Parallelized Search on Graphs with Expensive-to-Compute Edges

Shohin Mukherjee

CMU-RI-TR-23-10

April, 2023

The Robotics Institute
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Robotics.

For my parents, Dr. Sraban Mukherjee and Dr. Urmi Mukherjee, who have been my pillars of support. For my grandfather, Dr. Debabrata Banerjea, who has been my idol, and through his immense dedication and devotion, inspired me to pursue a career in science.

Abstract

Search-based planning algorithms enable robots to come up with well-reasoned long-horizon plans to achieve a given task objective. They formulate the problem as a shortest path problem on a graph embedded in the state space of the domain. Much research has been dedicated to achieving greater planning speeds to enable robots to respond quickly to changes in the environment. Additionally, as the task complexity increases, it becomes important to incorporate more sophisticated models like simulators in the planning loop. However, these complex models are expensive to compute and prohibitively reduce planning speed. Because of the plateau in CPU clock speed, single-threaded planning algorithms have hit a performance plateau. On the other hand, the number of CPU cores has grown significantly, a trend that is likely to continue. This calls for the need for planning algorithms that exploit parallelization. However, unlike sampling-based planning algorithms, parallelizing search-based planning algorithms is not trivial if optimality or bounded sub-optimality is to be maintained due to their sequential nature. A key feature of robotics domains is that the major chunk of computational effort during planning is spent on computing the outcome of an action and the cost of the resulting edge instead of searching the graph. In this thesis, we exploit this insight and develop several parallel search-based planning algorithms that harness the multithreading capability of modern processors to parallelize edge computations. We show that these novel algorithms drastically improve planning times across several domains.

Our first contribution is a parallelized lazy search algorithm, Massively Parallelized Lazy Planning (MPLP). The existing lazy search algorithms are designed to run as a single process and achieve faster planning by intelligently balancing computational effort between searching the graph and evaluating edges. The key idea that MPLP exploits is that searching the graph and evaluating edges can be performed asynchronously in parallel. On the theoretical front, we show that MPLP provides rigorous guarantees of completeness and bounded suboptimality.

As with all lazy search algorithms, MPLP assumes that successor states can be generated without evaluating edges, which allows the algorithm to defer edge evaluations and lazily proceed with the search. However, this assumption does not always hold, for example, in the case of simulation-in-the-loop planning, which uses a computationally expensive simulator to generate successors. To that end, our second contribution is Edge-Based Parallel A* for Slow Evaluations (ePA*SE) which interleaves planning with the parallel evaluation of edges while guaranteeing optimality. We also present its bounded suboptimal variant that trades off optimality for planning speed.

For its applicability in real-time robotics, ePA*SE must compute plans under a time budget and therefore have anytime performance. Though lower solution cost is desired, it is not the first priority in such settings. Our third contribution is Anytime Edge-Based Parallel A* for Slow Evaluations (A-ePA*SE), which brings the anytime property to ePA*SE.

ePA*SE targets domains with expensive but similar edge computation times. However, in several robotics domains, the action space is heterogenous in the computational effort required to evaluate the outcome of an action and its cost. Therefore, our fourth contribution is Generalized Edge-Based Parallel A* for Slow Evaluations (GePA*SE), which generalizes ePA*SE to domains where edge computations vary significantly. We show that GePA*SE outperforms ePA*SE and other baselines in domains with heterogenous actions by employing a parallelization strategy that explicitly reasons about the computational effort required for their evaluation.

Finally, we demonstrate the utility of parallelization in an algorithm that integrates graph search techniques with trajectory optimization (INSAT). Since trajectory optimization is computationally expensive, running INSAT on a single thread limits its practical use. The proposed parallelized version Parallelized Interleaving of Search and Trajectory Optimization (PINSAT) achieves several multiple increases in planning speed and significantly higher success rates.

Acknowledgments

The nature of research is collaborative. Likewise, this Ph.D. was not a solo endeavor, and there were several people that contributed to its success. Besides the people mentioned here, I would like to express my gratitude towards all my friends, colleagues and mentors that I met along the way.

I would first like to thank my advisor Prof. Maxim Likhachev for his many years of mentorship. Max, besides being an incredible scientist, is an extremely nice person. He is also a great teacher, and I flourished under his guidance. There are very few people I have met who can match his work ethic. His innate scientific curiosity, paired with his calm temperament, makes him the perfect advisor. A Ph.D. is always a challenging pursuit, but Max made it a pleasant experience for me. I am incredibly fortunate to have him as a mentor for life. I would also like to thank my committee members Prof. Oliver Kroemer, Prof. Stephen Smith and Prof. Oren Salzman for their guidance in shaping this thesis. I am grateful to my M.S. advisor Prof. Cameron Riviere for guiding me early in my research career at CMU. Prof. Yukinori Kobayashi at Hokkaido University and Prof. Ravi Vaidyanathan at Imperial College London gave me the opportunity to experience research in my early years as an undergraduate student, and these experiences were pivotal in setting the foundation for this Ph.D. I would like to thank the faculty at the Indian Institute of Technology, Guwahati, in particular my bachelor's thesis advisor Prof. Santosha K. Dwivedy, for teaching me the foundations of engineering.

The Search-Based Planning Lab is where I spend a large part of my Ph.D. life. I had the privilege of working with most of the lab members on various projects, and I learned a lot from all of them. In particular, I would like to thank Andrew Dornbush, who helped me with the software libraries that he wrote, which were extremely useful in my research. Dr. Sandip Aine co-authored several of my papers and helped me set the direction for my research. Dr. Chris Paxton, Dr. Arsalan Mousavian and Prof. Dieter Fox were my mentors during my time at Nvidia AI Robotics Research Lab, and because of them, it was a great learning experience.

Finally, I am grateful to my family for standing by me all these years. In particular, my parents, Dr. Sraban Mukherjee and Dr. Urmi Mukherjee, for their love, support and encouragement all my life. They instilled in me the importance of higher education at an early age. I would especially like to express my profound sense of gratitude towards my grandfather Dr. Debabrata Banerjea, who devoted his life to science and was a source of great inspiration for me. As I begin my scientific career, I will live by the principles of dedication, devotion and discipline that he ingrained in me.

Funding

This thesis was supported by the ARL-sponsored A2I2 program, contract W911NF-18-2-0218, and ONR grant N00014-18-1-2775.

Contents

When this dissertation is viewed as a PDF, the page header is a link to this Table of Contents.

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Graph search algorithms such as A* and its variants [1, 2, 3] are widely used in robotics for task and motion planning problems which can be formulated as the shortest path problem on a graph embedded in the state-space of the domain. Fig 1.1 shows a few examples of such problems. Faster graph search algorithms can be useful in various domains where the robot needs to quickly plan online to react to changes in the environment. The example shown in Figure 1.1 top involves online motion planning for a fleet of drones for persistent coverage of an area of interest. Fast motion planning is critical for the drone to be sufficiently reactive by planning at a high frequency, especially when flying at high speeds. They can also be useful to speed up planning problems that use computationally expensive models like simulators in the loop, which otherwise would take a prohibitively large amount of time. The example shown in Figure 1.1 bottom involves solving a long horizon manipulation task and motion planning problem of assembling a set of blocks. Each pick and place action in the task internally involves solving a computationally expensive motion planning problem. Additionally, the place action uses a simulator to assess the tower's stability after placing a block. Computing a feasible task and motion plan in this domain using wA* takes about an hour.

For the aforementioned reasons, a lot of research has been focused on exploiting specific characteristics of the domain to achieve faster planning. These include devel-

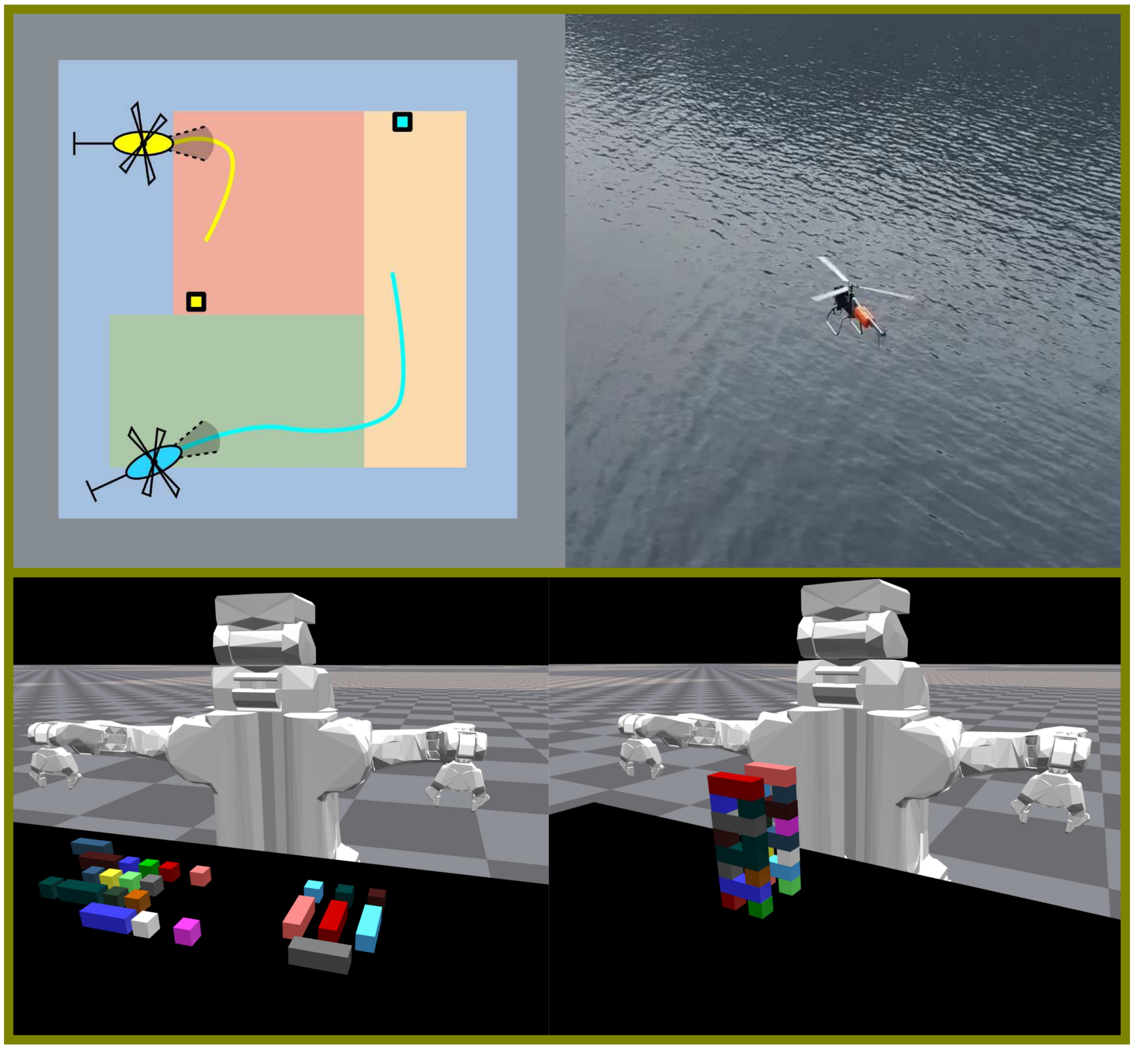


Figure 1.1: Examples of planning in robotics. Top: Motion planning for UAVs to navigate to assigned goals [4]. Bottom: Task and motion planning for a PR2 mobile manipulator to arrange blocks into a desired structure [5].

oping better heuristics, incorporating multiple heuristics [3], planning over multiple resolutions [7], etc. These approaches achieve improved performance by utilizing computation more intelligently to solve the planning problem. However, none of these improvements increase the rate of computation the algorithms can do, which is primarily dictated by the CPU speed. Unfortunately, single thread speeds have plateaued over the years [8]. However, the number of logical cores in a CPU has continued to increase, as seen in Figure 1.2. Therefore, there is a case to be made

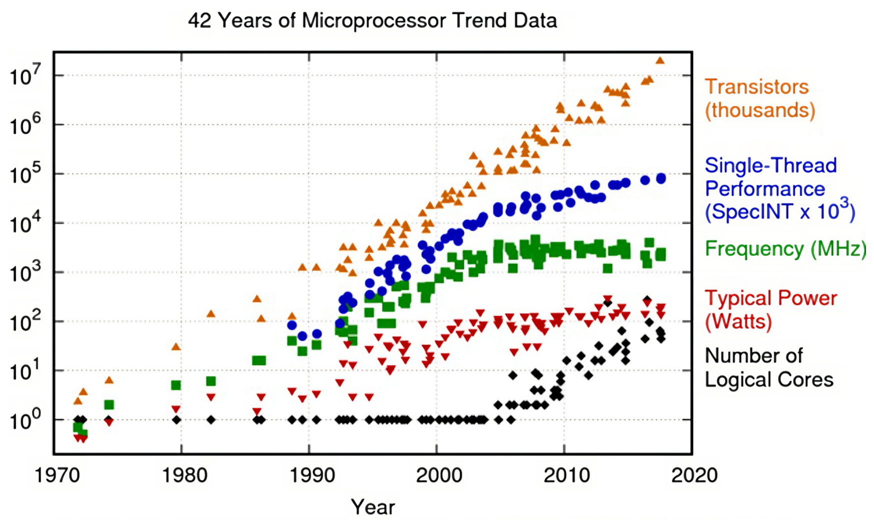


Figure 1.2: Four decades of microprocessor trend data [6]. The increase in the number of transistors is no longer resulting in an increase in single-thread speed. However, the number of logical cores has been on a steady and consistent rise.

for developing algorithms that exploit the parallel processing capability of modern processors.

The question that still needs to be answered is what part of the search should be parallelized. To answer this question, realizing what part of the shortest path problem requires the most computation is important. The computational cost of solving the shortest path problem using graph search can be split between 1) the cost of traversing through and searching the graph (discovering states, maintaining and managing ordered data structures, rewiring vertices, etc.) and 2) evaluating the cost of the edges. In planning for robotics, edge evaluation tends to be the bottleneck of solving the shortest path problem. For example, in planning for robot manipulation, edge evaluation typically corresponds to collision checks of a robot model against the world model at discrete interpolated states along the edge. Collision-checking an edge can be expensive depending on how these models are represented (meshes, spheres, etc.) and how finely the states to be collision checked are interpolated. Edge evaluations are prohibitively expensive when planners incorporate computationally expensive models in the loop, such as simulators [9, 10]. This is also the case for algorithms that incorporate trajectory optimization with the search [11, 12].

Therefore in this thesis, we develop algorithms that speed up search-based planning by parallelizing edge evaluations and demonstrate how these algorithms can benefit planning and decision-making in robotics.

1.2 Thesis Research Contributions

In this thesis, we develop several graph search algorithms that leverage parallelization to speed up planning in domains with expensive-to-compute edges.

1.2.1 MPLP: Massively Parallelized Lazy Planning

Lazy search algorithms achieve faster planning by deferring the evaluation of discovered edges during the search and instead using estimates of edge costs to search the graph optimistically. This thesis presents the first parallelized lazy search algorithm that optimistically searches the graph and evaluates edges asynchronously in parallel. This eliminates the need to toggle between these two operations, as is the case with serial lazy search algorithms. On the theoretical front, we show that MPLP provides rigorous guarantees of optimality or bounded suboptimality if heuristics are inflated. On the experimental front, we show that MPLP outperforms the state-of-the-art lazy search and parallel search algorithms. MPLP is discussed in detail in Chapter 4 and in our paper [5].

1.2.2 ePA*SE: Edge-based Parallel A* for Slow Evaluations

As with all lazy search algorithms, MPLP assumes that successor states can be generated without evaluating edges, which allows the algorithm to defer edge evaluations and lazily proceed with the search. However, this assumption doesn't hold for several planning domains in robotics. For such domains, we present ePA*SE that interleaves the search with the parallel evaluation of edges. ePA*SE changes the basic unit of search from state expansions to edge expansions. This decouples the evaluation of edges from the expansion of their common parent state, giving the search the flexibility to figure out what edges need to be evaluated to solve the planning problem. We show that ePA*SE provides rigorous optimality guarantees. In addition, ePA*SE can be trivially extended to handle an inflation weight on the heuristic resulting in a bounded suboptimal algorithm w-ePA*SE (Weighted ePA*SE) that trades off optimality for faster planning. ePA*SE is discussed in detail in Chapter 5 and in our paper [13].

1.2.3 A-ePA*SE: Anytime Edge-based Parallel A* for Slow Evaluations

Though ePA*SE drastically achieves lower planning times than its serial counterparts, it needs to come up with a solution under a strict time budget for its applicability in real-time robotics. Though the optimal solution is preferable, that is not the first priority in such settings. Therefore, we bring the anytime property to ePA*SE. We show that the resulting algorithm, A-ePA*SE, achieves higher efficiency than existing anytime algorithms. A-ePA*SE is discussed in detail in Chapter 6 and in our paper [14].

1.2.4 GePA*SE: Generalized Edge-based Parallel A* for Slow Evaluations

ePA*SE targets domains where the action space comprises actions with expensive but similar evaluation times. However, in several robotics domains, the action space is heterogenous in the computational effort required to evaluate the cost of an action and its outcome. Motivated by this, in Chapter 7, we develop GePA*SE, which generalizes the key ideas of PA*SE and ePA*SE, i.e., parallelization of state expansions and edge evaluations, respectively. This extends its applicability to domains that have actions requiring varying computational effort to evaluate them. GePA*SE is also discussed in detail in our paper [15].

1.3 Open Source Software Contribution

To enable wide use and further research, algorithms developed in this thesis and the relevant baselines have been open-sourced. The repository contains the domain-agnostic implementation of the algorithms developed in this thesis. Additionally, it contains efficient implementations of several other common search algorithms used

as baselines in this thesis. We believe this repository will be useful to communities in and beyond Robotics interested in parallelized search-based planning algorithms.
Parallel Search Repository: https://github.com/shohinm/parallel_search

1.4 Excluded Research Contributions

A portion of the doctoral research has been excluded to keep the thesis succinct, which is listed here.

- A planning framework for persistent, multi-UAV coverage with global deconfliction [4].
- Reactive Long Horizon Task Execution via Visual Skill and Precondition Models [16].

Chapter 2

Background

Search-based planning algorithms formulate the planning problem as a least-cost path problem on a graph embedded in the state space of the domain. They construct the graph by recursively applying a set of actions from every state, starting with the start state. The process of applying a set of actions to a state to generate its successor states is called a *state expansion* and is the basic algorithmic step of almost all search algorithms. However, different search algorithms differ in the order in which states are expanded. In this chapter, we will first define the search-based planning problem more formally and then discuss some common algorithms that can be used to solve it. These algorithms are the basic building blocks of this thesis.

2.1 Search-based Planning Problem Definition

Let a finite graph $G = (\mathcal{V}, \mathcal{E})$ be defined as a set of vertices $\mathcal{V}$ and directed edges $\mathcal{E}$. Each vertex $v \in \mathcal{V}$ represents a state $\mathbf{s}$ in the state space of the domain $\mathcal{S}$. An edge $e \in \mathcal{E}$ connecting two vertices v_1 and v_2 in the graph represents an action $\mathbf{a} \in \mathcal{A}$ that takes the agent from corresponding states $\mathbf{s}_1$ to $\mathbf{s}_2$. In this thesis, we assume that all actions are deterministic. Hence an edge e can be represented as a pair $(\mathbf{s}, \mathbf{a})$, where $\mathbf{s}$ is the state at which action $\mathbf{a}$ is executed. For an edge e, we will refer to the corresponding state and action as $\mathbf{s}$ and $e.\mathbf{a}$ respectively.

- $\mathbf{s}_0$ is the start state.

- $\mathcal{G}$ is the goal region.
- $c : \mathcal{E} \rightarrow [0, \infty]$ is the cost associated with an edge.
- $c : \mathcal{S} \times \mathcal{S} \rightarrow [0, \infty]$ is the minimum cost between a pair of states.
- $g(\mathbf{s})$ or g-value is the cost of the best path to $\mathbf{s}$ from $\mathbf{s}_0$ found by the algorithm so far.
- $h(\mathbf{s})$ is a consistent and therefore admissible heuristic [17]. It never overestimates the cost to the goal.
- A path π is an ordered sequence of edges $e_{i=1}^{N} = (\mathbf{s}, \mathbf{a})_{i=1}^{N}$, the cost of which is denoted as $c(\pi) = \sum_{i=1}^{N} c(e_i)$.

Objective: Find a path π from $\mathbf{s}_0$ to a state in the goal region $\mathcal{G}$ with the optimal cost c^*.

2.2 Key Ingredients of Search

All search algorithms described in this thesis share a common set of functions that we introduce here. Though search algorithms can be formulated in pseudo-code differently, in this thesis, we will stick to a consistent notation and style.

- $\{\mathbf{s}', c\} \leftarrow$ GetSuccessor$(\mathbf{s}, \mathbf{a})$: Returns the successor state $\mathbf{s}'$ and cost c when an action $\mathbf{a}$ is applied to a state $\mathbf{s}$. For some domains, it can be decomposed into GenerateSuccessor and EvaluateEdge.
- $\mathbf{s}' \leftarrow$ GenerateSuccessor$(\mathbf{s}, \mathbf{a})$: Returns the successor state $\mathbf{s}'$ when an action $\mathbf{a}$ is applied to a state $\mathbf{s}$. This is also where a discretization is imposed on the state space. Search algorithms maintain a hash map of states that have been discovered in the search so far. If $\mathbf{s}'$ is detected as a duplicate of a state already present in the hash map based on the discretization being used, the pre-existing state is returned instead. If there is no duplication detection, the search will run on an infinite graph in the continuous state space.
- $c \leftarrow$ EvaluateEdge$(\mathbf{s}, \mathbf{s}')$: Compute the cost of the edge from $\mathbf{s}$ to $\mathbf{s}'$.
- $\pi \leftarrow$ Backtrack$(\mathbf{s})$: Build a path from $\mathbf{s}_0$ to $\mathbf{s}$ by backtracking from the state $\mathbf{s}$ using the *parent* pointer of every state until the start state $\mathbf{s}_0$.

It is important to note that in practice, GetSuccessor cannot always be explicitly split into GenerateSuccessor and EvaluateEdge. In several domains, the successor cannot be generated without evaluating the connecting edge. For example, in the case of simulation-in-the-loop planning, actions involve forward simulation of an expensive physics simulator to generate successors [18]. The generation of the successor and the evaluation of the connecting edge are coupled together. In either case, in robotics, GetSuccessor is typically the most computationally expensive part of the search.

2.3 BFS: Breadth First Search

Algorithm 1 BFS: Breadth First Search

```
 1: $\mathbf{s}_0 \leftarrow$ start state , $\mathcal{G} \leftarrow$ goal region, $\mathcal{A} \leftarrow$ action space
 2: $OPEN \leftarrow$ queue
 3: procedure Plan
 4:     $OPEN$.Push($\mathbf{s}_0$)
 5:     while $OPEN \neq \emptyset$ do
 6:         $\mathbf{s} \leftarrow OPEN$.front()
 7:         if $\mathbf{s} \in \mathcal{G}$ then
 8:             return Backtrack($\mathbf{s}$)
 9:         else
10:             Expand($s$)
11:         end if
12:     end while
13:     return $\emptyset$
14: end procedure
15: procedure Expand($\mathbf{s}$)
16:     insert $\mathbf{s}$ in $VISITED$
17:     for $\mathbf{a} \in \mathcal{A}$ do
18:         $\{\mathbf{s}', c\} \leftarrow$ GetSuccessor($\mathbf{s}, \mathbf{a}$)
19:         if $\mathbf{s} \notin VISITED$ then
20:             $\mathbf{s}'.parent = \mathbf{s}$
21:             $OPEN$.Push($\mathbf{s}'$)
22:         end if
23:     end for
24: end procedure
25: procedure GetSuccessor($\mathbf{s}, \mathbf{a}$)
26:     $\mathbf{s}' \leftarrow$ GenerateSuccessor($\mathbf{s}, \mathbf{a}$)
27:     $c \leftarrow$ EvaluateEdge($\mathbf{s}, \mathbf{s}'$)
28:     return $\{\mathbf{s}', c\}$
29: end procedure
```

BFS (Alg. 1) is the simplest shortest path algorithm. The order of state expansions is imposed by a queue. When a state is expanded, its successors are generated, which are then marked *visited*, and their parent is set (Line 20). Once a state has been marked visited, it is never revisited. When a state in the goal region is popped from the queue, the path can be constructed by backtracking using the parent back pointers (Line 8). BFS has no notion of cost and returns the least-cost path only when all edges have the same cost. If the edges have a non-uniform cost, it returns the shortest length path.

2.4 Dijkstra

Algorithm 2 Dijkstra's Algorithm

```
1: $\mathbf{s}_0 \leftarrow$ start state , $\mathcal{G} \leftarrow$ goal region, $\mathcal{A} \leftarrow$ action space
2: $OPEN \leftarrow$ priority queue with priority $g(\mathbf{s})$, minimum priority in the front
3: procedure PLAN
4:     $OPEN$.PUSH($\mathbf{s}_0$)
5:     while $OPEN \neq \emptyset$ do
6:         $\mathbf{s} \leftarrow OPEN$.MIN()
7:         if $\mathbf{s} \in \mathcal{G}$ then
8:             return BACKTRACK($\mathbf{s}$)
9:         else
10:            EXPAND($s$)
11:        end if
12:    end while
13:    return $\emptyset$
14: end procedure
15: procedure EXPAND($\mathbf{s}$)
16:    for $\mathbf{a} \in \mathcal{A}$ do
17:        $\{\mathbf{s}', c\} \leftarrow$ GETSUCCESSOR($\mathbf{s}, \mathbf{a}$)
18:        if $g(\mathbf{s}) + c < g(\mathbf{s}')$ then
19:            $g(\mathbf{s}') = g(\mathbf{s}) + c$
20:            $\mathbf{s}'.parent = \mathbf{s}$
21:            $OPEN$.PUSH($\mathbf{s}', g(\mathbf{s}') + c$)
22:        end if
23:    end for
24: end procedure
25: procedure GETSUCCESSOR($\mathbf{s}, \mathbf{a}$)
26:    $\mathbf{s}' \leftarrow$ GENERATESUCCESSOR($\mathbf{s}, \mathbf{a}$)
27:    $c \leftarrow$ EVALUATEEDGE($\mathbf{s}, \mathbf{s}'$)
28:    return $\{\mathbf{s}', c\}$
29: end procedure
```

Dijkstra (Alg. 1) is the simplest least-cost path algorithm. Dijkstra keeps track of the best cost-to-come for every state from the start state as their g-value. The order of state expansions is imposed by a priority queue ordered by their g-values. When a state is expanded, its successors are generated, and their cost-to-come is updated (Line 19). Dijkstra provably expands states optimally, i.e., when a state is expanded, its g-value is the least cost to get to that state from the start. Therefore, when a goal state is popped for expansion, the least-cost path can be constructed by backtracking using the parent back pointers.

2.5 A*

Algorithm 3 A*

```
1: $\mathbf{s}_0 \leftarrow$ start state , $\mathcal{G} \leftarrow$ goal region, $\mathcal{A} \leftarrow$ action space
2: $OPEN \leftarrow$ priority queue with priority $g(\mathbf{s}) + h(\mathbf{s})$, minimum priority in the front
3: procedure PLAN
4:     $OPEN$.PUSH($\mathbf{s}_0$)
5:     while $OPEN \neq \emptyset$ do
6:         $\mathbf{s} \leftarrow OPEN$.MIN()
7:         if $\mathbf{s} \in \mathcal{G}$ then
8:             return BACKTRACK($\mathbf{s}$)
9:         else
10:            EXPAND($s$)
11:        end if
12:    end while
13:    return $\emptyset$
14: end procedure
15: procedure EXPAND($\mathbf{s}$)
16:    for $\mathbf{a} \in \mathcal{A}$ do
17:        $\{\mathbf{s}', c\} \leftarrow$ GETSUCCESSOR($\mathbf{s}, \mathbf{a}$)
18:        if $g(\mathbf{s}) + c < g(\mathbf{s}')$ then
19:            $g(\mathbf{s}') = g(\mathbf{s}) + c$
20:            $\mathbf{s}'.parent = \mathbf{s}$
21:            $OPEN$.PUSH($\mathbf{s}', g(\mathbf{s}') + h(\mathbf{s}')$)
22:        end if
23:    end for
24: end procedure
25: procedure GETSUCCESSOR($\mathbf{s}, \mathbf{a}$)
26:    $\mathbf{s}' \leftarrow$ GENERATESUCCESSOR($\mathbf{s}, \mathbf{a}$)
27:    $c \leftarrow$ EVALUATEEDGE($\mathbf{s}, \mathbf{s}'$)
28:    return $\{\mathbf{s}', c\}$
29: end procedure
```

A* (Alg. 3) [1] is a more informed algorithm that uses a heuristic function to guide the search and find a solution quicker than Dijkstra. The goal of the heuristic function is to use an easy-to-compute estimate of the cost to the goal to bias the search into expanding states that are more likely to be on the least-cost path. This bias is imposed by a priority queue ($OPEN$) that stores the discovered states ordered by their f-value, i.e., $f(\mathbf{s}) = g(\mathbf{s}) + h(\mathbf{s})$, minimum priority first (Line 21). As long as the heuristic is *admissible*, i.e., it never overestimates the true minimum cost to the goal for every state, A* expands states optimally.

$$h(\mathbf{s}) \leq c(\mathbf{s}, \mathbf{s}_g) \ \forall \mathbf{s}_g \in \mathcal{G}$$

Algorithm 4 wA*: Weighted A*

```
1: $\mathbf{s}_0 \leftarrow$ start state , $\mathcal{G} \leftarrow$ goal region, $\mathcal{A} \leftarrow$ action space
2: $OPEN \leftarrow$ priority queue with priority $g(\mathbf{s}) + w \cdot h(\mathbf{s})$, minimum priority in the front
3: procedure PLAN
4:     $OPEN$.PUSH($\mathbf{s}_0$)
5:     while $OPEN \neq \emptyset$ do
6:         $\mathbf{s} \leftarrow OPEN$.MIN()
7:         if $\mathbf{s} \in \mathcal{G}$ then
8:             return BACKTRACK($\mathbf{s}$)
9:         else
10:            EXPAND($s$)
11:            insert $\mathbf{s}$ in $CLOSED$
12:        end if
13:    end while
14:    return $\emptyset$
15: end procedure
16: procedure EXPAND($\mathbf{s}$)
17:    for $\mathbf{a} \in \mathcal{A}$ do
18:        $\{\mathbf{s}', c\} \leftarrow$ GETSUCCESSOR($\mathbf{s}, \mathbf{a}$)
19:        if $\mathbf{s}' \notin CLOSED$ and $g(\mathbf{s}) + c < g(\mathbf{s}')$ then
20:            $g(\mathbf{s}') = g(\mathbf{s}) + c$
21:            $\mathbf{s}'.parent = \mathbf{s}$
22:            $OPEN$.PUSH($\mathbf{s}', g(\mathbf{s}') + w \cdot h(\mathbf{s}')$)
23:        end if
24:    end for
25: end procedure
26: procedure GETSUCCESSOR($\mathbf{s}, \mathbf{a}$)
27:    $\mathbf{s}' \leftarrow$ GENERATESUCCESSOR($\mathbf{s}, \mathbf{a}$)
28:    $c \leftarrow$ EVALUATEEDGE($\mathbf{s}, \mathbf{s}'$)
29:    return $\{\mathbf{s}', c\}$
30: end procedure
```

2.6 wA*: Weighted A*

wA* (Alg. 4) [2] is a more greedy variant of A* that uses an inflation factor ($w > 1$) to inflate the heursitic in the priority of $OPEN$ i.e. $f(\mathbf{s}) = g(\mathbf{s}) + w \cdot h(\mathbf{s})$. If the heuristic is *consistent*, A* expands states bounded suboptimally i.e. $g(\mathbf{s}) \leq w \cdot g^*$, without re-expanding states [19]. Therefore, a state that has been expanded is added to a $CLOSED$ list (Line 11) and is never re-expanded. A heuristic is consistent if, for every state $\mathbf{s} \notin \mathcal{G}$ and its successor $\mathbf{s}'$,

$$h(\mathbf{s}) \leq c(\mathbf{s}, \mathbf{s}') + h(\mathbf{s}')$$

and for every state $\mathbf{s} \in \mathcal{G}$,

$$h(\mathbf{s}) = 0$$

A consistent heuristic is also admissible.

2.7 PA*SE: Parallel A* For Slow Expansions

In order to speed up planning in domains where state expansions are slow, an optimal parallelized planning algorithm PA*SE (Parallel A* for Slow Expansions) and its (bounded) suboptimal version w-PA*SE (Weighted PA*SE) were developed [20]. This thesis builds on the idea of *state independence* from PA*SE, which we describe in detail here.

Unlike other parallel search algorithms, in which the number of times a state can be re-expanded increases with the degree of parallelization [21, 22, 23], PA*SE expands states in a way that each state is expanded at most once. The key idea in PA*SE is that a state $\mathbf{s}$ can be expanded before another state $\mathbf{s}'$, if $\mathbf{s}$ is *independent* of $\mathbf{s}'$, i.e., expansion of $\mathbf{s}'$, cannot lead to a shorter path to $\mathbf{s}$. If the independence relationship holds in both directions, i.e., $\mathbf{s}'$ is also independent of $\mathbf{s}$, then $\mathbf{s}$ and $\mathbf{s}'$ can be expanded in parallel. A state $\mathbf{s}$ in the open list is independent of every other state in the open list with a larger priority, which is why it can be expanded before any of the states that are behind it in the open list. However, this cannot be said for the states that have a smaller priority than $\mathbf{s}$ and hence are in front of it in the open list. Therefore, a state that is not in front of the open list (and therefore does not have the smallest priority) can only be expanded in parallel with the states that have a smaller priority after undergoing the independence check. More specifically, this means that the state $\mathbf{s}$ has to be independent of the states that are in front of it in the open list as well as the states that are not in the open list but are in the process of being expanded. PA*SE assumes there exists a pairwise heuristic function $h(\mathbf{s}, \mathbf{s}')$ that provides an estimate of the cost between any pair of states. It is forward-backward consistent i.e.

$$h(\mathbf{s}, \mathbf{s}'') \leq h(\mathbf{s}, \mathbf{s}') + h(\mathbf{s}', \mathbf{s}'') \; \forall \; \mathbf{s}, \mathbf{s}', \mathbf{s}''$$

and

$$h(\mathbf{s}, \mathbf{s}') \leq c^*(\mathbf{s}, \mathbf{s}') \; \forall \; \mathbf{s}, \mathbf{s}'$$

Note that using h for both the unary heuristic $h(\mathbf{s})$ and the pairwise heuristic $h(\mathbf{s}, \mathbf{s}')$ is a slight abuse of notation since these are different functions.

In addition to $OPEN$ and $CLOSED$, PA*SE uses another data structure BE

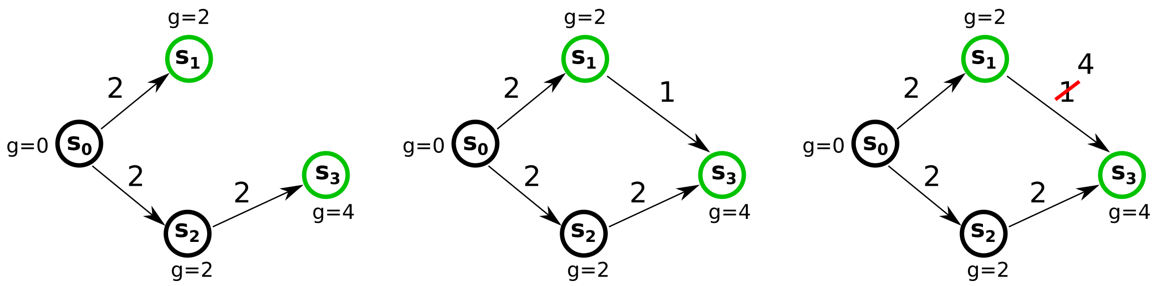


Figure 2.1: This figure illustrates the state independence check in PA*SE.
Left: The states in green are the search frontier, i.e., they are in the open list. For the sake of simplicity, assume that the state-to-goal heuristic value is 0 for all the states. Since $\mathbf{s}_1$ has a smaller g-value than $\mathbf{s}_2$, it should be expanded first. In order to expand $\mathbf{s}_3$ in parallel (i.e. before the expansion of $\mathbf{s}_1$ has been completed), it is necessary and sufficient that there is no path through $\mathbf{s}_1$ that can lower the g-value of $\mathbf{s}_3$.
Middle: In this case, since there exists an edge from $\mathbf{s}_1$ to $\mathbf{s}_3$, a path through $\mathbf{s}_1$ can lower the g-value of $\mathbf{s}_3$. Hence, $\mathbf{s}_3$ cannot be expanded in parallel with $\mathbf{s}_1$.
Right: In this case, since the edge from $\mathbf{s}_1$ to $\mathbf{s}_3$ has a higher cost, a path through $\mathbf{s}_1$ cannot lower the g-value of $\mathbf{s}_3$. Hence, $\mathbf{s}_3$ can be expanded in parallel with $\mathbf{s}_1$. The cost of a path from $\mathbf{s}_1$ and $\mathbf{s}_3$ can be estimated using the pairwise heuristic $h(\mathbf{s}_1, \mathbf{s}_3)$.

(Being Expanded) to store the set of states currently being expanded by one of the threads. It uses a pairwise independence check on states in the open list to find states that are safe to expand in parallel. A state **s** is safe to expand if $g(\mathbf{s})$ is already optimal. In other words, there is no other state that is currently being expanded (in BE), nor in $OPEN$ that can reduce $g(\mathbf{s})$. Formally, a state **s** is defined to be independent of state **s**' iff

$$g(\mathbf{s}) - g(\mathbf{s}') \leq h(\mathbf{s}', \mathbf{s}) \tag{2.1}$$

Fig.2.1 illustrates the state independence check. It can be proved that **s** is independent of states in $OPEN$ that have a larger priority than **s** [20]. However, the independence check has to be performed against the states in $OPEN$ with a smaller priority than **s**, as well as the states that are in BE. Formally, a state **s** is safe to expand if Equations 2.2 and 2.3 hold.

$$\begin{gathered} g(\mathbf{s}) - g(\mathbf{s}') \leq h(\mathbf{s}', \mathbf{s}) \\ \forall \mathbf{s}' \in OPEN \mid f\left(e'\right) < f\left(e\right) \end{gathered} \tag{2.2}$$

$$g(\mathbf{s}) - g(\mathbf{s}') \leq h(\mathbf{s}', \mathbf{s}) \ \forall \mathbf{s}' \in BE \tag{2.3}$$

Chapter 3

Related Work

3.1 Serial Planning Methods in Robotics

There are several approaches that are used to solve the planning problem in robotics, like sampling-based methods, search-based methods and optimization-based methods. Within them, sampling-based and search-based methods construct a graph embedded in the state space of the domain and then find a path from the start to the goal on this graph. The primary difference lies in the way these two categories of approaches explore the state space to construct the graph.

3.1.1 Sampling-Based Methods

Sampling-based algorithms use random state sampling as the means to construct the graph. Probabilistic Roadmap (PRM) [24] and its variants [25, 26] first construct the graph as a preprocessing step by random state sampling. Once the graph is constructed, for a give start-goal query, the shortest path is computed by running a graph search algorithm like Dijkstra or A* on it. PRM is a multi-query algorithm since it does not re-explore the space for every new query, instead uses the same roadmap across queries. RRT [27] and its variants [25, 28] construct a tree incrementally rooted at the start node by random state sampling. RRT is a single-query algorithm since the search tree has to be reconstructed for every new planning query. Since these approaches use random sampling as the means to explore the space, they generate inconsistent solutions for the same problem across runs. However, their inherent stochasticity is beneficial in planning for high-dimensional domains like manipulator planning.

3.1.2 Search-Based Methods

Search-based algorithms like A* and its variants [1, 2, 3] construct the graph by recursively applying a set of actions from every state. Since these methods explore the state space systematically, they generate consistent solutions for the same planning query across runs. Search-based methods are typically used in planning problems that are more complex than motion planning. For example, task and motion planning (TAMP) utilizes complex actions that use motion planners [29] or learned policy models [16, 30] internally to compute trajectories required to execute the task-level actions. The high-level task plan is typically computed using a search-based technique like wA* [18]. Search-based methods are also typically used in low-dimensional planning problems like 3D robot navigation because, in smaller state spaces, it is feasible to get close to the optimal solution. Additionally, the consistency of solutions generated by search methods provides predictable and interpretable behaviors, which is important in navigation. Even in high-dimensional planning problems like manipulator planning, search-based methods are used when consistent solutions are desired [31]. They are also used in kinodynamic planning [32] as the sampling-based planners are kinematic planners, and their extension to kinodynamic planning requires additional machinery like steering functions in the case of kinodynamic RRT [33]. Search-based methods are also used with sampling-based methods like PRM to solve for the shortest path on the constructed roadmap.

3.2 Parallel Planning Algorithms

Parallel planning algorithms seek to make planning faster by leveraging parallelization. They can be categorized into the following three groups.

3.2.1 Parallel Sampling-Based Algorithms

There are a number of approaches that parallelize sampling-based planning algorithms. Probabilistic roadmap (PRM) based methods, in particular, can be trivially parallelized, so much so that they have been described as "embarrassingly parallel" [34]. In these approaches, several parallel processes cooperatively build the roadmap in parallel [35]. Parallelized versions of RRT have also been developed in which multiple cores expand the search tree by sampling and adding multiple new states in parallel [36, 37, 38, 39]. However, in a lot of planning domains involving planning with controllers [40], sampling of states is typically not possible. One such class of planning domains is simulator-in-the-loop planning, which uses an expensive physics simulator to generate successors [18]. Unless the state space is simple such that the sampling distribution can be scripted, there is no principled way to sample meaningful states that can be realized in simulation. Yet another example is planning over higher dimensional image space [41], where the sampling distribution over states has to be explicitly modeled from data or experience.

3.2.2 Parallel Search-Based Algorithms

A trivial approach to achieve parallelization in Weighted A* is to generate successors in parallel when expanding a state. Since the degree of parallelization is limited to the branching factor of the domain, this approach leads to minimal improvement in performance in domains with a low branching factor. Another approach that Parallel A* [21] takes, is to expand states in parallel while allowing re-expansions to account for the fact that states may get expanded before they have the minimal cost from the start state. This leads to a high number of state expansions. There are a number of other approaches that employ different parallelization strategies. In Parallel Retracing A* (PRA*) [42], each processor gets its own open list and a state hashing function is used to map every generated state to a processor. Since it uses synchronous communication, in order to pass states between processors, these lists must be locked. Parallel Structured Duplicate Detection (PSDD) [43] groups states into blocks using a state abstraction function. Processors take entire blocks and expand the constituent states while enduring that neighboring blocks are not expanded in parallel to prevent locking. Parallel Best-NBlock-First (PBNF) [44] integrates ideas from PRA* and PSDD and is extended to handle weighted heuristics and anytime behavior. Hash Distributed A* (HDA*) [45] also employs to state to processor hashing idea from PRA* but uses an asynchronous message passing system to move states between processors, thereby preventing the overhead of locking the transmitting thread.

However, all of these algorithms could potentially expand an exponential number of states, especially when the heuristic is inflated. Heuristic inflation is typically needed in robotics, which is why the family of algorithms that allow state re-expansions is not appropriate in robotics. In contrast, PA*SE [20] parallelly expands states at most once, in such a way that does not affect the bounds on the solution quality. It has been shown to outperform PBNF in robot motion planning. Though PA*SE parallelizes state expansions while preventing re-expansions, as explained earlier, it is not maximally efficient in domains where edge evaluations are expensive since each PA*SE thread sequentially evaluates the outgoing edges of a state being expanded.

3.2.3 Parallel GPU-Based Algorithms

GPUs have a single-instruction-multiple-data (SIMD) execution model, which means that they can only run the same set of instructions on multiple data concurrently. This severely limits the design of planning algorithms in several ways. Firstly, if the goal is to parallelize state expansions, the code for expanding a state must be identical, irrespective of what state is being expanded. Secondly, the set of states must be expanded in a batch. Identifying a batch of states to expand is much more difficult than asynchronously identifying what states can be expanded in parallel. All these limitations also apply if the goal is to parallelize edge evaluations. This is problematic in domains that have complex actions that correspond to forward-simulating dissimilar controllers.

Nevertheless, there has been work on parallelizing A* search on a single GPU [22] or multiple GPUs [23] by utilizing multiple parallel priority queues. Besides the fact that none of these works handle complex action spaces, they have several other limitations. For example, the maximum number of state expansions increases linearly with the degree of parallelization, and these algorithms do not handle heuristic inflation. In contrast to these approaches, in this thesis, we develop algorithms that achieve massive parallelization of edge evaluations on the CPU, which has a multiple-instruction-multiple-data (MIMD) execution model. This allows us the flexibility to efficiently parallelize potentially dissimilar edges, and therefore generalize across all types of planning domains.

3.3 Lazy Search

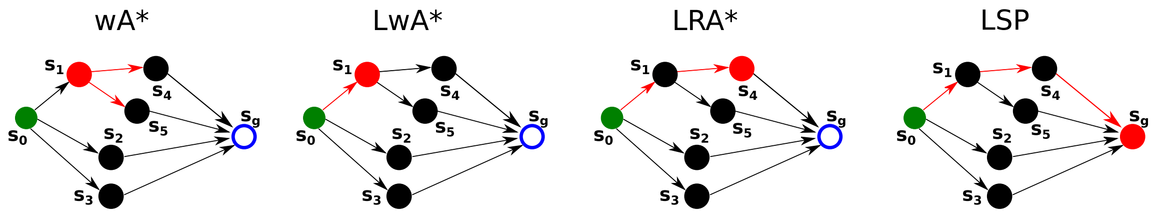


Figure 3.1: In wA*, when the state $\mathbf{s}_1$ is expanded, the outgoing edges shown in red are immediately evaluated. In LwA*, when $\mathbf{s}_1$ is expanded, the incoming edge from $state_0$ is evaluated. In LRA* with a lookahead of 2, when $\mathbf{s}_4$ is expanded, the edges on the best path from the start to $\mathbf{s}_4$ are evaluated. In LSP, when the goal is expanded, the edges on the best path from the start to the goal are evaluated.

Lazy search algorithms achieve greater time efficiency than regular graph search algorithms in domains where the planning time is dominated by edge evaluations. They do so by deferring the evaluation of edges generated during the search and proceeding with the search using cheap-to-compute estimates of the edge costs for the unevaluated edges. Different lazy search algorithms differ in how they toggle between searching the graph and evaluating the edges and the order in which the edges are evaluated. In A*, when a state is expanded, all outgoing edges are immediately evaluated. In Lazy Weighted A* (LWA*) [46] when a state is expanded, the outgoing edges are not immediately evaluated. Instead, the successors are added to the open list with cheap-to-compute underestimates of the true edge costs. When these states are expanded, only the incoming edges that connect their best predecessors are evaluated. In Lazy Shortest Path (LSP) [47], the search proceeds without evaluating any edge until the goal is expanded. It then evaluates the edges that are on the shortest path, updates the costs of these edges, and replans, until a path is found with no unevaluated edges. Lazy Receding Horizon A* (LRA*) [48] allows the search to proceed to an arbitrary lookahead before evaluating edges. In [49], a general framework for lazy search algorithms called Generalized Lazy Search (GLS) was formulated. It was shown that by employing different strategies to toggle between searching the graph and evaluating edges, as well as choosing the order in which the edges are evaluated, various lazy search algorithms can be recovered. GLS also leverages priors on edge validity to come up with more efficient policies that minimize planning time. In [50], ideas from GLS and incremental methods like LPA* were integrated into a lazy lifelong planning algorithm. There has also been work on anytime algorithms that leverage edge existence priors to come up with a strategy to evaluate edges, such that the suboptimality bound on the solution quality is minimized in expectation of the algorithm interruption time while reducing planning time [51]. In [52], the edge selection process for evaluation was formulated as an MDP, and prior experience was used to learn a policy for this MDP.

3.4 Anytime Search

Anytime search algorithms are useful for planning problems where a solution is desired under a limited time budget. They first strive to provide a feasible solution quickly and then attempt to improve it until the time budget expires. A naive approach to make wA* anytime is to sequentially run several iterations of it from scratch while reducing the heuristic inflation. A more elegant anytime algorithm Anytime Repairing A* (ARA*) [19] reuses the previous search tree to prevent redundant work, by keeping track of states whose cost-to-come can be further reduced in future iterations. An alternate approach is to not reuse the previous search tree since it biases subsequent searches towards the previous solution, but reuse only the expensive heuristic computation [53]. Anytime Multi-Heuristic A* (A-MHA*) [11] brings the anytime property to Multi-heuristic A* [3]. Anytime Multi-Resolution Multi-Heuristic A* (AMRA*) [54] is an anytime algorithm that searches over multiple resolutions of the state space. Anytime Multi-Resolution Multi-Heuristic A* (AMRA*) [54] extends AMRA* to handle multiple heuristics. Batch Informed Trees (BIT*) [55] integrates the strengths of search-based and sampling-based planning approaches and has anytime performance. It constructs the graph using batches of state samples that incrementally increase the density of the graph. It then uses incremental graph search techniques of LPA* [56] to search over the graph.

Chapter 4

MPLP: Massively Parallelized Lazy Planning

Lazy search algorithms [47, 48, 51, 57] defer the evaluation of discovered edges and instead use *estimates* of edge costs to search the graph whenever the true edge costs are *unknown*. Here, *knowing* the true edge cost implies running the computation to evaluate the edge. Instead, the estimate is an easier-to-compute approximation of the true edge cost. This makes them more time-efficient in domains where the cost of edge evaluation outweighs the cost of the search. Various lazy search algorithms mainly differ in how they toggle between searching the graph and evaluating the edges. In all of the current lazy search algorithms, performance depends on two critical design choices: 1) the strategy employed to toggle between searching the graph and evaluating the edges, and 2) the order in which edges are evaluated [49]. This is because these algorithms are designed to run as a single process.

Our key insight is that instead of toggling between searching the graph and evaluating the edges, these operations can happen asynchronously in parallel. This allows us to harness the parallelization capabilities of modern processors. In this chapter, we develop a new algorithm: Massively Parallel Lazy Planning (MPLP), that leverages this insight. MPLP eliminates the need for an explicit strategy to balance computational effort between the search and edge evaluations by parallelizing these two operations. On the theoretical front, we show that MPLP provides rigorous guarantees of optimality or bounded suboptimality if heuristics are inflated, as in wA*. MPLP can be used for any planning problem with expensive to evaluate edges and run efficiently on any processor that supports multiprocessing. We show this by evaluating and comparing MPLP against lazy search and parallel search baselines on two planning problems: 1) 3D indoor navigation of a humanoid and 2) a task and motion planning problem of stacking a set of blocks by a robot. The experimental results show that by combining ideas from lazy search and parallel search, MPLP achieves higher time efficiency than existing lazy search algorithms and parallel search algorithms.

4.1 Problem Definition

Let a finite graph $G = (\mathcal{V}, \mathcal{E})$ be defined as a set of vertices $\mathcal{V}$ and edges $\mathcal{E}$. Each vertex $v \in \mathcal{V}$ represents a state $\mathbf{s}$ in the state space of the domain $\mathcal{S}$. An edge $e \in \mathcal{E}$ connecting two vertices v_1 and v_2 in the graph represents an action $\mathbf{a} \in \mathcal{A}$ that takes the agent from corresponding states $\mathbf{s}_1$ to $\mathbf{s}_2$. In this work, we assume that all actions are deterministic. Hence an edge e can be represented as a pair $(\mathbf{s}, \mathbf{a})$, where $\mathbf{s}$ is the state at which action $\mathbf{a}$ is executed. Each edge has an associated true cost $c^t : \mathcal{E} \to [0, \infty]$ which can be computed using a typically expensive edge evaluation routine. A feasible edge is an edge with a finite true cost, i.e. $c^t(e) < \infty$. In addition, there is an optimistic cost associated with each edge $c : \mathcal{E} \to [0, \infty]$ that is easy to compute and underestimates the true cost i.e. $c(e) \leq c^t(e)$. Let $\mathcal{E}^{eval} \subset \mathcal{E}$ be the subset of edges that have been evaluated and hence for which the true costs are available.

A path π is defined by an ordered sequence of edges $(\mathbf{s}, \mathbf{a})_{i=1}^{N}$, the true cost of which is denoted as $c^t(\pi) = \sum_{i=1}^{N} c^t(e_i)$. A feasible path is a path with no infeasible edges, therefore $c^t(\pi) < \infty$. In addition, we define an optimistic cost for a path that is not *fully evaluated* (i.e., not all the edges in the path have been evaluated) using the true cost for the evaluated edges and the optimistic cost otherwise i.e.

$$c(\pi) = \sum_{i=1}^{N} \begin{cases} c^t(e_i), & \text{if } e_i \in \mathcal{E}^{eval} \\ c(e_i), & \text{otherwise} \end{cases}$$

MPLP seeks to find a path π from a given start state $\mathbf{s}_0$ to a goal region $\mathcal{G}$

comprising of only evaluated edges such that the true cost of the path satisfies the relationship $c^t(\pi) \leq \epsilon \cdot c^*$, where c^* is the optimal cost from $\mathbf{s}_0$ to $\mathcal{G}$ and $\epsilon \geq 1$ is suboptimality bound. There is a computational budget of N_t threads available which can run in parallel.

4.2 Method

Typical search-based planning algorithms like A* proceed by expanding states till an (optimal) path to the goal is computed. When a state is expanded, its successors are generated by applying the actions in the action space of the domain, which are represented by edges in a graph. When a successor is generated, the edge between the expanded state and the successor is evaluated for computing the edge cost, which is then used in the search. However, the search slows down dramatically in domains where edge evaluation is expensive. In lazy search methods, edge evaluation is deferred, and an *optimistic* (a fast-to-compute underestimate of the actual cost) estimate of the cost is used instead. The key idea behind MPLP is to run the search using optimistic costs while evaluating relevant edges using a pool of threads entirely asynchronously in parallel. To make this process effective, the research questions are what edges to evaluate, in what order, and how to incorporate these evaluations into the search.

4.2.1 Overview

MPLP runs three key aspects of the search asynchronously in parallel: 1) the optimistic search (a search that uses the true cost for the evaluated edges and the optimistic cost otherwise), 2) edge evaluations, and 3) an evaluation status and suboptimality check on the paths generated by the optimistic search. The suboptimality check is explained in Section 4.3. MPLP allocates the given budget of N_t threads to these aspects as illustrated in Figure 4.1.

The optimistic search runs on thread T_0 as an iterative sequence of wA* searches which proceed without evaluating any edge. Whenever a new edge is discovered, it is added to a priority queue and scheduled for evaluation in order of its priority (higher priority first), and the search proceeds with an optimistic underestimate of

Algorithm 5 MPLP: Search (T_0)

```
1: $G \leftarrow \emptyset$, $\mathcal{A} \leftarrow$ action space , $N_t \leftarrow$ number of threads , $\mathbf{s}_0 \leftarrow$ start state , $\mathcal{G} \leftarrow$ goal region   ▷ Shared variables
2: $E^{open} \leftarrow \emptyset$, $E^{closed} \leftarrow \emptyset$, $E^{eval} \leftarrow \emptyset$, $\Pi \leftarrow \emptyset$, $solution\_found \leftarrow$ False, $terminate \leftarrow$ False
3: procedure MPLP
4:     $path\_exists \leftarrow$ True
5:     Spawn MONITORPATHS on $T_1$ and DELEGATEEDGES on $T_2$
6:     while not $solution\_found$ and $path\_exists$ do
7:         $path\_exists$ = COMPUTEPATH($\mathbf{s}_0, \mathcal{G}$)
8:     end while
9:     $terminate$ = True
10:    return $solution\_found$
11: end procedure
12: procedure COMPUTEPATH($\mathbf{s}_0, \mathcal{G}$)
13:    $\forall \mathbf{s} \in G$, $\mathbf{s}.g \leftarrow \infty$, $\mathbf{s}.is\_closed$ = False   ▷ Reset discovered states
14:    $\mathbf{s}_0.g \leftarrow 0$, $OPEN \leftarrow \emptyset$, $OPEN$.PUSH($\mathbf{s}_0$, KEY($\mathbf{s}_0$))
15:    while $OPEN \neq \emptyset$ do
16:        $\mathbf{s} \leftarrow OPEN$.POP()
17:        if $\mathbf{s} \in \mathcal{G}$ then   ▷ Goal reached
18:            $c^{bound} = \max(c^{bound}, \mathbf{s}.g)$
19:            CONSTRUCTPATH($\mathbf{s}$)
20:            return True
21:        else
22:            EXPAND($\mathbf{s}$)
23:            $\mathbf{s}.is\_closed$ = True
24:        end if
25:    end while
26:    return False
27: end procedure
```

the true edge cost. For edges that have already been evaluated by the edge evaluation threads, the search uses the true cost. Initially, all newly discovered edges have the same priority of 1 for evaluation, in which case the edges in the priority queue follow FIFO ordering. When the search finds a path to the goal, the evaluation priorities of the unevaluated edges in the path are dynamically increased to 2. This is to ensure that the edges that belong to a path to the goal are evaluated before the other edges that are discovered during the search, which depending on the size of the graph and the greediness of the search, can be numerous (see ablation in Section 4.4.1). Though we use this naive priority update in this thesis, more intelligent strategies can potentially be employed.

Another thread T_2 acts as a delegator of edges awaiting evaluation and delegates the edges in the queue to a pool of threads ($T_{i=3:N_t}$) dedicated to edge evaluation.

Algorithm 6 MPLP: Expand and ConstructPath (T_0)

```
 1: procedure ConstructPath($\mathbf{s}$)
 2:     $\pi \leftarrow \emptyset$
 3:     while $\mathbf{s} \neq \mathbf{s}_0$ do
 4:         $(\mathbf{s}^P, \mathbf{a}^P) \leftarrow \mathbf{s}$.GetParent()
 5:         LOCK
 6:         if $(\mathbf{s}^P, \mathbf{a}^P) \in E^{open}$ then
 7:             $E^{open}$.Update($(\mathbf{s}^P, \mathbf{a}^P), 2$)                ▷ Increase priority to 2
 8:         end if
 9:         UNLOCK
10:         $\pi$.Append($(\mathbf{s}^P, \mathbf{a}^P)$)
11:         $\mathbf{s} \leftarrow \mathbf{s}^P$
12:     end while
13:     if $\pi \notin \Pi$ then
14:         $\Pi$.Append($\pi$)
15:     end if
16: end procedure
17: procedure Expand($\mathbf{s}$)
18:     for $\mathbf{a} \in \mathcal{A}$ do
19:         if $(\mathbf{s}, \mathbf{a}) \in E^{open} \cup E^{eval} \cup E^{closed}$ then
20:             $(\mathbf{s}', c) \leftarrow G$.GetSuccessor($\mathbf{s}, \mathbf{a}$)
21:         else
22:             $(\mathbf{s}', c) \leftarrow$ GenerateSuccessor($\mathbf{s}, \mathbf{a}$)
23:             $G$.AddEdge($(\mathbf{s}, \mathbf{a}), c$)
24:             LOCK
25:             $E^{open}$.Push($(\mathbf{s}, \mathbf{a}), 1$)                ▷ Initial priority of 1
26:             UNLOCK
27:         end if
28:         if not $\mathbf{s}'.is\_closed$ and $\mathbf{s}.g + c < \mathbf{s}'.g$ then
29:             $\mathbf{s}'.g = \mathbf{s}.g + c$
30:             $\mathbf{s}'$.SetParent($(\mathbf{s}, \mathbf{a})$)
31:             $OPEN$.Push($\mathbf{s}'$, Key($\mathbf{s}'$))
32:         end if
33:     end for
34: end procedure
35: procedure Key($\mathbf{s}$)
36:     return $\mathbf{s}.g + w \cdot$ GetHeuristic($\mathbf{s}$)
37: end procedure
```

Whenever these threads finish evaluating an edge, they update the graph with the true edge cost. Finally, another thread T_1 monitors the state of every path that has been found by the optimistic search, and when it finds a path that has been fully evaluated and satisfies a suboptimality bound (Theorem 5, Section 4.3), it returns it as the solution which terminates the algorithm. Running this asynchronously allows T_0 to proceed immediately to the next search iteration without waiting for

Algorithm 7 MPLP: Edge Evaluation ($T_2, T_{i=3:N_t}$)

```
1: procedure DelegateEdges
2:     while not $terminate$ do
3:         for $i = 3 : N_t$ do
4:             if $T_i$ is available and $E^{open} \neq \emptyset$ then
5:                 if thread $i$ has not been spawned then
6:                     Spawn EdgeEvaluateThread($i$)
7:                 end if
8:                 LOCK
9:                 $(\mathbf{s}, \mathbf{a}) \leftarrow E^{open}$.Pop()
10:                UNLOCK
11:                $E^{eval}$.Insert($(\mathbf{s}, \mathbf{a})$)
12:                Delegate $(\mathbf{s}, \mathbf{a})$ to thread $i$
13:            end if
14:        end for
15:    end while
16: end procedure
17: procedure EdgeEvaluateThread($i$)
18:    while not $terminate$ do
19:        if thread $i$ has been assigned an edge $(\mathbf{s}, \mathbf{a})$ then
20:            Evaluate $((\mathbf{s}, \mathbf{a}))$
21:        end if
22:    end while
23: end procedure
24: procedure Evaluate($(\mathbf{s}, \mathbf{a})$)
25:    $c^t \leftarrow$ EvaluateEdge($(\mathbf{s}, \mathbf{a})$)
26:    $E^{eval}$.Remove($(\mathbf{s}, \mathbf{a})$)
27:    $E^{closed}$.Insert($(\mathbf{s}, \mathbf{a})$)
28:    if $G$.Cost($(\mathbf{s}, \mathbf{a})$) $\neq c^t$ then
29:        $G$.UpdateEdgeCost($(\mathbf{s}, \mathbf{a}), c^t$)
30:    end if
31: end procedure
```

the generated path to be evaluated and undergo the suboptimality check.

Because of the asynchronous operation of MPLP, the edge evaluation threads $T_{i=3:N_t}$ can update the graph in the middle of an ongoing search on T_0. When any single wA* search on T_0 terminates, the resulting path is a solution on an implicit snapshot of the graph in which the cost of each edge is its cost at the time the source state of the edge was expanded during the search. This may be the true edge cost or the optimistic underestimate depending on whether the edge was evaluated by an edge evaluation thread.

Algorithm 8 MPLP: Monitor Paths (T_1)

```
1: procedure MonitorPaths
2:     while not $terminate$ do
3:         for $\pi \in \Pi$ do
4:             $path\_evaluated \leftarrow$ True, $c^\pi \leftarrow 0$
5:             for $(\mathbf{s}, \mathbf{a}) \in \pi$ do
6:                 if $(\mathbf{s}, \mathbf{a}) \in E^{closed}$ then
7:                     $c^\pi = c^\pi + G.\textsc{Cost}((\mathbf{s}, \mathbf{a}))$
8:                 else
9:                     $path\_evaluated =$ False
10:                    break
11:                end if
12:            end for
13:            if $path\_evaluated$ then
14:                if $c^\pi \le c^{bound}$ then
15:                    $solution\_found =$ True
16:                    return $\pi$
17:                else
18:                    $\Pi.\textsc{Remove}(\pi)$
19:                end if
20:            end if
21:        end for
22:    end while
23: end procedure
```

4.2.2 Details

Besides an open list ($OPEN$) for the states, MPLP uses the following data structures for the edges: A priority queue of edges that need to be evaluated (E^{open}), a list of edges that have been evaluated (E^{closed}) and a list of edges that are under evaluation (E^{eval}). Unlike in $OPEN$ where states with smaller keys are placed in the front of the queue, in E^{open}, edges with higher priorities are placed in front. The optimistic search runs an iterative sequence of weighted A* searches from scratch (ComputePath) in a loop (Line 7, Alg. 5) as a single process on thread T_0. For every state that is expanded for the first time, its successors are generated (Line 22, Alg. 6), but the corresponding edges are not evaluated. Instead, they are added to E^{open}. The search then proceeds with the optimistic edge cost c for the unevaluated edges, but for edges that have already been evaluated, it uses the true cost c^t. We use a constant priority of 1 for all edges when they are first inserted into E^{open}. When a state in the goal region $\mathcal{G}$ is expanded (Line 17, Alg. 5), the path π obtained by backtracking from

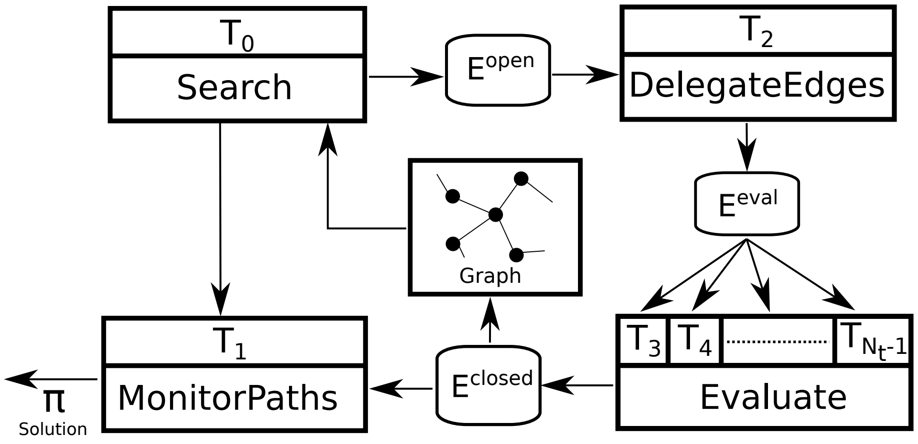


Figure 4.1: The figure depicts a high-level overview of MPLP. The search runs on thread T_0. The discovered edges are added to a priority queue E^{open}. On thread T_2, DelegateEdges delegates the evaluation of edges in E^{open} to a thread from a pool of threads ($T_{i=3:N_t}$) dedicated to edge evaluation. The edges under evaluation are moved from E^{open} to a list E^{eval}. When an edge has been evaluated by Evaluate, it is added to a list E^{closed} and the graph is updated to reflect the true edge cost. MonitorPaths monitors the generated paths in thread T_1 and waits for a fully evaluated path (all edges in E^{closed}) that satisfies a suboptimality check. Upon finding such a path, it returns it as the solution, and the algorithm is terminated.

the goal state to $\mathbf{s}_0$ is added to a list of generated paths Π in ConstructPath. The priorities of unevaluated edges in π are increased by a multiplicative factor of 2 in E^{open} to prioritize the evaluation of the edges in the paths (Line 7, Alg. 6). In addition, the maximum of the costs of the paths generated by ComputePath is stored in a variable c^{bound} (Line 18, Alg. 5). As explained in Section 4.3, c^{bound} is upper bounded by $w \cdot c^*$, where w is the heuristic inflation factor and c^* is the cost of an optimal path in G.

In a separate thread T_2, DelegateEdges (Alg. 7) delegates the evaluation of edges in E^{open} in order of their priorities to a pool of threads ($T_{i=3:N_t}$) dedicated to edge evaluations (Line 12). When a thread is available, the edge is evaluated in Line 25 to obtain the true cost c^t. When an edge is being evaluated, it is moved from E^{open} to E^{eval}. Once it has been evaluated, it is moved to E^{closed}. If the true cost is different from the estimated cost, the graph is updated (Line 29). A generated edge, therefore, belongs to one of the three containers, i.e., E^{open}, E^{eval} or E^{closed}, at any point in time. Therefore, if a state is revisited, the state along with its incoming edge need not be regenerated (Line 20, Alg. 6).

Another asynchronous process MonitorPaths (Alg. 8) on thread T_2 monitors the state of every edge in the paths in Π. If a path is found that has been fully evaluated and that has cost no greater than c^{bound}, it is returned as the solution. This check is necessary to guarantee bounded suboptimality as proved in Section 4.3. The optimistic search terminates when either MonitorPaths finds a solution and sets the variable *solution_found* (Line 15, Alg. 8) or when ComputePath terminates without a path by exhausting the open list (*path_exists* is false). An implementation detail to note is that E^{open} is modified by multiple threads asynchronously. Therefore to ensure thread safety by protecting against data race, E^{open} must be accessed under a synchronization lock.

4.2.3 Demo

We will run through a simple demo of MPLP on the graph shown in Figure 4.2a with $N_t = 6$. The state and goal states are shown in green and blue, respectively. Let e_i^j refer to an edge from state $\mathbf{s}_i$ to $\mathbf{s}_j$. For the sake of simplicity, we will assume that the edge evaluation either results in a valid edge with the same cost as the optimistic estimate or results in an invalid edge with an infinite cost. The search proceeds as follows.

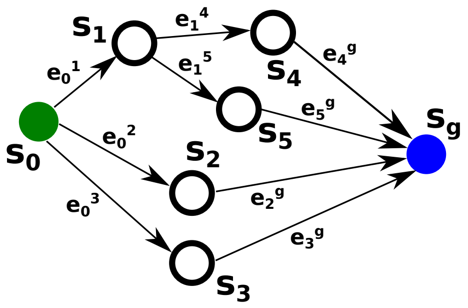
1. Fig 4.2b: The first optimistic search on T_0 returns the path $[e_0^1, e_1^4, e_4^g]$ shown in dotted blue. The edges in the optimistic path, along with the other edges discovered during the search (in this case e_1^5), are added to E^{open}. Since they all have the same initial priority, they get placed in E^{open} in the order they were discovered. The optimistic path is added to Π, and MonitorPaths will keep a watch on its evaluation status, allowing T_0 to proceed to the next search.
2. Fig 4.2c: The priorities of edges in the optimistic path i.e. $[e_0^1, e_1^4, e_4^g]$ are increased in E^{open}.
3. Fig 4.2d: DelegateEdges delegates the edges in E^{open}, in order of their priorities to a pool of threads allocated to edge evaluations. In this case, three threads are available for edge evaluations; therefore, only $[e_0^1, e_1^4, e_4^g]$ get delegated for evaluation.

4. Fig 4.2e: The graph is updated to account for the outcome of the edge evaluations, i.e., $[e_0^1, e_1^4]$ are valid edges whereas e_4^g is invalid. MonitorPaths removes the first optimistic path from Π since it has been invalidated. In parallel, the optimistic search returns another path $[e_0^2, e_2^g]$ and the discovered edges in the second search are added to E^{open}, and the priorities of $[e_0^2, e_2^g]$ are inflated. The path is added to Π.
5. Fig 4.2f: The edges $[e_0^2, e_2^g, e_1^5]$ in E^{open} are delegated to edge evaluation threads.
6. Fig 4.2g: $[e_0^2, e_2^g]$ are valid edges, whereas e_1^5 is invalid. At the same time, MonitorPaths finds that the path $[e_0^2, e_2^g]$ has been evaluated.
7. Fig 4.2h: Assuming that the path $[e_0^2, e_2^g]$ satisfies the suboptimality check, it is returned as the solution.

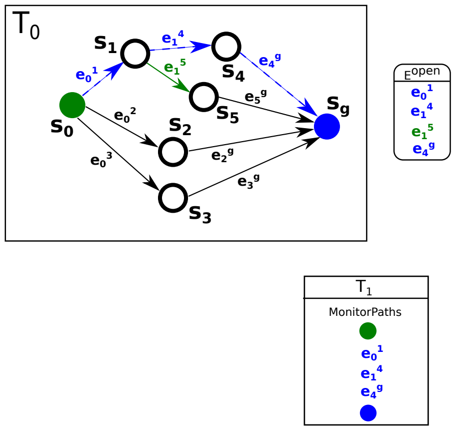
4.2.4 Discussion

MPLP has some key differences from other lazy search algorithms.

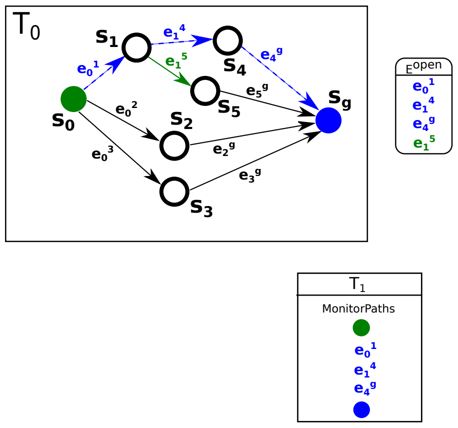
- The search and edge evaluations run completely asynchronously. Unlike in the GLS framework, there is no explicit strategy employed to toggle between the search and edge evaluations.
- All other lazy search algorithms like LwA*, LSP and LRA* only evaluate edges that are either on the shortest path to the goal or likely to be so. This is to ensure that computational effort is not wasted on evaluating edges that are not likely to be on the shortest path. MPLP, on the other hand, evaluates every edge that the search encounters while prioritizing edges that are on the shortest paths in the partially evaluated graphs. This allows it to exploit massive parallelization.



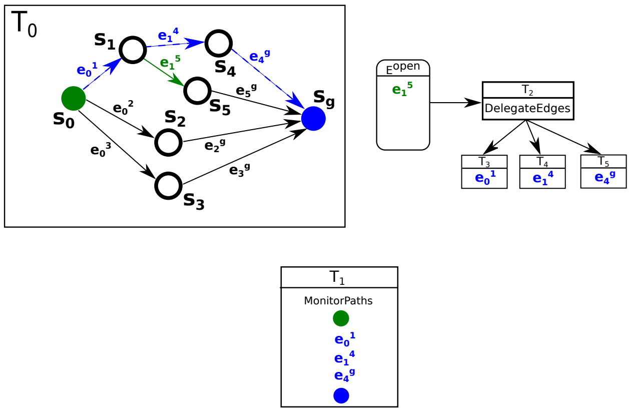
(a) Graph with the start state $\mathbf{s}_0$ shown in green and the goal state $\mathbf{s}_g$ shown in blue.



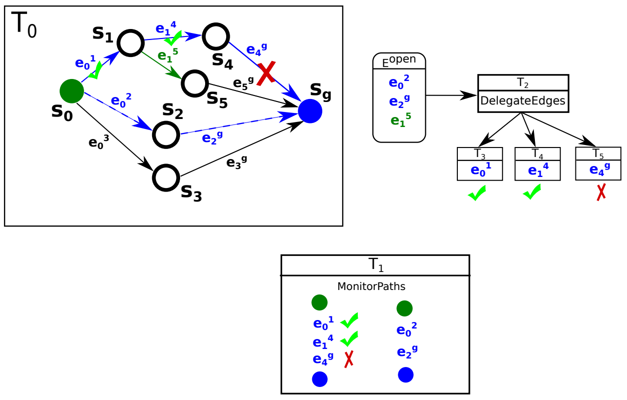
(b) The optimistic search returns the first path with edges $[e_0^1, e_1^4, e_4^g]$ shown in dotted blue. All discovered edges during the search i.e. $[e_0^1, e_1^4, e_4^g, e_1^5]$ are added to E^{open}.



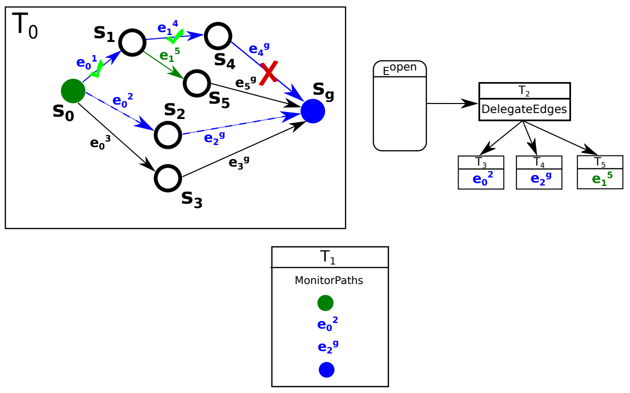
(c) The priorities of edges in the the optimistic path i.e. $[e_0^1, e_1^4, e_4^g]$ are increased in E^{open}.



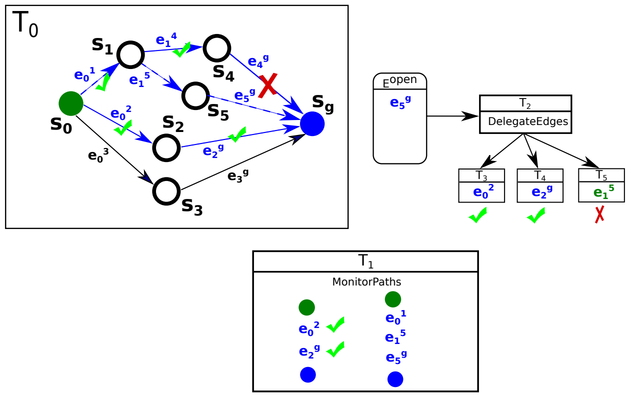
(d) DelegateEdges delegates the edges in E^{open}, in order of their priorities, to a pool of threads allocated to edge evaluations.



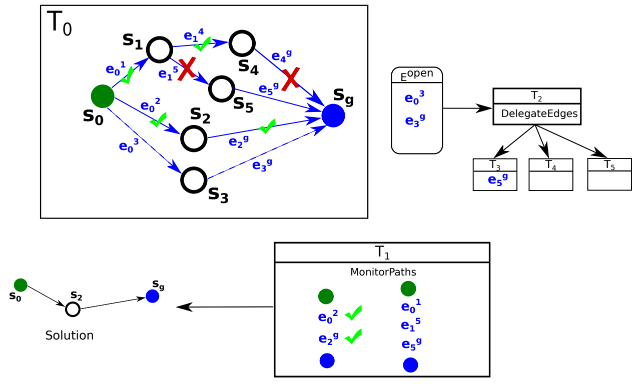
(e) The graph is updated to account for the outcome of the edge evaluations. Here $[e_0^1, e_1^4]$ are valid edges are e_4^g is invalid. In parallel, the optimistic search returns another path $[e_0^2, e_2^g]$ and the discovered edges in the second search are added to E^{open}, and the priorities of $[e_0^2, e_2^g]$ are inflated.



(f) $[e_0^2, e_2^g, e_1^5]$ are delegated to edge evaluation threads.



(g) $[e_0^2, e_2^g]$ are valid edges, whereas e_1^5 is invalid.



(h) MonitorPath finds that the path $[e_0^2, e_2^g]$ is valid, and assuming it satisfies the suboptimality check, it returns it as the solution.

Figure 4.2: Demo of MPLP.

4.3 Properties

MPLP is guaranteed to be complete and bounded suboptimal, and we prove these properties.

Lemma 1 *If there exists a path π in Π that is fully evaluated and satisfies the suboptimality bound (Line 14, Alg. 8), MONITORPATHS will return a solution in finite time.*

Proof Since Alg. 5 runs wA*, the paths computed by it are cycle-free. For a finite graph G, there are a finite number of cycle-free paths (possibly with a mix of evaluated and unevaluated edges) from $\mathbf{s}_0$ to $\mathcal{G}$ with a cost lower than a finite upper bound (Lemma 3). Moreover, the uniqueness check in Line 13 of Alg. 6 ensures that there are no duplicate paths in Π. Therefore Π is of finite size, and since MONITORPATHS iterates over Π repeatedly, it is bound to discover a path π in Π that is fully evaluated and satisfies the suboptimality bound in finite time if such a path exists in Π.

Theorem 2 (Completeness) *If there exists at least one feasible path π in G from $\mathbf{s}_0$ to $\mathcal{G}$, MPLP will return a solution in finite time.*

Proof MPLP runs a sequence of weighted A* searches on a finite graph G, which is a complete algorithm. The edges in G are being simultaneously evaluated by EVALUATEEDGES. In the worst case, the optimistic search will have discovered all edges in G and added them to E^{open} (Line 25, Alg. 6). Therefore EVALUATEEDGES will eventually evaluate all edges in G, in which case COMPUTEPATH will have access to the true costs of all edges and will add a feasible path to Π if such a path exists. Lemma 1 guarantees that the path will then be returned as the solution by MONITORPATHS in finite time.

Lemma 3 *The cost $c(\pi_i)$ of a path π_i computed by COMPUTEPATH in any iteration i of MPLP (Line 7, Alg. 5) satisfies the relationship $c(\pi_i) \leq w \cdot c^*$, where c^* is the cost of the optimal path in G.*

Proof At any iteration i of MPLP, the search runs on an implicit snapshot of G in which the true costs of some of the edges are known, and the remaining edges have an estimated cost which is an underestimate of the true cost. Let this intermediate graph be G_i. Let the cost of an optimal path in G be c^*, and the cost of the same path in G_i be c_i. Since the cost of the unevaluated edges in G_i are an underestimate of the corresponding edges in G, this implies that $c_i \leq c^*$. Let the cost of an optimal path in G_i be c_i^*. Therefore,

$$\implies c_i^* \leq c_i \leq c^*$$

Since ComputePath runs weighted A* on G_i, the cost of any path π_i computed by it in any iteration i satisfies $c(\pi_i) \leq w \cdot c_i^*$. Therefore,

$$\implies c(\pi_i) \leq w \cdot c_i^* \leq w \cdot c_i \leq w \cdot c^*$$

Theorem 4 (Soundness) *The path returned by MPLP is fully evaluated and feasible.*

Proof MonitorPaths only returns a fully evaluated path that has a true cost no greater than c^{bound} (Line 14, Alg. 8). For it to return an infeasible path, c^{bound} has to be ∞. However, since c^{bound} is initialized to $-\infty$ and gets updated when ComputePath finds a path (Line 18, Alg. 5), for it to have a value of ∞, ComputePath has to find a path with ∞ cost. This is not possible because Lemma 3 states that any path returned by ComputePath has a finite upper bound if the graph has a feasible solution.

Theorem 5 (Bounded suboptimality) *The path π returned as the solution by MPLP satisfies $c^t(\pi) \leq w \cdot c^*$ where c^* is the cost of the optimal path in G.*

Proof As per Lemma 3, a path π_i returned by ComputePath in any iteration i of MPLP will never have a cost greater than $w \cdot c^*$.

$$\implies \max_i c(\pi_i) = c^{bound} \leq \; w \cdot c^*$$

If a fully evaluated path π is found in MonitorPaths such that $c^t(\pi) \leq c^{bound}$ then $c^t(\pi) \leq w \cdot c^*$.

4.4 Evaluation

We evaluate MPLP in two planning domains where edge evaluation is expensive. To emphasize the computational budget, we append the number of threads (N_t) being used by MPLP as a suffix, i.e., MPLP-N_t. The proposed algorithm, along with the baselines, was implemented in C++.

4.4.1 3D navigation

The first domain is motion planning for 3D (x, y, θ) navigation of a PR2 robot in an indoor environment similar to the one used in [51] and shown in Figure 4.3. The robot can move along 18 simple motion primitives that independently change the three state coordinates by incremental amounts. Evaluating each primitive involves collision checking of the robot model (approximated as spheres) against the world model (represented as a 3D voxel grid) at interpolated states on the primitive. Though approximating the robot with spheres instead of meshes dramatically speeds up collision checking, it is still the most expensive component of the search. The computational cost of edge evaluation increases with the increasing granularity of interpolated states at which collision checking is carried out. We use two types of primitives: 1) primitives that change the (x, y) coordinates but do not change θ and 2) primitives that only change θ. We vary the computational cost of edge evaluations by varying the distance (d_{cc}) between two consecutive states along a primitive at which collision checking is carried out (i.e., the discretization of the primitives for collision checking) for the first type of primitives. For the second type of primitives, collision checking is always carried out at 1° increments in θ, and this parameter is not varied for the sake of simplicity. In general, a smaller d_{cc} produces a better approximation, while larger d_{cc} can cause the robot to tunnel through obstacles. In the optimistic approximation of the primitives, we collision check just the final state along the primitive. In this domain, this approximation is incorrect about 24% of the time. The search uses Euclidean distance as the admissible heuristic. The experiments were run on an Amazon Web Services (AWS) c5a.24xlarge instance with 96 vCPUs, running Ubuntu 18.04. We evaluate 50 trials, in each of which the start configuration of the robot and goal region are sampled randomly.

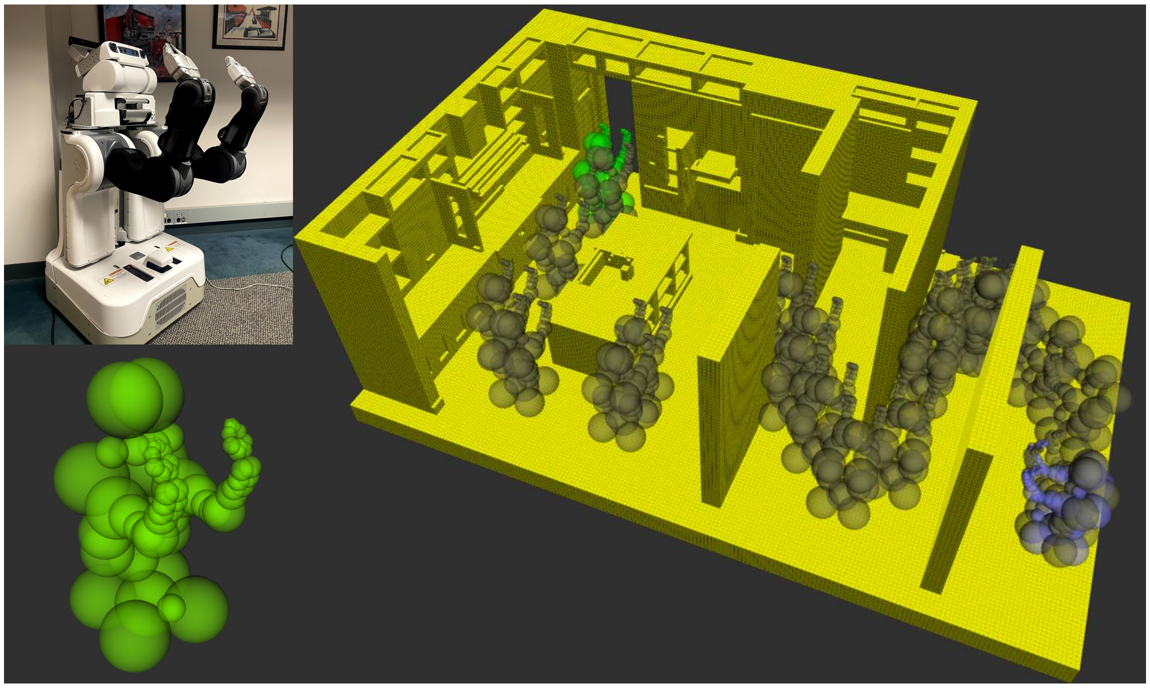


Figure 4.3: (**Navigation**) Left: The PR2's collision model is approximated with spheres. Right: The task is to navigate in an indoor map from a given start (purple) and goal (green) states using a set of motion primitives. States at the end of every primitive in the generated plan are shown in black.

Comparison to lazy search baselines

We compare MPLP-90 ($N_t = 90$) with weighted A* and lazy search baselines LwA* [46] and LSP [47] (which are instantiations of GLS [49]). Lazy search algorithms are designed to increase efficiency in domains where edge evaluations are expensive. Therefore we analyze the performance gain achieved by MPLP with increasing computational cost of edge evaluations by reducing d_{cc}. Figure 4.4 shows the mean speedup achieved by MPLP and the baselines over wA* for varying d_{cc} on a set of start and goal pairs with uninflated and inflated heuristics. Speedup over wA* is defined as the ratio of the mean runtime of wA* over the mean runtime of a specific algorithm. Table 4.1 shows the corresponding raw data. With increasing granularity of collision checking (decreasing d_{cc}), the speedup achieved by MPLP drastically outpaces that of the baselines.

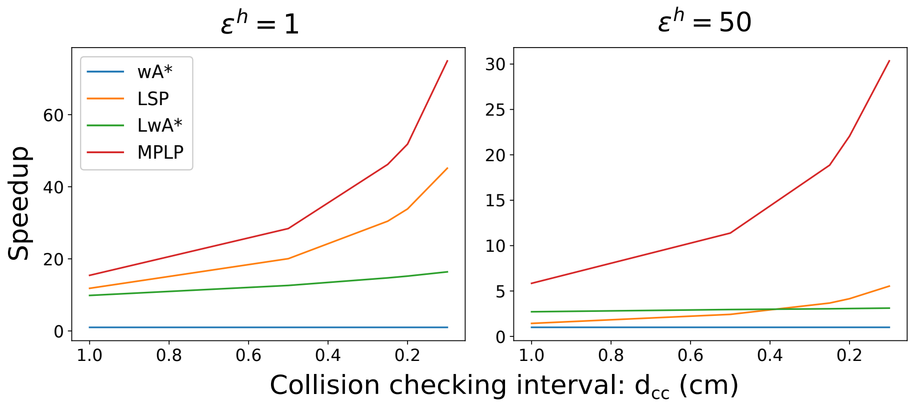


Figure 4.4: (**Navigation**) Mean speedup achieved by MPLP, LSP and LwA* over wA* with uninflated heuristic (left) and with a heuristic inflation factor of 50 (right). d_{cc} decreases along the x-axis, which increases edge evaluation time.

	Collision checking interval: d_{cc} (cm)				
	1	0.5	0.25	0.2	0.1
	$w = 1$				
wA*	3.55	6.82	13.40	16.59	32.96
LwA*	0.36	0.54	0.91	1.09	2.01
LSP	0.30	0.34	0.44	0.49	0.73
MPLP-90	**0.23**	**0.24**	**0.29**	**0.32**	**0.44**
	$w = 50$				
wA*	0.76	1.48	2.83	3.53	6.98
LwA*	0.28	0.50	0.93	1.15	2.24
LSP	0.53	0.61	0.77	0.85	1.26
MPLP-90	**0.13**	**0.13**	**0.15**	**0.16**	**0.23**

Table 4.1: (**Navigation**) Mean planning times (s) for MPLP, wA* and lazy search baselines for varying d_{cc}, with and without heuristic inflation.

Comparison to parallel search baselines

We also compare MPLP with parallel search baselines. The first baseline is a variant of weighted A* in which, during a state expansion, the successors of the state are generated, and the corresponding edges are evaluated in parallel. For lack of a better term, we call this baseline Parallel Weighted A* (PwA*). Note that this is very different from the Parallel A* (PA*) algorithm [21]. The second baseline is PA*SE [20]. These two baselines leverage parallelization differently. In PwA*, parallelization is at the level of generation of successors, whereas in PA*SE, parallelization is at the level of state expansions. Figure 4.5 shows the mean speedup achieved by MPLP and the baselines over wA* for various thread budgets, with uninflated and inflated heuristics and with two different values of d_{cc}. The corresponding raw data is shown in Table 4.2. For a single thread, PwA* and PA*SE have the same runtime as that of wA*. As described in Section 4.2, MPLP needs a minimum of 4 threads. Since PwA* parallelizes successor generation during an expansion, increasing the number of threads beyond a certain point does not lead to any performance improvement. The maximum speedup achieved by PA*SE is dependent on the number of states that can be safely expanded in parallel. The performance degrades with a higher number of threads, consistent with what was observed in [20]. Consistent with what was observed in comparison with lazy search baselines, the speedup obtained by MPLP is greater for a smaller d_{cc} (larger edge evaluation computational cost). In addition, MPLP's performance gain saturates at a higher N_t for a smaller d_{cc}. This shows that MPLP leverages multithreading more effectively with increasing computational cost of edge evaluations and is, therefore, more efficient than the baselines.

MPLP is effective in domains where the computational cost of evaluating edges relatively outweighs that of exploring the graph optimistically. This implies smaller graphs with expensive to evaluate edges. As is the case with all lazy search algorithms, in domains with larger graphs and inexpensive edges, MPLP is not effective. This can be observed in the top two plots in Figure 4.5. With $d_{cc} = 1$cm, PA*SE outperforms MPLP. However, with more expensive edges, as is the case in the bottom two plots, MPLP comprehensively outperforms PA*SE.

Ablation of priority inflation of edges in optimistic paths

As discussed earlier, to prioritize the evaluation of edges that are in the paths computed by the optimistic search over the other discovered edges, their priorities in E^{open} are dynamically increased (Line 7, Alg. 6). We ablate this to highlight the benefit of doing so and paying the small cost of rebalancing the priority queue. Figure 4.6 shows the mean planning times of MPLP with and without the priority inflation, for

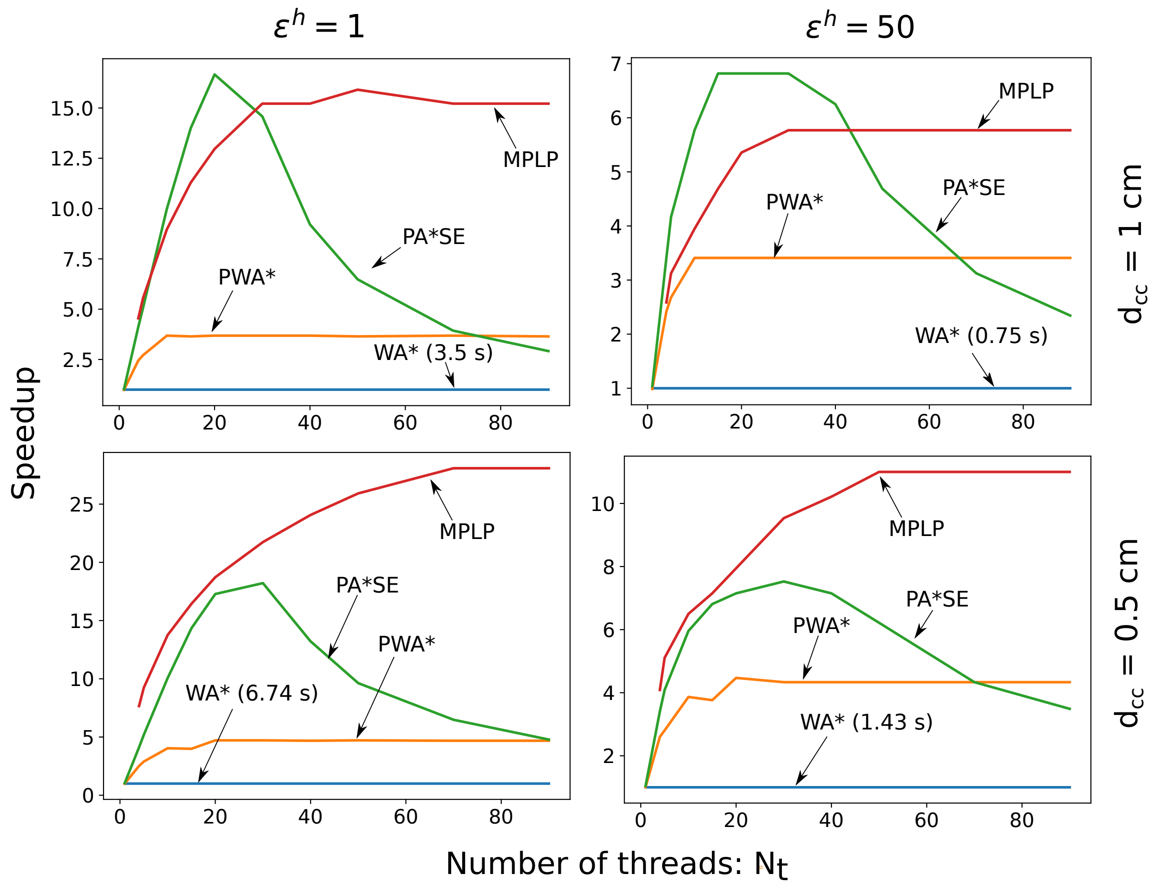


Figure 4.5: (**Navigation**) Mean speedup achieved by MPLP, PwA* and PA*SE over wA* with uninflated heuristic (left) and with heuristic inflation of 50 (right). For a single thread, PwA* and PA*SE have the same runtime as that of wA*. With an increasing number of threads, the performance of MPLP and the baselines improves over wA*. However, achieves a significantly higher speedup as compared to the baselines when $d_{cc} = 0.5$cm.

varying threads, with and without heuristic inflation. For smaller N_t, priority inflation significantly reduces planning time. With increasing N_t, the performance gain diminishes since there is enough computational resource available to evaluate all edges in E^{open} in parallel without having to preferentially evaluate edges in the optimistic paths. At the same time, with heuristic inflation, the performance gain of priority inflation is also lower. This is because the search is greedy and discovers fewer edges that need to be evaluated.

	Number of threads (N_t)										
	1	4	5	10	15	20	30	40	50	70	90
	$d_{cc} = 1\text{cm} \mid w = 1$										
wA*	3.5	-	-	-	-	-	-	-	-	-	-
PwA*	3.54	1.43	1.29	0.95	0.96	0.95	0.95	0.95	0.96	0.95	0.96
wPA*SE	3.40	0.85	0.69	0.35	0.25	0.21	0.24	0.38	0.54	0.89	1.20
MPLP	-	**0.77**	**0.63**	**0.39**	**0.31**	**0.27**	**0.23**	**0.23**	**0.22**	**0.23**	**0.23**
	$d_{cc} = 1\text{cm} \mid w = 50$										
wA*	0.75	-	-	-	-	-	-	-	-	-	-
PwA*	0.76	0.31	0.28	0.22	0.22	0.22	0.22	0.22	0.22	0.22	0.22
wPA*SE	0.72	0.22	0.18	0.13	0.11	0.11	0.11	0.12	0.16	0.24	0.32
MPLP	-	**0.29**	**0.24**	**0.19**	**0.16**	**0.14**	**0.13**	**0.13**	**0.13**	**0.13**	**0.13**
	$d_{cc} = 0.5\text{cm} \mid w = 1$										
WA*	6.74	-	-	-	-	-	-	-	-	-	-
PwA*	6.76	2.68	2.33	1.67	1.69	1.43	1.43	1.44	1.43	1.44	1.44
PA*SE	6.44	1.64	1.30	0.67	0.47	0.39	0.37	0.51	0.70	1.04	1.41
MPLP	-	**0.88**	**0.73**	**0.49**	**0.41**	**0.36**	**0.31**	**0.28**	**0.26**	**0.24**	**0.24**
	$d_{cc} = 0.5\text{cm} \mid w = 50$										
WA*	1.43	-	-	-	-	-	-	-	-	-	-
PwA*	1.42	0.55	0.51	0.37	0.38	0.32	0.33	0.33	0.33	0.33	0.33
PA*SE	1.37	0.42	0.35	0.24	0.21	0.20	0.19	0.20	0.23	0.33	0.41
MPLP	-	**0.35**	**0.28**	**0.22**	**0.20**	**0.18**	**0.15**	**0.14**	**0.13**	**0.13**	**0.13**

Table 4.2: (**Navigation**) Mean planning times (s) for MPLP, wA* and parallel search baselines (PwA* and PA*SE) for varying N_t, with different values of d_{cc} and w.

4.4.2 Assembly task

	wA*	LwA*	LSP	MPLP-40
Time (s)	496	259	133	**84**
Speedup	1	1.9	3.7	**5.9**

Table 4.3: (**Assembly**) Mean planning time and speedup over wA* of MPLP compared to those of the lazy search baselines.

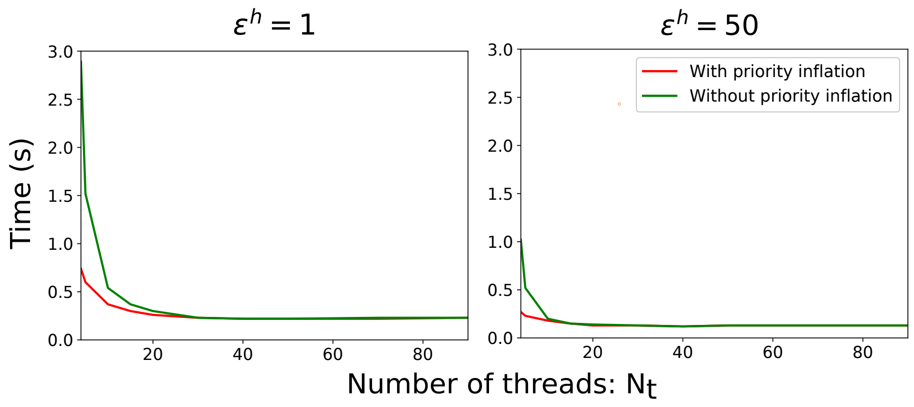


Figure 4.6: (**Navigation**) Mean planning time (s) of MPLP with and without edge priority inflation for a varying number of threads.

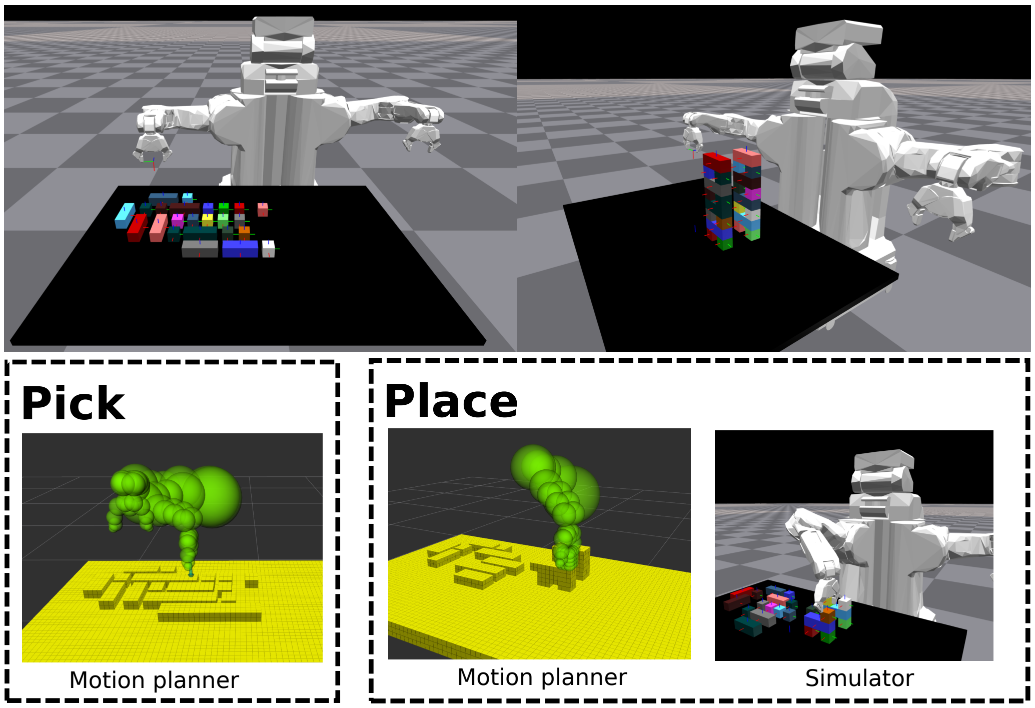


Figure 4.7: (**Assembly**) Top: The PR2 has to arrange a set of blocks on the table (left) into a given configuration (right). Bottom: It is equipped with Pick and Place controllers. The Pick controller uses the motion planner to reach a block. The Place controller uses the motion planner to place a block and simulates the outcome of releasing the block.

The second domain is a task and motion planning problem of assembling a set of blocks on a table into a given structure by a PR2, as shown in Figure 4.7. We assume full state observability of the 6D poses of the blocks and the robot's joint configuration. The goal is defined by the 6D poses of each block in the desired structure. The PR2 is equipped with Pick and Place controllers, which are used as macro-actions in high-level planning. Both of these actions use a motion planner internally to compute collision-free trajectories in the workspace. Additionally, Place has access to a simulator (NVIDIA Isaac Gym [58]) to simulate the outcome of placing a block in its desired pose. For example, if the planner tries to place a block in its final pose but has not placed the block underneath yet, the placed block will not be supported, and the structure will not be stable. This would lead to an invalid successor during planning. We set a simulation timeout of $t_s = 0.2$ s to evaluate the outcome of placing a block. Considering the variability in the simulation speed and the overhead of communicating with the simulator, this results in a total wall time of less than 1 s for the simulation. The motion planner has a timeout of $t_p = 60$ s based on the wall time; therefore, that is the maximum time the motion planning can take. Since the workspace is cluttered, the bottleneck in this domain is the motion planning component of these actions. In the optimistic approximation, these macro-actions are replaced by their approximate versions, which substitute the motion planner with an IK solver, while the motion planner is used when the corresponding edges are evaluated. Successful Pick and Place actions have unit real and optimistic costs and infinite otherwise. A Pick action on a block is successful if the motion planner finds a feasible trajectory to reach the block within t_p. A Place action on a block is successful if the motion planner finds a feasible trajectory to place the block within t_p, and simulating the block placement results in the block coming to rest at the desired pose within t_s. The cost of every feasible plan is twice the number of blocks since placing a block in its desired pose involves a single Pick and Place controller pair, each of which has a unit cost.

The experiments were run on an AWS g4dn.16xlarge instance with 64 vCPUs, running Ubuntu 18.04. The instance also has an NVIDIA T4 Tensor Core GPU to run the simulator. MPLP is run with $N_t = 40$, and the addition of more threads did not improve performance in this domain. The number of blocks not in their final desired pose is used as the admissible heuristic, with an inflation factor of 5. Table 4.3 shows planning times and speedup over wA* of MPLP-40 compared to the lazy search baselines. The numbers are averaged across 20 trials, in each of which the blocks are arranged in random order on the table. MPLP-40 achieves a 5.9x speedup over wA* and a 1.6x speedup over LSP.

4.5 Conclusion

In this work, we presented MPLP, a massively parallelized lazy search algorithm that integrates ideas from lazy search and parallel search. We proved that MPLP is sound, complete, and bounded suboptimal. Our experiments showed that MPLP achieves higher efficiency than both lazy search and parallel search algorithms on two very different planning domains.

MPLP assigns a uniform evaluation priority to edges when they are first discovered and only increases the priorities of edges that belong to a path. However, if there is a domain-dependent edge-existence prior available, that can be seamlessly integrated into the algorithm by assigning the evaluation priority derived from it. Instead of a naive implementation of LSP, where each shortest path search is run from scratch, an incremental approach of updating the graph and re-using the previous search tree using LPA* mechanics can be more efficient [49, 50]. However, because of the massive parallelization of edge evaluations in MPLP, a potentially large number of edges get updated in-between searches. Therefore running the search in each iteration from scratch is more efficient than incremental methods. However, based on the number of updated edges, an intelligent strategy can be employed to either re-use the previous search tree or plan from scratch.

Chapter 5

ePA*SE: Edge-Based Parallel A* for Slow Evaluations

MPLP achieves faster planning by running the search and evaluating edges asynchronously in parallel. Just like all lazy search algorithms, it assumes that successor states can be generated without evaluating edges, which allows the algorithm to defer edge evaluations and lazily proceed with the search. However, this assumption doesn't hold for a number of planning domains in robotics. In particular, consider planning problems that use a high-fidelity physics simulator to evaluate actions involving object-object and object-robot interactions [18]. The generation of successor states is typically not possible without an expensive simulator call. In such domains, where edge evaluation cannot be deferred, MPLP is not applicable. Instead, what is required is a parallel search algorithm that interleaves the search with the parallel evaluation of edges.

In Chapter 2, we described PA*SE, which speeds up search by parallelizing expansions of independent states. Though PA*SE parallelizes state expansions, for a given state, the successors are generated sequentially. This is not the most efficient strategy, especially for domains with large branching factors. In addition, this strategy is particularly inefficient in domains where there is a large variance in the edge evaluation times. Consider a state with several outgoing edges that have to be expanded, such that the first edge is expensive to evaluate, while the others are relatively inexpensive. In this case, since a single thread is evaluating all of the edges in sequence, the evaluations of the cheap edges will be held up by the one expensive edge. This often happens in planning for robotics. Consider full-body planning for a humanoid. Evaluating a primitive that moves just the wrist joint of the robot requires collision checking of just the wrist. However, evaluating the primitive that moves the base of the robot requires fully-body collision checking of the entire robot. One way to avoid this would be to evaluate the outgoing edges in parallel, which PA*SE doesn't do. However, this seemingly trivial modification does not solve another cause of inefficiency in PA*SE i.e., the evaluation of the outgoing edges from a given state is tightly coupled with the expansion of the state. This leads to more edges being evaluated than is necessary, as we will show in our experiments.

Therefore in this chapter, we develop an optimal parallel search algorithm, ePA*SE (Edge-based Parallel A* for Slow Evaluations), that eliminates these inefficiencies by parallelizing edge evaluations instead of state expansions. ePA*SE exploits the insight that the root cause of slow expansions is typically slow edge evaluations. Each ePA*SE thread is responsible for evaluating a single edge instead of expanding a state and evaluating all outgoing edges from it, all in a single thread, like in PA*SE. This makes ePA*SE significantly more efficient than PA*SE and we show this by evaluating it on two different planning domains: 1) 3D indoor navigation of a mobile manipulator and 2) a task and motion planning problem of stacking a set of blocks by a dual-arm robot.

5.1 Problem Definition

Let a finite graph $G = (\mathcal{V}, \mathcal{E})$ be defined as a set of vertices $\mathcal{V}$ and directed edges $\mathcal{E}$. Each vertex $v \in \mathcal{V}$ represents a state $\mathbf{s}$ in the state space of the domain $\mathcal{S}$. An edge $e \in \mathcal{E}$ connecting two vertices v_1 and v_2 in the graph represents an action $\mathbf{a} \in \mathcal{A}$ that takes the agent from corresponding states $\mathbf{s}_1$ to $\mathbf{s}_2$. In this work, we assume that all actions are deterministic. Hence an edge e can be represented as a pair $(\mathbf{s}, \mathbf{a})$, where $\mathbf{s}$ is the state at which action $\mathbf{a}$ is executed. For an edge e, we will refer to the corresponding state and action as $e.\mathbf{s}$ and $e.\mathbf{a}$ respectively. In addition, we will use the following notations:

- $\mathbf{s}_0$ is the start state and $\mathcal{G}$ is the goal region.

- $c : \mathcal{E} \rightarrow [0, \infty]$ is the cost associated with an edge.
- $g(\mathbf{s})$ or g-value is the cost of the best path to $\mathbf{s}$ from $\mathbf{s}_0$ found by the algorithm so far.
- $h(\mathbf{s})$ is a consistent and therefore admissible heuristic [17]. It never overestimates the cost to the goal.

A path π is defined by an ordered sequence of edges $e_{i=1}^{N} = (\mathbf{s}, \mathbf{a})_{i=1}^{N}$, the cost of which is denoted as $c(\pi) = \sum_{i=1}^{N} c(e_i)$. The objective is to find a path π from $\mathbf{s}_0$ to a state in the goal region $\mathcal{G}$ with the optimal cost c^*. There is a computational budget of N_t threads available, which can run in parallel. Similar to PA*SE, we assume there exists a pairwise heuristic function $h(\mathbf{s}, \mathbf{s}')$ that provides an estimate of the cost between any pair of states. It is forward-backward consistent i.e. $h(\mathbf{s}, \mathbf{s}'') \leq h(\mathbf{s}, \mathbf{s}') + h(\mathbf{s}', \mathbf{s}'')\ \forall\ \mathbf{s}, \mathbf{s}', \mathbf{s}''$ and $h(\mathbf{s}, \mathbf{s}') \leq c^*(\mathbf{s}, \mathbf{s}')\ \forall\ \mathbf{s}, \mathbf{s}'$. Note that using h for both the unary heuristic $h(\mathbf{s})$ and the pairwise heuristic $h(\mathbf{s}, \mathbf{s}')$ is a slight abuse of notation since these are different functions.

5.2 Method

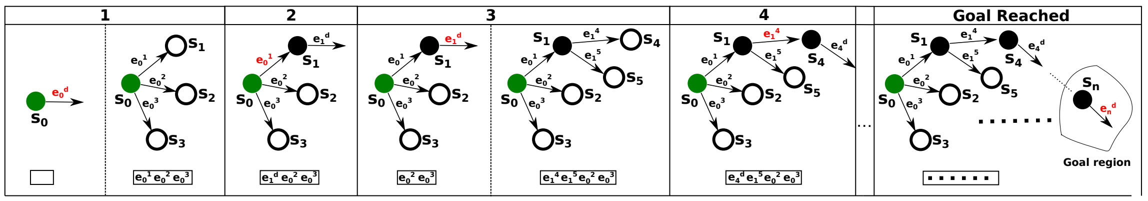


Figure 5.1: Example of eA*: (1) The dummy edge e_0^d originating from $\mathbf{s}_0$ is expanded and the real edges $[e_0^1, e_0^2, e_0^3]$ are inserted into $OPEN$. (2) e_0^1 is expanded, during which it is evaluated, and the successor $\mathbf{s}_1$ is generated. A dummy edge e_1^d from $\mathbf{s}_1$ is inserted into $OPEN$. (3) e_1^d is expanded and the real edges $[e_1^4, e_1^5]$ are inserted into $OPEN$. (4) e_1^4 is expanded, during which it is evaluated, and the successor $\mathbf{s}_4$ is generated, and a dummy edge e_4^d is inserted into E^{open}. This goes on until a dummy edge e_n^d is expanded whose source state belongs to the goal region, i.e., $\mathbf{s}_n \in \mathcal{G}$.

ePA*SE leverages the key algorithmic contribution of PA*SE, i.e., parallel expansions of independent states, but instead uses it to parallelize edge evaluations. In doing so, ePA*SE further improves the efficiency of PA*SE in domains with expensive-to-evaluate edges. ePA*SE obeys the same invariant as A* and PA*SE that when a state is expanded, its g-value is optimal. Therefore, every state is expanded at most once. However, unlike in PA*SE, where each thread is responsible for expanding a single state at a time, each ePA*SE thread is responsible for evaluating a single edge at a time. In order to build up to ePA*SE, we first describe a serial version of the proposed algorithm eA* (Edge-based A*). We then explain how eA* can be parallelized to get to ePA*SE, using the key idea behind PA*SE.

5.2.1 eA*

The first key algorithmic difference in eA* compared to A* is that the open list $OPEN$ contains edges instead of states. We introduce the term *expansion of an edge* and explicitly differentiate it from the expansion of a state. In A*, during the expansion of a state, all its successors are generated and, unless they have already been expanded, are either inserted into the open list or repositioned with the updated priority. In eA*, expansion of an edge $(\mathbf{s}, \mathbf{a})$ involves evaluating the edge to generate the successor state $\mathbf{s}'$ and adding/updating (but not evaluating) the edges originating from $\mathbf{s}'$ into $OPEN$ with the same priority of $g(\mathbf{s}') + h(\mathbf{s}')$. This choice of priority ensures that the edges originating from states that would have the same (state-) expansion priority in A* have the same (edge-) expansion priority in eA*. A state is defined as *partially expanded* if at least one (but not all) of its outgoing edges has been expanded or is under expansion, while it is defined as *expanded* if all its outgoing edges have been expanded. eA* uses the following data structures as the key ingredients of the algorithm.

- $OPEN$: A priority queue of edges (not states) that the search has generated but not expanded, where the edge with the smallest key/priority is placed in the front of the queue. The priority of an edge $e = (\mathbf{s}, \mathbf{a})$ in $OPEN$ is $f\left((\mathbf{s}, \mathbf{a})\right) = g(\mathbf{s}) + h(\mathbf{s})$.
- BE: The set of states that are partially expanded.
- $CLOSED$: The set of states that have been expanded.

Naively storing edges instead of states in $OPEN$ introduces an inefficiency. In A*, the g-value of a state $\mathbf{s}$ can change many times during the search until the state is expanded, at which point it is added to $CLOSED$. Every time this happens,

$OPEN$ has to be rebalanced to reposition $\mathbf{s}$. In eA*, every time $g(\mathbf{s})$ changes, the position of all outgoing edges from $\mathbf{s}$ needs to be updated in $OPEN$. This increases the number of times $OPEN$ has to be rebalanced, which is an expensive operation. However, since the edges originating from $\mathbf{s}$ have the same priority, i.e., $g(\mathbf{s}) + h(\mathbf{s})$, this can be avoided by replacing all the outgoing edges from $\mathbf{s}$ with a single *dummy* edge $e^d = (\mathbf{s}, \mathbf{a^d})$, where $\mathbf{a^d}$ stands for a dummy action. The dummy edge stands as a placeholder for all the *real* edges originating from $\mathbf{s}$. When $g(\mathbf{s})$ changes, only the dummy edge must be repositioned. Unlike when a real edge is expanded, when the dummy edge $(\mathbf{s}, \mathbf{a^d})$ is expanded, it is replaced by the outgoing real edges from $\mathbf{s}$ in $OPEN$. When a state's dummy edge is expanded or under expansion, it is also considered partially expanded and added to BE. When all the outgoing real edges of a state have been expanded, it is moved from BE to $CLOSED$. The g-value $g(\mathbf{s})$ of a state $\mathbf{s}$ in either BE or $CLOSED$ can no longer change; hence, the real edges originating from $\mathbf{s}$ will never have to be updated in $OPEN$.

eA* can be trivially extended to handle an inflation factor on the heuristic like wA*, which leads to a more goal-directed search w-eA* (Weighted eA*). Figure 5.1 shows an example of w-eA* in action. Let e_i^j refer to an edge from state $\mathbf{s}_i$ to $\mathbf{s}_j$, and e_i^d refers to a dummy edge from $\mathbf{s}_i$. The states that are generated are shown in solid circles. The hollow circles represent states that are not generated, and hence the incoming edges to these states are not evaluated. During the first expansion, the dummy edge e_0^d originating from $\mathbf{s}_0$ is expanded, and the real edges $[e_0^1, e_0^2, e_0^3]$ are inserted into $OPEN$. In the second expansion, the edge e_0^1 is expanded, during which it is evaluated, and the successor $\mathbf{s}_1$ is generated. A dummy edge (e_1^d) from $\mathbf{s}_1$ is inserted into $OPEN$. In the third expansion, e_1^d is expanded and the real edges $[e_1^4, e_1^5]$ are inserted into $OPEN$. In the fourth expansion, the edge e_1^4 is expanded, during which it is evaluated, the successor $\mathbf{s}_4$ is generated, and a dummy edge e_4^d is inserted into E^{open}. This continues until a dummy edge e_n^d is expanded whose source state belongs to the goal region, i.e., $\mathbf{s}_n \in \mathcal{G}$.

If the heuristic is informative, w-eA* evaluates fewer edges than wA*. In the example shown in Figure 5.1, the edges $[e_0^2, e_0^3, e_1^5]$ do not get evaluated. Since wA* evaluates all outgoing edges of an expanded state, these edges would be evaluated in the case of wA* (with the same heuristic and inflation factor) when their source states are expanded ($\mathbf{s}_0$ and $\mathbf{s}_1$). Additionally, similar to how wPA*SE parallelizes wA*, w-eA* can be parallelized to obtain a highly efficient algorithm w-ePA*SE. Since w-ePA*SE is a trivial extension of ePA*SE, we instead describe how eA* can be parallelized to obtain ePA*SE.

5.2.2 eA* to ePA*SE

eA* can be parallelized using the key idea behind PA*SE, i.e., parallel expansion of independent states and applying it to edge expansions, resulting in ePA*SE. ePA*SE has two key differences from PA*SE that make it more efficient:

1. Evaluation of edges is decoupled from the expansion of the source state, giving the search the flexibility to figure out what edges need to be evaluated.
2. Evaluation of edges is parallelized.

In addition to $OPEN$ and $CLOSED$, PA*SE uses another data structure BE (Being Expanded) to store the set of states currently being expanded by one of the threads. It uses a pairwise independence check on states in the open list to find states safe to expand in parallel. A state $\mathbf{s}$ is safe to expand if $g(\mathbf{s})$ is already optimal. In other words, there is no other state that is currently being expanded (in BE), nor in $OPEN$ that can reduce $g(\mathbf{s})$. Formally, a state $\mathbf{s}$ is defined to be independent of state $\mathbf{s}$' iff

$$g(\mathbf{s}) - g(\mathbf{s}') \leq h(\mathbf{s}', \mathbf{s}) \tag{5.1}$$

Like in eA*, BE in ePA*SE stores the partially expanded states, as per the definition of partial expansion in eA*. Since ePA*SE stores edges in $OPEN$ instead of states and each ePA*SE thread expands edges instead of states, the independence check has to be modified. An edge e is safe to expand if Equations 5.2 and 5.3 hold.

$$\begin{gathered} g(e.\mathbf{s}) - g(e'.\mathbf{s}) \leq h(e'.\mathbf{s}, e.\mathbf{s}) \\ \forall e' \in OPEN \mid f(e') < f(e) \end{gathered} \tag{5.2}$$

$$g(e.\mathbf{s}) - g(\mathbf{s}') \leq h(\mathbf{s}', e.\mathbf{s}) \; \forall \mathbf{s}' \in BE \tag{5.3}$$

Equation 5.2 ensures that there is no edge in $OPEN$ with a priority smaller than that of e, that upon expansion, can lower the g-value of $e.\mathbf{s}$ and hence lower the priority of e. In other words, the source state $\mathbf{s}$ of edge e is independent of the source states of all edges in $OPEN$, which have a smaller priority than e. Equation 5.3 ensures that there is no partially expanded state that can lower the g-value of $e.\mathbf{s}$. In other words, the source state $\mathbf{s}$ of edge e is independent of all states in BE.

5.2.3 Details

The pseudocode for ePA*SE is presented in Alg. 9. The main planning loop in Plan runs on a single thread (thread 0), and in each iteration, an edge is removed for expansion from $OPEN$ that has the smallest possible priority and is also safe to expand, as per Equations 5.2 and 5.3 (Line 14, Alg. 9). If such an edge is not found, the thread waits for either $OPEN$ or BE to change (Line 17, Alg. 9). If a safe-to-expand edge is found, such that the source state of the edge belongs to the goal region, the solution path is returned by backtracking from the state to the start state using back pointers (Line 24, Alg. 9). Otherwise, the edge is expanded and assigned to an edge expansion thread (thread $i = 1 : N_t$) (Line 33, Alg. 9). The edge expansion threads are spawned as and when needed to avoid the overhead of running unused threads (Line 31, Alg. 9). The search terminates when either a solution is found or when $OPEN$ is empty, and all threads are idle (BE is empty), in which case there is no solution.

If the edge to be expanded is a dummy edge, the source $\mathbf{s}$ of the edge is marked as partially expanded by adding it to BE (Line 11, Alg. 10). The real edges originating from $\mathbf{s}$ are added to $OPEN$ with the same priority as that of the dummy edge i.e. $g(\mathbf{s}) + h(\mathbf{s})$. If the expanded edge is not a dummy edge, it is evaluated (Line 18, Alg. 10) to obtain the successor $\mathbf{s}'$ and the edge cost $c\left((\mathbf{s}, \mathbf{a})\right)$. This is the expensive operation that ePA*SE seeks to parallelize, which is why it happens lock-free. If the expanded edge reduces $g(\mathbf{s}')$, the dummy edge originating from $\mathbf{s}'$ is added/updated in $OPEN$. A counter $n\_successors\_generated$ keeps track of the number of outgoing edges that have been expanded for every state. Once all the outgoing edges for a state have been expanded, and hence the state has been expanded, it is removed from BE and added to $CLOSED$ (Lines 29 and 30, Alg. 10).

Algorithm 9 w-ePA*SE: Planning Loop

```
1: $\mathcal{A} \leftarrow$ action space , $N_t \leftarrow$ number of threads, $G \leftarrow \emptyset$
2: $\mathbf{s}_0 \leftarrow$ start state , $\mathcal{G} \leftarrow$ goal region, $terminate \leftarrow$ False
3: procedure Plan
4:     $\forall \mathbf{s} \in G$, $\mathbf{s}.g \leftarrow \infty$, $n\_successors\_generated(s) = 0$
5:     $\mathbf{s}_0.g \leftarrow 0$
6:     insert $(\mathbf{s}_0, \mathbf{a}^{\mathbf{d}})$ in $OPEN$                ▷ Dummy edge from $\mathbf{s}_0$
7:     LOCK
8:     while not $terminate$ do
9:         if $OPEN = \emptyset$ and $BE = \emptyset$ then
10:            $terminate$ = True
11:            UNLOCK
12:            return $\emptyset$
13:        end if
14:        remove an edge $(\mathbf{s}, \mathbf{a})$ from $OPEN$ that has the
           smallest $f((\mathbf{s}, \mathbf{a}))$ among all states in $OPEN$ that
           satisfy Equations 5.2 and 5.3
15:        if such an edge does not exist then
16:            UNLOCK
17:            wait until $OPEN$ or $BE$ change
18:            LOCK
19:            continue
20:        end if
21:        if $\mathbf{s} \in \mathcal{G}$ then
22:            $terminate$ = True
23:            UNLOCK
24:            return Backtrack($\mathbf{s}$)
25:        else
26:            UNLOCK
27:            while $(\mathbf{s}, \mathbf{a})$ has not been assigned a thread do
28:                for $i = 1 : N_t$ do
29:                    if thread $i$ is available then
30:                        if thread $i$ has not been spawned then
31:                            Spawn EdgeExpandThread($i$)
32:                        end if
33:                        Assign $(\mathbf{s}, \mathbf{a})$ to thread $i$
34:                    end if
35:                end for
36:            end while
37:            LOCK
38:        end if
39:    end while
40:    $terminate$ = True
41:    UNLOCK
42: end procedure
```

Algorithm 10 w-ePA*SE: Edge Expansion

```
procedure EdgeExpandThread($i$)
    while not $terminate$ do
        if thread $i$ has been assigned an edge $(\mathbf{s}, \mathbf{a})$ then
            Expand $((\mathbf{s}, \mathbf{a}))$
        end if
    end while
end procedure
procedure Expand($(\mathbf{s}, \mathbf{a})$)
    LOCK
    if $\mathbf{a} = \mathbf{a^d}$ then
        insert $\mathbf{s}$ in $BE$
        for $\mathbf{a} \in \mathcal{A}$ do
            $f((\mathbf{s}, \mathbf{a})) = g(\mathbf{s}) + h(\mathbf{s})$
            insert $(\mathbf{s}, \mathbf{a})$ in $OPEN$ with $f((\mathbf{s}, \mathbf{a}))$
        end for
    else
        UNLOCK
        $\mathbf{s}', c((\mathbf{s}, \mathbf{a})) \leftarrow$ GenerateSuccessor $((\mathbf{s}, \mathbf{a}))$
        LOCK
        if $\mathbf{s}' \notin CLOSED \cup BE$ and
          $g(\mathbf{s}') > g(\mathbf{s}) + c((\mathbf{s}, \mathbf{a}))$ then
            $g(\mathbf{s}') = g(\mathbf{s}) + c((\mathbf{s}, \mathbf{a}))$
            $\mathbf{s}'.parent = \mathbf{s}$
            $f\left((\mathbf{s}', \mathbf{a^d})\right) = g(\mathbf{s}') + h(\mathbf{s}')$
            insert/update $(\mathbf{s}', \mathbf{a^d})$ in $OPEN$ with $f\left((\mathbf{s}', \mathbf{a^d})\right)$
        end if
        $n\_successors\_generated(\mathbf{s}) += 1$
        if $n\_successors\_generated(\mathbf{s}) = |\mathcal{A}|$ then
            remove $\mathbf{s}$ from $BE$
            insert $\mathbf{s}$ in $CLOSED$
        end if
    end if
    UNLOCK
end procedure
```

5.2.4 Thread management

In PA*SE, the state expansion threads are spawned at the start, and each of them independently pulls out states from the open list to expand. When the number of threads is higher than the number of independent states available for expansion at any point, the operating system has an unnecessary overhead of spinning unused threads. This causes the overall performance to decrease as the number of unused threads increases (see Figure 6 in [20]). Our initial experiments showed that using a similar thread management strategy in ePA*SE leads to a similar degradation in performance as the number of threads is increased beyond the optimal number of threads, even though the peak performance of ePA*SE is substantially higher than that of PA*SE. To prevent this degradation in performance, ePA*SE employs a different thread management strategy. There is a single thread that pulls out edges from the open list, and it spawns edge expansion threads as needed but capped at N_t (Line 31). When N_t is higher than the number of independent edges available for expansion at any point in time, only a subset of available threads get spawned, preventing performance degradation, as we will show in our experiments.

5.2.5 w-ePA*SE

w-ePA*SE is a bounded suboptimal variant of ePA*SE that trades off optimality for faster planning. Similar to wPA*SE, w-ePA*SE introduces two inflation factors, the first of which, $\epsilon \geq 1$, relaxes the independence rule (Equations 5.2 and 5.3) as follows.

$$\begin{gathered} g(e.\mathbf{s}) - g(e'.\mathbf{s}) \leq \epsilon h(e'.\mathbf{s}, e.\mathbf{s}) \\ \forall e' \in OPEN \mid f(e') < f(e) \end{gathered} \tag{5.4}$$

$$g(e.\mathbf{s}) - g(\mathbf{s}') \leq \epsilon h(\mathbf{s}', e.\mathbf{s}) \; \forall \mathbf{s}' \in BE \tag{5.5}$$

The second factor $w \geq 1$ is used to inflate the heuristic in the priority of edges in $OPEN$, i.e., $f((\mathbf{s}, \mathbf{a})) = g(\mathbf{s}) + w \cdot h(\mathbf{s})$ which makes the search more goal-directed. As long as $\epsilon \geq w$, the solution cost is bounded by $\epsilon \cdot c^*$ (Theorem 8). Note that w can be greater than ϵ, but then Equation 5.4 has to consider source states of all edges in $OPEN$ and the solution cost will be bounded by $w \cdot c^*$ (Theorem 6). Since this leads to significantly more independence checks, the $\epsilon \geq w$ relationship is typically recommended in practice.

5.3 Properties

w-ePA*SE has identical properties to that of w-PA*SE [20] and can be proved similarly with minor modifications.

Theorem 6 (Bounded suboptimal expansions) *When w-ePA*SE that performs independence checks against all states in BE and source states of **all** edges in $OPEN$, chooses an edge e for expansion, then $g(e.\mathbf{s}) \leq \lambda g^*(\mathbf{s})$, where $\lambda = \max(\epsilon, w)$.*

Proof Assume, for the sake of contradiction, that $g(e.\mathbf{s}) > \lambda g^*(e.\mathbf{s})$ directly before edge e is expanded, and without loss of generality, that $g(e'.\mathbf{s}) \leq \lambda g^*(e'.\mathbf{s})$ for all edges e' selected for expansion before e (**Assumption**). Let $\mathbf{s} = e.\mathbf{s}$ for ease of notation. Consider any cost-minimal path $\pi(\mathbf{s}_0, \mathbf{s})$ from $\mathbf{s}_0$ to $\mathbf{s}$. Let $\mathbf{s}_m$ be the closest state to $\mathbf{s}_0$ on $\pi(\mathbf{s}_0, \mathbf{s})$ such that either 1) there exists at least one edge in $OPEN$ with source state $\mathbf{s}_m$ or 2) $\mathbf{s}_m$ is in BE. $\mathbf{s}_m$ is no farther away from $\mathbf{s}_0$ on $\pi(\mathbf{s}_0, \mathbf{s})$ than $\mathbf{s}$ since e is in $OPEN$. Therefore, let $\pi(\mathbf{s}_0, \mathbf{s}_m)$ and $\pi(\mathbf{s}_m, \mathbf{s})$ be the subpaths of $\pi(\mathbf{s}_0, \mathbf{s})$ from $\mathbf{s}_0$ to $\mathbf{s}_m$ and from $\mathbf{s}_m$ to $\mathbf{s}$, respectively.

If $\mathbf{s}_m = \mathbf{s}_0$, then $g(\mathbf{s}_m) \leq \lambda g^*(\mathbf{s}_m)$ since $g(\mathbf{s}_0) = g^*(\mathbf{s}_0) = 0$ (**Contradiction 1**). Otherwise, let $\mathbf{s}_p$ be the predecessor of $\mathbf{s}_m$ on $\pi(\mathbf{s}_0, \mathbf{s})$. $\mathbf{s}_p$ has been expanded (i.e. all edges outgoing edges of $\mathbf{s}_p$ have been expanded) since every state closer to $\mathbf{s}_0$ on $\pi(\mathbf{s}_0, \mathbf{s})$ than $\mathbf{s}_m$ has been expanded (since every unexpanded state on $\pi(\mathbf{s}_0, \mathbf{s})$ different from $\mathbf{s}_0$ is either in BE or has an outgoing edge in $OPEN$, or has a state closer to $\mathbf{s}_0$ on $\pi(\mathbf{s}_0, \mathbf{s})$ that is either in BE or has an outgoing edge in $OPEN$). Therefore, since all outgoing edges from $\mathbf{s}_p$ have been expanded, $g(\mathbf{s}_p) \leq \lambda g^*(\mathbf{s}_p)$ because of **Assumption**. Then, because of the g update of $\mathbf{s}_m$ when the edge from $\mathbf{s}_p$ to $\mathbf{s}_m$ was expanded,

$$\begin{aligned} g(\mathbf{s}_m) &\leq g(\mathbf{s}_p) + c(\mathbf{s}_p, \mathbf{s}_m) \\ &\leq \lambda g^*(\mathbf{s}_p) + c(\mathbf{s}_p, \mathbf{s}_m) \end{aligned} \tag{5.6}$$

Since $\mathbf{s}_p$ is the predecessor of $\mathbf{s}_m$ on the cost-minimal path $\pi(\mathbf{s}_0, \mathbf{s})$,

$$\begin{aligned} & g^*(\mathbf{s}_m) = g^*(\mathbf{s}_p) + c(\mathbf{s}_p, \mathbf{s}_m) \\ \implies & g^*(\mathbf{s}_p) = g^*(\mathbf{s}_m) - c(\mathbf{s}_p, \mathbf{s}_m) \end{aligned} \tag{5.7}$$

Substituting $g^*(\mathbf{s}_p)$ from Equation 5.7 into Equation 5.6,

$$\begin{aligned}
&\implies g(\mathbf{s}_m) \leq \lambda g^*(\mathbf{s}_m) - (\lambda - 1)c(\mathbf{s}_p, \mathbf{s}_m) \\
&\implies g(\mathbf{s}_m) \leq \lambda g^*(\mathbf{s}_m) \\
&\implies \lambda c(\pi(\mathbf{s}_0, \mathbf{s}_m)) = \lambda g^*(\mathbf{s}_m) \geq g(\mathbf{s}_m)
\end{aligned} \tag{5.8}$$

Since $h(\mathbf{s}_m, \mathbf{s})$ satisfies forward-backward consistency and is therefore admissible, $h(\mathbf{s}_m, \mathbf{s}) \leq c(\pi(\mathbf{s}_m, \mathbf{s}))$. Since, $\lambda = \max(\epsilon, w)$, $\epsilon \leq \lambda$,

$$\lambda c(\pi(\mathbf{s}_m, \mathbf{s})) \geq \lambda h(\mathbf{s}_m, \mathbf{s}) \geq \epsilon h(\mathbf{s}_m, \mathbf{s}) \tag{5.9}$$

Adding 5.8 and 5.9,

$$\begin{aligned}
\lambda c(\mathbf{s}_0, \mathbf{s}) &= \lambda c(\mathbf{s}_0, \mathbf{s}_m) + \lambda c(\mathbf{s}_m, \mathbf{s}) \\
&\geq g(\mathbf{s}_m) + \epsilon h(\mathbf{s}_m, \mathbf{s})
\end{aligned} \tag{5.10}$$

Assuming w-ePA*SE performs independence checks against states in BE and source states of all edges in $OPEN$ when choosing an edge e with source $e.\mathbf{s}$ to expand, and $\mathbf{s}_m$ is either in BE or there exists at least one edge with source $\mathbf{s}_m$ in $OPEN$,

$$\begin{aligned}
&\epsilon h(e'.\mathbf{s}, e.\mathbf{s}) \geq g(e.\mathbf{s}) - g(e'.\mathbf{s}) \\
&\forall e' \in OPEN \mid e'.\mathbf{s} = \mathbf{s}_m \\
\implies &g(\mathbf{s}_m) + \epsilon h(\mathbf{s}_m, \mathbf{s}) \geq g(\mathbf{s})
\end{aligned} \tag{5.11}$$

Therefore,

$$\begin{aligned}
\lambda g^*(\mathbf{s}) =& \lambda c(\pi(\mathbf{s}_0, \mathbf{s})) \\
\geq& g(\mathbf{s}_m) + \epsilon h(\mathbf{s}_m, \mathbf{s}) && \text{(Using Eq. 5.10)} \\
\geq& g(\mathbf{s}) && \text{(Using Eq. 5.11)}
\end{aligned}$$

(**Contradiction 2**)

Contradiction 1 and **Contradiction 2** invalidate the **Assumption**, which proves **Theorem** 6.

Theorem 7 *If $w \leq \epsilon$, and considering any two edges e and e' in $OPEN$, the source state of e is independent of the source state of e' if $f(e) \leq f(e')$.*

Proof

$$
\begin{aligned}
& f(e) \leq f(e') \\
\implies & g(e.\mathbf{s}) + wh(e.\mathbf{s}) \leq g(e'.\mathbf{s}) + wh(e'.\mathbf{s}) \\
\implies & g(e.\mathbf{s}) \leq g(e'.\mathbf{s}) + w(h(e'.\mathbf{s}) - h(e.\mathbf{s})) \\
& \leq g(e'.\mathbf{s}) + wh(e'.\mathbf{s}, e.\mathbf{s}) \\
& \text{(forward-backward consistency)} \\
& \leq g(e'.\mathbf{s}) + \epsilon h(e'.\mathbf{s}, e.\mathbf{s}) \\
& \text{(since } w \leq \epsilon)
\end{aligned}
$$

Therefore, $e.\mathbf{s}$ is independent of $e'.\mathbf{s}$ by definition (Eq. 5.1).

Theorem 8 (Bounded suboptimality) *If $w \leq \epsilon$, and w-ePA*SE chooses a dummy edge $e^d = (\mathbf{s}, \mathbf{a^d})$ for expansion, such that the source state $\mathbf{s}$ belongs to the goal region i.e. $\mathbf{s} \in \mathcal{G}$, then $g(\mathbf{s}) \leq \epsilon g^*(\mathbf{s}) = \epsilon \cdot c^*$.*

Proof This directly follows from theorem 6 and 7.

Theorem 9 (Completeness) *If there exists at least one path π in G from $\mathbf{s}_0$ to $\mathcal{G}$, w-ePA*SE will find it.*

Proof This proof makes use of Theorem 8 and is similar to the equivalent proof of serial wA*.

5.4 Evaluation

We evaluate w-ePA*SE in two planning domains where edge evaluation is expensive. All experiments were carried out on Amazon Web Services (AWS) instances. All algorithms were implemented in C++.

5.4.1 3D Navigation

The first domain is the PR2 navigation domain introduced in Section 4.4 of Chapter 4 and shown in Figure 4.3. We evaluate 50 trials in each of which the start configuration of the robot and goal region are sampled randomly.

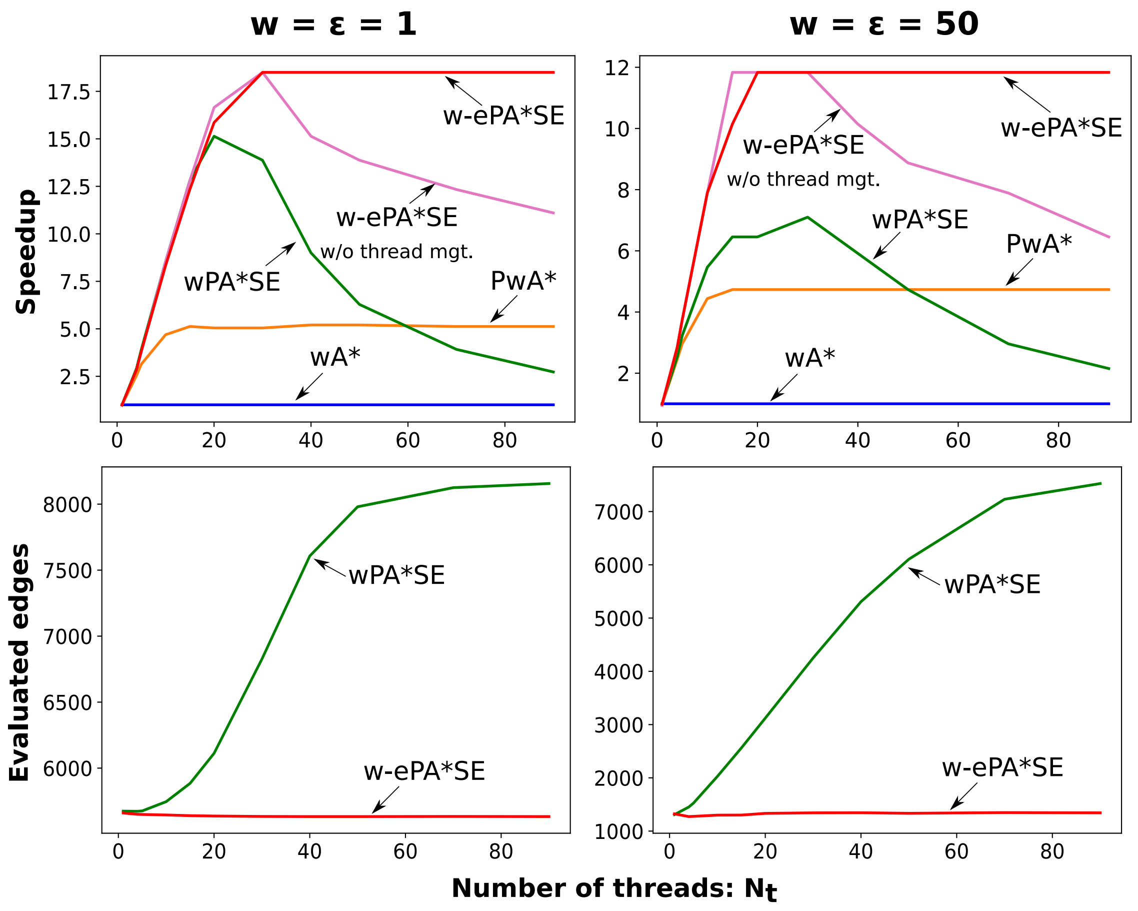


Figure 5.2: (**Navigation**) Top: Mean speedup achieved by PwA*, w-PA*SE and w-ePA*SE over wA*. Bottom: Mean number of edges evaluated by w-PA*SE and w-ePA*SE.

We compare w-ePA*SE with other CPU-based parallel search baselines. The first baseline is a variant of weighted A* in which, during a state expansion, the successors of the state are generated, and the corresponding edges are evaluated in parallel. For lack of a better term, we call this baseline Parallel Weighted A* (PwA*). Note that this is very different from the Parallel A* (PA*) algorithm [21], which has already

	Number of threads (N_t)										
	1	4	5	10	15	20	30	40	50	70	90
	$w = \epsilon = 1$										
wA*	3.33	-	-	-	-	-	-	-	-	-	-
PwA*	3.37	1.31	1.06	0.71	0.65	0.66	0.66	0.64	0.64	0.65	0.65
w-PA*SE	3.37	1.14	0.85	0.39	0.26	0.22	0.24	0.37	0.53	0.85	1.22
w-ePA*SE w/o thread mgt.	3.43	1.17	0.88	0.39	0.26	0.20	0.18	0.22	0.24	0.27	0.30
w-ePA*SE	**3.34**	**1.17**	**0.87**	**0.40**	**0.27**	**0.21**	**0.18**	**0.18**	**0.18**	**0.18**	**0.18**
	$w = \epsilon = 50$										
wA*	0.71	-	-	-	-	-	-	-	-	-	-
PwA*	0.72	0.29	0.24	0.16	0.15	0.15	0.15	0.15	0.15	0.15	0.15
w-PA*SE	0.71	0.28	0.22	0.13	0.11	0.11	0.10	0.12	0.15	0.24	0.33
w-ePA*SE w/o thread mgt.	0.75	0.25	0.19	0.09	0.06	0.06	0.06	0.07	0.08	0.09	0.11
w-ePA*SE	**0.72**	**0.25**	**0.19**	**0.09**	**0.07**	**0.06**	**0.06**	**0.06**	**0.06**	**0.06**	**0.06**

Table 5.1: (**Navigation**) Mean planning times (s) for wA*, PwA*, w-PA*SE and w-ePA*SE for varying N_t, with $w = \epsilon = 1$ (top) and with $w = \epsilon = 50$ (bottom).

been shown to underperform w-PA*SE [20]. The second baseline is w-PA*SE. These two baselines leverage parallelization differently. PwA* parallelizes the generation of successors, whereas w-PA*SE parallelizes state expansions. w-ePA*SE, on the other hand, parallelizes edge evaluations. We also compare against a variation of w-ePA*SE (w-ePA*SE w/o thread mgt.) that uses the thread management strategy of w-PA*SE as opposed to the improved thread management strategy described in the Method. Speedup over wA* is defined as the ratio of the mean runtime of wA* over the mean runtime of a specific algorithm.

Figure 5.2 (top) shows the mean speedup achieved by w-PA*SE and the baselines over wA* for varying N_t, for $w = \epsilon = 1$ and $w = \epsilon = 50$. The corresponding raw planning times are shown in Table 5.1. The speedup achieved by PwA* saturates at the branching factor of the domain. This is expected since PwA* parallelizes the evaluation of the outgoing edges of a state being expanded. If N_t is greater than the branching factor M, $N_t - M$ threads remain unutilized. For low N_t, the speedup achieved by w-ePA*SE matches that of w-PA*SE. However, for high N_t, the speedup achieved by w-ePA*SE rapidly outpaces that of w-ePA*SE, especially for the inflated heuristic case. This is because w-ePA*SE is much more efficient than w-PA*SE since it parallelizes edge evaluations instead of state expansions. This increased efficiency is more apparent with the availability of a larger computational budget in the form

	Number of threads (N_t)										
	1	4	5	10	15	20	30	40	50	70	90
						$w = \epsilon = 1$					
PA*SE	5674	5646	5644	5681	5796	5986	6709	7523	7864	8063	8000
w-ePA*SE	**5660**	**5654**	**5653**	**5650**	**5647**	**5644**	**5644**	**5649**	**5653**	**5662**	**5674**
						$w = \epsilon = 50$					
w-PA*SE	1309	1494	1560	2013	2556	3105	4098	5117	5886	7000	7427
w-ePA*SE	**1311**	**1274**	**1273**	**1296**	**1311**	**1325**	**1366**	**1396**	**1450**	**1529**	**1665**

Table 5.2: (**Navigation**) Number of edges evaluated by w-PA*SE and w-ePA*SE for varying N_t, with $w = \epsilon = 1$ (top) and with $w = \epsilon = 50$ (bottom).

of a greater number of threads to allocate to evaluating edges. The speedup of w-PA*SE reaches a peak and then rapidly deteriorates. This is also the case for w-ePA*SE w/o (improved) thread mgt. (described in the Method), even though the peak speedup of w-ePA*SE w/o thread mgt. is higher than that of w-PA*SE. However, the speedup of w-ePA*SE with the improved thread management strategy reaches a maximum and then saturates instead of degrading. This is due to the difference in the multithreading strategy employed by w-ePA*SE as explained in the Method.

Figure 5.2 (bottom) and Table 5.2 show that w-ePA*SE evaluates significantly fewer edges as compared to w-PA*SE. With a greater number of threads, the difference is significant. This indicates that beyond parallelization of edge evaluations, the w-eA* formulation that w-ePA*SE uses has another advantage that if the heuristic is informative, w-ePA*SE evaluates fewer edges than w-PA*SE, which contributes to the lower planning time of w-ePA*SE. The intuition behind this is that in wA*, the evaluation of the outgoing edges from a given state is tightly coupled with the expansion of the state because all the outgoing edges from a given state must be evaluated at the same time when the state is expanded. In w-eA*, the evaluation of these edges is decoupled from each other since the search expands edges instead of states.

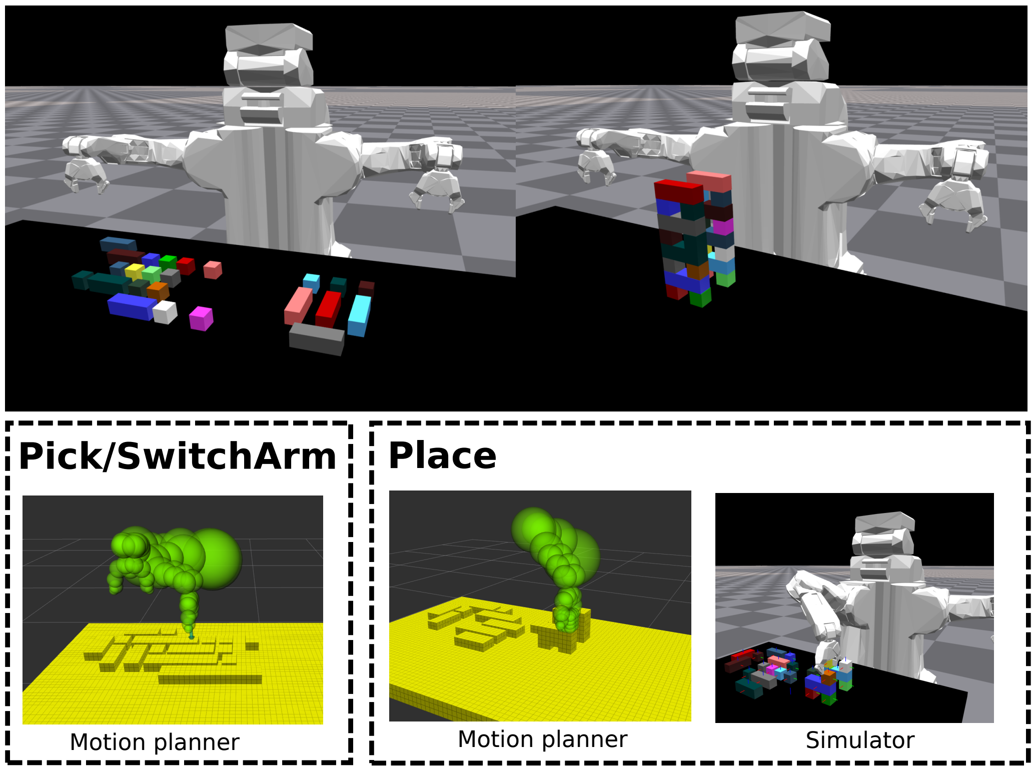


Figure 5.3: (**Assembly**) Top: The PR2 has to arrange a set of blocks on the table (left) into a given configuration (right). Bottom: It is equipped with Pick, Place and SwitchArm controllers. The Pick controller uses the motion planner to reach a block. The Place controller uses the motion planner to place a block and simulates the outcome of releasing the block. The SwitchArm controller uses the motion planner to move the active arm to a home position.

5.4.2 Assembly Task

The second domain is a task and motion planning problem of assembling a set of blocks on a table into a given structure by a PR2, as shown in Figure 5.3. This domain is similar to the one introduced in Section 4.4 of Chapter 4, but in this work, we enable the dual-arm functionality of the PR2. We assume full state observability of the 6D poses of the blocks and the robot's joint configuration. The goal is defined by the 6D poses of each block in the desired structure. The PR2 is equipped with Pick and Place controllers, which are used as macro-actions in high-level planning. In addition, there is a SwitchArm controller which switches the active arm by moving the current active arm to a home position. All of these actions use a motion plan-

	wA*	PwA*	w-PA*SE	w-ePA*SE
N_t	1	25	10	10
Time (s)	3010	1066	419	**301**
Speedup	1	2.8	7.2	**10**

Table 5.3: (**Assembly**) Mean planning times and speedup over wA* for w-ePA*SE and the baselines.

ner internally to compute collision-free trajectories in the workspace. Additionally, Place has access to a simulator (NVIDIA Isaac Gym [58]) to simulate the outcome of placing a block at its desired pose. For example, if the planner tries to place a block at its final pose but has not placed the block underneath yet, the placed block will not be supported, and the structure will not be stable. This would lead to an invalid successor during planning. We set a simulation timeout of $t_s = 0.2$ s to evaluate the outcome of placing a block. Considering the variability in the simulation speed and the overhead of communicating with the simulator, this results in a total wall time of less than 2 s for the simulation. The motion planner has a timeout of $t_p = 60$ s based on the wall time, and therefore that is the maximum time the motion planning can take. Successful Pick, Place and SwitchArm actions have unit costs and infinite otherwise. A Pick action on a block is successful if the motion planner finds a feasible trajectory to reach the block within t_p. A Place action on a block is successful if the motion planner finds a feasible trajectory to place the block within t_p, and simulating the block placement results in the block coming to rest at the desired pose within t_s. A SwitchArm action is successful if the motion planner finds a feasible trajectory to the home position for the active arm within t_p. The number of blocks that are not in their final desired pose is used as the admissible heuristic, with $w = \epsilon = 5$. Table 5.3 shows planning times and speedup over wA* for w-ePA*SE and those of the baselines. We use 25 threads in the case of PwA* because that is the maximum branching factor in this domain. The numbers are averaged across 20 trials, in each of which the blocks are arranged in random order on the table. Table 5.3 shows the mean planning times and speedup over wA* of w-ePA*SE as compared to those of the lazy search baselines. w-ePA*SE achieves a 10x speedup over wA* and outperforms the baselines in this domain as well.

5.5 Conclusion

We presented an optimal parallel search algorithm ePA*SE that improves on PA*SE by parallelizing edge evaluations instead of state expansions. We also presented a sub-optimal variant w-ePA*SE and proved that it maintains bounded suboptimality guarantees. Our experiments showed that w-ePA*SE achieves an impressive reduction in planning time across two very different planning domains, which shows the generalizability of our conclusions. Empirically, we have observed w-ePA*SE to be a strict improvement over w-PA*SE for domains with expensive-to-compute edges. Even though we also test with a relatively large budget of threads, the performance improvement is significant even with a smaller budget of fewer than 10 threads, which is the case with typical mobile computers.

Chapter 6

A-ePA*SE: Anytime Edge-Based Parallel A* for Slow Evaluations

Though ePA*SE is highly efficient, it needs to come up with a solution under a strict time budget for it to be applicable in real-time robotics. Though the optimal solution is preferable, that is often not the first priority. For such domains, anytime algorithms have been developed that first prioritize a quick feasible solution by allowing a high sub-optimality bound. This is typically done by incorporating a high inflation factor on the heuristic. They then attempt to improve the solution by incrementally decreasing the inflation factor and therefore tightening the sub-optimality bound, until the time runs out. In this chapter, we bring the anytime property to ePA*SE. We show that the resulting algorithm, A-ePA*SE, achieves higher efficiency than existing anytime algorithms.

6.1 Problem Definition

Let a finite graph $G = (\mathcal{V}, \mathcal{E})$ be defined as a set of vertices $\mathcal{V}$ and directed edges $\mathcal{E}$. Each vertex $v \in \mathcal{V}$ represents a state $\mathbf{s}$ in the state space of the domain $\mathcal{S}$. An edge $e \in \mathcal{E}$ connecting two vertices v_1 and v_2 in the graph represents an action $\mathbf{a} \in \mathcal{A}$ that takes the agent from corresponding states $\mathbf{s}_1$ to $\mathbf{s}_2$. In this work, we assume that all actions are deterministic. Hence an edge e can be represented as a pair $(\mathbf{s}, \mathbf{a})$, where $\mathbf{s}$ is the state at which action $\mathbf{a}$ is executed. For an edge e, we will refer to the corresponding state and action as $e.\mathbf{s}$ and $e.\mathbf{a}$ respectively. $\mathbf{s}_0$ is the start state and $\mathcal{G}$ is the goal region. $c : \mathcal{E} \rightarrow [0, \infty]$ is the cost associated with an edge. $g(\mathbf{s})$ or g-value is the cost of the best path to $\mathbf{s}$ from $\mathbf{s}_0$ found by the algorithm so far and $h(\mathbf{s})$ is a consistent heuristic [17]. Additionally, there exists a forward-backward consistent [20] pairwise heuristic function $h(\mathbf{s}, \mathbf{s}')$ that provides an estimate of the cost between any pair of states. A path π is defined by an ordered sequence of edges $e_{i=1}^{N} = (\mathbf{s}, \mathbf{a})_{i=1}^{N}$, the cost of which is denoted as $c(\pi) = \sum_{i=1}^{N} c(e_i)$. The objective is to find a path π from $\mathbf{s}_0$ to a state in the goal region $\mathcal{G}$ within a time budget T. There is a computational budget of N_t threads which can run in parallel.

6.2 Method

Algorithm 11 A-ePA*SE: Plan

```
1: $\mathcal{A} \leftarrow$ action space , $N_t \leftarrow$ thread budget, $T \leftarrow$ time budget
2: $w_0 \leftarrow$ initial heuristic weight, $\Delta w \leftarrow$ delta heuristic weight
3: $G \leftarrow$ graph, $\mathbf{s}_0 \leftarrow$ start state , $\mathcal{G} \leftarrow$ goal region
4: $terminate \leftarrow$ False
5: procedure PLAN
6:     $INCON \leftarrow \emptyset$
7:     $\forall \mathbf{s} \in G$, $\mathbf{s}.g \leftarrow \infty$
8:     $\mathbf{s}_0.g \leftarrow 0$, $w = w_0$
9:     insert $(\mathbf{s}_0, \mathbf{a}^\mathbf{d})$ in $OPEN$                    ▷ Dummy edge from $\mathbf{s}_0$
10:    while $w >= 1$ and not TIMEOUT $(T)$ do
11:        $INCON = \emptyset, CLOSED = \emptyset$
12:        IMPROVEPATH($w$)
13:        Publish current $w$ bounded sub-optimal solution
14:        $w = w - \Delta w$
15:        $OPEN = OPEN \cup INCON$
16:        Re-balance $OPEN$ with new $w$
17:    end while
18:    $terminate$ = True
19: end procedure
```

Inspired by ARA* [19], we extend w-ePA*SE to a parallelized anytime repairing algorithm A-ePA*SE by inheriting three algorithmic techniques:

1. Define locally inconsistent states as the states whose g-values change while they are in $CLOSED \cup BE$ during the current IMPROVEPATH execution ([19]). A-ePA*SE keeps track of locally inconsistent states by maintaining an inconsistent list $INCON$ (Line 30, Alg. 13).

Algorithm 12 A-ePA*SE: ImprovePath

```
 1: procedure ImprovePath
 2:     LOCK
 3:     while $f(\mathbf{s}_g) > min_{\mathbf{s} \in OPEN}(f(\mathbf{s}))$ do
 4:         if $OPEN = \emptyset$ and $BE = \emptyset$ then
 5:             UNLOCK
 6:             return $\emptyset$
 7:         end if
 8:         remove an edge $(\mathbf{s}, \mathbf{a})$ from $OPEN$ that has the
            smallest $f((\mathbf{s}, \mathbf{a}))$ among all states in $OPEN$ that
            satisfy Equations 5.2 and 5.3
 9:         if such an edge does not exist then
10:             UNLOCK
11:             wait until $OPEN$ or $BE$ change
12:             LOCK
13:             continue
14:         end if
15:         if $\mathbf{s} \in \mathcal{G}$ and $f(\mathbf{s}_g) > f(\mathbf{s})$ then
16:             $\mathbf{s}_g = \mathbf{s}$
17:             $plan$ = Backtrack($\mathbf{s}$)
18:         end if
19:         UNLOCK
20:         while $(\mathbf{s}, \mathbf{a})$ has not been assigned a thread do
21:             for $i = 1 : N_t$ do
22:                 if thread $i$ is available then
23:                     if thread $i$ has not been spawned then
24:                         Spawn EdgeExpandThread($i$)
25:                     end if
26:                     Assign $(\mathbf{s}, \mathbf{a})$ to thread $i$
27:                 end if
28:             end for
29:         end while
30:         LOCK
31:     end while
32:     UNLOCK
33:     return $plan$
34: end procedure
```

2. After every i^{th} ImprovePath call exits, A-ePA*SE initializes $OPEN$ for the next search iteration $i+1$ as $OPEN_{i+1} = OPEN_i \cup INCON$. (Line 15, Alg.11).
3. A-ePA*SE changes the termination condition of a search iteration (Line 3, Alg.12) so that it only expands states that 1) have g-value that can be lowered in the current ImprovePath iteration or 2) were locally inconsistent in the previous ImprovePath iteration.

Algorithm 13 A-ePA*SE: Edge Expansion

```
1: procedure EdgeExpandThread($i$)
2:     while not $terminate$ do
3:         if thread $i$ has been assigned an edge $(\mathbf{s}, \mathbf{a})$ then
4:             Expand $((\mathbf{s}, \mathbf{a}))$
5:         end if
6:     end while
7: end procedure
8: procedure Expand($(\mathbf{s}, \mathbf{a})$)
9:     LOCK
10:    if $\mathbf{a} = \mathbf{a^d}$ then
11:        insert $\mathbf{s}$ in $BE$
12:        for $\mathbf{a} \in \mathcal{A}$ do
13:            $f((\mathbf{s}, \mathbf{a})) = g(\mathbf{s}) + h(\mathbf{s})$
14:            insert $(\mathbf{s}, \mathbf{a})$ in $OPEN$ with $f((\mathbf{s}, \mathbf{a}))$
15:        end for
16:    else
17:        UNLOCK
18:        if NotEvaluated $((\mathbf{s}, \mathbf{a}))$ then
19:            $\mathbf{s}', c((\mathbf{s}, \mathbf{a})) \leftarrow$ GenerateSuccessor $((\mathbf{s}, \mathbf{a}))$
20:        else
21:            $\mathbf{s}', c((\mathbf{s}, \mathbf{a})) \leftarrow$ GetSuccessor $((\mathbf{s}, \mathbf{a}))$
22:        end if
23:        LOCK
24:        if $g(\mathbf{s}') > g(\mathbf{s}) + c((\mathbf{s}, \mathbf{a}))$ then
25:            $g(\mathbf{s}') = g(\mathbf{s}) + c((\mathbf{s}, \mathbf{a}))$
26:            $f\left((\mathbf{s}', \mathbf{a^d})\right) = g(\mathbf{s}') + wh(\mathbf{s}')$
27:            if $\mathbf{s}' \notin CLOSED \cup BE$ then
28:                update $(\mathbf{s}', \mathbf{a^d})$ in $OPEN$ with $f\left((\mathbf{s}', \mathbf{a^d})\right)$
29:            else
30:                update $(\mathbf{s}', \mathbf{a^d})$ in $INCON$ with $f\left((\mathbf{s}', \mathbf{a^d})\right)$
31:            end if
32:        end if
33:        $n\_successors\_generated(\mathbf{s}) += 1$
34:        if $n\_successors\_generated(\mathbf{s}) = |\mathcal{A}|$ then
35:            remove $\mathbf{s}$ from $BE$
36:            insert $\mathbf{s}$ in $CLOSED$
37:        end if
38:    end if
39:    UNLOCK
40: end procedure
```

A-ePA*SE extends w-ePA*SE with an additional outer control loop (Alg. 11) that sequentially reduces w. In the first iteration, ImprovePath is called with w_0. This is equivalent to running w-ePA*SE except for the algorithmic change described in technique 1. When ImprovePath returns, the current w-suboptimal solution is published (Line 13, Alg. 11). Before every subsequent call to ImprovePath, w is reduced by Δw and $OPEN$ is updated as described in technique 2. It is possible that no or very few states in $OPEN$ satisfy the termination check stated in technique 3 and ImprovePath returns right away or after a few expansions. This reusing of previous search effort is the fundamental source of efficiency gains for A-ePA*SE as compared to running w-ePA*SE from scratch with a reduced w. A-ePA*SE terminates when either 1) the time budget expires and the current best solution is returned or 2) ImprovePath finds a provably optimal solution with $w = 1$.

6.3 Properties

Theorem 10 *(**Anytime correctness**) Each time the ImprovePath function exits, the following holds: the cost of a greedy path from $\mathbf{s}_0$ to $\mathbf{s}_g$ is no larger than $\lambda g^*(\mathbf{s}_g)$, where $\lambda = \max(\epsilon, w)$.*

Theorem 11 *(**Anytime efficiency**) Within each call to ImprovePath a state $\mathbf{s}$ is expanded only if it was already locally inconsistent before the call to ImprovePath or its g-value was lowered during the current execution of ImprovePath.*

Proof sketch These properties were proved for each ImprovePath function call in ARA* (Corollary 13 & Theorem 2 in [59]) with $\lambda = w$. The anytime correctness properties are also proved for a single w-ePA*SE run (Theorem 8 in Chapter 5) with $\lambda = \max(\epsilon, w)$. Since we are inheriting the method to repair the graph and reuse the search effort of ARA*, these properties similarly follow for each ImprovePath function call in A-ePA*SE.

6.4 Evaluation

We evaluate A-ePA*SE on five scaled MovingAI 2D maps [60], with state space being 2D grid coordinates shown in Fig. 6.1. The agent has a square footprint with a side length of 32 units. The action space comprises moving along eight directions by 25 cell units. To check action feasibility, we collision-check the footprint at interpolated states with a 1-unit discretization. For each map, we sample 50 random start-goal

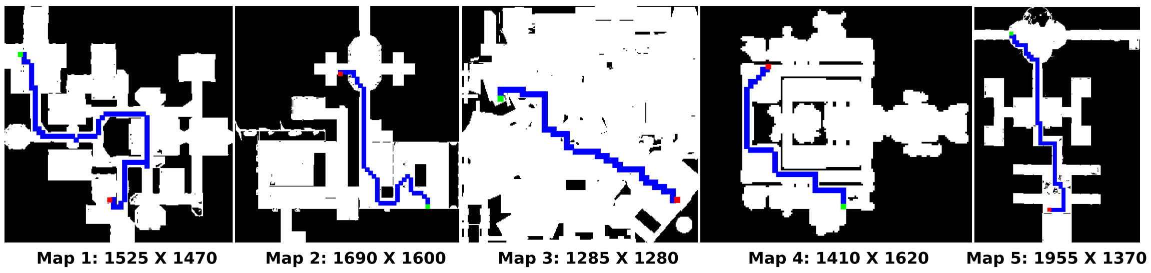


Figure 6.1: (**2D Grid World**) Scaled MovingAI maps with the start state shown in green, the goal state shown in red and the computed path shown in blue.

pairs and verify that there exists a solution by running wA* with a large timeout. All algorithms use Euclidean distance as the heuristic. We run the experiments with two cost maps: 1) Euclidean cost and 2) Euclidean cost multiplied with a random factor map generated by sampling a uniform distribution between 1 and 100. In the random cost map, there is a tendency for the solution to be improved more gradually with the decrease in w. In the case of Euclidean cost, the solution tends to improve only from one topology to another with w decrease, yielding less intermediate suboptimal solutions. We compare A-ePA*SE with three baselines: 1) ARA* 2) ePA*SE and 3) A-ePA*SE-naive which runs w-ePA*SE sequentially with decreasing w without reuse of previous search effort. For the anytime algorithms, w_0 is set to 50, and Δw is set to 0.5. All experiments were carried out on an AMD Threadripper Pro 5995WX workstation with a thread budget of 120. In all cases, we keep a high time budget, so none of the algorithms timeout.

Fig. 6.2 (top) shows the optimality ratio (optimal cost / actual cost) achieved by an anytime algorithm at a specific time point. For every problem, we calculate the optimality ratio at every time point when IMPROVEPATH returns. We then discretize time and assign each time point with the best optimality ratio achieved so far. This is then averaged across all maps and problems separately for the two different cost maps. To show the relative performance, Fig. 6.2 (bottom) divides the time it takes the baselines to achieve a given optimality ratio by the time of A-ePA*SE to achieve the same optimality ratio. Specifically, the plot represents how many times slower an algorithm is than A-ePA*SE in computing a solution with a certain optimality factor represented by the x-axis. We see that ARA* takes significantly longer to reach the same optimality ratio as compared to A-ePA*SE. A-ePA*SE-naive does

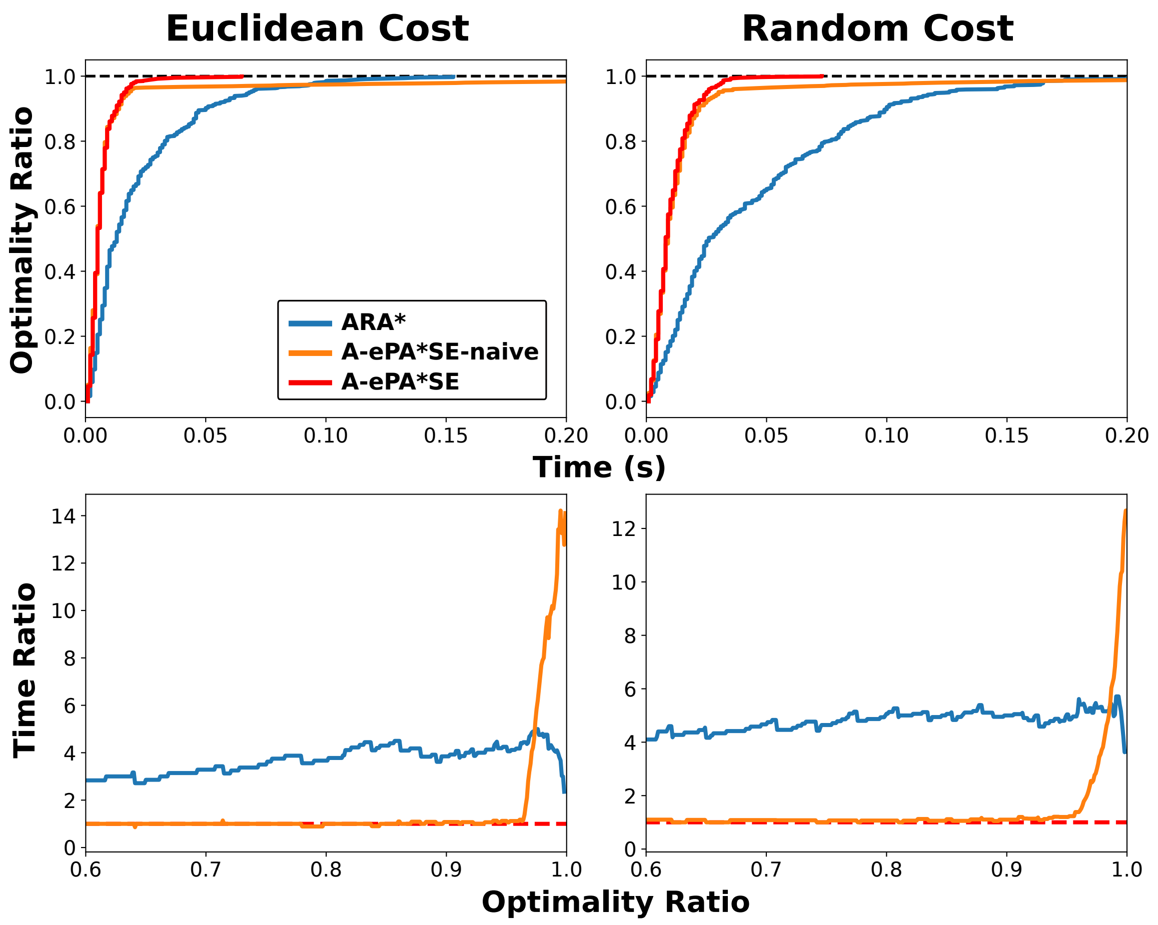


Figure 6.2: Top: The mean optimality ratio an algorithm achieves at a specific time point. Bottom: Planning times of the baselines divided by that of A-ePA*SE to achieve the same mean optimality ratio. In other words, for a specific algorithm, the bottom plots show how much slower (y-axis) than A-ePA*SE that algorithm gets to a solution that has a specific optimality factor (x-axis).

as good as A-ePA*SE for lower optimality ratios, but it takes significantly longer to achieve optimality because it does not reuse previous search effort. A-ePA*SE outperforms ARA* as predicted due to the efficiency gained from parallelization.

Table 6.1 top shows raw planning times for three stages. $\hat{t}_{init}$ is the mean time to generate the first solution, $\hat{t}_{opt}$ is the mean time to first discover the optimal solution in hindsight, and $\hat{t}_{term}$ is the mean time to provably generate the optimal solution by the final IMPROVEPATH call with $w = 1$. Table 6.1 bottom presents the average speedup of A-ePA*SE over the baselines ($t_{baseline}/t_{\text{A-ePA*SE}}$). This is generated by computing the speedup for each run and then averaging them over all runs and all maps. A-ePA*SE-naive and A-ePA*SE compute the initial solution faster than ARA* due to parallelization and than ePA*SE due to the high inflation on the heuristic. A-ePA*SE computes the provably optimal solution quicker than A-ePA*SE-naive and in ARA*, but slower than ePA*SE. This is expected since ePA*SE is not an anytime algorithm and runs a single optimal search. However, A-ePA*SE can discover the optimal solution in hindsight faster than ePA*SE. This means that even if the time budget runs out before the A-ePA*SE runs its final iteration with $w = 1$ to provably generate the optimal plan and the robot executes the best plan so far, it may still end up behaving optimally. This is an important and useful empirical result for real-time robotics.

Summary of results

The experimental evaluation demonstrates the advantages of A-ePA*SE over the baselines.

- Compared to ARA*, both Fig. 6.2 and Table. 6.1 indicate that A-ePA*SE outperforms ARA* in planning time.
- As shown in Table 6.1, A-ePA*SE and A-ePA*SE-naive both find the initial

	Euclidean Cost			Random Cost		
	$\hat{t}_{init}$	$\hat{t}_{opt}$	$\hat{t}_{term}$	$\hat{t}_{init}$	$\hat{t}_{opt}$	$\hat{t}_{term}$
ARA*	19 (0.923)	47	50	41 (0.902)	99	178
ePA*SE	11 (1.0)	11	11	38 (1.0)	38	38
A-ePA*SE-naive	6 (0.948)	159	200	10 (0.951)	396	767
A-ePA*SE	**6 (0.949)**	**14**	**16**	**10 (0.954)**	**27**	**44**
	$\hat{s}_{init}$	$\hat{s}_{opt}$	$\hat{s}_{term}$	$\hat{s}_{init}$	$\hat{s}_{opt}$	$\hat{s}_{term}$
ARA*	2.69	3.62	2.89	3.52	3.70	3.88
ePA*SE	1.75	1.00	0.70	4.17	1.82	0.86
A-ePA*SE-naive	0.98	9.19	11.94	1.01	13.12	16.44

Table 6.1: Top: Mean time (ms) to find the initial feasible solution ($\hat{t}_{init}$), discover optimal solution ($\hat{t}_{opt}$) and prove optimal solution ($\hat{t}_{term}$). Numbers in parenthesis in the $\hat{t}_{init}$ columns are the initial optimality ratios. Bottom: Speedup of A-ePA*SE over the baselines.

solution at around 0.95 optimality. However, Fig. 6.2 shows that A-ePA*SE improves the optimality ratio quicker than A-ePA*SE-naive in the 0.95-1.0 optimality region.

- Compared to ePA*SE, A-ePA*SE has anytime behavior where it quickly computes a feasible solution and then improves it over time. Additionally, it computes the optimal solution in hindsight ($\hat{t}_{opt}$) faster than ePA*SE, which is a useful insight in the real-time robotics context.

6.5 Conclusion

In this chapter, we presented an anytime version of ePA*SE termed A-ePA*SE. Our experiments demonstrated that A-ePA*SE achieves a significant speedup over ARA* in both computing an initial solution and then improving it to compute the optimal solution. Additionally, the anytime property of A-ePA*SE makes it potentially more useful than ePA*SE in a range of real-time robotics domains.

In the current formulation of A-ePA*SE, both the initial heuristic inflation w_0 and the decrement Δw between successive ImprovePath calls are parameters to be tuned. In the future, A-ePA*SE can be extended to a non-parametric formulation.

Chapter 7

GePA*SE: Generalized Edge-Based Parallel A* for Slow Evaluations

In Chapter 5, we presented a parallelized planning algorithm ePA*SE that changes the basic unit of search from state expansions to edge expansions. This decouples the evaluation of edges from the expansion of their common parent state, giving the search the flexibility to figure out what edges need to be evaluated to solve the planning problem. In domains with expensive edges, ePA*SE achieves lower planning times and evaluates fewer edges than PA*SE. ePA*SE is efficient in domains where the action space is homogenous in computational effort, i.e., all actions have similar evaluation times. In some domains, the action space can comprise a combination of cheap and expensive to evaluate actions. For the sake of concreteness, consider planning on manipulation lattices. The action space can comprise static primitives generated offline, each of which moves a single joint, and Adaptive Motion Primitives, which use an optimization-based IK solver to compute a valid goal configuration (based on the workspace goal) and then linearly interpolate from the expanded state to the goal [31]. The latter are generated online and are significantly more expensive to compute than the static primitives. In such domains, it is not efficient to delegate a new thread for every edge.

Motivated by these insights, we develop GePA*SE, which generalizes the key ideas in PA*SE and ePA*SE, i.e., state expansions and edge evaluations respectively. We show that GePA*SE outperforms both PA*SE and ePA*SE in domains with heterogenous action spaces by employing a parallelization strategy that handles cheap edges similar to PA*SE and expensive edges similar to ePA*SE. Additionally, it uses a more efficient strategy to carry out edge independence checks for large graphs. While GePA*SE is optimal, its bounded suboptimal variant w-GePA*SE inherits the bounded suboptimality guarantees of w-PA*SE and w-ePA*SE and achieves faster planning by employing an inflation factor on the heuristic. We evaluate w-GePA*SE in a 2D grid-world and a 7-DoF manipulation domain and demonstrate that it achieves lower planning times in both.

7.1 Problem Definition

Let a finite graph $G = (\mathcal{V}, \mathcal{E})$ be defined as a set of vertices $\mathcal{V}$ and directed edges $\mathcal{E}$. Each vertex $v \in \mathcal{V}$ represents a state $\mathbf{s}$ in the state space of the domain $\mathcal{S}$. An edge $e \in \mathcal{E}$ connecting two vertices v_1 and v_2 in the graph represents an action $\mathbf{a} \in \mathcal{A}$ that takes the agent from corresponding state $\mathbf{s}_1$ to $\mathbf{s}_2$. The action space is split into subsets of cheap ($\mathcal{A}^c$) and expensive actions ($\mathcal{A}^e$) s.t. $\mathcal{A}^c \cup \mathcal{A}^e = \mathcal{A}$ and corresponding cheap ($\mathcal{E}^c$) and expensive edges ($\mathcal{E}^e$) s.t. $\mathcal{E}^c \cup \mathcal{E}^e = \mathcal{E}$. An edge e can be represented as a pair $(\mathbf{s}, \mathbf{a})$, where $\mathbf{s}$ is the state at which action $\mathbf{a}$ is executed. For an edge e, we will refer to the corresponding state and action as $e.\mathbf{s}$ and $e.\mathbf{a}$ respectively. $\mathbf{s}_0$ is the start state, and $\mathcal{G}$ is the goal region. $c : \mathcal{E} \to [0, \infty]$ is the cost associated with an edge. $g(\mathbf{s})$ or g-value is the cost of the best path to $\mathbf{s}$ from $\mathbf{s}_0$ found by the algorithm so far. $h(\mathbf{s})$ is a consistent and therefore admissible heuristic [17]. A path π is an ordered sequence of edges $e_{i=1}^{N} = (\mathbf{s}, \mathbf{a})_{i=1}^{N}$, the cost of which is denoted as $c(\pi) = \sum_{i=1}^{N} c(e_i)$. The objective is to find a path π from $\mathbf{s}_0$ to a state in the goal region $\mathcal{G}$ with the optimal cost c^*. There is a computational budget of N_t parallel threads available. There exists a pairwise heuristic function $h(\mathbf{s}, \mathbf{s}')$ that provides an estimate of the cost between any pair of states. It is forward-backward consistent [20] i.e. $h(\mathbf{s}, \mathbf{s}'') \leq h(\mathbf{s}, \mathbf{s}') + h(\mathbf{s}', \mathbf{s}'')\ \forall\ \mathbf{s}, \mathbf{s}', \mathbf{s}''$ and $h(\mathbf{s}, \mathbf{s}') \leq c^*(\mathbf{s}, \mathbf{s}')\ \forall\ \mathbf{s}, \mathbf{s}'$.

7.2 Method

Similar to ePA*SE, GePA*SE searches over edges instead of states and exploits the notion of edge independence to parallelize this search. There are two key differences. First, GePA*SE handles cheap and expensive-to-evaluate edges differently. Second, it uses a more efficient independence check. In this section, we expand on these differences.

In A*, during a state expansion, all its successors are generated and are inserted/repositioned in the open list. In ePA*SE, the open list ($OPEN$) is a priority queue of edges (not states) that the search has generated but not expanded, where the edge with the smallest key/priority is placed in the front of the queue. The priority of an edge $e = (\mathbf{s}, \mathbf{a})$ in $OPEN$ is $f((\mathbf{s}, \mathbf{a})) = g(\mathbf{s}) + h(\mathbf{s})$. Expansion of an edge $(\mathbf{s}, \mathbf{a})$ involves evaluating the edge to generate the successor $\mathbf{s}'$ and adding/updating (but not evaluating) the edges originating from $\mathbf{s}'$ into $OPEN$ with the same priority of $g(\mathbf{s}') + h(\mathbf{s}')$. Henceforth, whenever $g(\mathbf{s}')$ changes, the positions of all of the outgoing edges from $\mathbf{s}'$ need to be updated in $OPEN$. To avoid this, ePA*SE replaces all the outgoing edges from $\mathbf{s}'$ by a single *dummy* edge $(\mathbf{s}', \mathbf{a^d})$, where $\mathbf{a^d}$ stands for a dummy action until the dummy edge is expanded. Every time $g(\mathbf{s}')$ changes, only the dummy edge has to be repositioned. Unlike what happens when a real edge is expanded, when the dummy edge $(\mathbf{s}', \mathbf{a^d})$ is expanded, it is replaced by the outgoing real edges from $\mathbf{s}'$ in $OPEN$. The real edges are expanded when they are popped from $OPEN$ by an edge expansion thread. This means that every edge gets delegated to a separate thread for expansion.

In contrast to ePA*SE, in GePA*SE, when the dummy edge $(\mathbf{s}, \mathbf{a^d})$ from $\mathbf{s}$ is expanded, the cheap edges from $\mathbf{s}$ are expanded immediately (Line 17, Alg. 15), i.e., the successors and costs are computed, and the dummy edges originating from the successors are inserted into $OPEN$. However, the expensive edges from $\mathbf{s}$ are not evaluated and are instead inserted into $OPEN$ (Line 13, Alg. 15). This means that the thread that expands the dummy edge also expands the cheap edges at the same time. This eliminates the overhead of delegating a thread for each cheap edge, improving the overall efficiency of the algorithm. The expensive edges are instead expanded when they are popped from $OPEN$ and are assigned to an edge evaluation thread. If $\mathcal{A}^e = \emptyset$, GePA*SE behaves the same as PA*SE, i.e., state expansions are

Algorithm 14 w-GePA*SE: Planning Loop

```
1: $terminate \leftarrow$ False
2: procedure PLAN
3:     $\forall \mathbf{s} \in G$, $\mathbf{s}.g \leftarrow \infty$, $n\_successors\_generated(s) = 0$
4:     $\mathbf{s}_0.g \leftarrow 0$
5:     insert $(\mathbf{s}_0, \mathbf{a^d})$ in $OPEN$                    ▷ Dummy edge from $\mathbf{s}_0$
6:     LOCK
7:     while not $terminate$ do
8:         if $OPEN = \emptyset$ and $BE = \emptyset$ then
9:             $terminate$ = True
10:            UNLOCK
11:            return $\emptyset$
12:        end if
13:        remove an edge $(\mathbf{s}, \mathbf{a})$ from $OPEN$ that has the
           smallest $f((\mathbf{s}, \mathbf{a}))$ among all states in $OPEN$ that
           satisfy Equations 5.2 and 5.4
14:        if such an edge does not exist then
15:            UNLOCK
16:            wait until $OPEN$ or $BE$ change
17:            LOCK
18:            continue
19:        end if
20:        if $\mathbf{s} \in \mathcal{G}$ then
21:            $terminate$ = True
22:            UNLOCK
23:            return BACKTRACK($\mathbf{s}$)
24:        end if
25:        UNLOCK
26:        while $(\mathbf{s}, \mathbf{a})$ has not been assigned a thread do
27:            for $i = 1 : N_t$ do
28:                if thread $i$ is available then
29:                    if thread $i$ has not been spawned then
30:                        Spawn EDGEEXPANDTHREAD($i$)
31:                    end if
32:                    Assign $(\mathbf{s}, \mathbf{a})$ to thread $i$
33:                end if
34:            end for
35:        end while
36:        LOCK
37:    end while
38:    $terminate$ = True
39:    UNLOCK
40: end procedure
```

parallelized and each thread evaluates all the outgoing edges from an expanded state sequentially. If $\mathcal{A}^c = \emptyset$, GePA*SE behaves the same as ePA*SE i.e. edge evaluations

Algorithm 15 w-GePA*SE: Edge Expansion

```
 1: procedure ExpandEdgeThread($i$)
 2:     while not $terminate$ do
 3:         if thread $i$ has been assigned an edge $(\mathbf{s}, \mathbf{a})$ then
 4:             Expand $((\mathbf{s}, \mathbf{a}))$
 5:         end if
 6:     end while
 7: end procedure
 8: procedure Expand($(\mathbf{s}, \mathbf{a})$)
 9:     if $\mathbf{a} = \mathbf{a^d}$ then
10:         insert $\mathbf{s}$ in $BE$ with priority $f(\mathbf{s})$
11:         for $\mathbf{a}' \in \mathcal{A}^e$ do
12:             $f((\mathbf{s}', \mathbf{a})) = g(\mathbf{s}) + wh(\mathbf{s})$
13:             insert $(\mathbf{s}, \mathbf{a})$ in $OPEN$ with $f((\mathbf{s}', \mathbf{a}))$
14:         end for
15:         UNLOCK
16:         for $\mathbf{a}' \in \mathcal{A}^c$ do
17:             ExpandEdge $((\mathbf{s}', \mathbf{a}))$
18:         end for
19:         LOCK
20:     else
21:         ExpandEdge $((\mathbf{s}, \mathbf{a}))$
22:     end if
23: end procedure
24: procedure ExpandEdge($(\mathbf{s}, \mathbf{a})$)
25:     $\mathbf{s}', c((\mathbf{s}, \mathbf{a})) \leftarrow$ GenerateSuccessor $((\mathbf{s}, \mathbf{a}))$
26:     LOCK
27:     if $\mathbf{s}' \notin CLOSED \cup BE$ and
28:             $g(\mathbf{s}') > g(\mathbf{s}) + c((\mathbf{s}, \mathbf{a}))$ then
29:         $g(\mathbf{s}') = g(\mathbf{s}) + c((\mathbf{s}, \mathbf{a}))$
30:         $\mathbf{s}'.parent = \mathbf{s}$
31:         $f\left((\mathbf{s}', \mathbf{a^d})\right) = g(\mathbf{s}') + wh(\mathbf{s}')$
32:         insert/update $(\mathbf{s}', \mathbf{a^d})$ in $OPEN$ with $f\left((\mathbf{s}', \mathbf{a^d})\right)$
33:     end if
34:     $n\_successors\_generated(\mathbf{s}) += 1$
35:     if $n\_successors\_generated(\mathbf{s}) = |\mathcal{A}|$ then
36:         remove $\mathbf{s}$ from $BE$ and insert in $CLOSED$
37:     end if
38:     UNLOCK
39: end procedure
```

are parallelized, and each thread expands a single edge at a time.

$$g(e.\mathbf{s}) - g(\mathbf{s}') \leq \epsilon h(\mathbf{s}', e.\mathbf{s}) \; \forall \mathbf{s}' \in BE \mid f(\mathbf{s}') < f(e) \tag{7.1}$$

In w-ePA*SE, the source states of the edges under expansion are stored in a set BE. However, in large graphs, BE can contain a large number of states, and performing the independence check against the entire set can get expensive. Therefore in w-GePA*SE, BE is a priority queue of states with priority $f(\mathbf{s}) = g(\mathbf{s}) + wh(\mathbf{s})$. To ensure the independence of an edge from all edges currently being expanded, it is sufficient to perform the independence check against only the states in BE that have a lower priority than the priority of the edge in $OPEN$ (Inequality 7.1). Independence of an edge e in $OPEN$ from a state $\mathbf{s}'$ in BE s.t. $f(e.\mathbf{s}) \leq f(\mathbf{s}')$ can be shown as follows:

$$
\begin{aligned}
&\implies g(e.\mathbf{s}) + wh(e.\mathbf{s}) \leq g(\mathbf{s}') + wh(\mathbf{s}') \\
&\implies g(e.\mathbf{s}) - g(\mathbf{s}') \leq w(h(\mathbf{s}') - h(e.\mathbf{s})) \\
&\implies g(e.\mathbf{s}) - g(\mathbf{s}') \leq wh(\mathbf{s}', e.\mathbf{s}) \leq \epsilon h(\mathbf{s}', e.\mathbf{s}) \\
&\qquad \text{(forward-backward consistency and } w \leq \epsilon)
\end{aligned}
$$

Beyond this, the bounded sub-optimality proof of w-GePA*SE is the same as that of w-ePA*SE [13] since the only other way in which w-GePA*SE differs from w-ePA*SE is in its parallelization strategy.

7.3 Evaluation

7.3.1 2D Grid World

We evaluate GePA*SE on five scaled MovingAI 2D maps with state space being 2D grid coordinates (Figure 6.1, Chapter 6). The agent has a square footprint with an edge length of 32 units. The action space comprises moving along eight directions by 25 cell units. To check for the feasibility of the actions, we collision-check the footprint at interpolated states with a 1-unit discretization. For 4 of the actions, we cache the footprint offline and apply the required offset for given state coordinates, which form the cheap actions set. For the remaining actions, we compute the footprint from scratch for every coordinate. Since footprint calculation is expensive, these actions form the expensive actions set. This difference in footprint computation simulates

2D Grid World ($r^c = 30$)	wA*	w-PA*SE			w-ePA*SE			w-GePA*SE		
Threads (N_t)	1	5	10	50	5	10	50	5	10	50
Mean Time (s)	0.81	0.41	0.27	0.17	0.18	0.08	0.02	0.13 (↓ 28%)	0.06 (↓ 25%)	0.02 (↓ 0%)
Edge Evaluations	522	1024	1513	3797	485	500	582	531	560	738
Mean Cost	901	885	902	959	958	980	988	956	959	969

2D Grid World ($r^c = 300$)	w-ePA*SE			w-GePA*SE		
Threads (N_t)	5	10	50	5	10	50
Mean Time (s)	1.65	0.74	0.15	1.13 (↓ 32%)	0.51 (↓ 31%)	0.12 (↓ 20%)
Edge Evaluations	484	494	534	518	529	700
Mean Cost	958	978	987	955	955	966

Table 7.1: (**2D Grid World**) Evaluation metrics for GePA*SE and the baselines with $r^c = 30$ (top) and $r^c = 30$ (bottom). The percentage reduction in planning time in w-GePA*SE from the best baseline based on mean planning time (colored blue) for the same thread budget is indicated with ↓.

the diversity in action computational effort.

On average, the ratio of computation time for the actions in $\mathcal{A}^e$ to those in $\mathcal{A}^c$ is $r^c = 30$. For each map, we sample 50 random start-goal pairs and verify that there exists a solution by running wA*. We compare w-GePA*SE against wA*, w-PA*SE and w-ePA*SE. All algorithms use Euclidean distance as the heuristic with an inflation factor of 50. We see that for smaller thread budgets, w-GePA*SE achieves >= 25% lower planning times than w-ePA*SE, which is the best baseline (Table 7.1). However, with $N_t = 50$, w-GePA*SE achieves identical performance to that of w-ePA*SE. This is because, in this domain, with a large number of available threads, there is no benefit of being selective about which edges should be expanded in parallel. Instead, parallelizing all edges like w-ePA*SE does is as good. However, with an additional increase in the computational cost of $\mathcal{A}^e$ by calling the footprint calculation in a loop 10 times ($r^c = 300$), w-GePA*SE achieves a 20% reduction in planning times from w-ePA*SE even for $N_t = 50$.

7.3.2 Manipulation

We also evaluate GePA*SE in a manipulation planning domain for a task of assembling a set of blocks on a table into a given structure by a PR2 (Figure 7.1 bottom).

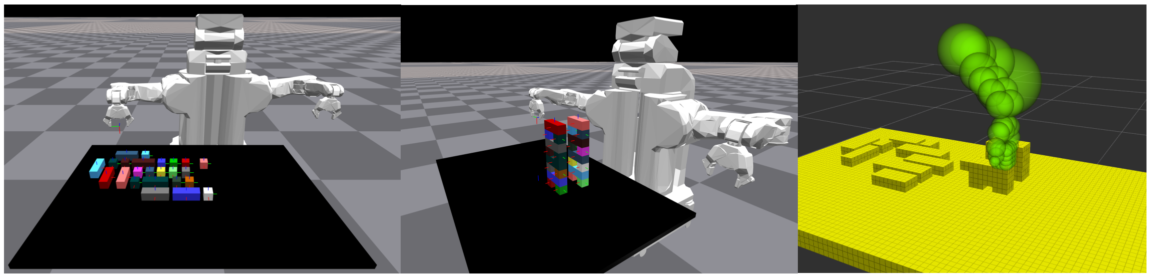


Figure 7.1: (**Manipulation**) The PR2 has to arrange a set of blocks on the table (left) into a given configuration (middle) with a motion planner (right) to compute PLACE actions.

Manipulation	wA*	w-PA*SE			w-ePA*SE			w-GePA*SE		
Threads	1	5	10	50	5	10	50	5	10	50
Success (%)	83	92	94	97	91	93	94	92	94	97
Mean Time (s)	0.36	0.25	0.19	0.16	0.25	0.18	0.12	0.19	0.12	0.09
								(↓ 24%|24%)	(↓ 37%|33%)	(↓ 44%|25%)

Table 7.2: (**Manipulation**) Evaluation metrics for GePA*SE and the baselines. The percentage reduction in mean planning time in w-GePA*SE from w-PA*SE and w-ePA*SE for the same thread budget is indicated with ↓.

We collect a problem set of PLACE actions for 40 assembly tasks in each of which the blocks are arranged in random order on the table. PLACE requires a motion planner internally to compute collision-free trajectories for the 7-DoF right arm of the PR2 in a cluttered workspace to place the blocks at their desired pose. $\mathcal{A}^c$ comprises 22 static motion primitives that move one joint at a time by 4 or 7 degrees in either direction. $\mathcal{A}^e$ comprises a single dynamically generated primitive that attempts to connect the expanded state to a goal configuration ($r^c = 20$). This primitive involves solving an optimization-based IK problem to find a valid configuration space goal and then collision checking of a linearly interpolated path from the expanded state to the goal state. All primitives have a uniform cost. A backward BFS in the workspace (x, y, z) is used to compute the heuristic with an inflation factor of $w = \epsilon = 100$. For the problem set generation, we use w-GePA*SE with six threads and a large timeout. We then evaluate all the algorithms with different thread budgets on this problem set with a timeout of 2 s. In the computation of the metrics (Table 7.2), we only consider the set of problems that are successfully solved and lead to a path length

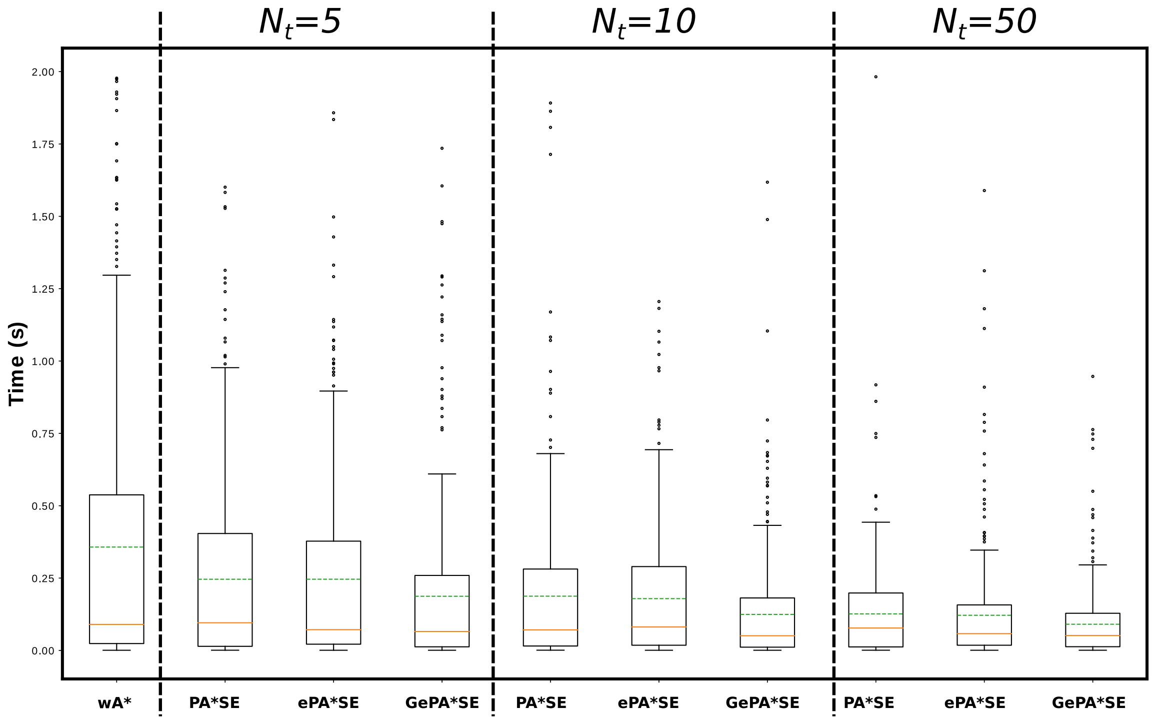


Figure 7.2: Box plot for planning time for GePA*SE and the baselines for several thread budgets, with the median indicated with an orange line, and the mean indicated with a dashed green line.

longer than two states for all algorithms. This is needed to not skew the statistics by the cases where the IK-based primitive connects the start state directly to the goal without any meaningful planning effort. We see that w-GePA*SE consistently achieves the lowest mean planning times while maintaining a high success rate across all thread budgets.

7.4 Conclusion

In this chapter, we presented GePA*SE, a generalized formulation of two parallel search algorithms i.e. PA*SE and ePA*SE for domains with action spaces comprising a mix of cheap and expensive to evaluate actions. We showed that by employing different parallelization strategies for edges based on the computation effort required to evaluate them, GePA*SE achieves higher efficiency on several planning domains.

‘

Chapter 8

Edge-Based Parallelization of INSAT

Trajectory optimization is a powerful tool for motion planning in high-dimensional state spaces and under differential constraints. However, long time horizons and planning around obstacles in non-convex spaces make them suffer from local minima, increased time complexity, and lack of convergence guarantees. As a result, discrete graph search planners and sampling-based planers are preferred when facing obstacle-cluttered environments. A recently developed algorithm called INSAT [12, 32] effectively combines graph search in the low-dimensional subspace and trajectory optimization in the full-dimensional space for kinodynamic planning with global reasoning in large environments and planning with contact. Although INSAT could reason about and solve complex planning problems, its planning times were large because of the several expensive calls to an optimizer, limiting its practical use. This chapter shows that systematic parallelization of INSAT using ePA*SE can achieve drastically lower planning times and higher success rates. We refer to this algorithm as PINSAT: Parallelized Interleaving of Search and Trajectory Optimization and demonstrate its utility in a kinodynamic motion planning problem for a 6 DoF manipulator.

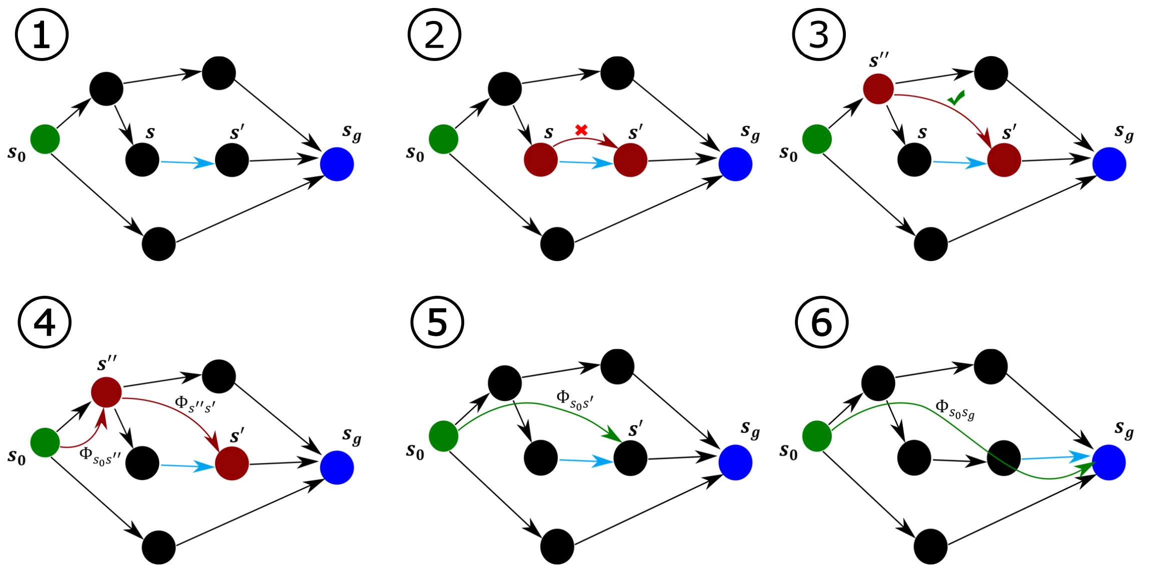


Figure 8.1: (1) When the edge connecting $\mathbf{s}$ and $\mathbf{s}'$ is expanded, PINSAT runs a trajectory optimizer to compute a trajectory to $\mathbf{s}'$ from its ancestors, starting with $\mathbf{s}$. (2) If the trajectory optimization from $\mathbf{s}$ fails, (3) it retries with the next ancestor $\mathbf{s}''$. (4) If it succeeds in generating this trajectory ($\Phi_{\mathbf{s}''\mathbf{s}'}$), a new trajectory optimization is attempted from the start state $\mathbf{s}_0$ to $\mathbf{s}$. This optimization us initialized using the trajectory from $\mathbf{s}_0$ to $\mathbf{s}''$ ($\Phi_{\mathbf{s}_0\mathbf{s}''}$) and the previously generated trajectory from $\mathbf{s}''$ to $\mathbf{s}'$ ($\Phi_{\mathbf{s}''\mathbf{s}'}$). (4=5) If successful, the resulting trajectory $\Phi_{\mathbf{s}_0\mathbf{s}'}$ is stored in $\mathbf{s}'$. (6) This process is repeated for every edge expansion until the goal is reached and a trajectory from the $\mathbf{s}_0$ to $\mathbf{s}_g$ is generated.

8.1 Method

PINSAT interleaves trajectory optimization with the expansion of an edge in ePA*SE as illustrated in Figure 8.1. When an edge $(\mathbf{s}, \mathbf{a})$ is expanded in the search to generate a successor $\mathbf{s}'$, PINSAT attempts to compute a trajectory to $\mathbf{s}'$ from its closest ancestor. If the closest ancestor from which a trajectory is successfully generated is $\mathbf{s}''$, PINSAT blends this trajectory $\Phi_{\mathbf{s}''\mathbf{s}'}$ with the trajectory from the start state to $\mathbf{s}''$. This trajectory, $\Phi_{\mathbf{s}_0\mathbf{s}''}$, was generated when $\mathbf{s}''$ was generated at some earlier point in the search. This blended trajectory, i.e., $\Phi_{\mathbf{s}_0\mathbf{s}''} \cup \Phi_{\mathbf{s}''\mathbf{s}'}$, is then used to warm state another trajectory optimization from $\mathbf{s}_0$ to $\mathbf{s}'$. If successful, this trajectory $\Phi_{\mathbf{s}_0\mathbf{s}'}$ is stored in $\mathbf{s}'$ (Line 25, Alg. 17). The cost of this trajectory $J_{total}(\Phi_{\mathbf{s}_0\mathbf{s}'})$ is used to improve the g-value of $\mathbf{s}'$ (Line 23, Alg. 17). When the goal is generated, a trajectory

$\Phi_{\mathbf{s}_0\mathbf{s}_g}$ is similarly computed from the start. When the dummy edge from the goal is popped from $OPEN$, the trajectory is returned as the solution to the planning problem (Line 24, Alg. 16). Since every edge expansion in PINSAT also involves several trajectory optimizations besides the generation of the successor, running it on a single thread is prohibitively expensive. However, because of ePA*SE style parallelization of edge expansions, PINSAT is able to overcome this limitation, as we will show in the next section. In w-ePA*SE, to maintain bounded sub-optimality, an edge can only be expanded if it is independent of all edges ahead of it in $OPEN$ and the edges currently being expanded. However, since INSAT and, therefore, PINSAT are not bounded sub-optimal algorithms, the expensive independence check can be eliminated. Additionally, this increases the amount of parallelization that can be achieved.

Algorithm 16 PINSAT: Planning Loop

```
1: $\mathcal{A} \leftarrow$ action space , $N_t \leftarrow$ number of threads, $G \leftarrow \emptyset$
2: $\mathbf{s}_0 \leftarrow$ start state , $\mathcal{G} \leftarrow$ goal region, $terminate \leftarrow$ False
3: procedure Plan
4:     $\forall \mathbf{s} \in G$, $\mathbf{s}.g \leftarrow \infty$, $n\_successors\_generated(s) = 0$
5:     $\mathbf{s}_0.g \leftarrow 0$
6:     insert $(\mathbf{s}_0, \mathbf{a^d})$ in $OPEN$                ▷ Dummy edge from $\mathbf{s}_0$
7:     LOCK
8:     while not $terminate$ do
9:         if $OPEN = \emptyset$ and $BE = \emptyset$ then
10:            $terminate$ = True
11:            UNLOCK
12:            return $\emptyset$
13:        end if
14:        $(\mathbf{s}, \mathbf{a}) \leftarrow OPEN.min()$
15:        if such an edge does not exist then
16:            UNLOCK
17:            wait until $OPEN$ or $BE$ change
18:            LOCK
19:            continue
20:        end if
21:        if $\mathbf{s} \in \mathcal{G}$ then
22:            $terminate$ = True
23:            UNLOCK
24:            return $\mathbf{s}.traj$
25:        else
26:            UNLOCK
27:            while $(\mathbf{s}, \mathbf{a})$ has not been assigned a thread do
28:                for $i = 1 : N_t$ do
29:                    if thread $i$ is available then
30:                        if thread $i$ has not been spawned then
31:                            Spawn EdgeExpandThread($i$)
32:                        end if
33:                        Assign $(\mathbf{s}, \mathbf{a})$ to thread $i$
34:                    end if
35:                end for
36:            end while
37:            LOCK
38:        end if
39:    end while
40:    $terminate$ = True
41:    UNLOCK
42: end procedure
```

Algorithm 17 PINSAT: Edge Expansion and Trajectory Optimization

```
 1: procedure EdgeExpandThread($i$)
 2:     while not $terminate$ do
 3:         if thread $i$ has been assigned an edge $(\mathbf{s}, \mathbf{a})$ then
 4:             Expand $((\mathbf{s}, \mathbf{a}))$
 5:         end if
 6:     end while
 7: end procedure
 8: procedure Expand($(\mathbf{s}, \mathbf{a})$)
 9:     LOCK
10:     if $\mathbf{a} = \mathbf{a^d}$ then
11:         insert $\mathbf{s}$ in $BE$
12:         for $\mathbf{a} \in \mathcal{A}$ do
13:             $f((\mathbf{s}, \mathbf{a})) = g(\mathbf{s}) + h(\mathbf{s})$
14:             insert $(\mathbf{s}, \mathbf{a})$ in $OPEN$ with $f((\mathbf{s}, \mathbf{a}))$
15:         end for
16:     else
17:         UNLOCK
18:         $\mathbf{s}', c((\mathbf{s}, \mathbf{a})) \leftarrow$ GenerateSuccessor $((\mathbf{s}, \mathbf{a}))$
19:         LOCK
20:         if $\mathbf{s}' \notin CLOSED \cup BE$ then
21:             $\Phi_{\mathbf{s_0 s'}}$ = GenerateTrajectory($\mathbf{s}, \mathbf{s}'$)
22:             if $\Phi_{\mathbf{s_0 s'}}$ is not NULL and $g(\mathbf{s}') > J_{total}(\Phi_{\mathbf{s_0 s'}})$ then
23:                 $g(\mathbf{s}') = J_{total}(\Phi_{\mathbf{s_0 s'}})$
24:                 $\mathbf{s}'.parent = \mathbf{s}$
25:                 $\mathbf{s}'.traj = \Phi_{\mathbf{s_0 s'}}$
26:                 $f\left((\mathbf{s}', \mathbf{a^d})\right) = g(\mathbf{s}') + h(\mathbf{s}')$
27:                 insert/update $(\mathbf{s}', \mathbf{a^d})$ in $OPEN$ with $f\left((\mathbf{s}', \mathbf{a^d})\right)$
28:             end if
29:         end if
30:         $n\_successors\_generated(\mathbf{s}) += 1$
31:         if $n\_successors\_generated(\mathbf{s}) = |\mathcal{A}|$ then
32:             remove $\mathbf{s}$ from $BE$
33:             insert $\mathbf{s}$ in $CLOSED$
34:         end if
35:     end if
36:     UNLOCK
37: end procedure
38: procedure GenerateTrajectory($\mathbf{s}, \mathbf{s}'$)
39:     for $\mathbf{s}'' \in$ Ancestors($\mathbf{s}$) $\cup\, \mathbf{s}$ do                ▷ From $\mathbf{s}$ to $\mathbf{s}_0$
40:         $\Phi_{\mathbf{s''s'}}$ = Optimize($\mathbf{s}'', \mathbf{s}'$)
41:         if $\Phi_{\mathbf{s''s'}}$ is collision free then
42:             $\Phi_{\mathbf{s_0 s'}}$ = WarmOptimize($\Phi_{\mathbf{s_0 s''}}, \Phi_{\mathbf{s''s'}}$)
43:             return $\Phi_{\mathbf{s_0 s'}}$
44:         end if
45:         return NULL
46:     end for
47: end procedure
```

8.2 Evaluation

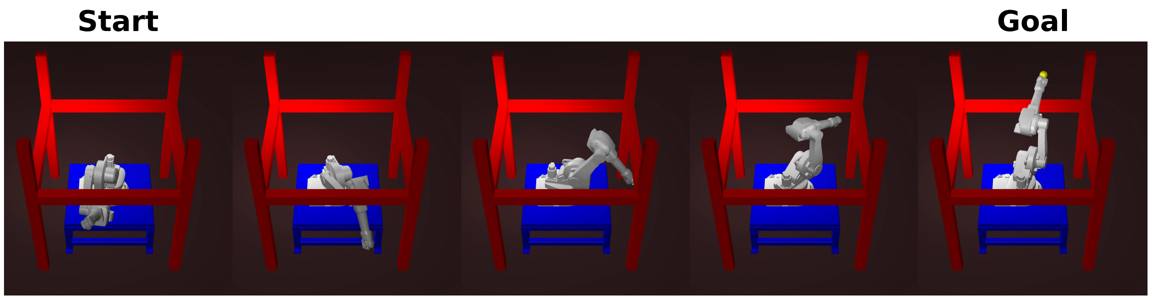


Figure 8.2: Kinodynamic motion planning for an ABB arm. Figure (left to right) shows waypoints along a plan.

	INSAT	PINSAT			
Threads (N_t)	1	1	10	60	120
Success (%)	38	37	56	69	68
Mean | Median Planning Time (s)	4.21 | 2.28	3.58 | 1.83	0.86 | 0.43	0.53 | 0.28	0.60 | 0.47

Table 8.1: Statistics for planning time and success rate for INSAT and PINSAT with a timeout of 20s for several thread budgets..

We evaluate PINSAT in a kinodynamic manipulation planning for an ABB robot shown in Figure 8.2. The red horizontal and vertical bars are obstacles and divide the region around the robot into eight quadrants, four below the horizontal bars and four above. We randomly sample 500 hard start and goal configurations. We do so by first ensuring that the end effector corresponding to one of those configurations is above the horizontal bars and the other is underneath. We also ensure that the end effector for the start and goal configurations are not sampled in the region enclosed by the same two adjacent vertical bars. For the trajectory optimization, we use a B-spline-based optimizer that respects the velocity, acceleration, and jerk constraints for the robot. However, we inflate the velocity limit of the ABB by a factor of 10 for every joint.

Table 8.1 and Figure 8.3 show success rate and planning time statistics for INSAT and PINSAT for different thread budgets. The planning time statistics were only computed over the problems that were successfully solved by both algorithms for a

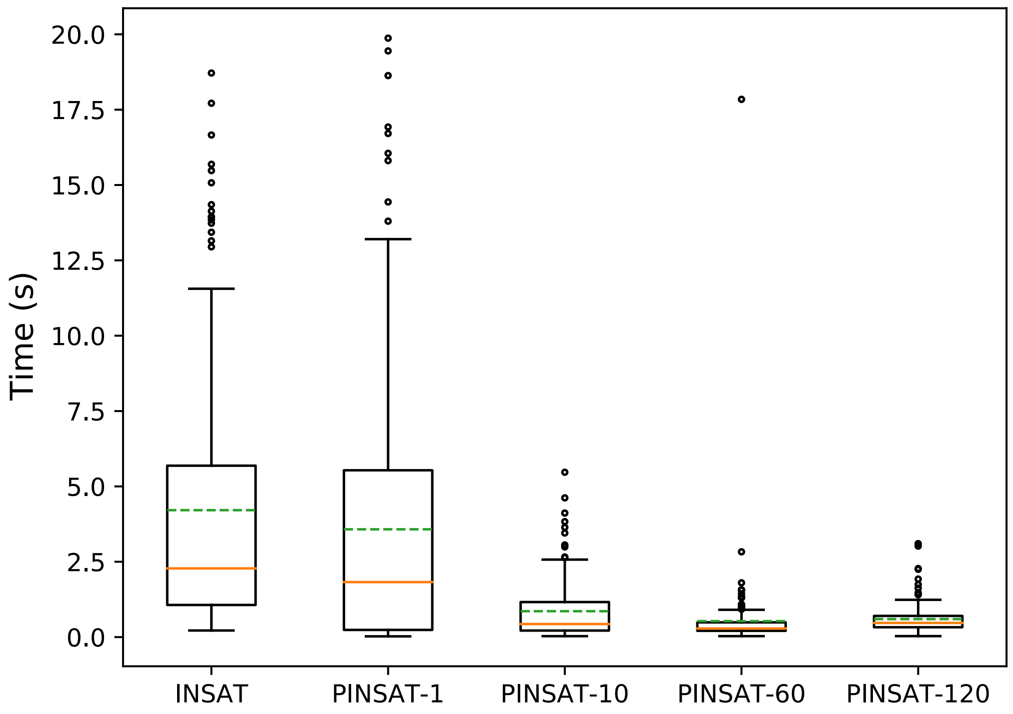


Figure 8.3: Box plot for planning time for INSAT and PINSAT for several thread budgets, with the median indicated with an orange line and the mean indicated with a dashed green line.

given thread budget. We also implemented a parallelized version of RRTConnect [28] as a baseline. The kinematic plan generated by RRTConnect was then used to initialize a trajectory optimizer to compute the final kinodynamic plan. Though RRT-Connect had a 100% success rate in computing a kinematic trajectory, the trajectory optimizer had a success rate of 1%. Therefore, we did not include this baseline in Table 8.1. PINSAT achieves a significantly higher success rate than INSAT for all thread budgets greater than 1. Specifically, it achieves a 7x improvement in mean planning time, a 9x improvement in median planning time, and a 1.8x improvement in success rate for $N_t = 60$. Even with a single thread, PINSAT achieves lower planning times than INSAT. This is because of the decoupling of edge evaluations from state expansions in ePA*SE which was explained in Chapter 5.

8.3 Conclusion

In this Chapter, we presented PINSAT, which parallelizes the interleaving of search and trajectory optimization using the ideas of ePA*SE developed in Chapter 5. In a kinodynamic manipulation planning domain, PINSAT achieved significantly higher success rates and a drastic reduction in planning time.

Chapter 9

Conclusion and Future Work

There is significant scope for further research in parallelized search algorithms. To conclude this thesis, we will suggest some future research directions that we think are worth exploring.

9.1 Conclusion

In this thesis, we developed a family of domain-agnostic search algorithms that utilize parallelization to solve planning problems in robotics efficiently. They do so by exploiting the characteristic feature of expensive to evaluate actions in robotics.

- In Chapter 4, we presented MPLP that searches the graph lazily and delegates the evaluation of discovered edges to a pool of parallel threads.
- In Chapter 5, we presented ePA*SE that interleaves the search and parallel evaluation of edges.
- In Chapter 6, we presented A-ePA*SE that brings the anytime property to ePA*SE.
- In Chapter 7, we presented GePA*SE that extends ePA*SE to domains with action spaces that are heterogenous in the computational effort required for their evaluation. It does so by pooling together cheap-to-compute edges and delegating their evaluation to the same thread while delegating the expensive edges to independent threads.

On the theoretical front, we proved that all the algorithms provide rigorous guarantees of completeness and bounded suboptimality. On the practical front, we evaluated them on several different types of task and motion planning domains and demonstrated that incorporating parallelization can drastically reduce planning times in robotics. In particular, in Chapter 8, we demonstrated the benefit of parallelization in INSAT, which is an algorithm that interleaves search with computationally expensive trajectory optimization. The resulting algorithm, PINSAT, achieves significantly higher success rates and an order of magnitude improvement in planning time statistics over INSAT.

9.1.1 A Practitioner's Guide to Parallel Search

To condense the ideas developed in this thesis into an easy-to-interpret format, Figure 9.1 provides a guide for practitioners to pick the right algorithm for their planning domain. Given a planning domain, the first question to ask is if edge evaluations are expensive. If they are not, the algorithms developed in this thesis are less applicable, and PA*SE would likely be the ideal choice. If they are, the second question to ask is if there is a way to generate successors lazily and if a good optimistic estimate of the edge costs is available. If the answer is yes, MPLP is the ideal choice. If lazy successor generation is not possible or if a good optimistic estimate of the edge cost is not available, the third question to ask is if anytime performance is required. If yes, A-ePA*SE is the appropriate algorithm. If anytime performance is not required, the fourth question to ask does the domain comprises a mix of cheap and expensive edges. If yes, GePA*SE is the appropriate choice, and if not, ePA*SE is the algorithm to go with.

9.2 Discussion and Future Work

9.2.1 MPLP with Parallelized Successors Generation

In MPLP, the size of the optimistic graph and, consequently, the number of edges that need evaluation depends on how optimistic the successor generation is. If the successor generation is extremely optimistic, the graph size can increase dramatically,

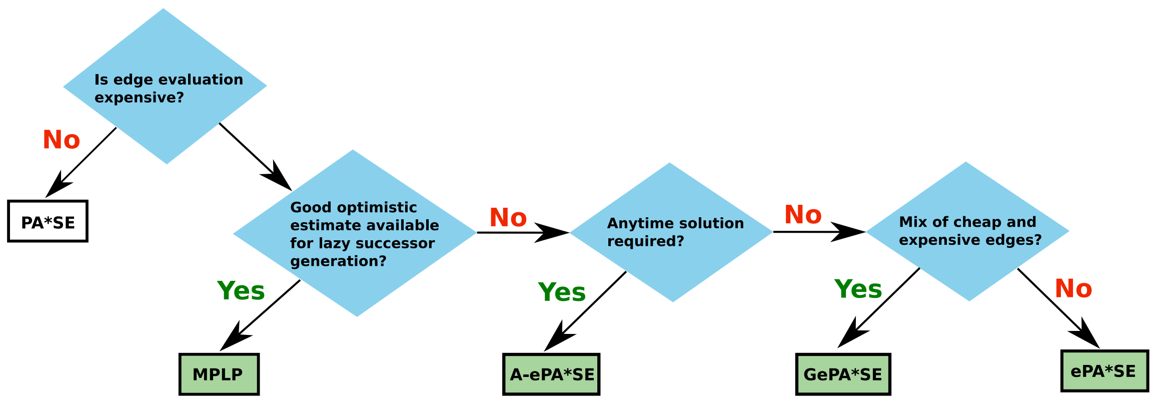

Is edge evaluation expensive?
No
PA*SE
Good optimistic estimate available for lazy successor generation?
Yes
MPLP
No
Anytime solution required?
Yes
A-ePA*SE
No
Mix of cheap and expensive edges?
Yes
GePA*SE
No
ePA*SE

Figure 9.1: A practitioner's guide to parallel search. The algorithms highlighted in green have been developed in this thesis.

resulting in an overall worse planning time as compared to wA*. Therefore, there is a case to be made in expending some computation in generating meaningful successors. An example is the assembly domain used in this thesis. One of the controllers (Place) uses a motion planner to compute a trajectory and then a simulator to simulate the outcome of placing a block in its desired pose. The simulator is required to compute the final positions of the blocks which are a part of the state. Since the generation of the state depends on the simulator, it needs to be a part of the optimistic controller. Considering the variability in the simulation speed and the overhead of communicating with the simulator, this can take a wall time of 1 s. Therefore, if the simulation can be parallelized, the optimistic search itself can be sped up, leading to overall faster planning.

In such domains, where the optimistic search takes a non-trivial amount of time, ePA*SE can be used to parallelize it. MPLP and ePA*SE use fundamentally different parallelization strategies. MPLP searches the graph lazily while evaluating edges in parallel but relies on the assumption that states can be generated lazily without evaluating edges. On the other hand, ePA*SE evaluates edges in a way that preserves optimality without the need for state (and edge) re-expansions but does not rely on lazy state generation. Therefore, in domains where states can be generated lazily, however, the lazy state generation takes a non-trivial amount of time, and these two different parallelization strategies can be combined.

9.2.2 Parallelization of Search on the GPU

In Chapter 3, we discussed the limitations of GPUs because of their SIMD execution model. Though GPUs are limited by their inability to parallelize heterogenous instructions, they are very good at large-scale parallelization of similar instructions. In several planning domains, this can be exploited by delegating the evaluation of actions that share the same code to GPU threads and the evaluation of the remaining actions to CPU cores. This idea can be particularly useful for domains that use GPU-parallelized simulators like Isaac Gym [58] as models for action evaluations.

9.2.3 Bounded Planning Time with Infinite Threads

The performance of ePA*SE generally increases with an increasing number of threads up until it saturates. The thread budget at the saturation point can be thought of as the maximum number of threads the search can utilize at any point in the search. The number of threads the search can utilize is limited by the independence check that is required to guarantee optimality. If optimality is not desired, the independence check can be eliminated and the number of edges available for expansion will be limited by the rate at which they are discovered by the search and the thread budget. In the presence of infinite threads, it is likely possible to express the planning time as a function of the quickest-to-evaluate path from the start to the goal. Without a lazy search that discovers edges quicker, this is the lower bound on the planning time, and theoretical analysis of this bound is likely to provide interesting insights. It will also be interesting to analyze how the bound changes with lazy successor generation.

9.2.4 Going Beyond Domains with Expensive Edges

The focus of this thesis has been on domains with expensive-to-compute edges. Beyond that, parallelization can also be a useful tool for domains where heuristic computations are expensive. For example, in many task planning heuristics, the heuristic value of a state is the solution to a relaxed version of the problem from the state [61]. Solving the relaxed version of the problem, though significantly cheaper than solving the original problem, takes a considerable amount of computation. The price, however, is worth paying since these heuristics guide the search in a state space that grows exponentially with the number of literals. In such domains, parallelization of heuristic computation can be extremely useful. Another type of domain that could benefit from the parallelization of open list operations is where the graph size is large, like in the case of domains with high branching factors. In such cases, the open list can grow exceedingly large in size, and therefore, parallelization of operations like insertion, removal, and rebalancing could be potentially useful.

9.2.5 Parallelization of Other Search Algorithms

The ideas of edge-based parallelization developed in this thesis can be employed in a variety of search algorithms that exploit specific characteristics of the domain to speed up planning. One such algorithm is Multi-Heursitic A* (MHA*) [3], which integrates a consistent and admissible heuristic with any number of arbitrary inadmissible heuristics to improve performance in domains where a single heuristic is unable to capture all the complexities of the domain. Another example is Multi-Resolution A* (MRA*) [7], which searches over multiple resolutions of the state space to combine the benefits of reduced search effort in coarser resolutions and maneuverability in finer resolutions. Yet another example is Anytime Multi-Resolution Multi-Heuristic A* [54] that combines the ideas of MHA* and MRA*. These algorithms are primarily used in robotics domains and, therefore, typically on graphs with expensive edges. There are also more general search algorithms that could potentially benefit from edge-based parallelization like Iterative Deepening A* [62], Beam Search, Bidirectional Search [63] and Multi-objective Search [64, 65, 66, 67].

Bibliography

[1] P. E. Hart, N. J. 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. 1.1, 2.5, 3.1.2

[2] I. Pohl, “Heuristic search viewed as path finding in a graph,” *Artificial intelligence*, vol. 1, no. 3-4, pp. 193–204, 1970. 1.1, 2.6, 3.1.2

[3] S. Aine, S. Swaminathan, V. Narayanan, V. Hwang, and M. Likhachev, “Multi-heuristic A*,” *The International Journal of Robotics Research*, vol. 35, no. 1-3, pp. 224–243, 2016. 1.1, 1.1, 3.1.2, 3.4, 9.2.5

[4] T. Kusnur, S. Mukherjee, D. M. Saxena, T. Fukami, T. Koyama, O. Salzman, and M. Likhachev, “A planning framework for persistent, multi-UAV coverage with global deconfliction,” in *Field and Service Robotics*. Springer, 2021, pp. 459–474. 1.1, 1.4

[5] S. Mukherjee, S. Aine, and M. Likhachev, “MPLP: Massively parallelized lazy planning,” *IEEE Robotics and Automation Letters*, vol. 7, no. 3, pp. 6067–6074, 2022. 1.1, 1.2.1

[6] K. Rupp. (2018) 42 years of microprocessor trend data. [Online]. Available: https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/ 1.2

[7] W. Du, F. Islam, and M. Likhachev, “Multi-resolution A*,” in *Proceedings of the International Symposium on Combinatorial Search*, vol. 11, no. 1, 2020, pp. 29–37. 1.1, 9.2.5

[8] H. Sutter *et al.*, “The free lunch is over: A fundamental turn toward concurrency in software,” *Dr. Dobb’s journal*, vol. 30, no. 3, pp. 202–210, 2005. 1.1

[9] M. P. Das, D. M. Conover, S. Eum, H. Kwon, and M. Likhachev, “MA3: Model-accuracy aware anytime planning with simulation verification for navigating complex terrains,” in *Proceedings of the International Symposium on Combinatorial Search*, vol. 15, no. 1, 2022, pp. 65–73. 1.1

[10] M. S. Saleem and M. Likhachev, “Planning with selective physics-based simulation for manipulation among movable objects,” in *2020 IEEE International*

Conference on Robotics and Automation (ICRA). IEEE, 2020, pp. 6752–6758. 1.1

[11] R. Natarajan, M. Saleem, S. Aine, M. Likhachev, and H. Choset, "A-MHA*: Anytime multi-heuristic A*," in *Proceedings of the International Symposium on Combinatorial Search*, vol. 10, no. 1, 2019, pp. 192–193. 1.1, 3.4

[12] R. Natarajan, H. Choset, and M. Likhachev, "Interleaving graph search and trajectory optimization for aggressive quadrotor flight," *IEEE Robotics and Automation Letters*, vol. 6, no. 3, pp. 5357–5364, 2021. 1.1, 8

[13] S. Mukherjee, S. Aine, and M. Likhachev, "ePA*SE: Edge-based parallel A* for slow evaluations," in *International Symposium on Combinatorial Search*, vol. 15, no. 1. AAAI Press, 2022, pp. 136–144. 1.2.2, 7.2

[14] H. Yang, S. Mukherjee, and M. Likhachev, "A-ePA*SE: Anytime edge-based parallel A* for slow evaluations," in *International Symposium on Combinatorial Search*. AAAI Press, 2023. 1.2.3

[15] S. Mukherjee and M. Likhachev, "GePA*SE: Generalized edge-based parallel A* for slow evaluations," in *International Symposium on Combinatorial Search*. AAAI Press, 2023. 1.2.4

[16] S. Mukherjee, C. Paxton, A. Mousavian, A. Fishman, M. Likhachev, and D. Fox, "Reactive long horizon task execution via visual skill and precondition models," in *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2021, pp. 5717–5724. 1.4, 3.1.2

[17] S. J. Russell, *Artificial intelligence a modern approach*. Pearson Education, Inc., 2010. 2.1, 5.1, 6.1, 7.1

[18] J. Liang, M. Sharma, A. LaGrassa, S. Vats, S. Saxena, and O. Kroemer, "Search-based task planning with learned skill effect models for lifelong robotic manipulation," in *2022 International Conference on Robotics and Automation (ICRA)*. IEEE, 2022, pp. 6351–6357. 2.2, 3.1.2, 3.2.1, 5

[19] M. Likhachev, G. J. Gordon, and S. Thrun, "ARA*: Anytime A* with provable bounds on sub-optimality," *Advances in neural information processing systems*, vol. 16, pp. 767–774, 2003. 2.6, 3.4, 6.2, 1

[20] M. Phillips, M. Likhachev, and S. Koenig, "PA* SE: Parallel A* for slow expansions," in *Twenty-Fourth International Conference on Automated Planning and Scheduling*, 2014. 2.7, 2.7, 3.2.2, 4.4.1, 5.2.4, 5.3, 5.4.1, 6.1, 7.1

[21] K. Irani and Y.-f. Shih, "Parallel A* and AO* algorithms- an optimality criterion and performance evaluation," in *1986 International Conference on Parallel Processing, University Park, PA*, 1986, pp. 274–277. 2.7, 3.2.2, 4.4.1, 5.4.1

[22] Y. Zhou and J. Zeng, "Massively parallel A* search on a GPU," in *Proceedings*

of the AAAI Conference on Artificial Intelligence, 2015. 2.7, 3.2.3

[23] X. He, Y. Yao, Z. Chen, J. Sun, and H. Chen, "Efficient parallel A* search on multi-GPU system," *Future Generation Computer Systems*, vol. 123, pp. 35–47, 2021. 2.7, 3.2.3

[24] 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. 3.1.1

[25] 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. 3.1.1

[26] R. Bohlin and L. E. Kavraki, "Path planning using lazy PRM," in *Proceedings 2000 ICRA. Millennium conference. IEEE international conference on robotics and automation. Symposia proceedings (Cat. No. 00CH37065)*, vol. 1. IEEE, 2000, pp. 521–528. 3.1.1

[27] S. M. LaValle and J. J. Kuffner, "Rapidly-exploring random trees: Progress and prospects: Steven m. lavalle, iowa state university, a james j. kuffner, jr., university of tokyo, tokyo, japan," *Algorithmic and computational robotics*, pp. 303–307, 2001. 3.1.1

[28] J. J. Kuffner and S. M. LaValle, "RRT-connect: An efficient approach to single-query path planning," in *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No. 00CH37065)*, vol. 2. IEEE, 2000, pp. 995–1001. 3.1.1, 8.2

[29] N. T. Dantam, Z. K. Kingston, S. Chaudhuri, and L. E. Kavraki, "An incremental constraint-based framework for task and motion planning," *The International Journal of Robotics Research*, vol. 37, no. 10, pp. 1134–1151, 2018. 3.1.2

[30] C. Paxton, Y. Barnoy, K. Katyal, R. Arora, and G. D. Hager, "Visual robot task planning," in *2019 international conference on robotics and automation (ICRA)*. IEEE, 2019, pp. 8832–8838. 3.1.2

[31] B. Cohen, S. Chitta, and M. Likhachev, "Single- and dual-arm motion planning with heuristic search," *The International Journal of Robotics Research*, vol. 33, no. 2, pp. 305–320, 2014. 3.1.2, 7

[32] R. Natarajan, G. L. Johnston, N. Simaan, M. Likhachev, and H. Choset, "Torque-limited manipulation planning through contact by interleaving graph search and trajectory optimization," *arXiv preprint arXiv:2210.08627*, 2022. 3.1.2, 8

[33] S. Karaman and E. Frazzoli, "Optimal kinodynamic motion planning using incremental sampling-based methods," in *49th IEEE conference on decision and control (CDC)*. IEEE, 2010, pp. 7681–7687. 3.1.2

[34] N. M. Amato and L. K. Dale, "Probabilistic roadmap methods are embarrassingly parallel," in *Proceedings 1999 IEEE International Conference on Robotics and Automation*, vol. 1, 1999, pp. 688–694. 3.2.1

[35] S. A. Jacobs, K. Manavi, J. Burgos, J. Denny, S. Thomas, and N. M. Amato, "A scalable method for parallelizing sampling-based motion planning algorithms," in *2012 IEEE International Conference on Robotics and Automation*, 2012, pp. 2529–2536. 3.2.1

[36] D. Devaurs, T. Siméon, and J. Cortés, "Parallelizing RRT on distributed-memory architectures," in *2011 IEEE International Conference on Robotics and Automation*, 2011, pp. 2261–2266. 3.2.1

[37] J. Ichnowski and R. Alterovitz, "Parallel sampling-based motion planning with superlinear speedup." in *IROS*, 2012, pp. 1206–1212. 3.2.1

[38] S. A. Jacobs, N. Stradford, C. Rodriguez, S. Thomas, and N. M. Amato, "A scalable distributed RRT for motion planning," in *2013 IEEE International Conference on Robotics and Automation*, 2013, pp. 5088–5095. 3.2.1

[39] C. Park, J. Pan, and D. Manocha, "Parallel motion planning using poisson-disk sampling," *IEEE Transactions on Robotics*, vol. 33, no. 2, pp. 359–371, 2016. 3.2.1

[40] J. Butzke, K. Sapkota, K. Prasad, B. MacAllister, and M. Likhachev, "State lattice with controllers: Augmenting lattice-based path planning with controller-based motion primitives," in *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2014, pp. 258–265. 3.2.1

[41] B. Eysenbach, R. R. Salakhutdinov, and S. Levine, "Search on the replay buffer: Bridging planning and reinforcement learning," *Advances in Neural Information Processing Systems*, vol. 32, 2019. 3.2.1

[42] M. Evett, J. Hendler, A. Mahanti, and D. Nau, "PRA*: Massively parallel heuristic search," *Journal of Parallel and Distributed Computing*, vol. 25, no. 2, pp. 133–143, 1995. 3.2.2

[43] R. Zhou and E. A. Hansen, "Parallel structured duplicate detection," in *AAAI*, 2007, pp. 1217–1224. 3.2.2

[44] E. Burns, S. Lemons, W. Ruml, and R. Zhou, "Best-first heuristic search for multicore machines," *Journal of Artificial Intelligence Research*, vol. 39, pp. 689–743, 2010. 3.2.2

[45] A. Kishimoto, A. Fukunaga, and A. Botea, "Scalable, parallel best-first search

for optimal sequential planning," in *Proceedings of the International Conference on Automated Planning and Scheduling*, vol. 19, no. 1, 2009. 3.2.2

[46] B. J. Cohen, M. Phillips, and M. Likhachev, "Planning single-arm manipulations with n-arm robots." in *Robotics: Science and Systems*, 2014. 3.3, 4.4.1

[47] C. Dellin and S. Srinivasa, "A unifying formalism for shortest path problems with expensive edge evaluations via lazy best-first search over paths with edge selectors," in *Proceedings of the International Conference on Automated Planning and Scheduling*, vol. 26, no. 1, 2016. 3.3, 4, 4.4.1

[48] A. Mandalika, O. Salzman, and S. Srinivasa, "Lazy receding horizon A* for efficient path planning in graphs with expensive-to-evaluate edges," in *Proceedings of the International Conference on Automated Planning and Scheduling*, vol. 28, no. 1, 2018. 3.3, 4

[49] A. Mandalika, S. Choudhury, O. Salzman, and S. Srinivasa, "Generalized lazy search for robot motion planning: Interleaving search and edge evaluation via event-based toggles," in *Proceedings of the International Conference on Automated Planning and Scheduling*, vol. 29, 2019, pp. 745–753. 3.3, 4, 4.4.1, 4.5

[50] J. Lim, S. Srinivasa, and P. Tsiotras, "Lazy lifelong planning for efficient replanning in graphs with expensive edge evaluation," in *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2022, pp. 8778–8783. 3.3, 4.5

[51] V. Narayanan and M. Likhachev, "Heuristic search on graphs with existence priors for expensive-to-evaluate edges," in *Twenty-Seventh International Conference on Automated Planning and Scheduling*, 2017. 3.3, 4, 4.4.1

[52] M. Bhardwaj, S. Choudhury, B. Boots, and S. Srinivasa, "Leveraging experience in lazy search," *Autonomous Robots*, vol. 45, pp. 979–996, 2021. 3.3

[53] S. Richter, J. Thayer, and W. Ruml, "The joy of forgetting: Faster anytime search via restarting," in *Proceedings of the International Conference on Automated Planning and Scheduling*, vol. 20, 2010, pp. 137–144. 3.4

[54] D. M. Saxena, T. Kusnur, and M. Likhachev, "AMRA*: Anytime multi-resolution multi-heuristic A*," in *2022 International Conference on Robotics and Automation (ICRA)*. IEEE, 2022, pp. 3371–3377. 3.4, 9.2.5

[55] J. D. Gammell, S. S. Srinivasa, and T. D. Barfoot, "Batch informed trees (BIT): Sampling-based optimal planning via the heuristically guided search of implicit random geometric graphs," in *2015 IEEE international conference on robotics and automation (ICRA)*. IEEE, 2015, pp. 3067–3074. 3.4

[56] S. Koenig, M. Likhachev, and D. Furcy, "Lifelong planning A*," *Artificial Intelligence*, vol. 155, no. 1-2, pp. 93–146, 2004. 3.4

[57] N. Haghtalab, S. Mackenzie, A. Procaccia, O. Salzman, and S. Srinivasa, "The provable virtue of laziness in motion planning," in *Proceedings of the International Conference on Automated Planning and Scheduling*, vol. 28, no. 1, 2018. 4

[58] V. Makoviychuk, L. Wawrzyniak, Y. Guo, M. Lu, K. Storey, M. Macklin, D. Hoeller, N. Rudin, A. Allshire, A. Handa *et al.*, "Isaac gym: High performance GPU based physics simulation for robot learning," in *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*, 2021. 4.4.2, 5.4.2, 9.2.2

[59] M. Likhachev, G. J. Gordon, and S. Thrun, "ARA*: formal analysis," 2003. 6.3

[60] N. Sturtevant, "Benchmarks for grid-based pathfinding," *Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 2, pp. 144 – 148, 2012. [Online]. Available: http://web.cs.du.edu/~sturtevant/papers/benchmarks.pdf 6.4

[61] B. Bonet and H. Geffner, "Planning as heuristic search," *Artificial Intelligence*, vol. 129, no. 1-2, pp. 5–33, 2001. 9.2.4

[62] R. E. Korf, "Depth-first iterative-deepening: An optimal admissible tree search," *Artificial intelligence*, vol. 27, no. 1, pp. 97–109, 1985. 9.2.5

[63] R. Holte, A. Felner, G. Sharon, and N. Sturtevant, "Bidirectional search that is guaranteed to meet in the middle," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 30, no. 1, 2016. 9.2.5

[64] B. S. Stewart and C. C. White III, "Multiobjective A*," *Journal of the ACM (JACM)*, vol. 38, no. 4, pp. 775–814, 1991. 9.2.5

[65] L. Mandow, J. P. De la Cruz *et al.*, "A new approach to multiobjective A* search." in *IJCAI*, vol. 8, 2005. 9.2.5

[66] C. H. Ulloa, W. Yeoh, J. A. Baier, H. Zhang, L. Suazo, and S. Koenig, "A simple and fast bi-objective search algorithm," in *Proceedings of the International Conference on Automated Planning and Scheduling*, vol. 30, 2020, pp. 143–151. 9.2.5

[67] Z. Ren, R. Zhan, S. Rathinam, M. Likhachev, and H. Choset, "Enhanced multi-objective A* using balanced binary search trees," in *Proceedings of the International Symposium on Combinatorial Search*, vol. 15, no. 1, 2022, pp. 162–170. 9.2.5