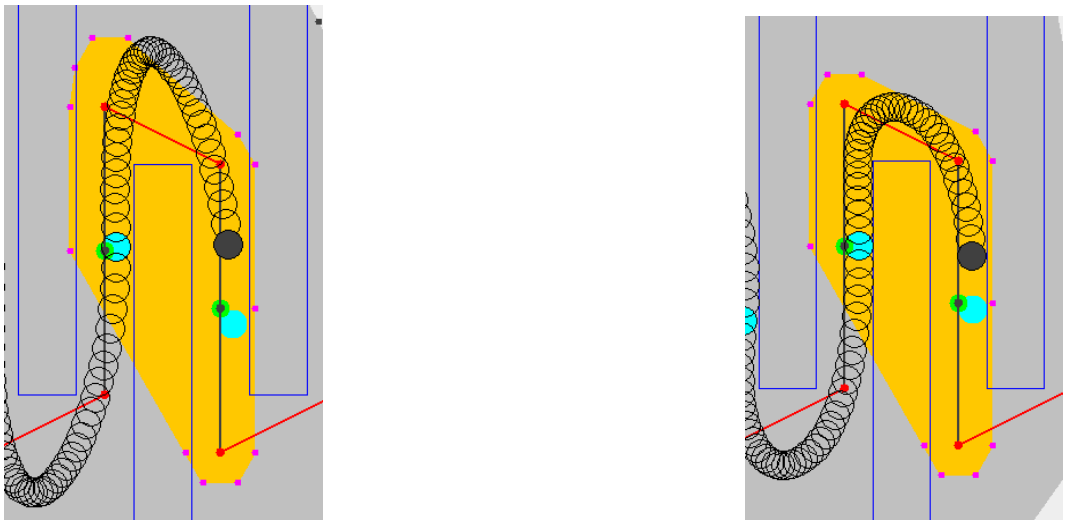
(a) (b)

Figure 4.5: An example of how transition between segments can fail. In 4.5a, shows the UAV right before the segment transition which happens at the blue circle. 4.5b shows the orange initial safe region and dark grey expanded safe region of the next segment, after the transition. The next segment fails to solve because the UAV cannot come to a stop within the safe region. The position of the UAV shows where the UAV came to a stop in the previous segment (the safe region of which is the grey region in 4.5a).



(a) (b) (c)

Figure 4.6: 4.6a shows the initial safe region without the extra stop points in orange. 4.6b shows the initial safe region with the extra stop points. The position marked with ”1” is the stop point for the initial velocity, while the position marked with ”2” is the stop point for the trajectory after the goal has been reached. 4.6c shows how the stop point ensured that the UAV could stop. The UAV leaves the initial safe region because it already has started to turn, however the stop point clearly provided enough space.



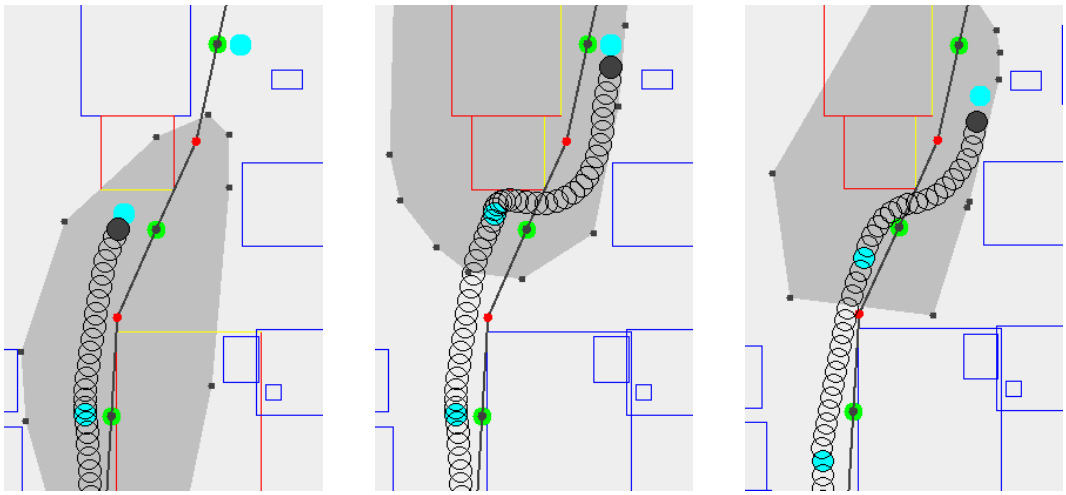
(a) (b)

Figure 4.7: 4.7a and 4.7 show the trajectory respectively without and with the maximum goal velocity enabled. In 4.7a, the UAV overshoots the corner which results in a slower trajectory.

The second issue comes into play at the end of a segment. The sub-trajectory for each segment does not end when the goal is reached. It must be calculated for all available time steps. The safe region may be very restrictive just after the UAV has reached its goal. The result is that the UAV has to slow down *before* it reaches its goal to ensure the rest of the trajectory stays within the safe region. This can also by counteracted by including a stop point in the safe region. This time, the velocity vector of the UAV is not known in advance (since the segment has not been solved yet). The velocity vector is assumed to point along the path and the magnitude is either the maximum goal velocity or maximum velocity, depending on whether the maximum goal velocity is defined.
Figure 4.6 shows the result when both of these stop points are included. In the example, the maximum goal velocity has been ignored to exaggerate the effect. However, the extra stop points should always be used together with the maximum goal velocity. The stop points allow the UAV to make the transition between segments at higher velocities, but that also means they can overshoot in turns. This is demonstrated in Figure 4.7.

4.4 Overlapping segment transitions

The initial safe region extensions (section 4.3.2), more tolerant goal condition (section 4.2.1) and the maximum goal velocity (section 4.3.1) often improve the quality of the trajectory. However, they do not deal with all cases in which a bad segment transition happens, as demonstrated in Figure 4.8. The trajectory up to the goal of



(a) (b) (c)

Figure 4.8: 4.8a shows the optimal approach to the goal of the previous segment. However, as seen in 4.8b, this is a very bad start state for the next segment. By starting the next segment 5 time steps earlier in 4.8c, this bad approach for can be partially mitigated.

the first segment is optimal (Figure 4.8a), but provides a really bad start for the next segment (Figure 4.8b).
This can be counteracted by starting the next segment earlier than usual. Instead of starting at the time step where the goal in the previous segment is reached, the next segment start several time steps earlier. This ensures the the previous segment still attempts to reach its goal as fast as possible, but also allows the next segment to correct for suboptimal transitions earlier. This can be seen in Figure 4.8c.

4.5 Graphical Visualization Tool

Using MILP to solve the trajectory planning problem makes my algorithm flexible. Different constraints can be added and removed without having to change complex algorithm. The solver takes those changes into account and still finds a solution. This declarative approach makes it much easier to experiment with different variants of the problem, however it does also have downsides. One of those downsides is that it becomes harder to understand why the solution to the problem is what it is. Especially since MILP solvers don't provide a lot of useful insight during execution. This problem is made worse by the fact that the trajectory planning problem is an optimization problem. We're not just interested in having any solution, but instead we want a good or possibly even the best solution.
These factors make the MILP solvers a "black box". This is especially problematic when they fail to find a solution. Most solvers will inform you which constraint

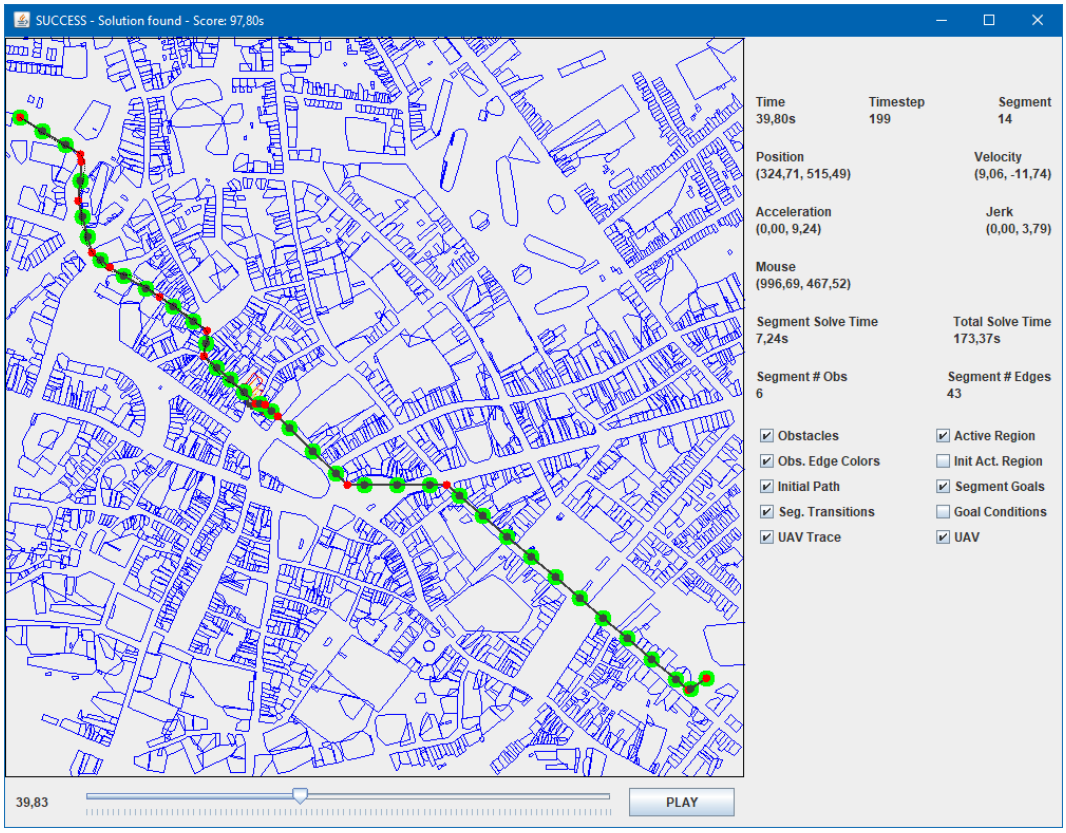


Figure 4.9: An overview of the visualization tool that shows the results of the algorithm.

caused the failure, but that constraint is not necessarily the one that is incorrect. Another constraint may have made the problem impossible to solve, but the solver will only fail by the time it leads to a contradiction.
Another problem is figuring out if all constraints are actually modeled properly. A badly modeled constraint may have no effect at all, or a different effect than intended. Gaining a deep understanding of what is happening is an issue as more and more constraints are added and the interactions between them increase.

For this reason, I spent a significant amount of time building a visualization tool which displays not only the solution, but also the constraints of the MILP problem and other debugging information. This proved to be a critical part of the development cycle of the algorithm. Figure 4.9 shows this tool. At first glance, the tool may look familiar. It has been used to create every example shown in this thesis.

4.5.1 Interface Elements

The graphical interface is divided into three parts.

World view The first and most prominent part is the visualization on the left. This shows a view of the world with a variety of information overlaid on top of it. The view can be translated by holding down the left mouse button and dragging around. Scrolling zooms in and out on the view.

Timeline The second element is the timeline on the bottom. By dragging the slider, the user can change the time step being visualized. There is also a play/pause button which can be used to animate the visualization. The animation runs in realtime according to the progress of time of the trajectory. The animations help spot more subtle changes in acceleration which are not immediately clear from the still view.

Data Panel The last element is the panel on the right. This panel contains more detailed information about the current time step being visualized. This information includes:

- The current time in the trajectory
- The current time step number
- The current segment number
- The position, velocity, acceleration and jerk (derivative of acceleration) at the current time step
- The world coordinates at the position of the mouse
- The solve time for the current segment
- The total execution time of the entire algorithm
- The amount of obstacles and edges being modeled in the current segment

On top of that information, there are also visibility toggles for the various elements of the visualization.

4.5.2 World View Elements

Obstacles The obstacles are the most common element in the visualization. Obstacles are visualized as blue polygons. See Figure 4.10.

Initial path The initial path if the result of the Theta* algorithm. It is shown as black lines. The turn events on this path are red. When multiple nodes on the path belong to the same turn event, the lines connecting them are also red. See Figure 4.10.

Figure 4.10: The obstacles are blue, initial path is black with turn events being red. The green circles are the segment goals.

Scenario goals The scenario goals are visualized as the green circles on the initial path. These goals mark both the end of a segment and the start of the next segment. See Figure 4.10.

Safe region The safe region for the current segment, the result from the genetic algorithm, is shows as a dark grey polygon. The UAV must stay within this polygon at all times. See Figure 4.11.

Obstacle Edge Colors The obstacles within the safe region are modeled in the MILP problem. The color of the edges of these obstacles display whether or not the constraint is active for that edge at the current time step. If the color is yellow, the UAV is on the safe side of the obstacle. Otherwise, the color is red. See Figure 4.11.

Segment Transitions The segment transitions do not always happen right at the goal positions of the segments. The light blue circle shows the position of the UAV at the time of the actual segment transition. See Figure 4.11.

Initial Safe Region Before the genetic algorithm is executed, an initial safe region is constructed which ensures the UAV can reach its goal in the current segment. This initial safe region is shown in orange. The (expanded) safe region from the genetic algorithm must completely contain this initial region. See Figure 4.12.

Goal Conditions The UAV does not have reach its goal exactly. The green square around the segment goal shows the tolerance region where the UAV is close enough

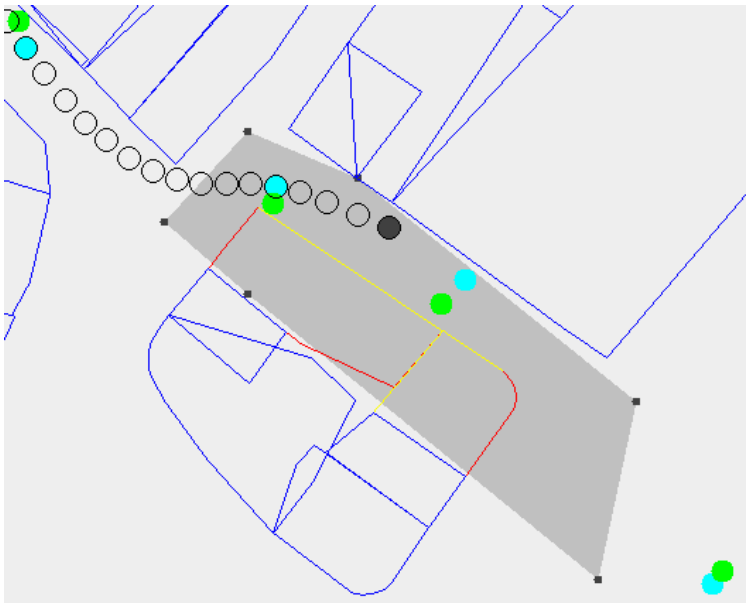


Figure 4.11: The safe region is dark grey and the segment transitions are light blue. The edges of the obstacles inside the safe region are yellow or red to signify whether or not the UAV is on the safe size of that edge.

to its goal. The blue line shows the finish line which the UAV must cross as well. See Figure 4.12.

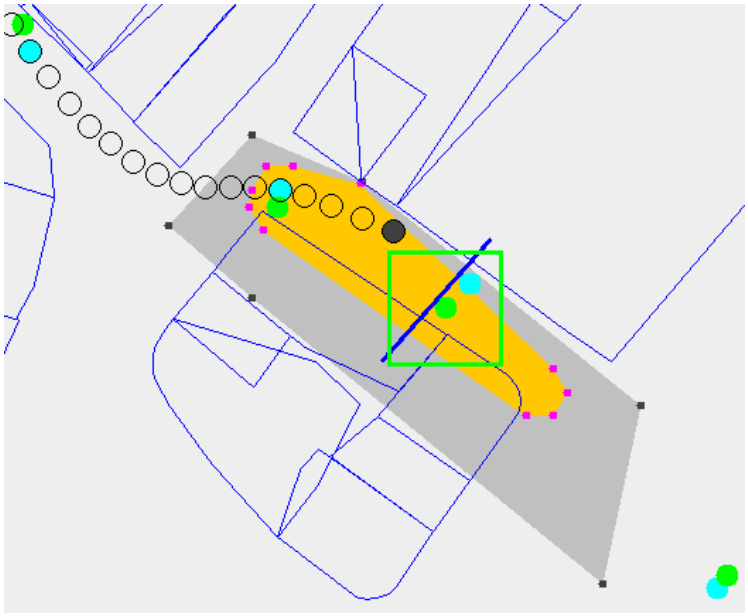


Figure 4.12: The initial safe region is shown in orange. To reach its goal, the UAV must be inside the green square and have crossed the blue finish line.

5

Experiments and Results

5.1 Scenarios

Several different scenarios are used to test the algorithm. Each scenario takes place in a world with a certain distribution of obstacles, has a start and goal position, and a UAV with certain characteristics.

There are three categories of scenarios, which will be discussed in section 5.1.1 to 5.1.3. All scenarios are tested in a general performance test (section 5.2) to get an overview of the performance of the algorithm with the default parameters. This test determines whether or not the algorithm scales to large and complex environments. In the other tests, only one scenario of each category has been used. Section 5.3 tests the performance of the algorithm as the characteristics of the UAV change. Section 5.4 looks at the stability of the algorithm. These tests should provide further insight on the limitations of the algorithm.

Sections 5.5 to 5.9 look at the effects of different parameters on both the performance of the algorithm and the quality of the trajectory. Sensible default parameters have been chosen. However, changing these parameters provides deeper insights in the characteristics of the algorithm.

All tests were executed on an Intel Core i5-4690k running at 4.4GHz with 16GB of 1600MHz DDR3 memory. The reported times are averages of 5 runs, unless stated otherwise. The machine runs on Windows 10 using version 12.6 of IBM CPLEX. Table 5.1 show the default parameters as used in all tests.

grid size	$2m$	turn tolerance	2
approach multiplier	2	population size	10
# generations	25	max. nudge distance	$5m$
min. # vertices	4	max. # vertices	12
P(add vertex)	0.1	P(remove vertex)	0.1
max nudge attempts	15	T_{max}	$5s$
time step size	$0.2s$	position tolerance	3
CPLEX max solve time	$120s$	CPLEX max delta	1
time limit multiplier	1.5	max segment time	$3s$
linear approx vertices	12		

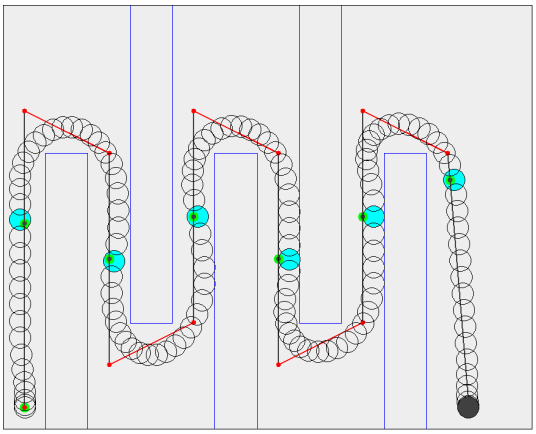
Figure 5.1: The parameters used for testing

5.1.1 Synthetic Scenarios

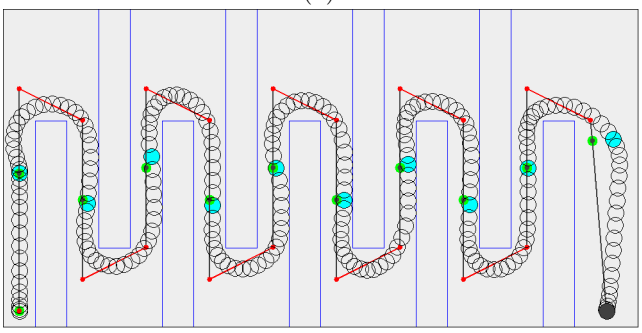
The synthetic scenarios have small, handmade worlds. They have few obstacles, but the obstacles are laid out in a way that makes them challenging to solve. There are two "Up/Down" scenarios, small (Figure 5.2a) and large (Figure 5.2b), in which the UAV has to move in a zig-zag pattern to reach its goal. The difference between those scenarios is the amount of obstacles, which also changes the amount of zig-zags required. These scenarios are built to see how the algorithm handles sharp turns. There is also a spiral scenario in which the UAV must go in an outwards spiral (Figure 5.2c.
The large Up/Down scenario represents this category in the tests. Table 5.1 shows the properties of the UAV.

v_{max} (ms^{-1})	a_{max} (ms^{-2})	radius (m)
3	4	0.5

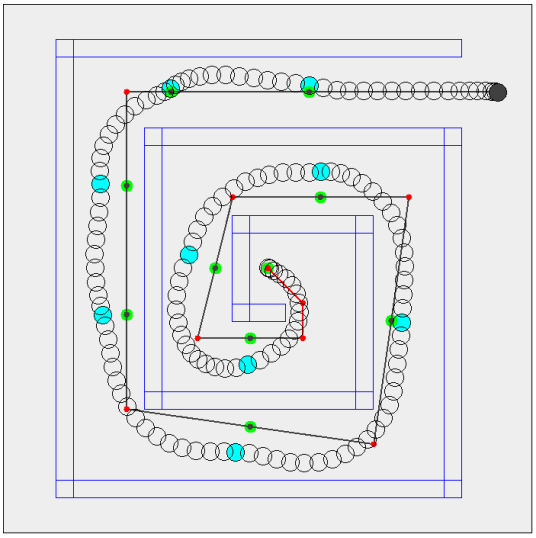
Table 5.1: The UAV properties for the synthetic scenarios



(a)



(b)



(c)

Figure 5.2: The synthetic scenarios

5.1.2 San Francisco Scenarios

The San Francisco scenarios contain a world which is based on a map of San Francisco. There are 2 small San Francisco scenarios (Figure 5.4a and 5.4b), which both share the same 1km by 1km area of San Francisco but have different start and goal locations. There is also a larger San Francisco scenario which takes places on a 3km by 3km world (Figure 5.4c).
All obstacles in the San Francisco dataset are grid-aligned rectangles, laid out in typical city blocks. Because of this, density of obstacles is predictable. These scenarios showcase that the algorithm can scale to realistic scenarios with much more obstacles than is typically possible with a MILP approach.
The first small San Francisco scenario is used to represent this category in the tests. Table 5.2 shows the properties of the UAV and Figure 5.5 shows a zoomed-in view of the first small scenario.

v_{max} (ms^{-1})	a_{max} (ms^{-2})	radius (m)
10	15	2.5

Table 5.2: The UAV properties for the San Francisco scenarios

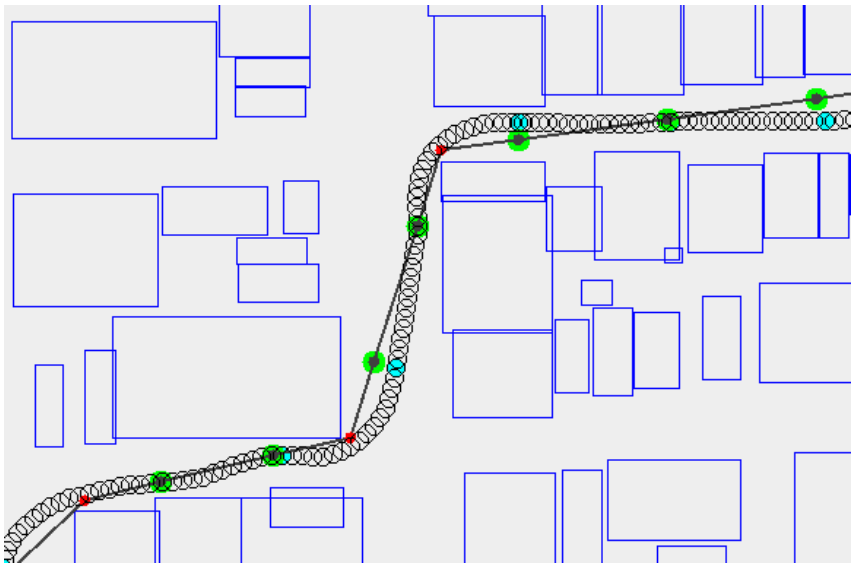
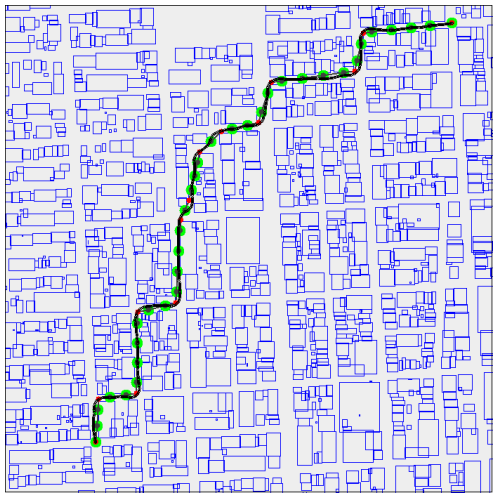
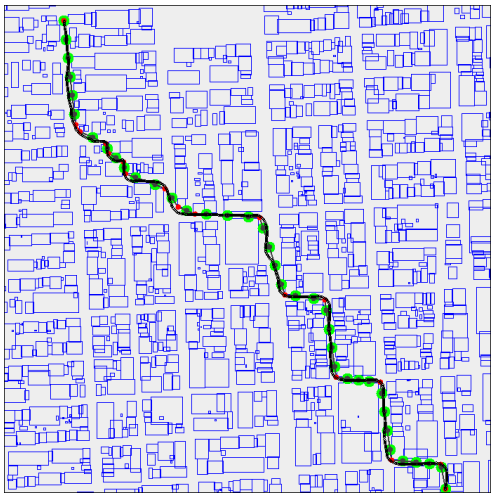


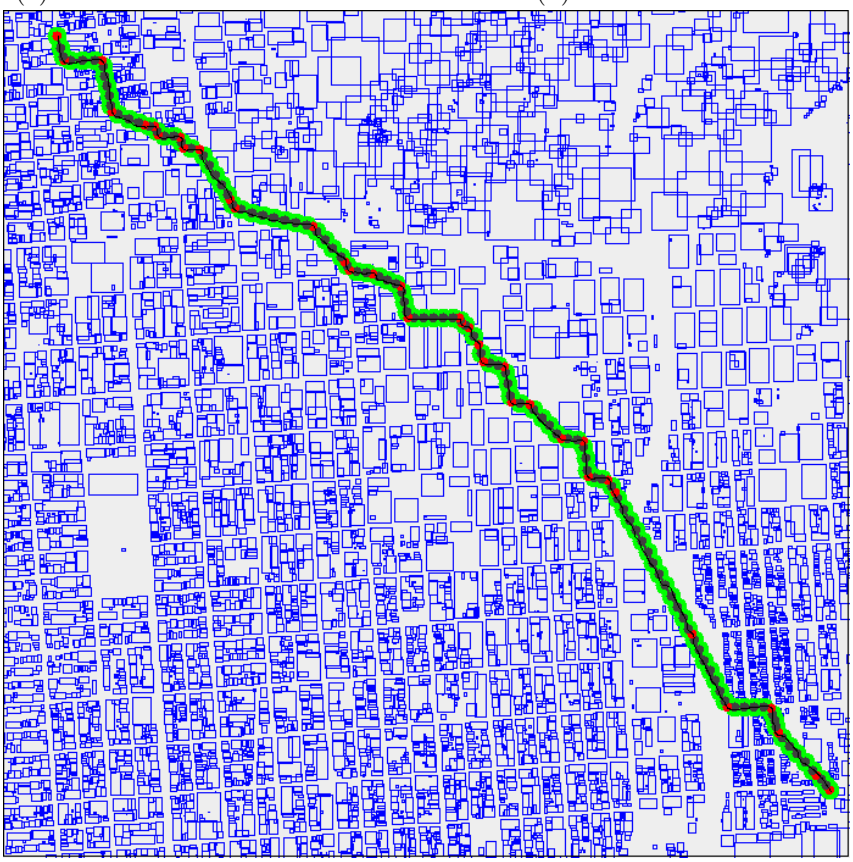
Figure 5.3: A zoomed-in view of the first small the San Francisco scenario



(a) San Francisco Small 1



(b) San Francisco Small 2



(c) San Francisco Large

Figure 5.4: The San Francisco scenarios

5.1.3 Leuven Scenario

The Leuven scenarios contain a world based on a map of the city of Leuven. This is an old city with a very irregular layout. The dataset, provided by the local government[1], also contains full polygons instead of the grid-aligned rectangles of the San Francisco dataset. While most buildings in the city are low enough so a UAV could fly over, it presents a very difficult test case for the path planning algorithm. The density of obstacles varies greatly and is much higher than in the San Francisco dataset across the board.
The first small Leuven scenario is used to represent this category in the tests. Table 5.3 shows the properties of the UAV and Figure **??** shows a zoomed-in view of the first small scenario.

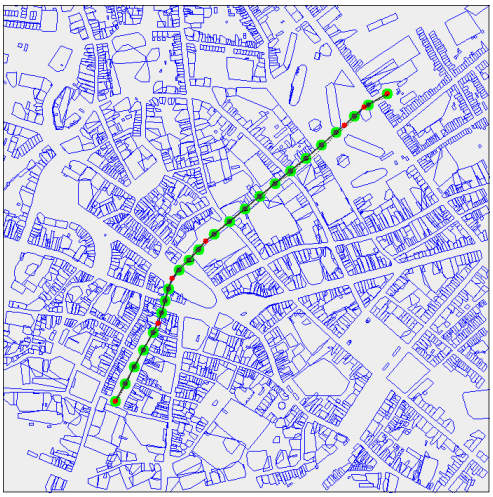
v_{max} (ms^{-1})	a_{max} (ms^{-2})	radius (m)
10	15	1

Table 5.3: The UAV properties for the Leuven scenarios

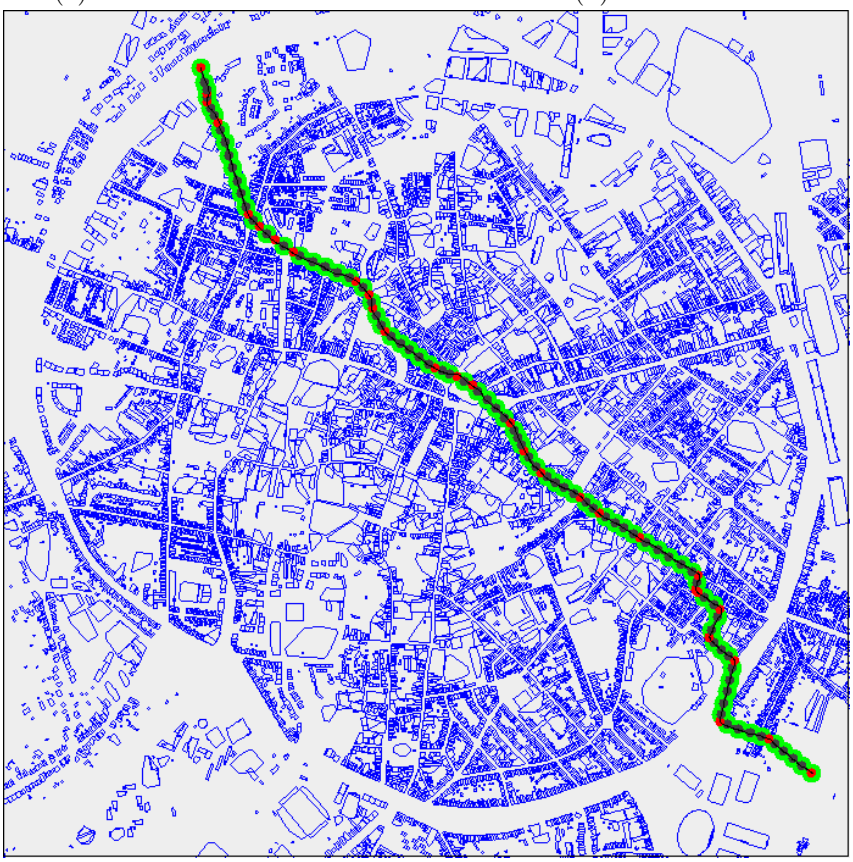
Figure 5.5: A zoomed-in view of the first small the Leuven scenario

[1] `https://overheid.vlaanderen.be/producten-diensten/basiskaart-vlaanderen-grb`

(a) Leuven Small 1



(b) Leuven Small 2



(c) Leuven Large

Figure 5.6: The Leuven scenarios

Scenario name	# obs.	# edges	world size	path (m)	# segments
Up/Down Small	5	5	25m x 20m	88	7
Up/Down Large	9	9	40m x 20m	146	11
Spiral	11	11	30m x 30m	96	10
SF Small 1	1235	4940	1km x 1km	1392	34
SF Small 2	1235	4940	1km x 1km	1490	38
SF Large	6580	26320	3km x 3km	4325	107
Leuven Small 1	3079	19941	1km x 1km	1312	34
Leuven Small 2	3079	19941	1km x 1km	864	22
Leuven Large	18876	111998	3km x 3km	3041	78

Table 5.4: Some information about the scenarios tested.

Scenario name	Theta* (s)	GA (s)	MILP (s)	total (s)	score (s)
Up/Down Small	0.00	0.33	10.48	10.97	27.24
Up/Down Large	0.00	0.65	17.95	18.87	44.76
Spiral	0.01	1.06	7.17	8.46	28.72
SF Small 1	1.37	7.68	32.81	42.43	106.20
SF Small 2	1.82	7.98	36.81	47.32	114.36
SF Large	15.88	15.41	75.44	108.28	325.10
Leuven Small 1	1.51	23.49	135.86	161.85	97.44
Leuven Small 2	0.53	14.00	62.03	76.99	65.52
Leuven Large	14.65	67.55	460.46	544.73	227.27

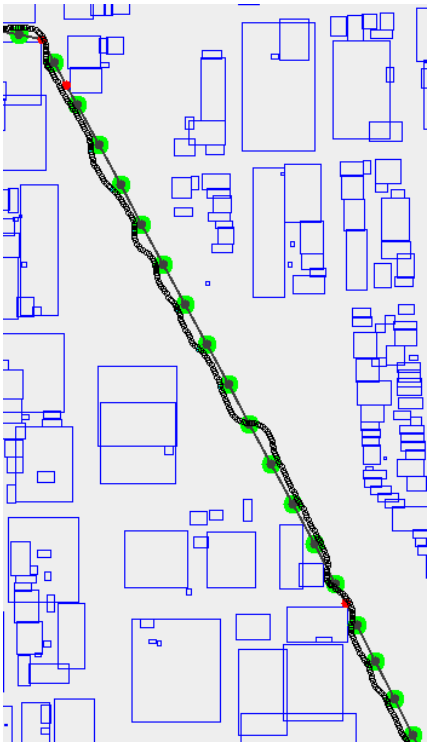
Table 5.5: A breakdown of the execution time for each scenario, as well as the score of the trajectory.

5.2 General Performance

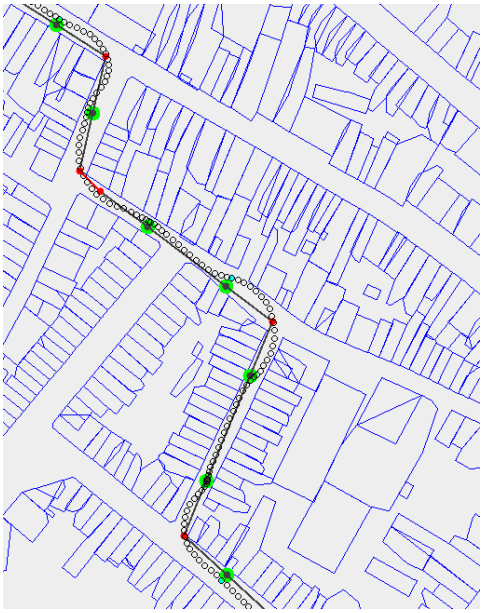
In this test, the general performance of the different parts of the algorithm are tested. Every scenario is tested with the default parameters. Table 5.4 shows some detailed information about the scenarios, including the length of the Theta* path and the amount of segments problem is divided into. Table 5.5 shows the execution parts for the most computationally expensive parts of the algorithm (the Theta* path, genetic algorithm and MILP solver), as well as the total time required. It also shows the score of the resulting trajectories. This score is in seconds and is the amount of time that the UAV needs to reach the goal position when following the trajectory.

5.2.1 Interpretation

With the default settings, the algorithm is capable of solving all scenarios within a reasonable amount of time. In every case, solving the MILP problem takes the majority of the time.
The amount of segments differs between these scenarios. Table 5.6 expresses the results relative to the amount of segments, as well as the path length per segment.



(a) The large San Francisco scenario has a region with very sparse obstacles.



(b) The large Leuven scenario passes through several regions with very dense obstacles

Figure 5.7

Scenario name	path (m)	Theta* (s)	GA (s)	MILP (s)	total (s)
Up/Down Small	12.57	0.00	0.05	1.50	1.57
Up/Down Large	13.27	0.00	0.06	1.63	1.72
Spiral	9.60	0.00	0.11	0.72	0.85
SF Small 1	40.94	0.04	0.23	0.97	1.25
SF Small 2	39.21	0.05	0.21	0.97	1.25
SF Large	40.42	0.15	0.14	0.71	1.01
Leuven Small 1	38.59	0.04	0.69	4.00	4.76
Leuven Small 2	39.27	0.02	0.64	2.82	3.50
Leuven Large	38.99	0.19	0.87	5.9	6.98

Table 5.6: A breakdown of the execution time per segment

A first observation is that the path length per segment is very similar for all San Francisco and Leuven segment. The UAV has the same maximum velocity and acceleration in those scenarios. Even though the density and layout of the obstacles is very different, the amount of segments scales linearly with the path length as long as the UAV properties remain the same.

Another observation is that the relative time needed to calculate the Theta* path increases as the path length increases. This is not surprising since Theta*, like A*, has an exponential worst-case complexity. This means that the scalability of Theta* puts an upper limit on the scalability of the entire algorithm.

The next observation is the genetic algorithm (GA) execution time and MILP solve times are very similar within categories. For the synthetic category, the Up/Down scenarios are virtually identical in both measures. The spiral scenario is hard to compare since it is very different.
For the San Francisco scenarios, the two small scenarios have very similar GA and MILP execution times, but the larger scenario comes in lower. The two small Leuven scenarios are also similar in GA execution time, although the first scenario has a higher MILP solve time. The large Leuven scenario has higher execution times for both the GA and MILP solver.

These variations can be explained by differences in densities between the scenarios. In the large San Francisco scenario, the trajectory crosses a region with very sparse buildings (Figure 5.7a). The sections in the regions are easier to solve than in the smaller scenarios. The opposite happens in the large Leuven scenario, which passes through several regions with very dense obstacles (Figure 6.1). These account for the significant increase in execution time for both the genetic algorithm as the MILP solver.

5.2.2 Comparison to pure MILP

Comparing these results to the "pure" MILP model without preprocessing is difficult. Only the small Up/Down scenario could be solved. For all other scenarios, no solutions could be found even after hours of computation. When given 900 seconds, the average trajectory score after 5 runs is $27.60s$. My algorithm solves the same problem in roughly $11s$, with a trajectory score of $27.24s$. The trajectory score for the pure MILP solution is worse than what my algorithm found, even though it is capable of finding the optimal trajectory. Finding a better solution would take even more time than the 900 seconds the solver was given.

5.3 Agility of the UAV

The properties of the segments strongly rely on the agility of the UAV. The size of the segments is determined by the maximum acceleration distance of the UAV.
This experiment tests the relation between the maximum velocity and the maximum acceleration of the UAV. This test was executed on the standard set of scenarios: the large Up/Down scenario, the small San Francisco scenario and the small Leuven scenario. For each scenario I tested nine configurations of the vehicle: Every combination between three different maximum velocities and three different maximum accelerations. Table 5.7 shows the different UAV properties for the Up/Down scenario, Table 5.8 shows the same properties for the San Francisco and Leuven Scenarios. Table 5.9, 5.10 and 5.11 shows the results for respectively the Up/Down, San Francisco and Leuven scenarios. Figure 5.8, 5.9 and 5.10 show the same data, but fix either the acceleration or velocity and vary the other property.

	Low	Med	High
Velocity (ms^{-1})	2	4	8
Acceleration (ms^{-2})	1	3	6

Table 5.7: The different maximum velocity and acceleration values for the vehicle for the Up/Down scenario.

	Low	Med	High
Velocity (ms^{-1})	5	15	30
Acceleration (ms^{-2})	3	10	20

Table 5.8: The different maximum velocity and acceleration values for the vehicle for the San Francisco and Leuven scenarios.

solve time (s)	Low vel	Med vel	High vel
Low acc	26.08	91.98	100.28
Med acc	9.93	16.93	15.17
High acc	7.23	6.39	4.85

Table 5.9: Up/Down

solve time (s)	Low vel	Med vel	High vel
Low acc	57.3	-	-
Med acc	22.78	31.85	483.17
High acc	19.53	13.75	39.64

Table 5.10: San Francisco

solve time (s)	Low vel	Med vel	High vel
Low acc	127.62	-	-
Med acc	64.16	118.86	-
High acc	60.83	53.8	-

Table 5.11: Leuven

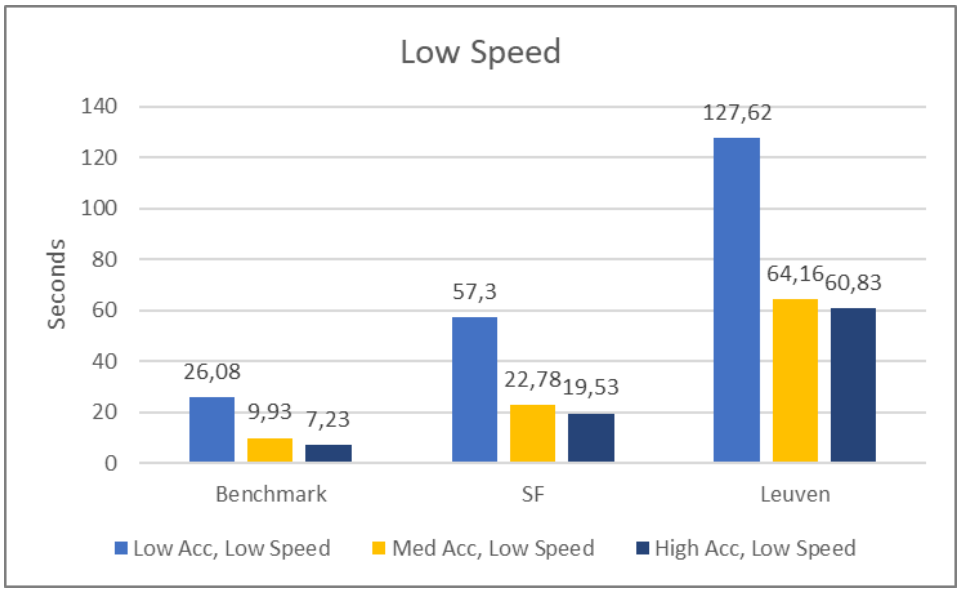
5.3.1 Interpretation

Several of the combinations failed to complete because the segments could not be solved within 120 seconds each. The general trend is that a higher maximum acceleration and a lower maximum velocity decrease the solve time. This is as expected, as both of those make the maximum acceleration distance smaller. A larger maximum acceleration distance leads to larger segments with more obstacles in them, which have a negative effect on the performance.

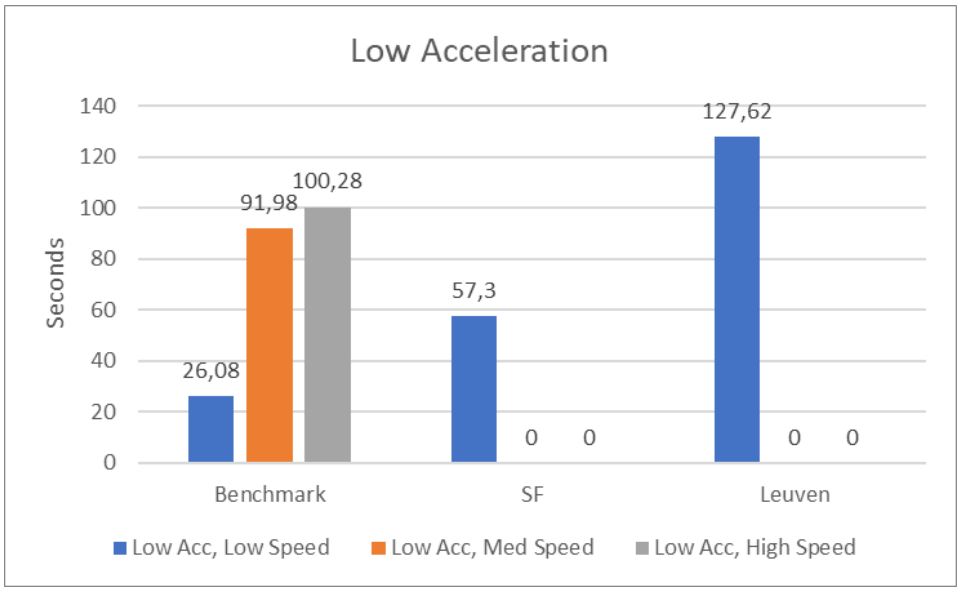
When the acceleration is high, varying the velocity has an unexpected effect. When going from a low to medium maximum velocity, the solve time actually decreases for all scenarios. I do not have an explanation for why this happens.

Another slightly unexpected result is that the combination of a high maximum acceleration and high maximum velocity fails for the Leuven scenario. This is not a particularly difficult combination for the other scenarios, so the failure is unexpected.

The default UAV properties seem to be right on the limit for the Leuven scenario. If the UAV is a bit less agile, the algorithm fails to find a solution. This can also be flipped on its head: the Leuven scenario is on the edge of what can be solved. The density of obstacles in the Leuven scenario is at the limit of what can be handled using the default parameters.

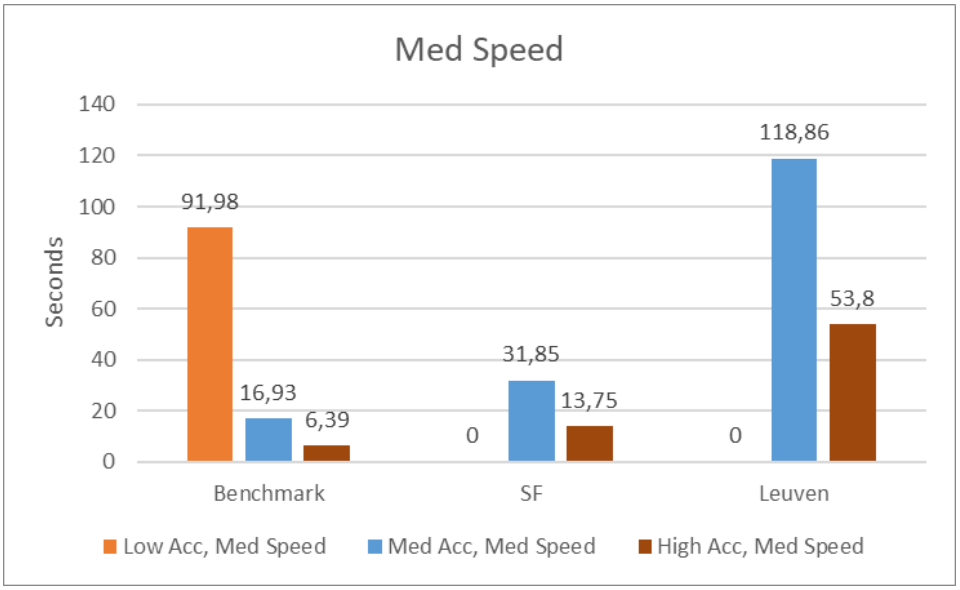


(a) The effects of a varying maximum acceleration and a low maximum velocity.

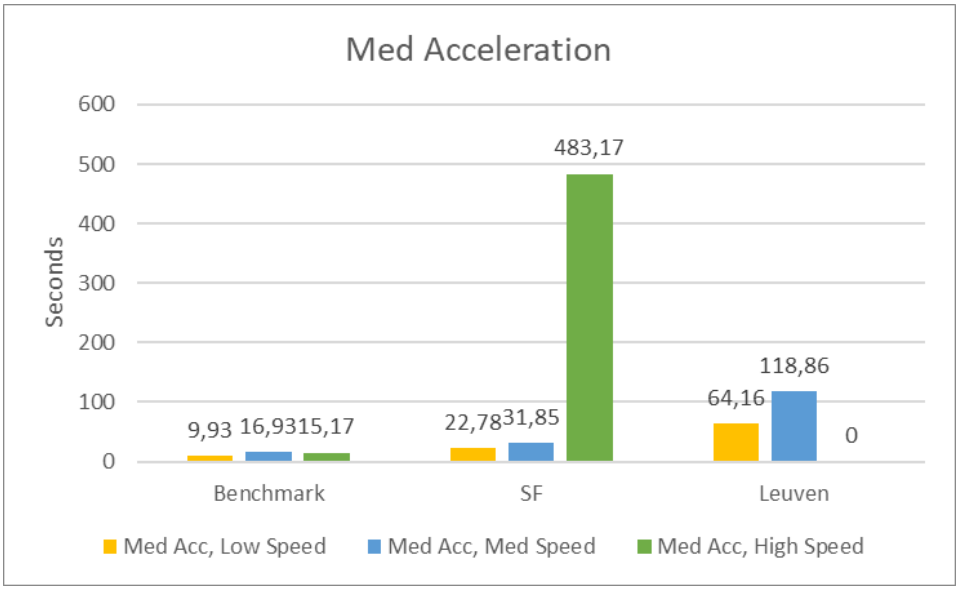


(b) The effects of a varying maximum velocity and a low maximum acceleration.

Figure 5.8

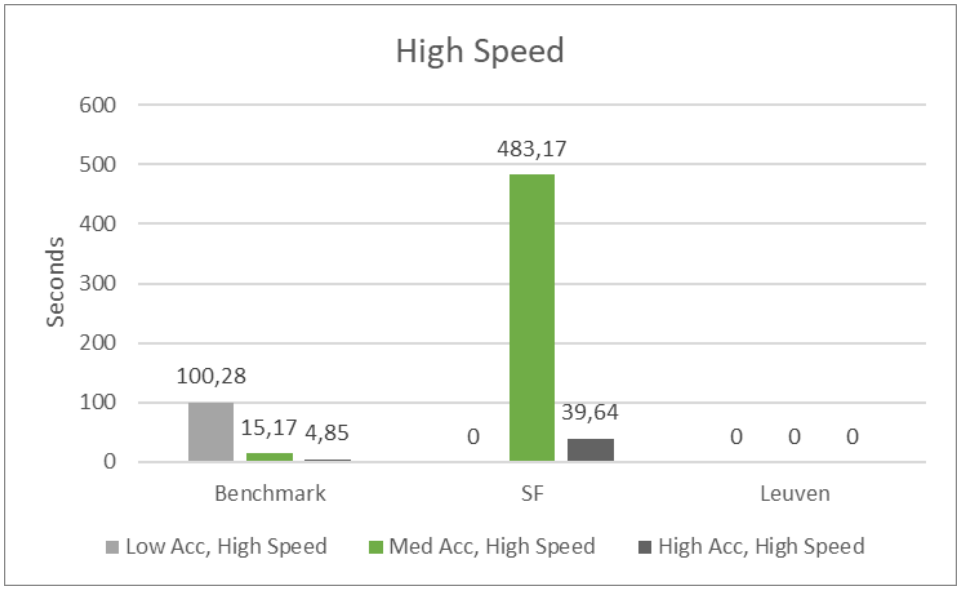


(a) The effects of a varying maximum acceleration and a medium maximum velocity.

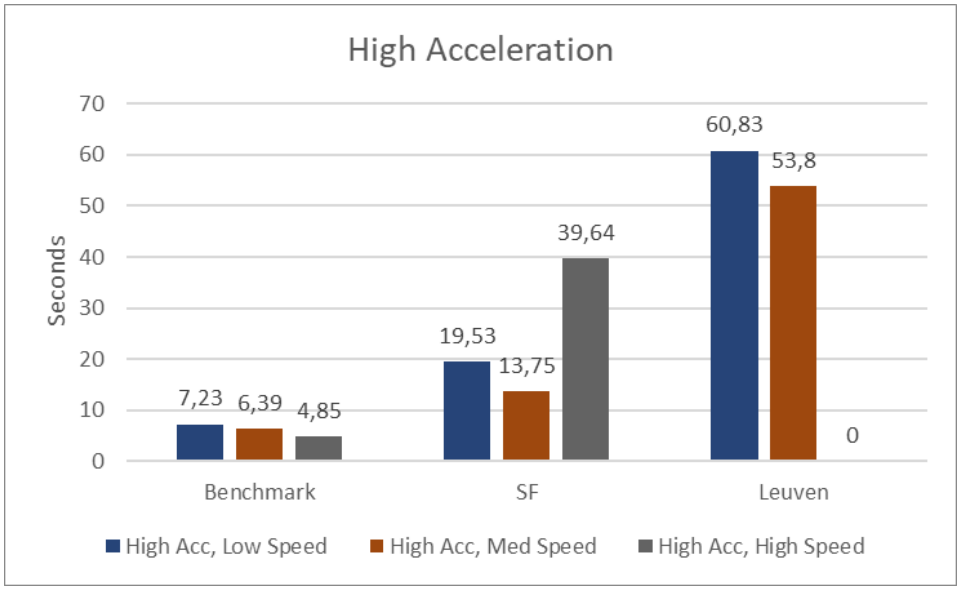


(b) The effects of a varying maximum velocity and a medium maximum acceleration.

Figure 5.9



(a) The effects of a varying maximum acceleration and a high maximum velocity.



(b) The effects of a varying maximum velocity and a high maximum acceleration.

Figure 5.10

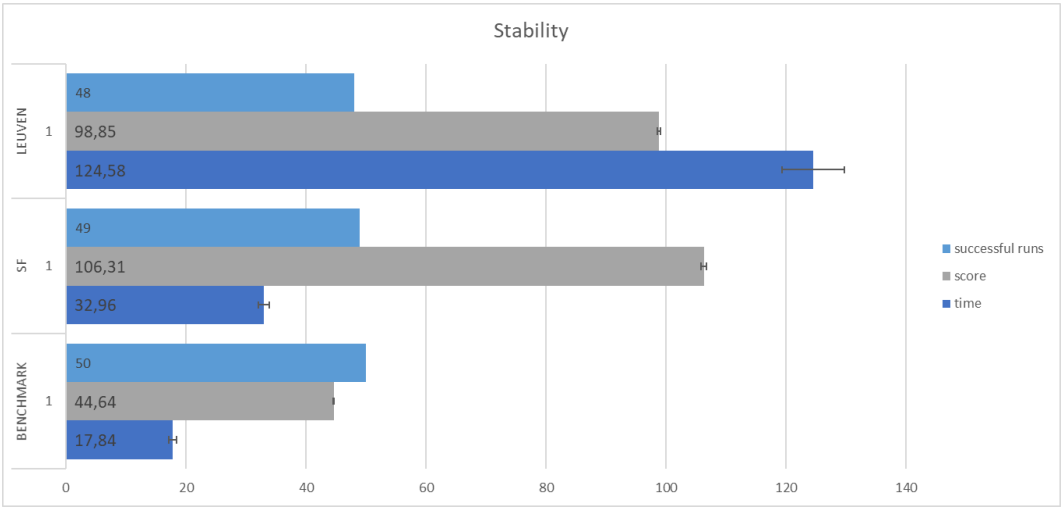


Figure 5.11: stability data

5.4 Stability

Stability is an important property of an algorithm. When the same problem is solved several times, the algorithm should not occasionally fail to solve the problem or require a wildly different amount of time to solve that problem.
This experiment aims to measure the stability of the algorithm. Each of the testing scenarios is executed 50 times, instead of 5 like in the other tests. Figure 5.11 shows the results of this test. The error bars show sample standard deviation.

5.4.1 Interpretation

Even though the each sub-problem should ensure that the next segment can be solved, occasionally this was not the case as demonstrated in Figure 5.12. It seems like a collision is inevitable, but that is actually not the case as demonstrated in 5.13. I believe this may be a bug in how the algorithm transitions between segments. I could not properly figure out what cases this bug in time.

However, the algorithm does find a good trajectory in nearly all cases. When it does succeed, the standard deviation of trajectory scores are 0.6%, 1.6% and 0.9% of the mean scores for the Up/Down, San Francisco and Leuven scenarios respectively. The standard deviations on execution time are higher, at respectively 13%, 10% and 15% of the mean values.

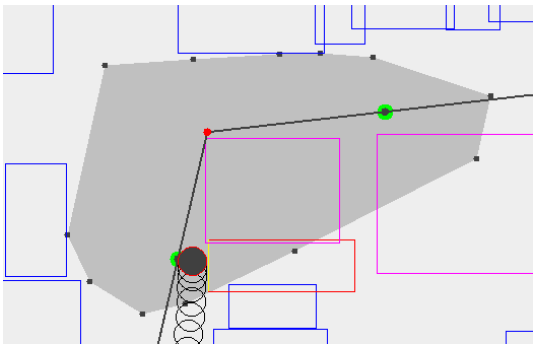
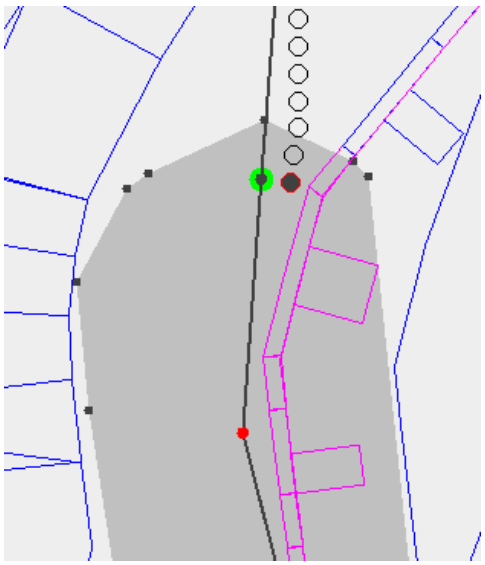
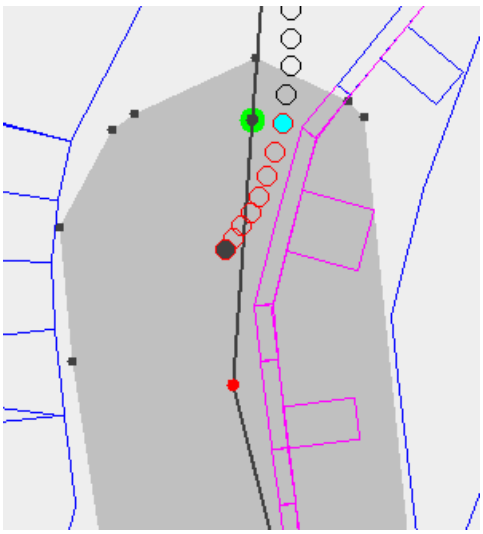


Figure 5.12: A case where the transition between segments fails



(a) This segment starts and fails with what seems like an inevitable collision...



(b) ... However, this shows the trajectory in the previous segment after the goal in that segment has been reached in red. Clearly a collision is not inevitable.

Figure 5.13

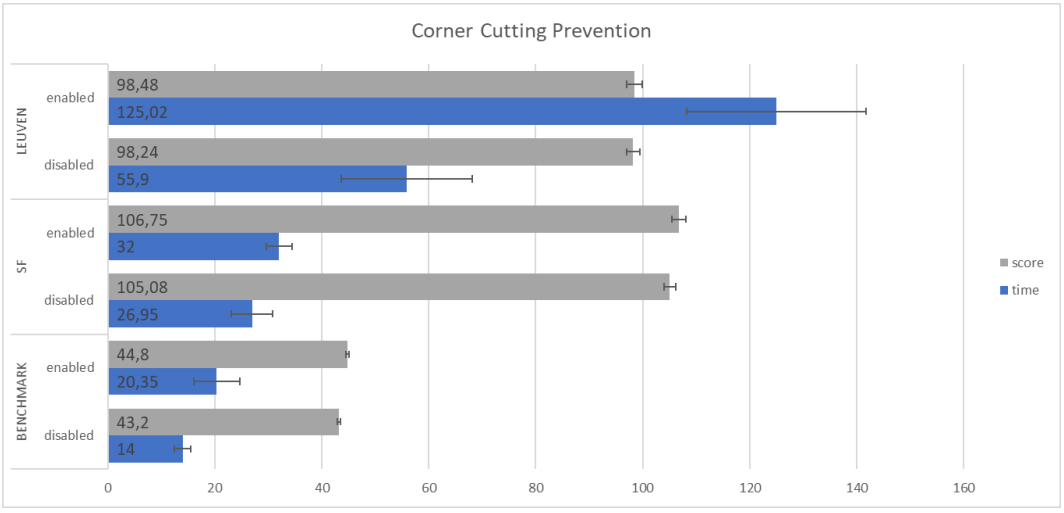


Figure 5.14: The error bars show the 95% confidence interval.

5.5 Cornercutting

A trajectory that allows corner cutting cannot be considered safe. However, additional constraints are required to prevent this from happening. This experiment attempts to measure the impact of the corner cutting prevention. The scenarios are solved with the corner cutting mitigation from section 4.2 both enabled and disabled. Figure 5.14 shows the results.

5.5.1 Interpretation

As expected, enabling the corner cutting prevention has a negative impact on performance. This effect is limited for the Up/Down and San Francisco scenarios. For the Leuven Scenario, the solve time more than doubles. This is likely due to the higher obstacle density and complexity.

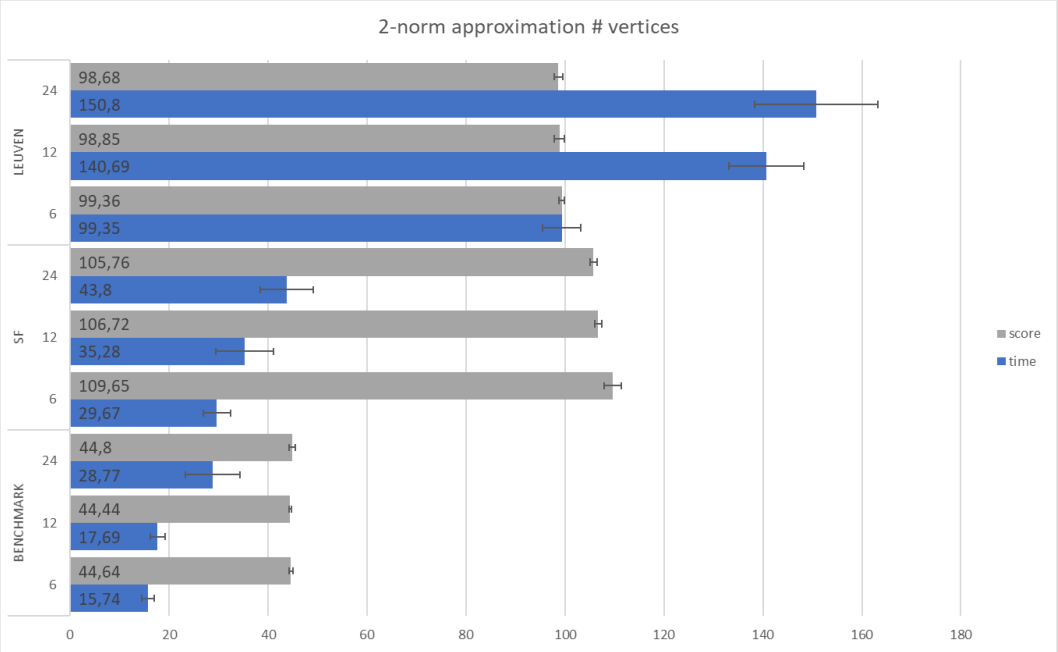


Figure 5.15: linear approx data

5.6 Linear approximation

The velocity and acceleration of the UAV are limited to some finite value. Because both of those quantities are vectors, that maximum can only be approximated with linear constraints. More constraints are needed to model this more accurately which can allow for faster solutions. However, more constraints also have a performance cost. This experiment analyses the trade-off that needs to be made. The amount of vertices used for the linear approximation is tested with values of 6, 12 and 24.

5.6.1 Interpretation

In all cases, increasing the amount of vertices used to approximate the 2-norm also increases the solve time. The effect on trajectory score seems to be minimal for the Up/Down and Leuven trajectory. However, for the San Francisco trajectory there is a noticeable improvement with the better approximation.

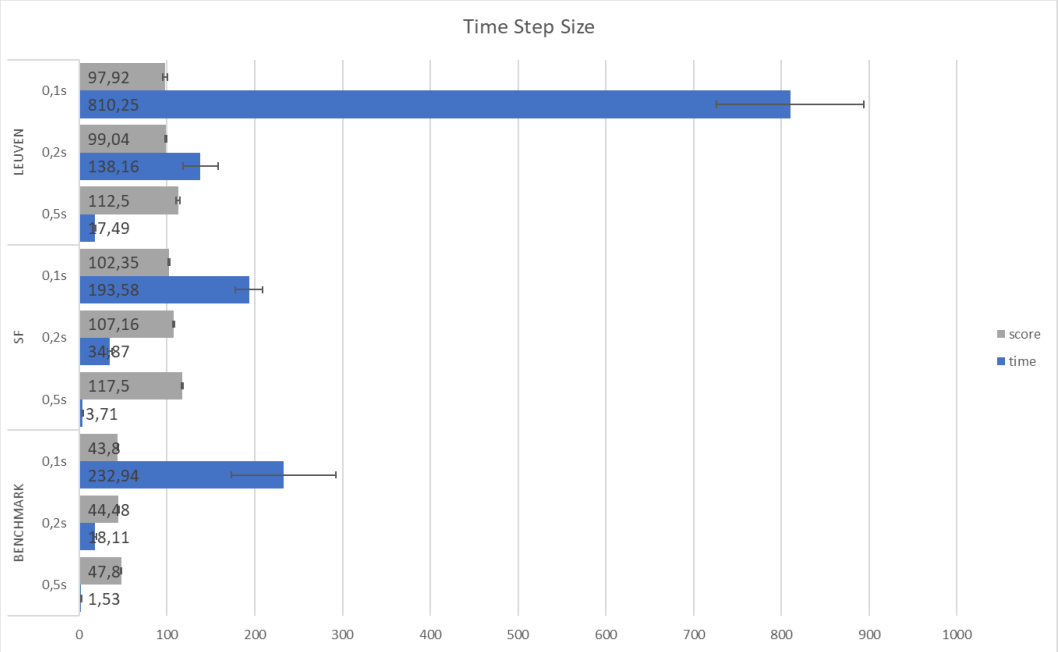


Figure 5.16: time step data

5.7 Time step size

The time step size determines how many time steps are used in each MILP problem. The discretized time steps are samples at regular intervals of the continuous trajectory that the UAV would actually travel in the real world. As a result, the trajectory defined by those time steps is a piece-wise linear approximation of this smooth, real world trajectory.
As the size of each step goes to zero, the approximation becomes more accurate. This also means that the trajectory should become faster, since the UAV can be controlled more precisely through time. This allows for more aggressive maneuvers. However, adding more time steps increases the amount of integer variables and constraints. This comes at a performance cost.
In this experiment, three different time step sizes are tested. The $0.2s$ default value, as well as $0.1s$ and $0.5s$ are tested.

5.7.1 Interpretation

The time step size has a dramatic impact on performance. Changing the time step size from $0.2s$ to $0.1s$ or $0.5s$ changes the solve time by a factor of 5 to 10 or even more for the Up/Down scenario. This experiment really shows the exponential complexity of MILP. Making the time steps smaller makes the algorithm much slower.

As expected, decreasing the time step size also improves the score of the trajectory. The largest gain is the step from $0.5s$ to $0.2s$, although there still is some improvement when going to $0.1s$.

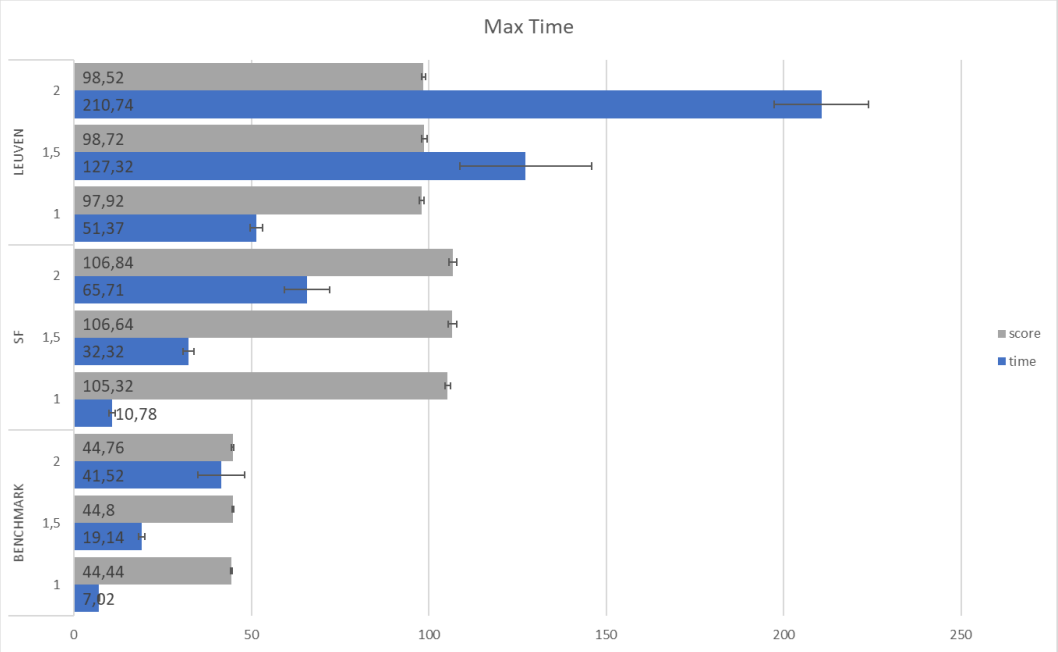


Figure 5.17: maxtime data

5.8 Max Time

For each sub-problem, the amount of time steps to model needs to be determined in advance. The algorithm calculates an estimated upper bound for the time (and thus the amount of time steps). In the ideal case, this upper bound is equal to the time needed for the optimal trajectory. However, if the upper bound is too low, no solution can be found.
This experiment looks at the importance of a low upper bound on the time needed. By default, the estimated upper bound is multiplier by 1.5 to ensure enough time steps are available. A multiplier of 1 is also tested, along with a multiplier of 2.

5.8.1 Interpretation

The time needed to solve the scenarios is heavily influenced by maximum time given. For these scenarios, the default multiplier of 1.5 seems unnecessary and could be lowered to 1 without issues. Increasing the multiplier to 2 nearly doubles the solve time across the board.

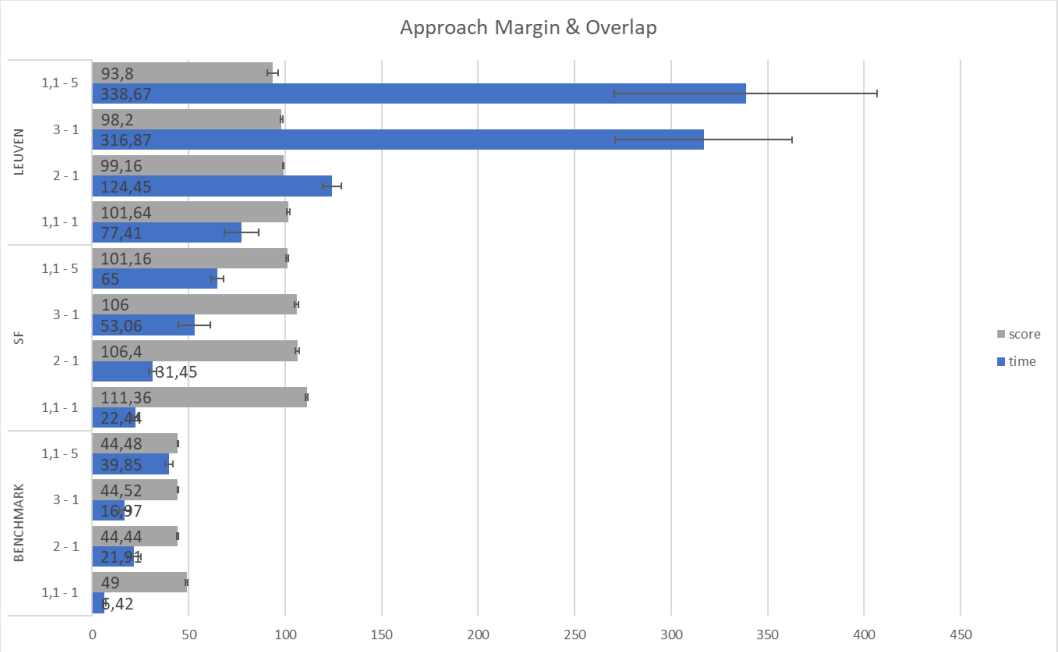


Figure 5.18: approach data

5.9 Approach Margin

The approach margin determines have far turn events are expanded outwards to create the boundaries between segments. A value of 1 is the minimum safe value and ensures that in the worst case, the UAV can just barely come to a stop before a turn. Higher values give the UAV more space to maneuver so more efficient approaches are possible. This experiment looks at 3 different values for the approach margin: 1.1, 2 and 3.
Additionally, it also looks at the effect of overlapping the segments. Usually, the segments only overlap by 1 time step: the time step in which the UAV reaches the goal in the previous segment, which is the starting state for the next segment as well. Since the overlap also helps to improve the efficiency of the approach of the UAV, a scenario was tested in which 5 time steps overlap between segments with an approach margin of 1.1.

5.9.1 Interpretation

As expected, a higher approach margin leads to a higher solve time. Larger segments tend to include more obstacles and need more time steps, negatively impacting performance. For the Up/Down scenario there is a small decrease in solve time with a margin of 3 compared to 2, which I cannot explain. The San Francisco scenario sees a steady increase in solve time as the approach margin is increased. This is the same for the Leuven scenario when going from a margin of 1.1 to 2, but between 2 and 3 there is a large leap in solve time. This is likely caused by the higher density

of obstacles.
The low approach margin of 1.1 with an overlap of 5 time steps results in the highest solve time for all 3 scenarios. This is unexpected, because a higher approach margin pushes the start of segments back more than the extra 4 time steps. TODO: extra!

In the Up/Down and San Francisco scenarios, going from an approach margin of 1.1 to 2 results in a clear improvement in the trajectory score. The Leuven scenario also shows an improvement, but the difference is smaller. Increasing the margin even more to 3 has no significant effect on any scenario. Overlapping the segments results in a large improvement in trajectory scores for the San Francisco and Leuven scenario, scoring the best out of all combinations. Overlapping segments seems to be more computationally expensive, but also more effective than having a high approach margin.

6

Discussion

The main focus points during this these were the performance and stability of the algorithm. Most decision were made with either performance or stability in mind, and often both. TODO: preview with refs

6.1 Performance

On the performance side, it is clear that the new algorithm with preprocessing is much faster than solve the pure MILP problem without preprocessing. Comparing the new algorithm to the pure approach is very difficult since the challenging scenarios for the new algorithm simply cannot be solved with the pure approach.

When it comes to scalability, there are few noteworthy observations to make.

6.1.1 Path Length Scalability

The first is that the time needed to solve each MILP subproblem does not depend on the length of the trajectory or the size of the world. Accounting for variations due to obstacle density, the average MILP solve time for scenarios using the same dataset (San Francisco or Leuven) are very similar. Since the amount of segments scales linearly with the path length, the MILP part of the algorithm also scales linearly with the length of the initial path. This is in stark contrast with the exponential worst-case performance of the pure MILP approach
However, this exponential complexity has not been eliminated. It has been shifted to the initial path planning algorithm, Theta*. This algorithm still has exponential worst-case complexity with respect to the length of the path. While Theta* does limit the scalability regarding the size of the world, it is a much easier problem to solve. It is part of the A* family of path planning algorithms which have been the subject of a large body of research.

The algorithm separates the "routing" aspect from the trajectory planning aspect of the problem. The exact properties of the initial path are not very important. What matters is that it determines where and when to turn. By the time the MILP solver runs, the navigation aspect of the problem has already been solved. The MILP solver only needs to find a viable trajectory. The two aspects of the problem are solved separately, making both of them easier to tackle.

I believe that identifying that these two aspects can be solved separately is the key insight that made the performance improvements possible. Solving these aspects separately means that the optimal trajectory is unlikely to be found. However, at this point that seems like a necessary sacrifice for long term trajectory planning through complex environments. I am not aware of any algorithm that scales as well as my algorithm does, and also finds the optimal trajectory.

6.1.2 Obstacle Density Scalability

The second observation is that the density of the obstacles plays a large role in the scalability of the algorithm. The San Francisco and Leuven scenarios are very similar, except for their obstacle density. The Leuven scenario has a significantly higher density of obstacles and is also much harder to solve. Without preprocessing, the scalability of the MILP problem is limited by the total amount of obstacles. Because each MILP subproblem in my algorithm is roughly the same size, the total amount of obstacles is no longer the limiting factor. The density of the obstacles is the limiting factor in my algorithm.

The Leuven scenarios can still be solved in an acceptable amount of time, but I do not believe that this will be the case if the density is increased even more. Luckily, the Leuven dataset is more detailed than it needs to be. Each building in the city is represented separately, even when multiple buildings are connected. It should be possible to reduce the obstacle density substantially with a minimal amount of effort. Figure 6.1 shows a dense region in the Leuven dataset where there is a lot of room for improvement.

Figure 6.1: One of the denser regions in the Leuven dataset

Given that the Leuven dataset is so unoptimized for this purpose and the algorithm still completes on average on the order of seconds per segment, I believe that the scalability with regards to the obstacle density is acceptable.

6.1.3 UAV Agility

The last observation is the importance of the agility of the UAV. The algorithm was developed with high-end consumer to professional grade multirotor UAVs in mind. These are very agile vehicles capable of impressive feats of acrobatics when properly piloted. This agility is one of the assumptions this algorithm is based on. The UAV must be able to hover and accelerate quickly.

The results from the UAV agility experiment in section 5.3 show that these assumptions are indeed a critical part of the performance of the algorithm. The algorithm fails when faced with UAVs with an (unreasonably) low agility.

This reliance on agility is one of the factors that made the dramatic improvement in performance possible, but it also limits the applicability of the algorithm. However, the goal of this thesis was not to developed a general algorithm. The algorithm performs well for reasonable estimates of the agility of a multirotor UAV. On top of that, the agility of any UAV can be increased by limiting its maximum velocity[1].

6.2 Stability

For the algorithm to be useful, it must be stable. The first aspect of stability is whether or not it can find a solution. If the algorithm is capable of finding a solution, it should find that solution every time. It should also be able to solve problems with a similar difficulty as well. The second aspect is that the solution for the same problem should always be similar. There should be no large differences in the trajectory scores when the same problem is solved multiple times, nor should there be a large difference between very similar problems. This is also applies to the execution time. The execution time for similar problems should also be similar without large variations.

My algorithm can find a solution most of the time. Due to what I believe to be a bug, it occasionally fails to find a solution. I was not able to fully understand why the bug occurs, but I believe it can probably be fixed.

The stability of the trajectory scores are excellent. All trajectories found are scored within a few percentage points of each other. When it comes to the execution time, there is more variation. However, with a standard deviation is around 10-15 % of the mean execution time, I believe that the stability is still acceptable for offline trajectory planning.

6.3 Important parameters

During development of the algorithm I settled on sensible default parameters which balance both the performance of the algorithm and the quality of the resulting trajectory. The experiments which tested different values for those parameters

[1] Limiting the maximum velocity of UAVs with a low acceleration when navigating through a city seems like a wise decision anyway. Such a UAV would not be able to react to unexpected obstacles quickly, so having it fly at a high velocity through the city seems like a dangerous proposition.

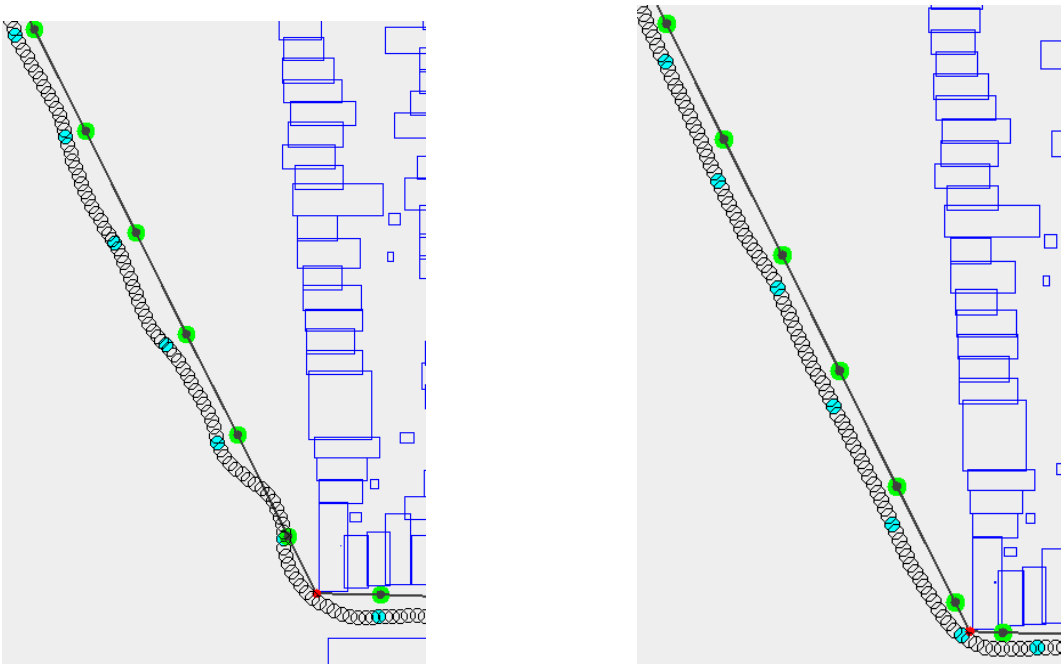
provide a deeper insight in the effects of those parameters. Many of those insights point to possible improvements to the current algorithm.

6.3.1 Time Step Size and Maximum Time

From the time step size experiment (section 5.7) and maximum time experiment (section 5.8) it becomes clear that the amount of time steps in each segment has a very large effect on the performance of the algorithm. The time step size should be chosen so the quality of the trajectory is high enough to be usable, without being more detailed than necessary. How large the time step size should be will depend on the specific use case. The maximum time should always be as low as possible, while still ensuring that the goal can be reached in that time.
By combining the effects of those parameters, the algorithm could be made faster without suffering a quality penalty. For each segment, the algorithm could first solve a MILP problem with a conservative maximum time and a large time step size. The solution to this problem shows how much time the UAV needed to reach its goal in that segment. This value can be used as a much tighter maximum time in a MILP problem with a smaller time step size. While I did not have time to properly implement this, a quick-and-dirty test showed promising results. Using this method and solving the segments first with a time step size of $0.5s$, the San Francisco scenario MILP solve time dropped from 32s to 10s. For the Leuven scenario the MILP solve time dropped from 136s down to 38s. In both cases, the execution time was cut by more than two thirds without impacting the quality of the trajectory.

6.3.2 Approach Margin

The approach margin experiment (section 5.9) shows that having some approach margin is beneficial, but that the gains are often relatively small. Overlapping the segments by just 5 time steps already results in a significantly better trajectory than a very large approach margin. Because of this, I believe that my idea that a larger approach margin leads to more efficient approach is not accurate. I suspect that the slight improvements in trajectory score when increasing the approach margin are caused by simply having fewer segments. Or to put it more accurately: it is caused by having fewer transitions between segments. Overlapping the segments smooths out bad transitions between those segments. A larger approach margin does not improve bad transitions, it only guarantees that the UAV can correct for it in time. These bad transitions are not immediately obvious when the UAV is constantly maneuvering, but they become very clear when the UAV is flying straight. Figure 6.2a shows a case where the UAV is not moving entirely along the path after a turn. This is corrected, but it starts an oscillation along the trajectory that takes many segments to die down. In Figure 6.2b, the slight overlap of the segments prevents that oscillation from starting in the first place. With my current algorithm there is a high performance cost involved with overlapping the segments. However, I do not believe this necessarily has to be the case. This is definitely something to look at in future work.



(a) (b)

Figure 6.2: Without overlapping the segments, oscillations tend to occur in the trajectory as seen in 6.2a. Even a small amount of overlap is an effective countermeasure as seen in 6.2b.

6.4 Genetic Algorithm

The genetic algorithm is the part of the thesis that got the least attention. This is mainly because the genetic algorithm itself is not an essential part of the trajectory planning algorithm. It is used to grow the convex safe region, and the current genetic algorithm does that well enough. Due to time constraints I have omitted detailed parameter tuning of this genetic algorithm.

Either way, the genetic algorithm is very crude. A genetic algorithm was chosen to grow the convex safe region because it is a quick way to get a reasonably good result. Since it never was the most pressing issue in the project, it never got replaced by a more refined algorithm (genetic or otherwise).

6.5 Future work

There are several extensions and improvements which may improve the performance of the algorithm or the quality of the trajectory. These extensions may also improve the general utility of the algorithm, making it useful for more different use cases

- As discussed in section 6.3, solving each segment multiple times with different parameters can result in performance improvements. However, when combined

with the observation that the segment transition can be improved with overlapping segments, I believe that a more drastic change may be in order. Solving the segments first with a high time step size may also be used in combination with a high amount of overlap with other segments. This rough trajectory does not only contain information on the time needed to solve a segment, but may also be used to subdivide the segments even further.

- Another step is extending this approach to 3D. The extra degree of freedom will likely come at a significant performance penalty, so this was not attempted during the thesis. One of the likely difficulties with the preprocessing as presented is that it treats all dimensions the same. This is fine for the horizontal dimensions, but due to gravity, movements the vertical dimension have different characteristics. The maximum acceleration of the UAV can no longer be assumed to be the same in all directions.
  A possible mitigation to the increasing complexity of obstacles may be using a "2.5D" representation. A 2.5D obstacle is a 2D obstacle which also has a height value. This would only need one additional integer variable per obstacle to model. In a city scenario, this may be an acceptable approximation.

- I would like to try using Mixed-Integer Quadratic programming. This eliminates the need to approximate the 2-norm. This may improve performance.

- The algorithm presented in this thesis is aimed at offline planning. To test the algorithm on actual physical UAVs, an online planner is necessary. This online planner will need to be able to use the trajectory generated by this algorithm and ensure that the UAV keeps following it even if there are perturbations.

6.6 Conclusion

The goal for this thesis was relatively open-ended: to build a scalable trajectory offline planning algorithm for multirotor UAVs. This scalability entailed that the algorithm had to work in very large, complex and realistic scenarios. The experiments show that the algorithm is capable of planning trajectories through such environments, and that it can do so consistently. There are still some minor issues with the algorithm, but the results show that clearly this approach is viable.
Even though the goals of this thesis have certainly been reached, the algorithm presented in this thesis is just a first step. The algorithm shows that MILP trajectory planning can scale to environments which are several orders of magnitude more complex than what has been considered before. However, the algorithm still has weaknesses.
One of the main challenges of this thesis was the lack of prior research into this subject. While there is a lot of material available on how to build a MILP trajectory planning model, the performance characteristics of those models are mostly unexplored. This made it difficult to identify which part of the problem to focus on next. With the

insights from this thesis, I believe that it is possible to improve upon this algorithm and reach even better results.

7

Conclusions

In this thesis, I presented a scalable trajectory planning algorithm using Mixed-Integer Linear Programming (MILP). This algorithm is capable of generating trajectories through environments on the scale of cities in 2D space. These environments can be on the order of square kilometers in size with trajectories spanning several kilometers. Previous approaches with MILP trajectory planning were not scalable enough to generate trajectories through such environments.

This improvement of performance is achieved by using several steps of preprocessing. This preprocessing approach is the main contribution of this thesis to the field. During preprocessing, a Theta* path planning algorithm is used to find a viable route to reach the goal. Based on this path, the trajectory planning problem is divided into many subproblems or segments, each of which solve a small part of the final trajectory. The segments are solved consecutively and their results are stitched together to form the final trajectory.

Dividing the trajectory planning problem reduces the amount of time steps and obstacles that have to be modeled in each subproblem. Minimizing both those properties is critical to ensure the MILP subproblem can be solved quickly. Special care was taken to ensure that the transitions between the segments do not cause the algorithm to fail.

The algorithm was tested with a variety of scenarios. Several of those scenarios are based on maps of actual cities, namely San Francisco and Leuven. The results show that the MILP part of the algorithm scales linearly (instead of exponentially) with the trajectory length and is dependent on the density of the obstacles instead of the total amount of obstacles. The Theta* algorithm still scales exponentially, but is a much easier problem to solve with a large body of research detailing possible improvements.

The transitions between segments present some challenges which have yet to be solved. They cause some inefficient movements which reduce the quality of the trajectory unnecessarily. Another weak point of the algorithm is the limited processing on the obstacles. Some preprocessing to simplify dense arrangements of obstacles would allow the algorithm to operate in even more complex environments.

Detailed analysis of the effects of some of the parameters have shown that there is still a lot of room for improvement for the algorithm. Further development of the algorithm based on the insights gained through this thesis can address some of the shortcomings of the current approach and continue to exploit the strengths.

Bibliography

[1] John Saunders Bellingham. *Coordination and control of uav fleets using mixed-integer linear programming.* PhD thesis, Massachusetts Institute of Technology, 2002.

[2] Atif Chaudhry, Kathy Misovec, and Raffaello D'Andrea. Low observability path planning for an unmanned air vehicle using mixed integer linear programming. In *Decision and Control, 2004. CDC. 43rd IEEE Conference on*, volume 4, pages 3823–3829. IEEE, 2004.

[3] Ian D Cowling, Oleg A Yakimenko, James F Whidborne, and Alastair K Cooke. A prototype of an autonomous controller for a quadrotor uav. In *Control Conference (ECC), 2007 European*, pages 4001–4008. IEEE, 2007.

[4] Kieran Forbes Culligan. *Online trajectory planning for uavs using mixed integer linear programming.* PhD thesis, Massachusetts Institute of Technology, 2006.

[5] Robin Deits and Russ Tedrake. Efficient mixed-integer planning for uavs in cluttered environments. In *Robotics and Automation (ICRA), 2015 IEEE International Conference on*, pages 42–49. IEEE, 2015.

[6] Michele Fliess, Jean Lévine, Philippe Martin, and Pierre Rouchon. Design of trajectory stabilizing feedback for driftless at systems. *Proceedings of the Third ECC, Rome*, pages 1882–1887, 1995.

[7] Melvin E Flores. *Real-time trajectory generation for constrained nonlinear dynamical systems using non-uniform rational B-spline basis functions.* PhD thesis, California Institute of Technology, 2007.

[8] Yongxing Hao, Asad Davari, and Ali Manesh. Differential flatness-based trajectory planning for multiple unmanned aerial vehicles using mixed-integer linear programming. In *American Control Conference, 2005. Proceedings of the 2005*, pages 104–109. IEEE, 2005.

[9] Leslie Lamport. A simple approach to specifying concurrent systems. *Communications of the ACM*, 32(1):32–45, 1989.

[10] Daniel Mellinger and Vijay Kumar. Minimum snap trajectory generation and control for quadrotors. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 2520–2525. IEEE, 2011.

[11] G Mitra, C Lucas, S Moody, and E Hadjiconstantinou. Tools for reformulating logical forms into zero-one mixed integer programs. *European Journal of Operational Research*, 72(2):262–276, 1994.

[12] Tom Schouwenaars, Bart De Moor, Eric Feron, and Jonathan How. Mixed integer programming for multi-vehicle path planning. In *Control Conference (ECC), 2001 European*, pages 2603–2608. IEEE, 2001.

Fiche masterproef

Student: Jorik De Waen

Titel: Scalable Multirotor UAV Trajectory Planning using Mixed-Integer Linear Programming

Nederlandse titel: Schaalbare Traject Planning voor Onbemande Multirotor Luchtvaartuigen met Gemengd Geheeltallig Lineair Programmeren

UDC: 004.8

Keywords: TODO:keywords

Titel van het artikel: Scalable Multirotor UAV Trajectory Planning using Mixed-Integer Linear Programming

Korte inhoud:
TODO

Thesis voorgedragen tot het behalen van de graad van Master of Science in de ingenieurswetenschappen: computerwetenschappen, hoofdoptie Artificiële intelligentie

Promotor: Prof. dr. T. Holvoet

Assessoren: Dr. M. Cruz Torres
Dr. B. Bogaerts

Begeleiders: H. T. Dinh
Dr. M. Cruz Torres