Fast-Replanning Motion Control with Short-Term Aborting A*

Marcell Missura Arindam Roychoudhury Maren Bennewitz

Abstract—Autonomously driving vehicles must be able to navigate in dynamic and unpredictable environments in a collision-free manner. So far, this has only been partially achieved in driverless cars and warehouse installations where marked structures such as roads, lanes, and traffic signs simplify the motion planning and collision avoidance problem. We are presenting a new control framework for car-like vehicles that is suitable for virtually any environment. It is based on an unprecedentedly fast-paced A* implementation that allows the control cycle to run at a frequency of 33 Hz. Due to an efficient heuristic consisting of rotate-translate-rotate motions laid out along the shortest path to the target, our Short Term Aborting A* (STAA*) can be aborted early in order to maintain a high and steady control rate. This enables us to place our STAA* algorithm as a low-level replanning controller that is well suited for navigation and collision avoidance in dynamic environments. While our STAA* expands states along the shortest path, it takes care of collision checking with the environment including predicted future states of moving obstacles, and returns the best solution found when the computation time runs out. Despite the bounded computation time, our STAA* does not get trapped in environmental minima due to the following of the shortest path. In simulated experiments, we demonstrate that our control approach is superior to an improved version of the Dynamic Window Approach with predictive collision avoidance capabilities [1].

I. INTRODUCTION

Unstructured dynamic environments as for example pedestrian zones or the corridors of an office building, where humans and other robots move freely without following official rules, pose a particular hard challenge for the design of reliable motion controllers that are effective at avoiding collisions and reaching goal locations. Such environments are unpredictable to a large extent and the assumptions made by model predictive control approaches hold only for a short amount of time at best. Fast replanning is the only way of reacting to unforeseen events such as the sudden change of direction of a nearby agent, or the appearance of an obstacle in the sensor range that is already on collision course. Fast replanning, however, also implies short computation times that do not suffice for the computation of a detailed plan all the way to the target, which in return can result in oscillatory motions and imprisonment in local minima. Due to this reason, two-tiered navigation controllers emerged that compute a high-level plan at a slow rate all the way to the target (e.g., the shortest path), and follow the plan with a short-sighted, low-level controller that cares for the path following and collision avoidance at a high frequency. The work of Willow Garage [2] constitutes an extensive test of this control paradigm where a vanilla A* search was used to find the shortest path in an occupancy map and an intermediate target along this path serves as a local goal for the Dynamic Window Approach (DWA) [3] that follows the path and avoids collisions.

Our control framework is two-tiered in the same fashion. We use the shortest path to guide the navigation system and extract from it a short-term target for the lower layer. On the lower layer, we use a bounded-time A* implementation that enables us to run it in a replanning control cycle at a frequency of 33 Hz. The A* algorithm applies a discrete set of acceleration commands for a short time to the current state of the nonholonomic agent and predicts their outcome including the expected motion of other moving agents. From those predicted states, it chooses the most promising one according to collision considerations and a heuristic function. It repeatedly expands the most promising state until either the target state has been reached, or the allowed computation time has run out. Our A* implementation takes special care for the early aborting of the algorithm by using an obstacle-aware heuristic function that includes the computation of the shortest path for every opened state. Expanding along the shortest path makes sure that aborting the search early does not trap the agent in a local minimum.

Furthermore, our system no longer relies on memory-consuming grid maps that can become prohibitively large. Instead, we assume the map of the environment to be a
obstacles move between updates. In all of the aforementioned optimization. While the path to begin with is collision free, quintic splines [9], B-splines [10], Bezier curves [11], or elastic bands [12] are used in a manner of online fly. Quintic splines [9], B-splines [10], Bezier curves [11], or elastic bands [12] are used in a manner of online optimization. While the path to begin with is collision free, the refined trajectory is not necessary so, especially not if the obstacles move between updates. In all of the aforementioned methods moving obstacles are assumed to be stationary.

Recently, several approaches to learning a controller have been presented. Pfeiffer et al. proposed to train an end-to-end deep network motion planner for small, static maps using the raw sensor data as input, the motor controls as output, and a ROS path planner as the expert teacher [13]. Chen, Everett et al. [14], [15] trained a deep network to navigate among moving obstacles. In order to process an undetermined number of obstacles, an LSTM layer was used to convert an arbitrary number of observed agents to a fixed size input for the network. The obstacles were represented as circular objects encoded by their position, velocity, and radius. This setup is not well suited to encode the large, nonconvex structures of indoor environments. Kretzschmar et al. [16] used inverse reinforcement learning to identify the parameters of human trajectories that were modeled by cubic splines to achieve a predictable system with human-like behavior.

Another approach to obstacle avoidance is based on the concept of velocity obstacles [17]. Velocity obstacles are geometric regions in velocity space that describe the set of velocities an agent A is not permitted to use, otherwise it will eventually collide with agent B, if both agents continue to travel with the same velocity. Assuming two or more agents are using the same control principle and choose a velocity for the next time step outside of their respective velocity obstacles in a reciprocal fashion [18], a simple linear program can be used to guarantee collision-free motion for n-bodies [19] as long as the modeling assumptions hold. Acceleration-velocity obstacles [20] and nonholonomic control [21] have also been investigated. Velocity obstacles assume circular or convex polygonal obstacles that travel at constant velocity and use the same controller. Our concept can deal with nonconvex polygons of arbitrary size and does not make the assumption of a certain type of controller being used by the other agents. Furthermore, as it is possible to enter and leave a velocity obstacle before the collision actually occurs, our A* approach provides for more precise collision checking than the velocity obstacle concept is able to. We also predict the state of other agents based on the assumption that their observed velocities remain constant for a short time, but we predict the state of the controlled agent using the planned controls and perform precise polygon vs polygon collision checking with the predicted states.

III. MOTION CONTROL FRAMEWORK

Our motion control framework is targeted to control nonholonomic vehicles. It computes acceleration commands at a high control rate using a Short Term Aborting A* (STAA*) motion planner. Our framework involves a number of components such as the environment representation, a bounded planning area, a unicycle model for motion prediction, collision checking, and cost and heuristic functions that determine the order of the STAA* node expansion. In the following, we first describe the setting of a larger navigation and obstacle avoidance system our framework is embedded in, introduce a mixture of polygonal and grid maps
we use as environment representation, and visit the working components of the STAA* algorithm.

A. Prerequisites

Our motion control framework is placed in a larger robot operating system where a global polygonal map $\mathcal{M}$ and a pose estimate in the map are given. Our framework takes as input the current pose of the robot $S = (x_S, y_S, \theta_S)$ and a goal pose $G = (x_G, y_G, \theta_G)$ in the global reference frame where $x$ and $y$ are the Cartesian coordinates and $\theta$ is the orientation of the robot.

B. Path Planning

We use an inflated copy $\mathcal{M}'$ of the map for the computation of the global path $P(S, G) = \{(x_i, y_i) \mid i = 0..k\}$ where $(x_0, y_0) = (x_S, y_S)$ and $(x_k, y_k) = (x_G, y_G)$. The inflation of the map by the radius of the agent is applied to avoid planning through too narrow passages the agent would not fit through. To calculate $P(S, G)$ in $\mathcal{M}'$, we use the Minimal Construct Algorithm [22]. We first compute a path regarding only the static polygons in the global map. Then, we compute a dynamic path using the polygons of the map and the polygons of dynamic obstacles in the sensor field that are not in the map. If the dynamic path is successfully found, we use the dynamic path for $P(S, G)$. In cases where a moving agent blocks a narrow passage such as a doorway and no dynamic path can be found, we fall back and use the path regarding the static obstacles only for $P(S, G)$. At this point, we do not make any predictions. Only the currently observed state of dynamic obstacles is used for global path planning, but the path is replanned in every cycle.

C. Planning in a Local Map

The STAA* motion planner operates in a bounded local map as shown in Fig. 1. In our current implementation, the local map is an 8m x 8m square that extends four meters to the left and right, six meters to the front, and two meters to the back of the robot. The local map bounds the complexity of the environment and has a strong impact on the computation times that can be achieved. Motion planning including shortest path computations during the node expansion of A* take place only within the local map. Outside of the local map, the global path $P$ completes the plan to the target.

To determine an intermediate goal pose $\tilde{G}$ that serves as a target for the planning within the local map, we intersect the global path from $S$ to $G$ with the local map, i.e., we follow $P$ from the start $S$ towards the goal $G$ until the first intersection with the boundary of the local map is found, or the goal is reached. Thus, the intermediate goal $\tilde{G}$ is located either on the boundary of the local map, or inside the local map if the entire path lies inside in which case $\tilde{G} = G$.

D. Local Map Representations

The local map consists of a mixture of grid and geometric structures used for either collision checking or path planning. Using this hybrid representation scheme, we can take advantage of the best of both, grid and polygonal maps.

The grid structure of the local map is an occupancy grid with a size of 160x160 and a resolution of 5 cm. We keep two such grids where one is a dilated version of the other. The first grid we compute by transforming the polygons of the global map $\mathcal{M}$ into local coordinates using the pose $S$ and drawing the polygons onto the grid using the drawContours function in OpenCV. We refer to this grid as the static grid $S$. $S$ is used as a lookup table for fast collision checking during the STAA* search. Then, we dilate and blur $S$ to an inflated and smoothed costmap $S'$, which is used for the cost function of the STAA*.

Then, we convert regions of occupied cells of the dilated static grid $S'$ to a map of nonconvex polygons $\mathcal{P}$ by means of contour detection. The polygon extraction operations are carried out according to our polygonal perception pipeline [23]. $\mathcal{P}$ is used for computing a large number of shortest paths for the heuristic of the STAA* search.

The detection of dynamic obstacles begins with projecting the point cloud acquired by an assumed RGB-D sensor into the local map and marking the occupied cells to create a
respectively. We take \( b \) of the local grid with respect to the world and the STAA* planner is the acceleration for the robot to apply until the next computational efficiency of the motion planner. The input of and a fast collision checking routine further contribute to the early abort. A simple trajectory approximation using arcs search, even though it has been specifically tailored for an E. Short-Term Aborting A*

The centerpiece of our framework is a traditional A* search, even though it has been specifically tailored for an early abort. A simple trajectory approximation using arcs and a fast collision checking routine further contribute to the computational efficiency of the motion planner. The input of the STAA* is the start pose \( S \), the intermediate goal pose \( G \), the static grid maps \( S \) and \( S' \), the static polygon map \( P \), and the dynamic polygon maps \( D \) and \( D' \). The output of the planner is the acceleration for the robot to apply until the next control cycle. The start state \( S \) defines the transformation of the local grid with respect to the world and the STAA* operates in the coordinate system of the local grid. Let \( s = (x, y, \theta, v, \omega) \) be a unicycle state with coordinates \((x, y)\), orientation \(\theta\), linear velocity \(v\), and angular velocity \(\omega\). Then, the initial unicycle state of the search \( s_0 = (0, 0, 0, \omega_0) \) lies in the origin of the local map and contains the sensed velocities \(v_0\) and \(\omega_0\). The initial state is pushed into the priority queue to initialize the search.

1) Action Set and Prediction: The action set

\[
A = \left\{ \left( a_{\min} + \frac{i}{n-1} (a_{\max} - a_{\min}) \right), \left( b_{\min} + \frac{i}{n-1} (b_{\max} - b_{\min}) \right) \mid i, j = 0..n-1 \right\},
\]

of the STAA* consists of tuples \((a, b)\), where \(a\) is the acceleration of the linear velocity and \(b\) is the acceleration of the angular velocity. The action set is systematically sampled from the permitted ranges \([a_{\min}, a_{\max}]\) and \([b_{\min}, b_{\max}]\), respectively. We take \(n = 7\) samples from each range, so the action set has 49 members in total. Given a unicycle state \(s_k\), we predict the successor state \(s_{k+1} = P(s_k, (a, b))\) using

\[
s_{k+1} = \begin{pmatrix} x_k + r (\sin(\theta_k + \bar{t}\omega) - \sin(\theta_k)) \\ y_k - r (\cos(\theta_k + \bar{t}\omega) - \cos(\theta_k)) \\ \theta_k + \bar{t}\omega \\ v_k + \bar{t}a \\ \omega_k + \bar{t}b \end{pmatrix}
\]

where \(\bar{t} = 0.3\) s is a constant prediction time. The future position \((x_{k+1}, y_{k+1})\) is computed by an arc approximation similar to [3]. The radius \(r = \frac{v_k + 0.5\bar{t}a}{m_\omega + m_v\bar{t}\omega} \) of the arc is obtained from the linear and angular velocities in the middle of the prediction interval \(\bar{t}\).

2) A* Node Expansion and Closing: The A* algorithm proceeds by popping the topmost state \(s_k\) from the priority queue, with \(k\) being the depth of the state in the search graph, and applying each acceleration \((a, b)\) \(\in A\) in the action set to predict new states \(s_{k+1,i} = P(s_k, (a, b)_i)\) according to Eq. (2). In the following, we drop the index \(i\) and just refer to a successor state \(s_{k+1}\). Note that our state space is continuous and we refrain from using a state lattice [7], or snapping \(s_{k+1}\) to a grid. We designed the state space to be continuous and maintain a table for closing states where we look up \(s_k\) and ignore it if its cell is already closed. If not, we close the cell. We also ignore all \(s_{k+1}\) that fall into a closed cell. The closed table coincides with the local map and has a cell size of 5 cm and 0.1 radians for the orientation, respectively.

3) Collision Checking: After the closed check, the predicted states \(s_{k+1}\) are checked for collisions with static and dynamic obstacles. First, we can quickly discard states where any of the corners of the agent polygon, or its centroid, fall into an occupied cell in the static grid \(S\). Only for the remaining states, we perform a polygon vs polygon collision test between the agent polygon at \(s_{k+1}\) and the polygons of the dynamic obstacles \(D(t)\) in their predicted states at time \(t = (k + 1)\bar{t}\). Since the agent polygon and the dynamic obstacles are always convex, this collision check can be computed efficiently with the SAT algorithm. Collided states are discarded and not pushed into the priority queue. Because of this, the queue may run empty when all states collide, in which case the agent executes an emergency brake maneuver.

The computations of the cost function and the heuristic function are placed after the collision check, because the collision check may discard states for which we do not need to compute the cost and the heuristic.

4) Cost Function: After the collision check, we compute \(g(s_{k+1})\), the path cost so far, \(h(s_{k+1}, G)\), the estimated cost-to-go to the intermediate goal \(G\), and \(f(s_{k+1}) = g(s_{k+1}) + h(s_{k+1}, G)\), the value the states are ordered by in the priority queue. We express the costs

\[
g(s_{k+1}) = (k + 1)\bar{t} + w_s S'(s_{k+1}) + w_d \Delta(D(t), s_{k+1}),
\]

mostly in terms of time needed to reach depth \(k + 1\), but also add the weighted static proximity cost obtained from the dilated and blurred costmap \(S'\) at the location of \(s_{k+1}\), and the weighted proximity to the dynamic obstacles with \(\Delta(D(t), s_{k+1}) = \max(1 - d(D(t), s_{k+1})/0)\) and \(d(D(t), s_{k+1})\) the distance to the closest edge or vertex in
the predicted polygon set $D(t)$ at time $t = (k + 1)\bar{t}$ at the location of $s_{k+1}$.

5) **Heuristic Function:** For the computation of the heuristic, we determine the shortest path $P(s_{k+1}, \hat{G})$ for each state $s_{k+1}$ to the intermediate goal $\hat{G}$. The shortest path is calculated using the Minimal Construct algorithm [22] in $P \cup D'(t)$, the union of the static polygons $P$ and the predicted and expanded dynamic polygons $D'(t)$ at time $t = (k + 1)\bar{t}$. Note that we only compute predictions to a depth of 3. At search depths deeper than 3, we assume that the dynamic polygons have disappeared.

Similar to our previous work on footstep planning with aborting A* [24], we use the rotate-translate-rotate (RTR) function to estimate the time needed to drive along a path to the intermediate goal and to attain the goal direction as shown in Fig. 3. The RTR function is given by

$$RTR(s, \hat{G}) = \frac{|\angle(s, \hat{G}) - \angle(s) - \angle(\hat{G} - s)|}{\omega_{\max}} + \frac{|\hat{G} - \hat{s}|}{v_{\max}} + \frac{|\hat{G} - \hat{s}|}{\omega_{\max}}$$

(4)

where $s$ and $\hat{G}$ are the start state and the intermediate goal, $\angle(s, \hat{G})$ denotes the angle of the vector from the start location to the intermediate goal, and $\hat{G}$ is the orientation at the intermediate goal. Rotational angles are divided by the maximum angular velocity $\omega_{\max}$ and the driven distance $|\hat{G} - s|$ is divided by the maximum linear velocity $v_{\max}$ to estimate the needed time. In order to involve the shortest path in the heuristic function, we extend the RTR function to a path RTR function by concatenating RTR motions along the sections of the path. Let $P(s, \hat{G}) = \{(x_i, y_i, \theta_i) \mid i = 0..k\}$ be a path to the intermediate goal where $P_0 = s$ and $P_k = \hat{G}$ with orientations $\theta_i > 0 = \angle(P_{i-1}, P_i)$. Then, the estimated time to drive along path $P$,

$$h(s_k, \hat{G}) = \sum_{i=0}^{k-1} RTR(P_i, P_{i+1}),$$

(5)

is obtained by summing up the RTR functions of the sections of the path. The RTR function overestimates the time costs of states that can be reached by a combination of high velocities, e.g., $(v_{\max}, \omega_{\max})$. Nonetheless, using this heuristic helps the A* algorithm to converge quickly towards the goal and is thus well suited for an aborting search.

6) **Early Aborting:** After we have computed $g(s_{k+1})$, the path cost so far, $h(s_{k+1}, \hat{G})$, the estimated cost-to-go, and $f(s_{k+1}) = g(s_{k+1}) + h(s_{k+1}, \hat{G})$, the value the states are ordered by in the priority queue, the new state $s_{k+1}$ is pushed into the priority queue and the next state is popped for expansion. When popping a state from the queue, we check whether the computation time has run out. If this is the case, STAA* chooses the state with the smallest $h$ value found so far as the best solution. We chose the state with the lowest heuristic, because the last opened state can be anywhere, but the state with the lowest $h$ value is sure to reach closest to the target. If a popped state reaches a heuristic value $h(s_k, \hat{G}) < 0.1$, the search is finished before time and the popped state is returned as a solution from which parent pointers are followed back to the root of the search graph in order to return the first acceleration command.

IV. EXPERIMENTS

A. **Runtime Analysis**

For maximum realtime capability, we targeted the RGB-D sensor rate at 33 Hz. This leaves us up to 33 ms to compute a cycle. Fig. 5 shows a breakdown of the major computation steps of our motion planner. Our perception pipeline takes 7.5 ms in total. Most of this time is spent on sorting the 300K RGB-D points of the point cloud into an occupancy grid. Postprocessing the grid takes only a fraction of the perception time. The execution time of the STAA* shown in blue shades is taken up mostly by the computation of the heuristic function shown in purple.
control framework manages to uphold a steady frequency. The STAA* processes up to 20,000 states in the given time. All runtimes were measured on a laptop with an Intel® Core i7-6800K 6 x 3.40GHz CPU.

B. Quantitative Evaluation

We evaluated our nonholonomic controller in three different maps that are shown in Fig. 6. The first map is an office building with a size of 30x30 meters. The second map is the floor plan of an apartment with a size of 12x10 meters. The third map is a 20x20 meters cluttered environment with small objects between 3 and 20 cm randomly scattered on the floor. In each map, we compared the performance of three different controllers. We used a simple PD controller that follows a “carrot” a short distance in front of it along the dynamic path, our own DWA implementation with predictive collision avoidance capabilities [1], and our new Short Term Aborting A* (STAA*). The PD controller and the DWA are following the dynamic path that leads around moving obstacles and are pushed away from obstacles by a force field whenever they enter the inflation zone. The STAA* does not use a force field. Each controller was tested with up to five agents in the map at the same time with every agent running the same controller. The task of each agent was to reach a random sequence of goal locations marked with circles in Fig. 6. Each setting was repeated 20 times and ran for 5 minutes (10000 frames) each time so that we have more than an hour and a half driving time per agent in every map in each setting of up to five agents. We evaluate the controllers in terms of their score, which is the number of goal locations reached by one observed agent, and the number of collisions that the observed agent experienced. The results are shown in Fig. 7.

Notably, all three controllers reach similar scores on all three maps. This is because all three controllers exhaust the driving capabilities of the agents of up to $2 \frac{\pi}{3}$ linear velocity and up to $3 \frac{\pi}{3}$ angular velocity. The PD controller drives the fastest by closely following the shortest path. It can be seen in Fig. 7 that when there is only one agent present, the PD controller achieves a slightly higher score than the other controllers. However, the PD controller, and also the DWA
to some extent, reach this score at the expense of collisions. The PD controller is entirely oblivious to the other moving agents and the dynamic path leading around the other agents is the only effort this controller makes to avoid collisions. The DWA has some predictive capabilities and it shows that it can effectively avoid the majority of the collisions the PD controller does not. The STAA* clearly excels in collision avoidance as the total number of collisions produced is near zero on every map.

We provide a demonstration video of our algorithm performing in simulation.

V. CONCLUSIONS

In conclusion, we proposed a fast-paced control framework with an aborting A* motion planner for nonholonomic agents. The control framework uses a geometric representation for the global map and performs collision checks with predicted states of dynamic obstacles. The runtime of the control cycle is bounded so that the system can be used as a low-level replanning controller. The achievement of lower than 33 ms runtimes is due to the early aborting capability of our A* implementation, which is based on a path RTR heuristic function. The bounding of the planning area to a local map also plays a load-bearing role. In our simulated experiments, we were able to show a significant decrease of collisions compared to two other state-of-the-art nonholonomic controllers.

REFERENCES

[1] Marcell Missura and Maren Bennewitz. Predictive collision avoidance for the dynamic window approach. In Proc. of the IEEE Int. Conf. on Robotics & Automation (ICRA), 2019.
[2] E. Marder-Eppstein, E. Berger, T. Foote, B.P. Gerkey, and K. Konolige. The office marathon: Robust navigation in an indoor office environment. In Proc. of the IEEE Int. Conf. on Robotics & Automation (ICRA), 2010.
[3] D. Fox, W. Burgard, and S. Thrun. The dynamic window approach to collision avoidance. IEEE Robotics Automation Magazine, Mar 1997.
[4] Oliver Brock and Oussama Khatib. High-speed navigation using the global dynamic window approach. In In IEEE Int. Conf. on Robotics and Automation (ICRA), pages 341–346, 1999.
[5] P. Ögren and N. E. Leonard. A convergent dynamic window approach to obstacle avoidance. IEEE Transactions on Robotics, 21(2):188–195, April 2005.
[6] S. Koenig and M. Likhachev. D* Lite. In Proc. of the National Conference on Artificial Intelligence (AAAI), 2002.
[7] Maxim Likhachev and Dave Ferguson. Planning long dynamically feasible maneuvers for autonomous vehicles. The International Journal of Robotics Research, 28(8):933–945, 2009.
[8] M. Likhachev, D. Ferguson, G. Gordon, A. Stentz, and S. Thrun. Anytime search in dynamic graphs. Artificial Intelligence, 172(14):1613 – 1643, 2008.
[9] B. Lau, C. Sprunk, and W. Burgard. Kinodynamic motion planning for mobile robots using splines. In Proc. of the IEEE/RSJ Int. Conf. on Intelligent Robots & Systems (IROS), 2009.
[10] Vladislav C. Usenko, Lukas von Stumberg, Andrej Pangercic, and Daniel Cremers. Real-time trajectory replanning for mavs using uniform b-splines and 3d circular buffer. CoRR, abs/1703.01416, 2017.
[11] Sterling McLeod and Jing Xiao. Real-time adaptive non-holonomic motion planning in unforeseen dynamic environments. In IROS, pages 4692–4699. IEEE, 2016.
[12] C. Rösmann, F. Hoffmann, and T. Bertram. Kinodynamic trajectory optimization and control for car-like robots. In Proc. of the IEEE/RSJ Int. Conf. on Intelligent Robots & Systems (IROS), pages 5681–5686, Sept 2017.
[13] Mark Pfeiffer, Michael Schaeuble, Juan I. Nieto, Roland Siegwart, and Cesar Cadena. From perception to decision: A data-driven approach to end-to-end motion planning for autonomous ground robots. CoRR, abs/1609.07910, 2016.
[14] Yu Fan Chen, Michael Everett, Miao Liu, and Jonathan P. How. Socially aware motion planning with deep reinforcement learning. In IROS, pages 1343–1350. IEEE, 2017.
[15] Michael Everett, Yu Fan Chen, and Jonathan P. How. Motion planning among dynamic, decision-making agents with deep reinforcement learning. CoRR, abs/1805.01956, 2018.
[16] Henrik Kretzschmar, Markus Spies, Christoph Sprunk, and Wolfram Burgard. Socially compliant mobile robot navigation via inverse reinforcement learning. I. J. Robotics Res., 35(11):1289–1307, 2016.
[17] Paolo Fiorini and Zvi Shiller. Motion planning in dynamic environments using velocity obstacles. The International Journal of Robotics Research, 17(7):760–772, 1998.
[18] Jur van den Berg, Ming Lin, and Dinesh Manocha. Reciprocal velocity obstacles for real-time multi-agent navigation. In 2008 IEEE International Conference on Robotics and Automation, pages 1928–1935, 2008.
[19] Jur van den Berg, Stephen J. Guy, Ming Lin, and Dinesh Manocha. Reciprocal n-body collision avoidance. In Robotics Research, pages 3–19, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.
[20] Jur van den Berg, Jamie Snape, Stephen J. Guy, and Dinesh Manocha. Reciprocal collision avoidance with acceleration-velocity obstacles. In 2011 IEEE International Conference on Robotics and Automation, pages 3475–3482, 2011.
[21] Jamie Snape, Jur van den Berg, Stephen J. Guy, and Dinesh Manocha. Smooth and collision-free navigation for multiple robots under differential-drive constraints. In Proc. of the IEEE/RSJ Int. Conf. on Intelligent Robots & Systems (IROS), pages 4584–4589. IEEE, 2010.
[22] Marcell Missura, Daniel D. Lee, and Maren Bennewitz. Minimal construct: Efficient shortest path finding for mobile robots in polygonal maps. In Proc. of the IEEE/RSJ Int. Conf. on Intelligent Robots & Systems (IROS), 2020.
[23] Marcell Missura, Arindam Roychoudhury, and Maren Bennewitz. Polygonal perception for mobile robots. In Proc. of the IEEE/RSJ Int. Conf. on Intelligent Robots & Systems (IROS), 2020.
[24] M. Missura and M. Bennewitz. Fast footstep planning with aborting A*. In Proc. of the IEEE Int. Conf. on Robotics & Automation (ICRA), IEEE, 2021.

1Video: https://youtu.be/fJKfezcw1XI