Scalable Multirotor UAV Trajectory Planning using Mixed-Integer Linear Programming

Jorik De Waen

May 29, 2017

Contents

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 large retail distributors like Amazon are investing heavily into delivering order by drones instead of courier. While these developments look promising, there are still many challenges that prevent these systems from being widely deployed.

One such challenge, which is especially important for areal vehicles, is path planning. Even though most modern quadrocopters are capable of flying by themselves, they are unable to generate a flight path that will get them to their destination reliably. Classic graph-based shortest path algorithms like Dijkstra's algorithm its many variants fail to take momentum and other factors into account. Mixed Integer Linear Programming (MILP) is one approach that shows promising results, however it is currently severely limited by computational complexity.

1.1 Motivation

One of the main advantages of using a constraint optimization approach like MILP is that they are extremely extendable by design. A system based on this can be deployed in many different scenarios with different goals and constraints without the need for significant changes to the algorithms that drive it. The solvers that construct the final path are general solvers which take constraints and a target function as input. This input can be generated by end users in the field to match their specific requirements, making the software controlling the drones as flexible as the hardware.

That flexibility is also the main limitation of using constraint optimization. The solvers are general purpose, which make them very slow compared to more direct approaches. They need to be carefully guided solve all but the most basic scenarios in a reasonable amount of time. While there have been some good results on small scales, I could not find any attempts at planning paths on the order of kilometers or more. Practical use cases involving drones often involve several minutes of flight and can cover several kilometers, so a path planner must be able to work at such a scale. This is the main goal of this thesis: To demonstrate how a MILP approach can be scaled to scenarios with a much larger scope, while preserving the advantages that make it interesting.

1.2 Contribution

TODO...

1.3 Structure of the Thesis

Section **??** summarizes the previous work that has been done in the field. The previous work in the field shows a common design to modeling the path planning problem as a MILP problem. This design forms the core of the approach in this thesis as well. Section 2.3 shows the implementation of this common design and explores the critical limitations to this approach.

Section 3 proposes a solution to these limitation. By finding a rough initial path, the planning problem can be split into smaller segments. Solving these segments on their own is significantly easier and can still enforce all the constraints. This approach is much faster than previous techniques, but at the cost of no longer finding the global optimum.

During the development of this algorithm, finding and solving bugs and other unwanted behavior proved to be a significant challenge. A visualization tool was developed to make it easier to see how the algorithm operates. Section 4.5 goes into detail of how nearly every variable in the MILP problem was visualized and how this information can be interpreted.

To demonstrate the flexibility of the approach, section **??** showcases some possible extensions that can be added with relative ease. Some of these have been fully implemented to look at the impact on the solution. This section should demonstrate that the path planner discussed in this thesis is a modular strategy built out of several different algorithms. The specific algorithms discussed are just one way of doing things, and can be easily swapped out for other, more advanced, algorithms.

Section **??** analyzes the performance of the path planner in several different scenarios. It also looks at how the extensions which have been implemented affect the both the performance and quality of the planner. Finally, section **??** summarizes the main observations in this thesis and concludes whether or not the goals have been realized.

1.4 Assumptions

1.5 Literature Review

TODO:PRM and RERT

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 different 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.

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.

1.6 goals

1.7 TODO: explain title

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 TODO:ref to TODO:ref describe how I modeled the trajectory planning problem. Section TODO:ref shows some initial results, and hypothesizes what the underlying problem could be.

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 con-

straints 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 1 and 2. $time_n$ represents the amount of time that has passed at a given time step n. Equation 1 initializes the time at the first time step to be zero. Equation 2 progresses time by Δt at each time step. In total, there are N time steps.

$$time_0 = 0 \tag{1}$$

$$time_{n+1} = time_n + \Delta t, \quad 0 \leq n < N - 1 \tag{2}$$

The position of the UAV at time step n is represented by p_n. Equation 3 initializes the position in the first time step to the starting position $\boldsymbol{p}_{start}$. Equation 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.

$$\boldsymbol{p}_0 = \boldsymbol{p}_{start} \tag{3}$$

$$\boldsymbol{p}_{n+1} = \boldsymbol{p}_n + \Delta t * \boldsymbol{v}_n \quad 0 \leq n < N - 1 \tag{4}$$

The velocity of the UAV is modeled the same way as the position. Equation 5 initialized the velocity in the first time step, and Equation 6 updates it at each time step based on the acceleration $\boldsymbol{a}_i$. Other derivatives can be modeled like this as well.

$$\boldsymbol{v}_0 = \boldsymbol{v}_{start} \tag{5}$$

$$\boldsymbol{v}_{n+1} = \boldsymbol{v}_n + \Delta t * \boldsymbol{a}_n \quad 0 \leq n < N-1 \tag{6}$$

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 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).

$$minimize \quad -\sum_{n=0}^{n<N} done_n \tag{7}$$

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 8.

$$done_{n+1} = done_n \vee cdone_{n+1}, \quad 0 \leq n < N-1 \tag{8}$$

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 9 resolves this issue by forcing $done_0$ to be false.

$$done_0 = 0 \tag{9}$$

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 10 forces $done_{N-1}$ to be true at the last time step. In combination with Equation 8 and 9, this ensures that the UAV must reach its goal eventually for the solution to be valid.

$$done_{N-1} = 1 \tag{10}$$

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 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 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.

$$cdone_n = cdone_{p,n} \wedge \neg done_n \quad 0 \leq n < N \tag{11}$$

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 12, some equations only apply to 2D worlds for simplicity. TODO: add appendix with generalized equations?
Equation 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

$$cdone_{p,n} = |x_n - x_{goal}| < \epsilon_p \wedge |y_n - y_{goal}| < \epsilon_p, \quad 0 \leq n < N \tag{12}$$

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 11 can be extended with a *cdone*$_{v,n}$ requirement as shown in 13. *cdone*$_{v,n}$ is defined by Equation 14, where vx_n and vy_n are the components of the UAV's velocity vector which must be smaller than some ϵ_v.

$$cdone_n = cdone_{p,n} \wedge cdone_{v,n} \wedge \neg done_n \quad 0 \leq n < N \tag{13}$$

$$cdone_{v,n} = |vx_n| < \epsilon_v \wedge |vy_n| < \epsilon_v, \quad 0 \leq n < N \tag{14}$$

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

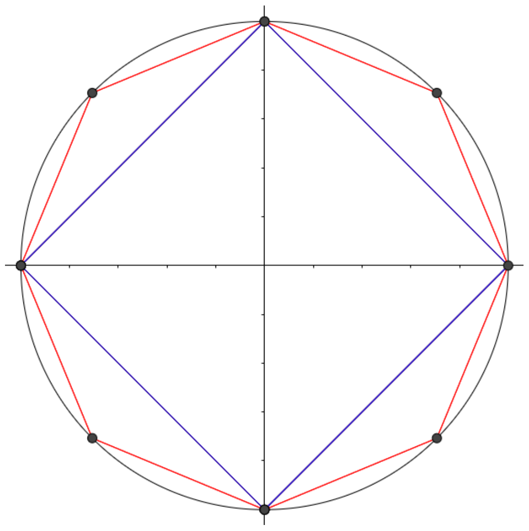


Figure 1: If the velocity is limited to a finite value, the velocity vector must lie within the circle centered on the origin with the radius equal to that value. This is represented by the black circle. This circle cannot be approximated in MILP, but it can be approximated using several linear constraints. 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.

need to be squared. However, the 2-norm can be approximated using multiple linear constraints.
Figure 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 15-17 in counter-clockwise order.

$$\theta = \frac{2\pi}{N_{points}} \tag{15}$$

$$q_{x,i} = v_{max} * cos(\theta i), \quad 0 \leq i < N_{points} \tag{16}$$

$$q_{y,i} = v_{max} * sin(\theta i), \quad 0 \leq i < N_{points} \tag{17}$$

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 18-20.

$$\Delta q_{x,i} = q_{x,i} - q_{x,i-1}, \quad \Delta q_{y,i} = q_{y,i} - q_{y,i-1} \tag{18}$$

$$a_i = \frac{\Delta q_{y,i}}{\Delta q_{x,i}} \quad 0 \leq i < N_{vertex} \tag{19}$$

$$b_i = q_{y,i} - a_i q_{x,i} \quad 0 \leq i < N_{vertex} \tag{20}$$

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 21. Similarly, for the bottom half, the velocity vector must stay *above* those edges as expressed by Equation 22. If N_{vertex} is odd, the left-most edge of the polygon is vertical. Equation 23 handles this special case: the velocity vector must stay on the right of the left-most edge. TODO: add figure to explain better.

$$vy_n \leq a_i vx_n + b_i, \qquad \Delta q_{x,i} < 0, \quad 0 \leq n < N \tag{21}$$

$$vy_n \geq a_i vx_n + b_i, \qquad \Delta q_{x,i} > 0, \quad 0 \leq n < N \tag{22}$$

$$vx_n \geq q_{x,i}, \qquad \Delta q_{x,i} = 0, \quad 0 \leq n < N \tag{23}$$

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.

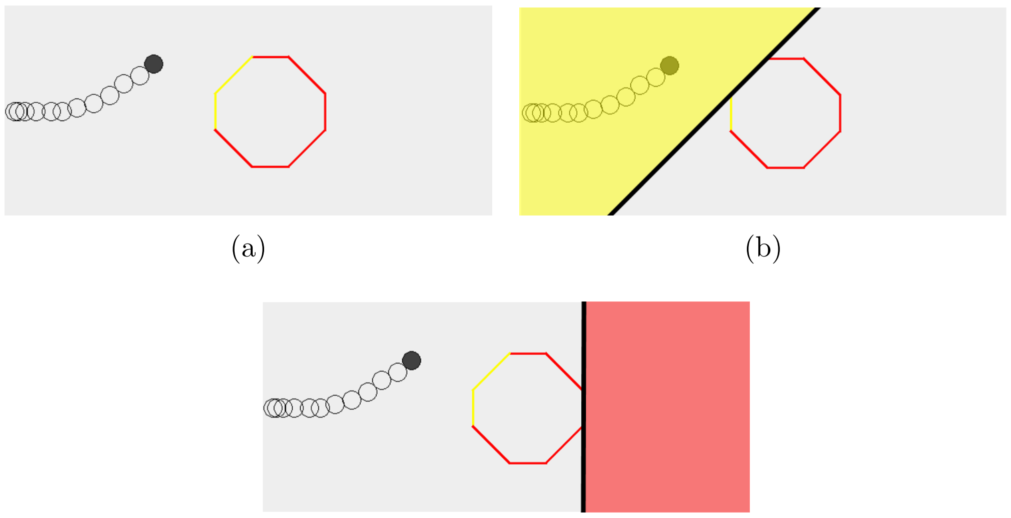

(a) (b) (c)

Figure 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.

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 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 be is no longer convex. This can be seen in Figure 3b.

convex nonconvex

(a) convex (b) nonconvex

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 TODO:REF, the edges of the obstacle are used to construct the constraints. Each edge defines a line, as determined by Equations 18-20. 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 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 24-27 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 18-20. For every time step n:

$$\begin{array}{lllll} y_n + M * slack_{i,n} & \geq & a_i x_n + b_i, & & \Delta q_{x,i} < 0 \qquad (24) \\ y_n - M * slack_{i,n} & \leq & a_i x_n + b_i, & & \Delta q_{x,i} > 0 \qquad (25) \\ x_n - M * slack_{i,n} & \leq & q_{x,i}, & \Delta q_{y,i} < 0, & \Delta q_{x,i} = 0 \qquad (26) \\ x_n + M * slack_{i,n} & \geq & q_{y,i}, & \Delta q_{y,i} > 0, & \Delta q_{x,i} = 0 \qquad (27) \end{array}$$

Like in Equations 21 and 22, edges on the top and bottom of the obstacle

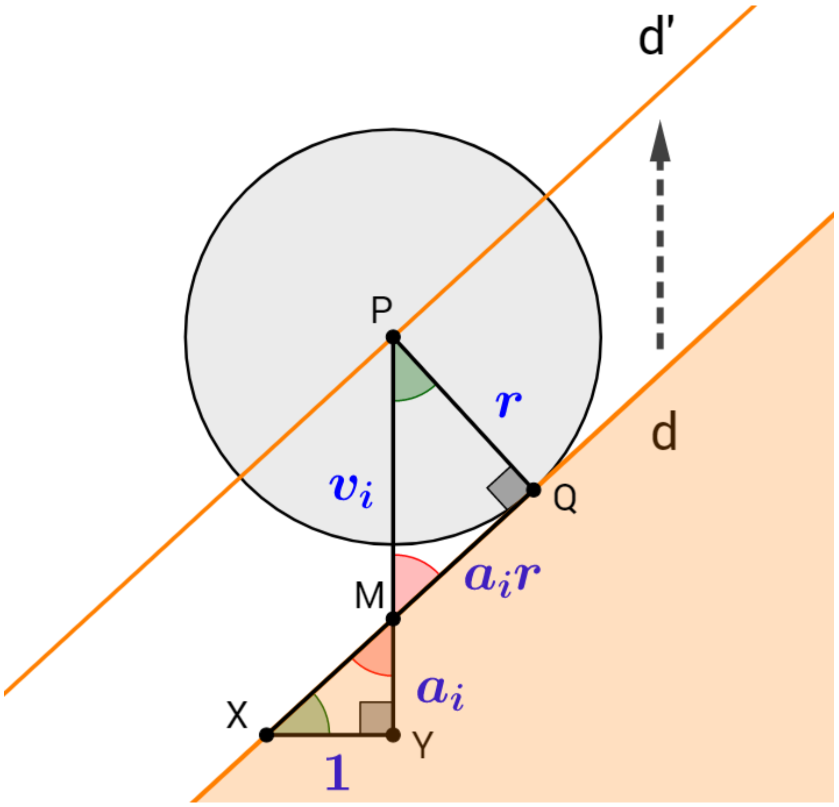


Figure 4: A geometric proof for the vehicle offset formula in Equation 29. Measures are marked in blue. P is the center of mass of the UAV. d is the line defined by edge i of an obstacle. The obstacle is the orange region. We wish to translate d vertically to d' such that when the UAV's center of mass 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: $v_i = \sqrt{r^2 + a_i^2 r^2} = r\sqrt{1 + a_i^2}$.

are treated differently. Equation 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 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 26 and 27 respectively. Finally Equation 28 ensures that at least one of the slack variables must be false at each time step.

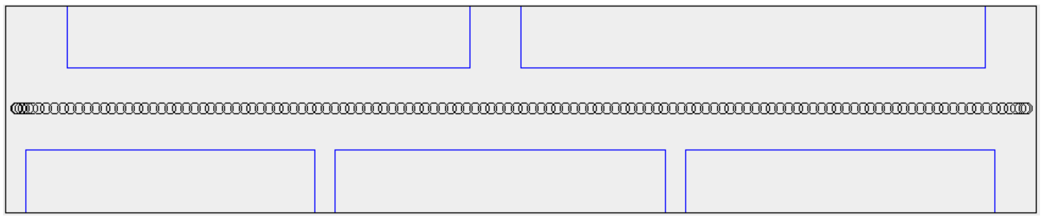
$$\neg \bigwedge_i slack_{i,n} \quad 0 \leq n \leq N \tag{28}$$

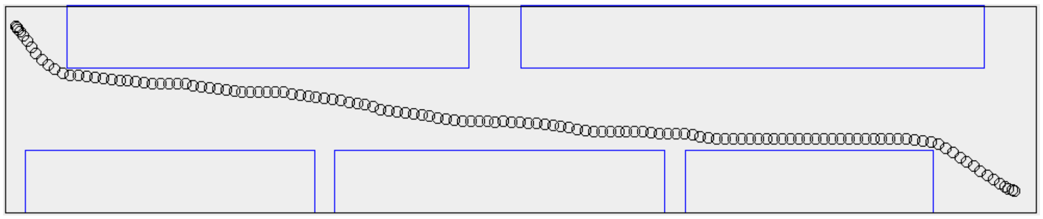
Equations 24 - 27 assume that the UAV has no physical size. The position of the UAV is the position of its center of mass. Because a UAV has certain physical dimensions, it will collide with an obstacle before its center of mass reaches the obstacle. The vehicle's shape is approximated as a circle with 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 29. A geometric proof is given in Figure 4.

$$v_i = r\sqrt{1 + a_i^2} \tag{29}$$

With the vertical offset calculated, Equations 24 - 27 can be extended to Equations 30-31 which take the UAV size into account.

$$\begin{aligned}
y_n + M * slack_{i,n} &\quad - \quad v_i \quad \geq \quad a_i x_n + b_i, & & \Delta q_{x,i} < 0 & (30)\\
y_n - M * slack_{i,n} &\quad + \quad v_i \quad \leq \quad a_i x_n + b_i, & & \Delta q_{x,i} > 0 & (31)\\
x_n - M * slack_{i,n} &\quad + \quad r \quad \leq \quad q_{x,i}, & \Delta q_{y,i} < 0, & \Delta q_{x,i} = 0 & (32)\\
x_n + M * slack_{i,n} &\quad - \quad r \quad \geq \quad q_{y,i}, & \Delta q_{y,i} > 0, & \Delta q_{x,i} = 0 & (33)
\end{aligned}$$


(a) The horizontal scenario.


(b) The diagonal scenario.

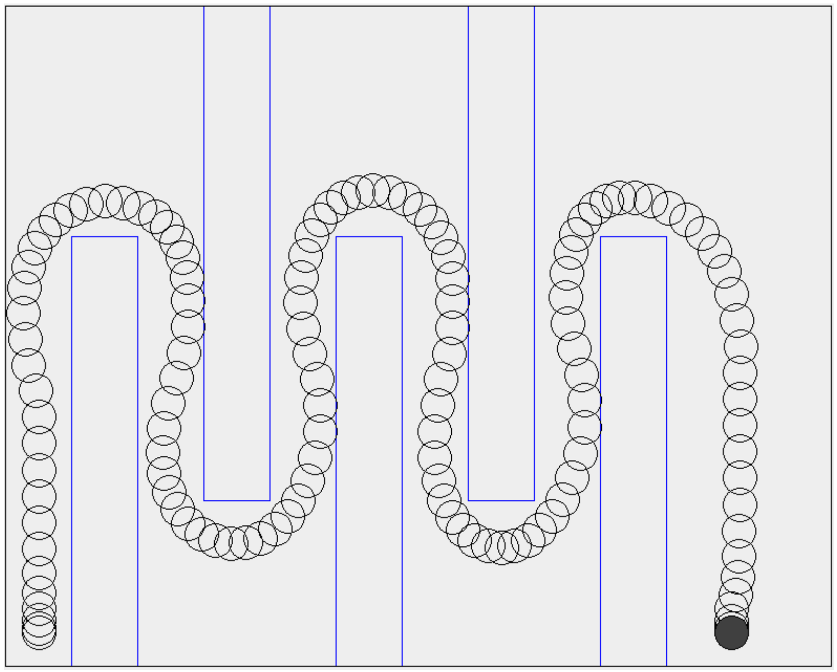

(c) The Up/Down Scenario.

Figure 5: Three different testing scenarios, each with the same amount of obstacles. In each scenario, the UAV needs nearly the same amount of time steps to reach its goal. The only difference is the layout of the obstacles.

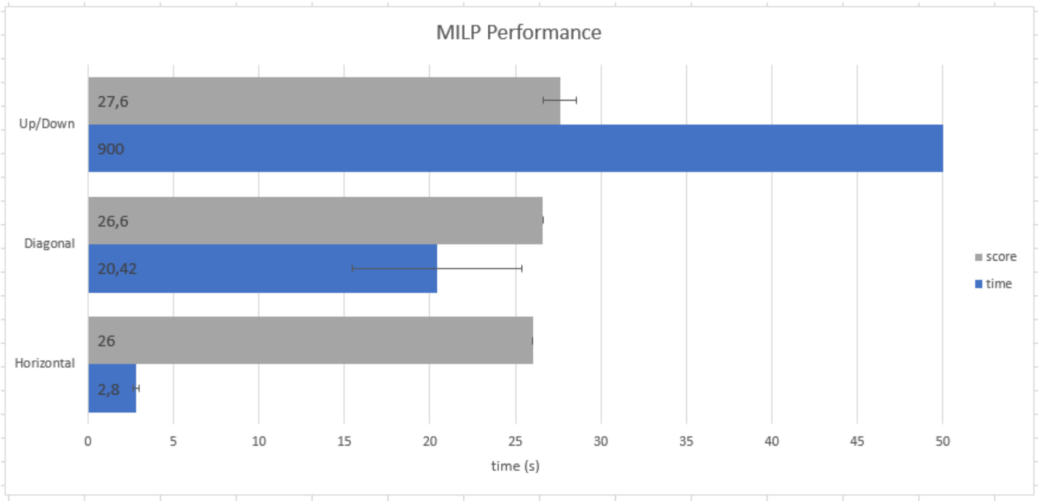


Figure 6: Performance of the MILP problem.

2.7 Performance

Typically, the amount of integer variables is used as a metric for the difficulty of a MILP problem. The amount of integer variables determines the worst case time needed to solve the MILP problem. An integer variable is needed for every edge, for every obstacle, for every time step. Increasing the amount of time steps and obstacles also increases the time needed to solve.
However, the amount of integer variables does not seem to be the only factor that determines the solve time. Figure 5 shows three different scenarios, each with the same amount of obstacles. The time step size is $0.2s$, and in each scenario there were 30 seconds worth of time steps. The scenarios are built such that the time needed for the UAV to reach the goal position is roughly the same (between 26 and 27 seconds). As a result, each scenario has the same amount of integer variables. With 4 edges for all 5 obstacles and 150 time steps, they each have 3000 integer variables to model the obstacles alone.

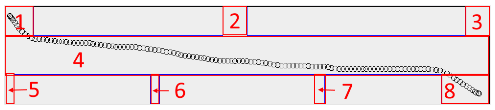
In the Horizontal scenario, the obstacles are not in the way of the UAV. The UAV can move in a straight line from the start position to its goal. In the Diagonal scenario, two of the obstacles are slightly in the way. The UAV is forced to make slight turns near the start and end of the trajectory. However, most of the trajectory is still straight. In the Up/Down scenario, the vehicle has to slalom around every obstacle.

2.7.1 Results

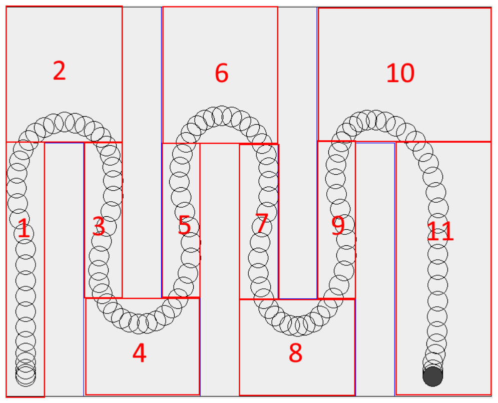
Each scenario was solved five times. Figure 6 shows the time needed to solve the models, as well as the score. The score is also expressed in seconds, and is the time needed for the UAV to reach the goal position when following the generated trajectory.
The time needed to solve these scenarios varies dramatically. The Horizontal scenario is solved consistently in just under 3 seconds. The Diagonal scenario comes in at around 20 seconds, although the 95% confidence interval is $\pm 4.9s$. The Up/Down scenario was limited to 900s of execution time. In this time, the solver could not find the optimal trajectory, which is why the score has a confidence interval.

2.7.2 Discussion

Intuitively, it is not surprising that the Up/Down scenario is harder to solve than the other two. A human pilot would also have more difficulties navigating the Up/Down scenario compared to the others. However, this is cannot be determined by looking a the amount of integer variables or constraints alone.
Problems where the trajectory requires maneuvers seem to be harder to solve. The maneuvers are necessary because there are obstacles blocking a straight approach to the goal position. This observation led me to believe that the integer variables themselves do not cause the poor performance, but the real issue is what they model: a non-convex search space.
The amount of integer variables determines the worst-case performance because it determines the degree to which the search space can be non-convex. An informal definition of this degree of non-convexity would be the minimum amount of (possibly overlapping) convex shapes needed to reconstruct the original shape. With a single boolean variable, you need at most 2 convex shapes. With n boolean variables, at most 2^n convex shapes are needed. TODO: cite.
The degree of convexity of the open space for the Up/Down scenario is 11 as shown in Figure 7b. For the Horizontal and Diagonal scenarios, this is 8 as shown in Figure 7a. There is a difference, but this still does not seem like a good explanation. The Horizontal and Diagonal scenarios have the



(a) 8 convex regions are needed to reconstruct the diagonal scenario.



(b) 11 convex regions needed to reconstruct the Up/Down Scenario.

Figure 7

same degree of convexity, even though there's almost an order of magnitude difference in the solve times.
Even though the global degree of convexity is the same for the Horizontal and Diagonal scenarios, this is not the case for the neighborhood around the trajectory. In the Horizontal scenario, the trajectory only goes through a single convex region. In the Diagonal scenario, the trajectory goes through 3 convex regions. For the Up/Down scenario, the trajectory goes through all 11 regions. I present the following hypothesis:

Hypthesis 1 *The degree of non-convexity around the optimal trajectory is a better indicator of the time needed to solve a MILP trajectory planning problem than the amount of integer variables.*

Thoroughly analyzing this hypothesis is outside the scope for this thesis. The goal of this these is to build a scalable MILP trajectory planning algorithm. Many decisions in this thesis assume that this hypothesis is true, even though I cannot provide a more thorough analysis than the arguments presented here.

3 Segmentation of the MILP problem

This section proposes a preprocessing algorithm to improve the scalability of MILP trajectory planning.
Assuming Hypothesis 1 is true, a high degree of non-convexity around the trajectory causes slow solve times. There are two issues with this. The first issue is that the trajectory is the solution for the problem. If the solution is known, the problem does not need to be solved anymore. The second issue is that even if the optimal trajectory is known, the degree of non-convexity around that trajectory cannot be reduced by much since it already is the optimal trajectory. For instance, the Up/Down scenario in Figure 7 cannot be made more convex without changing the original problem.
However, these issues can be resolved by relaxing the requirement of finding the optimal trajectory. The UAV operates in the real world, so a ”good enough” trajectory will do. The MILP trajectory planning problem can be divided into many sub-problems, each of which solve a small part of the trajectory. By dividing the MILP problem into smaller sub-problems, the guarantee of optimality no longer holds. The optimal trajectory for each sub-problem will be found, but the combined trajectories may not be the optimal solution for the original problem. However, the degree of non-convexity can be limited in each sub-problem, which improves the scalability.
The non-convexity of a trajectory planning problem manifests itself as turns in the trajectory. Because of this, limiting the amount of turns in each sub-trajectory also limits the degree of non-convexity around the sub-trajectory. Limiting the amount of turns in each sub-problem is one of the goals for the preprocessing algorithm.
Before the algorithm can take turns into account when building the sub-problems, the turns in the trajectory need to be known. This trajectory is not known in advance, but this is not required. A faster path planning algorithm can be used to find a path from the start to the goal position. Unlike a trajectory, a path is not time-dependent and does not take the dynamics of the UAV into account. These simplifications make path planning algorithms typically much faster than trajectory planning algorithms. Despite the limitations of the path, the turns in the path do correspond to turns in the trajectory. However, it also means the trajectory may not be the fastest trajectory. Once again, optimality is sacrificed.
TODO: feasibility?

3.0.3 General Algorithm Outline

Algorithm 1 General outline

```
1: $T \leftarrow \{\}$                                          ▷ The list of solved subtrajectories
2: $path \leftarrow$ THETA*($scenario$)
3: $events \leftarrow$ FINDTURNEVENTS($path$)
4: $segments \leftarrow$ GENSEGMENTS($path$, $events$)
5: for each $segment \in segments$ do
6:     UPDATESTARTSTATE($segment$)
7:     GENSAFEREGION($scenario$, $segment$)
8:     GENSUBMILP($scenario$, $segment$)
9:     $T \leftarrow T \cup \{$ SOLVESUBMILP $\}$
10: end for
11: $result \leftarrow$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 sub-problem, 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 with 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).

3.1 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.

3.1.1 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.

3.1.2 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.

Algorithm 2 Theta* Implementation

```
1: function THETA*($scenario$)
2:     $g(v_{start}) \leftarrow 0$
3:     $parent(v_{start}) \leftarrow null$
4:     $queue \leftarrow \emptyset$
5:     $queue$.Insert($v_{start}$, $g(v_{start}) + h(v_{start})$)
6:     $expanded \leftarrow \emptyset$
7:     while $queue \neq \emptyset$ do
8:         $s \leftarrow queue$.Pop()
9:         if $s = v_{goal}$ then
10:            return $s$.GetPath()
11:        end if
12:        $expanded \leftarrow expanded \cup \{s\}$
13:        for each $s' \in$ GenerateNeighbors($s$) do
14:            if $s' \notin expanded$ 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:        $s_{parent} \leftarrow parent(s)$
24:    else
25:        $s_{parent} \leftarrow s$
26:    end if
27:    if $g(s_{prev}) + c(s_{prev}, s') < g(s')$ then
28:        $g(s') \leftarrow g(s_{parent}) + c(s_{parent}, s')$
29:        $parent(s') \leftarrow s_{parent}$
30:        $queue$.Insert($s'$,$g(s') + h(s')$)
31:    end if
32: end function
```

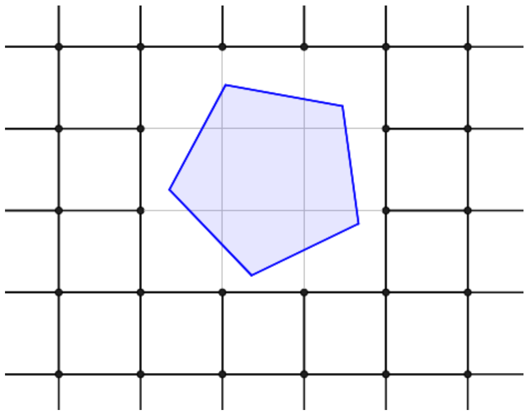


Figure 8: An example of how a grid is used to build the graph for the path finding algorithm. Each point is a node on the graph. If two points are connected by a line, their nodes in the graph also are connected by an edge. Diagonal edges are not shown here for clarity.

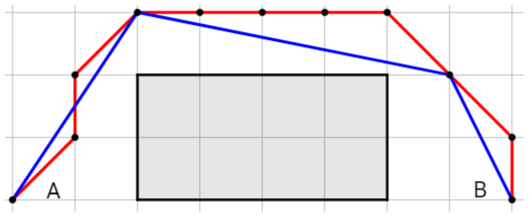


Figure 9: The red line shows a typical A* path, compared to the path found by Theta*. The gray rectangle is an obstacle.

3.1.3 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.
- 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.

3.1.4 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.

3.2 Detecting corner events

According to Hypothesis 1, 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 1 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.

3.2.1 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

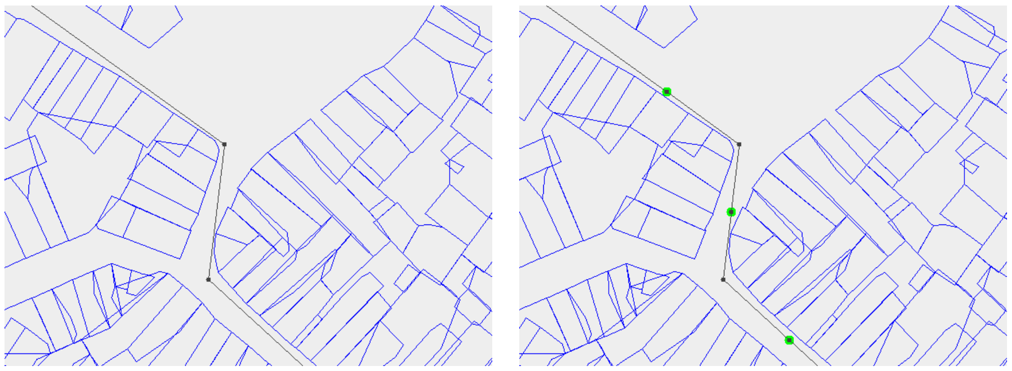


Figure 10: The left image shows the results after the Theta* algorithm has executed. The blue shapes are obstacles, while the gray line is the Theta* path. The right image shows the results after the path is segmented. Extra nodes have been added to the Theta* path, as marked by the green circles. These circles depict the transitions between segments.

- 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!

EXPLAIN ALGO!!!!!!!

Algorithm 3 Finding Turn Events

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

3.3 Generating path segments

3.3.1 Segment Generation Algorithm

code
approachmargin
maxtime
step by step figures with explanation?

3.3.2 Segment data

relevant points: pre path + stop point going in + stop point at finish. image with points clearly labeled
convex hull: quickhull. source + image for demo

Algorithm 4 Generating the segments

```
function GenSegments($path$, $events$)
    $segments \leftarrow \{\}$
    $catchUp \leftarrow true$
    $lastEnd \leftarrow path(0)$
    for $i \leftarrow 0, |events| - 1$ do
        $event \leftarrow events(i)$
        if $catchUp$ then
            expand $event.start$ backwards
            add segments from $lastEnd$ to $event.start$
            $lastEnd \leftarrow event.start$
        end if
        $nextEvent \leftarrow events(i+1)$
        if $nextEvent.start$ is close to $event.end$ then
            $mid \leftarrow$ middle between $event$ & $nextEvent$
            add segment from $lastEnd$ to $mid$
            $lastEnd \leftarrow mid$
            $catchUp \leftarrow false$
        else
            expand $event.end$ forwards
            add segment from $lastEnd$ to $event.end$
            $lastEnd \leftarrow event.end$
            $catchUp \leftarrow true$
        end if
    end for
    add segments from $lastEnd$ to $path(|path| - 1)$
    return $segments$
end function
```

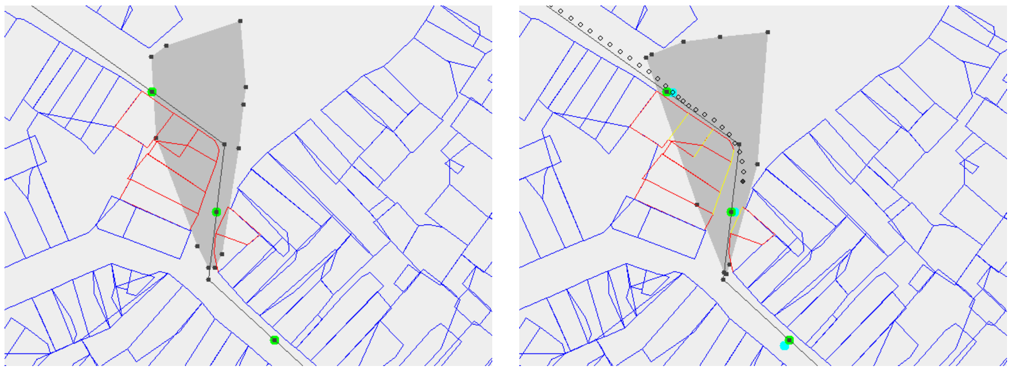


Figure 11: The left image shows the result after the genetic algorithm has executed. The obstacles in red have been selected to be modeled in the MILP problem. The dark grey shape is the convex allowed region generated by the genetic algortihm. Note how it does not overlap with any of the blue obstacles. 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

note: only when about to be solved!

3.4 Generating the active region for each segment

One of the main goals of segmenting the path is to reduce the amount of obstacles. Every segment has a set of active obstacles associated with it, being the obstacles that need to be modeled for the solver. Not only the obstacle that "causes" the corner is important, but obstacles which are nearby are important as well. Obstacles on the outside of the corner also may play a role in how the vehicle approaches the corner. To find all potentially relevant obstacles, the convex hull of the (Theta*) path segment is calculated and scaled up slightly. Every obstacle which overlaps with this shape is considered an active obstacle for that path segment. The convex hull step ensures that all obstacles on the inside of the corner are included, while scaling it up will cover any restricting obstacle on the outside of the corner.
The inactive obstacles also need to be represented. To do this, a convex polygon is constructed around the path. This polygon may intersect with the active obstacles (since they will be represented separately), but may not

intersect any other obstacle. The polygon is grown using a genetic algorithm. Genetic algorithms are inspired by natural selection in biology. A typical genetic algorithm consists of a population of individuals, a selection strategy and one or more operators to generate offspring. In each generation, the operators are applied on the population to produce offspring. These operators usually have a random element and are responsible for exploration of the search space. The selection strategy determines, often based on a fitness function, which individuals survive and form the population for the next generation. Selection is responsible for convergence towards fitter individuals, limiting how much of the search space is evaluated.

3.4.1 Implementation of the genetic algorithm

Alg. 5 shows our implementation. In our implementation, each individual in the population represents a single legal polygon. A legal polygon is convex, does not self-intersect, does not overlap with inactive 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 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 $N_{genes,min}$ and $N_{genes,max}$. The mutator attempts to nudge every vertex/gene to a random position at most Δ_{nudge} away (line 15). If the resulting polygon is not legal, it retries at most $N_{attempts}$ 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 active obstacles in yellow and red, as well as the polygon generated by the genetic algorithm in dark gray.

library
tournament selection
offspring generation: mutate only

Algorithm 5 Genetic Algorithm

```
 1: function GenActiveRegion(scenario, segment)
 2:     pop ← SeedPopulation
 3:     for $i \leftarrow 0, N_{gens}$ 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 probability $p_{add}$
13:         OR remove vertex with probability $p_{remove}$
14:         for each gene ∈ individual.chromosome do
15:             randomly move vertex by at most $\Delta_{nudge}$
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
```

initial population
chromosome/gene description
fitness function
validation
parameters
obstacle selection based on path length?

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.

4.1.1 Indicator constraints

Big M method: functional, but solvers struggle with large value of M. Low value of M -¿ may cause wrong behavior. Especially obvious when near vertical line.
Can approximate near vertical line as vertical line, but other solution is integer constraint from cplex. serves same role as big M method, but can't fail, is clearer and provides more information to solver.

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. 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.

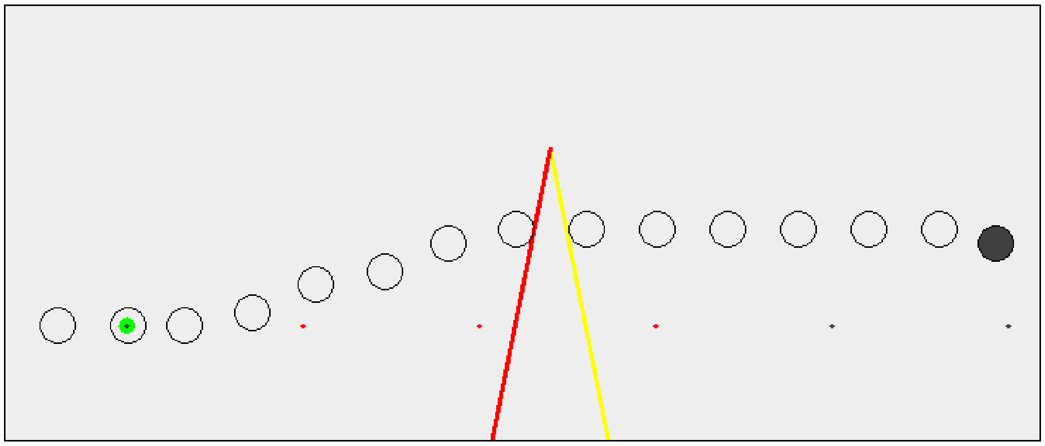


Figure 12: An example of corner cutting

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 12 shows an example of this. Each time step on its own is a valid position, but a collision is ignored between the time steps.
As described in section (TODO: ADD REF), Equation 34 and 35 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.

$$slack_{o,i,n} \Rightarrow \begin{cases} b_{o,i} \leq p_{n,y} - a_{o,i}p_{n,x} & dx_{o,i} < 0 \\ b_{o,i} \geq p_{n,y} - a_{o,i}p_{n,x} & dx_{o,i} > 0 \end{cases} \tag{34}$$

$$\neg \bigwedge_{i} slack_{o,i,n} \quad 0 \leq n < N \tag{35}$$

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. They apply the same constraints again,

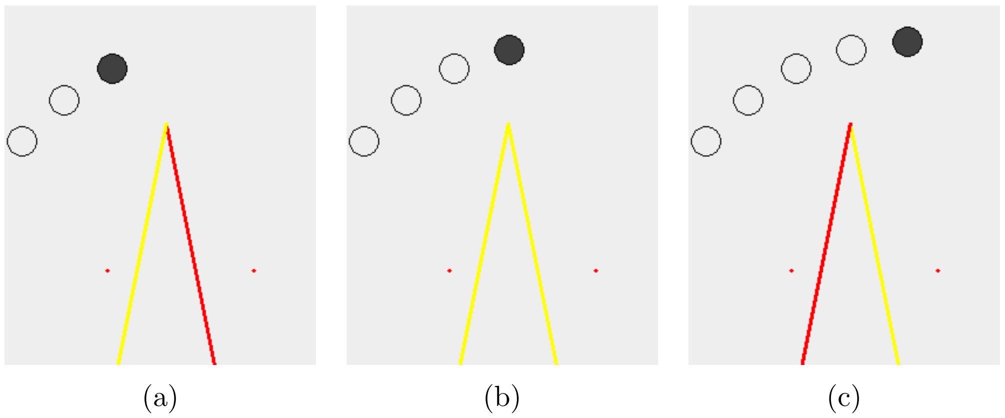


Figure 13

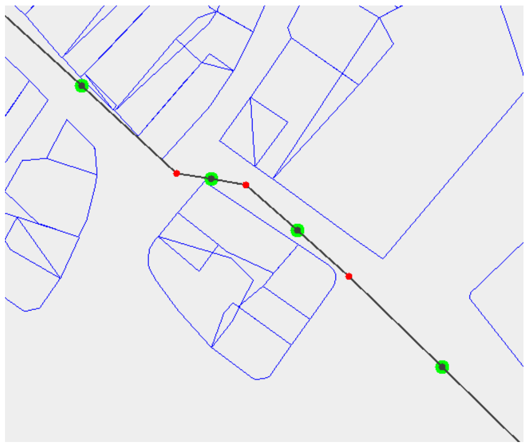


Figure 14: Max goal vel

but this time on the position of the UAV in the last time step:

$$slack_{o,i,n} \Rightarrow \begin{cases} b_{o,i} \leq p_{n-1,y} - a_{o,i}p_{n-1,x} & dx_{o,i} < 0 \\ b_{o,i} \geq p_{n-1,y} - a_{o,i}p_{n-1,x} & dx_{o,i} > 0 \end{cases} \tag{36}$$

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

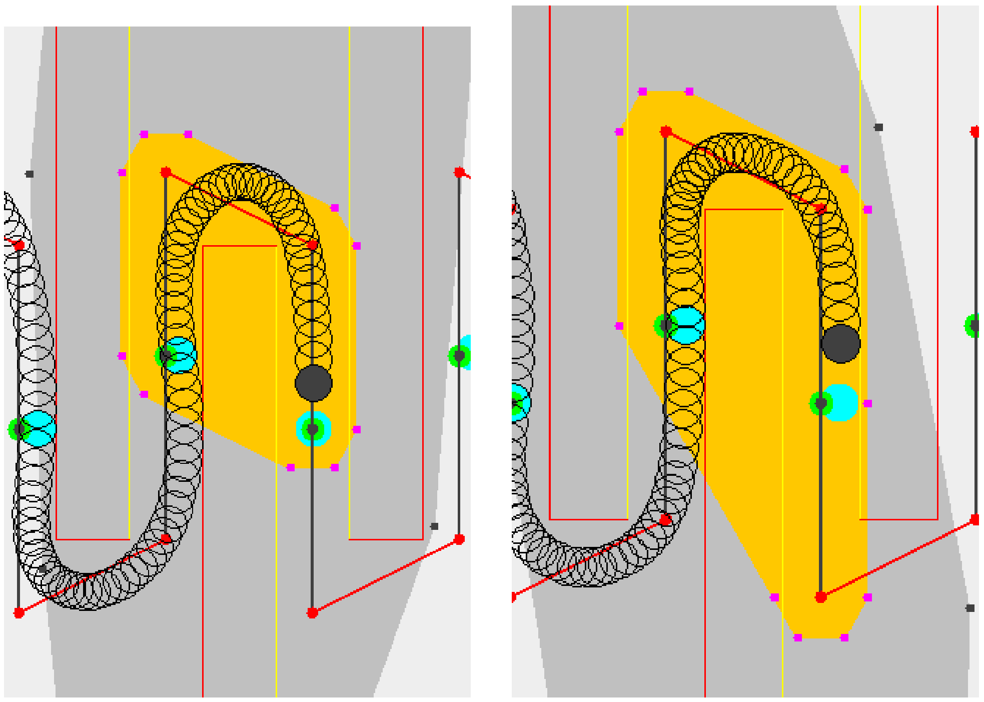

(a) (b)

Figure 15

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.
As a solution for this, the UAVs velocity at the goal of the first sub-problem can be limited so it can stop in time for the corner in the next segment. TODO: equation, ref figure.

4.3.2 GA start area

not just path: stop point based on start state stop point based on maximum goal velocity and goal: ensures vehicle can stop in next segment

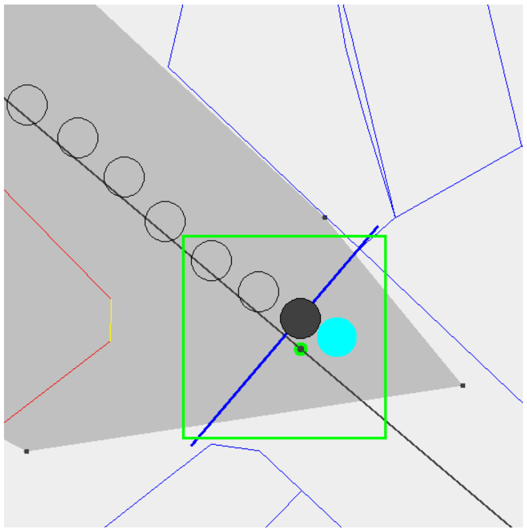


Figure 16: REPLACE! square is offset

4.3.3 Goal conditions

larger epsilon on goal condition prevent uav from finishing early -¿ use line

4.4 Overlapping segment transitions

overlap...

4.5 Visualizing solution

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 is a form of declarative programming. Instead of defining "how" to solve a problem (like with imperative programming) you define the problem itself.
This 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 caused the failure, but that constraint is not necessarily the one that is wrong. 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.

black box solver means debugging is tricky. Need other way to get insight in what's happening. Visualization tool

- obstacles
- pre path
- corners
- path segments
- active region
- active obstacles w/ slack vars

- solver path
- exact value readout

5 Analysis and Results

5.1 Scenarios

- Benchmark: a lot of corners with minimal amount of obstacles
- Spiral: Also many corners, minimal distance
- SF: real world grid. Many obstacles, corners nearly always 90 degrees. Very predictable for algorithm
- Leuven: real world and irregular. Even more obstacles. Very unpredictable

To test the complete algorithm, several different scenarios have been used. Each scenario has unique characteristics and was tested with two different problem sizes. Theta* is always executed with a grid size of 2m and each time step has a duration of 200ms. 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. The machine runs on Windows 10 using version 12.6 of IBM CPLEX. Figure 17 shows these scenarios visually. Table 18 shows detailed information about the scenarios and execution times.

5.1.1 Up/Down Scenario

The first test scenario has very few obstacles, but lays them out in a way such that the vehicle needs to slalom around them. The small scenario has only 5 obstacles, while the larger one has 9. This is a very challenging scenario for MILP because every obstacle has a large impact on the path. Without segmentation on the version of the scenario, the solver does not find the optimal path within 30 minutes. If execution is limited to 10 minutes, the best solution it finds takes 26.0s to execute by the vehicle. That is less than a second faster than the segmented result while it took more than 20 times more execution time to find that solution. For the larger scenario with 9 obstacles, the solver could not find a solution within 30 minutes. This scenario clearly shows the advantages of segmentation, even if there only are a few obstacles.

5.1.2 San Francisco Scenario

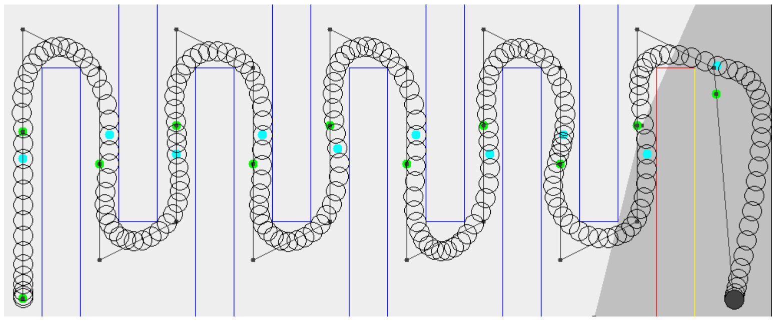
The San Francisco scenario covers a 1km by 1km section of the city for the small scenario, and 3km by 3km section for the large scenario. All the obstacles in this scenario are grid-aligned rectangles laid out in typical city blocks.

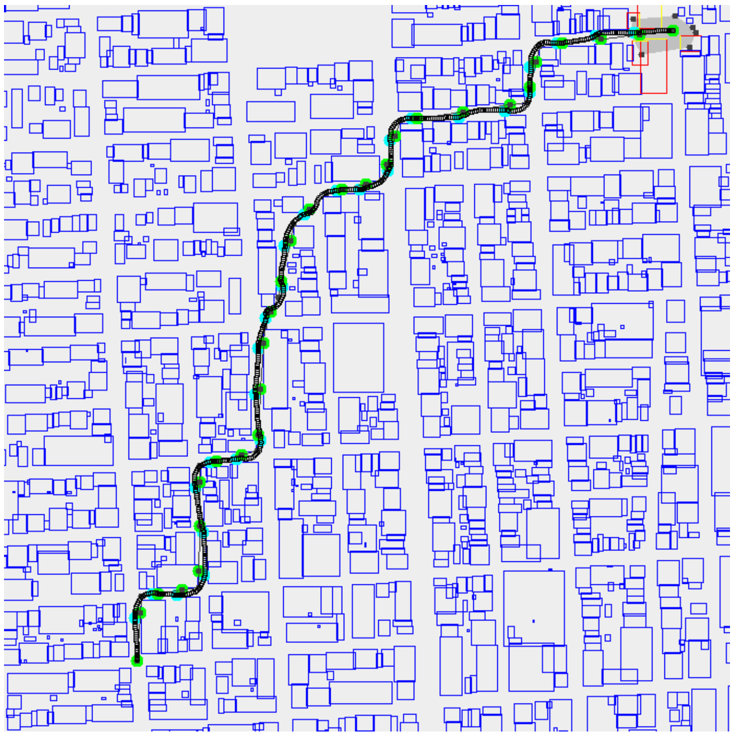
Because of this, density of obstacles is predictable. This scenario showcases that the algorithm can scale to realistic scenarios with much more obstacles than is typically possible with a MIP approach. With these constraints, parameters and hardware, the path can be planned faster than the vehicle can execute it.

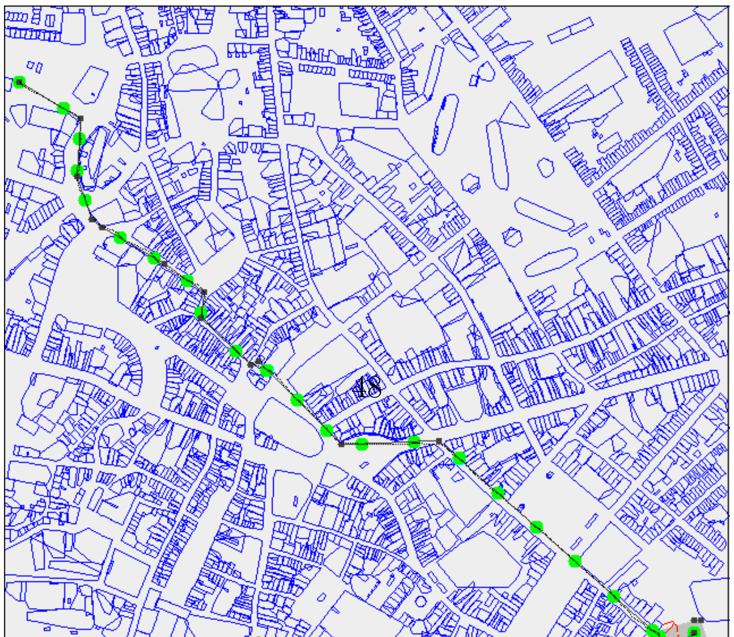
5.1.3 Leuven Scenario

The Leuven scenario also covers both a 1km by 1km and 3km by 3km section, this time of the Belgian 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 algorithm does slow down, but it is still fast enough for offline planning. As visible in figure 11, there are many obstacles clustered close to each other, with many edges being completely redundant. For a real application, a small amount of preprocessing of the map data should be able to significantly reduce both the amount of obstacles as the amount of edges.

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


(a)


(b)



scenario	# obstacles	world size	path length	# segments	Theta* time	Gen. Al
Up/Down Small	5	25m x 20m	88m	7	0.09s	1.10s
Up/Down Large	9	40m x 20m	146m	11	0.14s	1.62s
SF Small	684	1km x 1km	1392m	28	2.04s	9.56s
SF Large	6580	3km x 3km	4325m	84	18.14s	18.21s
Leuven Small	3079	1km x 1km	1312m	29	2.29s	29.83s
Leuven Large	18876	3km x 3km	3041m	61	18.14s	83.69s

Figure 18: The experimental results for the different scenarios

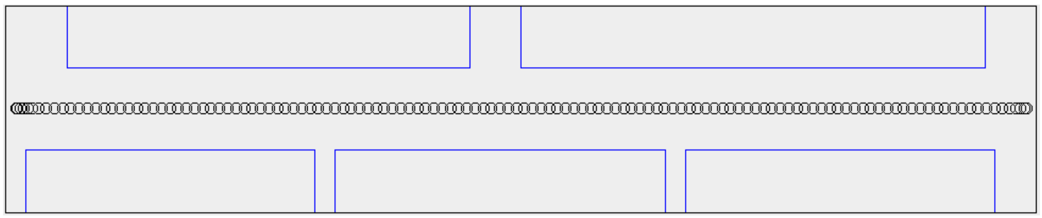
5.2 Convexity of Search Space

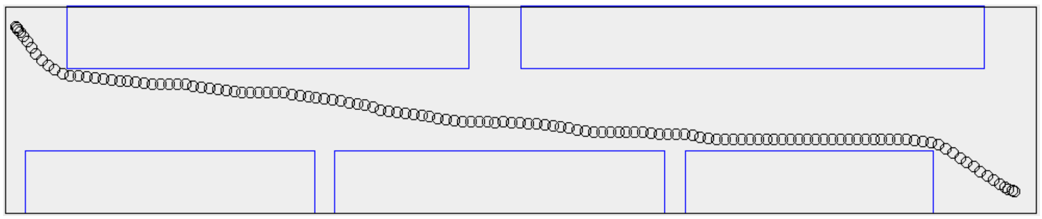
The convexity of the search space plays a large role in the difficulty of a MILP problem. My approach is built entirely around the idea of keeping the search space as convex as possible in each segment. To analyze the importance of convexity, I tested three scenarios without any form of preprocessing. These scenarios are designed so the optimal trajectory has roughly the same duration in each scenario. The scenarios also each have 5 grid aligned rectangles as obstacles.
As a result, MILP problems for all three scenarios have nearly identical amounts of integer variables. The difference between these scenarios is the convexity of the search space.
In the first scenario, the obstacles are not in the way of the UAV. The UAV can move in a straight line from the start position to its goal. In the second scenario, two of the obstacles are slightly in the way. The UAV is forced to make slight turns near the start and end of the trajectory. However, most of the trajectory is still straight. In the third scenario, the vehicle has to slalom around every obstacle.

5.2.1 Results

5.2.2 Interpretation

Even though all scenarios have nearly identical amounts of time steps, integer variables and obstacles, the solve time varies greatly. This confirms the core assumption behind my approach: The convexity of the search space plays a larger role in the performance than the amount of integer variables.


(a)


(b)

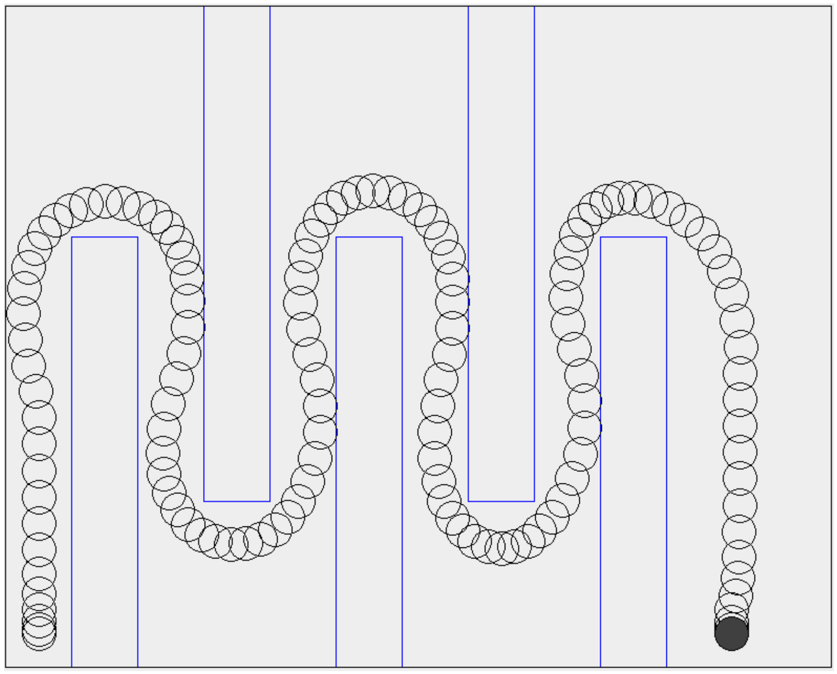

(c)

Figure 19

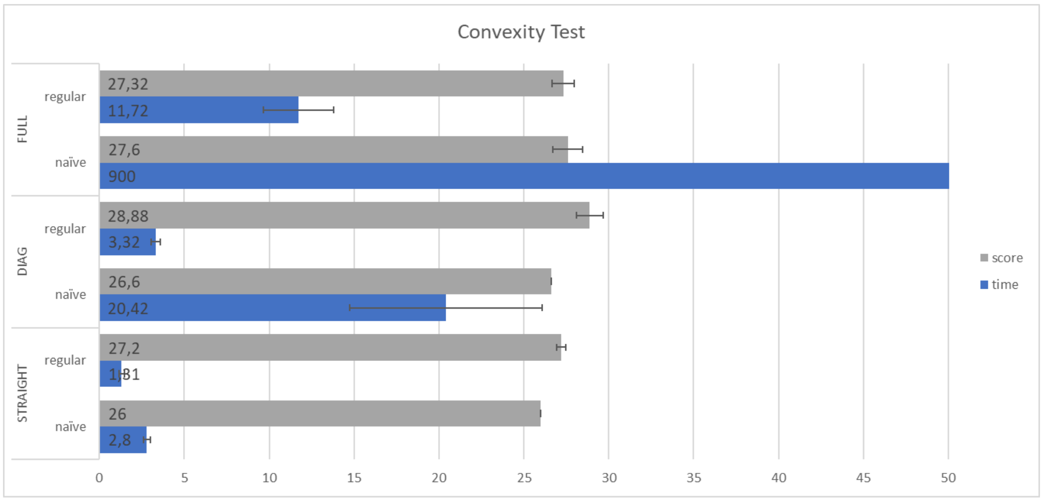
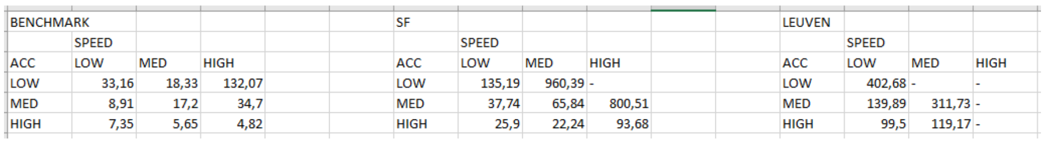


Figure 20: convexity data

5.3 Genetic Algorithm Parameters

tweak parameters...



BENCHMARK			
	SPEED		
ACC	LOW	MED	HIGH
LOW	33,16	18,33	132,07
MED	8,91	17,2	34,7
HIGH	7,35	5,65	4,82

SF			
	SPEED		
ACC	LOW	MED	HIGH
LOW	135,19	960,39	-
MED	37,74	65,84	800,51
HIGH	25,9	22,24	93,68

LEUVEN			
	SPEED		
ACC	LOW	MED	HIGH
LOW	402,68	-	-
MED	139,89	311,73	-
HIGH	99,5	119,17	-

caption

5.4 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 is the distance the UAV needs to accelerate from zero to its maximum velocity, or decelerate from the maximum velocity to zero.
In this experiment, I tested relation between the maximum velocity and the maximum acceleration of the UAV. I used 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.
TODO: add # segments

Even though some combinations did not finish for the larger scenarios, the results are relatively consistent across all three scenarios:
Within each speed category, a higher acceleration always reduces solve times. This is as expected, because a higher acceleration will reduce the expansion distance for the segments, making them both shorter in time as well as smaller so less obstacles are expected to be modeled.
Increasing the velocity for the low and medium acceleration tends to make things slower. Low velocity and high acceleration always score well

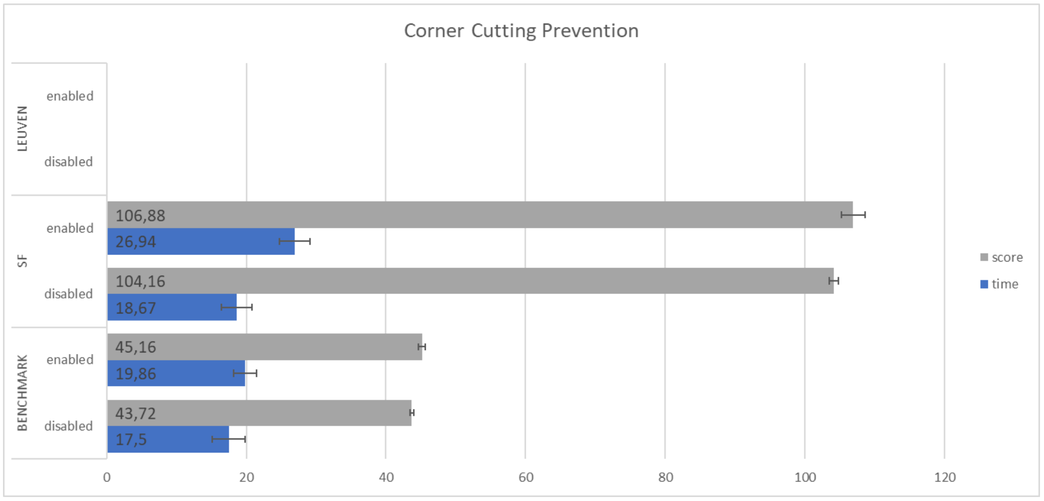


Figure 21: corner cutting data

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.

5.5.1 Result

The performance impact caused by preventing corner cutting seems minimal (TODO: more precise!).

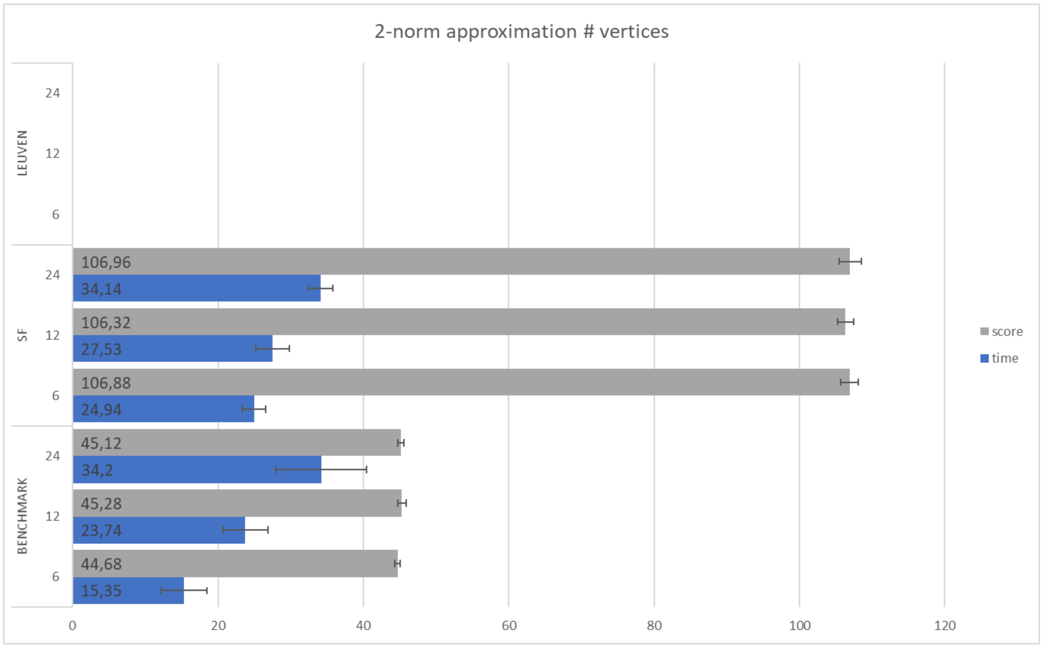


Figure 22: linear approx data

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.

5.6.1 Results

In all cases, increasing the amount of vertices used to approximate the 2-norm also increases the solve time. Increasing the amount of vertices did not have a measurable effect on the score of the trajectory (EXCEPT LEUVEN?),

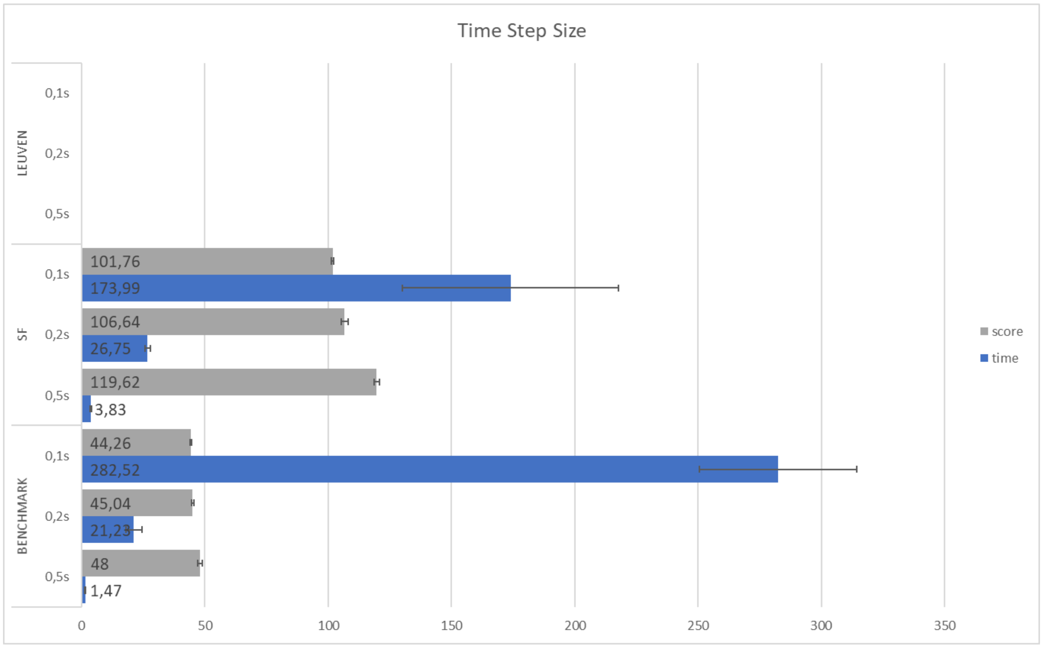


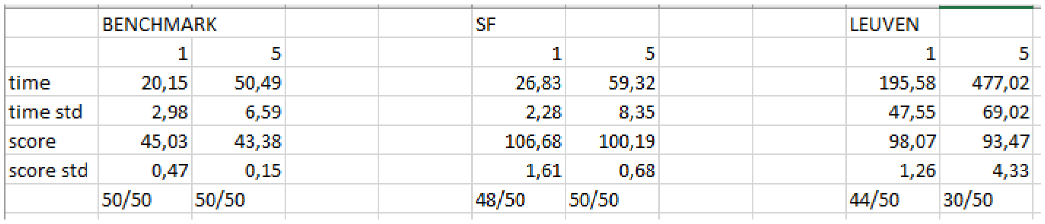
Figure 23: time step data

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 Results

The time step size has a dramatic impact on performance. For the benchmark scenario, each increase in the amount of time steps resulted in around 14x the computation time. For the San Francisco scenario, that is around 6-7x. In both cases the trajectory time improved with more time steps as well, although the difference is smaller when going from $0.2s$ to $0.1s$ compared to $0.5s$ to $0.2s$.



	BENCHMARK				SF				LEUVEN	
	1	5			1	5			1	5
time	20,15	50,49			26,83	59,32			195,58	477,02
time std	2,98	6,59			2,28	8,35			47,55	69,02
score	45,03	43,38			106,68	100,19			98,07	93,47
score std	0,47	0,15			1,61	0,68			1,26	4,33
	50/50	50/50			48/50	50/50			44/50	30/50

Figure 24: stability data

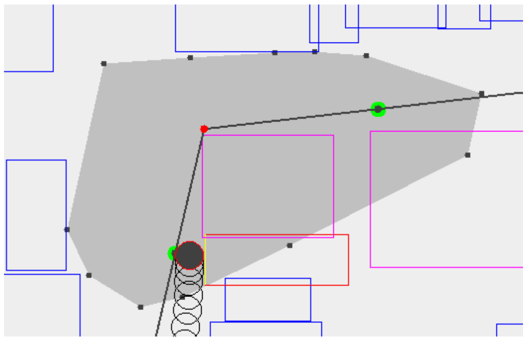


Figure 25: A case where the transition fails

5.8 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.

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 25. This may be due to a bug, or some assumption which is false. I did not manage to find the cause for this, but overlapping the segments should mitigate this problem. For this reason, each scenario was also tested with an overlap of 5 time steps per segment.

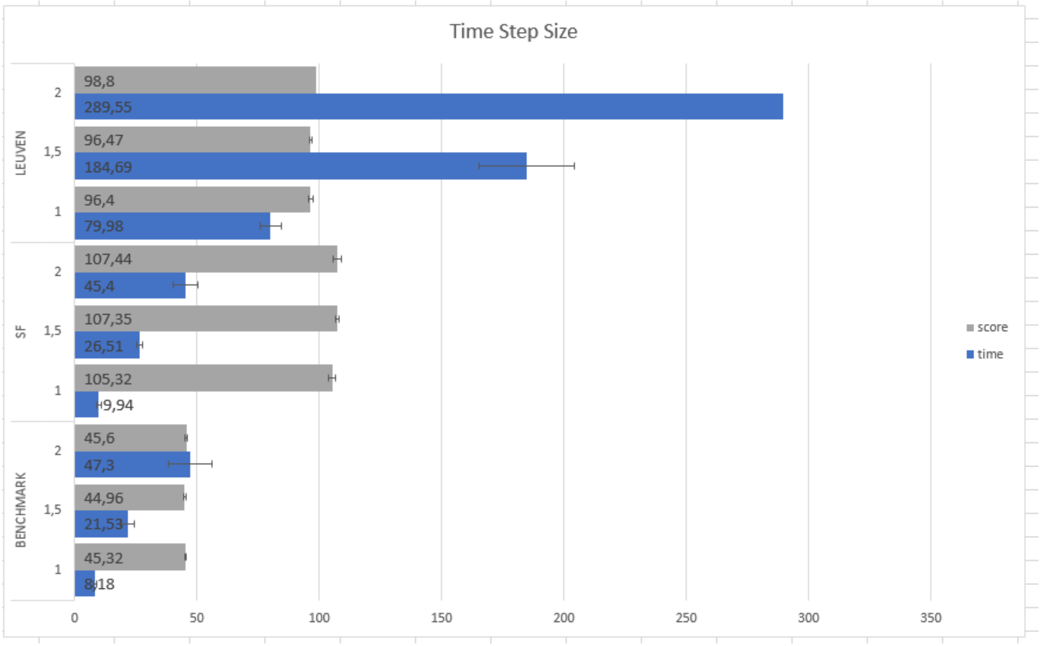


Figure 26: maxtime data

5.9 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 test, along with a multiplier of 2.

5.9.1 Results

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 27: approach data

5.10 Approach Margin

6 Discussion

The goal of this thesis is build a scalable approach for MILP trajectory planning. The target vehicles are multirotor UAVs, so the results in the previous section need to be analyzed in that context.
The first step is analyzing whether or not the approach is actually scalable enough when planning the trajectory for this kind of vehicle through complex environments.
Afterwards, the other experiments demonstrate the importance of some of the parameters used in the algorithm. The results provide insight in the limitations of the current approach, and how it may be improved in the future.

6.1 Research question result

The general performance results show that the algorithm is capable of planning a long trajectory through complex environments. Even for the smallest scenarios, the solver struggles to find a trajectory without preprocessing. With my algorithm, the UAV can successfully navigate an entire city.
The convexity test demonstrates that convexity of the search space is indeed a large factor. This supports the assumption that my entire algorithm is built upon: Maintaining convexity as much as possible is required to make the algorithm scale.
The stability test shows that, for the most part, my segmentation approach preserves stability. There still are some issues around the transitions between segments, but they can be eliminated by slightly overlapping the segments. This overlap comes at a performance cost, but it also improves the quality of the trajectory. Overall I am not entirely satisfied with the stability of my current implementation, however I do believe the small issues around segment transitions can be resolved.

The results show a large improvement in scalability in certain realistic scenarios, but the choice of those scenarios has drastically impacted the algorithm I have developed.

During development, I have always used realistic approximations of the capabilities of multirotor UAVs. Those can reach high accelerations but have relatively low maximum velocities compared to what fixed-wing aircraft can achieve. This makes those vehicles very agile, which is one of the contributing factors to their recent popularity.
The assumption of this agility means that my algorithm cannot be applied to UAVs which do not have that property. One of the properties my algorithm uses often is the maximum acceleration distance, which is the distance in which the UAV can always come to a complete stop. This works fine with multirotor UAVs, but is a meaningless concept when dealing with fixed-wing UAVs which cannot stop at all during flight.
However, those kinds of low-agility UAVs are unlikely to be deployed at low altitudes in dense city centers exactly because they lack agility. Even with perfect planning, cities are very unpredictable places. An multirotor UAV is much more likely to be able to safely react to an unexpected obstacle than a fixed-wing UAV.
While I picked out fixed-wing UAVs as an example, the same arguments hold for any kind of UAV that either cannot hover or has low agility. Some UAVs may be able to hover, but are not very agile due to a high maximum velocity and low acceleration. In this case, the agility can be improved by reducing the maximum velocity.

The density of obstacles is also an extremely important factor. The Leuven scenario is significantly harder than the San Francisco scenario because the obstacles are smaller and closer together. Not only are the obstacles closer together, but they are also polygons and can have many more edges per obstacle. For scenarios where the obstacles are significantly denser or complex than in the Leuven scenario, the approach may not improve the performance enough.
On the other hand, the Leuven map is much more detailed than is required for navigation. Many obstacles could be merged together without a significant on possible trajectories. The obstacles could also be simplified, reducing the amount of edges per obstacle. Given that the Leuven map is unprocessed except for calculating the convex hull of each obstacle, the algorithm should

be able to handle most real world maps when properly prepared.

Given these considerations, I conclude that my approach meets the basic requirements. The assumptions behind the design of the algorithm seem to be valid based on experiments.

6.2 Factors

While the chosen parameter values allow the algorithm to scale well, different values may be chosen to find a different balance between performance and solution quality.
The corner cutting prevention makes the trajectory slightly slower, but that is to be expected since cutting a corner is faster than going around. The performance does take a hit, but the extent is minimal. The corner cutting prevention constraints seem certainly worth being enabled.
The 2-norm approximation has a slightly larger effect on performance. However, there does not seem to be an impact on the trajectory speed. This value could be reduced for a small performance gain.

The time step size and maximum time both have a dramatic impact on the performance. The time step size should be no smaller than necessary, and the maximum time should be as small as possible. They are by far the most important performance factors. As a result, further improvements on performance should focus on these factors. A way to improve performance may be solving each segment twice: Once with a high time step size and a conservative maximum time, and another time with a smaller time step size and a very tight maximum time. The first run quickly and provides a tight upper limit on the amount of time needed for the segment. The second run would also run significantly faster, since the amount of time steps modeled is much closer to what is actually needed. Due to a lack of time, I was unable to implement this.

The approach margin data is probably the strangest. Going from a low to medium approach margin increases the solve time and improves the trajectory, as expected. However, increasing the approach margin again actually decreases the solve time again, as well as reducing the quality of the trajectory. This is unexpected since a approach margin should lead to larger segments which take longer to solve. The larger segments should also improve the quality of the solution because corners can be taken more efficiently, yet the the opposite happens in the data. TODO: find answer?

6.3 Future work

The obvious next 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.

Another extension I would try is using moving to Mixed Integer Quadratic programming. Linear approximations to limit the length of vectors works, but it also introduces artifacts into the path. Increasing the amount of constraints that model the norm helps minimize the impact, but comes at a performance cost. Stating those constraints directly as a quadratic function would reduce the amount of constraints needed per time step. Even though the performance cost of a more accurate linear approximation is limited, this could still improve performance while increasing the accuracy of the model. Especially when the problem is also extended to 3D, since the this would require the linear approximation of a sphere instead of a circle.

6.4 actual contribution

6.5 goals reached?

6.6 challenges

6.7 evolution of understanding

6.8 critical review

7 Conclusions

Path planning using MIP was previously not computationally possible in large and complex environments. The approach presented in this paper shows that these limitations can effectively be circumvented by dividing the path into smaller segments using several steps of preprocessing. The specific algorithms used in each step to generate the segments can be swapped out easily with variations. Because the final path is generated by a solver, the constraints on the path can also be easily changed to account for different use cases. The experimental results show that the algorithm works well in realistic, city-scale scenarios, even when obstacles are distributed irregularly and dense.

7.1 Future work

The results so far are promising, but have not been used on real hardware yet. Extending the software we built so it can be tested with actual hardware is an obvious next step. That also leads to the next possible extension: Currently the algorithm works in 2D, but extending it to 3D would allow it to be used in more kinds of environments. We'd also like to allow for more kinds of constraints on the path of the vehicle.

References

[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.