2024 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)
October 14-18, 2024. Abu Dhabi, UAE

Asymptotically Optimal Lazy Lifelong Sampling-based Algorithm for Efficient Motion Planning in Dynamic Environments

Lu HUANG[1] and Xingjian JING[1]

***Abstract*—The paper introduces an asymptotically optimal lifelong sampling-based path planning algorithm that combines the merits of lifelong planning algorithms and lazy search algorithms for rapid replanning in dynamic environments where edge evaluation is expensive. By evaluating only sub-path candidates for the optimal solution, the algorithm saves considerable evaluation time and thereby reduces the overall planning cost. It employs a novel informed rewiring cascade to efficiently repair the search tree when the underlying search graph changes. Simulation results demonstrate that the algorithm outperforms various state-of-the-art sampling-based planners in addressing both static and dynamic motion planning problems.**

***Index Terms*—Motion planning, lifelong planning, dynamic environments, sampling-based, asymptotically optimal, lazy search**

I. Introduction

Real-time motion planning is crucial for robots to navigate safely in partially known or completely unknown environments. In such scenarios, robots must interleave planning with action executions to ensure that their plans remain viable and safe, accounting for dynamic obstacles. Efficient planning algorithms are necessary in these cases. These algorithms should be capable of updating previous solutions based on new environment information in a short time.

Early researches proposed that reusing previous search can significantly enhance planning efficiency [1], [2]. The strategy, commonly known as lifelong planning in literature, was combined with graph-based path planning techniques, such as A* [3]. In particular, Lifelong Planning A* (LPA*) and its variants D*/D* Lite, repair the same search graph that approximates the problem domain throughout the entire navigation process. Any inconsistency resulting from edge changes in the graph is identified, and the inconsistencies are efficiently propagated to the relevant parts of the search graph to repair an A*-like solution. It ensures efficient adaptation to the dynamic nature of the problem domain. However, LPA* is only resolution optimal and complete with respect to the particular underlying graph used. In other words, they almost surely find sub-optimal paths with respect to the robot and environment. Improving completeness/optimality requires recomputing the search graph, which is infeasible in applications.

Sampling-based planning methods, such as RRT* [6], RRT$^\sharp$ [7], Informed RRT* [8], in contrast, avoid an explicit offline construction of the problem domain and refines an asymptotically dense Random Rapidly-explored Graph (RRG) directly in the robot's state-space. These methods aim to find asymptotically optimal paths and probabilistically complete in terms of the robot and its environment. However, a significant drawback of these methods is their inability to efficiently handle graph changes, as they require replanning from scratch whenever a change is detected. To address this limitation, previous works have explored reusing previous search trees to improve the replanning efficiency of sampling-based planning methods [9]–[13]. Unfortunately, these methods do not guarantee the quality of their solutions. In contrast, asymptotically optimal lifelong sampling-based methods [14], [15] stand out from previous approaches by offering asymptotically optimal solutions. They use a rewiring cascade to remodel the search tree around dynamic obstacles promptly. These methods offer significant benefits, particularly in applications that require high solution quality.

Unfortunately, the existing asymptotically optimal lifelong sampling-based methods primarily focused on enhancing solution quality but need to be more indifferent to the number of edge evaluations. Specifically, they must iteratively evaluate all edges to check their feasibility before rewiring, irrespective of their relevance to the solution. In motion planning problems, edge evaluation typically consists of computationally-cost procedures, such as solving two-point boundary value problems and conducting dynamic collision checks. As a result, the cost of edge evaluation dominates the overall replanning process, especially for motion planning problems in high-dimensional configuration spaces, where the RRG must be dense enough to contain a solution.

The lazy search technique has been introduced to mitigate the problem of excessive edge evaluations [16]. It assumes that a heuristic function can efficiently compute a lower bound on edge weight and uses the lower bound to estimate whether an edge has the potential to improve the solution. This approach defers the evaluation of the actual cost of an edge until it is deemed pertinent to the solution. Various sampling-based algorithms [17]–[19] have exploited

[1]Lu HUANG and Xingjian JING are with Department of Mechanical Engineering, City University of Hongkong, Tat Chee Avenue, Kowloon, Hong Kong SAR. (e-mail: {lhuang98-c@my., xingjing@}cityu.edu.hk)

the lazy search strategy to enhance their planning efficiency. Regrettably, none can achieve both the replanning efficiency of lifelong planning techniques and the edge evaluation efficiency of lazy search techniques.

In this paper, we propose an asymptotically optimal lifelong sampling-based algorithm, Lazy Lifelong Planning Tree* (LLPT*), for online motion planning in dynamic environments where edge evaluation is expensive. The algorithm integrates the lazy search framework and the lifelong planning framework. Specifically, LLPT* approximates the problem domain by an RRG with non-overestimating heuristic edges. The actual edge evaluations of the RRG are restricted to those with the potential to be part of the solution. Whenever the RRG changes (e.g., the actual cost of some edges are evaluated or the RRG is refined), a novel informed rewiring cascade is performed to quickly repair the search tree that spans the RRG to reflect the new information and update the solution. We show that LLPT* outperforms several state-of-the-art planning algorithms (RRT* [6], Informed RRT* [8], BIT* [19], RT-RRT* [14], RRT^X [15]) over a set of motion planning problems, including both static and dynamic ones.

II. Related Works

A. Lifelong Sampling-based Algorithms

Over the past decades, lifelong sampling-based planning for dynamic motion planning problems has evolved through various iterations. Rather than discarding all previous samples like brute-force replanning, some algorithms selectively eliminate invalid search tree branches obstructed by dynamic obstacles [9], [10]. They then incrementally densify the RRG until a new solution is derived. More efficient replanning schemes only remove invalid edges of the spanning tree and preserve the separate branches for further planning. For instance, LRF retains a forest of spanning trees that are dynamically split, regrown, and merged to adapt to obstacles and robot movements [11]. Similarly, MP-RRT retains obstructed branches and actively seeks to reconnect them to the root tree, bridging the gap between the start and goal configurations [12]. RRT*FND reduces replanning effort by only reconnecting or regrowing functional disconnected branches of the previous solution path [13].

However, the aforementioned approaches are lack of formal guarantees on solution optimality. Distinguished from previous works, RT-RRT* [14] and RRT^X [15] were designed to address dynamic shortest-path planning problems (DSPP), offering an asymptotically optimal solution given sufficient replanning time. These methods leverage rewiring cascades to adapt the search tree and navigate around dynamic obstacles. In particular, RT-RRT* rewires the entire search tree structure starting from the root, which can be computationally expensive when there are many nodes in the tree. On the other hand, RRT^X uses a local rewiring cascade, which focuses only on rewiring the parts of the search tree affected by dynamic changes. This makes it more efficient in scenarios where the environment experiences minor alterations.

It is worth noting that the local rewiring cascade can be enhanced in efficiency by ordering rewiring based on the potential solution quality and constraining rewiring only to the informed node set relevant to the solution, akin to methodologies employed in informed graph-based search techniques (i.e., LPA*, D*). In the later section, we propose an informed local rewiring cascade and integrate it into the proposed algorithm.

B. Sampling-Based Planning with Lazy Collision Detection

The lazy sampling-based planning is based on the assumption that a heuristic function can estimate a lower bound on edge costs effectively. The lower bound is used to avoid unnecessary edge evalution. Lazy PRM* and Lazy RRG* [17] build a roadmap of feasible configurations like PRM* and RRG* [6], but with the difference that edges are not immediately checked for collision. Only edges that are on a better candidate path to the goal are checked for collision. If an edge is found to collide, it is removed from the roadmap, and the candidate path is updated. Similarly, Lower Bound Tree-RRT (LBT-RRT) [18] proposed the lower bound graph $\mathcal{G}_{lb}$, an RRG with lazily-evaluated edges, to approximate the problem domain. $\mathcal{G}_{lb}$ poses a lower bound invariant to the nodes in the search tree, i.e., the solution cost of the nodes in the search tree is always smaller than their solution cost via $\mathcal{G}_{lb}$. An edge is only evaluated if its lazy insertion into $\mathcal{G}_{lb}$ causes the violation against the lower bound invariant. Batch Informed Tree* (BIT*) inserts a batch of samples concurrently into the search graph with lazily evaluated edges and prioritizes the evaluation of the new edges with minor lazily-evaluated solution cost. This mechanism boosts evaluation efficiency, albeit at the expense of queuing up edges in the process.

Our work brings together lifelong sampling-based algorithms and lazy search techniques to harvest replanning and edge evaluation efficiency when solving DSPP. Our algorithm outperforms existing lifelong and lazy sampling-based algorithms in terms of planning efficiency for both static and dynamic planning problems.

III. Lazy Lifelong Planning Tree*

In this section, we first introduce the notations that will be used throughout the paper. In Section III-B, we highlight the informed rewiring cascade, a crucial component of LLPT* that efficiently rewires the underlying graph upon detecting modifications. We then present the overall algorithm framework, with a particular focus on the lazy search strategy that is used to delay collision check. Finally, we describe the graph densification strategy used to enhance the solution.

A. Notations and Problem Definitions

Assume that a robot begins its journey from v_{start} to v_{goal}. An optimal path planning problem is to find a path $\pi^*(v_{start}, v_{goal})$, an ordered set of distinct vertices that connects v_{start} and v_{goal} and minimizes the total cost to traverse this path. LLPT* solves this problem by constructing a search graph $\mathcal{G} = (V, E)$ with lazily-evaluated edges within

the robot's configuration space $\mathcal{X}$, where V represents the node set and $E \subseteq V \times V$ denotes the edge set. To clarify, each node $v \in V$ implies the corresponding configuration $x \in \mathcal{X}$ and can be directly input into distance functions. For each node, LLPT* stores its neighbor nodes in the memory for future reference. The search tree of $\mathcal{G}$ is denoted by $\mathcal{T} = (V_{\mathcal{T}}, E_{\mathcal{T}})$, rooted at v_{goal}, where $V_{\mathcal{T}} \subseteq V$ and $E_{\mathcal{T}} \subseteq E$. As mentioned in [15], Rooting $\mathcal{T}$ in v_{goal} allows the same tree and root to be used for the entire navigation, avoiding rewiring the tree from the root while the robot moves.

For each edge $e \in E$ of $\mathcal{G}$, a weight function $w : E \to (0, \infty]$ assigns a positive real number, including infinity, to this edge, e.g., the distance/time/energy cost for traversing this edge. If traversing the edge is infeasible, the edge weight will be infinity. Denote the cost of traversing the local trajectory of an edge in the free space as $\hat{w}(e)$, and E_{eval} be the set of evaluated edges at the current replanning iteration, the edge weight of $\mathcal{G}$ is defined by a lazy weight function

$$\bar{w}(e) = \begin{cases} w(e), & \text{if } e \in E_{eval}, \\ \hat{w}(e), & \text{else.} \end{cases} \tag{1}$$

The solution path should minimize the total weight, i.e., $\pi^*(v_{start}, v_{goal}) = \arg\min_{\pi \in \mathcal{G}} \sum_{e \in \Pi} w(e)$, where Π is the set of all finite cost paths connecting v_{start} to v_{goal} in $\mathcal{G}$.

B. Informed Rewiring Cascade

LLPT* dynamically adjusts the solution path by rewiring $\mathcal{T}$ when the edge weights of $\mathcal{G}$ change. Drawing inspiration from RRT^X [15] and RRT$^\sharp$ [7], LLPT* employs two cost-to-goal estimates, g(v) and lmc(v), for each node to detect local inconsistencies arising from graph modifications. A node v is called *consistent* if g(v) = lmc(v)$< \infty$, and *inconsistent* otherwise. An inconsistent node is further categorized as locally *overconsistent* if lmc(v) $<$ g(v), or locally *underconsistent* if lmc(v) $>$ g(v) or lmc(v) = g(v) =∞. The g-value is the accumulated cost-to-goal by traversing the previous search tree, while the lmc-value is the cost-to-goal based on the up-to-date graph topology and is potentially better informed than the g-value. The lmc-value satisfies the following relationship when all nodes are locally consistent:

$$lmc(v) \leftarrow \begin{cases} 0, & \text{if } v = v_{goal}, \\ \min_{u \in N(v)} \bar{w}(v, u) + lmc(u), & \text{otherwise.} \end{cases} \tag{2}$$

Then the shortest path from v_{start} to v_{goal} can be found by traversing iteratively from v_{start} to the neighbor node u that minimizes $\bar{w}(v, u) + lmc(u)$.

Modifications on edge weights potentially trigger changes in the lmc-values of some nodes and render these nodes inconsistent. LLPT* collects and stores the inconsistent nodes in a priority queue $\mathcal{Q}$. The priority of each node in the queue is based on the order of their associated keys. If a node v becomes underconsistent, which happens when it solution path to goal is evaluated to be in collision, LLPT* will use the `PropagateCostToLeave` procedure (Algorithm 3) to propagate the cost ascendance to its leaf nodes. Afterwards, both v and its leaf nodes will be added to the priority queue $\mathcal{Q}$. On the other hand, if a node v becomes overconsistent, which can happen when some of its outgoing edges have decreasing weight or when $\mathcal{G}$ is extended, v will be added to $\mathcal{Q}$.

The `ComputeShortestPath` procedure, which is provided in Algorithm 4, is called to make the spanning tree consistent again by operating on the inconsistent

Algorithm 1: Lazy Lifelong Planning Tree*

```
1  E, E_T ← φ; V ← {v_goal, v_start}, V_T ← {v_goal} ;
2  Q ← φ, k_m = 0, π ← φ;
3  while v_start ≠ v_goal do
4      Update the environment information;
5      E_eval ← φ, E_collided ← φ;
6      Update the edge weight of G based on Eq.1;
7      E_modified ← the set of edges with modified
         weight;
8      for e = (v, u) ∈ E_modified do
9          UpdateNode (v, u);
10     while TimePermits() do
11         repeat
12             π ←ComputeShortestPath ();
13             if v_start, v_goal ∈ π then
14                 V_collided =EvaluateEdge(α, π);
15                 for v ∈ V_collided do
16                     lmc(v) ← ∞;
17                     RemoveFromTree(v);
18                     Q.update(v);
19                     PropagateCostToLeave (v);
20             else
21                 break ; // No solution exist.
22         until π ⊂ E_eval and π ∩ E_collided = φ;
23         ExtendSearchGraph ();
24     if robot is moving then
25         v_last ← v_start;
26         Update v_start;
27         k_m ← k_m + w(v_last, v_start);
```

Algorithm 2: EvaluateEdge(π, α)

```
1  E_uneval ← unevaluated edges in π;
2  V_collided ← φ;
3  if |E_uneval| > α then
4      E_uneval ← first α edges in E_uneval closest to
         v_goal;
5  for e(v, u) ∈ E_uneval do
6      if e is collided with some obstacles then
7          RemoveFromTree(v);
8          lmc(v) ← ∞;
9          E_collided +← {e}, V_collided +← {v};
10     E_eval +← {e};
11 return V_collided;
```

Algorithm 3: PropagateCostToLeave(v)

```
1  lmc(v) ← ∞, V_T ←⁻ v, Q.update(v);
2  for u ∈ Child(v) do
3      PropagateCostToLeave(u);
```

Algorithm 4: ComputeShortestPath()

```
1  while |Q| > 0 and (lmc(v_start) > g(v_start) or
     lmc(v_start) = g(v_start) = ∞ or v_start ∈ Q or
     Q.topkey() < CalculateKey(v_start) do
2      k_old ← Q.topkey();
3      v ← Q.pop();
4      if k_old <CalculateKey(v) then
5          Q.update(v);
6      else
7          if lmc(v) > g(v) or lmc(v) = g(v) = ∞
             then
8              for u ∈ V_T ∩ N(v) do
9                  if lmc(v) > w̄(v, u) + lmc(u) then
10                     MakeParentOf(v,u);
11                     lmc(v) ← w̄(v, u) + lmc(u);
12         if lmc(v) ≠ ∞ then
13             for u ∈ N(v)\ Parent(v) do
14                 if lmc(u) > w̄(u, v) + lmc(v) then
15                     MakeParentOf(u,v);
16                     lmc(u) ← w̄(u, v) + lmc(v);
17                     Q.update(u);
18         g(v) ← lmc(v);
19 return path from v_start to v_goal;
```

Algorithm 5: UpdateNode(v)

```
1  if lmc(v) > w̄(v, u) + lmc(u) then
2      MakeParentOf(v,u);
3      lmc(v) ← w̄(v, u) + lmc(u);
4      Q.update(v);
```

Algorithm 6: ExtendSearchGraph()

```
1  v_rand ← RandomSample();
2  v_nearest ← Nearest(v_rand);
3  v_new ←Steer(v_nearest, v_rand, δ);
4  if v_new is in the free configuration space then
5      V ←⁺ {v_new};
6  for u ∈ N(v_new) do
7      if local trajectory exists between v and u then
8          E ←⁺ {e(v_new, u)};
9          UpdateNode(v, u);
10     if local trajectory exists between u and v then
11         E ←⁺ {e(u, v_new)};
```

nodes stored in $\mathcal{Q}$ iteratively. For a underconsistent node, `ComputeShortestPath` iteratively searches among its neighbor nodes, and the underconsistent node is rewired to a new parent node if the new parent node resulting in a better cost-to-goal (line 7-11, Algorithm 4). For a overconsistent node, its local inconsistency is disseminated to its neighbors. Specifically, if v emerges as a better parent node for a neighbor than its current parent, the neighbor's parent is updated to v, and the neighbor will be included in $\mathcal{Q}$ for further processing until consistency is restored within the nodes relevant to the solution (line 13-17, Algorithm 4).

To save computational cost, we only operate on inconsistent nodes with the potential to be in the solution path. These nodes are called *promising* in [7]. For a promising node v, an admissible cost estimate of the path from v_{start} to v_{goal} constrained to pass through it is smaller than the current best solution cost. Define the function $h(v_{start}, v) : v \to \mathbb{R}^+$ as an admissible heuristic estimate of the cost from v_{start} to v, i.e., $h(v_{start}, v) \leq w(v_{start}, v)$, and the key of a node as $k(v) = [k_1(v), k_2(v)]$, where $k_1(v) = \min(lmc(v), g(v)) + h(v)$ and $k_2(v) = \min(lmc(v), g(v))$, the definition of the promising node set V_{prom} is given as follows:

$$V_{prom} = \{v \in V | k(v) \dot{<} k(v_{start})\}, \quad (3)$$

where $\dot{<}$ is a lexicographical comparator with $k(v)\dot{<}k(u)$ if either $k_1(v) < k_2(u)$ or $k_1(v) = k_1(u) \wedge k_1(\mathbf{v}) < k_2(u)$. The node in $\mathcal{Q}$ with the smallest key is regarded as the most promising one and will be popped out from $\mathcal{Q}$ and processed first until there is not node with smaller key than v_{start} and v_{start} has found a solution path to v_{goal} (line 1, Algorithm 4). Since v_{start} is updated as the robot moves, $h(v_{start}, v)$ is changing and thus the key used by the priority queue. To avoid having to reorder the queue, we adopt the key updating strategy as D* Lite [5], where a constant k_m (initialized as 0) that is added up with the cost of the robot movement at each replanning iteration is used to mitigate the priority inconsistency of the out-of-date priority in $\mathcal{Q}$ with a newly-inserted vertex (line 27, Algorithm 1). The key of a newly-inserted vertex $k(v)$ is calculated by $k_1(v) = \min(lmc(v), g(v)) + h(v_{start}, v) + k_m$ and $k_2(v) = \min(lmc(v), g(v))$. For each node popped from the top of $\mathcal{Q}$, its key will be recalculated, and the node will be reinserted into $\mathcal{Q}$ with updated priority if its old key is smaller than the new one (line 2-5, Algorithm 4).

C. Algorithmic framework

The proposed algorithm is provided in Algorithm 1 with its major sub-routines outlined in Algorithms 2-6. Notations such as $A \xleftarrow{+} B$ represents $A \leftarrow A \cup B$, and $A \xleftarrow{-} B$ represents $A \leftarrow A \backslash B$. Set cardinality is denoted by $|\cdot|$. `Child`(v) and `Parent`(v) retrieve child nodes and the parent of node v in $\mathcal{T}$, respectively. $N(v)$ returns the neighboring nodes of v in $\mathcal{G}$. The priority queue $\mathcal{Q}$ stores nodes sorted in ascending order of their keys. $\mathcal{Q}$.`update`(v) insert or re-order node v

based on its key value. $\mathcal{Q}$.pop() pops and returns the vertex with the highest priority in $\mathcal{Q}$. $\mathcal{Q}$.topkey returns the key of the vertex with the highest priority in $\mathcal{Q}$. The function MakeParentOf(v, u) establishes a child-parent relationship between v, u and adds v to $\mathcal{T}$. RemoveFromTree(v) eliminates the child-parent relationship between v and its parent node and removes v from $\mathcal{T}$. The comparison operator with ∞ is defined such that $x < \infty$ is true for any $x \neq \infty$, otherwise false.

LLPT* initiates by setting the goal node v_{goal} with lmc(v_{goal})=g(v_{goal})=0 and the start node v_{start} with lmc(v_{start})=g(v_{start})=∞. The replanning procedure (line 3-27, Algorithm 1) iterates indefinitely at a user-defined frequency until the robot reaches the goal configuration. At the start of each replanning cycle, the search graph $\mathcal{G}$ is refreshed with lazily-computed edges (line 6, Algorithm 1). For edges with updated weights, their starting nodes are rewired using UpdateNode (line 7-9, Algorithm 1). Each replanning cycle comprises two primary loops: the outer loop (line 10-23, Algorithm 1) and the inner loop (line 11-22, Algorithm 1). The inner loop serves as the main search process, rewiring the search tree $\mathcal{T}$ and identifying the shortest collision-free path. This loop alternates between rewiring $\mathcal{T}$ to compute the shortest path and evaluating the edges along the shortest path until a collision-free shortest path is discovered. The procedure EvaluateEdge evaluates the first α unevaluated edges of the shortest path, where $\alpha \in \mathbb{N}^*$ is a user-specified parameter to balance the competing computational costs of edge evaluation and rewiring, and returns the collided nodes. For the collided nodes, their lmc-value is increased to infinity. Then, they are removed from $\mathcal{T}$ and inserted into $\mathcal{Q}$ (line 15-18, Algorithm 1). The cost-to-goal increasing information is propagated to their descendants via the procedure PropagateCostToLeave (line 19, Algorithm 1). The inner loop then continues to rewire the search tree and updates the solution (line 12, Algorithm 1). If a solution regarding current $\mathcal{G}$ is not found, the inner loop reports failure and terminates. In case of early termination of the inner loop with remaining time, the search graph $\mathcal{G}$ is extended in the outer loop to enhance potential solution quality (line 23, Algorithm 1).

D. Search Graph Densification

The ExtendSearchGraph procedure (Algorithm 6) extends the underlying search graph $\mathcal{G}$ to improve both completeness and optimality of the solution. $\mathcal{G}$ is grew to incrementally approximate the problem domain by sampling a random point v_{rand} from $\mathcal{X}$ (line 1, Algorithm 6) and extending some parts of the graph towards v_{rand}. First, the closest node in the graph $v_{nearest}$ is found, then a new sample v_{new} is generated by steering $v_{nearest}$ towards a randomly sampled point v_{rand} by distance of δ (line 3, Algorithm 6). Then, the lazily-evaluated edges among v_{new} and its neighbor nodes are added to $\mathcal{G}$ (line 6-11, Algorithm 6). In this paper, the neighbor nodes are found by searching for nodes that are within radius r of v_{new}, which is given in [6] by

$$r = \min \left\{ 2\left(1+\frac{1}{d}\right)^{\frac{1}{d}} \left(\frac{\mu\left(\mathcal{X}_{\text{free}}\right)}{\zeta_d}\right)^{\frac{1}{d}} \left(\frac{\log(N)}{N}\right)^{\frac{1}{d}}, \delta \right\} \quad (4)$$

where N=$|V|$, d is the dimension of the problem, $\mu\left(\mathcal{X}_{free}\right)$ is the Lebesgue measure of the free configuration space $\mathcal{X}_{free}$, ζ_d is the volume of a unit ball in $\mathbb{R}^d$. The lmc- and g-value of each new node are initialized as ∞. If v_{new} is able to yield a solution path via one of its neighbor node, its lmc-value will be updated, and the node will be inserted into $\mathcal{Q}$ (line 9, Algorithm 6). Later, the ComputeShortestPath procedure will propagate the new information due to the extension across the graph and search for a better solution for promising nodes.

IV. Simulation Results

In this section, we present simulation results comparing LLPT* against RRT* [6], Informed RRT* [8], BIT* [19], RRT^X [15], and RT-RRT* [14] to verify the performance of LLPT*. We compare them for static and dynamic path planning problems in $\mathbb{R}^2$, $\mathbb{R}^3$. The length for each dimension is 30.0. The ϵ parameter of RRT^X is set as 0 for all simulations. BIT* samples 150 nodes per batch regardless of problem dimension. The Euclidean norm is used as the weight and the heuristic functions for all problems. The edge resolution is set as 0.02 for collision check. The saturate parameter δ, equivalent to the maximum edge length of the search graph, is set as 2 for $\mathbb{R}^2$ and 3 for $\mathbb{R}^3$. The simulations were developed in Python 3.0 and ran on a Macbook with an M2 chip and 16GB RAM.

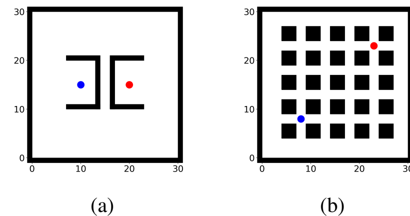
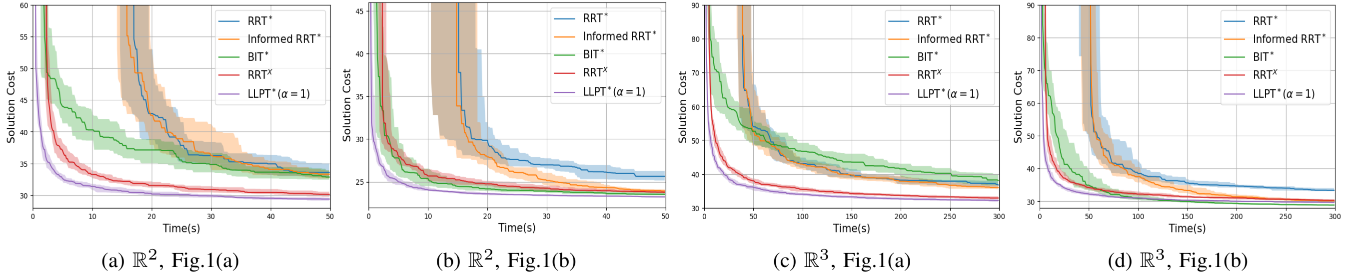


Fig. 1: The x-y plane of the obstacle configurations (represented by black regions) of the static planning problems. The red dot and the blue dot denotes the start and the goal configuration, respectively.

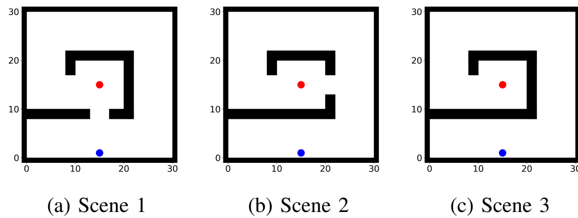
A. Static Planning Problems

The planners were tested on two abstract problems with different obstacle configurations. The x-y plane of the obstacle configurations is visualized in Fig.1. The start and goal are located at $[10, 15, 15, ..., 15]^T$ and $[20, 15, 15, ..., 15]^T$ for the problem illustrated in Fig.1(a), and $[8, 8, 8, ..., 8]^T$ and $[23, 23, 23, ..., 23]^T$ for the problem illustrated in Fig.1(b). The evaluation parameter α of LLPT* is set as 1. Each planner was tested with 50 different pseudo-random seeds.



(a) $\mathbb{R}^2$, Fig.1(a) (b) $\mathbb{R}^2$, Fig.1(b) (c) $\mathbb{R}^3$, Fig.1(a) (d) $\mathbb{R}^3$, Fig.1(b)

Fig. 2: Medium solution cost versus time for the static path planning problems illustrated in Fig.1. Each planner was tested with 50 different pseudo-random seeds. The median values are plotted with error bars denoting a non-parametric 99% confidence interval on the median.



(a) Scene 1 (b) Scene 2 (c) Scene 3

Fig. 3: The x-y plane of the obstacle configurations (represented by black regions) of the dynamic planning problems. The red dot and the black dot denotes the start and the goal configuration, respectively.

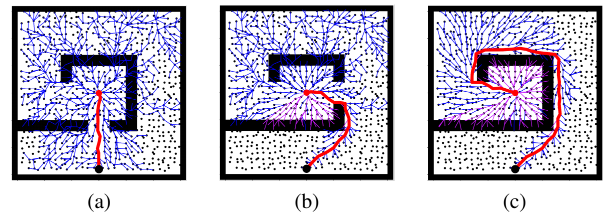
B. Dynamic Planning Problems

The simulation results are displayed in Fig.2, showcasing the medium solution cost plotted against run time. LLPT* emerges as the top performer among all algorithms, excelling in both the efficiency of initial solution discovery and the convergence rate towards the optimal solution. These results demonstrate the efficacy of lazily delaying edge evaluations in solving motion planning problems. It's worth noting that LLPT* and RRT^X outperform informed sampling-based methods (Informed RRT* and BIT*) in scenarios depicted in Fig.1(a), where the solution path is twisted. In such cases, the informed set comprises a considerable number of nodes, and the informed sampling strategies underperform. However, for the problems depicted in Fig.1(b), BIT* displays superior performance over RRT^X in $\mathbb{R}^2$, and marginally outperforms both RRT^X and LLPT* in $\mathbb{R}^3$ as run time increases. These findings suggest a potential avenue for performance enhancement by integrating LLPT* with informed sampling strategies for scenarios with simple obstacle structures.

We compared the replanning efficiency of LLPT* with RRT^X and RT-RRT in solving dynamic path planning problems in $\mathbb{R}^2$, $\mathbb{R}^3$. During planning, the obstacle configurations change from Scene 1 to Scene 3 (Fig.3). The start and goal are located at $[15, 1, 15, ..., 15]^T$ and $[15, 15, 15, ..., 15]^T$, respectively. To ensure a fair comparison, all planners assume that the problem domain is free and initialize by generating an RRG $\mathcal{G}$ that approximates the problem domain and surely contains a solution for all cases. Specifically, $\mathcal{G}$ has 3000 for $\mathbb{R}^2$ and 9000 nodes for $\mathbb{R}^3$. Each planner then undergoes three consecutive planning iterations on $\mathcal{G}$ to adapt their solutions to the changing environment. Note that the environmental change interval is more significant than the time required for replanning to ensure that the planners have enough time to optimize their solutions. Each planner was run 20 times with different pseudo-random seeds.

The average number of edge evaluations, the average number of node expansions, the average time cost, and the average solution cost for initialization and each replanning episode are tabulated in Table I. In this paper, the number of node expansions refers to the total number of nodes that are popped from the priority queue during the replanning procedure. The results demonstrate that LLPT* is able to replan successfully in all cases with significantly fewer edge evaluations while maintaining solution quality, showcasing a marked overall efficiency improvement. It's worth noting that LLPT* avoids unnecessary edge evaluations when updating its solution to Scene 1, unlike RRT^X and RT-RRT*, which perform iterative evaluations across the entire graph even when obstacle movement does not impact the solution.

Moreover, the impact of the evaluation parameter α on LLPT* performance is explored in Table 1, revealing its influence on edge evaluations and node expansions. The paper suggests careful selection of α to optimize LLPT* performance for specific problems, although the methodology for selecting the ideal value is beyond the paper's scope.



(a) (b) (c)

Fig. 4: LLPT* planning results to find the shortest path in a dynamic environment. The purple lines are the evaluated edges. The blue lines represent the edges belonging to the search tree. The red line represents the solution path.

TABLE I: The average number of edge evaluations, the average number of vertex expansions, the average replanning time, and the average solution cost for different planners over three consecutive planning in the dynamic environment shown in Fig.3. The absence of a solution was considered a cost of 200.

$\mathbb{R}^2$	LLPT*			RRT^X	RTRRT
	α=1	α=3	α=100		
Initialize					
# Eval.	10	10	10	119340	119340
# Expan.	799	799	799	11961	5728
Time (s)	47.5	47.5	47.5	67.7	67.0
Cost	14.1	14.1	14.1	14.1	14.1
1st Replan					
# Eval.	10	10	10	119340	119340
# Expan.	0	0	0	304	2677
Time (s)	0.001	0.001	0.001	11.5	13.6
Cost	14.1	14.1	14.1	14.1	14.1
2nd Replan					
# Eval.	205	341	415	119340	119340
# Expan.	3846	3530	3294	2104	2693
Time (s)	2.1	1.81	1.74	12.9	14.7
Cost	23.3	23.3	23.3	23.3	23.3
3rd Replan					
# Eval.	562	852	974	119340	119340
# Expan.	44817	42799	38216	151	2660
Time (s)	14.4	13.5	12.8	12.8	14.5
Cost	51.8	51.8	51.8	200	51.8
Total Time (s)	64.0	62.8	62.0	104.9	109.8
$\mathbb{R}^3$					
Initialize					
# Eval.	6	6	6	303122	303122
# Expan.	1486	1486	1486	20144	541619
Time (s)	430.0	430.0	430.0	516.7	521.4
Cost	14.8	14.8	14.8	14.8	14.8
1st Replan					
# Eval.	6	6	6	303122	303122
# Expan.	0	0	0	823	8067
Time (s)	0.001	0.001	0.001	46.0	48.9
Cost	14.8	14.8	14.8	14.8	14.8
2nd Replan					
# Eval.	475	798	839	303122	303122
# Expan.	6000	5766	5724	5914	8072
Time (s)	2.2	2.2	2.2	51.9	54.2
Cost	24.8	24.8	24.8	24.8	24.8
3rd Replan					
# Eval.	2087	3359	3629	303122	303122
# Expan.	99080	97759	97131	4100	7989
Time (s)	29.6	27.1	27.4	50.1	52.6
Cost	55.2	55.2	55.2	55.2	55.2
Total Time (s)	461.8	459.3	459.6	664.7	677.1

V. Conclusions

We introduce LLPT*, an asymptotically optimal lazy lifelong sampling-based path planning algorithm. LLPT* combines the replanning efficiency of lifelong planning, the search efficiency of heuristic-guided informed algorithms, and the evaluation efficiency of lazy search algorithms. Our simulations show that LLPT* outperforms several sampling-based algorithms for both static and dynamic motion planning problems. These results suggest LLPT* is well-suited for time-constrained scenarios, particularly for robots in dynamic environments.

References

[1] Ramalingam, Ganesan, and Thomas Reps. "An incremental algorithm for a generalization of the shortest-path problem." Journal of Algorithms 21.2 (1996): 267-305.

[2] Frigioni, D., Marchetti-Spaccamela, A. and Nanni, U. (2000) 'Fully dynamic algorithms for maintaining shortest paths trees', Journal of Algorithms, 34(2), pp. 251–281. doi:10.1006/jagm.1999.1048.

[3] P. Hart, N. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," IEEE Transactions on Systems Science and Cybernetics, vol. 4, no. 2, pp. 100–107, 1968.

[4] S. Koenig, M. Likhachev, and D. Furcy, "Lifelong planning A*," Artificial Intelligence, vol. 155, no. 1-2, pp. 93–146, 2004.

[5] S. Koenig and M. Likhachev, "Fast replanning for navigation in unknown terrain," IEEE Transactions on Robotics, vol. 21, no. 3, pp. 354–363

[6] S. Karaman and E. Frazzoli, "Sampling-based algorithms for optimal motion planning," The International Journal of Robotics Research, vol. 30, no. 7, pp. 846–894, 2011.

[7] O. Arslan and P. Tsiotras, "Use of relaxation methods in sampling-based algorithms for optimal motion planning," 2013 IEEE International Conference on Robotics and Automation, 2013.

[8] Gammell, Jonathan D., Siddhartha S. Srinivasa, and Timothy D. Barfoot. "Informed RRT: Optimal sampling-based path planning focused via direct sampling of an admissible ellipsoidal heuristic." 2014 IEEE/RSJ international conference on intelligent robots and systems. IEEE, 2014.

[9] D. Ferguson, N. Kalra, and A. Stentz, "Replanning with RRTS," Proceedings 2006 IEEE International Conference on Robotics and Automation, 2006.

[10] K. E. Bekris and L. E. Kavraki, "Greedy but safe replanning under kinodynamic constraints," Proceedings 2007 IEEE International Conference on Robotics and Automation, 2007.

[11] R. Gayle, K. R. Klingler, and P. G. Xavier, "Lazy Reconfiguration Forest (LRF) - an approach for motion planning with multiple tasks in dynamic environments," Proceedings 2007 IEEE International Conference on Robotics and Automation, 2007.

[12] M. Zucker, J. Kuffner, and M. Branicky, "Multipartite rrts for rapid replanning in Dynamic Environments," Proceedings 2007 IEEE International Conference on Robotics and Automation, 2007.

[13] O. Adiyatov and H. A. Varol, "A novel RRT*-based algorithm for motion planning in Dynamic Environments," 2017 IEEE International Conference on Mechatronics and Automation (ICMA), 2017.

[14] K. Naderi, J. Rajamäki, and P. Hämäläinen, "RT-RRT*," Proceedings of the 8th ACM SIGGRAPH Conference on Motion in Games, 2015.

[15] M. Otte and E. Frazzoli, "RRTx: Asymptotically optimal single-query sampling-based motion planning with quick replanning," The International Journal of Robotics Research, vol. 35, no. 7, pp. 797–822, 2015.

[16] Bohlin, R. and Kavraki, L.E. (2000) 'Path planning using Lazy PRM', Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No.00CH37065) [Preprint]. doi:10.1109/robot.2000.844107.

[17] Hauser, K. (2015) 'Lazy collision checking in asymptotically-optimal motion planning', 2015 IEEE International Conference on Robotics and Automation (ICRA) [Preprint]. doi:10.1109/icra.2015.7139603.

[18] Salzman, O. and Halperin, D. (2016) 'Asymptotically near-optimal RRT for fast, high-quality motion planning', IEEE Transactions on Robotics, 32(3), pp. 473–483. doi:10.1109/tro.2016.2539377.

[19] Gammell, J.D., Barfoot, T.D. and Srinivasa, S.S. (2020) 'Batch informed trees (BIT*): Informed Asymptotically Optimal Anytime Search', The International Journal of Robotics Research, 39(5), pp. 543–567. doi:10.1177/0278364919890396.

[20] Sucan, I.A., Moll, M. and Kavraki, L.E. (2012) 'The Open Motion Planning Library', IEEE Robotics and Automation Magazine, 19(4), pp. 72–82. doi:10.1109/mra.2012.2205651.