Provably Constant-time Planning and Replanning for Real-time Grasping Objects off a Conveyor Belt

Fahad Islam\textsuperscript{1} and Oren Salzman\textsuperscript{2} and Aditya Agarwal\textsuperscript{1} and Maxim Likhachev\textsuperscript{1}

Abstract
In warehouse and manufacturing environments, manipulation platforms are frequently deployed at conveyor belts to perform pick and place tasks. Because objects on the conveyor belts are moving, robots have limited time to pick them up. This brings the requirement for fast and reliable motion planners that could provide provable real-time planning guarantees, which the existing algorithms do not provide. Besides the planning efficiency, the success of manipulation tasks relies heavily on the accuracy of the perception system which is often noisy, especially if the target objects are perceived from a distance. For fast moving conveyor belts, the robot cannot wait for a perfect estimate before it starts executing its motion. In order to be able to reach the object in time, it must start moving early on (relying on the initial noisy estimates) and adjust its motion on-the-fly in response to the pose updates from perception. We propose a planning framework that meets these requirements by providing provable constant-time planning and replanning guarantees. To this end, we first introduce and formalize a new class of algorithms called Constant-Time Motion Planning algorithms (CTMP) that guarantee to plan in constant time and within a user-defined time bound. We then present our planning framework for grasping objects off a conveyor belt as an instance of the CTMP class of algorithms. We present it, give its analytical properties and show experimental analysis both in simulation and on a real robot.

Keywords
Motion Planning, Automation, Manipulation

1 Introduction
Conveyor belts are widely used in automated distribution, warehousing, as well as for manufacturing and production facilities. In the modern times robotic manipulators are being deployed extensively at the conveyor belts for automation and faster operations Zhang et al. (2018). In order to maintain a high-distribution throughput, manipulators must pick up moving objects without having to stop the conveyor for every grasp. In this work, we consider the problem of motion planning for grasping moving objects off a conveyor. An object in motion imposes a requirement that it should be picked up in a short window of time. The motion planner for the arm, therefore, must compute a path within a bounded time frame to be able to successfully perform this task.

Manipulation relies on high quality detection and localization of moving objects. When the object first enters the robot’s field of view, the initial perception estimates of the object’s pose are often inaccurate. Consider the example of an object (sugar box) moving along the conveyor towards the robot in Fig. 1, shown through an image sequence as captured by the robot’s Kinect camera in Fig. 2. The plot in Fig. 2 shows the variation of the error between the filtered input point cloud and a point cloud computed from the predicted pose from our Iterative Closest Point (ICP)-based perception strategy Islam et al. (2019) as the object gets closer to the camera. We observe that the error decreases as the object moves closer, indicating that the point clouds overlap more closely due to more accurate pose estimates closer to the camera.

However, if the robot waits too long to get an accurate estimate of the object pose, the delay in starting plan execution could cause the robot to miss the object. The likelihood of this occurring as the speed of the conveyor

\textsuperscript{1}The Robotics Institute, Carnegie Mellon University
\textsuperscript{2}Technion-Israel Institute of Technology

Corresponding author:
Fahad Islam.
Email: fahad.islam@fulbrightmail.org
We make the following contributions in this work:

1. We develop a provably constant-time planning and replanning framework for grasping moving objects off a conveyor belt.
2. We prove that the algorithm is complete and is guaranteed to run in constant time and within a user specified time bound.
3. We provide experimental analysis of our approach in simulation as well as on a physical robotic system.
4. We introduce and formalize a new class of algorithms called Constant-Time Motion Planning (CTMP) algorithms and show that the proposed approach for grasping objects off the conveyor is an instance of CTMP class of algorithms.
5. We develop a kinodynamic motion planner to account for the dynamic constraints of the robot including joint torque and velocity limits.

This article is in continuation of our previous work presented in Islam et al. (2020) and the contributions 4 and 5 specifically, are the extensions. In addition to these extensions, we provide space complexity analysis of our approach and report detailed preprocessing statistics of our experiments to highlight the improvement over the brute force method. We also remove one of the assumptions of Islam et al. (2020) which says that the environment remains static up to a certain time in execution.

2 Related work

2.1 Motion planning for conveyor pickup task

Existing work on picking moving objects has focused on different aspects of the problem ranging from closed-loop controls, servoing to object perception and pose estimation, motion planning and others Zhang et al. (2018); Allen et al. (1993); Han et al. (2019); Stogl et al. (2017). Here, we focus on motion-planning related work. Time-configuration space representation was introduced to avoid moving obstacles Fraichard (1993); Cefalo et al. (2013); Yang et al. (2018). Specifically in Yang et al. (2018), a bidirectional sampling-based method with a time-configuration space representation was used to plan motions in dynamic environments to pickup moving objects. While their method showed real-time performance in complex tasks, it used fully-specified goals; namely knowing the time at which the object should be picked, which weakens the completeness guarantee. Furthermore their method is probabilistically complete and therefore, does not offer constant-time behavior. Graph-search based approaches have also been used for the motion-planning problem Menon et al. (2014); Cowley et al. (2013). The former uses a kinodynamic motion planner to smoothly pick up moving objects i.e., without an impactful contact. A heuristic search-based motion planner that plans with dynamics and could generate high-quality trajectories with respect to the time of execution was used. While this planner provides strong guarantees on the solution quality, it is not real-time and thus cannot be used online. The latter work demonstrated online real-time planning capability. The approach plans to a pregrasp pose with pure kinematic planning and relies on Cartesian-space controllers to perform the pick up. The usage of the Cartesian controller limits the types of objects that the

1.1 Statement of Contributions

We make the following contributions in this work:

...
robot can grasp. It also provided no guarantee to be able to generate a plan in time to execute.

### 2.2 Preprocessing-based planning

Preprocessing-based motion planners often prove beneficial for real-time planning. They analyse the configuration space offline to generate some auxiliary information that can be used online to speed up planning. Probably the best-known example is the Probabilistic Roadmap Method (PRM) Kavraki et al. (1996) which precomputes a roadmap that can answer any query by connecting the start and goal configurations to the roadmap and then searching the roadmap. PRMs are fast to query yet they do not provide constant-time guarantees. To account for dynamic environments, several PRM method-based extensions have been proposed Yoshida and Kanehiro (2011); Belghith et al. (2006). However, these methods often require computationally expensive repairing operations which cause additional overheads.

A provably constant-time planner was recently proposed in Islam et al. (2019). Given a start state and a goal region, it precomputes a compressed set of paths that can be utilised online to plan to any goal within the goal region in bounded time. As we will see, our approach while algorithmically different, draws some inspiration from this work. Both of the above two methods (Kavraki et al. (1996); Islam et al. (2019)) mainly target pure kinematic planning and thus they cannot be used for the conveyor-planning problem which is dynamic in nature.

Another family of preprocessing-based planners utilises previous experiences to speed up the search Berenson et al. (2012); Coleman et al. (2015); Phillips et al. (2012). Experience graphs Phillips et al. (2012), provide speed up in planning times for repetitive tasks by trying to reuse previous experiences. These methods are also augmented with sparsification techniques (see e.g., Dobson and Bekris (2014); Salzman et al. (2014)) to reduce the memory footprint of the algorithm. Unfortunately, none of the mentioned algorithms provide fixed planning-time guarantees that we strive for in our application.

### 2.3 Online replanning and real time planning

The conveyor-planning problem can be modelled as a Moving Target Search problem (MTS) which is a widely-studied topic in the graph search-based planning literature Ishida and Korf (1991, 1995); Koenig et al. (2007); Sun et al. (2010). These approaches interleave planning and execution incrementally and update the heuristic values of the state space to improve the distance estimates to the moving target. Unfortunately, in high-dimensional planning problems, this process is computationally expensive which is why these approaches are typically used for two-dimensional grid problem such as those encountered in video games. More generally, real-time planning is widely considered in the search community (see, e.g., Koenig and Likhachev (2006); Koenig and Sun (2009); Korf (1990)). However, as mentioned, these works are typically applicable to low-dimensional search spaces.

### 3 Constant-Time Motion Planning (CTMP) for Indefinite Horizons

Before we formally define our motion-planning problem, we introduce a new class of algorithms that we call Constant-Time Motion Planning algorithms (CTMP) that are specially designed for repetitive robotic tasks. We will see that the algorithm we propose for real-time grasping objects off a conveyor belt is an instance of a CTMP algorithm. However, we believe that their definition is of independent interest regardless of the specific problem instance that is considered. While limited-horizon planners like real-time heuristic search-based planners also plan in constant-time, this paper specifically addresses the indefinite horizon planning problem, that is to plan all the way to the goal Koenig and Likhachev (2006); Koenig and Sun (2009); Korf (1990).

#### 3.1 CTMP Problem Definition

We consider the problem setting where the robot has a fixed start state \( s_{\text{start}} \) and a set of goals \( G \), the representation and specification of which are domain dependent. The algorithm is queried for a pair \((s_{\text{start}}, g)\) where \( g \in G \) and attempts to compute a feasible path from \( s_{\text{start}} \) to \( g \) within a (small) constant time.

**Definition 1. CTMP Algorithm.** Let \( \text{ALG} \) be a motion-planning algorithm, \( T_{\text{bound}} \) a user-controlled time bound and \( T_{\text{const}} < T_{\text{bound}} \) a small time constant whose value is independent of the size and complexity of the motion-planning problem. \( \text{ALG} \) is said to be a CTMP algorithm if it is guaranteed to answer any motion planning query within \( T_{\text{bound}} \).

The input time \( T_{\text{bound}} \) is a tunable parameter that can be used to trade off query time with the memory footprint and preprocessing efforts of the algorithm. It is important to note that an algorithm that returns NO_PATH for any query is CTMP. Thus, unless endowed with additional properties CTMP is a weak algorithmic property. We now define one such property, namely CTMP-Completeness, that makes a CTMP algorithm interesting and useful.

#### 3.2 CTMP-Completeness

We assume a CTMP algorithm has access to a regular motion-planning algorithm \( P \) (that is not necessarily CTMP). We introduce a new notion of completeness for CTMP algorithms. Prior to that, we need to define some preliminaries.

**Definition 2. Reachability.** A goal \( g \in G \) is said to be reachable from a state \( s_{\text{start}} \) if \( P \) can find a path to it within a (sufficiently large) time bound \( T_P \).

**Definition 3. Goal Coverage.** A reachable goal \( g \in G \) is said to be covered by the CTMP algorithm for a state \( s_{\text{start}} \) if (in the query phase) it can plan from \( s_{\text{start}} \) to \( g \) while satisfying Def. 1.

We are now equipped to define CTMP-Completeness.

**Definition 4. CTMP-Completeness.** An algorithm is said to be CTMP-complete if it covers all the reachable goals \( g \in G \) for \( s_{\text{start}} \).
A CTMP-algorithm provides a planning time bound reduction of $T_P : T_{\text{bound}}$, where $T_{\text{bound}} \ll T_P$, while still guaranteeing a success rate no worse than what $P$ would provide with a time bound of $T_P$. A CTMP-complete algorithm distinguishes from a CTMP algorithm for it must cover all reachable goals in $G$ for $s_{\text{start}}$ in the preprocessing phase, so that in the query phase, it can find a plan to any reachable goal within $T_{\text{bound}}$ time. A CTMP algorithm which is not CTMP-complete however, may return failure even if the queried goal is reachable.

Note that the notion of CTMP-completeness is decoupled from the completeness guarantee (by conventional definition) of the underlying motion planner $P$, hence we can view a CTMP algorithm as a meta planner. The CTMP algorithm only provides guarantees for the subset of $G$ which is reachable and hence its completeness properties differ from the completeness properties of $P$. However, the size of the reachable set of goals is largely dependent on the performance of $P$.

4 CTMP for Grasping Objects off A Conveyor Belt—Problem Setup

Our system is comprised of a robot manipulator, a conveyor belt moving at some known velocity, a set of known objects $O$ that need to be grasped and a perception system that is able to estimate the type of object and its location on the conveyor belt. Here, we assume that the geometric models of the target objects are known apriori and we are given a set of feasible grasps for each object $o \in O$.

Given a pose $g$ of an object $o \in O$, our task is to plan the motion of the robot such that it grasps $o$ from the conveyor belt at some future time. Unfortunately, the perception system may give inaccurate object poses. Thus, the pose $g$ will be updated by the perception system as the robot is executing its motion. To allow for the robot to move towards the updated pose in real time, we introduce the additional requirement that planning should be done within a user-specified time bound $T_{\text{bound}}$. For ease of exposition, when we say that we plan to a pose $g$ of $o$ that is given by the perception system, we mean that we plan the motion of the robot such that it will be able to pick $o$ from the conveyor belt at some future time. This is explained in detail in Sec. 6.2.1 and in Fig. 9.

We denote by $G_{\text{full}}$ the discrete set of initial object poses on the conveyor belt that the perception system can perceive. Finally, we assume that the robot has an initial state $s_{\text{home}}$ corresponding to the time $t = 0$ from which it starts planning to grasp any object.

Roughly speaking, the objective is to enable planning and replanning to any goal pose $g \in G_{\text{full}}$ in bounded time $T_{\text{bound}}$ regardless of the robot’s current state. Specifically we aim to develop a CTMP-complete algorithm, so that for any state $s$ that the system can be in and every reachable goal pose $g \in G_{\text{full}}$ from $s$ updated by the perception system, $g$ is covered by $s$.

We assume that the pose estimation error of the perception system is bounded by a distance $\varepsilon$. We also specify a replan cutoff time $t_{\text{rc}}$ after which the planner does not replan and continues to execute the last planned path until the goal is reached. The time $t_{\text{rc}}$ could be chosen based on when the perception system is expected to send an accurate pose estimate, and therefore no replanning is needed from there on.

5 Algorithmic framework

Our approach for constant-time planning relies on a preprocessing stage that allows to efficiently compute paths in a query stage to any goal. Before we describe our approach, we start by describing a naïve method that solves the aforementioned problem but requires a prohibitive amount of memory. This can be seen as a warmup before describing our algorithm which exhibits the same traits but does so in a memory-efficient manner.

5.1 Straw man approach

We first compute from $s_{\text{home}}$ a path $\pi_g$ to every reachable $g \in G_{\text{full}}$. These paths can be stored in a lookup (hash) table which can be queried in constant time $T_{\text{const}}$ (assuming perfect hashing Czech et al. (1997)). For the straw man approach, since there is no utility of having a $T_{\text{bound}}$ larger than $T_{\text{const}}$ as it only performs look up operations at query time, we have $T_{\text{bound}} = T_{\text{const}}$. By storing paths to all the goals, every goal is covered by $s_{\text{home}}$ and this allows us to start executing a path once the perception system gives its initial pose estimate. However, we need to account for pose update while executing $\pi_g$. This only needs to be done up until time $t_{\text{rc}}$, since no future improved estimates are expected from the perception system. Thus, we discretize each path uniformly with resolution $\delta_t$. We call all states that are less than $t_{\text{rc}}$ time from $s_{\text{home}}$ replanable states.

Next, for every replanable state along each path $\pi_g$, we compute a new path to all goals. This will ensure that all goals are covered by all replanable states. Namely, it will allow to immediately start executing a new path once the goal location is updated by the perception system. Unfortunately, the perception system may update the goal location more than once. Thus, this process needs to be performed recursively for the new paths as well.

The outcome of the preprocessing stage is a set of precomputed collision-free paths starting at states that are at most $t_{\text{rc}}$ from $s_{\text{home}}$ and end at goal states. The paths are stored in a lookup table $M : S \times G_{\text{full}} \rightarrow \{\pi_1, \pi_2, \ldots\}$ that can be queried in $T_{\text{bound}} = T_{\text{const}}$ time to find a path from any given $s \in S$ to $g \in G_{\text{full}}$.

In the query stage we obtain an estimation $g_i$ of the goal pose by the perception system. The algorithm then retrieves the path $\pi_{i_1}(s_{\text{home}}, g_i)$ from $s_{\text{home}}$ to $g_i$ from $M$ and the robot starts executing $\pi_{i_1}(s_{\text{home}}, g_i)$. For every new estimation $g_i$ of the goal pose obtained from the perception system while the system is executing path $\pi_{i-1}(s_{i-1}, g_{i-1})$, the algorithm retrieves from $M$ the path $\pi_i(s_i, g_i)$ from the nearest state $s_i$ along $\pi_{i-1}(s_{i-1}, g_{i-1})$ that is least $T_{\text{bound}}$ away from $s_{i-1}$. The robot then starts executing $\pi_i(s_i, g_i)$ once it reaches $s_i$. Hence, the straw man algorithm is trivially a CTMP-complete algorithm.

Clearly, every possible goal is covered for every possible configuration that the robot might be in during execution by this brute force approach, however it requires a massive amount of memory and preprocessing time. Let $n_{\text{goal}} = |G_{\text{full}}|$ be the number of goals and $\ell$ be the number of
states between $s_{\text{home}}$ and the state that is $t_{\text{rc}}$ time away. This approach requires precomputing and storing $O(n_{g_{\text{goal}}})$ paths which is clearly infeasible (see Fig. 3). In the next sections, we show how we can dramatically reduce the memory footprint of the approach without compromising on the system’s capabilities.

5.2 Algorithmic approach

While the straw man algorithm presented allows for planning to any goal pose $g \in G_{\text{full}}$ within $T_{\text{const}}$ time, its memory footprint is prohibitively large. We suggest to reduce the memory footprint by building on the observation that many paths to close-by goals traverse very similar parts of the configurations space. Instead of generating a plan strictly within $T_{\text{const}}$ time, our approach trades off preprocessing efforts with the bound on the planning time and guarantees to generate a solution within the user-specified time bound $T_{\text{bound}}$.

The idea is illustrated in Fig. 5.

First, we compute a set of root paths $\{\Pi_1, \ldots, \Pi_k\}$ from $s_{\text{home}}$ to cover $G_{\text{full}}$ by $s_{\text{home}}$ (here we will have that $k \ll n_{g_{\text{goal}}}$). Next, for all replanable states along these root paths, the algorithm recursively computes additional root paths so that their reachable goals are also covered. During this process, additional root paths are computed only when the already existing set of root paths does not provide enough guidance to the search to cover $G_{\text{full}}$ i.e., to be able to compute a path to any $g \in G_{\text{full}}$ within $T_{\text{bound}}$. The remainder of this section formalizes these ideas.

5.3 Algorithmic building blocks

We start by introducing the algorithmic building blocks that we use. Specifically, we start by describing the motion planner $P$ that is used to compute the root paths and then continue to describe how they can be used as experiences to efficiently compute paths to other goals. We implemented two types of motion planners. One operates in a time-configuration space and the other is a kinodynamic motion planner that plans with additional state dimensions of joint velocities in order to satisfy the kinodynamic constraints of the robot. While the former planning framework is simpler and works successfully on a physical robot in our experiments, the latter might be more desirable for operating a robot closer to its maximum performance limits.

We use a heuristic search-based planning approach with motion primitives (see, e.g., Cohen et al. (2010, 2011); Likhachev and Ferguson (2009)) as it allows for deterministic planning time which is key in our domain. Moreover, such planners can easily handle under-defined goals as we have in our setting—we define a goal as a grasp pose for the goal object. The grasp pose for a target object $o$ is manually selected.

5.3.1 Time-Configuration Motion Planner

State space and graph construction. We define a state $s$ as a pair $(q, t)$ where $q = (\theta_1, \ldots, \theta_n)$ is a configuration represented by the joint angles for an $n$-DOF robot arm (in our setting $n = 7$) and $t$ is the time associated with $q$. Given a state $s$ we define two types of motion primitives which are short kinodynamically-feasible motions that the robot can execute.

The first type of motion primitives are predefined primitives. These are small individual joint movements in either direction as well as wait actions. These primitives have non-uniform resolution. For each joint, we define two motion primitives of distance $\pm 1^\circ$. In addition, for the first four of the seven robot joints we define two additional primitives each, of distance $\pm 7^\circ$. We only allow moving one joint at a time which makes it a total of 23 predefined primitives. For each motion primitive, we compute its duration by using a nominal constant velocity profile for the joint that is moved.

The second type of primitives are dynamic primitives. They are generated by the search only at the states that represent the arm configurations where the end effector is close to the object. These primitives correspond to the actual grasping of the object while it is moving. The dynamic primitives are generated by using a Jacobian pseudo inverse-based control law similar to what Menon et al. (2014) used. The desired velocity of the end effector is computed for which the end-effector minimizes the distance to the grasp pose. Once the gripper encloses the object, it moves along with the object until the gripper is closed.

During the search, all motion primitives are checked for validity with respect to collision and joint limits.

Heuristic Function. The search is guided by an efficient and fast-to-compute heuristic function which in our case has two components. The first component drives the search to intercept the object at the right time and the second component guides the search to correct the orientation of the end effector as it approaches the object. Mathematically, our
The dimensionality of the state space hence becomes $\Delta \theta_1$. The primitive are nominal speed for the end-effector to solve the problem. The target object and the speed of the end-effector. We pick a the target object and the end-effector, the velocity of the each joint. Specifically, to generate a successor of a state, we use inverse dynamics to compute the required torques.

The coefficient $\lambda$ is used as a weighting factor.

### 5.3.2 Kinodynamic Motion Planner

The time-configuration motion planner does not guarantee that the plan when transformed into a trajectory will satisfy robot’s torque limits. To this end, we extend the previous motion planner framework to be able to plan within the kinodynamic constraints of the robot, while still being able to handle the conveyor speed of 0.2 $m/s$. In this section we describe the modified state space and the heuristic function for the kinodynamic motion planner.

**State space and graph construction.** We modify the state space to include joint velocities. A state $s$ is a tuple $(q, \dot{q}, t)$ where $\dot{q} = (\dot{\theta}_1, \ldots, \dot{\theta}_n)$ represent velocities for each joint. The dimensionality of the state space hence becomes 15 in our experiments. Similar to $q$, the velocities $\dot{q}$ are also discretized.

We modify the set of predefined motion primitives. The primitives are specified as magnitudes of accelerations for each joint. Specifically, to generate a successor of a state, for the desired accelerations (specified in the primitive), first we use inverse dynamics to compute the required torques. Second, we integrate using the forward dynamics for a fixed time step $dt$ to get the resulting state using Runge-Kutta integration. We use the Orocos Kinematics and Dynamics Library (KDL) to solve for forward/reverse dynamics and the integration*. Let a motion primitive be specified by a vector of accelerations $\ddot{q}$ of size $n$. The two steps to compute the primitive are

$$\tau = \text{ComputeInverseDynamics}(s, \ddot{q})$$

$$s' = \text{Integrate}(s, \tau, dt)$$

Since the robot dynamics is a function of the state $s$ of the robot, these primitives need to be computed on-the-fly during search. In addition to performing kinematic and collision checks to verify motion validity, we discard successors for which $\tau$ exceeds the robot’s torque or velocity limits.

In our experiments, we use 6 motion primitives for each of the 7 joints. These are accelerations $\pm(4, 8, 12)$ deg/s$^2$. We only accelerate or decelerate one joint at a time under these acceleration profiles, thus resulting in 42 primitives. In addition to these primitives, we use a “coasting” primitive that assigns zero acceleration to all the joints.

**Heuristic Function.** The heuristic function we used for the time-configuration space planner gives no guidance for the velocity profiling which is crucial in the case of kinodynamic planning. We, therefore, modify the heuristic in Eq. 1 by introducing an additional term $\Delta \ddot{x}$ that guides the search with respect to the velocity profile.

$$\Delta \ddot{x}(s) = \|\dot{x}^o - \dot{x}^e(s)\|.$$  \hspace{1cm} (2)

Namely, $\Delta \ddot{x}$ is the magnitude of the difference of the target object’s velocity $\dot{x}^o$ and the robot end-effector’s velocity $\dot{x}^e$ at state $s$ in 3D. $\dot{x}^e$ is computed using forward velocity kinematics.

The new heuristic function is given by

$$h(s, g) = \max(\Delta t(s, g), \lambda_1 \cdot \text{ANGLEDIFF}(s, g) + \lambda_2 \cdot \Delta \ddot{x}(s)).$$  \hspace{1cm} (3)

where $\lambda_1$ and $\lambda_2$ are the weighting factor. Intuitively, this additional term guides the search to match the end-effector velocity with $o$’s velocity as the end-effector approaches the object. This increases the likelihood of generating a dynamic primitive that satisfies the kinodynamic constraints of the robot.

### 5.3.3 Graph Search

The states and the transitions implicitly define a graph $G = (S, E)$ where $S$ is the set of all states and $E$ is the set of all transitions defined by the motion primitives. We use Weighted $\text{A}^*$ ($\text{wA}^*$) Pohl (1970) to find a path in $G$ from a given state $s$ to a goal $g$. $\text{wA}^*$ is a suboptimal heuristic search algorithm that allows a tradeoff between optimality and greediness by inflating the heuristic function $h$ by a given weight $w$. The cost of an edge is the time of its traversal.

### 5.3.4 Planning with Experience Reuse

We now show how previously-computed paths which we named as root paths can be reused as experiences in our framework. Given a heuristic function $h$, we define for a root path $\Pi$ and a goal $g \in G^{\text{full}}$ the shortcut state $s_{sc}(\Pi, g)$ as the state on the path $\Pi$ that is closest to $g$ with respect $h$. Namely,

$$s_{sc}(\Pi, g) := \arg\min_{s \in \Pi} h(s, g).$$  \hspace{1cm} (4)

Now, when searching for a path to a goal $g \in G^{\text{full}}$ using root path $\Pi$ as an experience, we add $s_{sc}(\Pi, g)$ as a successor for any state along $\Pi$ (subject to the constraint that the path along $\Pi$ to $s_{sc}$ is collision free). In this manner the search reuses previous experience to quickly reach a state close to $g$.

### 5.4 Algorithmic details

We are finally ready to describe our algorithm describing first the preprocessing phase and then the query phase.

---

*(KDL:https://orocos.org/kdl.html).*
Algorithm 1 Plan Root Paths

1: procedure PLANROOTPATHS($s_{\text{start}}, G_{\text{full}}$)
2: \hspace{1em} \(\Psi_{\text{start}} \leftarrow \emptyset\) \hspace{1em} \(\triangleright\) a list of pairs (\(\Pi_i, G_i\))
3: \hspace{1em} \(G_{\text{start}} \leftarrow \emptyset; \) \hspace{1em} \(i = 0\)
4: while \(G_{\text{full}} \neq \emptyset\) do \hspace{1em} \(\triangleright\) until all reachable goals are covered
5: \hspace{1em} \(g_i \leftarrow \text{SAMPLEGOAL}(G_{\text{full}})\)
6: \hspace{1em} \(G_{\text{uncov}} \leftarrow G_{\text{full}} \setminus \{g_i\}\)
7: if \(\text{Pi} \subseteq \text{PLANROOTPATH}(s_{\text{start}}, g_i)\) then \hspace{1em} \(\triangleright\) planner succeeded
8: \hspace{1em} \(G_i \leftarrow \{g_i\}\)
9: for each \(g_j \in G_{\text{uncov}}\) do
10: \hspace{1em} \(G_i \leftarrow G_i \cup \{g_j\}\)
11: \hspace{1em} \(G_{\text{uncov}} \leftarrow G_{\text{uncov}} \setminus \{g_j\}\)
12: \hspace{1em} \(\Psi_{\text{start}} \leftarrow \Psi_{\text{start}} \cup \{(\Pi_i, G_i)\}; \) \hspace{1em} \(i \leftarrow i + 1\)
13: else
14: \hspace{1em} \(G_{\text{uncov}} \leftarrow G_{\text{uncov}} \cup \{g_i\}\)
15: return \(\Psi_{\text{start}}, G_{\text{uncov}}\)
16: end if
17: end for
18: end while

5.4.1 Preprocessing

Our preprocessing stage starts by sampling a goal \(g_1 \in G_{\text{full}}\) and computing a root path \(\Pi_1\) from \(s_{\text{home}}\) to \(g_1\). We then associate with \(\Pi_1\) the set of goals \(G_1 \subseteq G_{\text{full}}\) such that \(\Pi_1\) can be used as an experience in reaching any \(g_j \in G_1\) within \(T_{\text{bound}}\). Additionally, the experience-based planner is constrained to reuse the root path up to \(t_c\). Thus, all goals in \(G_1\) are covered by \(s_{\text{home}}\). We then repeat this process but instead of sampling a goal from \(G_{\text{full}}\), we sample from \(G_{\text{full}} \setminus G_1\), thereby removing covered goals from \(G_{\text{full}}\) in every iteration. At the end of this step, we obtain a set of root paths. Each root path \(\Pi_i\) is associated with a goal set \(G_i \subseteq G_{\text{full}}\) such that (i) \(\Pi_i\) can be used as an experience for planning to any \(g_j \in G_i\) in \(T_{\text{bound}}\) and (ii) \(\bigcup G_i = \text{REACHABLE}(s_{\text{home}}, G_{\text{full}})\) (i.e., all reachable goals for \(s_{\text{home}}\) in \(G_{\text{full}}\)). Alg. 1 details this step (when called with arguments \((s_{\text{home}}, G_{\text{full}})\)). It also returns a set of unreachable goals that are left uncovered. The process is illustrated in Fig. 5.

So far we explained the algorithm for one-time planning when the robot is at \(s_{\text{home}}\) (\(t = 0\)); we now need to allow for efficient replanning for any state \(s\) between \(t = 0\) to \(t_c\). In order to do so, we iterate through all the states on these root paths and add additional root paths so that these states also cover their respective reachable goals. This has to be done recursively since newly-added paths generate new states which the robot may have to replan from. The complete process is detailed in Alg. 2. The PREPROCESS procedure takes in a state \(s_{\text{start}}\) the goal region that it has to cover \(G_{\text{uncov}}\) and region that it already has covered \(G_{\text{cov}}\). Initially, PREPROCESS is called with arguments \((s_{\text{home}}, G_{\text{full}}, \emptyset)\) and it runs recursively until no state is left with uncovered reachable goals.

At a high level, the algorithm iterates through each root path \(\Pi_i\) (loop at line 12) and for each state \(s \in \Pi_i\) (loop at line 14) the algorithm calls itself recursively (line 18). The algorithm terminates when all states cover their reachable goals. The pseudocode in blue constitutes an additional optimization step which we call “latching” and is explained later in Sec. 5.4.3.

In order to minimise the required computation, the algorithm leverages two key observations:

Algorithm 2 Preprocess

1: procedure TRYLATCHING(s, \(\Psi_{\text{start}}, G_{\text{uncov}}, G_{\text{cov}}\))
2: for each \((\Pi_i, G_i) \in \Psi_{\text{start}}\) do
3: if CANLATCH(s, \(\Pi_i\)) then
4: \(G_{\text{uncov}} \leftarrow G_{\text{uncov}} \setminus G_i\)
5: \(G_{\text{cov}} \leftarrow G_{\text{cov}} \cup G_i\)
6: return \(G_{\text{uncov}}, G_{\text{cov}}\)
7: procedure PREPROCESS($s_{\text{start}}, G_{\text{uncov}}, G_{\text{cov}}$)
8: \(s_{\text{start}}, G_{\text{start}} \leftarrow \text{PLANROOTPATHS}(s_{\text{start}}, G_{\text{full}})\)
9: if \(\Psi_{\text{start}} = \emptyset\) then \(\Psi_{\text{start}} \leftarrow \emptyset\)
10: \(G_{\text{uncov}} \leftarrow G_{\text{cov}} \cup \{G_{\text{start}}\}\)
11: if \(\{G_{\text{start}}\} \subseteq t_c\) then
12: for each \((\Pi_i, G_i) \in \Psi_{\text{start}}\) do
13: \(G_i \leftarrow G_{\text{uncov}} \cup \{G_i\} ; \) \hspace{1em} \(\triangleright\) states up to \(t_c\)
14: \(G_i \leftarrow G_i \cup \text{TRYLATCHING}(s, \Psi_{\text{start}}, G_{\text{uncov}}, G_{\text{cov}})\)
15: if \(G_i = \emptyset\) then break
16: break
17: return \(G_{\text{start}}, G_{\text{cov}}\)

O1 If a goal is not reachable from a state \(s \in \Pi\), it is not reachable from all the states after it on \(\Pi\).

O2 If a goal is covered by a state \(s \in \Pi\), it is also covered by all states preceeding it on \(\Pi\).

O1 is an assumption that we make about the planner \(P\). We use O1 to initialize the uncovered set for any state; instead of attempting to cover the entire \(G_{\text{full}}\) for each replanable state \(s\), the algorithm only attempts to cover the goals that could be reachable from \(s\), thereby saving computation. O2 is used by iterating backwards on each root path (loop at line 14) and for each state on the root path only considering the goals that are left uncovered by the states that appear on the path after it.

Specifically, O2 is used to have a single set of uncovered goals \(G_{\text{uncov}}\) for all states that appear on \(\Pi_i\) instead of having individual sets for each state and the goals that each \(s \in \Pi\) covers in every iteration of loop 14 are removed from \(G_{\text{uncov}}\). O1 is used to initialize \(G_{\text{uncov}}\) (in line 13). Namely, it is initialized not by the entire \(G_{\text{full}}\) but by the set of goals covered by \(s_{\text{start}}\). \(G_i\) is excluded since it is already covered via \(\Pi_i\). The iteration completes either when all goals in \(G_{\text{uncov}}\) are covered (line 19) or the loop backtracks to \(s_{\text{start}}\).

The process is illustrated in Fig. 6.

Thus, as the outcome of the preprocessing stage, a map \(M : S \times G_{\text{full}} \rightarrow \{\Pi_1, \Pi_2, \ldots\}\) is constructed that can be looked up to find which root path can be used as an experience to plan to a goal \(g\) from a state \(s\) within \(T_{\text{bound}}\).

5.4.2 Query Alg. 3 describes the query phase of our algorithm. Again, the lines in blue correspond to the blue pseudocode in Alg. 2 for the additional optimization step which is explained in Sec. 5.4.3. Assume that the robot was

\(^1\)To account for the lower bound \(T_{\text{const}}\) time that is required for the query phase which is consumed in operations, such as hash table lookups etc., the time \(T_{\text{const}}\) is subtracted from \(T_{\text{bound}}\) for the experience-based planner, to ensure that the overall query time is bounded by \(T_{\text{bound}}\). We use a conservative estimate of \(T_{\text{const}}\) in our experiments.
at a state $s_{curr}$ while executing a path $\pi_{curr}$ when it receives a pose update $g$ from the perception system. Alg. 3 will be called for the first time when $s_{start}$ that is $T_{bound}$ ahead of $s_{curr}$ along $\pi_{curr}$, allowing the algorithm to return a plan before the robot reaches $s_{start}$.

Alg. 2 assures that either the first state on $\pi_{curr}$ covers $g$ or there exists one state on $\pi_{curr}$ between $s_{start}$ and the state at $t_{curr}$ that covers $g$. The algorithm first checks if the first state on $\pi_{curr}$ covers $g$ or not. If it covers $g$ then the corresponding root path is used to find the new path $\pi_{new}$. Otherwise, it iterates over each $s \in \pi_{curr}$ backwards (similar to Alg. 2) between $s_{start}$ and the state at $t_{curr}$ and finds the one that covers $g$ by querying $\mathcal{M}$. Once found, the corresponding root path $\Pi_{next}$ is used as an experience to plan the path $\pi_{next}$ from $s$ to $g$. Finally the paths $\pi_{curr}$ and $\pi_{next}$ are merged together with $s$ being the transitioning state to return the final path $\pi$.

**5.4.3 Latching: Reusing Root Paths** We introduce an additional step called “Latching” to minimise the number of root paths computed in Alg. 2. With latching, the algorithm tries to reuse previously-computed root paths as much as possible using special motion primitives that allow transitions from one root path to another.

For the time-configuration motion planner, the primitive is computed from a state $s \in \Pi_1$ to $s' \in \Pi_1$ such that $t(s') = t(s) + \delta_t$ by simple linear interpolation while ensuring the feasibility of the motion. Specifically, given the nominal joint velocities of the robot, if $s'$ can be reached from $s$ in time $\delta_t$, while respecting the kinematic and collision constraints, then the transition is allowed.

Clearly, this approach does not ensure that the torque limits are respected. Therefore, for the kinodynamic planner,

```
Algorithm 3 Query

Inputs: $\mathcal{M}, s_{home}$
1: procedure PLANPATHBYLATCHING($s, g, \pi_{curr}$)
2: if $s_{home} = \mathcal{M}(s_{start}, g)$ exists then $\triangleright$ lookup root path
3: if CANLATCH($s, \Pi_{home}$) then
4: $\Pi_{home} = \text{PLANPATHWITHEXPERIENCE}(s_{home}, g, \Pi_{home})$
5: $\pi = \text{MERGEPATHSBYLATCHING}($($\pi_{curr}, \Pi_{home}$, $g$))
6: return $\pi$
7: return failure
8: procedure QUERY($g, \pi_{curr}, s_{start}$)
9: $s \leftarrow \pi_{curr}[0] \triangleright$ first state on $\pi_{curr}$
10: if $\pi_{curr} = \mathcal{M}(s, g)$ exists then $\triangleright$ lookup root path
11: $\pi_{new} \leftarrow \text{PLANPATHWITHEXPERIENCE}(s, g, \Pi_{curr})$
12: return $\pi_{new}$
13: for each $s \in \pi_{curr}$ (from last to $s_{curr}$) do $\triangleright$ states up to $t_{curr}$
14: if $\Pi_{next} = \mathcal{M}(s, g)$ exists then $\triangleright$ lookup root path
15: $\pi_{next} = \text{PLANPATHWITHEXPERIENCE}(s, g, \Pi_{next})$
16: $\pi = \text{MERGEPATHS}($($\pi_{curr}, \pi_{next}, s$))
17: return $\pi$
18: if $\pi = \text{PLANPATHBYLATCHING}($($s, g, \pi_{curr}$)) successful then
19: return $\pi$
20: return failure $\triangleright$ goal is not reachable
```
we interpolate by fitting a cubic polynomial from the state $s \in \Pi_i$ to $s' \in \Pi_i$ that satisfies the boundary conditions. We then ensure motion validity by checking the joint velocity and torque limits of each state along the interpolated trajectory.

In Alg. 2, before calling the PREPROCESS procedure for a state, the algorithm removes the set of goals that can be covered via latching, thereby reducing the number of goals that need to be covered by the PREPROCESS procedure. Correspondingly, in Alg. 3, an additional procedure is called to check if the path can be found via latching. These additions in the two pseudocodes are shown in blue. An iteration of the complete algorithm with latching is illustrated in Fig 7.

5.5 Theoretical Analysis

5.5.1 CTMP Properties

Lemma 1. For any state $s_{\text{start}}$ of the robot during execution, provided $t(s_{\text{start}}) \leq t_{\text{rc}}$, the algorithm is CTMP.

Proof. We can prove it by showing that the query stage (Alg. 3) returns within $T_{\text{bound}}$ time and has a constant-time complexity. The number of times the algorithm queries $\mathcal{M}$, which is an $O(1)$ operation assuming perfect hashing, is bounded by $O(1)$ where $l = t_{\text{rc}}/\delta_t$ is the maximum number of time steps from $t = 0$ to $t_{\text{rc}}$. The number of times the algorithm attempts to latch on to a root path (namely, a call to CANATCH which is a constant-time operation) is also bounded by $l$. As $l$ is constant for of a fixed time cut off $t_{\text{rc}}$, the execution time of the aforementioned operations constitute $T_{\text{const}}$ (constant-time). Finally, Alg. 3 calls the PLAN method only once. Since the execution time of PLAN is bounded by $T_{\text{bound}}- T_{\text{const}}$, the overall execution time of Alg. 3 is $T_{\text{bound}}$. Hence it is a CTMP algorithm.

Lemma 2. For any state $s_{\text{start}}$ of the robot during execution, provided $t(s_{\text{start}}) \leq t_{\text{rc}}$, the algorithm is CTMP-complete.

Proof. In order to prove it we need to show that for any $s_{\text{start}}$ that the system can be at, (1) if a $g$ is reachable from $s_{\text{start}}$, it is covered by $s_{\text{start}}$ in Alg. 2 and (2) if $g$ is covered by $s_{\text{start}}$, Alg. 3 is guaranteed to return a path within $T_{\text{bound}}$ time.

Alg. 2 starts by computing a set of root paths from $s_{\text{home}}$ that ensures that it covers all of its reachable goals. It then iterates over all states on these paths and adds additional root paths ensuring that these states also cover their reachable goals. It does it recursively until no state $s_{\text{start}}$ before $t_{\text{rc}}$ is left with any uncovered $g$ which could be reachable.

Alg. 2 covers $g$ via at least one state between $s_{\text{start}}$ and the state at $t_{\text{rc}}$ ( inclusively) (loop at line 14). In query phase, Alg. 3 iterates through all states between $s_{\text{start}}$ and the state at $t_{\text{rc}}$ (inclusively) to identify the one that covers $g$ (loop at line 13). Since $g$ is covered by at least one of these states by Alg. 2, Alg. 3 is guaranteed to find a path from $s_{\text{start}}$ to $g$. Moreover, from lemma 1 we have that the path is returned within $T_{\text{bound}}$ time. Hence the algorithm is CTMP-complete.

5.5.2 Space Complexity

Lemma 3. Let $n_{\Pi}$ be the maximum number of root paths needed to cover a goal region for a given state $s_{\text{start}}$ and $\ell'$ be the number of discretized time steps at which the algorithm computes root paths, then algorithm requires $O(n_{\Pi}^{\ell'})$ space.

Proof. At first, the algorithm stores $n_{\Pi}$ paths (worst case) from $s_{\text{home}}$ to $G^{\text{full}}$ consuming $O(n_{\Pi})$ space. It then computes root paths starting with the states at $t_{\text{rc}}$ and iterating backwards to previous time steps (see Alg. 2 line 14). The loop terminates when no more uncovered goals are left. If $\ell' - 1$ is the maximum number of iterations of loop at line 14 that Alg. 2 undergoes, then it requires $O(n_{\Pi}^{\ell'})$ space.

Note that Alg. 2 reduces the space complexity of the naive approach (Sec. 5) primarily by (1) compressing the number of paths required to cover a goal region from $n_{\text{goal}}$ to $n_{\Pi}$ by planning with experience reuse and (2) compressing the number of time steps for which the algorithm requires preprocessing for from $\ell$ to $\ell'$ by leveraging O2. Additionally, the latching feature allows the algorithm to further reduce the required number of paths by a significant amount.

Pertaining to the structure of our problem, the algorithm works under the assumption that $n_{\Pi} \ll n_{\text{goal}}$ and $\ell' \ll \ell$. To demonstrate the magnitude of compression, in preprocessing for the time-configuration planner (see Table. 2a), the algorithm covers 7197 goals with only nine root paths for $s_{\text{home}}$ and only computes root paths for a single time step $t = 0$ (i.e., $\ell' = 1$) out of a total of eight time steps up to $t_{\text{rc}}$.

6 Evaluation

We evaluated our algorithm in simulation and on a real robot. The conveyor speed that we used for all of our results was $0.2 m/s$. We used Willow Garage’s PR2 robot in our experiments using its 7-DOF arm. For the real robot experiments we tested with the time-configuration planner whereas for the simulated results we tested both for the time-configuration and kinodynamic planners. The additional time dimension made the planning problem eight dimensional. The experiments can be viewed at https://youtu.be/ILVP18wa5b8.

6.1 Experimental setup

6.1.1 Sense-plan-act cycle As object $o$ moves along the conveyor belt, we use the Brute Force ICP pose estimation baseline proposed in Narayanan and Likhachev (2016) to obtain its 3-Dof pose for each captured input point cloud. We follow the classical sense-plan-act cycle as depicted in Fig. 8. Specifically, the perception system captures an image (point cloud) of the object $o$ at time $t_{\text{msg}}$ followed by a period of duration $T_{\text{perception}}$ in which the perception system estimates the pose of $o$. At time $t_{\text{msg}} = t_{\text{msg}} + T_{\text{perception}}$, planning starts for a period of $T_{\text{planning}}$ which is guaranteed to be less than $T_{\text{bound}}$. Thus, at $t_{\text{plan}} = t_{\text{msg}} + T_{\text{perception}}$, the planner waits for an additional duration of $T_{\text{wait}} = T_{\text{bound}} - T_{\text{planning}}$. Finally, at $t_{\text{exec}} = t_{\text{plan}} + T_{\text{wait}}$, the robot starts executing the plan. Note that the goal $g$ that the planner plans for is not for the object pose at $t_{\text{msg}}$ but its forward projection in time to $t_{\text{exec}}$ to account for $T_{\text{perception}}$ and $T_{\text{bound}}$. While executing the plan, if we obtain an updated pose estimate, the execution is preempted and the cycle repeats.
Table 1. Simulation results averaged over 50 experiments. Here $T_s$ denotes the (possibly arbitrary) time bound that the algorithm uses. Note that for our method $T_b = T_{\text{bound}}$ is the time bound that the algorithm is ensured to compute a plan. CTMP algorithm uses a fixed $T_b = 0.2s$. For the baselines, since their planning times are unbounded, we test them with $T_b = 0.5, 1.0$ and $2.0s$. Our CTMP approach shows a task and planning success rate of 100% for both the planners. Among the baselines, E-Graph shows the best performance, however its performance drops significantly for the kinodynamic planning problem.

|                | CTMP | wA* | E-Graph |
|----------------|------|-----|---------|
|                | $T_b = 0.2$ | $T_b = 0.5$ | $T_b = 1.0$ | $T_b = 2.0$ | $T_b = 0.5$ | $T_b = 1.0$ | $T_b = 2.0$ |
| Pickup success [%] | 100 | 0.0 | 0.0 | 18.0 | 0.0 | 0.0 | 80.0 | 0.0 | 0.0 | 18.0 |
| Planning success [%] | 100 | 14.0 | 27.0 | 44.0 | 35.0 | 22.0 | 38.0 |
| Planning cycles | 3 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| Path cost [s] | 9.82 | 8.74 | 9.15 | 9.39 | 8.63 | 8.72 | 8.15 |

(a) Simulation results—Time-configuration Planner.

(b) Simulation results—Kinodynamic Planner.

6.1.2 Goal region specification To define the set of all goal poses $G^{\text{full}}$, we need to detail our system setup, depicted in Fig. 9. The conveyor belt moves along the $x$-axis from left to right. We pick a fixed $x$-value termed $x_{\text{exec}}$, such that when the incoming $o$ reaches $x_{\text{exec}}$ as per the perception information, at that point we start execution. Recall that a pose of an object $o$ is a three dimensional point $(x, y, \theta)$ corresponding to the $(x, y)$ location of $o$ and to its orientation (yaw angle) along the conveyor belt. $G^{\text{full}}$ contains a fine discretization of all possible $x, y$ and $\theta$ values in $[x_{\text{exec}} - 2\varepsilon, x_{\text{exec}} + 2\varepsilon]$. We select $G^{\text{full}}$ such that $\varepsilon = 2.5cm$, making the goal region 10cm long along $x$-axis. Its dimension along $y$-axis is 20cm, equal to the width of the conveyor belt. The discretization in $x, y$ and $\theta$ is 1.0cm and 10deg respectively.

In the example depicted in Fig. 9, the thick and the thin solid rectangles show the ground truth and estimated poses, respectively at two time instances in the life time of the object. The first plan is generated for the pose shown at $x_{\text{exec}}$. During execution, the robot receives an improved estimate and has to replan for it. At this point we back project this new estimate in time using the known speed of the conveyor and the time duration between the two estimates. This back-projected pose (shown as the dotted rectangle) is then picked as the new goal for replanning. Recall that under our assumption about the pose error in perception being $\varepsilon$, the back projected pose will always lie inside $G^{\text{full}}$.

6.2 Results

6.2.1 Real-robot experiments To show the necessity of real-time replanning in response to perception updates, we performed three types of experiments: using our approach to replan every time new object pose estimate arrives, single-shot planning based on the first object pose estimate and single-shot planning using the late (more accurate) pose estimate. For each set of experiments, we determined the pickup success rate to grasp the moving object (sugar box) off the conveyor belt. In addition, we report on the perception system’s success rate by observing the overlap between the point cloud of the object’s 3D model transformed by the predicted pose (that was used for planning) and the filtered input point cloud containing points belonging to the object. A high (low) overlap corresponds to an accurate (inaccurate) pose estimate. We use the same strategy to determine the range for which the perception system’s estimates are accurate and use it to determine the time for the best-pose planning. Further, for each method, we determine the pickup success rate given that the perception system’s estimate was or was not accurate.

The experimental results are shown in Table 3. Our method achieves the highest overall pickup success rate on the robot by a large margin, indicating the importance of continuous replanning with multiple pose estimates. First-pose planning has the least overall success rate due to inaccuracy of pose estimates when the object is far from the robot’s camera. Best-pose planning performs better overall than the first pose strategy, since it uses accurate pose estimates, received when the object is close to the robot. However it often fails even when perception is accurate, since a large number of goals are unreachable due to limited
time remaining to grasp the object when it is closer to the robot. A demonstration of our approach is given in Fig. 10.

6.2.2 Simulation experiments We simulated the real world scenario to evaluate our method against other baselines. We compared our method with wA* Pohl (1970), E-graph Phillips et al. (2012) and RRT LaValle (1998). For wA* and E-graph we use the same graph representation as our method. For E-graph we precompute five paths to randomly-selected goals in $G^{\text{full}}$. We adapt the RRT algorithm to account for the under-defined goals. To do so, we sample pre-grasp poses along the conveyor and compute IK solutions for them to get a set of goal configurations for goal biasing. When a newly-added node falls within a threshold distance from the object, we use the same dynamic primitive that we use in the search-based methods to add the final grasping maneuver. If the primitive succeeds, we return success. We also allow wait actions at the pre-grasp locations.

For any planner to be used in our system, we need to endow it with a (possibly arbitrary) planning time bound to compute the future location of the object from which the new execution will start. If the planner fails to generate the plan within this time, the robot fails to react to that pose update and such cases are recorded as planning failure. We label a run as a pickup success if the planner successfully replans once after the object crosses the 1.0m mark. The mark is the mean of accurate perception range that was determined experimentally and used in the robot experiments as described in Section 6.2.1. The key takeaway from our experiments (Table 1) is that having a known time bound on the query time is vital to the success of the conveyor pickup task.

Our method shows the highest pickup success rate, planning success rate (success rate over all planning queries) and an order of magnitude lower planning times compared to the other methods. We tested the other methods with several different time bounds. After our approach E-graph performed decently well. RRT suffers from the fact that the goal is under-defined and sampling based planners typically require a goal bias in the configuration space. Another important highlight of the experiments is the number of planning cycles over the lifetime of an object. While the other approaches could replan at most twice, our method was able to replan thrice due to fast planning times.

### 6.2.3 Preprocessing

The statistics of the preprocessing stage (i.e. running Alg. 2) are shown in Table 2. The offline time bound $T_p$ used in all of our experiments was 10s. In all experiments, we used $t_{rc}=3.5s$ and $\delta_e=0.5s$. For the time-configuration planner the preprocessing took 2,534s. Only nine root paths are computed to cover 7,197 goals (three goals being unreachable and hence uncovered). For the states at $t = t_{rc}(3.5s)$, there were no latching failures and therefore, the algorithm terminates without preprocessing for earlier time stamps. For the kinodynamic planner, the preprocessing takes about 7 hours. The dynamic constraints causes latching failures and therefore, the algorithm requires more preprocessing efforts. Due to latching failures at $t = 3.5$, it needs to compute root paths for some of the states at this time step. Finally, it covers all the uncovered goals for states at $t = 3.0$ via latching and finishes preprocessing.

| State time | Num. of states | Unreachable goals | Covered goals | Covered via root paths | Covered via Latching | Num. of root paths | Latching tries | Latching failures | Processing time |
|------------|----------------|-------------------|---------------|------------------------|----------------------|-------------------|---------------|------------------|-----------------|
| 0          | 1              | 2                 | 7197          | 7197                   | 0                    | 9                 | 1641          | 0                | 0.235           |
| 3.5        | 9              | 3                 | 7197          | 7197                   | 0                    | 7197              | 0             | 0                | 0.01            |

Table 2. Preprocessing statistics—We report the preprocessing statistics for the robot states from which the algorithm either computes a latching primitive or computes new root paths for replanning. The statistics are indexed based on the time step of the states. For each time step, we give the number of states at that time step and the average values for each preprocessing metric over all the states at that time stamp.

### 7 Conclusion and Discussion

To summarize, we developed a provably constant-time planning and replanning algorithm that can be used to grasp fast moving objects off conveyor belts and evaluated it in simulation and in the real world on the PR2 robot. Through this work, we advocate the need for algorithms that guarantee (small) constant-time planning for time-critical applications, such as the conveyor pickup task, which are often encountered in warehouse and manufacturing environments. To this end we introduce and formalize a new class of algorithms called CTMP algorithms.

An interesting future research direction could be to leverage roadmap-based representation instead of storing individual paths in a way that the algorithm remains CTMP-complete, namely it maintains constant-time planning guarantee. On the more practical side, a useful extension could be to parallelize the preprocessing computation over multiple CPU cores. One obvious way of doing so is within the Alg. 1. The required uncovered goal region $G_{\text{uncov}}$ can be divided over multiple threads.

Another useful extension to our CTMP approach is to be able to handle multiple objects simultaneously coming on the conveyor belt. This setting is common for a sorting task when multiple robot arms work at the same conveyor and they have to pick up one object while avoiding collisions with the other object. This makes the problem more challenging since the
|                  | Success rate | Accuracy of perception [%] | Success rate (Accurate perception) | Success rate (Inaccurate perception) |
|------------------|--------------|----------------------------|-----------------------------------|-------------------------------------|
| CTMP             | 69.23        | 42.31                      | 83.33                             | 57.14                               |
| First-pose Planning | 16.00      | 24.00                      | 66.67                             | 0.00                                |
| Best-pose Planning | 34.61        | 34.62                      | 55.56                             | 23.53                               |

Table 3. Real-robot experiments. Success rate for the three experiments (Our method (CTMP), First-pose planning and Best-pose planning) averaged over 50 trials.

Fig. 10. Snapshots from a real robot experiment. (a) The robot receives the first pose estimate from the perception system, generates the first plan within $T_{\text{bound}}$ and starts execution. (b) The robot receives the second pose estimate with a pose correction of distance 3cm and replans to the new goal (c) The robot receives the third and last pose estimate with no further correction and hence continues to follow the previous plan. (d) and (e) The robot executes the dynamic motion primitive to reach the grasp pose and glide along with the object. (f) The robot lifts up the sugar box from the conveyor.

planner has to consider more than one dynamic objects in the scene.

Acknowledgements

This work was supported by the ONR grant N00014-18-1-2775 and the ARL grant W911NF-18-2-0218 as part of the A2I2 program. In addition, it was partially supported by the Israeli Ministry of Science & Technology grant No. 102583 and by Grant No. 1018193 from the United States-Israel Binational Science Foundation (BSF) and by Grant No. 2019703 from the United States National Science Foundation (NSF).

The authors would like to thank Andrew Dornbush for providing support for the Search-Based Motion Planning Library (https://github.com/aurone/smpl) which is used in our implementation. The authors would also like to thank Ellis Ratner for fruitful discussions and Anirudh Vemula for helping out in restoring the PR2 robot which is used in our experiments.

References

Allen PK, Timcenko A, Yoshimi B and Michelman P (1993) Automated tracking and grasping of a moving object with a robotic hand-eye system. TRA 9(2): 152–165.

Belghith K, Kabanza F, Hartman L and Nkambou R (2006) Anytime dynamic path-planning with flexible probabilistic roadmaps. In: Proceedings of the IEEE International Conference on Robotics and Automation, 2006. ICRA 2006. IEEE, pp. 2372–2377.

Berenson D, Abbeel P and Goldberg K (2012) A robot path planning framework that learns from experience. In: Proceedings of the IEEE International Conference on Robotics and Automation (ICRA). pp. 3671–3678.

Cefalo M, Oriolo G and Vendittelli M (2013) Task-constrained motion planning with moving obstacles. In: 2013 IEEE/RSJ International Conference on Intelligent Robots and Systems. IEEE, pp. 5758–5763.

Cohen BJ, Chitta S and Likhachev M (2010) Search-based planning for manipulation with motion primitives. In: Proceedings of the IEEE International Conference on Robotics and Automation (ICRA). pp. 2902–2908.

Cohen BJ, Subramania G, Chitta S and Likhachev M (2011) Planning for Manipulation with Adaptive Motion Primitives. In: Proceedings of the IEEE International Conference on Robotics and Automation (ICRA). pp. 5478–5485.

Coleman D, Şucan IA, Moll M, Okada K and Correll N (2015) Experience-based planning with sparse roadmap spanners. In: Proceedings of the IEEE International Conference on Robotics and Automation (ICRA). pp. 900–905.

Cowley A, Cohen B, Marshall W, Taylor CJ and Likhachev M (2013) Perception and motion planning for pick-and-place of dynamic objects. In: Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). pp. 816–823.

Czech ZJ, Havas G and Majewski BS (1997) Perfect hashing. Theoretical Computer Science 182(1-2): 1–143.
Dobson A and Bekris KE (2014) Sparse roadmap spanners for asymptotically near-optimal motion planning. *The International Journal of Robotics Research (IJRR)* 33(1): 18–47.

Fraichard T (1993) Dynamic trajectory planning with dynamic constraints: A ‘state-time space’ approach. In: IROS, volume 2. pp. 1393–1400.

Han SD, Feng SW and Yu J (2019) Toward Fast and Optimal Robotic Pick-and-Place on a Moving Conveyor. *IEEE Robotics and Automation Letters (RA-L)* 5(2): 446–453.

Hinterstoisser S, Lepetit V, Ilic S, Holzer S, Bradski G, Konolige K and Navab N (2012) Model based training, detection and pose estimation of texture-less 3D objects in heavily cluttered scenes. In: *Proceedings of the Asian Conference on Computer Vision*. pp. 548–557.

Ishida T and Korf RE (1991) Moving Target Search. In: *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, volume 91. pp. 204–210.

Ishida T and Korf RE (1995) Moving-target search: A real-time search for changing goals. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 17(6): 609–619.

Islam F, Salzman O, Agarwal A and Likhachev M (2020) Provably Constant-time Planning and Replanning for Real-time Grasping Objects off a Conveyor Belt. In: *Proceedings of Robotics: Science and Systems (RSS)*.

Islam F, Salzman O and Likhachev M (2019) Provably Indefinite-Horizon Real-Time Planning for Repeatitive Tasks. In: *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*. pp. 716–724.

Kavraki LE, Svestka P, Latombe JC and Overmars MH (1996) Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Robotics and Automation Letters (RA-L)* 12(4): 566–580.

Koenig S and Likhachev M (2006) Real-time adaptive A* . In: *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*. pp. 281–288.

Koenig S, Likhachev M and Sun X (2007) Speeding up moving-target search. In: *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*. pp. 1–8.

Koenig S and Sun X (2009) Comparing real-time and incremental heuristic search for real-time situated agents. *Autonomous Agents and Multi-Agent Systems* 18(3): 313–341.

Korf RE (1990) Real-time heuristic search. *Artificial intelligence* 42(2-3): 189–211.

LaValle SM (1998) Rapidly-exploring random trees: A new tool for path planning .

Likhachev M and Ferguson D (2009) Planning Long Dynamically Feasible Maneuvers for Autonomous Vehicles. *The International Journal of Robotics Research (IJRR)* 28(8): 933–945.

Menon A, Cohen B and Likhachev M (2014) Motion planning for smooth pickup of moving objects. In: *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*. pp. 453–460.

Narayanan V and Likhachev M (2016) PERCH: Perception via search for multi-object recognition and localization. In: *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*. pp. 5052–5059.

Phillips M, Cohen BJ, Chitta S and Likhachev M (2012) E-Graphs: Bootstrapping Planning with Experience Graphs. In: *Proceedings of Robotics: Science and Systems (RSS)*.

Pohl I (1970). *Artificial intelligence* 1(3-4): 193–204.

Salzman O, Shaharabani D, Agarwal PK and Halperin D (2014) Sparsification of motion-planning roadmaps by edge contraction. *The International Journal of Robotics Research (IJRR)* 33(14): 1711–1725.

Stogi D, Zumkeller D, Navarro SE, Heilig A and Hein B (2017) Tracking, reconstruction and grasping of unknown rotationally symmetrical objects from a conveyor belt. In: *IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE, pp. 1–8.

Sun X, Yeoh W and Koenig S (2010) Moving target D* lite. In: *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*. pp. 67–74.

Yang Y, Merkt W, Ivan V and Vijayakumar S (2018) Planning in time-configuration space for efficient pick-and-place in non-static environments with temporal constraints. In: *2018 IEEE-RAS 18th International Conference on Humanoid Robots (Humanoids)*. IEEE, pp. 1–9.

Yoshida E and Kanehiro F (2011) Reactive robot motion using path replanning and deformation. In: *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, pp. 5456–5462.

Zhang Y, Li L, Ripperger M, Nicho J, Veeraraghavan M and Fumagalli A (2018) Gilbreth: A conveyor-belt based pick-and-sort industrial robotics application. In: *International Conference on Robotic Computing (IRC)*. pp. 17–24.