Scalable 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 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.

Our approach currently only works in 2D. 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 vehicle starts executing its task. These limitations were considered, but we decided to keep the problem simple and focus on demonstrating that the new approach does in fact work. We believe that the approach should also be effective in 3D, although more work is necessary.

Other papers have used MILP for trajectory planning [**?**], 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. [**?**] 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

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

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

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[**?**] 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[**?**] and Deits et al[**?**] 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 [**?**], [**?**], [**?**], [**?**] 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 [**?**] 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. [**?**] 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. [**?**] 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. [**?**].

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 [**?**].

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.

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

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

Eq. (**??**) and (**??**) 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.

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

Eq. (**??**) 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.

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

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

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

Eq. (**??**) initializes $done_0$ to be false, ensuring that the initial state is not finished. Eq. (**??**) 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 [**?**] state transition axiom method was used to model state transitions. In (**??**), 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 (**??**) 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

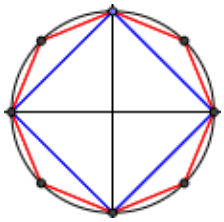


Fig. 1: If the velocity is limited to a finite value, the velocity vector must lie within the circle centered on the origin with the radius equal to that value. This is represented by the black circle. The blue and red polygons show the approximation using 4 and 8 linear constraints respectively.

the state transition, like a maximum velocity, can be added here.

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

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 (**??**). Fig. **??** shows a visual representation of these constraints.

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

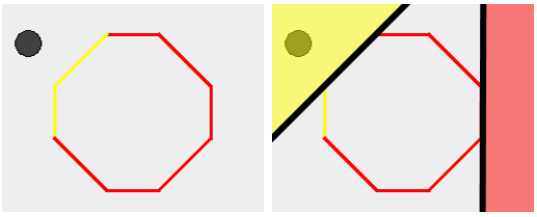
$$b_i = y_i - a_i x_i \quad 0 \leq i < N_{points} \tag{10}$$

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

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.

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



(a) (b)

Fig. 2: A visual representation of how obstacle avoidance works. Fig. **??** shows the vehicle's current position as the black circle. 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. Fig. **??** shows the safe zones defined by a yellow and red edge in yellow and red respectively.

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. Fig. **??** demonstrates this visually. The most common way to turn off individual constraints is using the "Big M" method, like Schouwenaars et al. [**?**] used in their work. However some solvers 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 (**??**) and (**??**):

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

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

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

The occurrences of $offset_{o,i}$ in (**??**) 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

Algorithm 1 General outline

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

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

The first step is finding an initial path with the Theta* algorithm (line 2). 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 (line 3). The third step is generating segments based on those corners (line 4). Finally, for each segment, the active obstacles to be modeled in the MILP problem are selected while the others are approximated with a genetic algorithm (line 7). For all but the first segment, the starting state for the vehicle is updated to match the final state of the vehicle in the previous segment (line 6). Once all the segments have been solved, their trajectories are merged into the final result (line 10).

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.

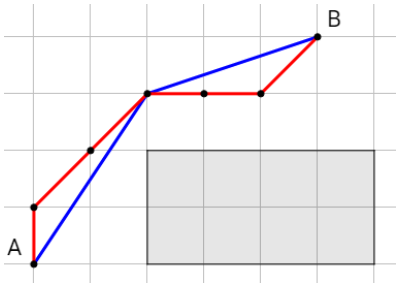


Fig. 3: A typical A* path in blue compared to a Theta* path in red. The gray rectangle is an obstacle.

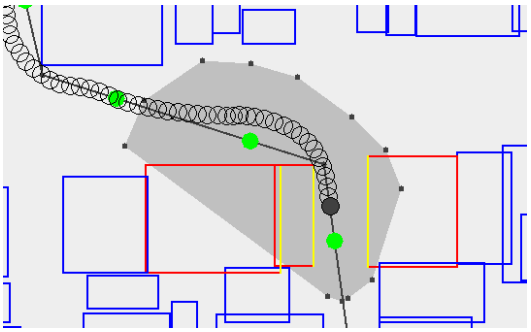


Fig. 4: A visualisation of the output of the algorithm The blue shapes are inactive obstacles. The red/yellow shapes are active obstacles using the same color scheme as in Fig. **??**. The green circles depict the transitions between segments. The dark gray shape is the convex safe area generated by the genetic algorithm. The solid black circle represents the current position of the vehicle, with the hollow circles showing the position in previous time steps.

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. Fig. **??** shows a comparison between A* and Theta*. The grid size is represented as S_{grid}.

B. Detecting corner events

Algorithm 2 Finding Corner Events

```
1: function FindCornerEvents(path)
2:     Δmax ← Δv_max * tol_corner
3:     events ← {}    ▷ The list of corner events found so far
4:     i ← 1    ▷ Skip the start node, it can't be a corner
5:     while i < |path| − 1 do    ▷ Skip the goal node
6:         event ← {path(i)}    ▷ Start new corner 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
```

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.

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.

In the Theta* path, all but the first and last nodes are corners in the path. Alg. **??** shows how the corner events are found. The second node is always the first node in a new corner event (line 5). Subsequent nodes in the path which are not too far away from the previous node (line 9) or turn in the opposite direction (line 12) are added to the current corner event (line 15). The maximum distance between nodes in the same corner depends on the agility of the vehicle and a tolerance parameter tol_{corner} (line 2). The agility of the vehicle is represented by Δv_{max}, the distance the vehicle needs to accelerate from zero to its maximum velocity. 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

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. **??** considers each corner 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 corner event and the start of the next corner event. The distance both events would expand in the ideal case is $\Delta_{approach} = mul_{approach} * \Delta v_{max}$, with $mul_{approach}$ being the approach multiplier. If the multiplier is larger than 1, the vehicle can come to a complete stop before the corner, ensuring it can always navigate the corner successfully. A larger value allows for a more efficient approach. On line 13, two events are too close to eachother if they are less than $3\Delta_{approach}$ separated. 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 event is moved forwards along the path by $\Delta_{approach}$ and is used as the end of the segment (line19-21). Because the next corner 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 than the distance the vehicle can travel at maximum velocity in T_{max} time.

D. Generating the active region for each segment

Algorithm 4 Genetic Algorithm

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

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

The inactive obstacles also need to be represented. To do this, a convex polygon is constructed around the path. This polygon may intersect with the active obstacles (since they will be represented separately), but may not intersect any other obstacle. The polygon is grown using a genetic algorithm. Genetic algorithms are inspired by natural selection in biology. A typical genetic algorithm consists of a population of individuals, a selection strategy and one or more operators to generate offspring. In each generation, the operators are applied on the population to produce offspring. These operators usually have a random element and are responsible for exploration of the search space. The selection strategy determines, often based on a fitness function, which individuals survive and form the population for the next generation. Selection is responsible for convergence towards fitter individuals, limiting how much of the search space is evaluated.

S_{grid}	$2m$	tol_{corner}	2	$mul_{approach}$	2
N_{pop}	10	N_{gens}	25	Δ_{nudge}	$5m$
$N_{genes,min}$	4	$N_{genes,max}$	12	p_{add}	0.1
p_{remove}	0.1	$N_{attempts}$	15	T_{max}	$5s$
f_{solver}	$5Hz$				

Fig. 6: The parameters used for testing

Alg. **??** shows our implementation. In our implementation, each individual in the population represents a single legal polygon. A legal polygon is convex, does not self-intersect, does not overlap with inactive obstacles and contains every node in the Theta* path for that specific segment. The latter requirement prevents the polygon from drifting off. Each individual has a single chromosome, and each chromosome has a varying number of genes. Each gene represents a vertex of the polygon.

The only operator is a mutator (line 4). Contrary to how mutators usually work, the mutation does not change the original individual. This means that the every individual can be mutated in every generation, since there is no risk of losing information. This mutator can add or remove vertices of the polygon by adding or removing genes (lines 12-13), but only if the amount of genes stays between $N_{genes,min}$ and $N_{genes,max}$. The mutator attempts to nudge every vertex/gene to a random position at most Δ_{nudge} away (line 15). If the resulting polygon is not legal, it retries at most $N_{attempts}$ times (line 16-19).

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

IV. Results

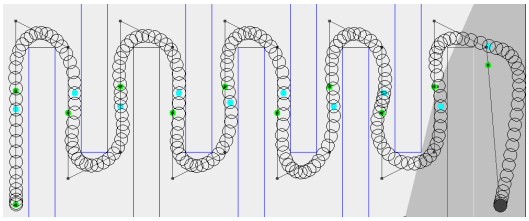
We test out algorithm in several different scenarios. Each scenario was tested with two different problem sizes. Theta* is executed with a grid size of 2m and each time step has a duration of 200ms. All tests were executed on an Intel Core i5-4690k running at 4.4GHz with 16GB of 1600MHz DDR3 memory. The reported times are averages of 5 runs. The machine runs on Windows 10 using version 12.6 of IBM CPLEX. Fig. **??** shows these scenarios visually. Fig. **??** shows a table with detailed information about the scenarios and execution times.

A. Up/Down Scenario

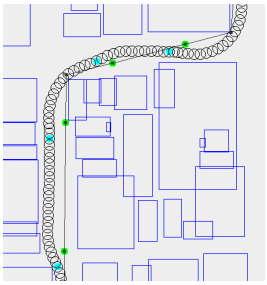
The first test scenario has very few obstacles, but lays them out in a way such that the vehicle needs to slalom around them. The small scenario has only 5 obstacles, while the larger one has 9. Fig. **??** shows the large variant. This is a challenging scenario for MILP because every obstacle has changes the path significantly. Without segmentation on the

scenario	#obstacles	world size	path length	#segsments	Theta* time	GA time	MILP time	score
Up/Down Small	5	25m x 20m	88m	7	0.00s	0.42s	14.21s	26.6s
Up/Down Large	9	40m x 20m	146m	11	0.01s	1.00s	29.7s	43.4s
SF Small	684	1km x 1km	1392m	28	1.31s	9.69s	73.2s	103.9s
SF Large	6580	3km x 3km	4325m	84	15.76s	20.53s	231s	319.6s
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

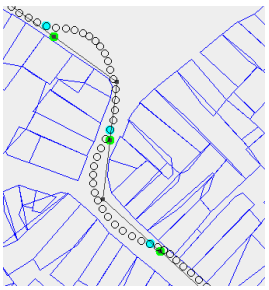
Fig. 5: The experimental results for the different scenarios



(a) Up/Down Scenario



(b) A small part of the San Francisco Scenario



(c) A small part of the Leuven Scenario

Fig. 7

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. Fig. **??** shows the small variant. 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 Fig. **??**, 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.

The results show that this 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 vehicle. Extending the algorithm to 3D is the next step. A complimentary, short-term online planner is necessary for a physical vehicle 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.

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

[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.

[13] 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.