Scalable Multirotor UAV Trajectory Planning using Mixed Integer Linear Programming

Jorik De Waen[1], Hoang Tung Dinh[2], Mario Henrique Cruz Torres[2] and Tom Holvoet[2]

***Abstract*— Trajectory planning using Mixed Integer Linear Programming (MILP) is a powerful approach because vehicle dynamics and other constraints can be taken into account. However, it is currently severely limited by poor scalability. This paper presents a new approach which improves the scalability regarding the amount of obstacles and the distance between the start and goal positions. While previous approaches hit computational limits when the problem contains tens of obstacles, our approach can handle tens of thousands of polygonal obstacles successfully on a typical consumer computer. This performance is achieved by dividing the problem into many smaller MILP subproblems using two sets of heuristics. Each subproblem models a small part of the trajectory. The subproblems are solved in sequence, gradually building the desired trajectory. The first set of heuristics generate each subproblem in a way that minimizes its difficulty, while preserving stability. The second set of heuristics select a limited amount obstacles to be modeled in each subproblem, while preserving consistency. To demonstrate that this approach can scale enough to be useful in real, complex environments, it has been tested on maps of two cities with trajectories spanning over several kilometers.**

I. Introduction

Trajectory planning for multirotor UAVs is a complex problem because flying is inherently a dynamic process. Proper modeling of velocity and acceleration are required to generate a feasible trajectory that is both fast and safe, that is, the UAV should be able to effectively navigate corners while maintaining momentum. The fastest trajectory is not always the shortest one, since the UAV's velocity may be different. The UAV 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 that 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. We model the trajectory planning problem as a Mixed Integer Linear Program (MILP). The trajectory is represented in discrete time steps where each step describes the UAV's dynamic state at that moment. 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.

[1]Jorik De Waen is a student at KU Leuven, 3001 Leuven, Belgium jorik.dewaen@student.kuleuven.be

[2]Hoang Tung Dinh, Mario Henrique Cruz Torres and Tom Holvoet are with imec-DistriNet, KU Leuven, 3001 Leuven, Belgium {hoangtung.dinh, mariohenrique.cruztorres, tom.holvoet}@cs.kuleuven.be

We demonstrate our approach in 2D environments. We assume that all obstacles are polygons, static and known in advance. Our algorithm is designed for offline planning, ensuring that a feasible trajectory exists before the UAV starts executing its task. Other papers have used MILP for trajectory planning [1], their approaches could not be used to generate long trajectories through complex environments. Our main contribution is an approach which improves the scalability by dividing the problem into many MILP subproblems. Each subproblem models only a part of the trajectory. The subproblems are solved sequentially. A first set of heuristics uses a Theta* path to generate the subproblems. The second set of heuristics select which obstacles should be modeled in each subproblem, limiting the amount of obstacles that need to be modeled while ensuring that no collisions can occur.

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. Obstacles are modeled as grid-aligned rectangles. To limit the computational 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 getting stuck behind obstacles, even when using a receding horizon. However Bellingham's approach still scales poorly in environments with many obstacles. The basic formulations of constraints we present in this paper are extensions of the work by Bellingham.

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 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.

II. Modeling Path Planning as a MILP problem

This section covers how a trajectory planning problem can be represented as a MILP problem. The problem representation is based on the work by Bellingham[2]. MILP is an extension of linear programming. In a linear program, decision variables are unknown real numbers and the goal is to maximize or minimize a linear objective function subjected to a number of linear equality or inequality constraints. In MILP, some or all decision variables can be integers, which allows more flexibility in modeling at computational cost.

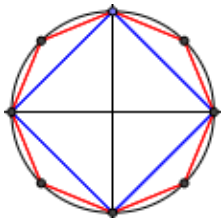


Fig. 1: If the velocity is limited to a finite value, the velocity vector must lie within the circle centered on the origin with the radius equal to that value. This is represented by the black circle. The blue and red polygons show the linear approximation using 4 and 8 constraints respectively.

A. Time and UAV state

The trajectory planning problem can be represented with N discrete time steps and a set of state variables at each time step [2]. The number of time steps determines the maximum amount of time the UAV has to reach its goal.

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

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

Eq. (1) and (2) represent the state of the UAV at each time step. For each time step n, 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 time step size Δt. Velocity, acceleration and other derivatives are represented the same way. The number of derivatives needed depends on the specific use case.

B. Objective function

The objective is to minimize the time before the UAV reaches the goal position.

$$\text{minimize} \quad -\sum_{n=0}^{n \leq N-1} done_n \tag{3}$$

Eq. (3) shows the objective function [2]. Reaching the goal causes a state transition from not being done to being done. This is represented as the value of binary variable $done_n$. When $done_n = 1$, the UAV has reached its goal on or before time step n.

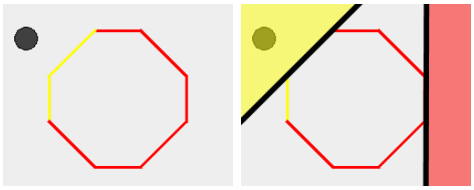
$$done_0 = 0, \quad done_{N-1} = 1 \tag{4}$$

$$done_{n+1} = done_n \vee cdone_{n+1}, \quad 0 \leq n < N-1 \tag{5}$$

Eq. (4) states that the UAV must reach its goal eventually. Lamport's state transition axiom method [5] was used to model state transitions. In Eq. (5), 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 $cdone_{n+1}$.

$$cdone_n = |x_n - x_{goal}| < \epsilon_p \wedge |y_n - y_{goal}| < \epsilon_p, 0 \leq n < N \tag{6}$$

The goal position requirement is fulfilled if the UAV is closer than ϵ_p to the goal position in both dimensions [2].



(a) (b)

Fig. 2: A visual representation of how obstacle avoidance works. Fig. 2a shows the UAV's current position as the black circle. The color of the edges of the obstacle represent whether or not the UAV is in the safe zone for that edge. An edge is yellow if the UAV is in the safe zone, and red otherwise. Fig. 2b shows the safe zones defined by a yellow and red edge in yellow and red respectively.

C. UAV state limits

Modeling the maximum velocity of a UAV requires calculating the the velocity vector's 2-norm which is non-linear. However, the maximum velocity constraint can be approximated to an arbitrary degree using multiple linear constraints [2]. Fig. 1 shows a visual representation of this approximation. The acceleration and other vector properties of the UAV can be limited in the same way.

D. Obstacle avoidance

The most challenging part of the problem is modeling obstacles. Any obstacle between the UAV and its goal will inherently make the search space non-convex. Because of this, integer variables are needed to model obstacles. Assuming that obstacles are convex polygons, for each obstacle, the UAV needs to be on the "safe" side of at least one edge to not collide with the obstacle (see Fig. 2). Indicator constraints were used to model obstacle avoidance. This requires one boolean variable $slack$ per edge. If $slack$ is false, the UAV is on the safe side of the edge. For each obstacle at least one of the $slack$ variables need to be false. For every convex obstacle o with the coordinates of vertex i being $x_{o,i}$ and $y_{o,i}$ [2]:

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

$$a_{o,i} = \frac{dy_{o,i}}{dx_{o,i}}, \quad b_{o,i} = y_{o,i} - a_{o,i}x_{o,i} \quad 0 \leq i < N_{vertices}$$

$$slack_{o,i,n} \Rightarrow \begin{cases} b_{o,i} \leq p_{n,y} - a_{o,i}p_{n,x} & dx_{o,i} < 0 \\ b_{o,i} \geq p_{n,y} - a_{o,i}p_{n,x} & dx_{o,i} > 0 \end{cases} \tag{7}$$

$$\neg \bigwedge_i slack_{o,i,n} \quad 0 \leq n < N \tag{8}$$

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 [6].

III. Segmentation of the MILP problem

In this section we propose a preprocessing pipeline that makes the problem more scalable. Alg. 1 shows the outline of the algorithm.

Algorithm 1 General outline

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

First, we find an initial path with the Theta* algorithm (line 2). Unlike a trajectory, a path is not time-dependent and does not take dynamic properties into account. Then, we find all the turns in that path (line 3). After that, we generate path segments based on those turns (line 4). Each path segment contains the information needed to construct a MILP subproblem. Finally, for each segment, a MILP subproblem is constructed and solved (line 8-9). Before the MILP subproblem is solved, a heuristic selects several obstacles to be modeled in the problem. A genetic algorithm generates a convex safe region which is allowed to overlap those selected obstacles only (line 7). Because the UAV must stay within the safe region, it cannot collide with obstacles. For all but the first segment, the starting state for the UAV in the MILP problem is updated to match the final state of the UAV in the previous segment (line 6). Once all the segments have been solved, their trajectories are merged into the final result (line 11).

Our goal is to divide the problem into subproblems so that only a minimal number of obstacles need to be modeled in each subproblem, while still resulting in a relatively fast trajectory. Subproblems based on shorter segments with fewer obstacles are easier to solve, but the UAV 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 turn, the UAV may not be able to slow down enough if it is going too fast. Longer segments allow the UAV 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 turn in the Theta* path, we can make the segments just large enough so the UAV can always slow down in time to execute the turn. This way the UAV will always turn efficiently, without making the segments too large to solve in an acceptable amount of time.

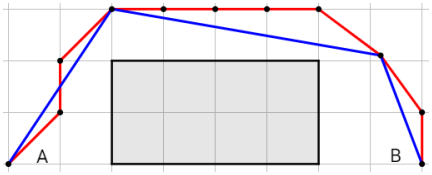


Fig. 3: A typical A* path in blue compared to a Theta* path in red. The gray rectangle is an obstacle.

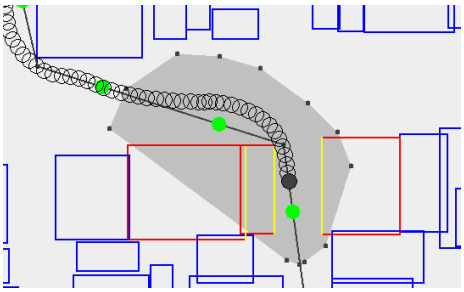


Fig. 4: The red/yellow shapes are the obstacles modeled in the MILP problem, using the same color scheme as in Fig. 2a. The blue shapes are the remaining obstacles. The green circles depict the transitions between segments. The dark gray shape is the convex safe area generated by the genetic algorithm. The solid black circle represents the current position of the UAV, with the hollow circles showing the position in previous time steps.

A. Finding the initial path

The first step in Alg. 1 is finding the Theta* path (line 2), which will be used to divide the problem into segments. The MILP problem generated from each segment needs an intermediate goal to guide the UAV closer the final goal position.

We use Theta* [7] to find an initial path which connects the start and goal positions. Theta* is a variant of A* which eliminates the jagged paths associated with A* as demonstrated in Fig. 3. This path does not take any of the vehicle dynamics into account. Multirotor UAVs can hover, so the UAV can always follow this path by moving in straight lines and stopping at each node of the path. This ensures that successful navigation to the goal position is possible.

B. Detecting turn events

Because the shortest path between two points in Euclidian geometry is a straight line, any turn at a node in the Theta* path must have at least one obstacle on the inside of that turn. A turn without an obstacle on the inside can always replaced by a shorter, straight line segment. This means that by definition these ”inner” obstacles make the search space non-convex. A shape is convex if every point on the line between any two points inside the shape is also inside the shape. Because moving in a straight line between two valid positions on either side of the turn is not possible, the search space must be non-convex if turns in the Theta* path exist.

Non-convexity of the search space is the main cause for the poor performance of Mixed Integer Programming [4]. We have shown that the turns in the path correspond with parts of the problem which must be non-convex. Because of this, we have chosen to generate the segments such that there is at most a single turn in each segment. This limits the non-convexity for each subproblem, significantly improving execution time.

In some cases, a Theta* path contains multiple nodes for a single turn. Alg. 2 groups those nodes together into turn events.

Algorithm 2 Finding Turn Events

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

In the Theta* path, all but the first and last nodes are turns in the path. The second node is always the first node in a new turn event (line 5). Subsequent nodes in the path which are not too far away from the previous node (line 9) and turn in the same direction (line 12) are added to the current turn event (line 15). The maximum distance between nodes in the same turn depends on the maximum acceleration distance of the UAV and a turn tolerance parameter (line 2). The maximum acceleration distance is the distance the UAV needs to accelerate from zero to its maximum velocity, or slow down from the maximum velocity to zero. Once a node is found which does not belong in the event, the event is stored (line 18), a new event is created for that node (line 5) and the process repeats until no more nodes are left.

C. Generating path segments

Each path segment contains the information needed to construct a MILP subproblem. Alg. 3 constructs the segments using turn events. Each segment needs to be large enough so the UAV can safely approach and exit each turn. A multirotor UAV can always safely navigate a turn if it can come to a complete stop before the turn. That is satisfied if the segment starts at least the maximum acceleration distance away from the turn. If the segment starts even earlier, the UAV has more space to maneuver and can navigate the turn more efficiently. However, as the segment gets larger, so does the difficulty of the segment. The approach margin multiplier determines the expansion distance around turn events, based on the maximum acceleration distance.

Algorithm 3 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
```

To generate the segments, Alg. 3 considers each turn event in turn. It keeps track of the end point of the last segment it generated with $lastEnd$. When constructing a segment, it considers the distance between the end of the current turn event and the start of the next turn event. On line 13, two events are too close to each other if they are separated by less than three times [1] the expansion distance. In that case, the segment is constructed to end in the middle between the current and next event (line 14-16). If the events are far enough apart, the end of the current event is expanded forwards by the full expansion distance(line 19-21). Because the next turn event is a long distance away, the $catchUp$ flag is set to true (line 22), ensuring that one or more segments are added to catch up to the start of the next event (line 7-10). To limit the size of segments, they can be no longer

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

than the distance the UAV can travel at maximum velocity in T_{max} time.

D. Generating the safe region for each segment

Algorithm 4 Genetic Algorithm

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

The last step of preprocessing determines which obstacles will be modeled in the MILP subproblem for each segment. Not all obstacles need to be modeled in the MILP problem to prevent collisions. Obstacles on the inside of turns in the Theta* path "cause" those turns and make the search space non-convex. However, collisions with obstacles which are further away or on the outside of the turn can be avoided without the reducing the convexity of the search space.

We select the obstacles to be modeled in the MILP by constructing the convex hull of the start and goal positions of the segment, as well as all Theta* path nodes between them. Any obstacle which overlaps this convex hull will be modeled. In some tight turns, obstacles on the outside of a turn also play an important role in the shape of the trajectory. The convex hull is scaled up slightly to also overlap these restricting outer obstacles if they exist.

The convex hull can be considered a safe region. If the UAV stays inside this region, it cannot collide with obstacles since any obstacle that overlaps with the safe region is modeled in the MILP problem. However, this safe region restricts the movements of the UAV more than necessary.

To make the safe region less restrictive, we use a genetic algorithm which attempts to grow it. In our implementation (Alg. 4), each individual in the population represents a single legal polygon. A legal polygon is convex, does not self-intersect, can only overlap with the selected obstacles and contains every node in the Theta* path for that specific segment. The latter requirement prevents the polygon from drifting off. Each individual has a single chromosome, and each chromosome has a varying number of genes. Each gene represents a vertex of the polygon.

The only operator is a mutator (line 4). Contrary to how mutators usually work, the mutation does not change the original individual. This means that the every individual can be mutated in every generation, since there is no risk of losing information. This mutator can add or remove vertices of the polygon by adding or removing genes (lines 12-13), but only if the amount of genes stays between a specified minimum and maximum. The mutator attempts to "nudge" every vertex/gene to a random position inside a circle around the current position (line 15). If the resulting polygon is not legal, it retries a limited number of times (line 16-19).

Tournament selection is used as the selector, with the fitness function being the surface area of the polygon (line 5-6). Fig. 3 Shows the obstacles modeled in the MILP problem in yellow and red, as well as the polygon generated by the genetic algorithm in dark gray.

IV. Results

We test our algorithm in several different scenarios. Each scenario was tested with two different problem sizes. 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. Fig. 7 shows these scenarios visually. Fig. 5 shows a table with detailed information about the scenarios and execution times. Fig. 6 shows the parameters used in the execution of the algorithm.

A. Up/Down Scenario

The first test scenario has very few obstacles, but lays them out in a way such that the UAV needs to slalom around them. The small scenario has only 5 obstacles, while the larger one has 9. Fig. 7a shows the large variant. This is a challenging scenario for MILP because every obstacle causes a turn, making the problem significantly less convex. Without segmentation on the small version of the scenario, the solver does not find the optimal trajectory within 30 minutes. If execution is limited to 10 minutes, the best trajectory it finds takes 26.0s to execute by the UAV. That is less than a second faster than the segmented result while it took more than 20 times more execution time to find that trajectory. For the larger scenario with 9 obstacles, the solver could not find a trajectory within 30 minutes. This scenario clearly shows the advantages of segmentation, even if there only are a few obstacles.

B. San Francisco Scenario

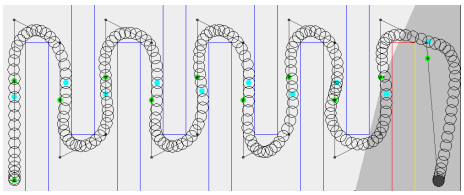
The San Francisco scenario covers a 1km by 1km section of the city for the small scenario, and 3km by 3km section for the large scenario. Fig. 7b shows the small variant. All the obstacles in this scenario are grid-aligned rectangles laid out

scenario	#obstacles	world size	path length	#segsments	Theta* time	GA 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

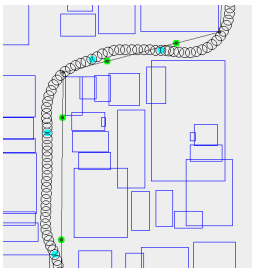
Fig. 5: The experimental results for the different scenarios

grid size	$2m$	turn tolerance	2
approach multiplier	2	population size	10
# generations	25	max. nudge distance	$5m$
min. # vertices	4	max. # vertices	12
P(add vertex)	0.1	P(remove vertex)	0.1
max nudge attempts	15	T_{max}	$5s$
time step size	$0.2s$		

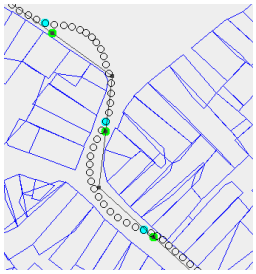
Fig. 6: The parameters used for testing



(a) Up/Down Scenario



(b) A small part of the San Francisco Scenario



(c) A small part of the Leuven Scenario

Fig. 7

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[2], 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 trajectory planning algorithm. The density of obstacles varies greatly and is on average much higher than in the San Francisco dataset. The algorithm does slow down compared to the San Francisco dataset, but still runs in an acceptable amount of time. As visible in Fig. 7c, 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.

[2] 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 final trajectory is generated by a solver so the constraints on the trajectory can easily be 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.
We demonstrate that our new approach can be used to improve the scalability of MILP trajectory planning. However, more work is required to use the algorithm with an actual UAV. Extending the algorithm to 3D is the next step. A complimentary, short-term online planner is necessary for a physical UAV to execute the generated trajectory.

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

[6] R. M. Karp, "Reducibility among combinatorial problems," in *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York.*, pp. 85–103, 1972.

[7] K. Daniel, A. Nash, S. Koenig, and A. Felner, "Theta*: Any-angle path planning on grids," *Journal of Artificial Intelligence Research*, vol. 39, pp. 533–579, 2010.