Scalable Path Planning for UAVs using Mixed Integer Linear Programming

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

***Abstract*—Path planning for UAVs using Mixed Integer Programming is a promising technique, but 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 first calculating a rough path with the Theta* algorithm. Based on this rough path, the problem is split into many smaller segments. A heuristic selects a small set of nearby obstacles to be fully modeled in the MIP problem. A genetic algorithm is used to construct a convex area around the segment to approximate all the obstacles which have not been selected by the heuristic. This ensures only relevant obstacles are modeled while still preventing collisions with any obstacle. 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 paths spanning several kilometers.**

I. Introduction

Path 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 path that is both fast and safe. A path that is both safe and fast requires precise control of the trajectory to effectively navigate corners while maintaining momentum. The vehicle dynamics are often not the only constraints placed on the path. Different laws in different countries also affect the properties of the path. 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 safe and fast paths, while also being easily extensible by design. 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. A goal function encodes one or more properties, like time or path length, to be optimized. A general solver is then used to find the optimal solution for the problem. Because the problem is defined using mathematical equations to be solved by a general solver, additional constraints can easily be added.

However, the focus of this paper is not on the extensibility aspect. The idea of using MILP for path planning is not new [1], but scalability limitations meant that it could not be used to generate long paths through complex environments. Our strategy for making this approach more scalable revolves around dividing the long path into many smaller segments. Several steps of preprocessing collect information about the path. 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 rough path using the Theta* algorithm. This path does not take any dynamic properties into account, making it much easier to calculate. In the second step, the corners in this rough path are extracted and used to generate the segments. The third step attempts to minimize the amount of obstacles that need to be modeled in each segment. A heuristic selects specific obstacles to model during each segment. To ensure that the path does not intersect with any of the other obstacles, a convex area is generated by a genetic algorithm. The UAV is constrained stay within the convex area at all times in the MILP problem. The genetic algorithm grows that area as much as possible, while not overlapping with any of the obstacles which have not been selected by the heuristic (as they will modeled separately in the MILP problem).

A. Related work

Schouwenaars et al. [1] 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[2] recognized that issue and proposed a method to prevent the path from get stuck behind obstacles, even when using a receding horizon.

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 regions to limit (or in their work: completely eliminate) the need to model obstacles directly.

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

Fig. 1. A typical linear problem

Several different papers [5], [6], [7], [8] 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 [9] 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. [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 really explore the effects of longer paths on their algorithm.

II. Modeling Path Planning as a MILP problem

A. Introduction

This section covers how a path planning problem can be represented as a mixed integer linear program. As mentioned before, the way the problem is represented is based on the work by Schouwenaars et al. [1].

B. 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. 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].

C. 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.

D. 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.

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

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

$$fin_N = 1 \quad (9)$$

$$fin_{t+1} = fin_t \vee cfin_{t+1}, \quad 0 \le t < N \quad (10)$$

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 [12] 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 $cfin_{t+1}$.

$$cfin_t = cfin_{p,t} \wedge \neg fin_t \quad 0 \le t \le N \quad (11)$$

$cfin_t$ 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.

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

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, $cfin_{p,t} = 1$.

E. 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.

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

$$b_i = y_i - a_i x_i \quad 0 \le i < N_{points} \quad (14)$$

$$v_{t,1} \le a_i v_{t,0} + b_i \quad 0 \le i < N_{points}, \ 0 \le t \le N \quad (15)$$

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 III.

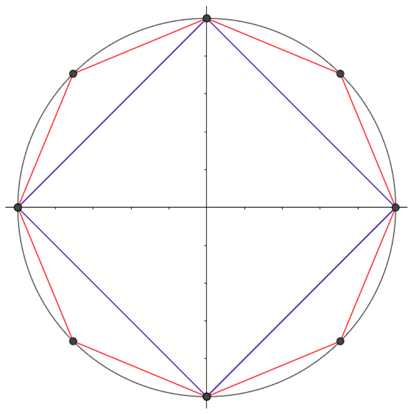

Fig. 2. If the velocity is limited to a finite value, the velocity vector must lie within the circle centered on the origin with the radius equal to that value. This is represented by the black circle. This circle cannot be approximated in MILP, but it can be approximated using several linear constraints.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

F. 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 3 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 13 and 14:

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

$$slack_{o,i,t} \Rightarrow \begin{cases} b_i + buf_{o,i} \le p_{t,1} - a_i p_{t,0} & dx_{o,i} < 0 \\ b_i - buf_{o,i} \ge p_{t,1} - a_i p_{t,0} & dx_{o,i} > 0 \\ x_{o,i} + buf_{o,i} \le p_{t,0} & dx_{o,i} = 0, dy_{o,i} > 0 \\ x_{o,i} - buf_{o,i} \ge p_{t,0} & dx_{o,i} = 0, dy_{o,i} < 0 \end{cases} \quad (16)$$

$$\neg \bigwedge_i slack_{o,i,t} \quad 0 \le t \le N \quad (17)$$

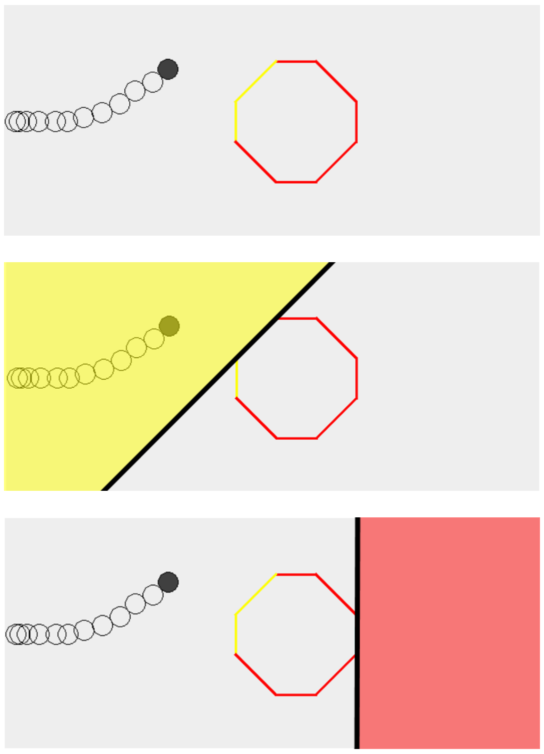

Fig. 3. A visual representation of how obstacle avoidance works. The top image shows the vehicle's current position as the filled circle, with its path in previous time steps as hollow circles. The color of the edges of the obstacle represent whether or not the vehicle is in the safe zone for that edge.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

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

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

III. Segmentation of the MILP problem

A. Introduction

The model described in the previous section is enough to find a path, but the solver will take a very long time when the length of the path and the amount of obstacles are increased. This section presents the preprocessing pipeline that makes the problem more scalable. The first step is finding an initial path with the Theta* algorithm. The second step is finding the corners in that path. The third step is generating segments based on those corners. Finally, the 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 path. As the segments get smaller, fewer obstacles need to be modeled. However, the path will also become slower. By increasing the size of the segment, the path will get closer to the optimal path, but at a performance penalty because much more obstacles need to be modeled.
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.

B. 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.

C. 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,

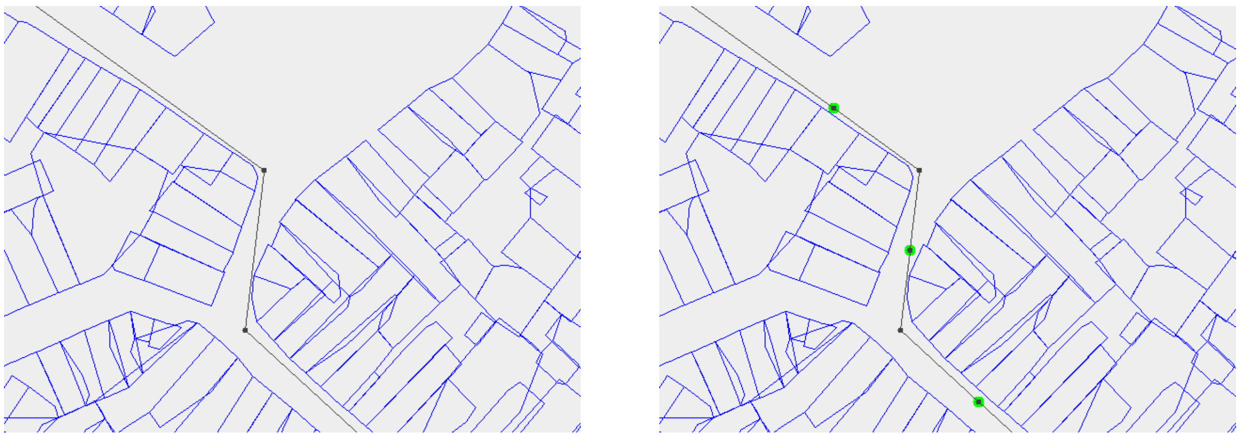

Fig. 4. The left image shows the results after the Theta* algorithm has executed. The blue shapes are obstacles, while the gray line is the Theta* path. The right image shows the results after the path is segmented. Extra nodes have been added to the Theta* path, as marked by the green circles. These circles depict the transitions between segments.

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.

D. Generating path segments

These corner events are in turn grown outwards to cover the approach and departure from the corner. How much depends on the maximum acceleration of the vehicle. As a rule of thumb: 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. This means that every segment has a set of 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.

E. Generating the active region for each segment

Even though the most important obstacles are taken care of, all other obstacles also need to be represented. To do this, a convex polygon is grown 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. It uses a single mutator which nudges the vertices of the polygon while ensuring it stays convex and does not intersect itself or any non-active obstacle. 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 5 show the active obstacles in red, and the convex polygon generated by the genetic algorithm in dark gray.

IV. Results

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

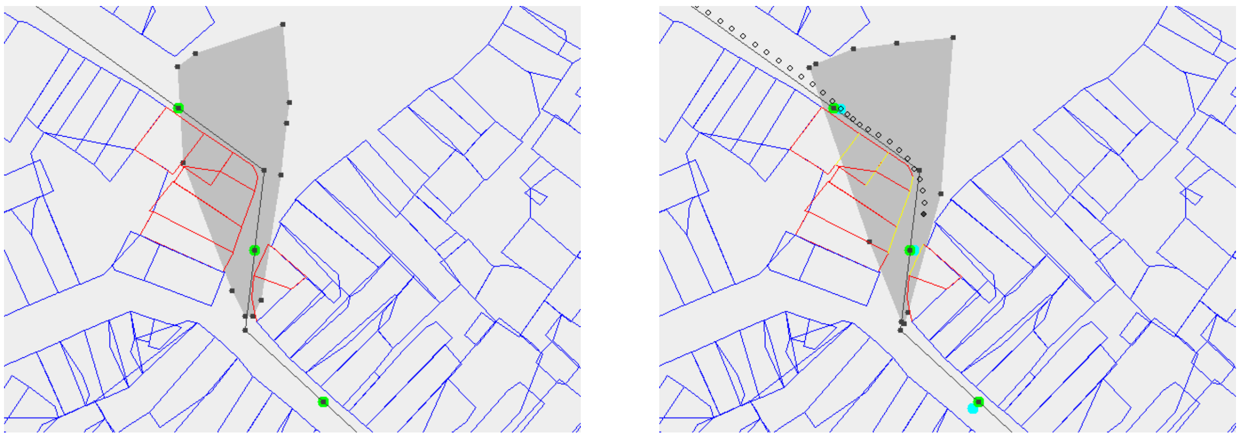

Fig. 5. The left image shows the result after the genetic algorithm has executed. The obstacles in red have been selected to be modeled in the MILP problem. The dark grey shape is the convex allowed region generated by the genetic algortihm. Note how it does not overlap with any of the blue obstacles. 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

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.

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. With these constraints, parameters and hardware, the path can be planned faster than the vehicle can execute it.

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, 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.

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.

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

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. 7. The experimental results for the different scenarios

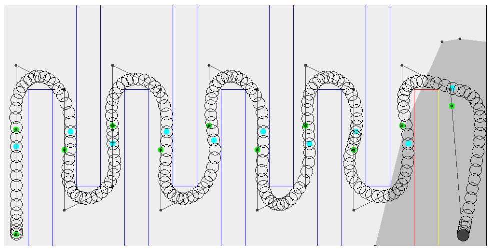
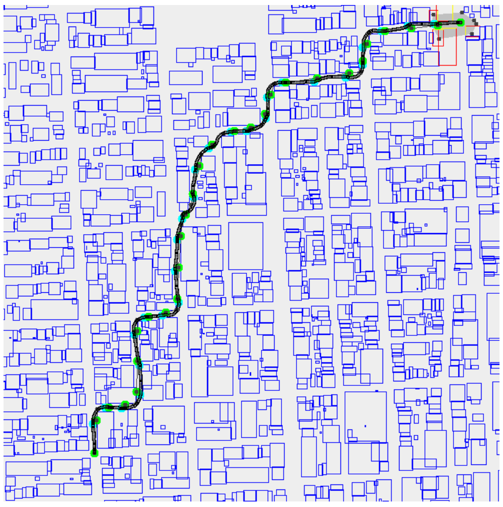
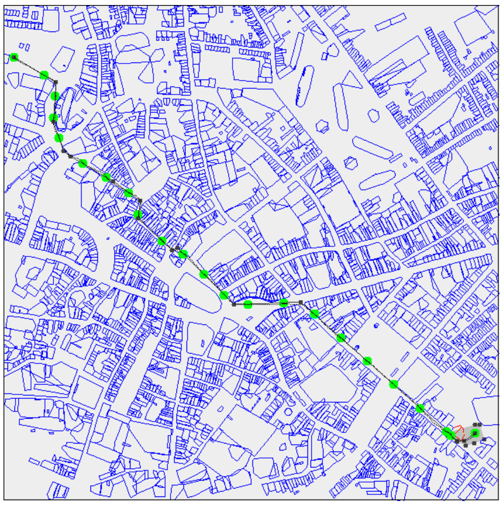

Fig. 6. These are the three different worlds which were tested. The top image shows the large Up/Down scenario. The middle image shows the small San Francisco scenario. Note how the obstacles are only grid-aligned rectangles laid out in a grid pattern. The bottom image shows the small Leuven scenario. The obstacles are polygons and distributed in a much more irregular pattern compared to the San Francisco scenario.

A. *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] 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.