Implementation of the algorithm 1 from "Optimisation based path planning for car parking in narrow environments"

Maksym Ivashechkin, Jiří Vlasák, Antonín Novák
Tuesday, 14:30-16:00
Open Informatics
ivashmak@fel.cvut.cz

New experiments starting on the page 6 in the separate subsection "New experiments" with time measurements. 27.05.2020

I. Assignment

A. Problem Statement

The task is to implement the algorithm 1 from the "Optimisation based path planning for car parking in narrow environments"[1]. The algorithm is used for searching the optimal sequence of car positions (x,y-coordinates and orientation angle) to the predefined parking slot. The procedure includes also solving the optimization problem of finding the best control input of a car. Control input is a tuple of steering angle and step length of a car. Both values are bounded such that steering angle can not increase the properties of the car and the recommended in [1] step length is relatively low number equals to 0.2 meter. The control input describes how vehicle moves in the specific moment (iteration of the algorithm). The optimization is a quadratic function with non-linear constraints, modeled in the way to minimize the distance of the current position of the car (x,y coordinate and orientation) to the target position (i.e., parking slot).

II. Problem solution

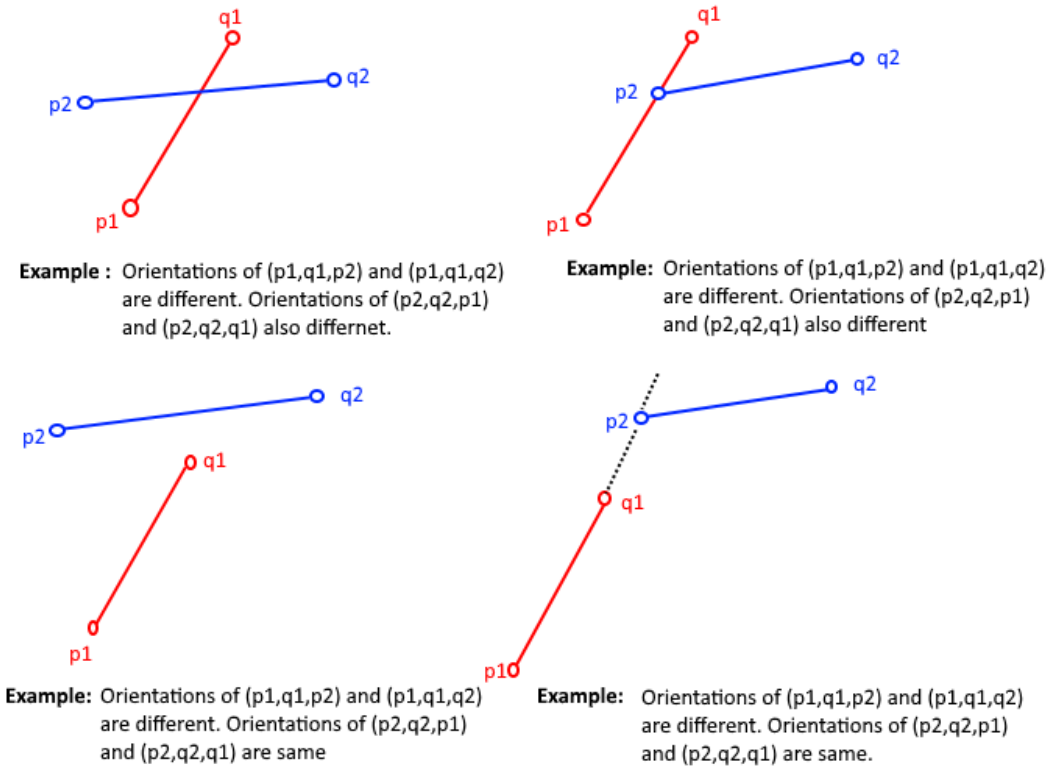
A. Design

The algorithm 1 contains one while-loop which terminates when either number of maximum iterations reached or number of maximum direction changes reached, or distance of current position and the target one falls under threshold. In the beginning vehicle is located in the phase B – area, where it has to do a manoeuvre. After that it switches to the phase A and moves directly to the parking place. Every iteration requires to solve the optimization task to find the optimal control input which subjects to physics of the vehicle and satisfies that all collisions (obstacles) are avoided. If such optimization procedure fails then car changes its direction (e.g., moves not forward but backward and vice versa). Otherwise, if the cost of current optimization function is smaller than previous cost then the new position of the car is updated by found control input; however, if cost is higher than previous then car changes its direction again.

B. Implementation

The core is implementing the optimization procedure of the algorithm 1. It has been done in Python with Gurobi solver [2]. Since the cost function is quadratic and the constraints and equalities are non-linear (including also trigonometric functions) so in Gurobi 9.0 the parameter NonConvex has to be set to 2, otherwise Gurobi can not solve it (see here).

The most difficult part is dealing with obstacles (which by assumptions are convex polygons). The proposed method in [1] is using Minkowski sum, however these constraints must be written in Gurobi as simple as possible (something using standard operations like additive, multiplication, inequalities), this is though not the case for Minkowski sum which requires a lot of local variables and computations. So, another way to avoid collisions is to check segments' intersection. For example, if car has 4 segments (4 sides), the obstacles has in average M segments (around 4) and there are N obstacles then $4 \times M \times N$ segments' intersections has to be checked. The simple method to verify if two segments intersect is shown in GeeksForGeeks: it takes one segment and computes orientation of points of the second segments and vice versa. Two segments intersect if they both have different orientations for both pair of points (it is nicely visualized in 1). Let us have two segments of points $p1, p2$ and $q1, q2$, this method could be written in the following way:



Obrázek 1: Intersection of segments from GeeksforGeeks.

$$\begin{cases} sign((p2_y - p1_y)(q1_x - p2_x) - (p2_x - p1_x)(q1_y - p2_y)) \neq sign((p2_y - p1_y)(q2_x - p2_x) - (p2_x - p1_x)(q2_y - p2_y)) \\ sign((q2_y - q1_y)(p1_x - q2_x) - (q2_x - q1_x)(p1_y - q2_y)) \neq sign((q2_y - q1_y)(p2_x - q2_x) - (q2_x - q1_x)(p2_y - q2_y)) \end{cases}$$

Since, constraints for Gurobi must be the opposite (segments must not intersect) then equations should be negated, so either the orientation is the same for the first segment or for the second. Also the signum function should be rewritten to something simpler, for example:

$$sign(x) = sign(y) \quad \text{is equivalent to} \quad 0 \leq x \cdot y \tag{1}$$

The OR condition is rewritten using the big-M format and binary variable s, so final constraints for two segments are

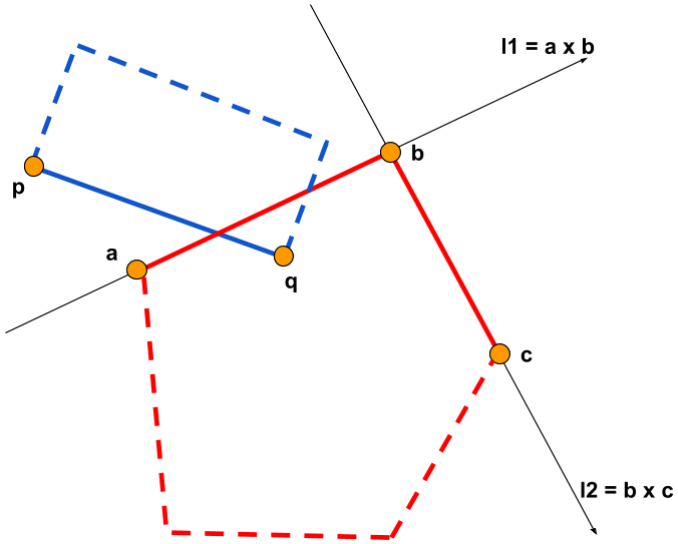
$$\begin{cases} 0 \leq ((p2_y - p1_y)(q1_x - p2_x) - (p2_x - p1_x)(q1_y - p2_y)) \cdot ((p2_y - p1_y)(q2_x - p2_x) - (p2_x - p1_x)(q2_y - p2_y)) + (1 - s)M \\ 0 \leq ((q2_y - q1_y)(p1_x - q2_x) - (q2_x - q1_x)(p1_y - q2_y)) \cdot ((q2_y - q1_y)(p2_x - q2_x) - (q2_x - q1_x)(p2_y - q2_y)) + \ sM \end{cases}$$

The advantage of this segments' verification over many others is that it does not require division – which could not be used in Gurobi solver.

Since, the maximum range where a vehicle could move in one iteration is known (by maximum step length and maximum steering angle – control input) then only a few neighborhood obstacles could be considered – which is a significant speed up. If there are no obstacles the optimization task is solved quite fast (around a few milliseconds). If there are one or two obstacles it solves usually a few seconds but sometimes if obstacles are very closed to the car then optimization process can take even minutes. Unfortunately, the precise correlation when and why this happens is not fully clear but Gurobi has a time limit option and furthermore even if one optimization step fails it does not mean that everything fails – the car has to change direction. However, if the optimization procedure fails two times in a row it means that the whole algorithm fails, because the optimization task failed for both directions and there is no sense to continue (this was not mentioned in the [1] but during the implementation it seems reasonable to add such condition).

The implementation of the algorithm 1 is under gitlab repository: https://gitlab.fel.cvut.cz/ivashmak/ko_semestral_project

C. Lazy constraints for obstacles



Obrázek 2: Blue (resp. red) (dashed) lines are segments of the car (resp. convex obstacle). There is an intersection of segments $(\mathbf{p}, \mathbf{q})$ and $(\mathbf{a}, \mathbf{b})$.

There could be many obstacles around the car in the moment for solving the optimal input control. Obviously, checking all segments of car and obstacles may be computationally exhaustive. Furthermore, it is not even necessary to write constraints for all segments because car may not intersect obstacle at all or only some segments may intersect.

Fortunately, Gurobi deals with lazy constraints when optimal solution is found. So, when the best optimal input is obtained then segments of near obstacles must be checked. If there exists two segments which intersect then constraint for those segments has to be added.

The method presented in previous section for writing constraint for two segments intersection can not be used in lazy constraints because proposed inequalities are quadratic (w.r.t. unknown car position) but Gurobi requires lazy constraints to be linear (see cbLazy). In this case let us consider car segment $(\mathbf{p}, \mathbf{q})$ and obstacle segment $(\mathbf{a}, \mathbf{b})$ which intersect (see figure 2). Either the point $\mathbf{p}$ or $\mathbf{q}$ must be inside convex obstacle. To find out which point, let $\mathbf{l}_1 = \mathbf{a} \times \mathbf{b}$ be a line passing through points $\mathbf{a}$ and $\mathbf{b}$ and similarly let $\mathbf{l}_2 = \mathbf{b} \times \mathbf{c}$ be a line passing through points $\mathbf{b}$ and $\mathbf{c}$. Lines $\mathbf{l}_1$ and $\mathbf{l}_2$ are two consecutive lines of the obstacle, so if point $\mathbf{t}$ lies inside the obstacle then $sign(\mathbf{l}_1^\top \mathbf{t}) = sign(\mathbf{l}_2^\top \mathbf{t})$ so the linear constraint for Gurobi will be the following:

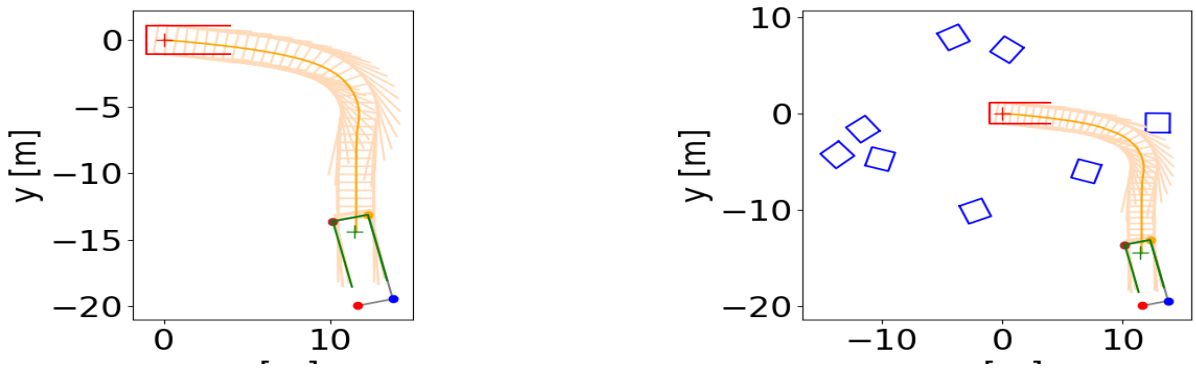
$$\begin{cases} \mathbf{l}_1^\top \mathbf{t} \leq 0 & \text{if } \mathbf{l}_1^\top \mathbf{t} > 0 \\ \mathbf{l}_1^\top \mathbf{t} > 0 & \text{if } \mathbf{l}_1^\top \mathbf{t} \leq 0 \end{cases}$$

It means that the point $\mathbf{t}$ (in figure 2, $\mathbf{t} = \mathbf{q}$) is forced to be outside the convex obstacle.

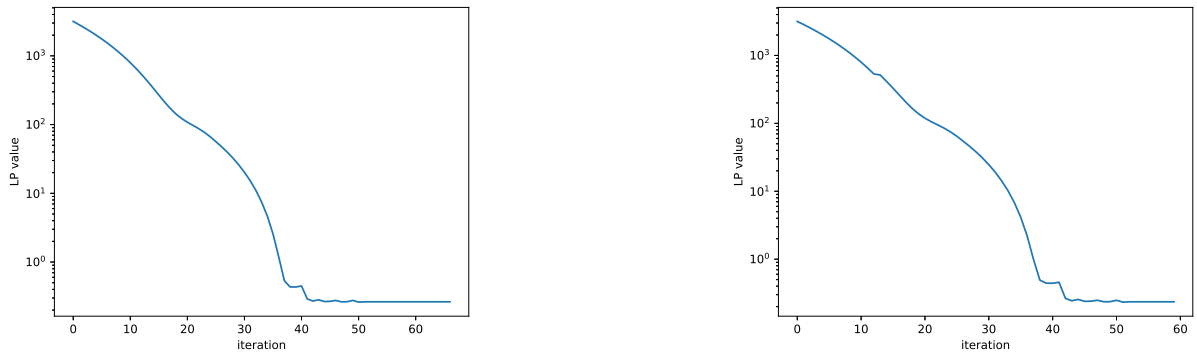
The disadvantage is of course in verifying segments' intersection for every found optimal solution but it is still much faster than writing all constraints at one time. Experimentally, this approach is much faster comparing to described in the previous section.

III. Experiments

A. Results



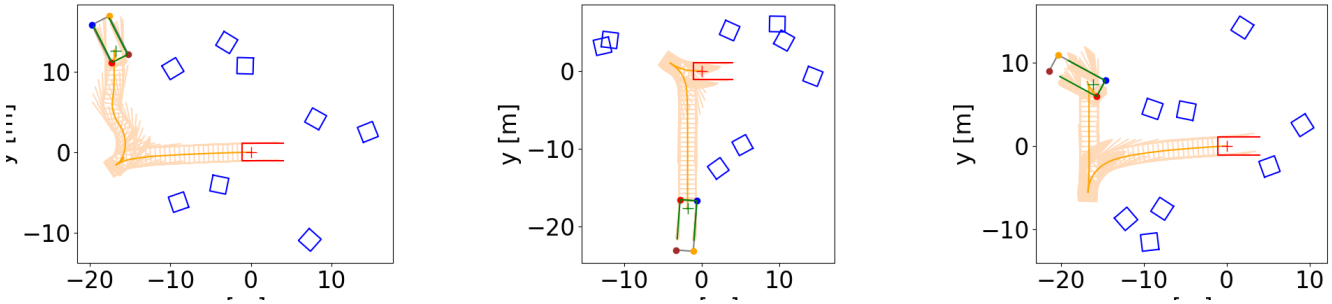
Obrázek 3: The sequence of moves in the left image for data with removed obstacles. In the right image the same data with obstacles.



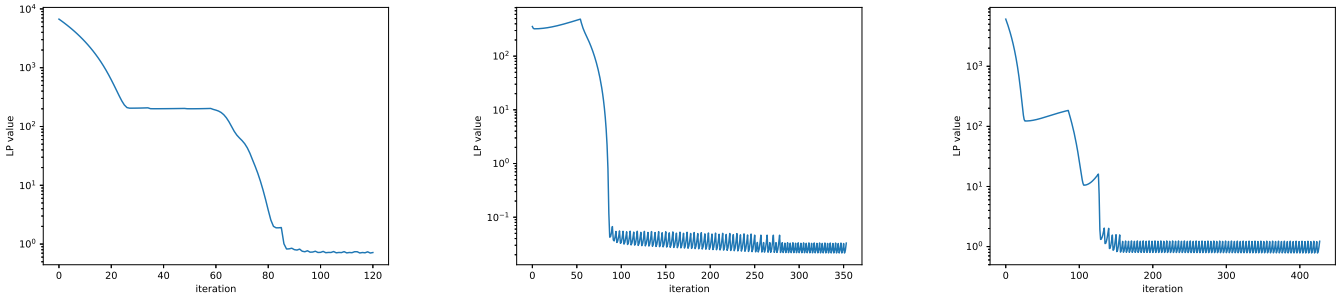
Obrázek 4: The y-axis shows (log) costs of optimization function. The x-axis shows the iteration. The right graph has a small difference in iterations 15-16 (compared to the left graph) caused by obstacle.

In the figure 3 is example of sequence of moves from the initial car position (red box with a cross) to the parking slot (green box with a cross and colorful points). Pink boxes and yellow curve show the found sequence. Blue squares in 3 are obstacles. Graphs below in the figure 3 show the cost of optimization function. Note, that cost values sometimes higher than previous. The reason is that in [1] is proposed to accept in some cases a worse solution (in terms of the cost value) but not a k-worse (to be discussed in Discussion section).

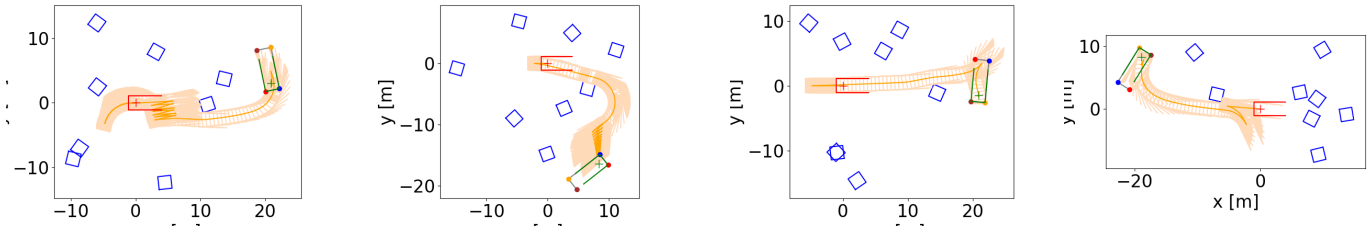
The optimization procedure struggles at most when the vehicle is almost at the target position. As could be seen from images 3, 5, 7, 21 and graphs 4, 6, 8, 22. Something strange is going on near parking slot, similarly the costs frequently alternate and could not converge.



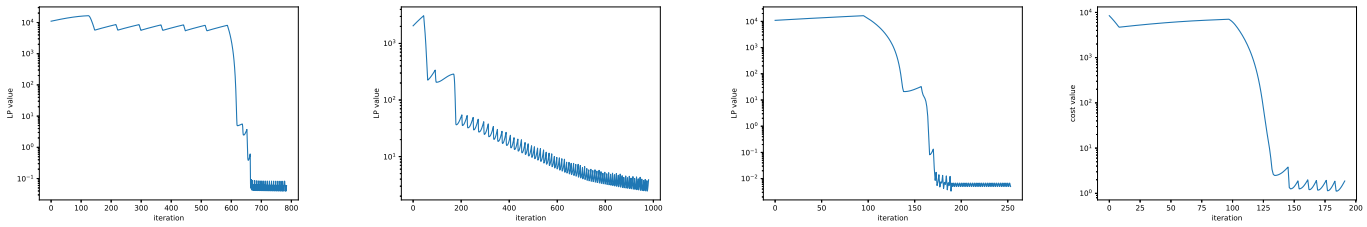
Obrázek 5: Easy cases with obstacles.



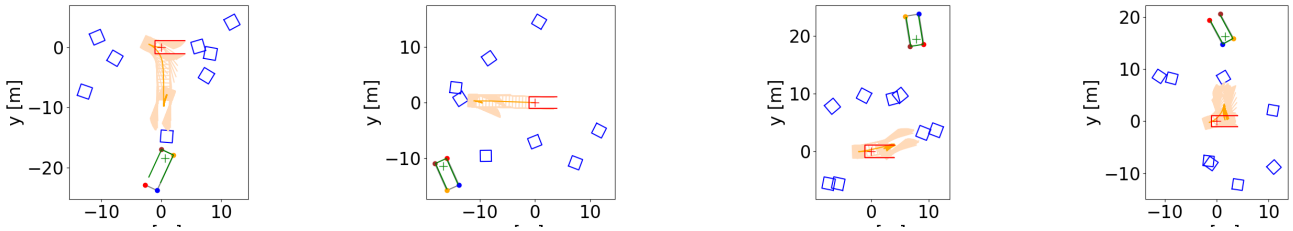
Obrázek 6: The corresponding cost values



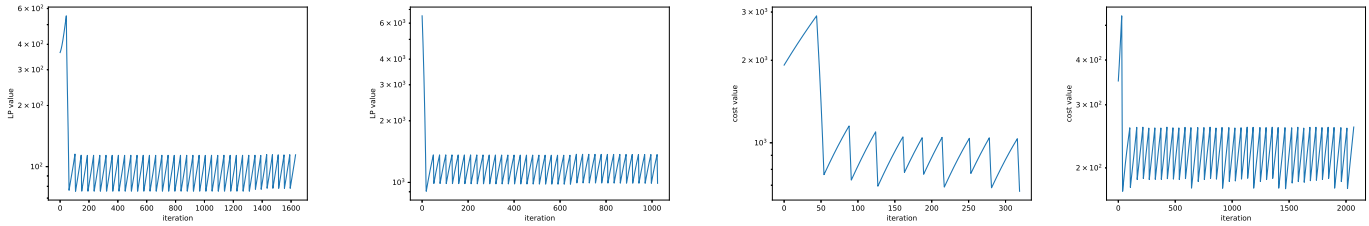
Obrázek 7: Hard cases with obstacles.



Obrázek 8: The corresponding cost values



Obrázek 9: "Failed"cases with obstacles. The vehicle tried to avoid obstacle but the maximum number of iterations / direction changes reached.

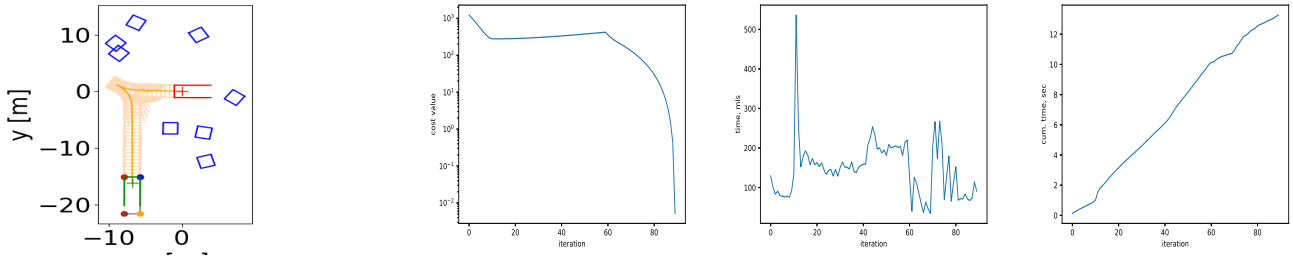


Obrázek 10: The corresponding cost values

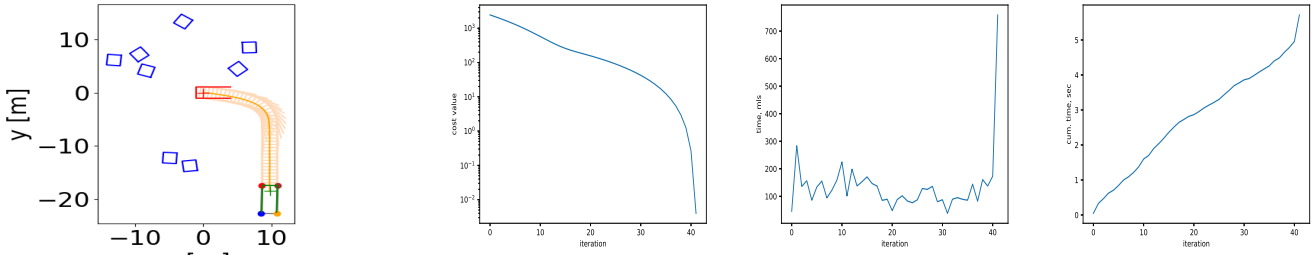
B. New experiments

In this subsection are briefly discussed (in captions) 10 different scenarios. The first image in a row is output sequence of the car. The second figure is a graph of the cost function. The third figure is time spent for one iteration (in milliseconds). And the last figure shows cumulative time of iterations (in seconds, so the last value is the total time of the algorithm). The experiments were run on the server INTEL(R) XEON(R) CPU E5-2630 v2 @ 2.60GHz with 24 processors. Gurobi was using all 24 threads. The time limit for one iteration was set to 1 minute (for slower computers timeout should be bigger).

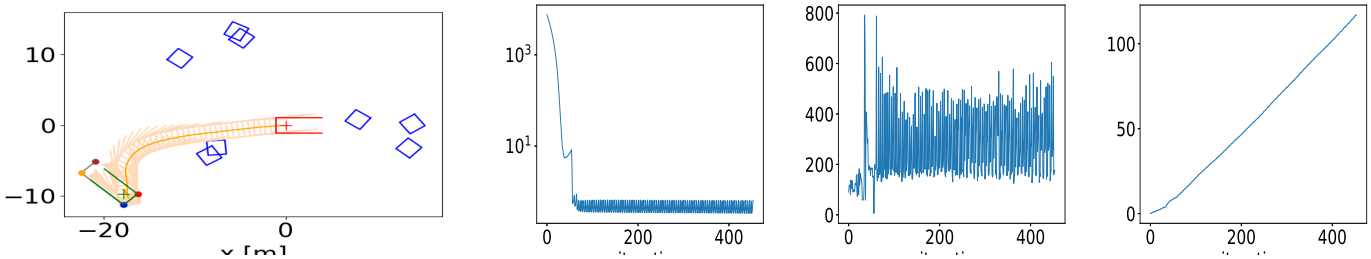
The new experiments were run with lazy constraints for avoiding collisions so the average time of one iteration even containing obstacles is several hundred of milliseconds. By my observation the number of lazy constraints which was added during the running is in average less than 1 equation (many times none was needed).



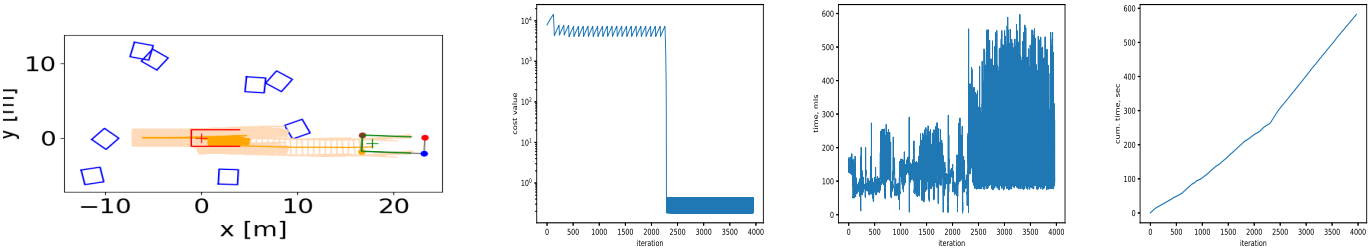
Obrázek 11: Easy case. Algorithm finished in less than 100 iterations. The cost function decreased very quickly. The total time of the whole sequence is 13.27 seconds, the average time of one iteration is around 150 milliseconds.



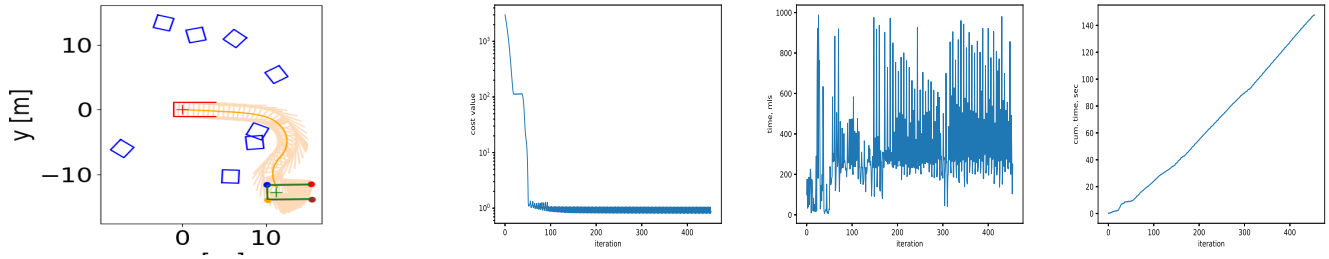
Obrázek 12: Another easy case. The sequence was found in less than 100 iterations. The total time is 5.71 seconds.



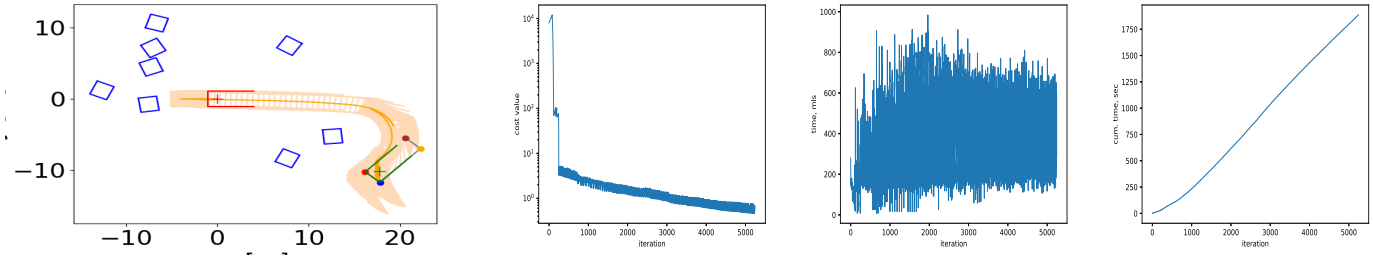
Obrázek 13: The car successfully avoided obstacle (in a little risky way) in around 70 iterations. However, other 400 iterations the algorithm was struggling to obtain desired angle of parking. It can be seen from graph that the cost was alternating after the drop and also, the time for solving increased. The total time is 116.92 seconds.



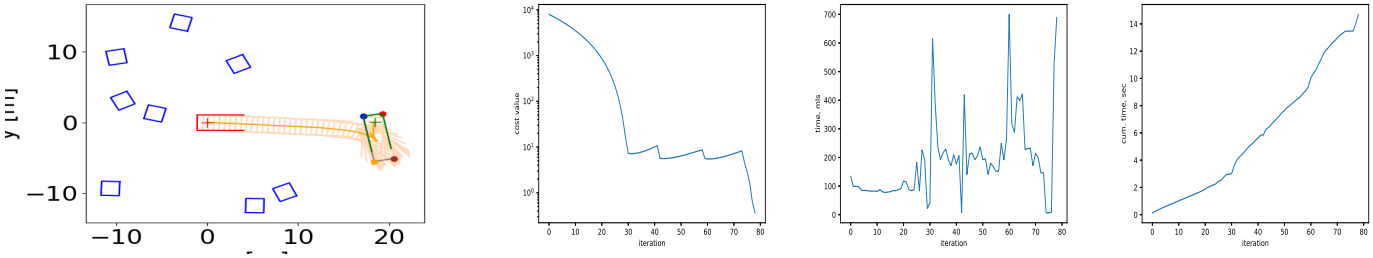
Obrázek 14: Because of the obstacle ahead it took around 2000 iterations for algorithm to avoid collision. After there was a significant drop in the cost. Then algorithm spent other few thousand of iterations to find the right angle of parking. The total time is 581.91 seconds.



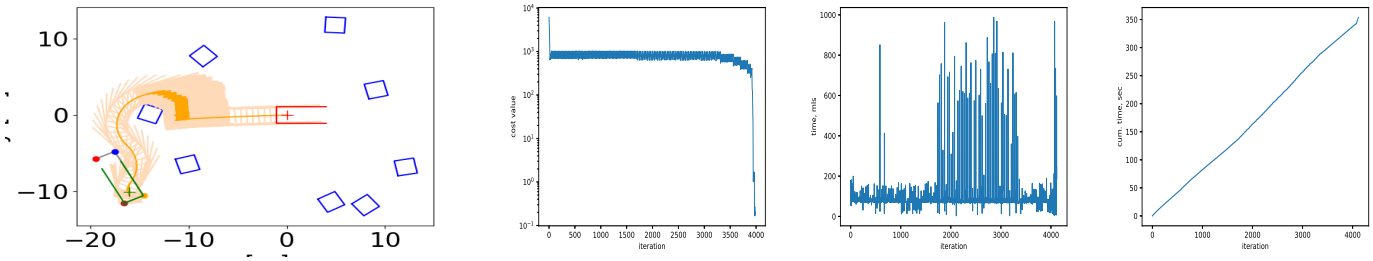
Obrázek 15: The algorithm reached the target position relatively quickly (in around 180 iterations) avoiding obstacles. However, most of the time took rotating to the desired parking angle. The total time is 146.78 seconds.



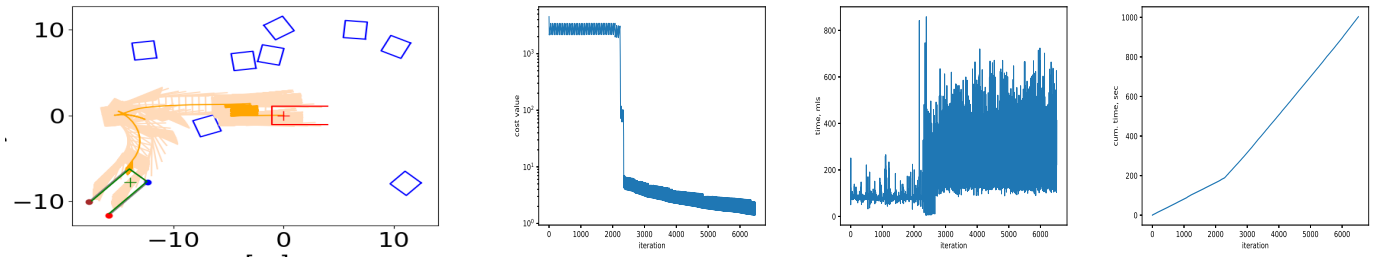
Obrázek 16: Example, where the problem was again in reaching the target parking angle, as the car got to the desired position fairly quickly. From graph it can be seen that rotating around parking spot took enormously a lot of time and iterations but without success by the end. The total time is 1885.67 seconds $\cong$ 31 minutes.



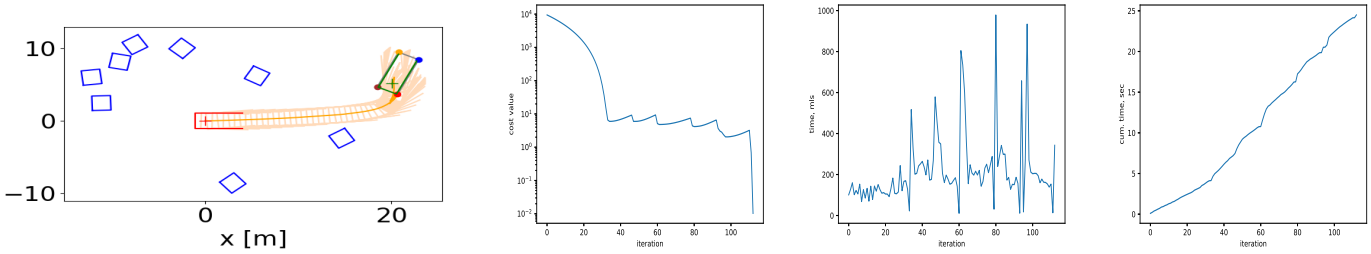
Obrázek 17: Easy case, the algorithm spent some time for obtaining target angle of parking. The total time is 14.69 seconds.



Obrázek 18: Car parked backwards omitting the obstacle. It took more than 3000 iterations to find the sequence to avoid collision. The total time is 353.65 seconds.



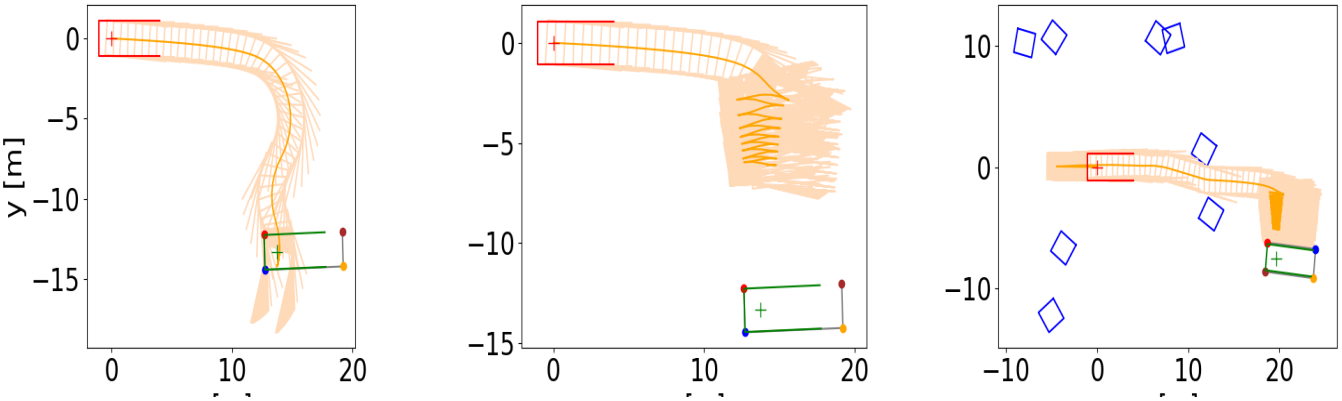
Obrázek 19: Similarly as in previous scenario the car started parking backwards but after avoiding obstacle it rotated to get the target parking angle (it took most of the time). The total time is 1003.02 seconds.



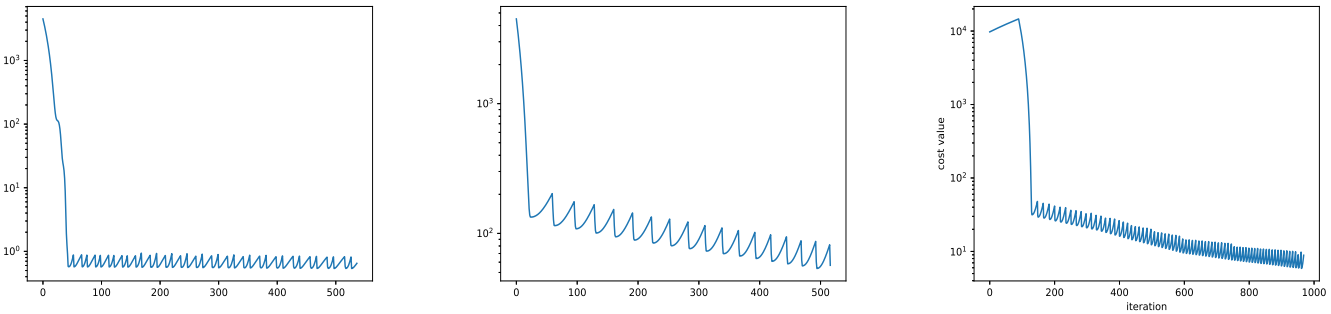
Obrázek 20: Easy case of parking forwards. The total time is 24.49 seconds.

So, seems to be that one of the main weakness of the algorithm (probably implementation itself or Gurobi solver) is getting the desired angle of parking. For many scenarios the car reached target position (x, y-coordinates) mostly without problems, but rotating in the parking spot took many iterations and time. However, this can be regulated by the cost function, i.e., by decreasing the weight of having parking angle closed to the target one.

C. Discussion



Obrázek 21: Two first images share the same scenario with different weight of the target angle for the optimization function. The left image has lower weight so the car reached the target position faster. On the right image the angle's weight is higher so the vehicle moves in the way to keep the angle closed to the target one. The third image is different scenario but with the same angle weight as on the second image.



Obrázek 22: The corresponding costs values.

There at least two things to discuss:

- The parameter $k \in [1, 2)$ (the proposed value in [1] is 1.5), which is used in algorithm 1 to compare the current and previous cost of optimization function ($l^*_{o_i} > k\, l^*_{o_{i-1}}$), allows to accept the solution which may be worse than previous ($l^*_{o_i} > l^*_{o_{i-1}}$) but less than $k\, l^*_{o_{i-1}}$. This was done to make a possibility for the car to change directions when

going to parking slot. However, in the experiments were a lot of cases when the sequence of costs was increasing $l^*_{o_t} < l^*_{o_{t+1}} < ... < l^*_{o_{t+n}}$ but the condition $l^*_{o_t} \leq k\, l^*_{o_{t-1}}$ for each pair was satisfied, the car was moving nowhere. So, there are two options to avoid this. First, set k to very low number (e.g., 1.0001), so a little bit worse solution is still accepted. The second option, which was used in the experiments, is to keep k the same as recommended in [1] but compare the current cost versus the minimum over all costs, i.e., $l^*_{o_i} > k \min_i l^*_{o_{i-1}}$ so the worse solution can be still accepted but not k-worse than the minimum one.

- The weight of target angle can influence the algorithm very much as shown in figure 21. If the weight of target angle is a quite low number then car reaches the target position faster, otherwise the vehicle tries to save the target angle and moves in an awkward way.

IV. Conclusion

The implementation is of course not very accurate neither super fast. The are a few reasons for this:

- The bugs in the implementation. I hope there are not but since I am biased to implementation it is difficult to spot mistakes.
- The slowness of Python or Gurobi (which for academic usage does not use all computational power) or laptop.
- The constants proposed in [1] are very important and should be re-considered for our tasks.

There is definitely a way to speed up collision avoidance considering not whole neighborhood obstacles but also the separate segments which could possibly intersect in the solving step.

Reference

[1] Patrik Zips, Martin Böck, and Andreas Kugi. 2016. *"Optimisation based path planning for car parking in narrow environments"*. Robot. Auton. Syst. 79, C (May 2016), 1-11.

[2] Gurobi Optimization, LLC, *"Gurobi Optimizer Reference Manual"*, 2020, *"http://www.gurobi.com"*.