# Provably Constant-Time Planning and Re-planning for Real-time Grasping Objects off a Conveyor

Fahad Islam[1], Oren Salzman[2], Aditya Agarwal[1], Maxim Likhachev[1]
[1]The Robotics Institute, Carnegie Mellon University
[2]Technion-Israel Institute of Technology

***Abstract*—In warehousing 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 often is 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 execution. 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 an approach that meets these requirements by providing provable constant-time planning and replanning guarantees. We present it, give its analytical properties and show experimental analysis in simulation and on a real robot.**

## I. 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 [26]. 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 field of view of the robot, 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. In order to understand how pose estimation error varies as the object gets closer, we filter the input point cloud and remove all points lying outside the conveyor. We then compute the pair-wise minimum distance between the input point cloud and a predicted point cloud that corresponds to the object's 3D model transformed with the predicted pose through ICP [3] refinement. The plot in Fig. 2 shows the variation of the average pairwise minimum distance between the two point clouds as the object gets closer. We can observe that the distance decreases as the object moves closer, indicating that the point clouds overlap more closely due to more accurate pose estimates closer to the camera.

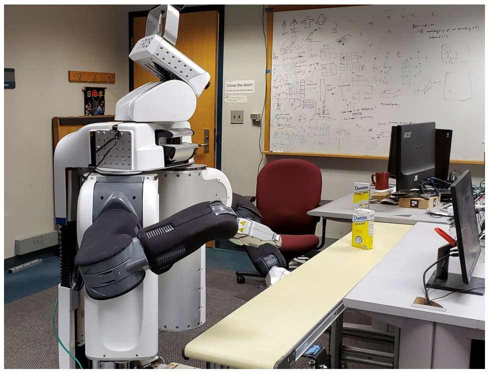
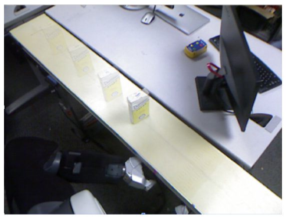


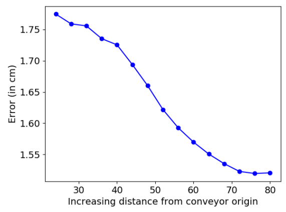
Fig. 1: A scene demonstrating the PR2 robot picking up a moving object (sugar box) off a conveyor belt.

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 increases proportionately with the speed of the conveyor. Therefore, the robot should start executing a plan computed for the initial pose and as it gets better estimates, it should repeatedly replan for the new goals. However, for every replanning query, the time window for the pickup shrinks. This makes the planner's job difficult to support real-time planning.

Furthermore, the planning problem is challenging because the motion planner has to account for the dynamic object and thus plan with time as one of the planning dimension. It should generate a valid trajectory that avoids collision with the environment around it and also with the target object to ensure that it does not damage or topple it during the grasp. Avoiding collisions with the object requires precise geometric collision checking between the object geometry and the geometry of the manipulator. The resulting complexity of the planning problem makes it infeasible to plan online for this task.

Motivated by these challenges, we propose a planning framework that leverages offline preprocessing to provide bounds on the planning time when the planner is invoked online. Our key insight is that in our domain the manipulation task is highly repetitive—even for different object poses, the computed paths


(a)



(b)

Fig. 2: (a) Depiction of an object moving along a conveyor towards the robot. (b) Pose error as a function of the distance from the conveyor's start.

are quite similar and can be efficiently reused to speed up online planning. Based on this insight, we derive a method that precomputes a representative set of paths offline with some auxiliary datastructures and uses them online to plan in constant time. Here, we assume that the models of the target objects are known. Namely, the planner is provided with the geometric model of the target object apriori. To the best of our knowledge, our approach is the first to provide provable constant-time planning guarantees on generating motions all the way to the goal for dynamic environments.

We experimentally show that constant-time planning and re-planning capability is necessary for a successful conveyor pickup task. Specifically if we only perform one-time planning, (namely, either following the plan for the initial noisy pose estimate or from a delayed but accurate pose estimate) we frequently fail to pick the object.

## II. Related work

### A. *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 to object perception and pose estimation, motion planning and others [1, 9, 24, 26]. Here, we focus on motion-planning related work. Graph-searched based approaches have been used for the motion-planning problem [7, 20]. 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 optimal trajectories with respect to the time of execution was used. While this planner provides strong optimality guarantees, it is not real-time and thus cannot be used online. The latter work was demonstrated in the real world showing real-time planning capability. While the planner is fast, it does pure kinematic planning and does not require collision checking with the target object. The approach plans to a pregrasp pose and relies on Cartesian-space controllers to perform the pick up. The usage of the Cartesian controller limits the types of objects that the robot can grasp.

### B. *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 Probablistic Roadmap Method (PRM) [14] 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. Moreover for a moving object (as we have in our setting), they would require edge re-evaluation which is often computationally expensive.

A provably constant-time planner was recently proposed in [12]. 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 state within the goal region in bounded time. As we will see, our approach bears close resemblance with this work in the context of the paths-compression mechanism.

Using either of these two methods ([12, 14]) in our context is not straightforward as they are only applicable to pure kinematic planning and thus they cannot be used for the conveyor-planning problem.

Another family of preprocessing-based planners utilises previous experiences to speed up the search [2, 6, 21]. Experience graphs [21], 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., [8, 23]) to reduce the memory footprint of the algorithm. Unfortunately, none of the mentioned algorithms provide bounded planning-time guarantees that are required by our application.

### C. *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 [10, 11, 17, 25]. 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.

Similar to MTS, real-time planning was widely considered in the search community (see, e.g., [15, 16, 18]). However, as mentioned, these works are typically applicable to low-dimensional search spaces. Finally it is worth mentioning that some finite-time properties have been obtained for sampling-based planners [13] but these come in form of big-$O$ notation, thus they are not applicable to our setting, when we are aiming for tight planning times (e.g., less than one second).

## III. Problem definition

Our system is comprised of a robot manipulator $\mathcal{R}$, a conveyor belt $\mathcal{B}$ moving at some known velocity, a set of known objects $\mathcal{O}$ that need to be grasped and a perception system $\mathcal{P}$ that is able to estimate the type of object and its location on $\mathcal{B}$.

Given a pose $g$ of an object $o \in \mathcal{O}$, our task is to plan the motion of $\mathcal{R}$ such that it will be able to pick $o$ from $\mathcal{B}$ at some future time. Unfortunately, the perception system $\mathcal{P}$ may give inaccurate object poses. Thus, the pose $g$ will be updated by $\mathcal{P}$ as $\mathcal{R}$ is executing its motion. To allow for $\mathcal{R}$ to move towards the updated pose in real time, we introduce the additional requirement that planning should be done within a predefined 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 $\mathcal{P}$, we mean that we plan the motion of $\mathcal{R}$ such that it will be able to pick $o$ from $\mathcal{B}$ at some future time. This is explained in detail in Sec. V and in Fig. 7.

We denote by $G^{\text{full}}$ the discrete set of initial object poses on $\mathcal{B}$ that $\mathcal{P}$ can perceive. Finally, we assume that $\mathcal{R}$ has an initial configuration $s_{\text{home}}$ from which it will start its motion to grasp any object.

Roughly speaking, the objective, following the set of assumptions we will shortly state, is to enable planning to any goal pose $g \in G^{\text{full}}$ in bounded time $T_{\text{bound}}$ regardless of $\mathcal{R}$'s current configuration. To formalize this idea, let us introduce the notion of a *reachable* state:
**Definition 1.** *A goal pose $g \in G^{full}$ is said to be* reachable *from a state $s$ if there exists a path from $s$ to $g$ and it can be computed in finite time.*

Given a state $s$ we denote the set of all goals that are reachable from $s$ as $G^{\text{reach}}(s)$ and we say that $G^{\text{reach}}(s)$ is *reachable* from $s$.
**Definition 2.** *A reachable pose $g \in G^{full}$ is said to be* covered *by a state $s$ if the system can plan a path from $s$ to $g$ within time $T_{bound}$.*

Thus, we wish to build a system such that for any state $s$ that the system can be in and every reachable goal pose $g \in G^{\text{full}}$ updated by $\mathcal{P}$, $g$ is covered by $s$.

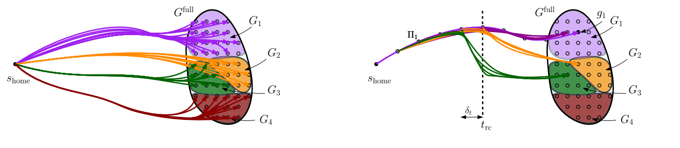
We are now ready to state the assumptions for which we can solve the problem defined.

We make the following assumptions about the system.

**A1** There exists a replan cutoff time $t_{\text{rc}}$ from when the robot starts moving, after which the planner does not replan and continues to execute the last planned path.

**A2** If the robot starts moving at $t = 0$ then for any time $t < t_{\text{rc}}$, the environment is static. Namely, objects on $\mathcal{B}$ cannot collide with $\mathcal{R}$ during that time.

**A3** Given an initial goal pose $g_{\text{init}} \in G^{\text{full}}$ by $\mathcal{P}$, any subsequent pose $g_{\text{new}} \in G^{\text{full}}$ is at most $\varepsilon_{\mathcal{P}}$ distance away from $g_{\text{init}}$ (after accounting for the fact that the object moved along $\mathcal{B}$).



(a) (b)

Fig. 3: The two steps of the preprocessing stage of the straw man algorithm. In both figures, paths that are very similar (and their corresponding goal states) are depicted using the same color. (a) First step—the algorithm computes a path from $s_{\text{home}}$ to every state in $G^{\text{full}}$. (b) Second step—the algorithm computes for each path a new path to each goal state for every state along the path. Here, one representative path is depicted together with three different goal states. This is repeated recursively for the new paths but not visualized.

Assumptions **A1**-**A2** enforce a requirement that $\mathcal{P}$ must provide an accurate estimate $g$ while $o$ is at a safe distance from $\mathcal{R}$. Assumption **A3** corresponds to the error tolerance of the perception system. This is explained in detail in Sec. V and in Fig. 7

## IV. Algorithmic framework

Our approach for bounded-time planning relies on a *preprocessing* stage that allows to efficiently compute paths in a *query* stage to any goal state (under Assumptions **A1**-**A3**). 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 doing so in a memory-efficient manner.

### A. Straw man approach

We divide the preprocessing stage into two steps. In the first step, we compute from $s_{\text{home}}$ a path $\Pi_g$ to every reachable $g \in G^{\text{full}}$. . Thus, all goal states are covered by $s_{\text{home}}$ and this allows us to start executing a path once $\mathcal{P}$ gives its initial pose estimate. However, we need to allow for updated pose estimations while executing $\Pi_g$.

Following **A1** and **A2**, this only needs to be done up until time $t_{\text{rc}}$. Thus, we discretize each path such that consecutive states along the path are no more than $\delta_t$ time apart. As we will see, this will allow the system to start executing a new path within $T_{\text{bound}} + \delta_t$ after a new pose estimation is obtained from $\mathcal{P}$. We call all states that are less than $t_{\text{rc}}$ time from $s_{\text{home}}$ *replanable states*.

In the second step of the preprocessing stage, for every replanable state along each path $\Pi_g$, we compute a new path to all goal states. See Fig. 3. This will ensure that all goal states are covered by the replanabale states that were introduced in the first step of the preprocessing stage. Namely, it will allow to immediately start executing a new path once the goal location is updated by $\mathcal{P}$. Unfortunately, $\mathcal{P}$ 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 that can be queried in $O(1) < T_{\text{bound}}$ time.

In the query stage we obtain an estimation $g_1$ of the goal pose by $\mathcal{P}$. The algorithm then retrieves the path $\Pi_1(s_{\text{home}}, g_1)$ from $s_{\text{home}}$ to $g_1$ and the robot starts executing $\Pi_1(s_{\text{home}}, g_1)$. For every new estimation $g_i$ of the goal pose obtained from $\mathcal{P}$ while the system is executing path $\Pi_{i-1}(s, g_{i-1})$, the algorithm retrieves from the lookup table the path $\Pi_i(s', g_i)$ from the first state $s'$ along $\Pi_{i-1}(s, g_{i-1})$ that is $T_{\text{bound}} + \delta_t$ time from $s$. The robot $\mathcal{R}$ will then start executing $\Pi_i(s', g_i)$ once it reaches $s'$.

Clearly, every state is covered by this brute-force approach, however it requires a massive amount of memory. Let $n_{\text{goal}} = |G^{\text{full}}|$ be the number of goal states 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 $O(n_{\text{goal}}^{\ell})$ paths which is clearly infeasible. In the next sections, we show how we can dramatically reduce the memory footprint of the approach without compromising on the system's capabilities.

## B. Algorithmic approach

While the straw man algorithm presented allows for planning to any goal pose $g \in G^{\text{full}}$ in bounded time $T_{\text{bound}}$, 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 (see Fig. 3).

Similar to the straw man algorithm, our preprocessing stage runs in two steps. In the first step we compute from $s_{\text{home}}$ a path $\Pi_g$ to every goal state $g \in G^{\text{full}}$. However, this is done by computing a set of so-called "root paths" $\{\Pi_1, \ldots, \Pi_k\}$ from $s_{\text{home}}$ to a subset of goals in $G^{\text{full}}$ (here we will have that $k \ll n_{\text{goal}}$). As we will see, these paths will allow to efficiently compute paths to every goal state in the query stage and the system only needs to explicitly store these root paths in memory (and not a path to every goal state as in the straw man algorithm). In the second step of the preprocessing stage, the algorithm recursively computes for all replanabale states along all root paths a new path to each goal state. However, this is done by attempting to re-use previously-computed root paths which, in turn, allows for a very low memory footprint. It is important to note that replanable states are not only the ones that lie on the root paths from $s_{\text{home}}$—whenever the recursion happens we compute a new set of root paths and generate new replanable states The remainder of this section formalizes these ideas.

## C. Algorithmic building blocks

We start by introducing the algorithmic building blocks that we use. Specifically, we start by describing the motion planner 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 near-by goals.

*1) Motion planner:* We use a heuristic search-based planning approach with motion primitives (see, e.g, [4, 5, 19]) 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 six DoF grasp pose for the goal object while our robot has seven DoFs and we need to acount for the grasping time.

**State space and graph construction.** We define a state $s$ as a pair $(q, t)$ where $q = (\theta_1, ..., \theta_n)$ 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 $\mathcal{R}$ can execute. The first, which we term *predefined* primitives are used to move to a pre-grasp position while the second, termed *dynamic* primitives are used to perform grasping after the robot reached a pre-grasp position.

The predefined primitives are small individual joint movements in either direction as well as wait actions. For each motion primitive, we compute its duration by using a nominal constant velocity profile for the joint that is moved.

The dynamic primitives are generated by using a Jacobian pseudo inverse-based control law similar to what [20] uses. The velocity of the end effector is computed such that 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. Additional details omitted due to lack of space.

**Motion planner.** The states and the transitions implicitly define a graph $\mathcal{G} = (S, E)$ where $S$ is the set of all states and $E$ is the set of all transitions defined by the motion primitives. We use Weighted $\mathsf{A}^*$ ($\mathsf{wA}^*$) [22] to find a path in $\mathcal{G}$ from a given state $s$ to a goal state $g$. $\mathsf{wA}^*$ is a suboptimal heursitic search algorithm that allows a tradeoff between optimality and greediness by inflating the heuristic function by a given weight $w$. 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. More formally, our heuristic function is given by

$$h(s, g) = \max(w \cdot t(s, g), \textsc{AngleDiff}(s, g)).$$

Here, $t(s, g)$ is the expected time to intercept the object which can be analytically computed from the velocities and positions

**Algorithm 1** Plan Root Paths

```
 1: procedure PLANROOTPATHS($s_{\text{start}}, G^{\text{uncov}}$)
 2:   $\Psi_{s_{\text{start}}} \leftarrow \emptyset$                     ▷ a list of pairs $(\Pi_i, G_i)$
 3:   $G^{\text{uncov}}_{s_{\text{start}}} \leftarrow \emptyset$;   $i = 0$
 4:   while $G^{\text{uncov}} \neq \emptyset$ do      ▷ until all reachable goals are covered
 5:     $g_i \leftarrow$SAMPLEGOAL($G^{\text{uncov}}$)
 6:     $G^{\text{uncov}} \leftarrow G^{\text{uncov}} \setminus \{g_i\}$
 7:     if $\Pi_i \leftarrow$ PLANROOTPATH($s_{\text{start}}, g_i$) then ▷ planner succeeded
 8:       $G_i \leftarrow \{g_i\}$                          ▷ goals reachable
 9:       for each $g_j \in G^{\text{uncov}}$ do
10:         if $\pi_j \leftarrow$PLANPATHWITHEXPERIENCE($s_{\text{start}}, g_j, \Pi_i$) then
11:           $G_i \leftarrow G_i \cup \{g_j\}$
12:           $G^{\text{uncov}} \leftarrow G^{\text{uncov}} \setminus \{g_j\}$
13:       $\Psi_{s_{\text{start}}} \leftarrow \Psi_{s_{\text{start}}} \cup \{(\Pi_i, G_i)\}$;   $i \leftarrow i + 1$
14:     else
15:       $G^{\text{uncov}}_{s_{\text{start}}} \leftarrow G^{\text{uncov}}_{s_{\text{start}}} \cup \{g_i\}$             ▷ goals unreachable
16:   return $\Psi_{s_{\text{start}}}, G^{\text{uncov}}_{s_{\text{start}}}$
```

of the target object and the end-effector and ANGLEDIFF$(s, g)$ gives the magnitude of angular difference between the end-effector's current pose and target pose.

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

$$s_{\text{sc}}(\Pi, g) := \arg\min_{s_i \in \Pi} h(s_i, g).$$

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

## *D. Algorithmic details*

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

*1) Preprocessing:* Our preprocessing stage starts by sampling a goal state $g_1 \in G^{\text{full}}$ and computing a root path $\Pi_1$ from $s_{\text{home}}$ to $g_1$. We then associate with $\Pi_1$ all goal states $G_1 \subset G^{\text{full}}$ such that $\Pi_1$ can be used as an experience in reaching any $g_j \in G_1$ within $T_{\text{bound}}$. Thus, all goal states in $G_1$ are covered by $s_{\text{home}}$. We then repeat this process but instead of sampling a goal state 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) $\bigcup_i G_i = \text{REACHABLE}(s_{\text{home}}, G^{\text{full}})$ (i.e all reachable goals for $s_{\text{home}}$ in $G^{\text{full}}$). Alg. 1 details this step (if called for arguments ($s_{\text{home}}$,$G^{\text{full}}$)) and is illustrated in Fig. 4. Alg. 1 also returns a set of unreachable goals that are left uncovered.

**Algorithm 2** Preprocess

```
 1: procedure TRYLATCHING($s, \Psi_{s_{\text{home}}} G^{\text{uncov}}, G^{\text{cov}}$)
 2:   for each $(\Pi_i, G_i) \in \Psi_{s_{\text{home}}}$ 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:   $\Psi_{s_{\text{start}}}, G^{\text{uncov}}_{s_{\text{start}}} \leftarrow$ PLANROOTPATHS($s_{\text{start}}, G^{\text{uncov}}$)
 9:   if $s_{\text{start}} = s_{\text{home}}$ then $\Psi_{s_{\text{home}}} = \Psi_{s_{\text{start}}}$
10:   $G^{\text{cov}}_{s_{\text{start}}} \leftarrow G^{\text{cov}} \cup (G^{\text{uncov}} \setminus G^{\text{uncov}}_{s_{\text{start}}})$
11:   if $t(s_{\text{start}}) \leq t_{\text{rc}}$ then
12:     for each $(\Pi_i, G_i) \in \Psi_{s_{\text{start}}}$ do
13:       $G^{\text{cov}}_i \leftarrow G_i$;   $G^{\text{uncov}}_i \leftarrow G^{\text{cov}}_{s_{\text{start}}} \setminus G_i$;
14:       for each $s \in \Pi_i$ (from last to first) do      ▷ states up to $t_{\text{rc}}$
15:         $G^{\text{uncov}}_i, G^{\text{cov}}_i \leftarrow$ TRYLATCHING($s, \Psi_{s_{\text{home}}}, G^{\text{uncov}}_i, G^{\text{cov}}_i$)
16:         if $G^{\text{uncov}}_i = \emptyset$ then
17:           break
18:         $G^{\text{uncov}}_i, G^{\text{cov}}_i \leftarrow$ PREPROCESS($s, G^{\text{uncov}}_i, G^{\text{cov}}_i$)
19:         if $G^{\text{uncov}}_i = \emptyset$ then
20:           break
21:   return $G^{\text{uncov}}_{s_{\text{start}}}, G^{\text{cov}}_{s_{\text{start}}}$
```

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 $t_{\text{rc}}$. 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 algorithm is initially called for 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 constitute an additional optimization step which we call "latching" and is explained later in Sec. IV-D3.

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

**O1** If a goal is not reachable from a state $s \in \Pi$, it is not reachable from all the states proceeding 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 $\Pi$

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 its proceeding states on that root path.

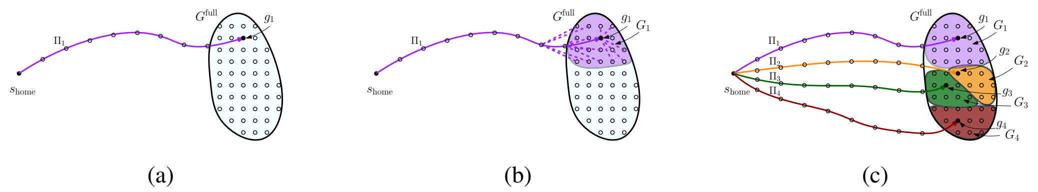


Fig. 4: First step of the preprocessing stage. (a) A goal state $g_1$ is sampled and the root path $\Pi_1$ is computed between $s_{\text{home}}$ and $g_1$. (b) The set $G_1 \subset G^{\text{full}}$ of all states that can use $\Pi_1$ as an experience is computed and associated with $\Pi_1$. (c) The goal region covered by four root paths from $s_{\text{home}}$ after the first step of the preprocessing stage terminates.

Specifically, using **O2**, a single set $G_i^{\text{uncov}}$ is initialized for all states on $\Pi_i$ and the goals that each $s \in \Pi_i$ covers are removed from it in every iteration. Using **O1**, $G_i^{\text{uncov}}$ is initialized only with the goals covered by $s_{\text{start}}$ (line 13). $G_i$ is excluded since it is already covered via $\Pi_i$. The iteration completes either when all goals in $G_i^{\text{uncov}}$ are covered (line 19) or the loop backtracks to $s_{\text{start}}$.

Thus, as the outcome of the preprocessing stage a map $\mathcal{M} : S \times G^{\text{full}} \rightarrow \{\Pi_1, \Pi_2, \dots\}$ is constructed mapping which root path can be as an experience to plan to a goal $g$ from a state $s$ within $T_{\text{bound}}$.

*2) Query:* Alg. 3 describes the query phase of our algorithm. Again, the code in blue correspond to the blue code in Alg. 2 for the additional optimization step which is explained in Sec. IV-D3. Assume that the robot was at a state $s_{\text{curr}}$ while executing a path $\pi_{\text{curr}}$ when it receives a pose update $g$ from the perception system. Alg. 3 will be called for a state $s_{\text{start}}$ that is $T_{\text{bound}}$ ahead of $s_{\text{curr}}$ along $\pi_{\text{curr}}$, allowing the algorithm to return a plan before the robot reaches $s_{\text{start}}$.

Alg. 2 assures that there exists one state on $\pi_{\text{curr}}$ between $s_{\text{start}}$ and the state at $t_{\text{rc}}$ that covers $g$. Therefore, we iterate through each $s \in \pi_{\text{curr}}$ backwards (similar to Alg. 2) between $s_{\text{start}}$ and the state at $t_{\text{rc}}$ and find the one that covers $g$ by quering $\mathcal{M}$. Once found, we use the corresponding root path $\Pi_{\text{next}}$ as an experience to find the path $\pi_{\text{next}}$ from $s$ to $g$. Finally the paths $\pi_{\text{curr}}$ and $\pi_{\text{next}}$ are merged together with $s$ being the transitioning state to return the final path $\pi$.

*3) Latching: Reusing Root Paths:* We introduce an additional step called "Latching" to minimise the number of root paths computed in the preprocessing phase **??**. With latching, the algorithm tries to reuse previously computed root paths as much as possible.

## *E. Theoretical guarantees*

**Lemma 1** (Completeness). *Let $s$ be a state that $\mathcal{R}$ is at in the query phase and $g \in G^{full}$ is a goal state. If $g$ is reachable from a state $s'$ that is (i) reachable from $s$ and (ii) is $T_{bound}$ away from $s$ then the planner will compute a path to $g$.*

We omit the proof due to lack of space.

**Lemma 2** (Constant-time planning). *Let $s$ be a replanable state and $g$ a goal state provided by $\mathcal{P}$. If $g$ is reachable, the planner is guaranteed to provide a solution in constant time.*

**Algorithm 3** Query

```
Inputs: $\mathcal{M}$, $s_{\text{home}}$
procedure PlanPathByLatching($s_{\text{start}}, g$)
  if $\Pi_{\text{home}} \leftarrow \mathcal{M}(s_{\text{home}}, g)$ exists then        ▷ lookup root path
    if CanLatch($s, \Pi_{\text{home}}$) then
      $\pi_{\text{home}} \leftarrow$PlanPathWithExperience($s_{\text{start}}, g, \Pi_{\text{home}}$)
      $\pi \leftarrow$ MergePathsByLatching($\pi_{\text{curr}}, \pi_{\text{home}}, s$)
      return $\pi$
  return failure

procedure Query($g, \pi_{\text{curr}}, s_{\text{start}}$)
  for each $s \in \pi_{\text{curr}}$ (from last to $s_{\text{start}}$) do        ▷ states up to $t_{\text{rc}}$
    if $\Pi_{\text{next}} \leftarrow \mathcal{M}(s, g)$ exists then        ▷ lookup root path
      $\pi_{\text{next}} \leftarrow$PlanPathWithExperience($s_{\text{start}}, g, \Pi_{\text{next}}$)
      $\pi \leftarrow$ MergePaths($\pi_{\text{curr}}, \pi_{\text{next}}, s$)
      return $\pi$
    if $\pi \leftarrow$PlanPathByLatching($s_{\text{start}}, g$) successful then
      return $\pi$
  return failure        ▷ goal is not reachable
```

*Proof:* We have to show that the query stage (Alg. 3) has a constant-time complexity. The number of times the algorithm queries for a root path (namely, computing $\mathcal{M}()$ which is a hash look-up that is a constant-time operation) is bounded by $n_{\text{steps}} = t_{\text{rc}}/\delta_t$ which is the maximum number of time steps from $t = 0$ to $t_{\text{rc}}$. The number of times the algorithm will attempt to latch on to a root path (namely, a call to CanLatch which is a constant-time operation) is also bounded by $n_{\text{steps}}$. Finally, Alg. 3 calls the Plan method only once. Since we are using a deterministic planner, the computation time is constant. Hence the overall complexity of Alg. 3 is $O(1)$. ■

The analysis provided in Lemma 2 highlights the tradeoff our algorithm takes. If we use a fine discretization of states along the path (namely, we choose a small value for $\delta_t$) then the guaranteed planning time $T_{\text{bound}}$ is going to be high but there is a higher chance that any goal $g$ will be reachable. On the other hand, if we use a course discretization (namely, we choose a large value for $\delta_t$) then we reduce $T_{\text{bound}}$ at the price that some goal states may not be reachable.

## V. Evaluation

We evaluated our algorithm in simulation and on a real robot. The conveyor speed that we used for all of our results is $0.2 m/s$. We used Willow Garage's PR2 robot in our experiments using its 7-DOF arm. The additional time dimension makes the planning problem eight dimensional.

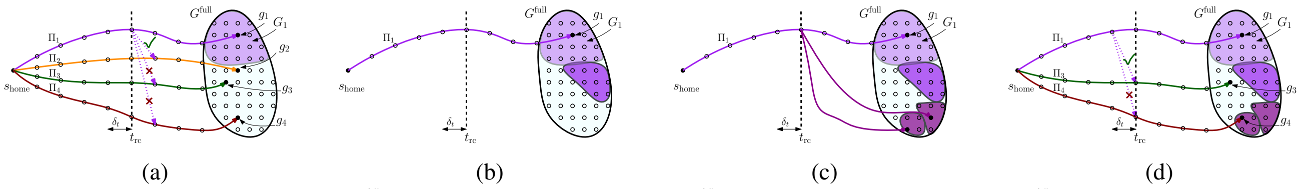
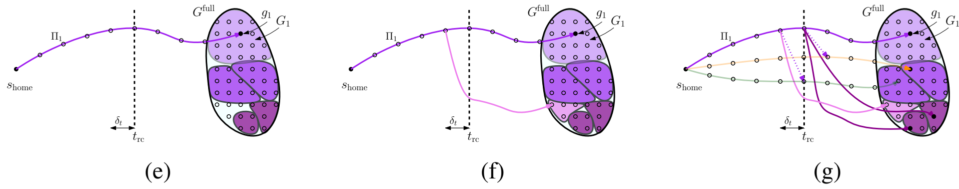



Fig. 5: Preprocess loop. (a) We start by trying to latch on to every other root path. (b) For successful latches, we add the corresponding goal states to our covered region. (c) We then try to compute new root path with its corresponding goal states (see also Fig. 4b). (d)-(f) This process is repeated by backtracking along the root path. (g) Outcome of a preprocessing step for one path. $G^{\text{full}}$ is covered either by using $\Pi_1$ as an experience, latching on to $\Pi_2$ or $\Pi_3$ (at different time steps) or by using newly-computed root paths.

| | **Our Method** | wA* | | | E-Graph | | | RRT | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | $T_b$ = 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 | $T_b$ = 0.5 | $T_b$ = 1.0 | $T_b$ = 2.0 |
| Pickup success rate [%] | 92.0 | 0.0 | 0.0 | 18.0 | 0.0 | 0.0 | 80.0 | 0.0 | 0.0 | 18.0 |
| Planning success rate [%] | 94.7 | 4.0 | 17.0 | 19.0 | 31.0 | 80.0 | 90.0 | 12.0 | 9.0 | 13.0 |
| Planning time [s] | 0.069 | 0.433 | 0.628 | 0.824 | 0.283 | 0.419 | 0.311 | 0.279 | 0.252 | 0.197 |
| Planning episodes per pickup | 3 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| Path cost [s] | 10.11 | 8.19 | 8.28 | 7.60 | 8.54 | 8.22 | 7.90 | 9.68 | 8.96 | 8.04 |

TABLE I: Simulation results. Here $T_b$ denotes the (possibly arbitrary) timebound 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.

## A. Experimental setup

*1) Perception system $\mathcal{P}$:* In order to pick a known object $o$ moving object along the conveyor $\mathcal{B}$, we need a method to estimate its 3-DoF pose at various locations across $\mathcal{B}$. Thus, we created an ICP-based pose estimation strategy that uses the known 3D model of $o$ and the initial pose estimate obtained after pre-processing the input point cloud. The following process is performed at every frame obtained by $\mathcal{P}$. We start by removing all points that lie outside $\mathcal{B}$ which yields only points corresponding to the object of interest. This is followed by statistical outlier removal in order to remove any points that have been left in scene after the first step. In the final step, we compute the mean of the points left and use that as an initial translation estimate for ICP. The rotation estimate is applied in a way that it places the object parallel to its direction of motion along the conveyor. The object's 3D model is transformed with this initial estimate, creating a point cloud and ICP refinement is performed between this point and the pre-processed observed point cloud. The resulting transform is concatenated with the initial estimate to return the final predicted pose. For speeding up ICP, we downsample all point clouds to a leaf size of 15mm.

*2) Sense-plan-act cycle:* We follow the classical sense-plan-act cycle as depicted in Fig. 6. Specifically, $\mathcal{P}$ captures an image (point cloud) of the object $o$ at time $t_{\text{img}}$ followed by a period of duration $T_{\text{perception}}$ in which the $\mathcal{P}$ estimates the pose of $o$. At time $t_{\text{msg}} = t_{\text{img}} + T_{\text{perception}}$, planning starts for a period of $T_{\text{planning}}$ which is guaranteed to be less than $T_{\text{bound}}$.

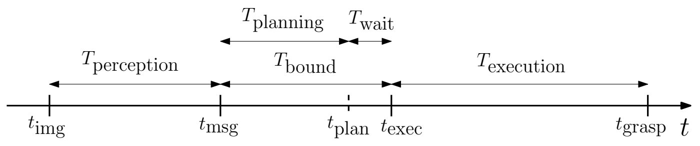


Fig. 6: Timeline of the sense-plan-act cycle.

Thus, at $t_{\text{plan}} = t_{\text{msg}} + T_{\text{planning}}$ 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 $t_{\text{img}}$ 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.

*3) Goal region specification:* To define the set of all goal poses $G^{\text{full}}$, we need to detail our system setup, depicted in Fig. 7. The conveyor belt $\mathcal{B}$ 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 $\mathcal{B}$. $G^{\text{full}}$ contains a fine discretization of all possible $y$ and $\theta$ values and $x$ values in $[x_{\text{exec}} - \varepsilon_{\mathcal{P}}, x_{\text{exec}} + \varepsilon_{\mathcal{P}}]$. We select $G^{\text{full}}$ such that $\varepsilon_{\mathcal{P}} = 0.5$ cm, dimension along $y$-axis is 20 cm. The discretization in $x, y$ and $\theta$ is 1.0 cm and 10 degrees respectively.

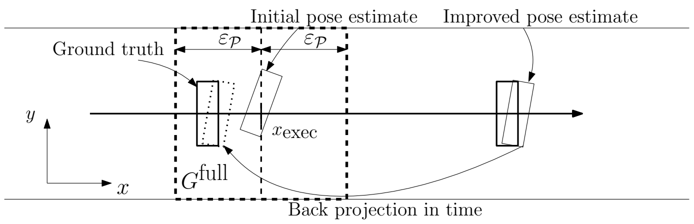


Fig. 7: A depiction of $G^{\text{full}}$-specification on a conveyor belt (overhead view) and perception noise handling.

| | Overall Pickup | Perception | Pickup (Perception = 1*) | Pickup (Perception = 0*) |
|---|---|---|---|---|
| E1 | **69.23** | **42.31** | **83.33** | **57.14** |
| E2 | 16.00 | 24.00 | 66.67 | 0.00 |
| E3 | 34.61 | 34.62 | 55.56 | 23.53 |

TABLE II: Real-robot experiments. Success rate for the three experiments (E1—our method, E2—First-pose planning and E3—Best-pose planning). Perception = 1 is accurate estimate, Perception = 0 is inaccurate

In the example depicted in Fig. 7, 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 the assumption **A3** the back projected pose will always lie inside $G^{\text{full}}$.

## B. Results

Before we report on our system-level results comparing our work with alternative implementation (namely, results from the query stage), we mention that the preprocessing stage took roughly 3.5 hours and the memory footprint following this stage is less than 20 MB. This backs up our intuition that the domain allows for efficient compression of a massive amount of paths in a reasonable amount of preprocessing time. In all experiments, we used $t_{\text{rc}} = 3.5$ seconds

*1) Real-robot experiments:* To show the necessity of real-time replanning we performed three types of experiments, (E1) replanning with multiple pose estimates, (E2) first-pose planning from the first object pose estimate (E3) best-pose planning from 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 $\mathcal{B}$. In addition, we report on $\mathcal{P}$'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 (resp. low) overlap results in an accurate (resp. inaccurate) pose estimate. We use the same strategy to determine a range for which $\mathcal{P}$'s estimates are accurate and use it for best-pose planning. Further, for each method, we determine the pickup success rate given that the perception was or wasn't accurate.

The experimental results are shown in Table II. Our method achieves the highest overall pickup success rate on the robot, 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, due to the limited time remaining to grasp object when it's closer to the robot.

*2) Simulation experiments:* We simulated the real world scenario to evaluate our method against other baselines. We compared our method with wA*, E-GRAPHS and RRT. For wA* and E-GRAPHS we use the same graph representation as our method. For E-GRAPHS 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 (see Fig. 6). If the planner fails to generate the plan within this time, then the robot misses the object for that cycle and such cases are recorded as failures. 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 V-B1. The key takeaway from our experiments (Table I) 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. The planning success rate being lower than 100% can be attributed to the fact that some goals were unreachable during the runs. We tested the other methods with several different time bounds. Besides our approach E-GRAPHS perform significantly 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.

## References

[1] Peter K Allen, Aleksandar Timcenko, Billibon Yoshimi, and Paul Michelman. Automated tracking and grasping of a moving object with a robotic hand-eye system. *TRA*, 9(2):152–165, 1993.

[2] Dmitry Berenson, Pieter Abbeel, and Ken Goldberg. A robot path planning framework that learns from experience. In *ICRA*, pages 3671–3678, 2012.

[3] Paul J Besl and Neil D McKay. Method for registration of 3-D shapes. In *Sensor fusion IV: control paradigms and data structures*, volume 1611, pages 586–606, 1992.

[4] Benjamin J. Cohen, Sachin Chitta, and Maxim Likhachev. Search-based planning for manipulation with motion primitives. In *ICRA*, pages 2902–2908, 2010.

[5] Benjamin J. Cohen, Gokul Subramania, Sachin Chitta, and Maxim Likhachev. Planning for Manipulation with Adaptive Motion Primitives. In *ICRA*, pages 5478–5485, 2011.

[6] David Coleman, Ioan A Şucan, Mark Moll, Kei Okada, and Nikolaus Correll. Experience-based planning with sparse roadmap spanners. In *ICRA*, pages 900–905, 2015.

[7] Anthony Cowley, Benjamin Cohen, William Marshall, Camillo J Taylor, and Maxim Likhachev. Perception and motion planning for pick-and-place of dynamic objects. In *IROS*, pages 816–823, 2013.

[8] Andrew Dobson and Kostas E. Bekris. Sparse roadmap spanners for asymptotically near-optimal motion planning. *IJRR*, 33(1):18–47, 2014.

[9] Shuai D Han, Si Wei Feng, and Jingjin Yu. Toward Fast and Optimal Robotic Pick-and-Place on a Moving Conveyor. *RAL*, 2019.

[10] Toru Ishida and Richard E Korf. Moving Target Search. In *IJCAI*, volume 91, pages 204–210, 1991.

[11] Toru Ishida and Richard E. Korf. Moving-target search: A real-time search for changing goals. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 17 (6):609–619, 1995.

[12] Fahad Islam, Anirudh Vemula, Sung-Kyun Kim, Andrew Dornbush, Oren Salzman, and Maxim Likhachev. Planning, Learning and Reasoning Framework for Robot Truck Unloading. *arXiv:1910.09453*, 2019.

[13] Lucas Janson, Brian Ichter, and Marco Pavone. Deterministic sampling-based motion planning: Optimality, complexity, and performance. *IJRR*, 37(1):46–61.

[14] Lydia E Kavraki, Petr Svestka, J-C Latombe, and Mark H Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *RAL*, 12(4):566–580, 1996.

[15] Sven Koenig and Maxim Likhachev. Real-time adaptive A*. In *AAMAS*, pages 281–288, 2006.

[16] Sven Koenig and Xiaoxun Sun. Comparing real-time and incremental heuristic search for real-time situated agents. *JAAMAS*, 18(3):313–341, 2009.

[17] Sven Koenig, Maxim Likhachev, and Xiaoxun Sun. Speeding up moving-target search. In *AAMAS*, pages 1–8, 2007.

[18] Richard E Korf. Real-time heuristic search. *Artificial intelligence*, 42(2-3):189–211, 1990.

[19] Maxim Likhachev and Dave Ferguson. Planning Long Dynamically Feasible Maneuvers for Autonomous Vehicles. *IJRR*, 28(8):933–945, 2009.

[20] Arjun Menon, Benjamin Cohen, and Maxim Likhachev. Motion planning for smooth pickup of moving objects. In *ICRA*, pages 453–460, 2014.

[21] Michael Phillips, Benjamin J. Cohen, Sachin Chitta, and Maxim Likhachev. E-Graphs: Bootstrapping Planning with Experience Graphs. In *RSS*, 2012.

[22] Ira Pohl. . *Artificial intelligence*, 1(3-4):193–204, 1970.

[23] Oren Salzman, Doron Shaharabani, Pankaj K. Agarwal, and Dan Halperin. Sparsification of motion-planning roadmaps by edge contraction. *IJRR*, 33(14):1711–1725, 2014.

[24] Denis Stogl, Daniel Zumkeller, Stefan Escaida Navarro, Alexander Heilig, and Björn Hein. Tracking, reconstruction and grasping of unknown rotationally symmetrical objects from a conveyor belt. In *EFTA*, pages 1–8. IEEE, 2017.

[25] Xiaoxun Sun, William Yeoh, and Sven Koenig. Moving target D* lite. In *AAMAS*, pages 67–74, 2010.

[26] Yizhe Zhang, Lianjun Li, Michael Ripperger, Jorge Nicho, Malathi Veeraraghavan, and Andrea Fumagalli. Gilbreth: A conveyor-belt based pick-and-sort industrial robotics application. In *IRC*, pages 17–24, 2018.