2019 International Conference on Robotics and Automation (ICRA)
Palais des congres de Montreal, Montreal, Canada, May 20-24, 2019

Improving Incremental Planning Performance through Overlapping Replanning and Execution

Matthew Orton, Siyu Dai, Shawn Schaffert, Andreas Hofmann, and Brian Williams

Abstract— **Deployment of motion planning algorithms in practical applications has lagged due to their slow speed in reacting to disturbances. We believe that the best way to address this is to reuse learned planning and control information across queries. In previous work, we introduced Chekov, a reactive, integrated motion planning and execution system that reuses learned information in the form of an enhanced roadmap. We have previously shown how we can use Chekov to formulate trajectory optimization problems that result in superior performance in static environments. In this work, we show how incremental planning can be incorporated into the formulation of optimized trajectories from roadmap seed trajectories. Further, we show how an incremental planner can be adapted to reduce the overhead incurred for replanning when trajectories become invalid during execution.**

I. Introduction

Deployment of motion planning in commercial automation has lagged due to a number of key deficiencies. For a motion planner to be practical for such deployments, it must be able to react to changing scenarios. Further, the generated trajectories should be intuitively pleasing, and should not alarm human agents that may be collaborating with the robot. It also implies that they should be near optimal and repeatable; if the motion planner is repeatedly given the same formulation, the trajectories it produces should be the same.

Existing motion planners can solve problems with complex obstacle configurations with guarantees on optimality, but many of these planners are too slow in the less complex environment configurations typical of practical deployments. Additionally, changing environments create uncertainty about the current and future states of obstacles, but most current motion planners assume static, known environments. Further, many of the current planners lack the capability to reuse learned information for future queries. Such information might be computed offline, or may be learned through previous queries. This deficiency means that persistent control policy information cannot be used across queries. While incremental planners do reuse information from one planning problem to the next, many still pause for replanning when any part of the current plan is invalidated.

Robotic systems deployed in the real world have to contend with a variety of challenges. The robot's sensors may provide noisy data, and the actuators may not provide completely precise movements. For example, in a mobile robot, the wheels often slip against the ground. Humans in the environment may move quickly and in unpredictable ways. Robots cannot spend an unbounded amount of time searching for an optimal motion plan; they must find solutions quickly in order to react effectively to new information.

The problem of moving a robot in uncertain environments is challenging. Often, there is significant complexity with path planning alone, due to the robot and environment geometry. Adding dynamic obstacles, noisy environment estimates, dynamics and actuation limits, and temporal constraints makes the problem even more challenging. Current motion planning and execution systems do not adequately address all of these challenges simultaneously: they assume the environment is static, or at least, predictable; they do not support task-level plans with temporal constraints; and many do not simultaneously support collision avoidance and complex dynamics.

We have previously developed *Chekov*, a reactive motion execution system that addresses these requirements [1]. Chekov avoids obstacles, incorporates dynamic models and control policies, and observes temporal constraints. We have also have shown the benefits of combining a sparse roadmap approach [2] with recent advances in obstacle-aware trajectory optimization [3], [4]. The resulting optimized trajectories are superior to the trajectories produced by the roadmap alone [5].

In this work, we address the limitations in the previous Chekov with regard to changing environments. This framework features a pre-computed roadmap and cache of shortest path solutions. However, these solutions are with respect to static environment obstacles, and must be validated to ensure that they do not collide with dynamic obstacles. We address this by incorporating incremental search techniques into the roadmap planner to reduce the computational cost. Additionally, we show how the effective time required for replanning can be reduced by parallel execution and planning.

Incremental search has been an active area of research for many years. With the D* Lite algorithm [6], a robot moves to a neighboring state that minimizes its cost to goal and updates any edges within a scan radius whose costs have changed due to changes in the environment. Successors to D* Lite include Adaptive A*(AA*) [7], Generalized Adaptive A* (GAA*) [7], and Multipath Adaptive A* (MPAA*) [8]. These algorithms all require updating all states that have been affected by changed edge costs within a visibility range along with establishing consistency in the heuristic values that have been affected. This can require a substantial computation and time if the visibility range is large, which is the reason these algorithms were not implemented for the research presented here. In fact, the the authors of MPAA* and MPGAA* address how MPGAA* performance suffers when presented with extended visibility ranges due to the number of heuristic updates required. While developments

978-1-5386-6027-0/19/$31.00 ©2019 IEEE

have been made as recently as 2017 to address this limitation [9], the problem scenarios in the research we are presenting use a full visibility range, which is problematic for MPGAA* and its improved variants. For this reason, we have opted for an approach involving repeated A* searches with lazy collision checking.

II. Problem Statement and Approach

The problem solved by Chekov is to plan and execute robot motions that accomplish a task specified by a set of temporal and spatial constraints. The resulting motions must be near-optimal with respect to a specified objective function. After plan execution has started, the system must react quickly to disturbances. This fast reaction is key to providing robots the capability to operate effectively in uncertain, fast-changing environments.

The inputs to Chekov are: 1) an environment containing obstacles; 2) a plant model representing the actuation limits of the robot; 3) the current state of the robot; 4) a set of spatial and temporal constraints that represent goals to be achieved; and 5) an objective function for optimization. The outputs of Chekov are control commands to the robot such that all constraints are observed, including achievement of goal regions and avoidance of obstacles, and such that its behavior is optimal according to the objective function (see also [1]). We make a number of key assumptions that we believe are consistent with a large class of practical robot manipulation problems. First, we assume that the manipulation workspace can be characterized by a limited set of pre-grasp poses. Second, we assume that the pre-grasp to grasp motion is short, and is best handled by visual and force servoing loops, rather than open-loop planners. Third, we assume that the collision environments are not overly complex. We are not trying to solve "piano mover" problems, rather we assume a small set of potential dynamic obstacles, such as a workpiece, another robot, or a human.

In previous work, we demonstrated key innovations that we endeavor to build upon here. First, we extended the roadmap approach used previously in Chekov by incorporating obstacle-aware trajectory optimization [3], [4] in order to improve optimality. Second, we developed three new environments that represent typical scenarios and made use of a fourth developed previously in the motion planning community to characterize planner performance in a set of "practically" relevant tests. In that work, we focused on static rather than dynamic obstacles because static obstacles occupy the majority of the workspace in many practical applications. The separation of static and dynamic obstacles is a key insight that enables superior performance.

In this work we address the problem of dynamic obstacles. We take an integrated approach that considers both planning and plan execution. As in our previous work, we begin by providing the motion plan query as an input to the roadmap planner. We then provide the output of this planner as input to obstacle-aware trajectory optimization, called TrajOpt [3], [4], resulting in an optimized, smooth plan. As part of this process, we maintain a map of correspondence points between the output of the roadmap planner and the output of TrajOpt. This allows us to divide the plan into segments, and maintain a correspondence between the segments in the roadmap plan, and the segments in the TrajOpt plan.

When the TrajOpt plan is output, plan execution begins. We continually monitor plan execution, checking for dynamic obstacles that may intersect future segments. In particular, before the next segment of a TrajOpt trajectory is executed, it is checked for collisions. If it is not collision-free, execution is halted and replanning occurs. Otherwise, the next segment is sent to the joint controller for execution and the remaining segments of the TrajOpt plan to be executed are checked for collisions to determine if replanning during execution should occur.

A key, innovative feature of our replanning approach is that it combines trajectory optimization with incremental search through use of the segment map. When a segment is identified to be in collision, the segment map is used to find the corresponding segment in the roadmap, allowing for fast incremental repair using the roadmap planner. The repaired roadmap plan is then input to TrajOpt for subsequent optimization, as before.

A second innovative feature of our approach is reduction of delays through replanning during the execution of upcoming segments of an invalid trajectory if the particular segments being executed are still valid. This parallel replanning and execution can dramatically reduce delays, and also give rise to a new metric, which we call *Effective Planning Time*. This metric provides a more appropriate indication of the actual temporal penalty imposed by the disturbance.

III. Implementation

From our previous work in [5], we have four representational environments: a tabletop with a pole (Fig. 1), a tabletop with a container, a kitchen and a shelf with boxes (Fig. 2) environment. The environments have varying degrees of difficulty that are reflected in the results from our previous work. Additionally, we generated 5000 planning tests for comparing different versions of our planner to one another as well as to other existing planners. Each of the 5000 cases consists of randomly sampled start and target end-effector pose pairs that are collision-free and kinematically feasible. These cases were developed for the Baxter robot [10] with its 7-DOF left arm as the manipulator. Based on our initial tests, TrajOpt works quite similarly on other manipulators, so here we take the Baxter left arm as our primary testbed.

A. Addressing Dynamic Obstacles

In order to address dynamic obstacles, new methods had to be developed for our roadmap-based planner to allow it to provide collision-free solutions to TrajOpt. One of the avenues pursued was to develop solution caches that contain multiple alternative solutions for each pair of roadmap nodes in addition to the shortest path. The alternative solutions were found by obstructing the robot workspace with obstacles representative in size, shape and pose to what would be expected in the given environment. While the results found

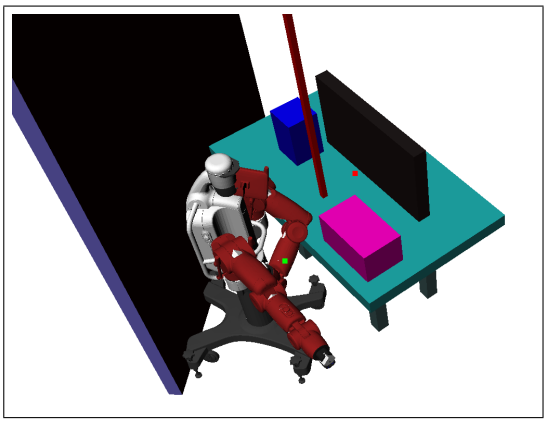

Fig. 1. The "tabletop with a pole" environment

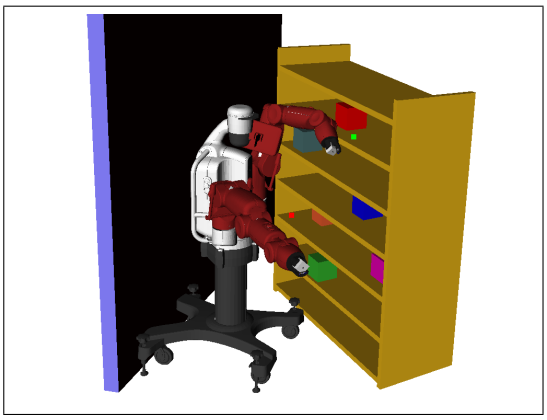

Fig. 2. The "shelf with boxes" environment

from these augmented solution caches did not strike us as noteworthy, the tests developed through extending the 5000 cases for each environment with sets of obstacles inserted into the workspace proved to be useful for the next avenue we pursued: online search.

The online strategies we developed for avoiding dynamic obstacles are based on an implementation of A* adapted for the roadmap-based planner (Fig. 3). When expanding search nodes, the algorithm performs collision checks if necessary, and then caches collision check results. This lazy collision checking approach limits collision checking to newly encountered edges. Since our test workspaces tend to be fairly open, the constructed roadmaps have many edges relative to the number of sampled nodes, and most solutions contain only a few (around 5) nodes. This allows A* search for a given roadmap to be fairly quick to return a solution relative to the time spent checking the solution for collisions. As a result, repeated searches are not costly since edges already encountered do not have to be collision checked. While the approach described here does not attempt to predict how the environment will change in the future, the capabilities of this approach would also be useful for systems that do make predictions about dynamic obstacles.

In developing our approach, we also considered D*Lite [6]. At a high level, D* Lite consists of the following steps:

1) Search for a plan from the goal to the current state of the robot
2) Move from the current state to the state that brings the robot closest to the goal and update the current state accordingly
3) Check if knowledge of obstacles has changed within a scan radius of the robot
   - If so, update all edges with changed costs and update the shortest path from the goal to the new current state
4) Repeat steps 2 and 3 until the goal has been reached

However, D* Lite is generally used to address scenarios involving mobile robots operating in environments with limited visibility. In contrast, motion planning for robot arms tends towards scenarios where the full state of the environment along a planned trajectory can be observed. We have found that D* Lite does not perform well when changes occur near the goal configuration for a planning problem because these changes often invalidate work done in previous search iterations. This concern along with some initial simulations involving an adaptation of D* Lite to fit our problem characteristics have led us to move away from this algorithm.

The approach we are presenting here incorporates incremental execution and execution monitoring with A* Repair and TrajOpt (Fig. 4). It starts out using TrajOpt to optimize a collision-free path returned from A* Repair. When the roadmap path is optimized, a mapping is created from the points in the optimized trajectory to the seed trajectory, so the optimized trajectory can be broken up into segments according to the roadmap edges.

At each execution iteration, the remaining trajectory to execute is checked segment by segment. If any part of the trajectory is in collision, replanning will occur on that iteration. However, if the trajectory segment that would be executed next is collision-free, that segment will be executed this iteration regardless of collisions found in any other segment. This leads to a key innovation of this paper: replanning can occur while this segment is being executed. In turn, replanning is performed between the goal and the endpoint of the segment executed during replanning. This effectively reduces the time it takes to replan by the time it takes to execute the collision-free segment.

To properly evaluate this innovation, we make use of a new metric in our analysis. This metric, *Effective Planning Time*, is the difference between the entire time taken for replanning and execution and the time taken just for execution. This provides a clear measure of how much time is saved through overlapping replanning and execution.

```
1: function GetCollisionFreePath(G, s_start, s_goal, validEdges)
2:     for all e ∈ G.Edges do
3:         validEdges.Add(e);
4:     path = GetShortestCachedPath(s_start, s_goal);
5:     success = True;
6:     while success do
7:         collisionFree, _ = CheckPathCollisions(
               path, validEdges);
8:         if collisionFree then return path;
9:         success, path = ComputeShortestPath(G, s_start,
               s_goal, validEdges);
10:    return Failure;

11: function ComputeShortestPath(G, s_start, s_goal, validEdges)
12:    evaluated = ∅;
13:    discovered = ∅;
14:    parentMap = ∅;
15:    for all s ∈ G.States do
16:        f(s) = g(s) = ∞;
17:    discovered.Add(s_start);
18:    g(s_start) = 0;
19:    f(s_start) = h(s_start, s_goal);
20:    while discovered is not empty do
21:        s_current = discovered.Pop();
22:        if s_current = s_goal then
23:            path = ReconstructPath(parentMap, s_current)
24:            return True, path;
25:        discovered.Remove(s_current);
26:        evaluated.Add(s_current);
27:        for neighbor ∈ G.Neighbors(s_current) do
28:            if neighbor ∈ evaluated then
29:                continue;
       ▷ Checking validEdges is the only deviation from standard A*
30:            if G.Edge(s_current, neighbor) ∉ validEdges then
31:                continue;
32:            if neighbor ∉ evaluated then
33:                discovered.Add(neighbor);
34:            score = g(s_current) + cost(s_current, neighbor);
35:            if score ≥ g(neighbor) then
36:                continue;
37:            parentMap(neighbor) = s_current;
38:            g(neighbor) = score;
39:            f(neighbor) = g(neighbor) + h(neighbor, s_goal);
40:    return False, []
```

Fig. 3. A* Repair and helper functions. GetCollisionFreePath is the main A* Repair method and ComputeShortestPath is an A* implementation adapted to incorporate knowledge of in-collision edges with h serving as the heuristic, g as the cost from start to current, and f as the cost from start to goal through current. ReconstructPath works as it would in a standard A* implementation. CheckPathCollisions checks the path for collisions, obtain pairs of nodes surrounding in-collision edges, and updates $validEdges$ accordingly.

IV. Simulations and Results

In this section, we highlight results from the roadmap framework used in [5] to establish a baseline of performance for Chekov in static environments with and without collision checking. Then we show how online search can be incorporated to improve performance in environment configurations that differ from the static case. Finally we adapt this approach to the incremental case and show how replanning overhead can be reduced for incremental algorithms through interleaving replanning with execution of previously planned trajectory components that are confirmed to still be valid at the time of execution. While one of the key innovations of our motion planner is the use of TrajOpt in conjunction with a roadmap-based planner, only the last simulation discussed in this section involves trajectory optimization. The preceding two simulations provide comparisons only within the roadmap-based planner.

```
1: function Main(G, s_goal, overlapExecution)
2:     for all e ∈ G.Edges do
3:         validEdges.Add(e);
4:     success, path = GetCollisionFreePath(G, s_current, s_goal,
           validEdges);
5:     if success = False then
6:         return False
7:     optimizedPath, pathMap = OptimizePath(path);
8:     pathIndex = 0;
9:     while s_current ≠ s_goal do
10:        nextSegment = optimizedPath[pathMap[pathIndex] :
               pathMap[pathIndex + 1]];
11:        excuteSegment = validPath = True;
12:        if nextSegment is in collision then
13:            excuteSegment = validPath = False;
14:        for i from pathIndex + 1 to Size(path) − 1 do
15:            segment = optimizedPath[pathMap[i] :
                   pathMap[i + 1]];
16:            if segment is in collision then
17:                validPath = False;
18:        if overlapExecution = False and (validPath = False) then
19:            excuteSegment = False;
20:        if excuteSegment = True then
21:            Move the robot along nextSegment and update s_current;
22:            pathIndex = pathIndex + 1;
       ▷ Only scan for changes if there is not a valid full path to the goal
23:        if validPath = False then
24:            success, path = GetCollisionFreePath(G,
                   s_current, s_goal, validEdges);
25:            if success = False then
26:                return False
27:            optimizedPath, pathMap = OptimizePath(path);
28:            pathIndex = 0;
```

Fig. 4. A* Repair with incremental execution and execution monitoring.

A. Roadmap Performance in All Environments

The first set of simulations evaluates roadmap performance in each of the four static environments. The results in Table I denoted Connections have no collision checks performed on roadmap edges and were previously shown in [5]. Collision checks are still performed on connections made from the start and end test points to existing roadmap nodes for all simulations. We present a comparison of these results to simulation results with collision checking for roadmap edges in order to better contextualize the remaining results in this paper.

Since the roadmaps are constructed to be collision-free in the static environment, all nodes and edges in the roadmap will be collision-free for these tests. This means that all failures in these simulations are caused by not being able to make a straight-line, collision-free connection from the start or end point for a case and an existing node in the roadmap. Additionally, all paths returned by the roadmap come from a precomputed solution cache, so the majority of the time for a roadmap query is consumed by checking roadmap edges for collisions in some cases and establishing the collision-free connections to the roadmap in all cases. The main takeaway is that checking roadmap edges for collisions results in a

TABLE I

ROADMAP PERFORMANCE IN ALL ENVIRONMENTS

Roadmap	Collision Checking[1]	Failure Rate	Average Runtime(s)	Average Path Length(rad)
Tabletop with a Pole				
500 Nodes	Connections	0.18%	0.1596	1.284
	All Edges	0.18%	1.033	1.284
1000 Nodes	Connections	0.18%	0.1434	1.238
	All Edges	0.18%	1.001	1.238
Tabletop with a Container				
500 Nodes	Connections	1.40%	0.2054	1.310
	All Edges	1.40%	1.170	1.310
1000 Nodes	Connections	0.76%	0.1806	1.320
	All Edges	0.76%	1.159	1.320
Kitchen				
500 Nodes	Connections	2.80%	0.4248	1.289
	All Edges	2.80%	2.441	1.289
1000 Nodes	Connections	1.92%	0.3792	1.285
	All Edges	1.92%	2.279	1.285
Shelf with Boxes				
500 Nodes	Connections	15.50%	0.4456	1.308
	All Edges	15.50%	1.569	1.308
1000 Nodes	Connections	12.06%	0.3876	1.302
	All Edges	12.06%	1.566	1.302

[1] Collision checks are either just performed on connections made to existing roadmap nodes or on all edges in a trajectory, roadmap edges and connections to roadmap nodes.

TABLE II

OBTAINING A COLLISION-FREE PATH IN MODIFIED ENVIRONMENTS

Environment	Path Retrieval[1]	Success Rate[2]	Average Runtime(s)
Tabletop with a Pole	APSP Cache	11.75%	0.5041
	A* Repair	44.92%	0.8118
Tabletop with a Container	APSP Cache	10.62%	0.4982
	A* Repair	35.65%	0.6903
Kitchen	APSP Cache	8.24%	0.9681
	A* Repair	33.93%	1.377
Shelf with Boxes	APSP Cache	6.32%	0.6046
	A* Repair	24.70%	0.7939

[1] Path retrieval refers to how a collision-free trajectory is obtained by the roadmap. In on case, failure occurs when the cached solution is in collision for the pair of roadmap nodes connected to. For the other case, the roadmap is searched with A* Repair if the original cached solution is in collision and failure occurs if no collision-free path exists in the roadmap for the pair of points connected to.

[2] Success rate here refers to the percentage of cases where a collision-free solution could be found out of cases where the solution used for the static environment has been forced into collision by an inserted obstacle.

substantial increase in average query time, so any planning approach where a static environment is not assumed should seek to minimize unnecessary collision checks.

B. Online Planning to Avoid Obstacles

The goal for the roadmap-based planner is to quickly produce collision-free seed trajectories for optimization. One of the key aspects of our approach to the roadmap-based planner is the utilization of a pre-computed set of paths that are collision-free in a static environment. While this may be sufficient for many cases, it should be expected that scenarios will arise due to dynamic obstacles where no collision-free solution exists in the cache that will satisfy the planning query. To address the dynamic obstacles, we use the following approach. First, we check cached solutions to see if any are collision free for the query. If no solution is collision-free, the collision checks that were performed are used (cached) to invalidate parts of the roadmap to speed up the subsequent search for a solution that is in fact collision-free (see A* Repair algorithm description above).

The simulations used to observe the benefits provided by A* Repair to the existing all-pairs shortest path (APSP) solution cache approach build off the original 5000 cases used to test planners in our four static environments. Each of the individual 5000 cases is run multiple times for a given simulation, but every time a case is run, an obstacle from a small set of obstacles is inserted into the environment at one of a finite set of poses associated with that obstacle. This served to obstruct parts of the environment in ways that appeared representative of what may be encountered in a similar environment in the real world. Obstacles were placed in a way to ensure that the robot still had some range of motion, but unlike with the original 5000 cases, no effort was made to guarantee that a case was still solvable by any planner once an obstacle was inserted.

The results in Table II shows how often a different cached solution can still be used when the shortest possible solution, including added length from establishing connections, is in collision. More importantly, the results provide insight into one of the trade-offs of using online search: more collision-free solutions will be found in changing environments at the expense of a longer average query time. The degree to which this trade-off is worth it would depend on the nature of the planning scenario, how often failures occur, and what the price of any given failure is.

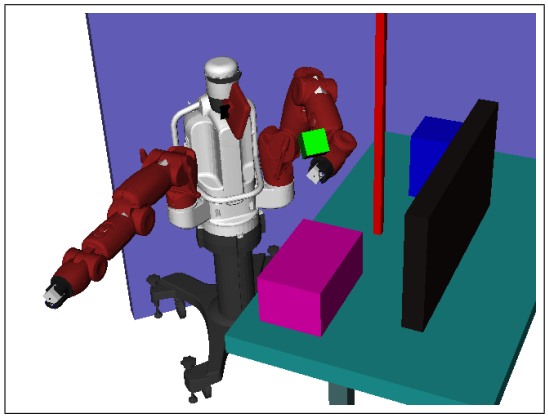

Fig. 5. A cube is inserted in the path of the end-effector to force replanning to occur.

C. Replanning During Execution

With A* Repair, we can see how online search can be incorporated into the existing roadmap-based planner and APSP solution cache. Through relying on a lazy collision checking approach and reusing information from previous checks, A* Repair minimizes collision checking, which is the

most time consuming step for our planner. However, in its original implementation, A* Repair requires full replanning when a path becomes invalid during execution. This led to an investigation into incremental planners and how they could incorporate trajectory optimization. While this did not result in the development of new incremental planning algorithms, it did yield an interesting innovation nonetheless. We have found that there are benefits to be had for any existing incremental planner through the interleaving of replanning and execution. When a trajectory is discovered to be invalid, it does not need to result in the immediate cessation of execution. If the next segment in the trajectory is still valid, it can still be executed and replanning can begin during this execution.

These simulations conducted to demonstrate the benefit of overlapping replanning with execution also start with the 5000 cases we generated for each of the four environments. For each case, the planner produces an initial optimized trajectory using the roadmap-based planner with TrajOpt. A 10cm cube is then inserted at the end-effector pose halfway along the optimized trajectory right before execution (see Fig. 5). This invalidates the original trajectory and forces replanning either before or during execution depending on whether or not overlapping is allowed.

From the results in Table III, we see that the executed path length increases when overlapping is allowed between replanning and execution. This should be expected because the trajectory segment that is executed during replanning is no longer part of an optimal plan for the environment in that state. The longer paths are also the primary cause for the lack of improvement in total time and execution time. However, what is most noteworthy about these results is the consistent improvement in effective planning time through interleaving replanning and execution. These results demonstrate a potential for incremental planners to take advantage of time delays that naturally come with trajectory execution on physical systems. They also showcase a metric to allow us to capture the benefits from utilizing what is generally lost time during execution. All of this aligns with our goal of developing a more reactive and intuitive motion planner through minimizing any delays where the robot is not in motion.

TABLE III

PERFORMANCE COMPARISON OF OVERLAPPING AND NON-OVERLAPPING (SERIAL) REPLANNING AND EXECUTION

Roadmap	Approach[1]	Executed Path Length(rad)	Total Time(s)[2]	Execution Time(s)	Effective Planning Time(s)[3]
Tabletop with a Pole					
500 Nodes	Overlapping	6.447	7.346	5.064	2.282
	Serial	6.061	7.161	4.781	2.380
1000 Nodes	Overlapping	6.065	7.165	4.794	2.371
	Serial	5.665	6.939	4.493	2.446
Tabletop with a Container					
500 Nodes	Overlapping	6.619	7.431	5.148	2.283
	Serial	6.351	7.386	4.953	2.433
1000 Nodes	Overlapping	6.457	7.435	5.075	2.36
	Serial	6.193	7.463	4.884	2.579
Kitchen					
500 Nodes	Overlapping	6.249	7.919	4.871	3.048
	Serial	5.970	7.845	4.682	3.163
1000 Nodes	Overlapping	6.099	8.001	4.780	3.221
	Serial	5.829	8.027	4.596	3.431
Shelf with Boxes					
500 Nodes	Overlapping	6.769	7.920	5.361	2.559
	Serial	6.582	8.037	5.229	2.808
1000 Nodes	Overlapping	6.514	7.891	5.177	2.714
	Serial	6.357	8.059	5.066	2.993

[1] Whether replanning and execution overlapped or were handled serially
[2] The elapsed time from when the cube is inserted in the environment to when execution of the trajectory has completed
[3] The difference between the Total Time metric and the Execution Time metric

V. DISCUSSION

Our results here build off the benefits we have previously demonstrated through combining a roadmap-based planner with obstacle-aware trajectory optimization. We have expanded the existing planning framework to account for dynamic obstacles. The approach we took is to implement online search that utilizes work performed checking precomputed shortest path solutions for collisions. From there we investigated how incremental search can be incorporated with the roadmap and trajectory optimization.

It was in this investigation that we realized the benefits that could be achieved through overlapping replanning and execution when using incremental planners. The goal is to complete as much of the replanning process as possible while executing trajectory segments that are still valid within a larger plan that is invalid. In order to capture this benefit, we had to define a new metric. This metric, effective planning time, measures the difference between total replanning and execution time and the time of just execution for a given planning query.

A trade-off exists in this approach with regards to the resolution of path segments within a full trajectory that was not explored in this paper. If path segments are longer, the execution of individual segments allow more time for replanning during execution, but longer segments allow obstacles to invalidate larger portions of a trajectory than may be necessary. Also, there may be cases when it is better to not execute the next segment of a trajectory due to it bringing the robot closer to a collision. However, if incremental planners can be tuned to complete the replanning process within the execution of these segments, then effective planning time can be reduced to zero and there will be no perceptible delay caused by environment changes that invalidate trajectories that had been collision-free. This helps move us towards our goal of having reactive and intuitive motion planning and execution systems for robots that could one day be commonplace in everyday lives.

REFERENCES

[1] A. Hofmann, E. Fernandez, J. Helbert, S. Smith, and B. Williams, "Reactive integrated motion planning and execution." AAAI Press/International Joint Conferences on Artificial Intelligence, 2015.

[2] L. E. Kavraki, P. Svestka, J.-C. Latombe, and M. H. Overmars, "Probabilistic roadmaps for path planning in high-dimensional configuration spaces," *IEEE transactions on Robotics and Automation*, vol. 12, no. 4, pp. 566–580, 1996.
[3] J. Schulman, J. Ho, A. X. Lee, I. Awwal, H. Bradlow, and P. Abbeel, "Finding locally optimal, collision-free trajectories with sequential convex optimization." in *Robotics: science and systems*, vol. 9, no. 1. Citeseer, 2013, pp. 1–10.
[4] J. Schulman, Y. Duan, J. Ho, A. Lee, I. Awwal, H. Bradlow, J. Pan, S. Patil, K. Goldberg, and P. Abbeel, "Motion planning with sequential convex optimization and convex collision checking," *The International Journal of Robotics Research*, vol. 33, no. 9, pp. 1251–1270, 2014.
[5] S. Dai, M. Orton, S. Schaffert, A. Hofmann, and B. Williams, "Improving trajectory optimization using a roadmap framework." International Conference on Intelligent Robots and Systems, 2018.
[6] S. Koenig and M. Likhachev, "D*lite," in *Eighteenth National Conference on Artificial Intelligence*. American Association for Artificial Intelligence, 2002, pp. 476–483.
[7] X. Sun, S. Koenig, and W. Yeoh, "Generalized adaptive a*," in *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems - Volume 1*. International Foundation for Autonomous Agents and Multiagent Systems, 2008, pp. 469–476.
[8] C. Hernandez, J. A. Baier, and R. J. A. Acha, "Making a* run faster than d*-lite for path-planning in partially known terrain," 2014.
[9] C. Hernandez, R. Asin, and J. Baier, "Improving mpgaa* for extended visibility ranges," 2017.
[10] RethinkRobotics, "Baxter," http://www.rethinkrobotics.com/baxter/.