2017 IEEE Intelligent Vehicles Symposium (IV)
June 11-14, 2017, Redondo Beach, CA, USA

From videogames to autonomous trucks: A new algorithm for lattice-based motion planning

Marcello Cirillo[1]

***Abstract*— Autonomous navigation in real-world environments is still a challenging task in many respects. One of the key open challenges is fast planning of physically executable complex maneuvers under non-holonomic constraints. In recent years, lattice-based motion planners have been successfully used to generate kinematically and kinodynamically feasible motions for non-holonomic vehicles. However, it is not clear yet what algorithms are best to efficiently explore the lattice state space, while at the same time ensuring real-time performance. Here, we show how motion planning can greatly benefit from tapping into the latest results in path planning on grids, and we present a new version of *Time-Bounded* A^*. Our version is designed to work for high-dimensional motion planning problems in real-world robotic applications. We demonstrate our algorithm in simulation and on a full-size autonomous truck.**

I. Introduction

In recent years, the interest for autonomously driving vehicles has steadily increased. Many big actors in the car industry, as well as competitive outsiders have joined the race to provide the world with the first fully autonomous cars, trucks or buses [1]. Thanks to this interest, great resources have been allocated worldwide to develop the new algorithms and techniques necessary to reach the ambitious goal, and the by-product of the race has been the commercialization of new advanced safety systems. When it comes to industrial tasks, such as in mining or intra-logistic scenarios, solutions which totally or partially rely on autonomous vehicles have been available for many years [2]. This is because some of the most challenging problems that must be addressed in urban environments are muted when autonomous vehicles operate in special, enclosed areas. However, industrial solutions can still greatly benefit from recent advancements.

The industry standard approach to motion planning for autonomous vehicles still relies largely on fixed paths [3], which have been either previously driven by a human operator or manually drawn during system deployment. Both approaches, although effective, present the major drawback that even small modifications to the environment require defining new paths. Moreover, vehicles on pre-defined paths can only deal with unexpected obstacles by reducing their velocity or by employing very simple avoidance strategies.

In this paper, we address the problem of motion planning for non-holonomic vehicles in unstructured environments, that is, in possibly large areas where a vehicle cannot follow roads and needs to perform complex maneuvers. More specifically, we focus on heavy transport vehicles, as our work is driven by the goal of introducing a new level of

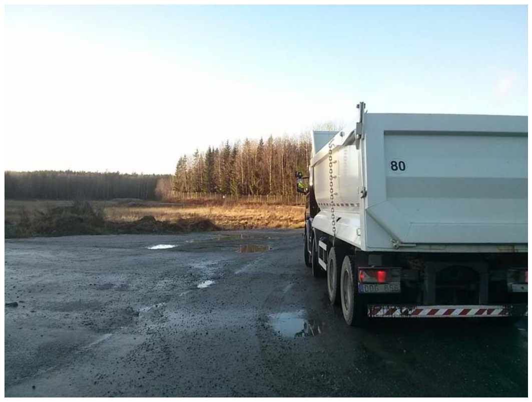


Fig. 1: The autonomous truck used in our tests and an area were the truck is required to maneuver.

autonomy in environments such as open and underground mines, or construction sites. Although these environments typically contain a low number of other agents, trucks operating there require advanced motion planning algorithms: The transportation tasks effectively change the landscape of the areas, loading and unloading locations change over time, and the maneuvering spaces can be quite narrow. As a practical example, consider the area in Fig. 1. Here, the truck arrives on the maneuvering area through a narrow road, and it needs to get to a loading place whose location can change over time and can be precisely identified only by sensor readings. Also, other trucks and loaders may have moved material around, thus creating new unexpected obstacles. Although motion planning has been the focus of extensive studies in the past decade, and many solutions have been proposed to deal with non-holonomic vehicles [4], [5], a definitive solution for the problem described above does not exist as yet, and the existing ones can still be greatly improved.

Our main contribution is to show how motion planning can be further improved by tapping into the wealth of algorithms developed for graph search outside the robotics community. We describe why and how we adapted *Time-Bounded* A^* [6] for working with robotic systems, and we demonstrate its effectiveness in simulation and on an autonomous truck.

II. Related Work

Motion planning has been extensively studied in the past decades. Whenever differential constraints and obstacles are considered, combinatorial methods and analytical solutions are of limited use [4]. The former are not well suited in the presence of differential constraints, while the latter cannot effectively cope with obstacles. When it comes to planning motions for car-like vehicles on roads, several assumptions

[1]Marcello Cirillo is with Scania, Autonomous Transport Solutions, Södertälje, Sweden `marcello.cirillo@scania.com`

can be made that led to the development of specialized planners [7]. Most of these results, however, are not suitable for maneuvering in unstructured environments. In these environments, sampling-based methods have been proven to be effective: Probabilistic Roadmaps (PRMs) [8], Rapidly-exploring Random Trees (RRTs) [9] and lattice-based motion planners [10], all of which can work in high-dimensional configuration spaces. PRMs have two major drawbacks: First, several parameters must be selected beforehand (e.g., the duration of the learning phase); and, second, a new roadmap has to be built every time the environment is subject to substantial changes. After their initial introduction [11], RRTs have been extensively studied, and many variants of the original algorithm have been proposed [12], [13]. RRTs do not guarantee convergence (termination is usually implemented with a timeout) and, unless the space is analyzed beforehand, they cannot verify whether a problem offers no solution. Lattice-based motion planners combine the strengths of the previous approaches with classical AI graph-search algorithms, such as A^*, ARA^* [14] and D^*Lite [15]. Differential constraints are incorporated into the state space by means of pre-computed motion primitives which trap the motions onto a regular lattice. The state space is then explored using graph-search algorithms.

Lattice-based planners have proven to be particularly effective for quickly calculating accurate, complex maneuvers (e.g., three-point turns) in cluttered environments [16]. Existing planners, however, generally use only a very restricted subset of the graph search algorithms developed in recent years. More important still, most of the algorithms currently used need to find a complete solution before starting execution. This can be computationally expensive and even useless, as the environment around a mobile vehicle is usually observable only as far as its sensors' range allows. Therefore, the solutions found often need to be corrected or completely re-calculated as new information is acquired.

Parallel to the research on motion planning, there have been very interesting developments in the area of path finding on grids [17]. Videogames have very stringent requirements on the amount of time that can be allotted to path finding for each character, especially when the number of agents to be moved is high and the map only partially observable [18]. The algorithms employed must be effective and must work in real time, providing also partial solutions in the little time available. Although pathfinding algorithms on grids have to consider only a few possible alternative moves at each state (depending if a 4- or an 8-connected grid is used), they can be easily adapted to work on lattices. There is a wealth of algorithms [19], [6] that can be used to improve lattice-based motion planners, and in this paper we show how.

III. Motion planning framework

Given a model of vehicle maneuverability, the intuition behind lattice-based motion planning is to sample the state space in a regular fashion and to constrain the motions of the vehicle to a lattice graph $\mathcal{G} = \langle \mathcal{V}, \mathcal{E} \rangle$, that is, a graph embedded in a Euclidean space $\mathbb{R}^n$ which forms a regular tiling [5], [20]. Each vertex $v \in \mathcal{V}$ represents a state, or pose of the vehicle, while each edge $e \in \mathcal{E}$ encodes a motion which respects its non-holonomic constraints. Here, we focus on kinematic constraints, as in our system the generation of a velocity profile is decoupled from the maneuver calculated by the motion planner. The reason for this is two-fold: First, removing the velocity at planning time we effectively reduce the size of the state space, thus speeding up the search for solutions. Also, the speed profiling can then be better tailored to the requirements of the low-level controller. Second, calculating the velocity using a dedicated module, we can easily take into account dynamic obstacles and avoid them by adapting the cruising speed without re-planning.

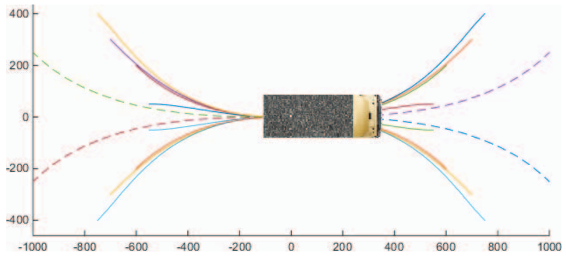


Fig. 2: The motion primitives in the state $s = \langle 0, 0, \pi/2, 0 \rangle$. In this model, $r = 50$ cm, $|\Theta| = 16$ and $|\Phi| = 1$.

In a motion planning problem, a vehicle is fully specified by its *model*. A model encodes the geometric measurements of the vehicle, the discretization of the dimensions of the lattice on which the vehicle moves and a set of *motion primitives* P. The discretization of the lattice defines what states the vehicle can reach. A valid state for a model is represented by a four-dimensional vector $s = \langle x, y, \theta, \phi \rangle$: (x, y) lies on a grid of resolution r, $\theta \in \Theta$ and $\phi \in \Phi$, where Θ and Φ are a finite set of allowed orientations and of allowed steering angles, respectively. The set of motion primitives P captures the mobility of the vehicle while intrinsically taking into account its kinematic constraints. Under the assumption of even terrain, we can design P to be position-invariant.[1] Every $p \in P$ is calculated by using a *boundary value problem* (BVP) solver to connect a set of initial states $s = \langle 0, 0, \theta, \phi \rangle$ to a set of neighboring states in a discrete, bounded neighborhood in free space. The BVP solver guarantees that the motions respect the kinematic constraints of the vehicle, while the position-invariant property ensures that the primitives are translatable to other states. P can then be reduced for efficiency using the techniques described in [21], by removing those primitives that can be decomposed into other primitives in P, without affecting the reachability of the state space of the vehicle when obstacles are not considered. Finally, a cost $g(p)$ is associated with each $p \in P$. In our implementation, $g(p)$ is calculated by multiplying the distance covered by p by a cost factor which penalizes backwards and turning motions. An example of motion primitives for the vehicle model of a truck can be seen in Fig. 2, where $r = 50$ cm, $|\Theta| = 16$ and $|\Phi| = 1$. The figure represents all the primitives applicable

[1]This assumption can be relaxed if the low-level controller of the vehicle can absorb minor perturbations or by means of a post-processing step.

in the starting state $s = \langle 0, 0, \pi/2, 0\rangle$.

A planning problem is defined by a starting state $start$, a goal state $goal$ and a world representation $\mathcal{W}$, in which are included all known obstacles. A *valid solution* is a sequence of collision-free primitives $(p_0, \dots, p_n)$ connecting $start$ to $goal$. Given the set of all valid solutions to a problem, an optimal solution is the one with minimum cost.

IV. TIME-BOUNDED A*

Time-Bounded A^* (TBA^*) was introduced in [6] and was designed for efficient path finding on grids. The most prominent features of the algorithm (described below in pseudo-code as in the original publication) are that it achieves real-time operation, it allows to interleave search periods with action execution, and it avoids many unnecessary state re-expansions compared to other solutions. These characteristics, and the fact that TBA^* maintains completeness, make the algorithm a great choice for gaming applications.

Procedure TBA^* (*start,goal,*$\mathcal{W}$)

```
1  $solutionFound \leftarrow false$
2  $solutionFoundAndTraced \leftarrow false$
3  $traceDone \leftarrow false$
4  $loc \leftarrow start$
5  while $loc \neq goal$ do
6      if ($\neg solutionFound$) then
7          $solutionFound \leftarrow A^*(lists, start, goal, \mathcal{W}, N_E)$
8      if ($\neg solutionFoundAndTraced$) then
9          if ($doneTrace$) then
10             $pathNew \leftarrow lists.mostPromisingState()$
11         $traceDone \leftarrow traceBack(pathNew, loc, N_T)$
12         if ($doneTrace$) then
13             $pathFollow \leftarrow pathNew$
14             if ($pathFollow.back() = goal$) then
15                 $solutionFoundAndTraced \leftarrow true$
16     if ($pathFollow.contains(loc)$) then
17         $loc \leftarrow pathFollow.popFront()$
18     else
19         if ($loc \neq start$) then
20             $loc \leftarrow lists.stepBack(loc)$
21         else
22             $loc \leftarrow loc\_last$
23     $loc\_last \leftarrow loc$
24     move agent to loc
```

Given an initial state $start$, a desired final state $goal$ and a representation of the world $\mathcal{W}$, TBA^* searches the state space as A^* would do. However, while the latter would continue until it finds a complete path from *start* to *goal*, TBA^* stops the search after a finite number N_E of expansions (line 7), while retaining the open and the closed lists. In case a solution has not yet been found after N_E expansions, the algorithm extracts from the open list the most promising state, tracing it back ($pathNew$) either to *start*, or to the current location of the agent loc, in case the agent is already on $pathNew$. Note that also the tracing operation is done in a time sliced manner, and only N_T steps are traced at each iteration (line 11). When tracing is done, $pathNew$ becomes the path to follow ($pathFollow$, line 13) and the algorithm executes sequentially its actions (line 17). However, the agent might not be on $pathFollow$, but on another path that was extracted as most promising during a previous iteration. In such case it will have to backtrack its steps (line 20) to reach the state where the two paths meet (*start*, in the worst case). Move actions (lines 24) are performed one per iteration, while expansion and tracing steps (N_E and N_T) are fixed in number at each algorithm's iteration. The authors also considered the special case in which the agent has reached *start*, but no new path is available. Here, the agent is forced to act, and it moves back to the state it came from (loc_last line 22)

Conceptually, it would be straightforward to modify the algorithm to explore a lattice, rather than a grid. However, robotic systems are not equivalent to videogame agents, and domain-specific adaptations are required.

V. LATTICE TIME-BOUNDED A*

Moving from videogames to autonomous vehicles, there are many aspects of TBA^* which require adjusting. The resulting new algorithm, *Lattice Time-Bounded* A^* ($LTBA^*$), is summarized in pseudo-code below. $LTBA^*$ maintains the operational principles of TBA^* and works in a time-sliced manner, where A^* is repeatedly called at each iteration with the same lists (line 12). However, this new algorithm takes into account that we are planning for a physical system, which cannot be safely steered on a new path without considering its velocity and inertia, and which should not exhibit erratic behaviours. Moreover, the algorithm accounts for the fact that new goals could arrive during execution, and that other systems (such as the low-level controller) may fail, or steer the vehicle out of its intended path. Finally, new sensor data arrive at every iteration, and new obstacles may be perceived. Here, we detail all the major differences between TBA^* and its adaptation for robotic systems.

Procedure $LTBA^*$ (*goal,* $\mathcal{W}$)

```
1  $loc \leftarrow$ getSensorData()
2  $start \leftarrow$ createState($loc$)
3  $lastState, committedState \leftarrow start$
4  $t \leftarrow$ now()
5  while $loc \neq goal$ do
6      $(currentGoal, loc, \mathcal{W}) \leftarrow$ getSensorData()
7      if ($currentGoal \neq goal$) $\vee$ outOfPath($loc$) then
8          $return$
9      $(lastState, committedState) \leftarrow$ trackLoc($loc$)
10     if $committedState \neq startState$ then
11         updateLists($committedState$)
12     $solutionFound \leftarrow A^*(lists, start, goal, \mathcal{W}, \Delta T - t)$
13     $t \leftarrow$ now()
14     $pathNew \leftarrow lists.mostPromisingState()$
15     if $pathNew.size() > 0$ then
16         sendPath($pathNew$)
17     else
18         sendPath($lastState$)
```

1) Continuous sensor update: The first difference between $LTBA^*$ and TBA^* lies in the distinction between *states*, that are a discrete representation used for exploring the lattice space, and the status of the vehicle, contained in the variable *loc*. *loc* does not only contain the position of the vehicle in the continuous space, but it also includes

```
Procedure mostPromisingState
1  if $solutionFound$ then
2      $pathNew \leftarrow solution$
3  else
4      $pathNew \leftarrow openList.pop()$
5  while $\neg$ collFree($pathNew$) $\wedge$ $openList.size() > 0$ do
6      removeCollisionStates($lists, pathNew$)
7      $pathNew \leftarrow openList.pop()$
8  if collFree($pathNew$) then
9      return($pathNew$)
10 else
11     $pathNew \leftarrow \{\}$
12     return($pathNew$)
```

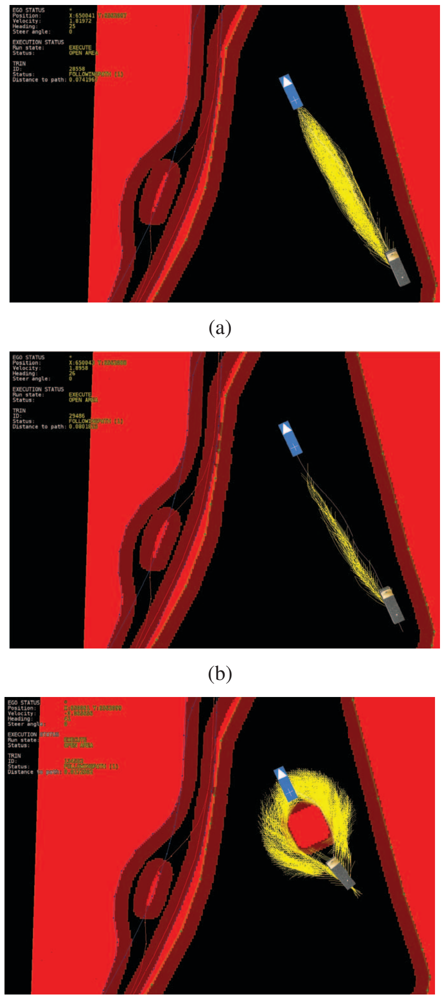
information such as current speed and mass.[2] Because of this difference, the starting state of the vehicle is not passed as an argument, but it is inferred directly from sensor data, so that $start$ reflects where the vehicle is expected to be by the end of the first planning cycle (lines 1-2).

Sensor updates are repeated at each iteration of the algorithm (line 6). New readings are processed for updating the representation of the world $\mathcal{W}$ (e.g., the position of detected obstacles), the location of the truck loc and to verify if the current goal has changed. If, for any reason, the truck is outside its designed path, or in case a new $goal$ is received, the procedure terminates and a new one is immediately instantiated with a new $start$ (lines 7-8).

2) Path selection and path commitment: After each planning iteration (line 12), $pathNew$ is updated as it was in TBA^*. However, here, the procedure which selects the most promising state is more complex and it is detailed in pseudo code in Procedure *mostPromisingState*. As it is customary in robotic applications, the planner works under the free world assumption, which means that the area outside the range of the truck's sensors, with the exception of fixed infrastructure, is considered as obstacle-free. Hence, it may happen that the extracted $pathNew$ is invalidated once a new obstacle enters within the sensors' range (see example in Fig. 3).

The selection of $pathNew$ works as follows: First, the algorithm checks if A^*, during the last search iteration, has reached a solution, which would obviously become the first candidate for $pathNew$. In case a solution does not exist, the most promising node is extracted from the $openList$ (Procedure *mostPromisingState*, lines 1-4). Once a candidate has been selected, it is checked for obstacles. In case the result of the check is positive, the edge of the lattice graph on which the first collision point occurs is identified and removed, along with all its successors. Such states, now unreachable, are deleted from the lists (line 6). New candidates are then evaluated until one of two possibilities occurs: Either a candidate $pathNew$ is collision-free, or the $openList$ is empty. In the latter case, an empty $pathNew$ is returned. $pathNew$ is then sent for execution ($LTBA^*$, line 16). In the unlikely case that $pathNew$ is empty, the algorithm sends the position contained in $lastState$ for execution. If the truck is moving, this would cause an emergency braking procedure.

[2]Information about the mass could be included into the vehicle's model. However, here we allow for variations on the truck's load.



(a)

(b)

(c)

Fig. 3: The truck moves from its current position (lower right) to $goal$ (blue). The lattice is explored (a), and a $pathNew$ selected. In successive search iterations, $LTBA^*$ finds a solution (white), and discards the branches that stem from behind the $committedState$ (b). When an obstacle is detected (c), the current $pathNew$ is abandoned, and the lattice is explored for alternative solutions.

Note that this mechanism was never triggered in the course of our experiments, both simulated and with the real truck, but it is nevertheless necessary to ensure overall system safety.

Finally, dealing with a heavy vehicle traveling at considerable speed entails that sudden stops and changes of direction are not acceptable. Nor we can allow for the truck to trace back a dismissed path in reverse. Hence, the $traceBack$ function in TBA^* is replaced by a mechanism to ensure that such occurrences never happen: At each cycle, we use the information about position and speed of the truck on the $pathNew$ to calculate both the $lastState$ visited and the $committedState$ (line 9). The $committedState$ is the state that lies on $pathNew$ before which the truck, at its current speed, is not allowed to deviate from $pathNew$. The calculation of $committedState$ takes into account two factors: The position at which the truck is predicted to be at the end of the current planning cycle and the momentum of the truck, so as to avoid infeasible maneuvers. Once calculated, $committedState$ becomes the new root of the search,

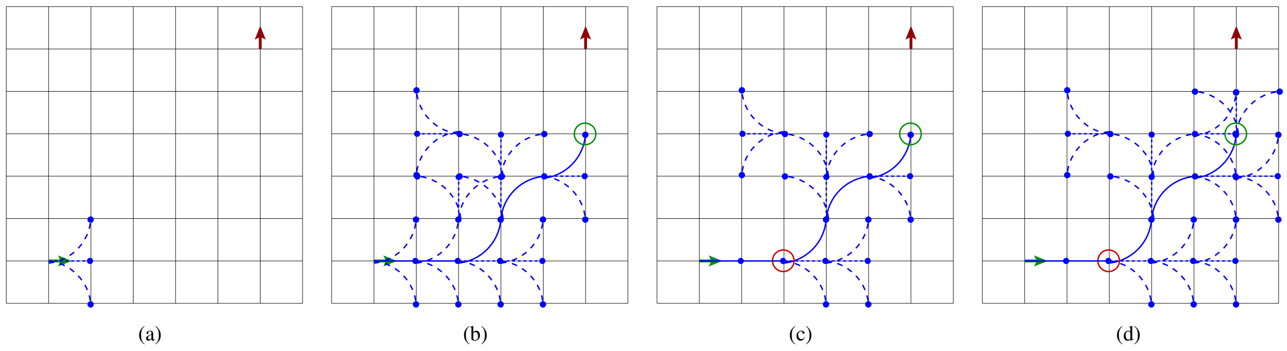


Fig. 4: A simplified example of how the explored lattice is pruned after the algorithm has selected a $committedState$. (a): A vehicle, captured by a simple model with three motion primitives, must move from $start$ (green arrow, bottom left corner) to $goal$ (red arrow, top right); (b): After the first search iteration, the algorithm selects a $pathNew$, represented by a continuous line and terminating at the state circled in green, which is sent for execution; (c): Before the next search iteration, the algorithm commits to a $committedState$ (circled in red), where the truck is going to be after the search episode. The lattice is pruned, so that no further exploration is possible between $start$ and $committedState$; (d): The next search iteration further explores the lattice starting from the states expanded in previous searches.

as the states that branch from $start$ to $committedState$ are removed, thus forcing the search to continue from $committedState$ onwards. A simplified example of this procedure can be seen in Fig. 4, while a real test case is shown in Fig. 3. Note that the discarded states can be reached again, but only by first passing through $committedState$. This last step ensures that the next $pathNew$ will share its first segment with the previous one.

3) Real-time execution: Maintaining real-time performance in $LTBA^*$ is of the utmost importance. As the algorithm needs to send a collision-free $pathNew$ for execution every fixed ΔT (in our implementation, $\Delta T = 0.5$ seconds), we need to make sure that the time allotted for search takes into account other possibly time-consuming procedures, such as the collision checking and the selection of $pathNew$. Also, when dealing with lattice state, the time required for each expansion may greatly vary, as collision checking is more complex than on a grid. For this reason, we prefer to put a hard limit on the *execution time* of the search phase, rather than on the number of expansions as originally done in TBA^*. Thus, before the main loop begins (line 4) and right after the search on the lattice space (line 13), a time t variable is reset. A^* is called as the last operation in each time cycle, so that we can calculate exactly how much time is available for exploring the lattice.

A. Notes on completeness and complexity

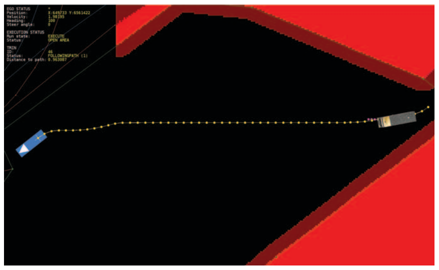
Lattice-based motion planners can be complete with respect to the discretization of the state lattice and the selection of the motion primitives [22], provided that the algorithm used to search the lattice is complete. $LTBA^*$ relies on a time-sliced A^* search, which is per se complete. However, contrarily to TBA^*, the new algorithm does not expand all the nodes as A^* would do, as some branches are pruned away because of the mechanism of the $committedState$. To ensure completeness, it would be required to modify the algorithm in two ways: (1) whenever a solution is not found within a number of search cycles, the truck is required to reduce speed or even to stop, so as to give enough time to complete a full A^* search from the current $committedState$; and (2) the set P of motion primitives should be designed to have symmetric forward and backward maneuvers.

The memory complexity of $LTBA^*$ is the same of A^*: in the worst-case, the algorithm would require to explore the entire state space. As the state space can be very large, we rely on two heuristic functions to direct the search: A simple euclidean distance and a state-to-state heuristic table with exact costs in free space. This last heuristic has been already successfully used in [5] and it is calculated by running Dijkstra's algorithm starting from a few selected starting states. Both heuristics are consistent, and we can combine them so that the resulting function is also consistent.

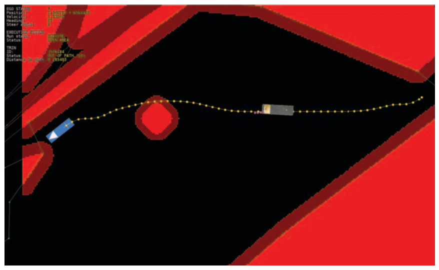
VI. Experimental Results

We tested our planner both in simulation and on our robotic platform (Fig. 1). The simulation duplicates all the sub-systems of the real platform, from behaviour selection to the low level controller. The physical truck is equipped with different sensors for obstacle detection and with an RTK GPS for precise localization. The simulation, on the other hand, assumes perfect sensing and localization. All the components of the simulation run on a virtual machine with Ubuntu OS, which in turn is running on a normal laptop equipped with an Intel Core i7-4810QM CPU @ 2.80GHz and 16 GB RAM (8 GB available to the virtual machine). The computer running the components on the robotic platform is Debian-based, and with comparable hardware specifications. The tests were carried out with two truck models: one with $|\Theta| = 16$, $|\Phi| = 1$ and $|P| = 192$, the second with $|\Theta| = 32$, $|\Phi| = 1$ and $|P| = 1312$ ($r = 50$ cm in both cases). The second model allows for more precise maneuvering, but its state space is larger. Both models were used with a heuristic table for speeding up close-quarters maneuvering.

We intensively tested $LTBA^*$ for several weeks on the robotic platform, and the behavior of the planner was the same observed in simulation. We could then use the simulation to perform systematic tests. The first tests included parking scenarios (Fig. 6), point-to-point movements and evasive maneuvers when new obstacles appear on the map unexpectedly during execution (Fig. 3). Here, in particular, the truck has to move from one position to another



(a)



(b)

Fig. 5: $LTBA^*$ can efficiently cope with new obstacles detected along the path. As the initially calculated path (a) is not viable any longer, the algorithm resume the exploration of the lattice, to find an alternative, collision-free route (b).

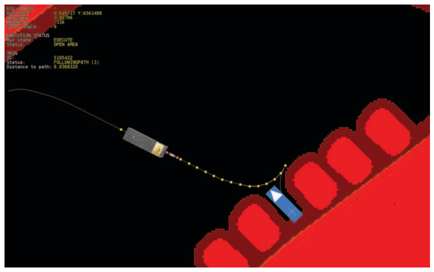


Fig. 6: The planner quickly calculates parking maneuvers.

(Fig. 3(a)). While interleaving search and execution, and updating the $committedState$, the planner finds a path to the goal (Fig. 3(b)). When a new obstacle is detected, the exploration of the lattice is resumed to find safe alternative solutions (Fig. 3(c)). All tests were successful: the low level controller was able to execute all the maneuvers planned and the obstacles were correctly avoided. The second set of tests demonstrated that $LTBA^*$ can cope with new obstacles better than A^*. We designed 5 similar scenarios, where the planner is invoked with fixed $start$ and $goal$. After 10 seconds, an obstacle appears between the truck and its destination (Fig. 5). We run the scenarios first using $LTBA^*$ and then A^*. In the second case, the planner had to find a complete solution before committing to a path, and to start from scratch when the solution was invalidated. Here, not only our algorithm expanded less nodes than A^* ($LTBA^*$: avg 6017 [max 8505] ; A^*: avg 10473 [max 11850]), but it never required abrupt braking for calculating a new path.

VII. Conclusions and Future Work

In this paper we have shown how motion planning for robotic systems can greatly benefit from the latest findings in the area of path finding on grids. We adapted TBA^*, an algorithm designed for videogames, to lattice-based motion planning. The new algorithm, $LTBA^*$, was described, analyzed and tested in simulation and on an autonomous truck. Future work include a comparison with other search algorithms and the extension to multi-robot systems.

References

[1] P. Ross, "Robot, you can drive my car," *IEEE Spectrum*, vol. 51, no. 6, pp. 60–90, 2014.

[2] L. Thrybom, J. Neander, E. Hansen, and K. Landernas, "Future challenges of positioning in underground mines," *IFAC-PapersOnLine*, vol. 48, no. 10, pp. 222–226, 2015.

[3] J. Marshall, T. Barfoot, and J. Larsson, "Autonomous underground tramming for center-articulated vehicles," *Journal of Field Robotics*, vol. 25, no. 6-7, pp. 400–421, 2008.

[4] S. M. LaValle, *Planning Algorithms*. Cambridge, U.K.: Cambridge University Press, 2006.

[5] M. Pivtoraiko, R. A. Knepper, and A. Kelly, "Differentially constrained mobile robot motion planning in state lattices," *Journal of Field Robotics*, vol. 26, no. 3, pp. 308–333, 2009.

[6] Y. Björnsson, V. Bulitko, and N. R. Sturtevant, "TBA*: Time-bounded A*," in *Proc. of the 21st Int. Joint Conf. on Artificial Intelligence (IJCAI)*, 2009.

[7] J. Levinson, J. Askeland, J. Becker, J. Dolson, D. Held, S. Kammel, J. Z. Kolter, D. Langer, O. Pink, V. Pratt, M. Sokolsky, G. Stanek, D. Stavens, A. Teichman, M. Werling, and S. Thrun, "Towards fully autonomous driving: systems and algorithms," in *Proc. of the IEEE Intelligent Vehicles Symposium (IV)*, 2011.

[8] L. Kavraki, P. Svestka, J. Latombe, and M. Overmars, "Probabilistic roadmaps for path planning in high-dimensional configuration spaces," *IEEE Trans. on Robotics and Autom.*, vol. 12, no. 4, 1996.

[9] S. M. LaValle, "Rapidly-exploring random trees: A new tool for path planning," Oct. 1998, TR 98-11, Iowa State University.

[10] M. Pivtoraiko and A. Kelly, "Fast and feasible deliberative motion planner for dynamic environments," in *Proc. of the ICRA Workshop on Safe Navigation in Open and Dynamic Environments: Application to Autonomous Vehicles*, 2009.

[11] S. M. LaValle and J. J. Kuffner, "Rapidly-exploring random trees: Progress and prospects," in *Algorithmic and Computational Robotics: New Directions*, B. R. Donald, K. M. Lynch, and D. Rus, Eds. Wellesley, MA: A K Peters, 2001, pp. 293–308.

[12] Y. Kuwata, S. Karaman, J. Teo, M. Frazzoli, J. P. How, and G. Fiore, "Real-time motion planning with applications to autonomous urban driving," *IEEE Transactions on Control Systems Technology*, vol. 17, no. 5, pp. 1105–1118, 2009.

[13] S. Karaman and M. Frazzoli, "Sampling-based optimal motion planning for non-holonomic dynamical systems," in *Proc. of the IEEE Int. Conf. on Robotics and Autom. (ICRA)*, 2013.

[14] M. Likhachev, G. Gordon, and S. Thrun, "ARA*: Anytime A* with provable bounds on sub-optimality," *Advances in Neural Information Processing Systems*, vol. 16, 2003.

[15] S. Koenig and M. Likhachev, "D* lite," in *Proc. of the National Conf. on Artificial Antelligence (AAAI)*, 2002.

[16] H. Andreasson, J. Saarinen, M. Cirillo, T. Stoyanov, and A. J. Lilienthal, "Fast, continuous state path smoothing to improve navigation accuracy," in *IEEE Int. Conf. on Robotics and Autom. (ICRA)*, 2015.

[17] N. Sturtevant, J. Traish, J. Tulip, T. Uras, S. Koenig, B. Strasser, A. Botea, D. Harabor, and S. Rabin, "The grid-based path planning competition: 2014 entries and results," in *Eighth Annual Symposium on Combinatorial Search*, 2015.

[18] N. Sturtevant, "An introduction to search for games," in *Game AI Pro 2: Collected Wisdom of Game AI Professionals*. CRC Press, 2015.

[19] S. Koenig and M. Likhachev, "Real-time adaptive A*," in *Proc. of the 5th Int. Joint Conf. on Auton. Agents and Multiagent Systems (AAMAS)*, 2006.

[20] M. Cirillo, T. Uras, and S. Koenig, "A lattice-based approach to multi-robot motion planning for non-holonomic vehicles," in *Proc. of the IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)*, 2014.

[21] M. Pivtoraiko and A. Kelly, "Kinodynamic motion planning with state lattice motion primitives," in *Proc. of the IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)*, 2011.

[22] M. Cirillo, T. Uras, S. Koenig, H. Andreasson, and F. Pecora, "Integrated motion planning and coordination for industrial vehicles," in *Proc. of the 24th Int. Conf. on Automated Planning and Scheduling (ICAPS)*, 2014.