Real-time Heuristic Search in Dynamic Environments

BY

Chao Chi Cheng

Senior Thesis

Submitted to the University of New Hampshire
in Partial Fulfillment of
the Requirements for the Degree of

Bachelor of Science

in

Computer Science

May, 2019

ABSTRACT

Real-time Heuristic Search in Dynamic Environments

by

Chao Chi Cheng

University of New Hampshire, May, 2019

In dynamic environments, agents often do not have time to find a complete plan to reach a goal state, but rather must act quickly under changing circumstances. Real-time heuristic search models this setting by requiring that the agent's next action be selected within a prespecified time bound. In this thesis, we study real-time search algorithms that can tolerate a fully dynamic environment, in which action costs are not fully predictable and can change to become either more or less expensive. We present a combination of two previously-proposed methods that fully utilize the execution time of an agent and adopt an advanced action selection method in a dynamic environment. We evaluate these methods in random maps and discretized city grid maps within a dynamic environment setting. The results suggest that one of the new algorithms is significantly superior to the originals.

CHAPTER 1

Introduction

AI applications for agents to find a feasible path from start to goal configuration have been heavily studied in the past. Off-line planning and on-line planning are two major approaches to the problem. Off-line planning requires agents to have a complete plan before they start executing. In a dynamic setting, agents in off-line planning need to know environmental changes in advanced to be able to create a collision-free trajectory. Therefore, for an agent trying to drive in an urban environment, off-line planning is often insufficient. Planning in a dynamic environment such as a city requires an agent to update its plan frequently to respond to the changes around it. The setting of real-time heuristic search models on-line planning by requiring the agent to commit only to its next action within a strict time limit. To prevent uncontrolled behavior, the time bound for planning is set to the time at which the actions to which the agent has already committed will end. In this way, planning and execution unfold in parallel. The real-time problem setting has been addressed by many previous algorithms, such as the paradigmatic LSS-LRTA* algorithm (Koenig & Sun, 2009), but most of these only tolerate dynamic environments in which changes result in state transitions that are more expensive than originally anticipated. One exception is PLRTA* (Cannon, Rose, & Ruml, 2014), which is similar to LSS-LRTA* but learns separately about static and dynamic aspects of the state space. As a result, PLRTA* has been shown to achieve much higher performance in navigation tasks in dynamic environments.

Traditional real-time heuristic search selects actions by choosing the best node on the lookahead frontier as its next action to execute. However, Kiesel, Burns, and Ruml (2015) noted that, because f values tend to rise along paths when h is admissible, the node with lowest f value on the frontier is often not the node closest to a goal. They addressed this

situation by learning online an inadmissible heuristic $\hat{h}$ and using it to derive $\hat{f}$, which is used for prioritizing frontier expansion and action selection. They also introduce another enhancement, dynamic lookahead, in which the agent commits to multiple actions and then take full advantage of that execution time to increase the time bound of the next planning iteration. They call the combination of these technique Dynamic-$\hat{f}$.

In this thesis, we propose two new algorithms, $\hat{f}$-PLRTA* and Dyn-$\hat{f}$-PLRTA*, that enhance PLRTA* with the $\hat{f}$ and dynamic lookahead techniques from Dynamic-$\hat{f}$. We discuss how these techniques can be combined and adapted in the real-time dynamic world setting. In an empirical evaluation, we find that Dyn-$\hat{f}$-PLRTA* outperforms the original algorithm in the majority of scenarios tested, particularly for difficult instances and small lookaheads. $\hat{f}$-PLRTA*, on the other hand, does not seem advantaged relative to the original PLRTA* algorithm. We hope this work spurs further investigation into algorithms for real-time fully dynamic environments.

1.1 Previous Work

Real-time search has been deeply studied in the past, and there are numerous real-time algorithms that target different problem settings. In this section, we discuss some of the previous approaches related to our algorithms.

Real-time A* (RTA*) was proposed by (Korf, 1990) and it is the very first real-time algorithm. It has become the foundation of real-time search since it first came out. RTA* does not require the agent to find a complete plan before executing; instead, it chooses the next action from the neighbor node with lowest f value by performing the minimin lookahead search until prespecified depth. Then it updates its heuristic value from the second best f value to prevent the agent from falling into the cycle. The algorithm will repeat the process until it reaches the goal. As RTA* admissibly updates states' heuristic values as it goes, it can always find a solution from the start to the goal if one exists. That is to say, RTA* has been proven a complete algorithm when the state space is finite and action cost is always positive. The drawbacks of RTA* are apparent. When the heuristic is

misleading, the algorithm may take many iterations try to escape from the local minimum as RTA* only updates the heuristic values one time per iteration. The second drawback for RTA* is that it uses the depth as lookahead limitation causeing the time to spend on each iteration to be difficault to estimate.

Local Search Space-Learning Real-Time A* (LSS-LRTA*) is one of the most popular algorithms in real time heuristic search(Koenig & Sun, 2009). It addresses some of the problems in RTA*. In each iteration, LSS-LRTA* performs a node-limited A* search to generate its local search space (LSS). Then it uses a Dijkstra-like propagation algorithm to back up the heuristic values of the states in the LSS from the frontier. The updated heuristic values are then used during the next iteration to prevent the agent from becoming trapped in a 'local minimum' of the heuristic function. Then the agent commits to the actions leading from its current location to the best node on the LSS frontier. Since LSS-LRTA* has the mechanism to prevent the agent from trapping in local minimum, it is complete if the state space is finite and the heuristic is consistent. Compared to RTA*, LSS-LRTA* updates the heuristic value of all states in LSS, which is much greater than the number that RTA* can update in one iteration. This gives the LSS-LRTA* the ability to escape from local minimum faster than the RTA*. Also LSS-LRTA* uses node expansion number to limit the search in each iteration. Compare to RTA*, it can provide a more precise timing when performing the time-bound search.

In an environment that contains dynamic obstacles, we assume that the agent has, at any given time, some current predictions of the future trajectories of the dynamic obstacles (Figure 1-1) that it can use to assess the cost of future actions in terms of the probability of a collision times a collision penalty (Kushleyev & Likhachev, 2009). Since these predictions can change over time, it can happen that the predicted cost of an action can be larger or smaller than it was in some previous planning iteration when that action was considered. Unlike offline heuristic search, online search in a dynamic environment inherently features these inadmissible g values.

In a dynamic environment, the state space usually includes time as a state variable, as

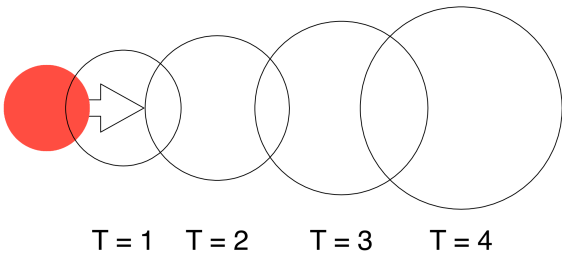

Figure 1-1: Prediction of Future Position of A Dynamic Obstacles

being in the same position at a different time could have a very different cost or outcome due to the dynamic obstacles. As the time variable is a priori unbounded, this implies that the state space is infinite. This means that standard real-time search h learning does not work effectively, as learned states will become unreachable and new states always appear. The algorithm Partitioned Learning Real-Time A* (PLRTA*) was proposed to cope with this problem (Cannon et al., 2014). Since most environments include static elements that do not change over time, PLRTA* distinguishes between costs arising from static vs dynamic elements and ignores the time when learning the static cost-to-go. Therefore, the evaluation function of PLRTA* consists of four components — static cost-so-far $g_s(n)$, static cost-to-go $h_s(n)$, dynamic cost-so-far $g_d(n)$, and dynamic cost-to-go $h_d(n)$ — which are combined to form the f value:

$$f(n) = g_d(n) + g_s(n) + h_d(n) + h_s(n) \tag{1.1}$$

A sketch of the algorithm is shown in Algorithm 1. Like LSS-LRTA*, PLRTA* has two phases in each iteration. In the first, PLRTA* expands the search space until it reaches the lookahead limit (line 3). In the second, PLRTA* separates the learning process into two steps: static learning, and dynamic learning. For static learning, PLRTA* conflates all states that differ only in time into a single abstract state. Abstract states inherit the union of all the predecessors of their preimage states, so that backups can be performed properly.

Algorithm 1 Partitioned Learning Real-Time A*

PLRTA*(S_{start},lookahead)

1: open = { S_{start} }

2: closed = { }

3: ASTAR(open, closed, lookahead)

4: $g' \leftarrow$ peek(open)

5: LEARNSTATIC(open,closed)

6: LEARNDYNAMIC(open,closed)

7: return the path from start to g'

PLRTA* learns a single static heuristic value for each abstract state, thus generalizing to future states (line 5). For dynamic learning, PLRTA* learns dynamic heuristic for each states by performing the standard Dijkstra-style backup across the LSS, considering only costs arising from the dynamic elements of the environment (line 6). For example, in a grid world domain, the cost of dynamic obstacles, which their trajectory are not fully predictable, will be learned in the dynamic learning stage. As presented by Cannon et al. (2014), the algorithm commits to only one step in the selected path (line 7), and then replans using (presumably) updated information. Although completeness is not possible in dynamic environments (imagine an obstacle teasing the agent by temporarily unblocking a shortcut), PLRTA* is guaranteed to reach a goal if there are no dynamic obstacles.

Kiesel et al. (2015) suggest two general enhancements to LSS-LRTA*. The first concerns the heuristic. Most heuristics used with real-time search algorithms are admissible, meaning that they are lower bounds on the actual cost. Kiesel et al. (2015) suggest that real-time decision-making is best made on the basis of expected value, rather than lower bounds. They attempt to derive an estimate of expected cost $\hat{h}$ by learning online an estimate of how underestimating the provided h function is. If h were accurate, the f value of a parent node should equal the lowest f value among its successors and any difference represents an error. By tracking the mean ϵ of this one-step error and multiplying it by an estimate $d(n)$

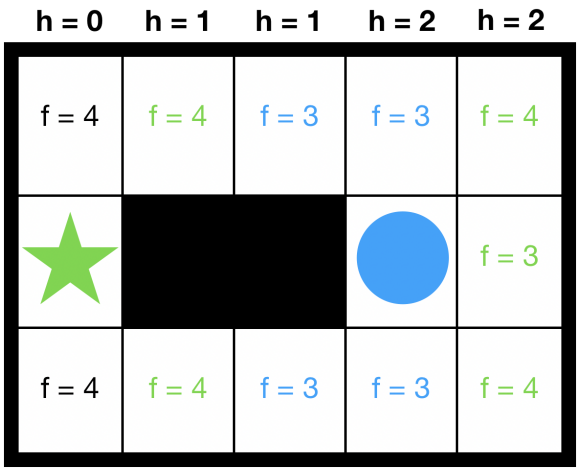

Figure 1-2: The example shows the effect of the heuristic error. Blue states represent the search space of an algorithm and green states represent the frontier states that an algorithm needs to select for the next action.

of the number of hops in the state space graph to a goal, they obtain an estimate of the h error and an estimate of the expected cost $\hat{f}$:

$$h_{error}(n) = \hat{d}(n) * \epsilon \tag{1.2}$$

$$\hat{h}(n) = h(n) + h_{error} \tag{1.3}$$

$$\hat{f}(n) = g(n) + \hat{h}(n) \tag{1.4}$$

In the example of Figure 1-2, the underestimation of the heuristic leads the algorithm to select the action which will bring the agent away from the goal. Consider the situation in the example where the agent is only allowed to expand four nodes per iteration. The agent expands four states with f value equal to 3 (blue states), then it selects the best state of the frontier states(green states) as its action. For an algorithm that sorts states base on f value, the best state in the frontier is the state with f value equal to 3; even though, the selected state is not the state toward the goal. The situation is caused by the heuristic error. As the heuristic is the lower bound of true cost-to-go, the algorithm often underestimates the heuristic value. Prioritize the node that is fewer steps away from the goal can improve the performance of the algorithm. As $\hat{f}$ trying to estimate true cost-to-go more accurately, it will penalize the states that are far away from the goal so that the state that is close to the

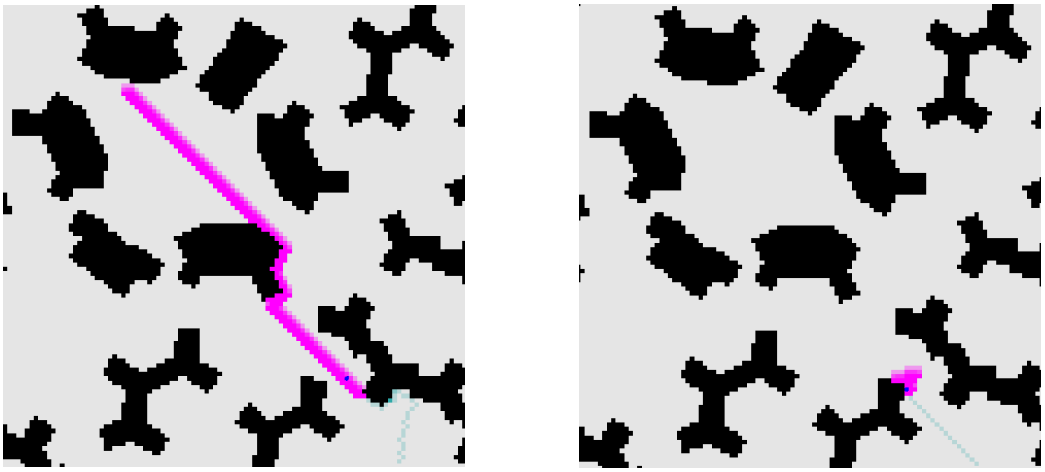

Figure 1-3: Left: Shows the search space of the algorithm with dynamic lookahead Right: Shows the search space of the algorithm without dynamic lookahead.

goal will have higher chance to either be selected as next state to expand or next action to commit.

The second enhancement concerns the treatment of lookahead. In real-time search, the lookahead size is strictly limited to ensure that an action can be selected within the real-time bound. This bound derives from the requirement that the agent never exhibit uncontrolled behavior: a new action must be ready by the time the currently executing action has completed. However, if the agent commits to multiple actions after a single planning phase, this means a larger real-time bound can be used for the next iteration. By dynamically adjusting lookahead to fill the available time in this way, action selection can be more informed, leading to better performance.

Figure 1-3 shows a comparison between the standard algorithm and the one with the dynamic lookahead technique. As states with magenta color display in the graph, the search space can have a huge difference when dynamically adjust the lookahead size.

CHAPTER 2

Dynamic-$\hat{f}$ has proven its success in improving the overall trajectory cost of LSS-LRTA*. In a situation with no dynamic obstacles, PLRTA* is similar to LSS-LRTA* where we can treat the static learning phase as the learning step of LSS-LRTA* by substitute the h_s to h. Therefore, in this chapter, we explore combining Dynamic-$\hat{f}$ and PLRTA*, resulting in Dyn-$\hat{f}$-PLRTA* and its simpler variant $\hat{f}$-PLRTA*.

2.1 $\hat{f}$-PLRTA*

As in PLRTA*, $\hat{f}$-PLRTA* copes with dynamic environments by partitioning the cost into static cost(g_s) and dynamic cost(g_d), and learning the static cost-to-go(h_s) and dynamic cost-to-go(h_d) separately. The central difference between $\hat{f}$-PLRTA* and PLRTA* is that $\hat{f}$-PLRTA* considers the heuristic error when sorting the open list:

$$g(n) = g_d(n) + g_s(n) \tag{2.1}$$

$$h(n) = h_d(n) + h_s(n) \tag{2.2}$$

$$\hat{h}(n) = h(n) + \hat{d}(n) * \epsilon \tag{2.3}$$

$$\hat{f}(n) = g(n) + \hat{h}(n) \tag{2.4}$$

$\hat{f}$-PLRTA* uses a best-first search on $\hat{f}$ to generate the local search space. Then it selects the node with the lowest $\hat{f}$ value as the next target state (g', line 4 in Algorithm 1) for the agent. Ties are broken in favor of the state with the highest time.

Algorithm 2 Static Learning For PLRTA* with $\hat{f}$

LEARNSTATIC(open,closed)

```
1: closed,open ← Initialize_Static(open,closed)
2: while closed AND open is not empty do
3:    s ← peek(open)
4:    open = open \{s}
5:    if s ∈ closed then
6:       closed = closed \{s}
7:    end if
8:    for p ∈ pred(s) do
9:       if p ∈ closed AND h_s(p) ≥ h_s(s) + c_s(p, s) then
10:         h_s(p) ← h_s(s) + c_s(p, s)
11:         if d_err(p) > d_err(s) then
12:            d_err(p) ← d_err(s)
13:            d(p) ← d(s)
14:         end if
15:      end if
16:   end for
17: end while
```

2.1.1 Calculating heuristic error

In our experiment, we used one-step heuristic error model (Kiesel et al., 2015) to estimate the heuristic error of a state.

$$\hat{d}(n) \quad = \quad \frac{d_{err}(n)}{1 - \bar{\epsilon_d}} \tag{2.5}$$

$\hat{d}$ is an estimation of the true number of steps remain to the goal from the frontier. To calculate $\hat{d}$, we average the difference of the d value of the parent and its best child (lowest f) to get the $\bar{\epsilon_d}$. We used the (d_{err}), which represents the estimation of the number of steps remain to the goal from the frontier, divide by 1 - $\bar{\epsilon_d}$ to get $\hat{d}$.

To estimate the one-step heuristic error ϵ, we averaged the difference of the $f_{static} = g_s + h_s$ value of a parent and its best child. In the same time, we also calculated the difference of the estimate steps to the goal value (d) between them to estimate the average distance error of one iteration. The average ϵ and average ϵ_d from one planning iteration is used to compute h_{error} for the next iteration, thereby avoiding any need to resort the open list as ϵ changes.

Following the procedure used in Dynamic-$\hat{f}$, during the static learning phase, the algorithm backs up d and $derr$ from the frontier, as shown in Algorithm 2. Following (Thayer, Dionne, & Ruml, 2011), we assume that the error is evenly distributed for all steps of the path. So our heuristic error is estimated by the distance-to-goal backup from frontier (d_{err}) times the per-step error ϵ and divide by 1 minus average distance error. To back up the distance from the frontier, the distance of a predecessor would be updated to its child distance value:

$$d_{err}(parent) = d_{err}(child) \tag{2.6}$$

$$d(parent) = d(child) + 1 \tag{2.7}$$

Although there is of course also error in the dynamic heuristic values, it is very hard to predict due to the inherent unpredictability of dynamic obstacles. Therefore, we only account for the static heuristic error.

2.1.2 Static world learning

Similar to PLRTA* algorithm, $\hat{f}$-PLRTA* also has a static learning phase that used to back up the value of the static heuristic from the frontier node. The main difference is that static learning phase of $\hat{f}$-PLRTA*also backs up the steps to go(d), and steps to go from the frontier(d_{err}). During the static learning phase, the algorithm updates the static heuristic value of an abstract state, that is, the state without time variable. To create the abstract state, the algorithm first converts all the states with same abstract state value into a single abstract state and combined their parent pointers to associate with the abstract state and insert them into the open list. Then the algorithm assigns infinity to every static heuristic

Algorithm 3 PLRTA* With Dynamic Lookahead

```
1: lookahead ← exp_per_step
2: while True do
3:    path ← PLRTA*(Sstart, lookahead)
4:    lookahead ← sizeof(path) * exp_per_step
5:    while path is not empty do
6:       s ← path
7:       path \{s}
8:       move agent to s
9:       if cost_new(path) > cost(path) * factor then
10:         lookahead ← exp_per_step break
11:      end if
12:   end while
13: end while
```

that is currently in the open list. The algorithm will then use abstract states to perform the Dijkstra-like algorithm to back up the d_err, d, and h_s value.

2.1.3 Heuristic Decay

In the dynamic environment, the movement of a dynamic obstacle is usually unpredictable. We may overestimate the dynamic heuristic value when an obstacle changes its moving direction suddenly. Therefore, the dynamic heuristic we learn during the dynamic learning phase may become misleading over time. The heuristic decay has been proposed to address this problem. The value of dynamic heuristic of each state will decrease at each iteration. By this, the algorithm will discount the importance of the dynamic heuristic value that it learns before.

2.2 Dyn-$\hat{f}$-PLRTA*

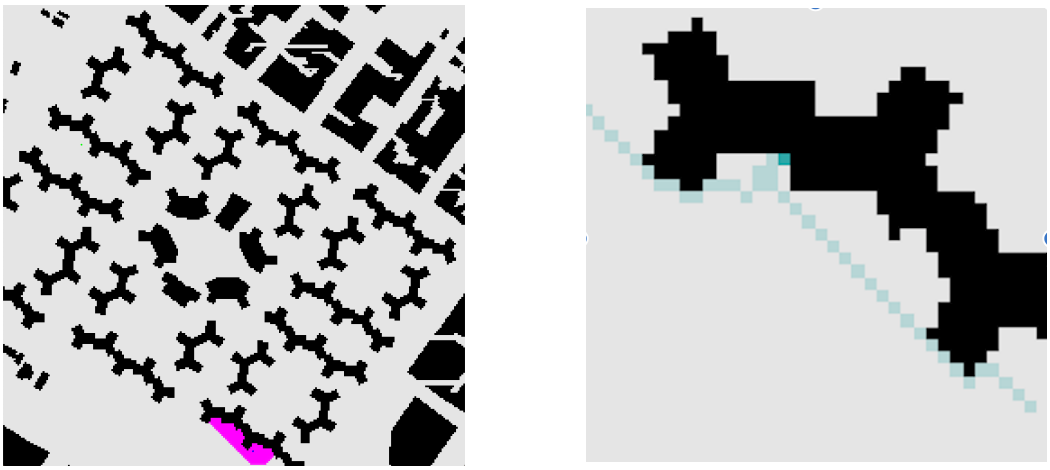

Figure 2-1: Path of Dyn-$\hat{f}$-PLRTA*

Dyn-$\hat{f}$-PLRTA* adds the dynamic lookahead technique to $\hat{f}$-PLRTA*. A sketch is shown in Algorithm 3. After the lookahead phase, the basic version of Dyn-$\hat{f}$-PLRTA* commits to all the actions along the path from the current state to the target frontier node. To take into account the changing predictions of the future locations of dynamic obstacles, Dyn-$\hat{f}$-PLRTA* checks the cost of the remaining committed actions at each time step to see if the new cost of the path is now greater than the original cost times a replanning factor (1.1 in our experiments below). If yes, then the previously-committed actions are abandoned and the real-time search is restarted with a time bound of one action duration.

2.2.1 Constrained Dynamic Lookahead

In preliminary experiments, we found that dynamic lookahead actually performed worse than committing to a single action at a time. Figure 2-1 shows an example. In the left panel, the magenta color represent the agent's LSS. The right panel shows the overall path for the agent from start to goal, and indicates that part of the trajectory contains actions that were committed at the edge of the lookahead, as they lead into a wall. Other have also reported that multiple-step commitment can perform worse than single-step (Luštrek & Bulitko, 2006; Kiesel et al., 2015). Kiesel et al. (2015) presents the comparison of LSS-LRTA* with single-step LSS-LRTA* (single-step f) in Figure 10 where single-step LSS-

Algorithm 4 PLRTA* With Constrained Dynamic Lookahead

```
1: allows_step ← 1
2: while True do
3:    lookahead ← allows_step * lookahead_size
4:    path ← PLRTA*(Sstart, lookahead)
5:    if allows_step + 1 < sizeof(path) then
6:       allows_step ← allows_step + 1
7:    else
8:       allows_step ← sizeof(path)
9:    end if
10:   while path is not empty do
11:      s ← path
12:      path \{s}
13:      move agent to s
14:      if cost_new(path) > cost(path) * factor then
15:         allows_step ← 1
16:         break
17:      end if
18:   end while
19: end while
```

LRTA* outperforms LSS-LRTA*. This makes sense, as committing to only a single step ensures lookahead from every state the agent visits. In order to allow increasing the amount of planning while ensuring some lookahead ahead of every committed actions, we developed balanced approach that constrains the number of committed actions to be at most one more than in the previous iteration (Algorithm 4). In the experiments reported below, we use this modified version of dynamic lookahead in our Dyn-$\hat{f}$-PLRTA*algorithm.

2.2.2 Garbage Collection

Both Dyn-$\hat{f}$-PLRTA* and $\hat{f}$-PLRTA* have infinite state space due to the time variable in the state. It important for these algorithms to have the garbage collection technique to save computer memory. Ideally, the garbage collection for Dyn-$\hat{f}$-PLRTA* and $\hat{f}$-PLRTA* are simply truncate all the states that have smaller timestamp than the given start state. But in practice, Dyn-$\hat{f}$-PLRTA* can only perform the garbage collection on states that are smaller than the current agent's timestamp as the Dyn-$\hat{f}$-PLRTA* might be required to replan when path cost increase.

2.2.3 Properties

Theorem 1 *In a finite state space with an admissible h and no dynamic obstacles or dead ends, Dyn-$\hat{f}$-PLRTA* is complete.*

Proof: With no dynamic obstacles, g_d and h_d will remain 0. Even if time is a state variable, h_s applies to abstract states, effecitvely removing it from consideration. Dyn-$\hat{f}$-PLRTA* will therefore be equivalent to Dynamic-$\hat{f}$, which is complete.(Kiesel et al., 2015) □

Theorem 2 *If all the action cost are non-negative and heuristic is consistent, the static heuristic value of the same abstract state is monotonically nondecreasing over time and becomes more informed over time.*

Proof: We follow Cannon et al. (2014)'s proof on nondecreasing static heuristic value. We have same environmental setting and similar static learning step. As we use the same backup strategy as they use and dynamically adjust lookahead size will not change the way how heuristic value update, we met all the precondition of their proof and we can apply their proof here. □

2.3 Experimental Results

To evaluate the practical performance of these new methods, we tested them on a grid pathfinding problem with moving obstacles. (This is reminiscent of the domain used by

Kiesel et al. (2015) to test Dynamic-$\hat{f}$, although their obstacles were fully predictable.) The state space is $\{x, y, t\}$, where x and y represent position and the t represents time. The agent has 9 actions: 8-way movement plus wait. Each action will be incurred a cost if the agent is not on the goal. The dynamic obstacles are represented by a tuple {x,y,v,h,t}, where h represents the direction of the obstacle, v represents the speed, x,y represent the position, and t represents the time of an obstacle. Obstacles move randomly with various speed and direction. The obstacles do no follow the constraint of the grid cells, and they change the direction and speed at the random time-step. The future movements of obstacles are unknown to the agent, so the agent needs to predict the possible future states of a dynamic obstacle when planning. This is done by linear extrapolation from the the two most recent time steps and assuming Gaussian uncertainty that increases at a constant value over time. The static heuristic is the octile distance. $cost_{dynamic}$ is calculated by the probability of collision times collision cost. The probability of collision is taken from the Gaussian predictions and the collision cost of $cost_{dynamic}$ is set to 200 and the $cost_{static}$ is 0 if on the goal otherwise 1. The dynamic heuristic is 0, as it is very hard to predict the true cost of dynamic obstacles.

We ran our experiments with different lookahead size range from 1 to 1000. The lookahead of the dynamic version of the algorithm is calculated by lookahead size times the number of actions committed in the previous iteration.

We use the replanning factor to decide if the dynamic algorithm need to replan the trajectory due to the increase of the dynamic cost. We experimentally compared the performance of the replanning factor from 1.0 to 2.0. And we discovered that when the replanning factor increased, the performance of the dynamic lookahead algorithm also increased. The factor parameter we chose for replanning in dynamic lookahead version of the algorithm is 1.1. Although factor 2.0 yielded the best performance, a greater factor can lead to an increased probability of hitting dynamic obstacles while it only provided a negligible improvement over the factor 1.1.

We use two different types of maps. The first is a single 256 * 256 city map from (Sturte-

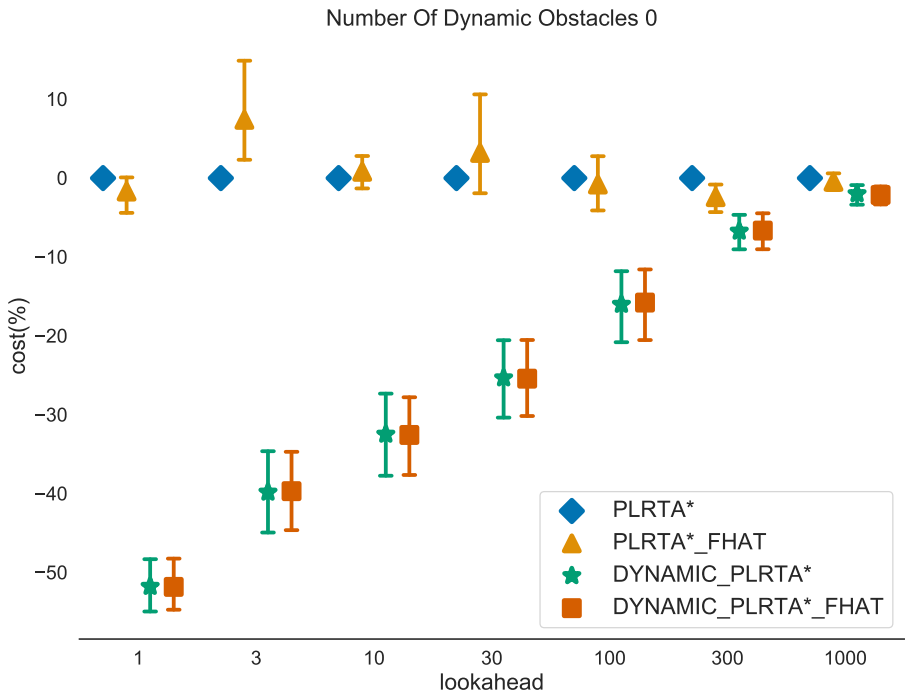

Figure 2-2: Comparison of Different Algorithms in citymap with 0 dynamic obstacles

vant, 2012), shown in Figure 2-1. Each instance of the map has a different start and goal pair that is randomly generated and at least 128 away from each other in Euclidean distance. For every start goal pair we have sub-instances that have different numbers(0,20,40,60) of random generated dynamic obstacles. The configurations of dynamic obstacles are the same across all start goal pairs The second type is the class of uniform random maps, with static obstacles generated randomly with the density of 10,30, and 50%. Each instance of the random maps has randomly generated start goal pairs and each instance has 20 dynamic obstacles. The start goal pair has the same constraint as in city map: at least 128 away from each other in Euclidean distance. The behavior of the algorithms run on different maps can refer to (Cheng & Ruml,)

Figure 2-2 to 2-6 presents the results for the city map with different dynamic obstacles number. The experiment results are shown as a relative percentage of the difference between the algorithm and PLRTA* (lower is better). One-step $\hat{f}$-PLRTA* does not show any improvement over the original PLRTA*. However, adding balanced dynamic lookahead

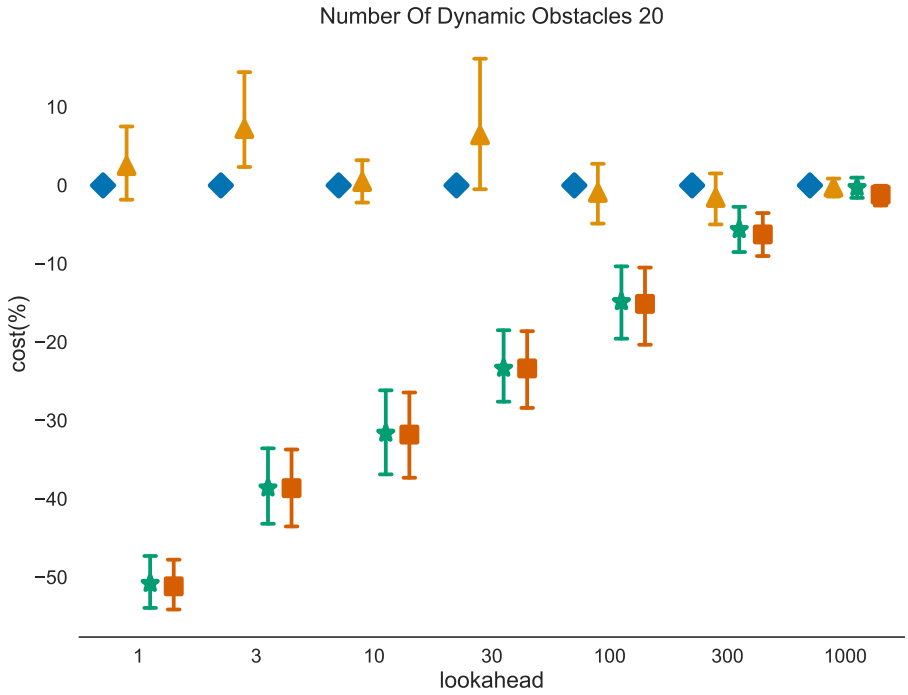

Figure 2-3: Comparison of Different Algorithms in citymap with 20 dynamic obstacles

strongly improves performance, with a stronger improvement for shorter lookaheads. (For large lookaheads, all algorithms will converge to optimal offline A*.) In the context of dynamic lookahead, using $\hat{f}$ does not seem to make much difference.

Figure 2-7 to 2-18 presents the results on random maps with different densities and dynamic obstacles.

Figure 2-7 to 2-14 shows a similar result to those from the city map, although we observe that, as the obstacle density increases and the heuristic becomes less accurate, the advantage of the dynamic lookahead algorithm strongly increases.

Figure 2-15 to 2-18 shows a situation where when there is less or no static obstacles, the performance of dynamic lookahead algorithm strongly decreases when the number of dynamic obstacles increases. The performance decreasing indicates the dynamic obstacles gradually become the crucial factor of the performance when static obstacles approach 0. It is reasonable since when there is no static obstacles and dynamic obstacles, Dyn-$\hat{f}$-PLRTA* and PLRTA* will have the same performance. When the environment only exists dynamic

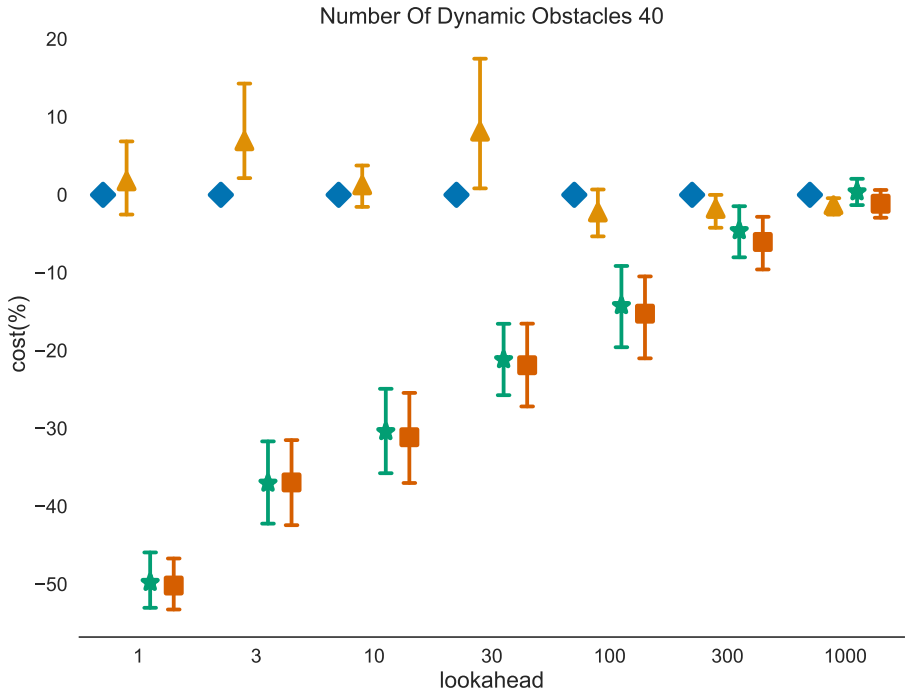

Figure 2-4: Comparison of Different Algorithms in citymap with 40 dynamic obstacles

obstacles, PLRTA* may have more advantage to react with environmental changes as it requires to replanning for every action it commits. On the other hand, Dyn-$\hat{f}$-PLRTA*may detour even if there is a better path exists. This can happen at the time that the dynamic obstacles do not follow the predicted trajectory where it frees the way it initially blocks. In this situation, Dyn-$\hat{f}$-PLRTA* will not be able to find the best route.

Figure 2-19 compares between the constraint dynamic lookahead method and the original dynamic lookahead method on Dyn-$\hat{f}$-PLRTA*. The new constrained dynamic lookahead is a clear improvemet over the original dynamic lookahead.

2.3.1 Hand Crafted Scenarios

Following (Cannon et al., 2014), we test our algorithms on the handcrafted scenario in a 20 x 20 grid world shown at Figure 2-20 to study the behavior of each algorithm. Cannon et al. (2014) set the algorithms lookahead size as 1000 and they let each algorithm to run for 30 seconds and the time bound of one iteration is 0.5 seconds. To replicate their experiment,

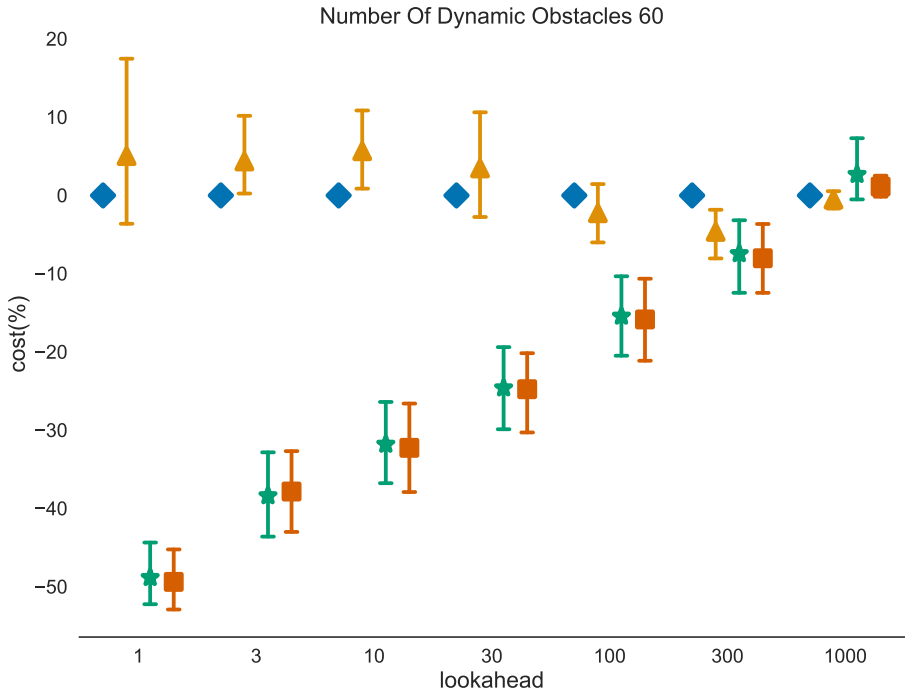

Figure 2-5: Comparison of Different Algorithms in citymap with 60 dynamic obstacles

Scenario	PLRTA*	$\hat{f}$-PLRTA*	Dyn-PLRTA*	Dyn-$\hat{f}$-PLRTA*
1	25(good)	25(good)	21(ok)	21(ok)
2	60(bad)	37(good)	37(good)	37(good)
3	18(good)	18(good)	18(good)	18(good)
4	7(good)	7(good)	7(good)	7(good)
5	19(good)	19(good)	20(good)	20(good)
6	21(good)	21(good)	25(bad)	25(bad)

Table 2-1: Performance of different algorithms on each scenario

we allow each algorithm to have 60 movements. In our domain 1000 lookaheads can cover the whole map; therefore, we decided to decrease the lookahead size to 100 to better observe the algorithm behavior. we compare the overall trajectory cost and examine the algorithm's behavior by its path.

Table 2-1 presents the result of the algorithms run on each scenario. In most of the

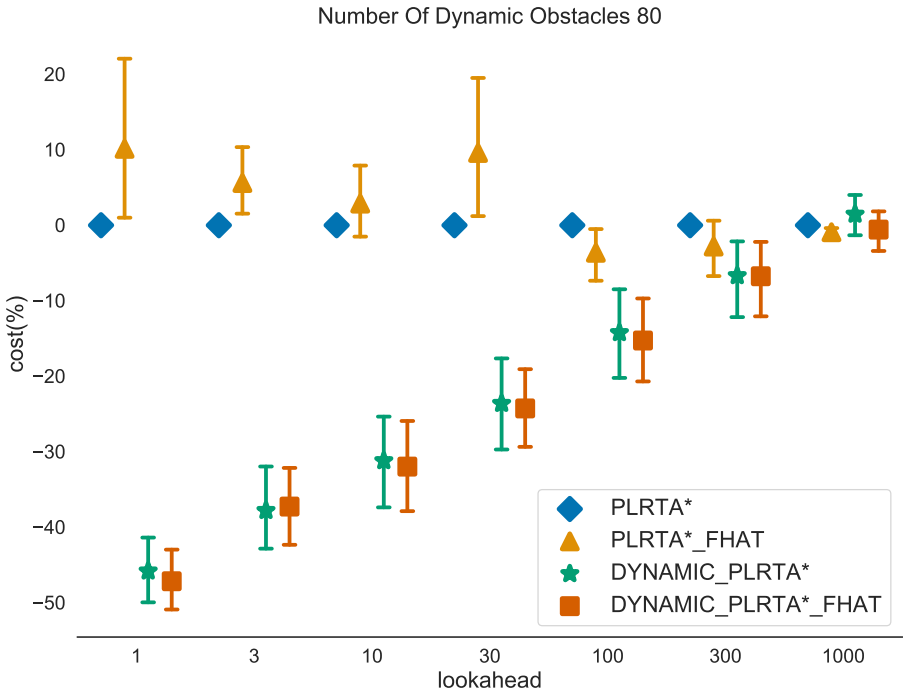

Figure 2-6: Comparison of Different Algorithms in citymap with 80 dynamic obstacles

scenario, the algorithms generally perform well and follow the behavior we expect it to have in each scenario. Only in scenario 1, the dynamic algorithm will not wait aside. Instead, it will just ask the agent to go through the space near the dynamic obstacle. In scenario 2, PLRTA* follow the slow object. Therefore, it can never reach the goal within the time-bound. In scenario 6, the dynamic algorithm will not wait at the desired location. Instead, it would randomly travel when its road dynamic obstacles block its road. For videos of different algorithms behavior can refer to (Cheng & Ruml,)

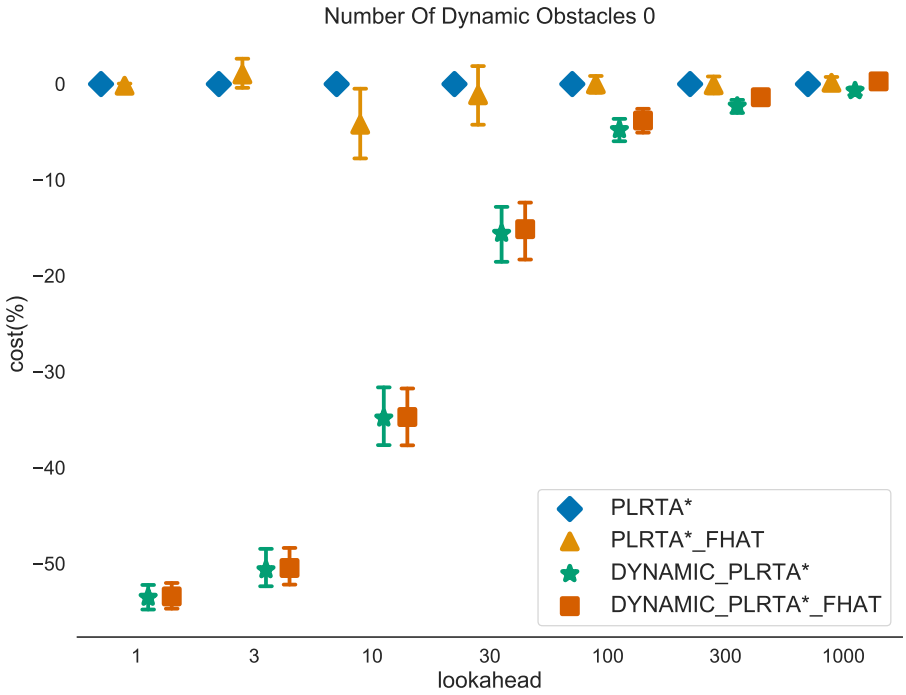

Figure 2-7: Comparison of Different Algorithms in random map with 50% density and 0 dynamic obstacles

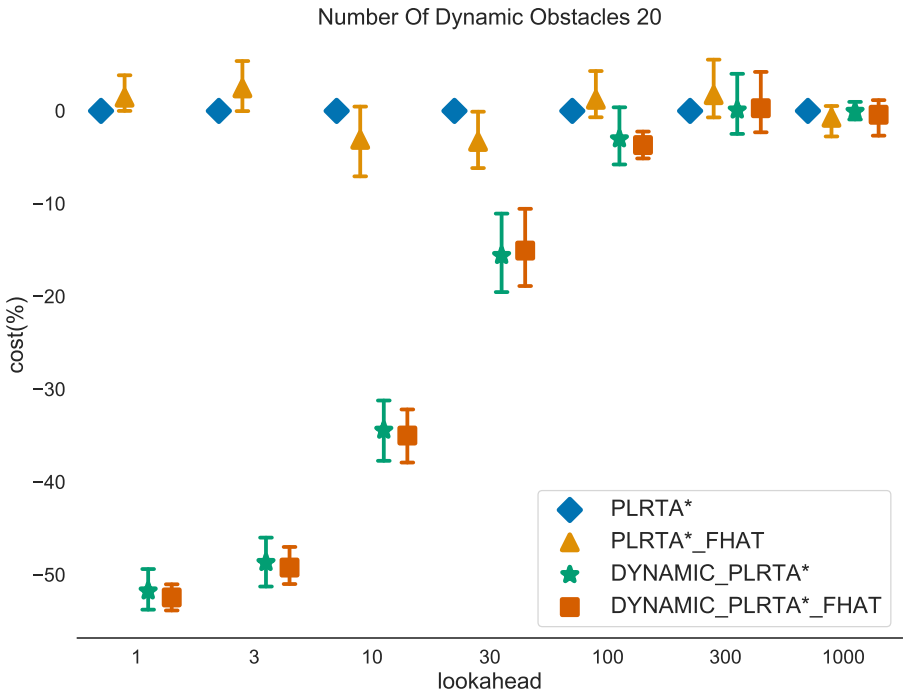

Figure 2-8: Comparison of Different Algorithms in random map with 50% density and 20 dynamic obstacles

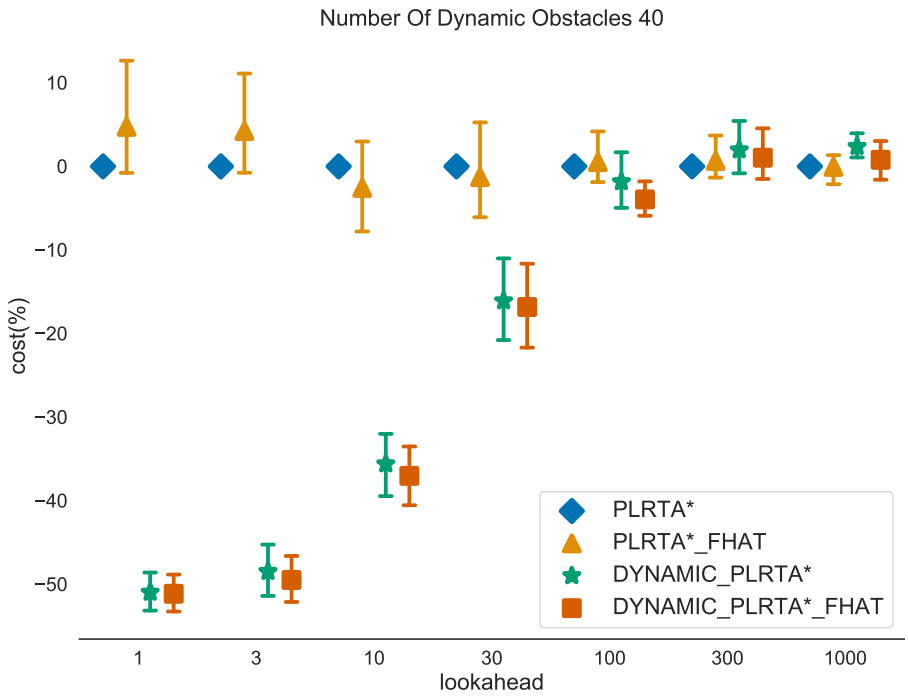

Figure 2-9: Comparison of Different Algorithms in random map with 50% density and 40 dynamic obstacles

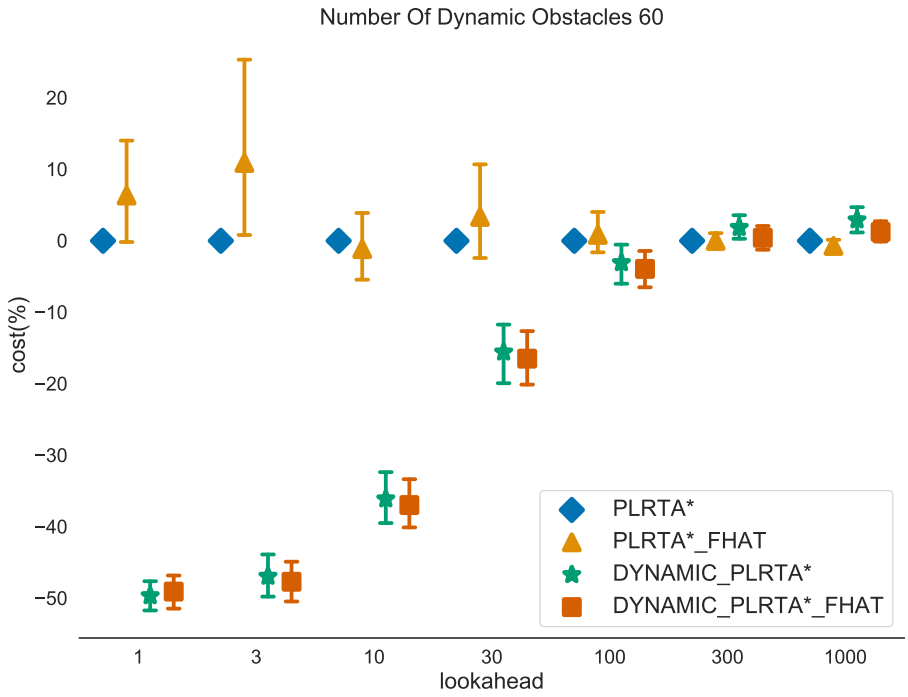

Figure 2-10: Comparison of Different Algorithms in random map with 50% density and 60 dynamic obstacles

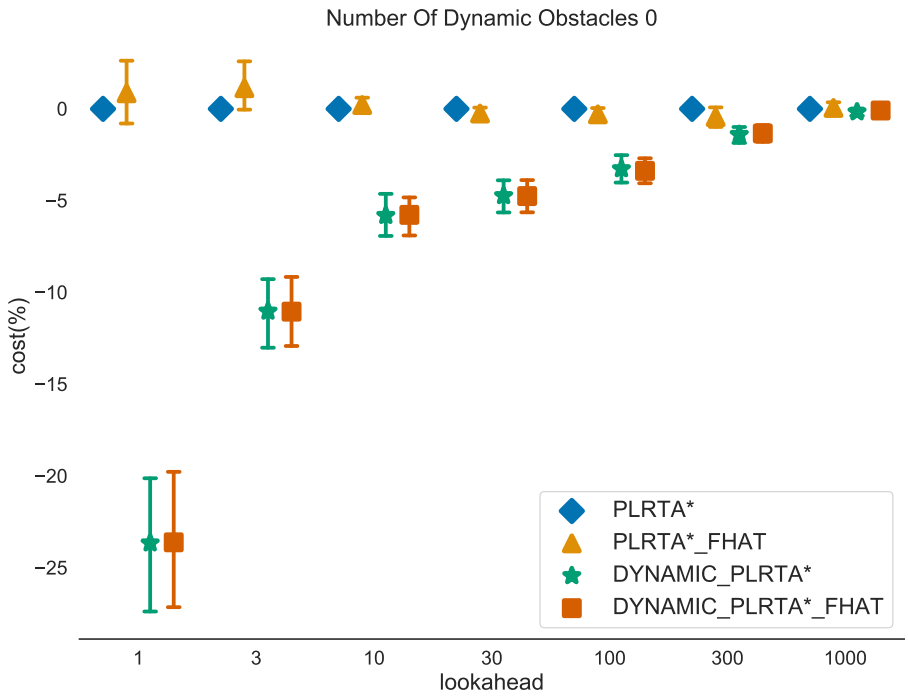

Figure 2-11: Comparison of Different Algorithms in random map with 30% density and 0 dynamic obstacles

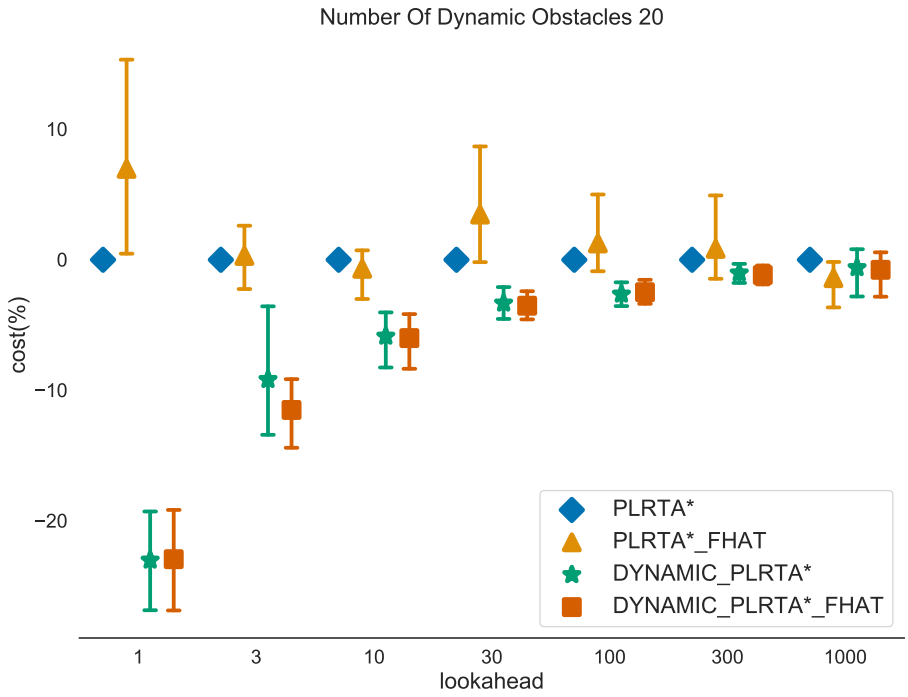

Figure 2-12: Comparison of Different Algorithms in random map with 30% density and 20 dynamic obstacles

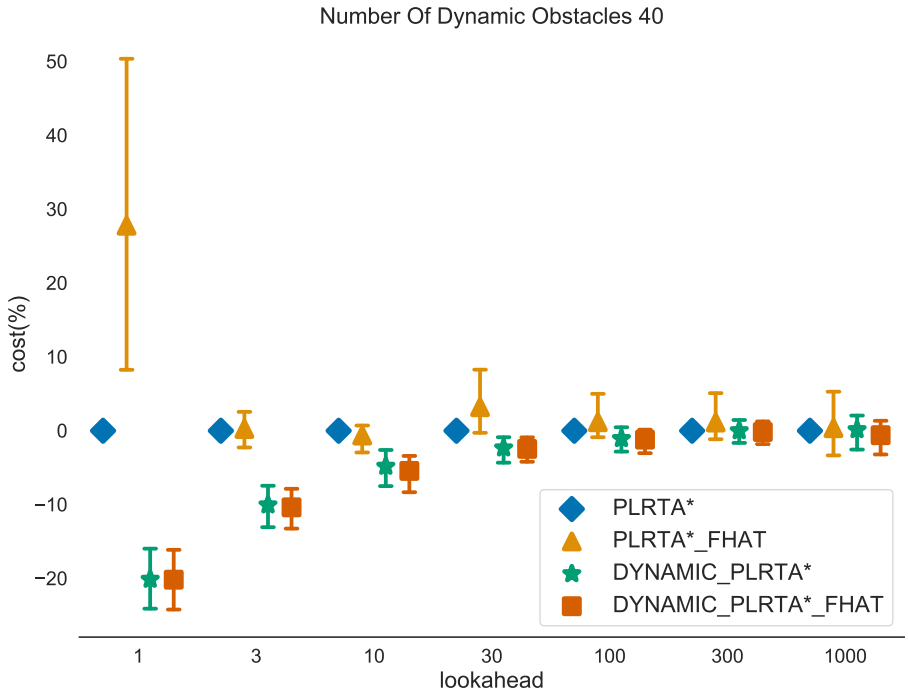

Figure 2-13: Comparison of Different Algorithms in random map with 30% density and 40 dynamic obstacles

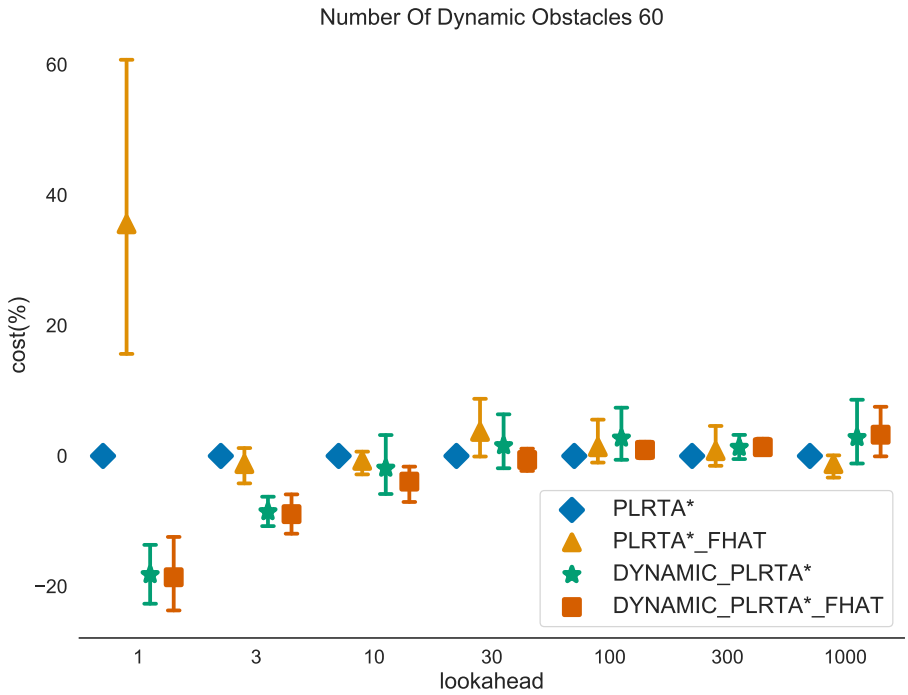

Figure 2-14: Comparison of Different Algorithms in random map with 30% density and 60 dynamic obstacles

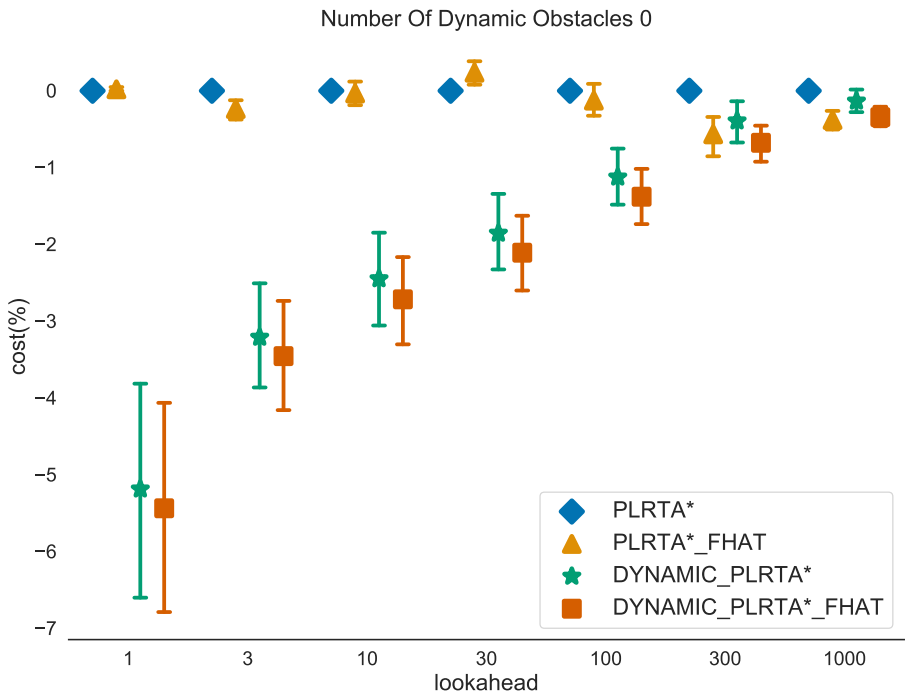

Figure 2-15: Comparison of Different Algorithms in random map with 10% density and 0 dynamic obstacles

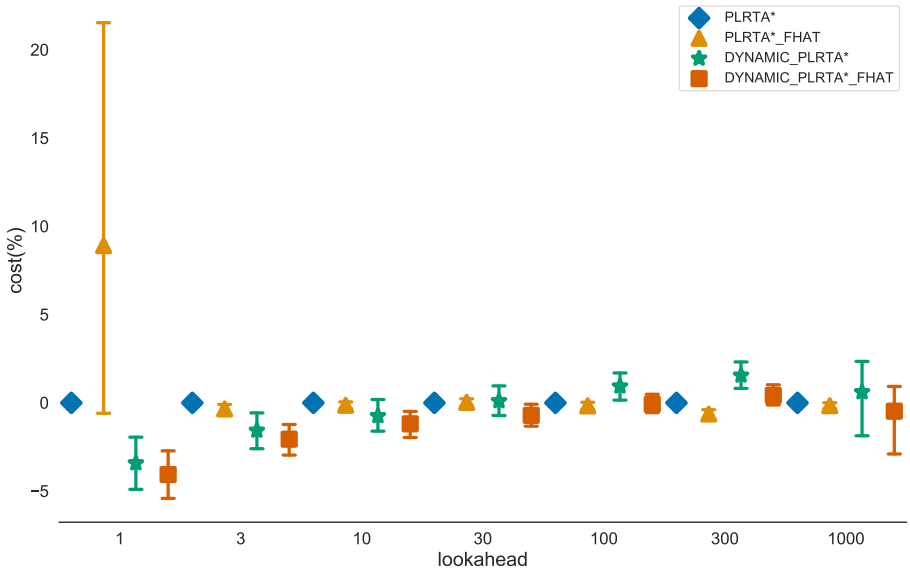

Figure 2-16: Comparison of Different Algorithms in random map with 10% density and 20 dynamic obstacles

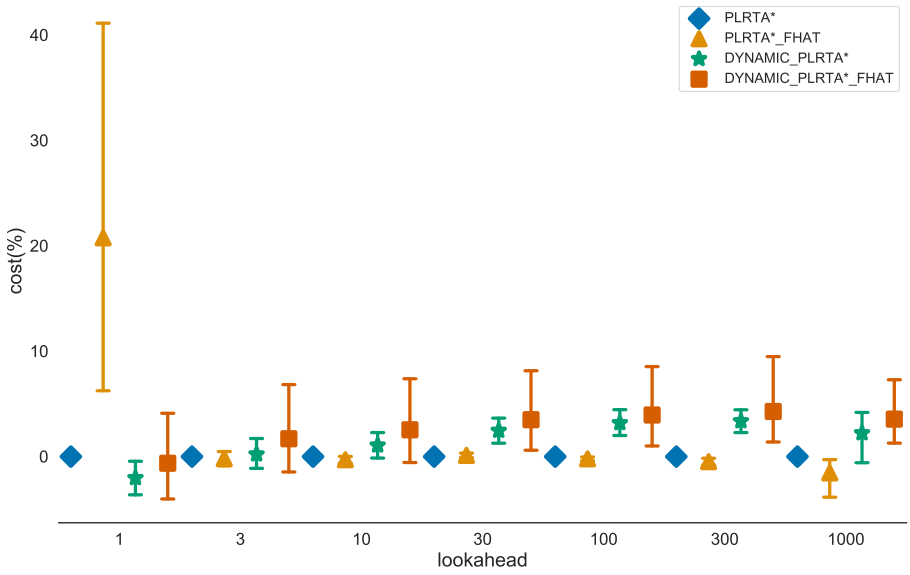

Figure 2-17: Comparison of Different Algorithms in random map with 10% density and 40 dynamic obstacles

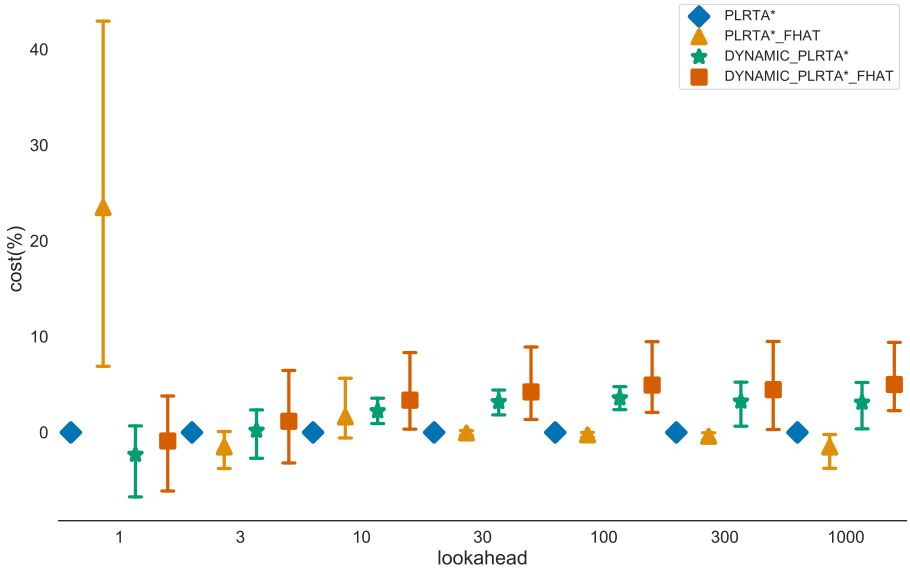

Figure 2-18: Comparison of Different Algorithms in random map with 10% density and 60 dynamic obstacles

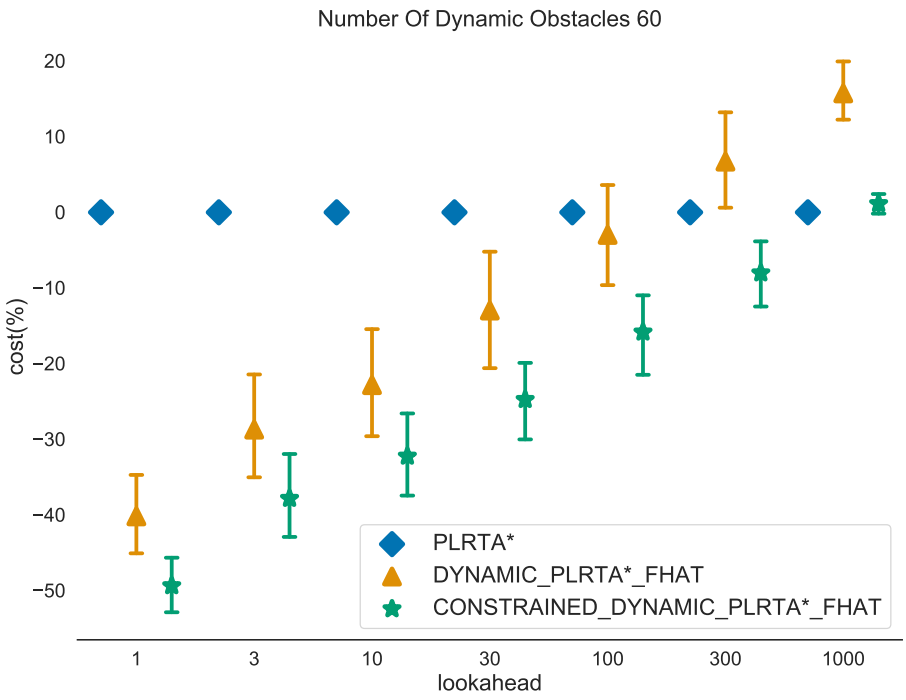

Figure 2-19: Comparison of Different Dynamic Lookahead Algorithms on Dyn-$\hat{f}$-PLRTA*

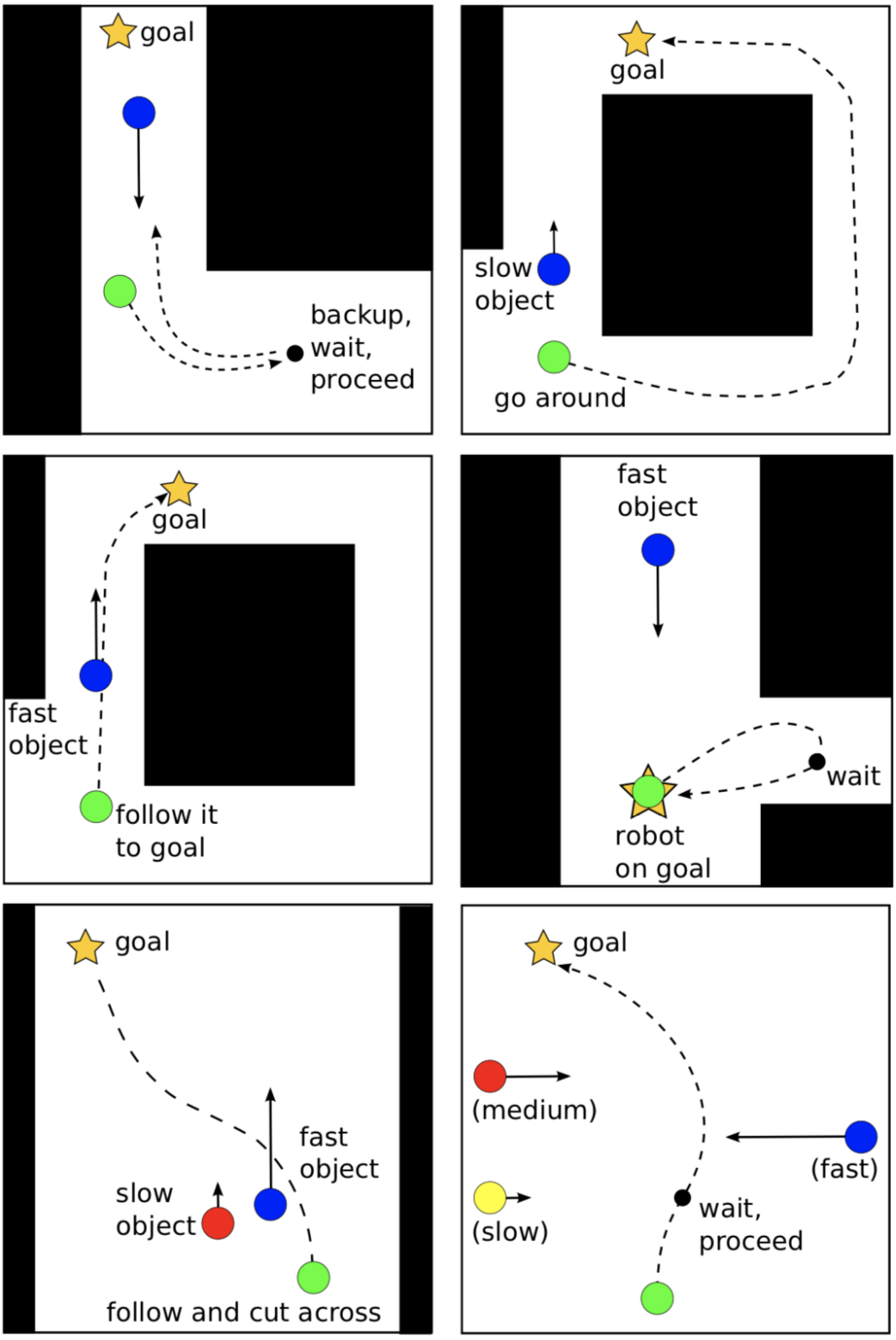

Figure 2-20: Hand Crafted Scenarious From Cannon et al. (2014)

CHAPTER 3

Conclusion

3.1 Possible Future Extension

While our experimental result suggests Dyn-$\hat{f}$-PLRTA* offers significant improvement compared to PLRTA*, $\hat{f}$-PLRTA* does not shows clear advantage. The reason $\hat{f}$-PLRTA* shows no advantage may be due to our simplistic model of heuristic error and our lack of a dynamic cost heuristic. These could be promising directions for future work. The other point we should notice is that the balanced dynamic lookahead scheme replans only when the cost of the current plan increases. It does not handle the situation where a better route appears (Bond, Widger, Ruml, & Sun, 2010). Improving this could be another direction for future work.

A large replanning factor can increase the performance of the dynamic lookahead algorithm by reducing the replanning rate. But it is also dangerous to have a big replanning factor as it will lead the algorithm to become less sensitive to environmental changes. Therefore increasing the replanning factor also increase the probability of the agent get hit by dynamic obstacles. Although we ran the experiment on the replanning factor, we did not reach the point where the algorithm starts performing worse(hit by dynamic obstacles) when the factor increase. It may be one direction that people can further improve the Dyn-$\hat{f}$-PLRTA* algorithm.

Compared to PLRTA*, Dyn-$\hat{f}$-PLRTA*shows slightly worse performance when there is no or a few static obstacles with a lot of dynamic obstacles. The reason may because of the Dyn-$\hat{f}$-PLRTA* dynamic obstacles information may be outdated when the agent commits the action in the trajectory. And this result in the agent could travel to an unnecessary state since the dynamic obstacles already leave the place. Implement a fancier replanning

condition may be able to help in this situation. Current algorithm only replan when the cost of path increase. It is interesting to see when replanning conditions include decrased of the path cost as one of its condition

While the performance of Dyn-$\hat{f}$-PLRTA* is promising in the dynamic grid pathfinding domain we tested, it will be important to understand its behavior in other domains as well. The domain used by (Cannon et al., 2014) is slightly more complex, for example, modeling orientation and velocity of the agent.

3.2 Conclusion

In this paper, we import the enhancements to LSS-LRTA* proposed by Kiesel et al. (2015) into the PLRTA* algorithm of Cannon et al. (2014), resulting in two new algorithms: $\hat{f}$-PLRTA* and Dyn-$\hat{f}$-PLRTA*. While our experimental results indicate that $\hat{f}$-PLRTA* does not offer significant benefits, Dyn-$\hat{f}$-PLRTA* outperforms the original PLRTA* algorithm in most of the tested situation and appears quite promising. For AI systems that interact with the real world, timely planning can be a crucial capability, and we hope that this work spurs additional investigation into real-time search in dynamic environments.

Bibliography

Bond, D., Widger, N. A., Ruml, W., & Sun, X. (2010). Real-time search in dynamic worlds. In *Third Annual Symposium on Combinatorial Search*.

Cannon, J., Rose, K., & Ruml, W. (2014). Real-time motion planning with dynamic obstacles. *AI Communications*, *27*, 345–362.

Cheng, C. C., & Ruml, W. Video of different algorithms behavior. `youtu.be/vKC7tPatEiQ`. Accessed: 2019-05-11.

Kiesel, S., Burns, E., & Ruml, W. (2015). Achieving goals quickly using real-time search: Experimental results in video games. *Journal of Artificial Intelligence Research*, *54*, 123–158.

Koenig, S., & 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, R. E. (1990). Real-time heuristic search. *Artificial Intelligence*, *42*(2-3), 189–211.

Kushleyev, A., & Likhachev, M. (2009). Time-bounded lattice for efficient planning in dynamic environments. In *2009 IEEE International Conference on Robotics and Automation*, pp. 1662–1668. IEEE.

Luštrek, M., & Bulitko, V. (2006). Lookahead pathology in real-time path-finding. In *Proceedings of the National Conference on Artificial Intelligence (AAAI), Workshop on Learning For Search*, pp. 108–114.

Sturtevant, N. (2012). Benchmarks for grid-based pathfinding. *Transactions on Computational Intelligence and AI in Games*, *4*(2), 144 – 148.

Thayer, J. T., Dionne, A., & Ruml, W. (2011). Learning inadmissible heuristics during search. In *Proceedings of the Twenty-First International Conference on International*

Conference on Automated Planning and Scheduling, ICAPS'11, pp. 250–257. AAAI Press.