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 new algorithm for Combined Task and Motion Planning (TAMP) under partial observability. Our approach builds on the Logic-Geometric Programming framework (LGP) presented in prior work [1, 2]. To represent partial observability, the decision tree is formulated in belief space. The presented algorithm plans policies that react to the observations that the agent receives, both on the symbolic the motion level. To this end, we optimize *trajectory trees* that describe the branchings of optimal motions depending on the observations. The algorithm works in two stages: First, the symbolic policy is optimized using approximate path costs estimated from independent optimization 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. The algorithm performance is compared against a state-of-the-art TAMP planner in the fully observable case.**

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 limiting the field of view of the ego vehicle.

In this paper we extend the Logic-Geometric Programming approach (LGP) presented in prior work to handle partial observability. Under partial observability, policies have branching points due to observations. On the geometric level, this means that motion planning consists in optimizing a trajectory tree, i.e. trajectories with ramifications. Our TAMP solver works in two stages: First piece-wise optimization and then joint optimization. During the first stage, the trajectories of actions are optimized independently (piece-wise). The second stage of the algorithm is the joint optimization. The trajectory tree of the best policy obtained in the first stage is re-optimized globally: 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. Our prior work presented in [1][2] states TAMP problems as an optimization problem. These approaches assume full observability, and plans are linear sequences of actions. To our knowledge, the system developed by Lozano-Perez and Kaebling [4] is the only other TAMP planner considering 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). Sequences of actions are planned by approximating the system dynamics (results of actions and observations). Replanning is triggered once the robot ends up in a state not covered by the plan. Our approach aims at planning a full policy from the starting state to the final state. In addition, we aim for smooth and locally optimal trajectories.

III. Problem Formulation

A. Overview

Our problem is defined in terms of a decision tree and, for every action edge in this tree, a cost and constraint functions on the continuous trajectory. The decision tree is in belief space, including action and observation branchings. To define a problem we first define the partially observable decision process that spans this tree. Second, we define the cost and constraints functions that define the continuous trajectory problems 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.

B. Decision Process

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

- S is a finite set of states
- A is a finite set of actions
- T is a set of conditional transition probabilities between states

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

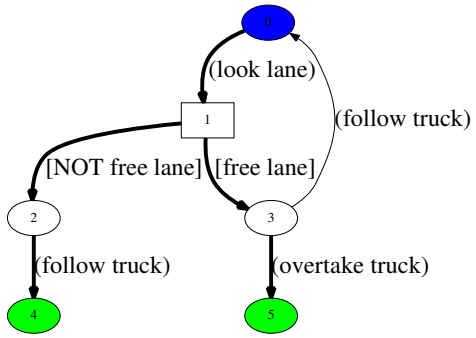


Fig. 2: Decision graph for overtaking, the thick edges represent a possible policy

- Ω is a finite set of observations
- O is a set of conditional observation probabilities
- γ is a discount factor
- G is a set of goal conditions defined over the belief state space. A goal condition indicates if a belief state 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 uniquely corresponds a belief state computed with the standard belief update.

1) Example for a decision graph: We consider a car behind a truck, the car wishes to overtake but cannot see the opposite lane because of the truck, see Fig. **??**. 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. **??** is the decision graph of this problem, assuming loop detection to turn the tree into a graph. The blue node is the start belief state. The green nodes are terminal belief states.

2) Policy: A policy π is a mapping from belief states to actions. The thick edges in Fig. **??** represent a possible policy. The decision graph potentially contains cycles, however, in the following, we will only consider policies that are trees (no cycles). In the case of full observability, the policy would boil down to a sequence of actions. A policy reaching the goal condition, on the symbolic level, is a tree whose leafs are terminal belief states. We call it 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 $p(b|\pi,b_0)$.

C. Trajectory Trees

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

More formally, let χ 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 χ over $[t_k, t_{k+1}]$, taking action a implies costs

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

$$s.t \quad g_a(x(t), \dot{x}(t), \ddot{x}(t)) <= 0 \tag{2}$$

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

if feasible, and $+\infty$ if the constraints cannot be satisfied.

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 decision tree. 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 cost,

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

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. **??**. First, the decision graph is build. 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 associated with actions. In the third stage we fix π^* and optimize the full trajectory tree jointly.

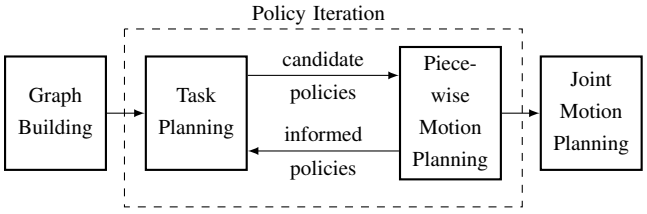


Fig. 3: TAMP algorithm

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 maximal 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 initially 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) we compute more precise estimates using piece-wise trajectory optimization, as described below.

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

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

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

As π^* is a deterministic policy, it 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 during the feasibility check, 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).

Is pose optimization reports feasibility, we then optimize the trajectory piece x, minimizing (**??**), 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 dynamic 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 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 were lazy to 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 optimistic, we loose 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 car 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 the root to each one of the reached terminal belief nodes *independently* (see opt_1 and opt_2 on Fig. **??**). 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 tied 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 means to decompose and parallelize large structured optimization problems.

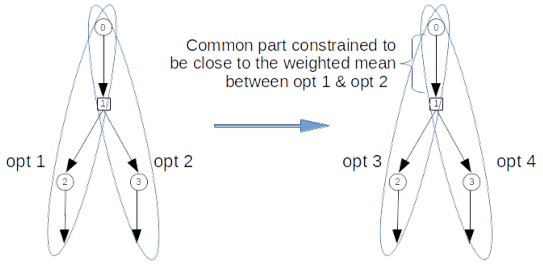


Fig. 4: Joint optimization

V. Experimental Results

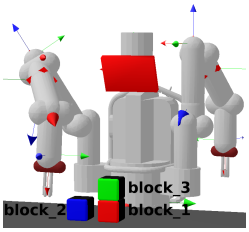
A. Block stacking with the Baxter

We consider the Baxter and three blocks on a table (see Fig. **??**). 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 as *block*_1, *block*_2, *block*_3). However it cannot see the colored side from behind and has explore to identify the blocks and to build the stack in the correct order. There are 3 possible actions:

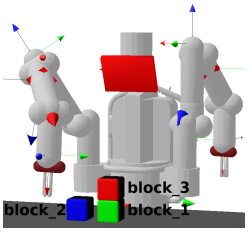
- **Look** a block: the robot aligns its head with the colored side of the block. This typically leads the robot to both move its head and its arm simultaneously (see Fig. **??**). An observation is received after this action(block's color).
- **Grasp** a block: only the right arm can grasp
- **Place** a block at a location: the block is placed on the table, or onto another block.

To evaluate the scalability, we test with three different initial belief states configurations. In the planning problem A, the agent has prior knowledge of the blocks color. This boils down to the fully observable case. In the problem B, 2 blocks are unknown (see Fig. **??** and **??**). 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).

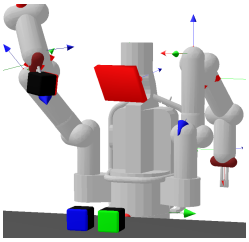
To evaluate the robustness against motion planning failures, another problem (D) is considered. In B and C, the *Look* action has a precondition: the robot should have a block in hand before looking at it. This precondition is removed in the variation D. This causes the *Look* action to be symbolically 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 the table **??**.



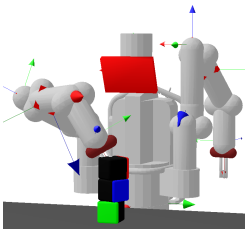
(a) first start configuration with model B



(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

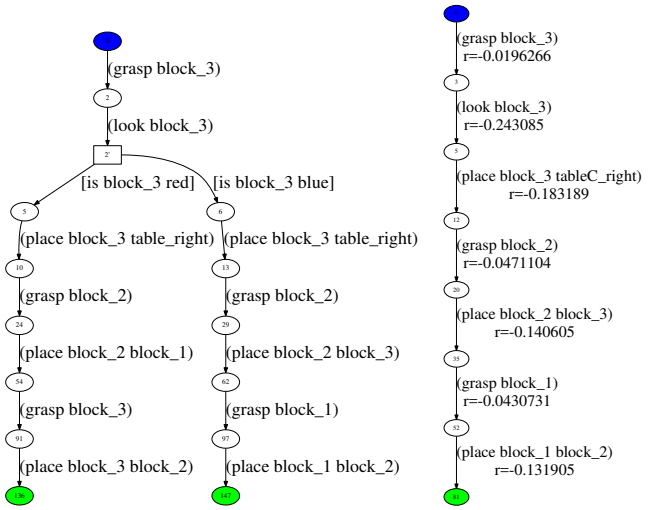
	Belief state size	Blocks known	*precondition* for *Look* action	Graph size	Graph building time(s)
A	1	3/3	no *Look* action fully observable	63	0.023
B	2	1/3	yes	196	0.083
C	6	0/3	yes	1084	0.69
D	6	0/3	no	1486	1.18

TABLE I: Summary of the considered problem variations

	C_0	Iterations	N of actions*	Task planning	Piecewise motion planning	Joint motion planning	Total**
A	0.25	2	6	0.027	1.62	3.27	4.95
	0.1	2	6	0.034	1.67	3.54	5.27
	0.015	2	6	0.029	1.67	4.02	5.75
B	0.25	5	12	0.10	8.83	14.1	23.1
	0.1	5	12	0.10	8.02	15.1	23.3
	0.015	7	12	0.17	10.7	14.1	25.1
C	0.25	11	33	0.81	42.6	66.9	111.0
	0.1	22	33	1.69	61.8	66.9	131.1
	0.015	22	33	1.77	66.3	67.2	135.2
D	0.25	80	33	10.2	97.2	84.9	193.6
	0.1	199	33	27.6	234.2	86.2	349.2
	0.015	252	33	40.0	385.5	114.1	540.9

* Number of actions of the final policy
** The total planning time also includes the graph building time given in the table **??**

TABLE II: Number of iterations and planning times



(a) result policy for B2. The block 3 is checked, then the stacking is continued sequentially

(b) result policy for A2

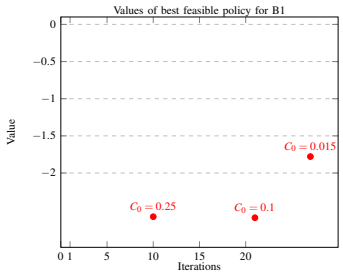


Fig. 7: Policy improvement over iterations depending on the cost initialization $c(a,b) = C_0$

1) Influence of C_0 initialization: With a cost initialization of 0.25 (pessimistic), the search finishes as soon as a first policy is found. This happens after one single iteration with the B1. With the B2, the search encounters infeasible actions, the first possible policy is found at the 7th iteration. This policy is less optimal than the results found with the other initializations.

The lowest cost initialization (0.015) is always optimistic and leads to the most exploratory behavior. The value of candidate policies (containing at least one unexplored action) are consequently always higher than the result policies (see the orange curves on Fig. **??**. It requires the larges number of iterations. In particular, task planning also explores deeper policies (with more steps). In some cases a slightly deeper policy results in a better trajectory costs (see B2 with C_0 = 0.015). The search converges to a better policy than with the other cost initializations. The small improvements in the last iterations are due to small rearrangements of the target location when placing blocks (e.g. place a block on table-left instead of table-center).

2) Influence of the action precondition: Removing the precondition increases strongly the decision graph size, more iterations are needed. The search reaches an optimal policy which is as good as the policy obtained with the model with precondition. We think that 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, in the general case, we think that it is not always possible / convenient to incorporate such geometric reasoning (reachability of a view point, reachability of an object) in the logical reasoning.

3) Execution time and scalability: The overall planning time is dominated by the motion planning (see table. **??** As long as the model is simple (A or B) or the exploration kept low (C_0 = 0.25), motion planning is dominated by the single pass of joint optimization. The execution time of this pass mainly depends on the total number of action steps in the policy and on the belief state size but is independent from the number of iterations occurring before. In the configurations requiring the biggest number of iterations (C1 and C2 with C_0 = 0.015), motion planning and the overall planning time are dominated by the piecewise motion planning phases of the policy improvement iterations. Parameterizing the search with a very exploratory behavior may be feasible for problems of small sizes but suffers from the curse of dimensionality. One way to still enable some exploration while maintaining a bounded planning time is to save the best candidate policy planned so far and interrupt the iterations when a given time limit is reached.

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 an 4-th block in the example to further test the scalability. The benchmark was conducted on an Intel®Core™i3-4100M CPU under Ubuntu 16.04. **??** shows planning metrics obtained by running each planner 100 times on each problem.

	Avg. path length	std-dev	Task Planning	Motion Planning	Total (s)
3 blocks LGP	4.03	0.04	0.10	2.88	2.98
3 blocks TMKit	3.28	1.08	1.84	6.09	6.98
4 blocks LGP	5.79	0.04	0.36	4.62	4.98
4 blocks TMKit	7.54	1.12	11.32	20.39	31.72

TABLE III: Planner comparison

The proposed planner is faster than the TMKit for this given problem. 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. One reason for the longer planning time of the TMKit lies in the interface between the Motion and Task Planner. Motion planning queries are composed of the start and goal pose of a given frame (robot end-effector or object). This requires to know

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 of the search which makes it less efficient. Within the LGP framework, a motion planning problem is specified by a start pose plus cost and constraints functions. The final poses are naturaly obtained as a result of the trajectory optimization and don't rely on a discretization of the scene.

C. Overtaking behavior

We consider the overtaking problem introduced previously. Albeit simple, this example highlights the advantage of the joint (vs. piece-wise) trajectory-tree optimization which leads to anticipatory optimal trajectories. Fig. **??** is the decision graph and Fig. **??** shows two start configurations. In the first configuration (a), the opposite lane is free enough to overtake. In (b) overtaking is not possible. The Fig. **??** shows the optimal policy. 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 get sight of the lane (see Fig. **??**). The action *Follow* is implemented as a constraint which is satisfied if the ego-car is behind the truck (with a safety distance) at the end of the action. The action *Overtake truck* is a constraint satisfied if the ego-car is in front of the truck.

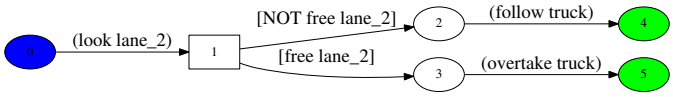
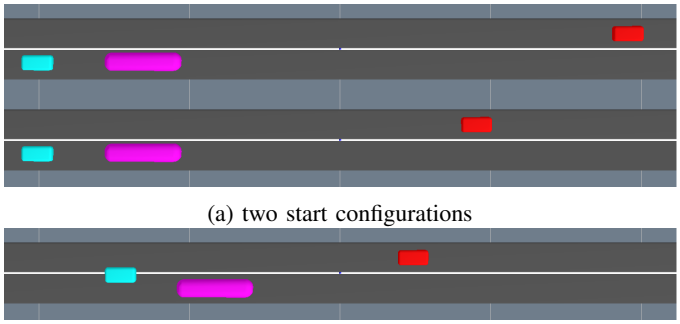


Fig. 8: Overtaking optimal policy



(a) two start configurations

(b) *Look* action, the ego vehicle moved to the road center to observe the lanes

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

(d) the ego vehicle moves back to its lane (opposite lane not free)

This example emphasizes the improvement brought by the joint trajectory optimization. The curves on Fig. **??** represent the longitudinal velocities of the trajectory tree for different

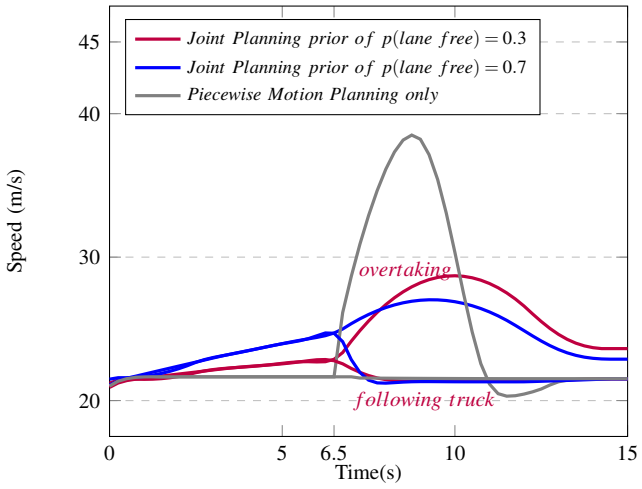


Fig. 10: Longitudinal speed of the overtaking maneuver

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 move back to follow the truck.

The gray curve results from the piecewise motion planning. When looking at the lanes, the car keeps exactly the same speed (gray curve is flat for $t < 6.5s$). When starting to overtake, the car is, still quite far from the truck and accelerates strongly to overtake. On the other hand, the blue and purple curves (Joint Optimization) are much smoother. To avoid a too strong acceleration, the car anticipates and accelerates slightly when looking. If overtaking is possible, the car pursues its acceleration, otherwise, it slows down and go back in the lane. 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 finite decision graph, and a trajectory tree that allows 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). Within planning, the degree of exploration over the space of all possible manipulations policies can be controlled by one single parameter (cost initialization), akin to Rmax.

Scalability becomes an issue when the number of manipulations and or the size of the belief state increases. An efficient way to speed-up the search is to have a task-level model that is accurately tailored for the problem to solve

(example of the grasp precondition), this prevents too many motion planning failures and limits the branching factor. In future work, we intend to explore, how the task-level model can be refined and learned using the results from multiple planning queries.

Our current method computes policies that address every possible outcome of probabilistic observations. To scale better, we plan to investigate in an approach where the policy is planned only to handle the most probable belief state trajectories. As such a policy couldn't handle every outcome at execution time, re-planning would be triggered once the execution layer detects that the system is evolving toward a belief state which is not covered by the current policy.

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