Scalable UAV Trajectory Planning using Mixed Integer Linear Programming

Jorik De Waen*, Hoang Tung Dinh†, Mario Henrique Cruz Torres‡ and Tom Holvoet§
imec-DistriNet, University of Leuven, 3001 Leuven, Belgium
*jorik.dewaen@student.kuleuven.be, †hoangtung.dinh@cs.kuleuven.be,
‡mariohenrique.cruztorres@cs.kuleuven.be, §tom.holvoet@cs.kuleuven.be

***Abstract*—Trajectory planning using Mixed Integer Programming is currently severely limited by its poor scalability. This paper presents a new approach which improves the scalability with respect to the amount of obstacles and the distance between the start and goal positions. Where previous approaches hit computational limits when dealing with tens of obstacles, this new approach can handle tens of thousands of polygonal obstacles successfully on a typical consumer computer. This is achieved by dividing the trajectory into many smaller segments using multiple heuristics. Only obstacles in the local neighborhood of a segment are modeled, significantly reducing the complexity of the optimization problem. To demonstrate this approach can scale enough to be useful in real, complex environments, it has been tested on unprocessed maps of real cities with trajectoriess spanning several kilometers.**

I. Introduction

Trajectory planning for UAVs is a complex problem because flying is inherently a dynamic process. Proper modeling of the velocity and acceleration are required to generate a trajectory that is both fast and safe. A trajectory that is both fast and safe requires precise control of the trajectory to effectively navigate corners while maintaining momentum. The fastest trajectory is not always the shortest one, since the vehicle's velocity may be different. The vehicle dynamics are often not the only constraints placed on the trajectory. Different laws in different countries also affect the properties of the trajectory. The operators of the UAV may also wish to either prevent certain scenarios or ensure specific criteria are always met.
In this paper, we present a scalable approach which is capable of generating fast and safe trajectories, while also being easily extensible by design. The trajectory is represented with discrete time steps, where each step describes the vehicle's dynamic state at that moment. We used Mixed Integer Linear Programming (MILP), a form of mathematical programming, to achieve this goal. In mathematical programming, the problem is modeled using mathematical equations as constraints. An objective function encodes one or more properties, like time or trajectory length, to be optimized. A general solver is then used to find the optimal solution for the problem. Because the problem is defined declaratively, additional constraints can easily be added.

Other papers have used MILP for trajectory planning [1], but scalability limitations meant that it could not be used to generate long trajectories through complex environments. This paper's main contribution is a preprocessing pipeline which makes MILP trajectory planning more scalable.. Our strategy for making this approach more scalable revolves around dividing the long trajectory into many smaller segments. Several steps of preprocessing collect information about the trajectory. This information is used to generate the smaller segments, as well as reduce the difficulty of each specific segment.

The first step consists of finding an initial path using the Theta* algorithm. This trajectory does not take dynamic properties into account, making it faster to calculate. In the second step, the corners in this initial path are extracted and used to generate the segments. We define a corner to be a distinct change in the path's direction, with that change in direction being necessary because at least one obstacle is in-between the position where the turn starts and the goal position. The third step attempts to minimize the amount of obstacles that need to be modeled in each segment. A heuristic selects important obstacles for each segment. An obstacle is important if its absence could have a large impact on the trajectory. These obstacles are considered active and will be fully modeled in the MILP problem, ensuring that the vehicle will not collide with an active obstacle.
To ensure that the trajectory does not intersect with the inactive obstacles, a safe area is constructed by a genetic algorithm. This safe area is constructed such that it does not contain inactive obstacles and is convex. The vehicle is constrained stay within the safe area at all times in the MILP problem. To avoid restricting the movement of the vehicle unnecessarily, the genetic algorithm attempts to maximize the size of the safe area.

Schouwenaars et al. [1] were the first to demonstrate that MILP could be applied to trajectory 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. Obstacles are modeled as grid-aligned rectangles. 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 presented a receding horizon technique so the problem can be solved in multiple steps. However, this technique is essentially blind and could

easily get stuck behind obstacles. Bellingham[2] recognized that issue and proposed a method to prevent the trajectory from get stuck behind obstacles, even when using a receding horizon. However Bellingham's approach still scales poorly in environments with many obstacles.

Flores[3] and Deits et al[4] 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 safe regions to solve the scalability issues when faced with many obstacles.

Several papers [5], [6], [7], [8] show how the full state of a quadrocopter, including motor thrust, can be determined using only the 3D position of the vehicle's center of mass, the yaw and their derivatives. This demonstrates that, when the properties of a vehicle are accurately modeled, trajectory planners like the one in this paper 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 [9] provides an online approach. However, their approach can only find a suitable trajectory for the next few seconds of flight and updates that trajectory constantly.

More work has been done on modeling specific kinds of constraints or goal functions. For instance, Chaudhry et al. [10] 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. [4] could work, but did not explore the effects of longer trajectories on their algorithm.

II. Modeling Path Planning as a MILP problem

This section covers how a trajectory planning problem can be represented as a mixed integer linear program. The problem representation is based on the work by Schouwenaars et al. [1].

A. 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. A problem typically also contains a number linear inequalities which constrain the values of the variables in the objective function. When some (or all) of the variables are integers, the problem is a mixed integer problem.
More complex mathematical relations can be modeled by combining multiple constraints. Some of those relations, like logical operators or the absolute value function, can only be expressed when integer variables are allowed [11].

B. Time and vehicle state

The trajectory 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. There are $N+1$ time steps, with a fixed amount of time Δt between them.

$$\boldsymbol{p}_0 = \boldsymbol{p}_{start} \tag{1}$$

$$\boldsymbol{p}_{n+1} = \boldsymbol{p}_n + \Delta t * \boldsymbol{v}_n \quad 0 \leq n < N \tag{2}$$

Equation 1 and 2 model the position of the vehicle at each time step. For each time step t, the position in the next time step p_{n+1} is determined by the current position p_n, the current velocity v_n and the duration of the time step Δt. Velocity, acceleration and other derivatives are represented the same way. How many derivatives need to be modeled will vary depending on the specific use case.

C. Objective function

The objective is to minimize the time before a goal position is reached.

$$minimize \quad N - \sum_{n=0}^{n \leq N} done_n \tag{3}$$

Equation 3 shows the objective function. Reaching the goal causes a state transition from not being done to being done. This is represented as the value of $done_n$, which is a binary variable (which is an integer variable which can only be 0 or 1). When $done_n$ is true (equal to 1), the vehicle has reached its goal on or before time step n.

$$done_0 = 0 \tag{4}$$

$$done_N = 1 \tag{5}$$

$$done_{n+1} = done_n \vee cdone_{n+1}, \quad 0 \leq n < N \tag{6}$$

Equation 4 initializes $done_0$ to be false, ensuring that the initial state is not finished. Equation 5 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.
Lamport's [12] state transition axiom method was used to model state transitions. In equation 6, the state will be done at time step $n+1$ if the state is done at time step n or if there is a state transition from not done to done at time step $n+1$, represented by $cdone_{n+1}$.

$$cdone_n = cdone_{p,t} \wedge \neg done_n \quad 0 \leq n \leq N \tag{7}$$

$cdone_n$ in Equation 7 is true at time step n if the goal position requirement $cdone_{p,n}$ is true and the done state has not already been reached ($\neg done_n$). Additional constraints on the state transition, like a maximum velocity, can be added here.

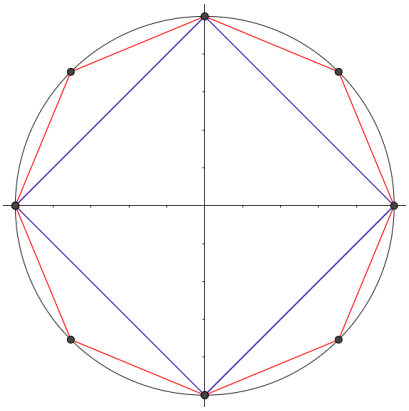

Fig. 1. If the velocity is limited to a finite value, the velocity vector must lie within the circle centered on the origin with the radius equal to that value. This is represented by the black circle. This circle cannot be approximated in MILP, but it can be approximated using several linear constraints.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

$$cdone_{p,n} = \bigwedge_{i=0}^{i<Dim(\boldsymbol{p}_n)} |p_{n,i} - p_{goal,i}| < \epsilon_p, \quad 0 \leq n \leq N \tag{8}$$

The goal position requirement is represented by $cdone_{p,n}$ and is satisfied when $cdone_{p,n} = 1$. The coordinate in dimension i of the position vector $\boldsymbol{p}_n$, is $p_{n,i}$. 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 n, $cdone_{p,n} = 1$.

D. Vehicle state limits

Vehicles have a maximum velocity. 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 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 11. Figure 1 shows a visual representation of these constraints.

$$a_i = \frac{y_i - y_{i-1}}{x_i - x_{i-1}} \quad 0 \leq i < N_{points} \tag{9}$$

$$b_i = y_i - a_i x_i \quad 0 \leq i < N_{points} \tag{10}$$

$$v_{n,1} \leq a_i v_{n,0} + b_i \quad 0 \leq i < N_{points},\ 0 \leq n \leq N \tag{11}$$

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.

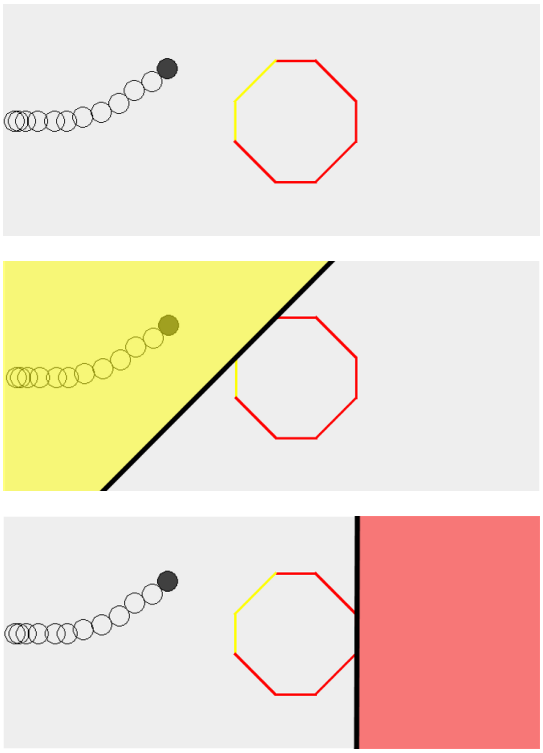

Fig. 2. A visual representation of how obstacle avoidance works. The top image shows the vehicle's current position as the filled circle, with its trajectory in previous time steps as hollow circles. The color of the edges of the obstacle represent whether or not the vehicle is in the safe zone for that 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.

E. Obstacle avoidance

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, 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 2 demonstrates this visually.

The most common way to turn off individual constraints is using the "Big M" method, like Schouwenaars et al. [1] 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 9 and 10:

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

$$slack_{o,i,t} \Rightarrow \begin{cases} b_i + offset_{o,i} \leq p_{t,1} - a_i p_{t,0} & dx_{o,i} < 0 \\ b_i - offset_{o,i} \geq p_{t,1} - a_i p_{t,0} & dx_{o,i} > 0 \\ x_{o,i} + offset_{o,i} \leq p_{t,0} & dx_{o,i} = 0, \; dy_{o,i} > 0 \\ x_{o,i} - offset_{o,i} \geq p_{t,0} & dx_{o,i} = 0, \; dy_{o,i} < 0 \end{cases} \tag{12}$$

$$\neg \bigwedge_i slack_{o,i,n} \quad 0 \leq n \leq N \tag{13}$$

The occurrences of $offset_{o,i}$ in equation 12 are necessary because the vehicle is not a point. The position variables represent the position of the center of mass of the vehicle. With the shape of the vehicle approximated as a circle of radius S and $\alpha_{o,i}$ the angle perpendicular to edge i of obstacle o:

$$\alpha_{o,i} = tan^{-1}(-1/a_{o,i}) \tag{14}$$

$$offset_{o,i} = |\frac{S}{sin(\alpha_{o,i})}| \tag{15}$$

Modeling obstacles this way is problematic because an integer variable is needed for every edge of every obstacle, for every time step. Each integer variable makes the solution space less convex, increasing the worst case execution time exponentially [**?**].

III. Segmentation of the MILP problem

In this section we propose a preprocessing pipeline that makes the problem more scalable. The first step is finding an initial path with the Theta* algorithm. Note how we call this a path and not a trajectory. Unlike a trajectory, a path is not time-dependent and does not take dynamic properties into account. The second step is finding the corners in that path. The third step is generating segments based on those corners. Finally, the active obstacles to be modeled in the MILP problem are selected while the others are approximated with a genetic algorithm.

The goal is to segment the problem in such a way that only a minimal amount of obstacles need to be modeled in each segment, while still resulting in a relatively fast trajectory. Shorter segments with fewer obstacles are easier to solve, but the vehicle will need to travel at a lower velocity. This is because there is no information available about the next segment. If the next segment contains a tight corner, the vehicle may not be able to slow down enough if it is going too fast. Longer segments allow the vehicle to travel faster, but they need more time to solve.

For the best results, we want to find segments which are as large as possible but contain as few obstacles as possible. By generating a segment for each corner in the path, we can make the segments just large enough so the vehicle can always slow down in time to navigate the corner. This way the vehicle will always navigate corners efficiently, without making the segments too large to solve in an acceptable amount of time.

A. Finding the initial path

Because the trajectory is divided into segments, the vehicle cannot reach the goal immdiately and needs to be guided towards the goal.

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 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 using an approximate path as a heuristic using an algorithm like A*. 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.

As always with a heuristic, there needs to be a balance between the quality and the execution time of the heuristic. A more advanced path planning algorithm could be used as the heuristic, but that will inherently also increase the execution time of the heuristic. We 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 artificial "zig zags" that A* produces. The left image in figure 3 shows the result of the Theta* algorithm.

B. Detecting corner events

With an initial path generated, the next problem is dividing it into segments. When solving each segment, the solver has no knowledge of what will happen in the next segment. This is can cause issues 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 3 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. These obstacles on the inside of the corner 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.

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

Fig. 3. The left image shows the results after the Theta* algorithm has executed. The blue shapes are obstacles, while the gray line is the Theta* path. The right image shows the results after the path is segmented. Extra nodes have been added to the Theta* path, as marked by the green circles. These circles depict the transitions between segments.

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.

C. Generating path segments

The corner events are in turn grown outwards to cover the approach and departure from the corner. The distance by which they are grown depends on the maximum acceleration of the vehicle: If the vehicle can come to a complete stop from its maximum speed before the corner, it can also successfully navigate that corner. When corners appear in quick succession, their expanded regions may overlap. In that case, the middle between those corners is chosen. Long, straight sections are also divided into smaller path segments.
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.

D. Generating the active region for each segment

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. 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 last 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. Tournament selection is used as the selector, with the fitness function being the surface area of the polygon. No crossover operator is used. Instead, the individuals are mutated to produce a single individual as offspring each. Both the parents and offspring compete together for survival.
When an individual is mutated, vertices can be added, removed or nudged. The nudge mutator moves a vertex of the polygon randomly by a small amount and checks if the resulting polygon is legal. If the new polygon is legal, the mutation is kept. Otherwise the mutator tries again until the maximum amount of attempts is reached, producing no offspring.
The genetic algorithm is just one way to generate the convex polygon which represents the active region. Deits and Tedrake [4] have demonstrated how another algorithm can solve the same problem. The left image of figure 4 show the active obstacles in red, and the convex polygon generated by the genetic algorithm in dark gray.

IV. Results

We test out algorithm in several different scenarios. Each scenario was tested with two different problem sizes. Theta* is 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 5 shows these scenarios visually. Table 6

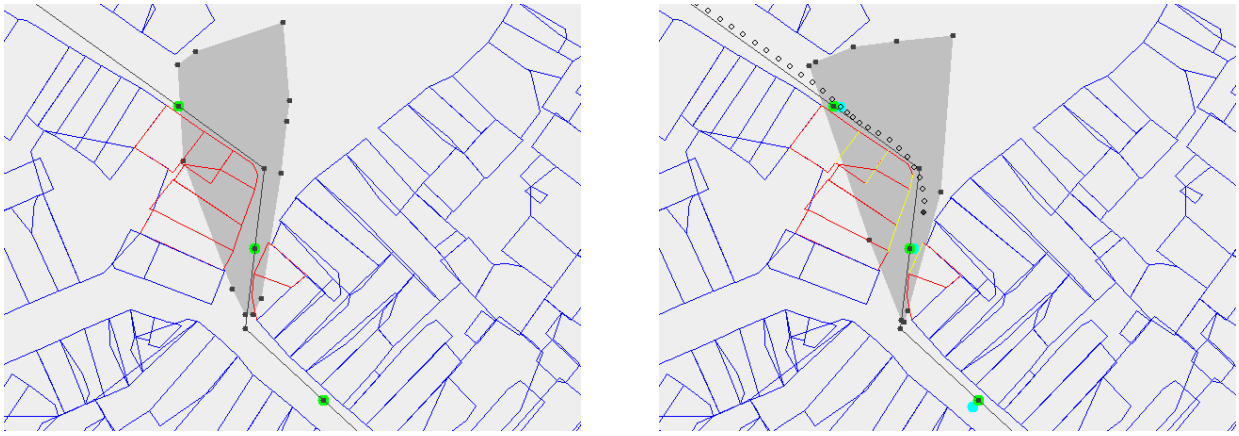

Fig. 4. The left image shows the result after the genetic algorithm has executed. The obstacles in red have been selected to be modeled in the MILP problem. The dark grey shape is the convex allowed region generated by the genetic algortihm. Note how it does not overlap with any of the blue obstacles. The right image shows the final result. The trail of circles show the path of the vehicle up to the current time step, which is represented by the filled circles. The red and yellow colors depict the same information as in figure 2

shows detailed information about the scenarios and execution times.

A. 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.

B. 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.

C. 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 compared to the San Francisco dataset, but still runs in an acceptable amount of time. As visible in figure 4, 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

V. Conclusion

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.

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.

scenario	# obstacles	world size	path length	# segments	Theta* time	Gen. Al. time	MILP time	score
Up/Down Small	5	25m x 20m	88m	7	0.09s	1.10s	20.8s	26.6s
Up/Down Large	9	40m x 20m	146m	11	0.14s	1.62s	40.1s	43.6s
SF Small	684	1km x 1km	1392m	28	2.04s	9.56s	59.2s	105.7s
SF Large	6580	3km x 3km	4325m	84	18.14s	18.21s	231s	316.0s
Leuven Small	3079	1km x 1km	1312m	29	2.29s	29.83s	152s	95.9s
Leuven Large	18876	3km x 3km	3041m	61	18.14s	83.69s	687s	217.6s

Fig. 6. The experimental results for the different scenarios

Fig. 5. These are the three different worlds which were tested. The top image shows the large Up/Down scenario. The middle image shows the small San Francisco scenario. Note how the obstacles are only grid-aligned rectangles laid out in a grid pattern. The bottom image shows the small Leuven scenario. The obstacles are polygons and distributed in a much more irregular pattern compared to the San Francisco scenario.

REFERENCES

[1] T. Schouwenaars, B. De Moor, E. Feron, and J. How, "Mixed integer programming for multi-vehicle path planning," in *Control Conference (ECC), 2001 European*, pp. 2603–2608, IEEE, 2001.

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

[3] M. 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.

[4] R. Deits and R. Tedrake, "Efficient mixed-integer planning for uavs in cluttered environments," in *Robotics and Automation (ICRA), 2015 IEEE International Conference on*, pp. 42–49, IEEE, 2015.

[5] M. Fliess, J. Lévine, P. Martin, and P. Rouchon, "Design of trajectory stabilizing feedback for driftless at systems," *Proceedings of the Third ECC, Rome*, pp. 1882–1887, 1995.

[6] Y. Hao, A. Davari, and A. 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*, pp. 104–109, IEEE, 2005.

[7] I. D. Cowling, O. A. Yakimenko, J. F. Whidborne, and A. K. Cooke, "A prototype of an autonomous controller for a quadrotor uav," in *Control Conference (ECC), 2007 European*, pp. 4001–4008, IEEE, 2007.

[8] D. Mellinger and V. Kumar, "Minimum snap trajectory generation and control for quadrotors," in *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pp. 2520–2525, IEEE, 2011.

[9] K. F. Culligan, *Online trajectory planning for uavs using mixed integer linear programming*. PhD thesis, Massachusetts Institute of Technology, 2006.

[10] A. Chaudhry, K. Misovec, and R. 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*, vol. 4, pp. 3823–3829, IEEE, 2004.

[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*, vol. 72, no. 2, pp. 262–276, 1994.

[12] L. Lamport, "A simple approach to specifying concurrent systems," *Communications of the ACM*, vol. 32, no. 1, pp. 32–45, 1989.