Combined Task and Motion Planning under Partial Observability: An Optimization-Based Approach

Camille Phiquepal **Marc Toussaint**
Machine Learning & Robotic Lab, University of Stuttgart
{firstname.surname}@ipvs.uni-stuttgart.de

***Abstract*— We propose a novel approach to Combined Task and Motion Planning (TAMP) under partial observability. Previous optimization-based TAMP methods [1][2] compute optimal plans and paths assuming full observability. However, partial observability requires the solution to be a policy that reacts to the observations that the agent receives. We consider a formulation where observations introduce additional branching in the symbolic decision tree. The solution is now given by a reactive policy on the symbolic level together with a *path tree* that describes the branchings of optimal motion depending on the observations. Our method works in two stages: First, the symbolic policy is optimized using approximate path costs estimated from independent optimizations of trajectory pieces. Second, we fix the best symbolic policy and optimize a joint trajectory tree. We test our approach on object manipulation and autonomous driving examples. We also compare the algorithm's performance to a state-of-the-art TAMP planner in fully observable cases.**

I. Introduction

Robots must combine the ability to reason symbolically about discrete actions (task planning) and geometrically about the realization in the real world (motion planning). Integrated approaches are referred to in the literature as Task and Motion Planning (TAMP). With the exception of [4], current TAMP research assumes full observability. However partial observability is pervasive in many real world situations, e.g. when objects are hidden or partially hidden. If some objects to manipulate are inside containers, the robot has to explore its environment to perform its task. Self driving cars face the same problem when operating in the presence of other vehicles that limit the field of view of the ego vehicle.

In this paper we extend the Logic-Geometric Programming (LGP) approach [1][2] to handle partial observability. Under partial observability, policies need to be reactive due to observations. In this paper, rather than focussing on control noise as in stochastic optimal control, we focus on observations at discrete points in time that reveal essential information for the task, e.g., whether an object is in a container or not, or whether a car is approaching in the opposite lane. In this case, not only does the symbolic policy need to be reactive, but also the optimal motion needs to branch at observations. Computing an optimal motion plan now means to compute an optimal trajectory tree. Our TAMP solver works in two stages: First piece-wise optimization and then joint tree optimization. During the first stage, the trajectories of actions are optimized independently (piece-wise). In the second stage, we fix the best symbolic policy obtained in the first stage and optimize the joint trajectory tree that corresponds to this symbolic policy. That is, instead of optimizing the sequential motion of each action in isolation, the whole trajectory tree is optimized all at once.

II. Related Work

Concerning Combined Task and Motion Planning, a number of approaches [5][8][9] rely on the discretization of the configuration spaces or action/skeleton parameters to leverage CSP methods. Recently, Dantam and al. introduced an off-the-shelf open source TAMP framework [10] combining constraints-based task planning and sampling-based motion planning [11]. Prior work [1][2] states TAMP problems as an optimization problem. All these approaches assume full observability and plan a single sequence of actions together with a single path. To our knowledge, the system in [4] is the only other TAMP planner handling partial observability. A *Look* action is used to actively move the robot sensor to acquire information. This approach (Hierarchical Planning in the Now) interweaves planning with execution (in the now). Replanning is triggered once the robot ends up in a state not covered by the plan. In contrast, our approach aims at planning a full reactive policy from the starting state to the final state. In addition, we aim for smooth and locally optimal trajectories.

III. Problem Formulation

A. Overview

We formulate the problem in terms of a decision tree and, for every action edge in this tree, cost and constraint objectives on the continuous trajectory. The decision tree is in belief space, including action and observation branchings. To define a problem we first define the symbolic partially observable decision process that spans this tree. Second, we define the cost and constraints functions that define the optimization objectives for the continuous trajectory problem associated with transitioning through this tree. An optimal policy is then comprised of a reactive policy that transitions the tree depending on observations, and an optimal trajectory tree which, depending on observations, smoothly switches into different motion options. To our knowledge, this is the first approach to propose trajectory trees as optimal reactive motion plans in the case of partial observability.

Fig. 1: The ego vehicle (cyan) wants to overtake but cannot observe if a vehicle arrives in the opposite direction because of the truck.

B. Decision Process

We assume a decision process defined by a 7-tuple $(S, A, T, \Omega, O, G, \gamma)$, where γ is a discount factor and,

- S is a finite set of states
- A is a finite set of actions
- T is a set of transition probabilities between states
- Ω is a finite set of observations
- O is a set of conditional observation probabilities
- G is a set of goal conditions defined over the belief state space. A goal condition indicates if a decision tree node is terminal or not.

This is a POMDP except for the missing reward structure, which we replace by the associated trajectory optimization problems defined below. In addition, we have a goal condition over the belief space.

Given the initial belief b_0, the decision process spans a decision tree with two kinds of nodes:

- In *action nodes* the agent has to choose which action to take; action nodes have $|A|$ children.
- In *observation nodes* the agent receives an observation; observation nodes have $|\Omega|$ children.

Every node in the tree corresponds to a belief state computed with the standard belief update.

1) Example for a decision graph: Consider a car behind a truck. The car wishes to overtake but cannot see the opposite lane because of the truck, see Fig. 1. The car can take three actions: look into the opposite lane, overtake the truck, or continue to follow the truck. After looking into the lane (move slowly toward the center of the road), the car receives an observation (lane free or not). Fig. 2 is the decision graph of this problem. The blue node is the start belief state. The green nodes are terminal belief states. The observations are assumed to be valid only for one step (a free lane might become non-free, hence edges from 3 to 0 and 4 to 0).

2) Policy: A policy π is a mapping from belief states to actions. The thick edges in Fig. 2 represent a possible policy. The decision graph potentially contains cycles, however, in the following, we will only consider policies that are trees. In the case of full observability, the policy would boil down to a single sequence of actions. A policy reaching the goal condition, on the symbolic level, is a tree whose leafs are terminal belief states. We call this a candidate policy.

The observation model O, the initial belief state at the root node b_0, and the policy π define the prior probability of reaching any given node b in the decision tree. We will denote this probability by $p(b|\pi, b_0)$.

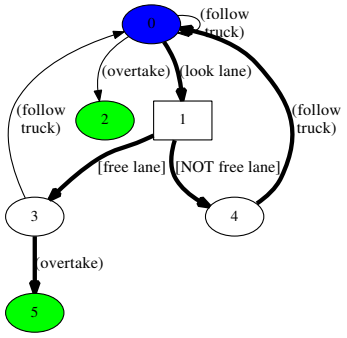


Fig. 2: Decision graph for overtaking, the tree of thick edges represent a possible policy

C. Trajectory Trees

The decision process described above defines sets of candidate policies that, on the symbolic level, reach goal conditions. As there are no rewards defined, all candidate policies are equal. However, on the geometric level, some candidate policies may be infeasible or lead to different path costs. We define here the path optimization problem associated with a candidate policy.

In typical trajectory optimization, the optimization objective is given as a set of constraints and sum of cost terms along the trajectory. In our setting, we generalize this to a set of constraints and sum of cost terms for each action edge in the tree, weighted by the probability of being in the corresponding belief.

More formally, let $\mathcal{X}$ be the configuration space of the whole environment, including the robot and all object configurations. Consider that the agent takes an action $a \in A$ at a belief node b. We assume this implies cost and constraint functions on the trajectory during the time interval $[t_k, t_{k+1}]$. Namely, let x be a trajectory in $\mathcal{X}$ over $[t_k, t_{k+1}]$, taking action a implies costs

$$c(a,x) = \int_{t_k}^{t_{k+1}} f_a(x(t), \dot{x}(t), \ddot{x(t)})\ dt \tag{1}$$

$$\text{if}\quad g_a(x(t), \dot{x}(t), \ddot{x(t)}) \leq 0 \tag{2}$$

$$h_a(x(t), \dot{x}(t), \ddot{x(t)}) = 0\ . \tag{3}$$

and $c(a,x) = +\infty$ if the constraints are not satisfied by x.

Planning motions in the partially observable case means that we need to have motions pre-computed for all cases, i.e., for all observations that might happen during execution. This means that our planner computes trajectory pieces x for all action edges in the policy. Together, they form a trajectory tree. We require this tree to be smooth at its ramifications (which is implicit in smoothness costs and equality constraints). We use ψ to denote a full trajectory tree.

D. Optimal Policy and Trajectory Tree

We can now define the problem as finding a symbolic policy π and a trajectory tree ψ that minimize the expected

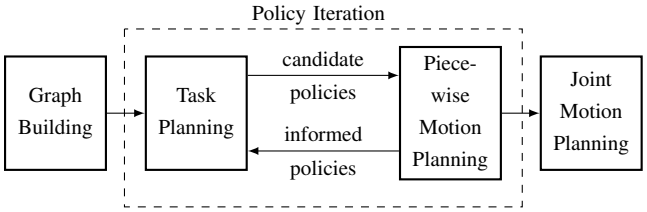


Fig. 3: TAMP algorithm

cost,

$$\Pi^* = \underset{(\pi,\psi)}{\operatorname{argmin}} \sum_{b\in\pi} p(b|\pi,b_0)\ \gamma^{k(b)}\ c(\pi(b),\psi(b))\ , \quad (4)$$

given the initial belief b_0. Here, the expectation is w.r.t. the probability of visiting a belief node in the decision tree, and discounted with gamma.

IV. Solver

We propose a solver that works in three stages, schematized on Fig. 3. First, the decision graph is built. Second, we alternate Value Iteration and piece-wise trajectory optimization to compute the symbolic policy π^* jointly with a set of trajectory pieces. These pieces do not yet form a globally optimal trajectory tree, but inform the symbolic policy optimization about the cost and feasibility associated with actions. In the third stage we fix π^* and optimize the full trajectory tree jointly.

The optimization of trajectory pieces in stage 2 raises substantial computational costs. Therefore our solver involves several mechanisms to decide on whether it is worth to "explore" actually computing these trajectory pieces. These mechanisms include using a simple approximation to check feasibility, and a mechanism akin to Rmax [3] to decide on whether an action should be "explored".

A. Graph Building

The decision graph, is expanded from the start belief state using a breadth-first strategy. In the general case, the number of reachable belief states is infinite, leading to an infinite decision graph. We limit the graph size by expanding only to a certain maximum depth.

B. Value Iteration on the decision graph

We assume that at any point in time we have cost estimates $c(a,b)$ for each action a in belief state b. However, these costs are all initialized with a low, optimistic number C_0, as in Rmax. Only when an optimal policy makes it likely to actually visit (a,b) do we compute more precise estimates using piece-wise trajectory optimization, as described below.

Given $c(a,b)$ we use Value Iteration

$$V^\star_{i+1}(b) \leftarrow \min_a \left[c(a,b) + \gamma \sum_{o\in O(b,a)} O(o|b,a) V^\star_i(T(b,a,o))\right] \quad (5)$$

until convergence to compute the value function for all belief nodes. Non-terminal (non-goal) leaf nodes are initialized with infinite value; terminal leaf nodes with zero value. This defines the optimal policy π^* w.r.t. the current estimates $c(a,b)$.

C. Piece-wise Trajectory Optimization

A given policy π^* transitions only a small subset of action edges of the full decision graph. For this set of action edges we compute $c(a,b)$ (if it was not already computed in previous iterations). For the sake of computational efficiency, we estimate $c(a,b)$ in two stages:

We first optimize key-frames only (robot pose at each node). This step is much quicker than optimizing a full trajectory piece. If an action is impossible at this stage, the optimization is not pursued further. In addition, this node and its sub-tree are labeled infeasible (with infinite costs) and thereby unreachable by the optimal policies. The pose feasibility check is optimistic, it might succeed even if the path itself is infeasible (no possible trajectory without collision between two key-frames for example).

If pose optimization reports feasibility, we then optimize the trajectory piece x, minimizing (1), to get the cost estimate $c(a,b)$. We use a time discretization of 20 frames per action. In addition to the cost and constraint functions defined by the action, the robot dynamics and collision avoidance are included. The trajectory optimization methods are adopted from [1][2]. We save the cost $c(a,b)$, the computed trajectory piece x, and in particular its final configuration for this action edge.

There may be a strong overlap between candidate policies generated by Value Iteration (same edge in many candidate policies). This is especially the case in the last iterations of Policy improvement, saving computational costs as computing $c(a,b)$ is performed only once. Intuitively, as we alternate between Value Iteration and evaluating relevant trajectory pieces, the decision graph is filled with geometric information.

D. Initialization of $c(a,b)$ and graph exploration

The use of optimistic initializations C_0 of $c(a,b)$ is analogous to the Rmax algorithm [3] and allows us to control exploration vs. exploitation within the policy optimization. An optimistic initial $c(a,b)$ (e.g., zero costs) encourages exploration. This Rmax mechanisms in combination with Value Iteration can be viewed as an alternative to other exploration mechanisms in tree search, such as admissible heuristics, or UCB in the probabilistic case. In our particular use case, we do not have uncertainty about the transitions, but only about the costs (negative rewards) as we lazily compute them. This relates well to the notion of "unknown states" in Rmax model-based reinforcement learning.

On the other hand, when initializing $c(a,b)$ less optimistically, we lose the guarantees that come with admissible heuristics, but may converge faster to reasonable policies. Our experiments will investigate the influence of the cost initialization on the number of iterations.

E. Joint Optimization of the Trajectory Tree

In the third stage of the solver, we fix the symbolic policy π^* found as described so far, and focus on the joint optimization of the trajectory tree ψ. So far we have only computed pieces x for each action edge. Concatenating

these independently optimized pieces cannot capture long-term dependencies in the trajectories, e.g. when final actions influence earlier parts of the trajectory. The autonomous driving example will exemplify this, when the velocity early in the trajectory tree should depend on the probability that an overtake maneuver will be possible. The joint optimization of the trajectory tree leads to better and smoother motions. It is computationally costly, but performed only once for the best symbolic policy π^*.

We again solve the problem stage-wise. We first optimize all linear trajectories from the root to each one of the reached terminal belief nodes *independently* (see *opt*_1 and *opt*_2 on Fig. 4). Secondly, the trajectories are re-optimized with additional equality constraints enforcing that the common parts between trajectories are identical (see *opt*_3 and *opt*_4). The re-optimization is potentially performed multiple times until the equality constraints across trajectories are fully satisfied (in practice, one iteration is often enough).

This procedure is akin to ADMM (Alternating Direction Multiplier Method), where we decomposed the tree problem into multiple linear trajectory problems, but introduce additional equality constraints to tie the common parts. Each linear trajectory problem can be addressed using efficient path optimization methods, exploiting its Markovian structure (e.g., banded-diagonal Hessian) [1][2]. ADMM is guaranteed to converge to the joint optimum and recently received great attention as a mean to decompose and parallelize large structured optimization problems.

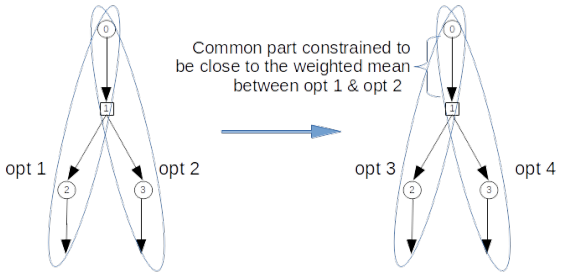


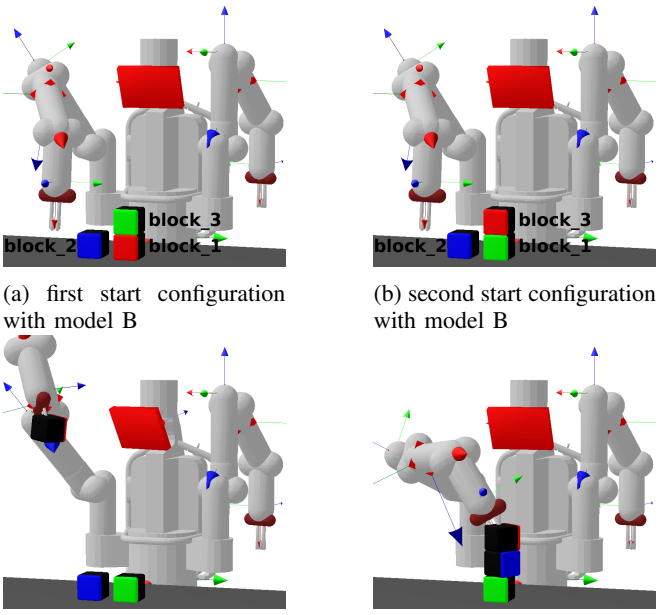
Fig. 4: Joint optimization

V. Experimental Results

A. Block stacking with the Baxter

We consider the Baxter and three blocks on a table (see Fig. 5). The robot has to stack the blocks in a given color order (blue, green, and red on the top). The blocks just have one side colored (assumed to be the opposite side). The robot knows where the blocks are (referred to as *block*_1, *block*_2, *block*_3). However it cannot see the colored side from behind and has to explore to identify the blocks and build the stack. There are 3 possible actions:

- **Look** at a block: the robot aligns its head with the colored side of the block. This typically leads the robot to move both its head and its arm simultaneously (see Fig. 5c). An observation is received after this action (block's color).
- **Grasp** a block: only the right arm can grasp



(a) first start configuration with model B

(b) second start configuration with model B

(c) *Look* with the block in hand

(d) example of goal configuration

Fig. 5: Example of configurations. The robot must stack the blocks in a given color order. The block colors are not visible from behind

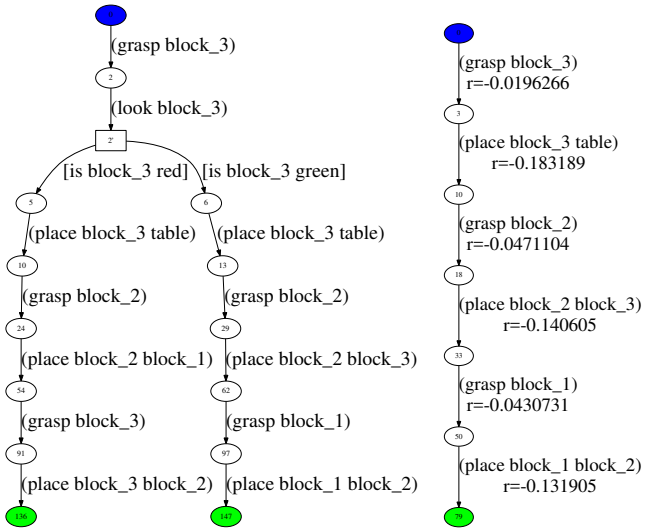
- **Place** a block at a location: the block is placed on the table, or onto another block.

To evaluate the scalability, three different initial belief states configurations are tested. In the planning problem A, the agent has prior knowledge of the blocks color. This boils down to the fully observable case. In problem B, 2 blocks are unknown (see Fig. 5a and 5b). In problem C, no prior knowledge is assumed, leading to 6 possible initial states. The initial belief state is uniform (1/6 likelihood for each possible state).

To evaluate robustness against motion planning failures, another problem (D) is considered. In B and C, the *Look* action has a precondition: the robot should have a block in hand before looking at it. This precondition is removed in the variation D. This causes the *Look* action to be symbolically possible more often. However, in most cases, if the robot doesn't hold the block, no robot motion allows the robot to align its head with the colored side of the block causing the motion planning to fail. The planning problems are summed up in table I.

	Belief state size	Blocks known	*precondition* for *Look* action	Graph size	Graph building time(s)
A	1	3/3	no *Look* action fully observable	63	0.023
B	2	1/3	yes	196	0.083
C	6	0/3	yes	1084	0.69
D	6	0/3	no	1486	1.18

TABLE I: Summary problem variations



(a) result policy for B (b) policy for A

	C_0	Iterations	Traj-cost	Task planning	Piecewise motion planning	Joint motion planning	Total** (s)
A	0.25	1	0.15	0.05	1.47	2.60	4.17
	0.1	1	0.15	0.05	2.54	2.33	4.98
	0.015	1	0.15	0.04	1.46	2.24	3.80
B	0.25	2	1.23	0.069	4.53	7.07	11.7
	0.1	5	0.74	0.16	14.4	9.42	24.1
	0.015	5	0.74	0.17	10.7	14.1	25.1
C	0.25	2	1.52	0.36	26.9	56.3	84.4
	0.1	13	1.17	1.35	133.3	74.4	209.8
	0.015	15	1.14	1.59	153.1	76.5	231.9
D	0.25	60	1.54	7.39	69.7	74.4	152.8
	0.1	187	1.17	26.5	323.2	83.2	434.2
	0.015	209	1.16	31.4	400.2	80.5	513.4

** Also includes the graph building time of table I

TABLE II: Number of iterations and planning times

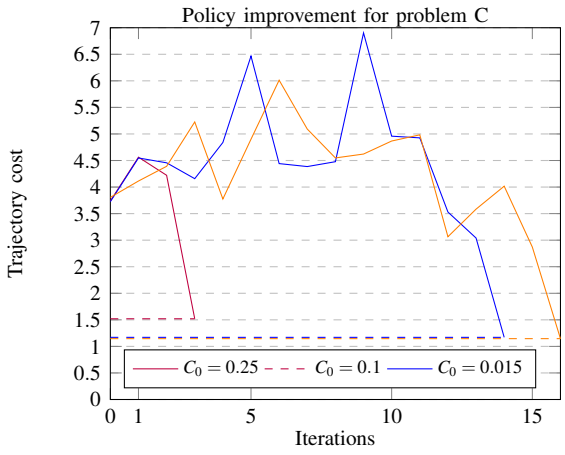


Fig. 7: Policy improvement over iterations depending on the cost initialization $c(a,b) = C_0$.

1) Influence of C_0: With $C_0 = 0.25$, planning time is minimized. The search even stops after one single iteration for A. However, this is at the expense of optimality. With lower values of C_0, the search explores more potential policies (more iterations) and the final policy costs are lower. In particular, deeper policies are considered (with a higher number of actions). Deeper policies are not necessarily more costly, this depends on the trajectory cost functions. In this example, the squared acceleration is minimized and slightly deeper policies allowing the robot arm to move more smoothly lead to better costs.

2) Influence of the action precondition: Removing the precondition increases the decision graph size. Many more iterations are needed. However, the search reaches a policy which is as good as the policy obtained with the precondition. This is an important quality of the proposed solution. Adding domain specific knowledge in the task planning (to ensure that motion planning will succeed) speeds up the search. However, generally speaking, we think that is it not always possible, nor convenient to incorporate geometric reasoning (reachability of a view point, reachability of an object) in the logical reasoning.

3) Execution time and scalability: The planning time is dominated by the motion planning (see table. II). For the simplest problems requiring the smallest number of iterations (A or B), joint optimization is the longest step. The execution time of this step depends on the total number of actions (typically around 33 for C or D) and on the belief state size, but it is independent from the number of iterations having occurred before. When more iterations take place (C, D), the planning time is dominated by the piecewise motion planning.

B. Comparison against another TAMP Planner

We used the block example introduced above to benchmark the proposed solver against the *Task Motion Kit* (TMKit) [10] in the fully-observable case. This solver implements the *Incremental Task and Motion Planning* algorithm (IDTMP) (see [11]). We added a 4th block in the example to further test the scalability. The benchmark was conducted on an Intel®Core™i3-4100M CPU under Ubuntu 16.04. Table III shows planning metrics obtained by running each planner 100 times on each problem.

	Avg. path length (m)	std-dev	Task planning	Motion planning	Total (s)
3 blocks - LGP	4.03	0.04	0.10	2.88	2.98
3 blocks - TMKit	3.19	1.01	1.88	5.53	7.41
4 blocks - LGP	5.79	0.04	0.36	4.62	4.98
4 blocks - TMKit	7.54	1.12	11.32	20.39	31.71

TABLE III: Planner comparison

The proposed planner is faster than the TMKit for this given problem. One reason lies in the interface between the Motion and Task Planner. In the TMKit, Motion planning queries are composed of the start and goal poses of a given frame (robot end-effector or object). This requires knowledge of the final pose before motion planning. To generate candidate

goal poses, the surfaces where an object can be placed (e.g. the table) are discretized. This increases the branching factor, which makes the search less efficient. In this experiment, the table is relatively big (1.5 m x 0.8 m), and the discretization factor used was 10 cm (default value provided by the TMKit for similar examples). Further experiments showed that setting it to 20 cm decreases the average total planning time to 4.76 s (3 blocks) and 8.45 s (4 blocks). With the LGP, the final poses are naturaly obtained as a result of the trajectory optimization and don't rely on a discretization of the scene. We also report on path length but note that LGP actually optimizes for path smoothness (TMKit paths are not smooth). The trajectories planned with the LGP are similar across multiple runs (low standard deviation). On the other hand, the paths of the TMKit (obtained with the RRT-Connect algorithm) vary much more.

C. Overtaking behavior

We consider the overtaking problem introduced previously (see III-B.1). While simple, this example highlights the advantage of the joint (vs. piece-wise) trajectory-tree optimization, which leads to anticipatory optimal trajectories. Fig. 9a shows two initial configurations. In the first configuration (a), the opposite lane is free enough to overtake. In (b) overtaking is not possible. The trajectory cost of the action *Look* is implemented as the distance between the car and the center of the road. It moves the car toward the center to catch sight of the lane (see Fig. 9b). The action *Follow* is implemented as a constraint, which is satisfied if the ego-car is behind the truck (keeping a safe distance) at the end of the action. The action *Overtake* is a constraint satisfied if the ego-car is in front of the truck. The Fig. 8 shows the obtained policy. The solver excluded the possibility of overtaking directly without looking at the lane (edge 0-2 of the decison graph) because the motion planner can't satisfy the collision avoidance constraint in this case.

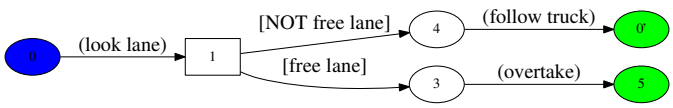
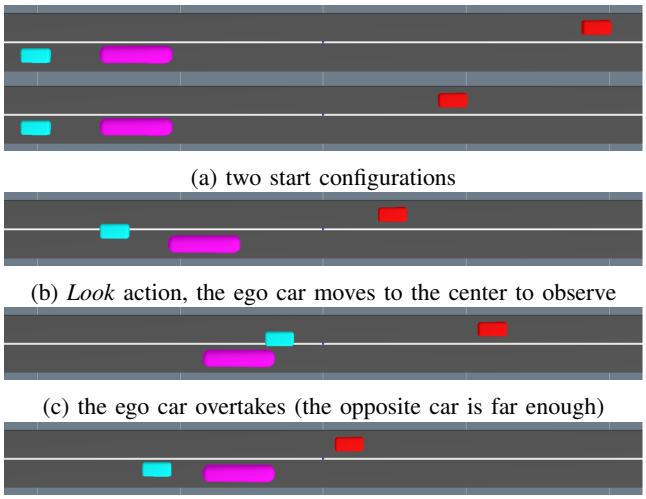


Fig. 8: Overtaking optimal policy

The curves on Fig. 10 represent the longitudinal velocities of the trajectory tree for different planning configurations. At $t = 6.5s$, the car receives the observation $[lane\ free]$ or $[NOT\ lane\ free]$. If the lane is free, the car accelerates to overtake and then slows down once the truck is overtaken. Otherwise, the car slows down and moves back to follow the truck.

The gray curve stems from the piecewise motion planning. When looking at the lanes, the car maintains exactly the same velocity (gray curve is flat for $t < 6.5s$). When the overtaking kicks in, the car - still quite far from the truck - accelerates vigorously. On the other hand, the blue and purple curves (Joint Optimization) are much smoother. To avoid too strong an acceleration, the car anticipates and accelerates



(a) two start configurations

(b) *Look* action, the ego car moves to the center to observe

(c) the ego car overtakes (the opposite car is far enough)

(d) the ego car moves back (opposite lane not free)

Fig. 9: Geometric configurations for the car example

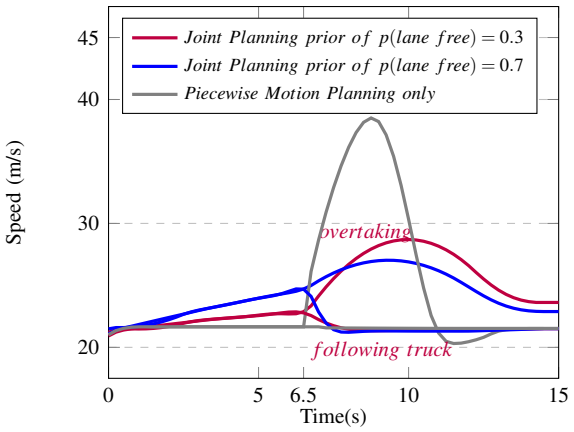


Fig. 10: Longitudinal speed of the overtaking maneuver

slightly when looking. We think that this mimics what human drivers do in case of "tense" overtaking maneuver. The initial belief state also influences the behavior. If it is likely that overtaking is possible (0.7 likelihood for the blue curve), the car will accelerate more when looking. In practice, the initial belief state could come from a service providing global information about the traffic in the area.

VI. Conclusion & Future Work

We proposed a new, optimization-based approach to TAMP problems under partial observability. It computes a reactive policy in the belief space, spanned by a symbolic decision graph and its corresponding trajectory tree that allow the agent to reactively choose from pre-computed motion options depending on the observations. Thereby it can plan policies that combine exploratory actions (mostly sensor trajectories) and exploitative actions (e.g. grasp, place). The degree of exploration over the space of all possible manipulation policies is controlled by the cost initialization parameter. We are currently working on extending the approach to multi-agent domains (see [12]).

REFERENCES

[1] M. Toussaint and M. Lopes, Multi-Bound Tree Search for Logic-Geometric Programming in Cooperative Manipulation Domains. Accepted at ICRA 2017.

[2] M. Toussaint, Logic-Geometric Programming: An Optimization-Based Approach to Combined Task and Motion Planning. In Proc. of the Int. Joint Conf. on Artificial Intelligence (IJCAI 2015), 2015.

[3] Brafman, Ronen I. and Tennenholtz, Moshe: R-MAX - a general polynomial time algorithm for near-optimal reinforcement learning. Journal of Machine Learning Research, Volume 3, 213-231, 2003

[4] Leslie Pack Kaelbling, Tomas Lozano-Perez. Integrated Task and Motion Planning in Belief Space, International Journal of Robotics Research, 2013

[5] T. Lozano-Pérez and L. P. Kaelbling. A constraint-based method for solving sequential manipulation planning problems. In Intelligent Robots and Systems (IROS 2014), 2014 IEEE/RSJ International Conference on, pages 3684-3691. IEEE, 2014

[6] Katrakazas, C., Quddus, M., Chen, W.-H., Deka, L.: Real-time motion planning methods for autonomous on-road driving: State-of-the-art and future research directions. Transp. Res. Part C 60, 416-442, 2015

[7] Paden, B., Cap, M., Yong, S.Z., Yershov, D., Frazzoli, E.: A survey of motion planning and control techniques for self-driving urban vehicles. IEEE Trans. Intell. Veh. 1(1), 33-55, 2016

[8] F. Lagriffoul, D. Dimitrov, A. Saffiotti, and L. Karlsson. Constraint propagation on interval bounds for dealing with geometric backtracking. In Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on, pages 957-964. IEEE, 2012

[9] F. Lagriffoul, D. Dimitrov, J. Bidot, A. Saffiotti, and L. Karlsson. Efficiently combining task and motion planning using geometric constraints. The International Journal of Robotics Research, 2014

[10] N. T. Dantam, S. Chaudhuri, and L. E. Kavraki, Incremental Task and Motion Planning: The Task Motion Kit. IEEE Robotics and Automation Magazine, 2018

[11] N. T. Dantam, Z. K. Kingston, S. Chaudhuri, and L. E. Kavraki, Incremental Task and Motion Planning: A Constraint-Based Approach, in Robotics: Science and Systems, 2016

[12] C. Phiquepal and M. Toussaint, Multi-Agent Task and Motion Planning: An Optimization based Approach. RSS, Workshop on Integrated Task and Motion Planning, 2018.