Two-Phase Scalable Mixed-Integer Path Planning for UAVs

Jorik De Waen

May 12, 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 and Goal for the Thesis

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 Structure of the Thesis

Section 2.2 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 6 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.

1.3 Related work

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.

2 Modeling Path Planning as a MILP problem

2.1 Introduction

TODO: REWRITE MILP is a form of mathematical programming, form of declarative programming. contrast with imperative programming: say what the solution looks like instead of how to get there. Mathematical programming: subset of declarative programming where problem is defined as mathematical problem. Solver computes result.
Big advantages:
- dont need to say how it is done, focus is on precisely modeling problem.
- flexible: can easily change rules: make problem more restrictive or relaxed, change the goal, etc "just works"
- can be really fast thanks to work on solvers

disadvatages:
- no free lunch (find reference): for anything more than very basic cases, solver cannot guarantee that a solution will be found any faster than random search
–¿ cant rely on just the solver doing the work, problem needs to be stated in a way that guides solver in the right direction –¿ even when careful, as problem becomes more complex, solvers struggle
- hard to understand: solver finds solution or not. –¿Solution is optimal, but if it is not as expected it does not give any information why. –¿when no solution: can show which constraint failed, but problem is not necessarily there. complex interplay between constraints is extremely hard to debug

2.2 Overview of MILP

Mixed integer linear programming is and extension of linear programming. In a linear programming problem, there is a single (linear) target function which needs to be minimized or maximized by the solver. For the work in this paper, we used the IBM CPLEX solver. A problem typically also contains a number linear inequalities which constrain the values of the variables in the target function.

In the example in figure 1, $f(\boldsymbol{x})$ is the linear goal function to be maximized.

maximize $\quad f(\boldsymbol{x}) = a_0 x_0 + a_1 x_1 + ...$
subject to
$b_{0,0} x_0 + b_{0,1} x_1 + ... \leq c_0$
$b_{1,0} x_0 + b_{1,1} x_1 + ... \leq c_1$
...
$x_0, x_1, ... \in \mathbb{R}$

Figure 1: A typical linear problem

When a symbol is in **bold**, it represents a vector, otherwise it represents a scalar value. In this case, $\boldsymbol{x}$ is the vector containing all variables x_i. The linear inequalities below the goal function are the constraints as linear inequalities with coefficients $b_{j,i}$ and the maximum value c_j. The final line ensures that all variables have a real domain. When some (or all) of the variables have an integer domain, the problem is a mixed integer problem. Using multiple inequalities and possibly additional variables, it is possible to model more complex mathematical relations. Some of those relations, like logical operators or the absolute value function, can only be expressed when integer variables are allowed [11].

2.3 Time and vehicle state

The path planning problem can be represented with a finite amount discrete time steps with a set of state variables for each epoch. The amount of time steps determines the maximum amount of time the vehicle has in solution space to reach its goal. The actual movement of the vehicle is modeled by calculating the acceleration, velocity and position at each time step based on the state variables from the previous time step.

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

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

Equation 1 and 2 model the progress of time. $time_t$ represents the current time at timestep t. Δt is the duration of a single time step. There are $N + 1$ time steps.

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

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

Equation 3 and 4 model the position of the vehicle at each time step. p_0 is initialized with a certain starting position p_{start}. For each time step t, the position in the next time step p_{t+1} is determined by the current position p_t, the current velocity v_t and the duration of the time step Δt. Note that both the position and velocity use vector notation.

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

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

Similarly to the position, equation 5 and 6 model the velocity of the vehicle at each time step. This time the starting velocity is initialized with v_{start} and updated based on the current velocity and current acceleration.
The acceleration can be modeled the same way based on the jerk, which can in turn be modeled based on the snap, etc. How many derivates need to be modeled will vary depending on the specific use case.

2.4 Goal function

The problem also needs a goal function to optimize. In this model, the goal is to minimize the time before a goal position is reached. This is expressed in equation 7. Reaching the goal causes a state transition from not being finished to being finished. This is represented as the value of fin_t, which is a binary variable (which is an integer variable which can only be 0 or 1). When fin_t is true, the vehicle has reached its goal on or before time step t.

$$minimize \quad N - \sum_{t=0}^{t \leq N} fin_t \tag{7}$$

$$fin_0 = 0 \tag{8}$$

$$fin_N = 1 \tag{9}$$

$$fin_{t+1} = fin_t \vee cfin_{t+1}, \quad 0 \leq t < N \tag{10}$$

Equation 8 initializes fin_0 to be false, ensuring that the initial state is not finished. Equation 9 forces the state in the last time step to be finished. This means that the vehicle must reach its goal eventually for the problem to be considered solved.
Modeling state transitions directly can be error-prone, so Lamport's [9] state

transition axiom method was used. In this simple model it is still possible to model the state transition directly, but the state transition axiom notation is easier to extend. In equation 10, the state will be finished at time step $t+1$ if the state is finished at time step t or if there is a state transition from not finished to finished at time step $t+1$, represented by $cfin_{t+1}$.

$$cfin_t = cfin_{p,t} \wedge \neg fin_t \quad 0 \leq t \leq N \tag{11}$$

$cfin_t$ in equation 11 is true at time step t if the goal position requirement $cfin_{p,t}$ is true and the finished state has not already been reached ($\neg fin_t$). This shows the advantages of the state transition axiom method: Constraints on the state transition can easily be added and removed here without having to change equation 10.

$$cfin_{p,t} = \bigwedge_{i=0}^{i<Dim(\boldsymbol{p}_t)} |p_{t,i} - p_{goal,i}| < \epsilon_p, \quad 0 \leq t \leq N \tag{12}$$

The goal position requirement is represented by $cfin_{p,t}$ and is satisfied when $cfin_{p,t} = 1$. The coordinate in dimension i of the position vector $\boldsymbol{p}_t$, is $p_{t,i}$. The goal position coordinate in that dimension is $p_{goal,i}$. If the difference between those values is smaller than some value ϵ_p in every dimension at time step t, $cfin_{p,t} = 1$.

2.5 Vehicle state limits

Vehicles have a maximum velocity they can achieve. Calculating the velocity of the vehicle means calculating the 2-norm of the velocity vector. This is not possible using only linear equations. However, the maximum velocity can be approximated to an arbitrary degree using multiple linear constraints. For simplicity we will only cover the 2D case, although this can easily be extended to 3D as well. With x_i and y_i the N_{points} vertices of the approximating polygon listed in counter-clockwise order, the velocity can be limited to v_{max} with equation 15. Figure 2 shows a visual representation of this.

$$a_i = \frac{y_i - y_{i-1}}{x_i - x_{i-1}} \quad 0 \leq i < N_{points} \tag{13}$$

$$b_i = y_i - a_i x_i \quad 0 \leq i < N_{points} \tag{14}$$

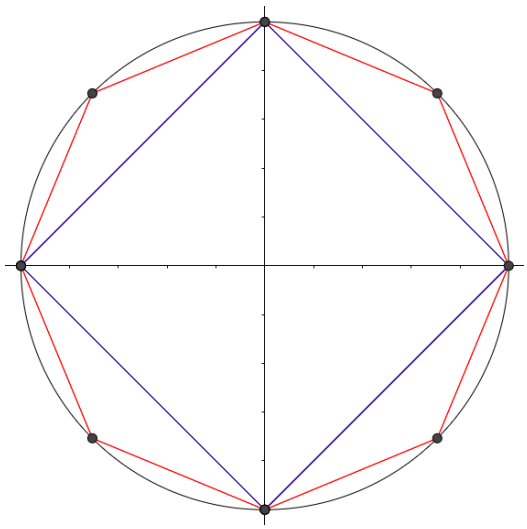


Figure 2: If the velocity is limited to a finite value, the velocity vector must lie within the circle centered on the origin with the radius equal to that value. This is represented by the black circle. This circle cannot be approximated in MILP, but it can be approximated using several linear constraints. 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.

$$v_{t,1} \leq a_i v_{t,0} + b_i \quad 0 \leq i < N_{points},\ 0 \leq t \leq N \tag{15}$$

The acceleration and other vector properties of the vehicle can be limited in the same way. This method can also be applied to keep the vehicle's position inside a convex polygon. The importance of this will become clear section 3.

2.6 Obstacle avoidance

TODO: FULLY EXPLAIN BIG M
The most challenging part of the problem is modeling obstacles. Any obstacle between the vehicle and its goal will inherently make the search space non-convex. Because of this, integer variables are needed to model obstacles. For each edge of the polygon obstacle, the line through that edge is constructed. If the obstacle is convex, the obstacle will be entirely on one side of that line. This means that the other side can be considered a safe area. However,

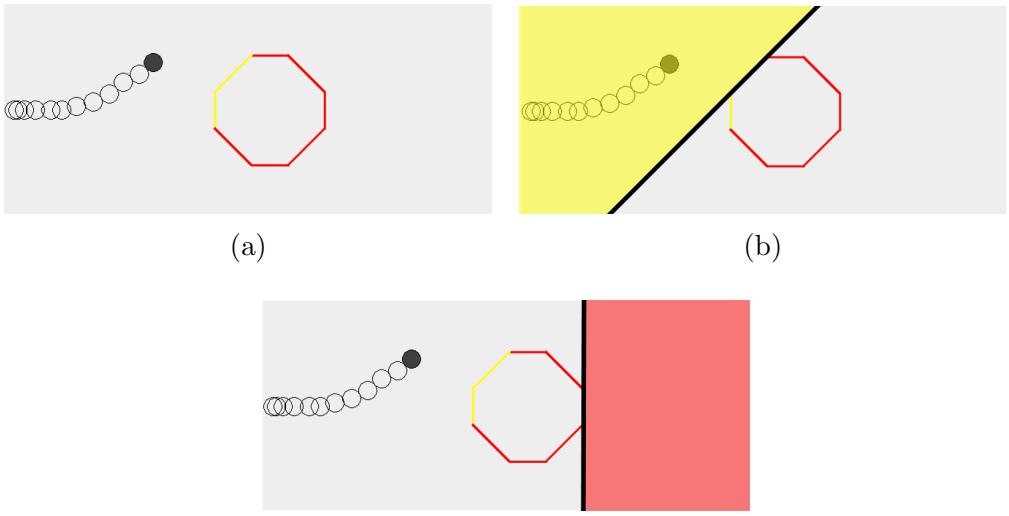


Figure 3: A visual representation of how obstacle avoidance works. The top image shows the vehicle's current position as the filled circle, with its path in previous time steps as hollow circles. The color of the edges of the obstacle represent whether or not the vehicle is in the safe zone for that 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.

the vehicle cannot be in the safe area of all edges at the same time, so a mechanism is needed to turn off these constraints when needed. As long as the vehicle is in the safe area of at least one edge, it cannot collide. Figure 3 demonstrates this visually.

The most common way to turn off individual constraints is using the "Big M" method, like Schouwenaars et al. [12] used in their work. However CPLEX supports a better method called "indicator constraints". Just like with the Big M method, it requires one boolean variable per edge. If the variable is true, the corresponding constraint is ignored. As long as at least one of those variables if false, a collision cannot happen. We will call these slack variables. For every convex obstacle o with vertices $x_{o,i}$ and $y_{o,i}$ with $a_{o,i}$ and

$b_{o,i}$ calculated as in equations 13 and 14:

$$dx_{o,i} = x_{o,i} - x_{o,i-1}, \quad dy_{o,i} = y_{o,i} - y_{o,i-1}$$

$$slack_{o,i,t} \Rightarrow \begin{cases} b_i + buf_{o,i} \leq p_{t,1} - a_i p_{t,0} & dx_{o,i} < 0 \\ b_i - buf_{o,i} \geq p_{t,1} - a_i p_{t,0} & dx_{o,i} > 0 \\ x_{o,i} + buf_{o,i} \leq p_{t,0} & dx_{o,i} = 0, dy_{o,i} > 0 \\ x_{o,i} - buf_{o,i} \geq p_{t,0} & dx_{o,i} = 0, dy_{o,i} < 0 \end{cases} \tag{16}$$

$$\neg \bigwedge_i slack_{o,i,t} \quad 0 \leq t \leq N \tag{17}$$

The occurrences of $buf_{o,i}$ in equation 16 are necessary because the vehicle is not a point. With $\alpha_{o,i}$ the angle perpendicular to edge i of obstacle o, and S the radius of the vehicle:

$$\alpha_{o,i} = tan^{-1}(-1/a_{o,i}) \tag{18}$$

$$buf_{o,i} = |\frac{S}{sin(\alpha_{o,i})}| \tag{19}$$

2.7 Terminology

TODO: FILL OUT variable constraint solution space feasible region solver (cplex) convex

2.8 Importance of integers and convexity

TODO: REWRITE linear programming one of the most simple forms of mathematical programming. Only linear functions allowed, all variables have a real domain. If feasible region exists: inherently convex. When a problem is convex, typically efficient solvers can be written (TODO: find source). This is the case for LP. LP problems can be solved reliably and quickly.
Problem: only real variables is very limiting. Logical operators are impossible to model. Result is that not every problem can be modeled without integers. Adding integers seems like a minor change, but it no longer guarantees that the feasible region is convex. When the problem is no longer convex, it becomes intractible. An intractible problem is a problem for which we do not have a strategy that will on average find the solution faster than random

guessing.
Path planning when obstacles are present inherently needs integer variables and thus is an intractible problem. The goal of this thesis is to make this approach to path planning scalable. This will rely on two concepts:// - The solvers are on average not faster than random, but that's the average of all problems. We are only concerned with a very specific kind of problem. By being aware of how the solver works and simply experimenting with different representations, the problem can be solved much faster
- We can cheat. The problem can be greatly simplified and approximated. This means that the solution is no longer guaranteed to be optimal, but as long as it is good enough this is not a problem.

2.9 Characteristics of the basic model

Nothing new in basic model, motivate changes based on flaws in the model. demonstration with simple scenario (benchmark and spiral) where basic model already struggles. Approximate amount of integer variables needed, use as representation of nonconvexity of problem.
Also show issue with time allowed: Need to allocate way more time than is needed, making the problem even harder to solve. Limiting time makes problem faster to solve, but cannot know without prior information.
Demonstrate issue with corners. Corners make execution time go up even as amount of integer variables stays the same. If no corners are needed, feasible space stays more convex even though integer values are same. Nuance that integer variables are only apprixmation of non-convexity, convexity is real property that matters.
Conclude: need to reduce amount of obstacles, need to reduce timesteps, limit amount of corners at a time (so problem becomes more convex). Dividing problem into smaller pieces tackles all these goals.

3 Segmentation of the MILP problem

3.1 Introduction

The MILP model described in section 2 is sufficient to solve the path planning problem for short flights with few obstacles. However, it scales poorly when the duration of the flight or the amount of obstacles is increased. Mixed-Integer programming belongs to the "NP-Complete" class of problems [**?**]. This is a class of problems which is considered very hard to solve. As the amount of integer variables grows, the time needed to solve the problem increases exponentially. An integer variable is needed for every edge of every polygon, for every time step. By reducing both the amount of time steps needed and obstacles that need to be modeled, the execution time can be reduced dramatically.
The key insight that allows my algorithm to scale well beyond what's usually possible is that the path does not need to be solved all at once. If the path can be split into many different segments, the amount of time steps per segments is a fraction of what is needed for the full path. Because each segment is relatively small, it is also possible to only model obstacles which are in the neighborhood of that segment. An obstacle that cannot possibly be reached in the amount of time steps for that segment does not need to be modeled.
While segmentation does make it much easier to solve the problem, it also has an important down side: Finding the shortest path can no longer be guaranteed. Segmentation makes it easier to find a solution, but fundamentally the problem of finding the optimal path is still just as hard. The necessary tradeoff for better performance is that the optimal path will likely not be found. Luckily, the optimal path is often not required in navigation. A reasonably good path will do.

3.1.1 The importance of convexity

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

problem form some geometric shape with a dimension for every variable.
When only linear constraints with real values are used, the solution space will always be convex. It is this convexity that makes a standard linear program easy to solve. When integer variables are introduced, it is possible to construct solution spaces which are not convex. As the solution space becomes less and less convex, the problem becomes harder to solve. Integer variables can be seen as a tool which allows non-convex solution spaces to be modeled. When trying to improve the difficulty of a problem, the actual goal is making the problem more convex (or smaller, which always helps). Reducing the amount of integer variables is only a side effect.
This insight allows for an intuitive understanding of why obstacles make the problem so difficult. Each obstacle "punches a hole" in the area where the vehicle is allowed to be as seen in Figure TODO:FIGURE, making the solution space less convex.

3.1.2 General Algorithm Outline

Algorithm 1 General outline

```
1: T ← {}                                   ▷ The list of trajectories solved so far
2: path ← Theta*(scenario)
3: events ← FindCornerEvents(path)
4: segments ← GenSegments(path, events)
5: for each segment ∈ segments do
6:     UpdateStartState(segment)
7:     GenActiveRegion(scenario, segment)
8:     T ← T ∪ { SolveTrajectory(segment) }
9: end for
10: result ←MergeTrajectories(T)
```

3.2 Finding the initial path

The first issue is that because the goal cannot be reached immediately, the goal function for the MILP problem needs to change.
One option is to simply get as close as possible to the goal during each segment. Because the distance that can be traveled during a segment is limited, the amount of obstacles that need to be modeled is also limited. This works

well when the world is very open with little obstacles. However, this greedy approach is prone to getting stuck in dead ends in more dense worlds like cities.
The second option is finding a complete path using a method that's easier to compute. An algorithm like A* can be used to find a rough estimation for the path. This A* path is the shortest path, but does not take constraints or the characteristics of the vehicle into account. A very curvy direct path may be the shortest, but a detour which is mostly straight and allows for higher speeds may actually be faster.
It is possible to use more advanced algorithms that model more of the constraints. This will significantly improve the execution time of the MILP solver and quality of the solution, but it will also come at a performance cost. A balance needs to be found between the execution time of the preprocessing step and that of the MILP solver. The preprocessing step needs to do just enough so the solver can find a good solution in an acceptable amount of time.
I have decided to use Theta*. This is a variant of A* that allows for paths at arbitrary angles instead of multiples of 45 degrees. The main reason for this is that it eliminates the "zig zags" that A* produces. This makes the next step of the preprocessing much easier. The left image in figure 4 shows the result of the Theta* algorithm.

3.2.1 Theta* implementation

3.2.2 Theta* optimizations

heuristic
indexed obstacles
possible position: first fuzzy collision, after: LOS check with last point
line of sight: first bounding box overlap, only after LOS

3.2.3 Scalability of Theta*

TODO: REWRITE... KEEP? Theta* provides the information that makes it possible to make MILP fast, but Theta* in itself is intractable. Why is

this not simply moving the issue?
Theta* is indeed intractable, but still much better than MILP. Holomomic vs non-holonomic. Concept of navigation mesh as part of map, often used in video games to allow AI to navigate large worlds. Also same concept behind PRM. Only needs to pre precomputed once per map. Can serve as heuristic for Theta*, or completely replace it.

3.3 Detecting corner events

With an initial path generated, the next problem is dividing it into segments. Dividing the path into equal parts presents problems, because when solving each segment, the solver has no knowledge of what will happen in the next segment. This is especially problematic when the vehicle needs to make a tight corner. If the last segment ends right before the corner, it may not be possible to avoid a collision. Because of this, corners need to be taken into account when generating the segments. The right image in figure 4 shows the transitions between segments as green circles.
In Euclidian geometry, the shortest path between two points is always a straight line. When polygonal obstacles are introduced between those points, the shortest path will be composed of straight lines with turns at one or more vertices of the obstacles. The obstacle that causes the turn will always be on the inside of the corner. This shows why corners are important for another reason: They make the search space non-convex. For obstacles on the outside of the corner it is possible to constrain the search space so it is still convex, but this is not possible for obstacles on the inside of a corner.
Because of these reasons, isolating the corners from the rest of the path is advantageous. With enough buffer before the corner, the vehicle is much more likely to be able to navigate the corner successfully. It also means that the computationally expensive parts of the path are as small as possible while still containing enough information for fast navigation through the corners.
The reason for using Theta* becomes clear now. Every single node in the path generated by the algorithm is guaranteed to be either the start, goal or near a corner. A corner can have more than one node, so nodes which turn in the same direction and are close to each other are grouped together and considered part of the same corner. For each corner, a corner event is generated.

Figure 4: 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.

3.3.1 Algorithm Implementation

code
parameter tolerance, link to acc distance

3.3.2 Tight Coupling with Theta*

This algorithm relies on assumptions from Theta*, so if algorithm changes, this will need to be updated.

3.4 Generating path segments

3.4.1 Segment Generation Algorithm

code
approachmargin
maxtime
step by step figures with explanation?

Algorithm 2 Generating the segments

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

Figure 5: 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 3

3.4.2 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
note: only when about to be solved!

3.5 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.

3.5.1 Implementation of the genetic algorithm

Alg. 3 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.

Algorithm 3 Genetic Algorithm

```
1: function GenActiveRegion(scenario, segment)
2:     pop ← SeedPopulation
3:     for i ← 0, N_gens do
4:         pop ← pop∪ Mutate(pop)
5:         Evaluate(pop)
6:         pop ← Select(pop)
7:     end for
8:     return BestIndividual(pop)
9: end function
10: function Mutate(pop)
11:     for each individual ∈ pop do
12:         add vertex with 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 Δ_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
```

library
tournament selection
offspring generation: mutate only
initial population
chromosome/gene description
fitness function
validation
parameters
obstacle selection based on path length?

4 Extensions

4.1 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.

4.2 Corner cutting

ref [**?**] implemented simple version TODO: implement more complex version compare no vs simple vs complex

4.3 Max time

limit execution time, allows for clean failure if it takes to long and backtrack.

4.4 Max delta

Allows for earlier return. Limited usefulness?

4.5 Abs speed

4.6 Max speed

4.7 Min speed

4.8 Backtracking

implement better!

5 Improving the MILP problem

Vehicles have a maximum velocity they can achieve. Calculating the velocity of the vehicle means applying Pythagoras' theorem on the axis of the velocity vector. This is not possible using only linear equations. However, it can be approximated to an arbitrary degree using multiple linear constraints. The components of the velocity vector can be positive or negative, but only the absolute value matters for the actual velocity.

ABS
MAXSPEED

Because obstacles make the problem non-convex and thus require integer constraints to model, the execution time scales very poorly with the amount of obstacles. This can be mitigated by only modeling a certain amount of obstacles relatively close to the vehicle, while limiting the vehicle to a convex region which does not overlap any of the ignored obstacles. Modeling this convex allowed region is very similar to modeling the obstacles, except that this time no integer variables are needed.

ACTIVE REGION

$$minimize \quad N - \sum_{t=0}^{t \leq N} fin_t$$

$$fin_0 = 1$$

$$fin_{t+1} = fin_t \vee cfin_{t+1}, \quad 0 \leq t < N$$

$$cfin_{pos,t} = \bigwedge_{i=0}^{i<Dim(\boldsymbol{pos}_t)} |pos_{t,i} - pos_{goal,i}| < pos_{tol}, \quad 0 \leq t \leq N$$

$$cfin_{vel,t} = \begin{cases} \bigwedge_{i=0}^{i<Dim(\boldsymbol{vel}_t)} |vel_{t,i} - vel_{goal,i}| < vel_{tol} & \text{if } \boldsymbol{vel}_{goal} \text{ exists, } 0 \leq t \leq N \\ true & \text{otherwise, } 0 \leq t \leq N \end{cases}$$

$$cfin_t = cfin_{pos,t} \bigwedge cfin_{vel,t} \quad 0 \leq t \leq N$$

6 Visualizing solution

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

7 Analysis and Results

7.1 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 6 shows these scenarios visually. Table 7 shows detailed information about the scenarios and execution times.

7.1.1 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.

7.1.2 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.

7.1.3 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 5, 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.

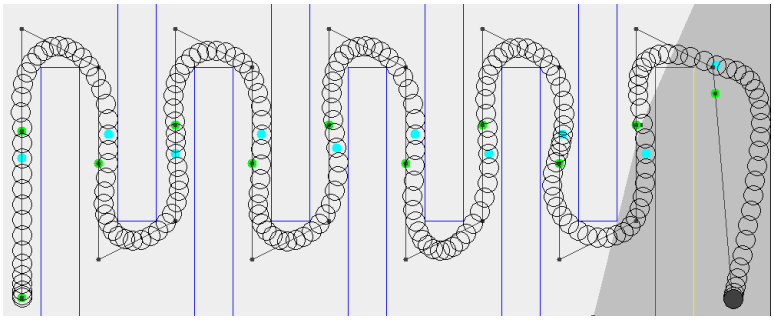
7.2 Theta* vs A*

A* is faster, Theta* is easier to post-process. Theta* grid simplified.
Theta* needs line of sight checks. Slower than A*, but also prevents corner cutting in pre-path
Hierarchic indexed data structure for obstacles can greatly cut down on amount of LOS checks needed
IMPLEMENT?

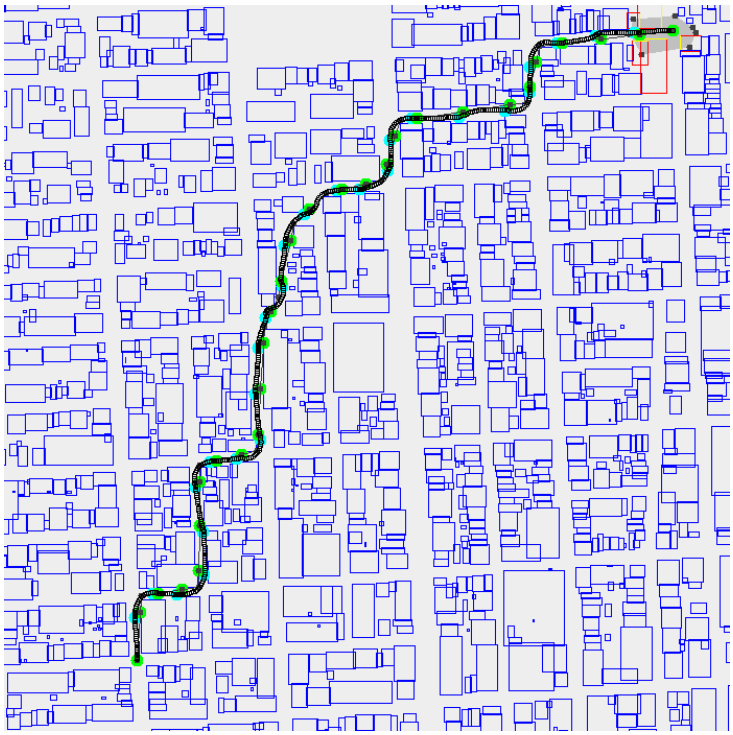
7.3 Genetic Algorithm

Genetic algorithm takes extra time, bounding box includes more obstacles. cant use convex hull because potentially too restrictive

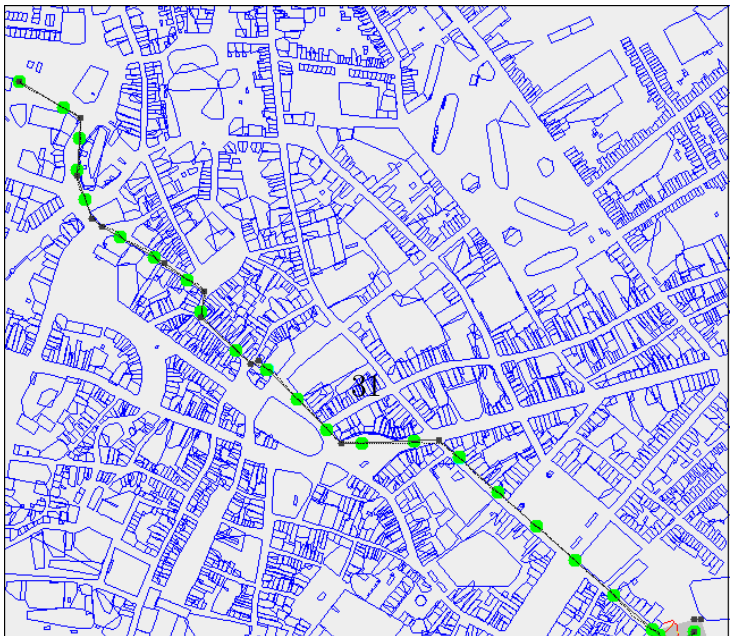
[1] `https://overheid.vlaanderen.be/producten-diensten/basiskaart-vlaanderen-grb`



(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 7: The experimental results for the different scenarios

check GA parameters

7.4 Curnercutting allowed vs not

compare 2 corner cutting mitigations, show impact on time, path length

7.5 Min/Max speed impact

abs speed impact when not used (none?)
max speed impact
min speed impact

7.6 Safety

guaranteed by constraints, however:
no incentive to stay away from walls. Can increase vehicle size, but prevents occasionally getting closer for a short while.

7.7 Stability and Performance

-maxtime (max length of segment)
- approach margin: larger -¿ better approach, longer solve time
- tolerance: larger -¿ when corners are close, merge faster: more execution time

- position tolerance: larger -¿ smoother path, strays further from preprocessed path so needs to backtrack more often
- backtracking effect?
-can quickly recalculate based on measured state? IMPLEMENT WARM START EXPERIMENT?

7.8 Other weaknesses

waviness
bad approaches to corners
transitions between segments

eigen abs:
slack -¿ abs = val
-slack -¿ abs = -val

–¿ 2 vars voor abs

abs norm:
speed ¿= dot product[i]
slack[i] -¿ speed = dot product[i]

–¿ speed: num points / 4 vars

norm:
slack[i] and absslackx -¿ circle x
slack[i] and - absslackx -¿ - circle x

–¿ norm: geen extra vars

8 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.

8.1 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.