SPEEDING UP MULTI-OBJECTIVE SEARCH ALGORITHMS

by

Han Zhang

A Dissertation Presented to the
FACULTY OF THE USC GRADUATE SCHOOL
UNIVERSITY OF SOUTHERN CALIFORNIA
In Partial Fulfillment of the
Requirements for the Degree
DOCTOR OF PHILOSOPHY
(COMPUTER SCIENCE)

December 2024

Acknowledgements

I want to thank my advisor, Sven Koenig, for his guidance and support throughout my Ph.D., the greatest freedom he has offered that allows me to explore different research ideas, and the countless valuable lessons on things from writing papers to fostering great collaborations.

To the rest of my committee, T. K. Satish Kumar, Ariel Felner, Bistra Dilkina, and Satyandra Kumar Gupta, thank you for your valuable time and thoughtful suggestions. I would especially like to thank Satish for the valuable advice throughout my Ph.D. and Ariel for introducing me to the world of heuristic search. I also want to thank Lars Lindemann for the helpful feedback on the proposal for this dissertation.

Much of the research presented in this dissertation would not have been possible without the help and inspiration of my extremely bright collaborators: Oren Salzman, Shawn Skylar, and Carlos Hernández Ulloa. Our collaboration has always been productive and fun. I have learned a lot from every one of you.

I have had a great time here at USC with many wonderful people. Thanks to the former members of the lab, Jiaoyang Li, Hang Ma, and Liron Cohen, for their advice that helped me immensely as a new Ph.D. student. Thanks to my colleagues Shao-Hung Chan, Yi Zheng, Taoan Huang, Ang Li, Weizhe Chen, Christopher Leet, Thomy Phan, Yimin Tang, and Kexin Zheng for the fun discussion and brainstorming. I was also fortunate to work with some brilliant undergraduate students, including Fangji Wang, Cheng Ge, Yutong Li, Mingze Yao, Ziang Liu, and Lucas Terr.

Throughout my Ph.D., I am fortunate to have worked with many great researchers on various interesting projects other than the ones presented in this dissertation: Pavel Surynek, William Yeoh, Thayne Walker, Jingkai Chen, Fandel Lin, Zhongqiang Ren, Daniel Harabor, Mike Chen, Yue Zhang, and Neelesh Tiruviluamala. Many thanks to all of them.

Finally, I would like to thank my family: I want to thank my wife, Mengxiao He, for the joy she has brought to my life. I want to thank my parent, Fengzhu Ren and Chungui Zhang, for their love and support.

Table of Contents

List of Tables

List of Figures

List of Algorithms

Abstract

The multi-objective search problem is the problem of finding paths from a start state to a goal state in a graph where each edge is annotated with multiple costs. Each cost quantifies a type of resource consumed when traversing the edge. This problem is important for many applications, such as transporting hazardous materials, where travel distance and risk are two costs that need to be considered. We say that a path π dominates another path π' iff π is no larger than π' in all costs and is smaller than π' in at least one cost. A typical task of multi-objective search is to find the *Pareto frontier*, that is, a maximal subset of all undominated paths from the start state to the goal state such that no two paths have the same cost. Researchers have developed a class of multi-objective search algorithms that generalize best-first (single-objective) search algorithms, such as A*, to multi-objective search.

Different from single-objective search, two issues need to be carefully considered in multi-objective search. First, the size of the Pareto frontier can be exponential in the size of the graph. Therefore, computing the entire Pareto frontier can be very time-consuming. Second, multi-objective search algorithms need to maintain multiple paths from the start state to each state encountered during the search. This requires multi-objective search algorithms to perform additional check operations to determine if a path should be discarded or not, which slows down the

search. Therefore, existing multi-objective search algorithms do not scale well to large graphs or many objectives.

While researchers have developed various techniques over the past years for speeding up (single-objective) searches on large graphs, many of them have not been investigated in the context of multi-objective search. I hypothesize that one can speed up multi-objective search algorithms by applying insights gained from single-objective search algorithms after proper generalization. Specifically, I consider the following four classes of techniques that have been used to speed up single-objective search algorithms, namely, (1) by trading off solution quality with efficiency, (2) by anytime search, (3) by preprocessing techniques, and (4) by efficient data structures for time-consuming operations.

To validate our hypothesis, we make the following contributions:

1. We introduce A*pex, which speeds up multi-objective search by trading off solution quality with efficiency. Given an approximation factor ε, A*pex computes an ε-approximate Pareto frontier, that is, for every path π from the start state to the goal state, A*pex finds at least one path π' from the start state to the goal state such that π' is no worse than $1 + \varepsilon$ times π in every cost. A*pex relies on a clever data structure that merges paths with similar costs. Merging paths reduces the search effort and hence speeds up the search. We empirically show the efficiency advantage of A*pex over state-of-the-art multi-objective search algorithms and their approximate variants. Our experimental results also validate that the runtime of A*pex decreases as the given approximation factor increases.

2. We introduce A-A*pex, an anytime multi-objective search algorithm that builds upon A*pex. It computes an initial approximate Pareto frontier quickly and then finds better

approximate Pareto frontiers until eventually finding the entire Pareto frontier. In each iteration of its main loop, A-A*pex runs A*pex with an approximation factor that is smaller than those of the previous iterations. Additionally, we propose a speed-up technique that reuses previous search effort by resuming the search from paths that were pruned in the previous iterations. We empirically show that, given the same amount of time, A-A*pex often computes paths that collectively approximate the Pareto frontier much better than state-of-the-art multi-objective search algorithms.

3. We introduce a preprocessing technique for multi-objective search based on Contraction Hierarchies (CHs). CHs have been successfully used as a preprocessing technique in single-objective search. Our approach generalizes CHs to any number of objectives. Furthermore, we observe that CHs in multi-objective search can contain a large number of edges, which slows down the search algorithm. Consequently, we introduce a (general) partial-expansion technique that dramatically speeds up the search algorithm by reducing the number of unnecessarily generated search nodes. We empirically show that our CH-based approach can speed up state-of-the-art multi-objective search algorithms by up to three orders of magnitude.

4. We introduce bucket arrays, a data structure for speeding up time-consuming operations in multi-objective search algorithms. When a multi-objective search algorithm considers a new path from the start state to some state s, it needs to check if this path is dominated by any of the previously found paths to s. These check operations are called dominance checks. We propose to use bucket arrays to store the cost vectors of the previously found paths. In bucket arrays, these cost vectors are slotted into different predefined buckets

based on their values. Therefore, when performing dominance checks, the multi-objective search algorithm often does not need to iterate over all vectors in a bucket. We empirically show that bucket arrays can speed up LTMOA*, a state-of-the-art multi-objective search algorithm, by up to 4.3 times on average for problem instances with five objectives.

Chapter 1

Introduction

In the single-objective search problem, one is given a graph, a start state, and a goal state. Each edge in the graph is annotated with a cost, which quantifies a resource consumed when traversing the edge, such as the traversal time. The task is to find a path from the given start state to the given goal state in the graph that minimizes the path cost, that is, the sum of the costs of the edges that it contains. On the other hand, many applications are concerned with two or more competing resources, represented by multiple costs that annotate each edge. For example, when planning a route for transporting hazardous material, one needs to consider costs such as the travel time and the number of residents that would be exposed to the hazardous material in case of a traffic accident [11]. When planning a bicycling route, one needs to consider costs such as cycling time and climbing altitude gain [69]. Often, a path with a small amount of one cost can have a large amount of some other cost and vice versa, meaning that no path minimizes all costs simultaneously. Therefore, we are often interested in computing a set of paths that trade-off between the different costs.

The multi-objective search problem generalizes the single-objective search problem to consider multiple costs. In multi-objective search, each edge in the graph is annotated with a cost vector of length N. The cost of a path is the component-wise sum of the costs of the edges that this path contains. Path π *dominates* another path π' iff π is no larger than π' in all costs and is smaller than π' in at least one cost. A *solution* is a path from the start state to the goal state. Each component of the solution cost corresponds to an *objective* function to minimize, and we have N objectives. A typical task of multi-objective search is to find a *Pareto frontier*, that is, a maximal subset of all undominated solutions such that no two solutions have the same cost. Intuitively, the set of all undominated solutions contains all "efficient" candidate solutions that allow the decision-maker to choose from and make trade-offs. When there are multiple undominated solutions with the same cost, we are interested in finding only one of them. The multi-objective search problem is important for many real-world applications, including route planning for trucks [11] and robots [15], planning power-transmission lines [6], scheduling satellites [26], and inspecting regions of interest with robots [23, 24].

Researchers have developed several multi-objective search algorithms that extend single-objective best-first search algorithms, such as A*, to finding Pareto frontiers. These algorithms include BOA* [37], EMOA* [59], and LTMOA* [36]. Like A*, these algorithms conform to a best-first search framework and utilize heuristic guidance. Unlike A*, they do not terminate when finding the first solution because they need to find a set of solutions instead of only one solution. While A* needs to consider only the minimum-cost path from the start state to each state, a multi-objective search algorithm needs to consider the set of undominated paths from the start state to each state during the search. When a multi-objective search algorithm considers a new path to some state s, it checks if this path is dominated by any of the previously found paths to state s and prunes the path if so. These check operations are called *dominance checks*.

Compared to single-objective search, two issues need to be carefully considered in multi-objective search. The first one is that the size of the Pareto frontier can be exponential in the size of the graph being searched [19, 10], which often makes existing multi-objective search algorithms very time-consuming. Moreover, dominance checks are performed frequently, that is, in the inner loop of the search, and require iterating over sets of vectors, which introduces additional complexity to multi-objective search algorithms and slows down the search. Therefore, existing multi-objective search algorithms do not scale well to large graphs or many objectives.

In a broader perspective, the multi-objective search problem is a special case of multi-objective optimization problems [62, 20]. Different approaches have been proposed to solve different multi-objective optimization problems. These approaches include SAT-based approaches [38, 66] and multi-objective evolutionary algorithms [9, 16, 20]. Although these approaches are out of the scope of this dissertation, the techniques proposed in this dissertation might be relevant to them.

1.1 Hypotheses

Due to the similarity of multi-objective and single-objective search algorithms, my hypothesis is as follows:

> *One can speed up multi-objective search algorithms by applying insights gained from single-objective search algorithms after proper generalization.*

Specifically, I consider the following four classes of techniques that have been used to speed up single-objective search algorithms in existing work:

1. One can speed up multi-objective search algorithms by trading off solution quality with efficiency. In single-objective search, a bounded-suboptimal search algorithm finds a solution whose cost is at most the minimum solution cost times a given suboptimality factor. Such algorithms include WA* [53], focal search [50], optimistic search [73], and EES [72]. Bounded-suboptimal search algorithms have been shown to find solutions much faster than A*. Their runtime typically decreases as the given suboptimality factor increases. In multi-objective search, instead of computing the Pareto frontier, one can compute an approximate Pareto frontier, which satisfies that, for every solution π of the problem instance, there exists a solution in the approximate Pareto frontier that "approximately dominates" π for a given approximation factor. I hypothesize that one can find an approximate Pareto frontier much faster than finding the Pareto frontier, and the runtime will decrease as the given approximation factor increases.

2. One can speed up multi-objective search algorithms by anytime search. In single-objective search, an anytime search algorithm computes a suboptimal solution quickly and then finds better solutions until eventually finding an optimal solution. Examples of anytime single-objective search algorithms are AWA* [33], ARA* [43], and AFS [13]. Typically, an anytime algorithm calls a bounded-suboptimal search algorithm with a decreasing sequence of suboptimality factors. It is slower than A* in finding the optimal solution but can find better solutions than A* when the given time is insufficient for finding the optimal solution. I hypothesize that one can generalize anytime search to the multi-objective case by using the approximate multi-objective search algorithms that are addressed in the previous

point. Such an anytime multi-objective search algorithm can find a set of solutions that approximates the Pareto frontier better than the one found by existing multi-objective search algorithms when the given time is insufficient for finding the Pareto frontier.

3. One can speed up multi-objective search algorithms by preprocessing techniques. In single-objective search, using preprocessing techniques is a common approach to speeding up solving multiple problem instances on the same graph. Such techniques include contraction hierarchies [28], true distance heuristics [70], embedding in Euclidean spaces [14], and sub-goal graphs [76]. These preprocessing techniques often exploit the structure of the given graph. For example, contraction hierarchies and sub-goal graphs are based on the observation that many minimum-cost paths in graphs, such as road networks and grids, traverse the same set of "important" states, and, hence, the search can be performed in a hierarchical manner. In multi-objective search, the graph can also have an exploitable structure with each individual cost. Therefore, I hypothesize that one can speed up multi-objective search algorithms via preprocessing techniques.

4. One can speed up multi-objective search algorithms by using efficient data structures for time-consuming operations. In single-objective search, heap operations are often the most time-consuming part of the search, and existing work has investigated using bucket-based heaps to speed up the search algorithms [12]. Different from single-objective search, in multi-objective search, dominance checks are often the most time-consuming operations of the search. Therefore, I hypothesize that one can speed up multi-objective search algorithms via efficient data structures for storing vectors and enabling more efficient dominance checks.

1.2 Contributions

Although multi-objective search algorithms can be viewed as a generalization of single-objective search algorithms, they are sufficiently different from single-objective search algorithms in that many techniques of single-objective search algorithms cannot be trivially applied. In this dissertation, we make the following contributions to speeding up multi-objective search algorithms:

1. To validate the hypothesis that multi-objective search algorithms can be sped up by trading off solution quality with efficiency, we introduce A*pex, an approximate multi-objective search algorithm. Although A*pex and (single-objective) bounded-suboptimal search algorithms are based on similar insights, they use very different techniques: While bounded-suboptimal search algorithms rely on different node expansion orders to guide the search and find solutions faster, A*pex relies on a clever data structure that merges paths with similar costs. By merging paths, A*pex reduces the search effort and hence speeds up the search. Given an approximation factor ε, A*pex guarantees that, for every solution π of the given problem instance, it computes a solution π' such that π' is no worse than $(1+\varepsilon)$ times π in every cost. A*pex extends PP-A* [31], a previous approximate bi-objective search algorithm (which only works with two objectives) by generalizing the data structure used by PP-A* to make PP-A* more efficient for bi-objective search and work with any number of objectives. We empirically show the efficiency advantage of A*pex over PP-A* for bi-objective search and an approximate baseline algorithm derived from LTMOA*, a state-of-the-art multi-objective search algorithm, for problem instances with more than two objectives. Our experimental results also validate that the runtime of A*pex decreases as the given approximation factor increases.

2. To validate the hypothesis that multi-objective search algorithms can be sped up by anytime search, we introduce *Anytime A*pex* (A-A*pex), which builds upon A*pex. From one iteration to the next, A-A*pex can either reuse its previous search effort or restart the search from scratch. We propose a technique for reusing previous search effort by resuming the search from paths that were pruned in the previous iteration. Additionally, we propose a hybrid variant of A-A*pex which first restarts the search from scratch for each iteration and then starts to reuse its search effort in later iterations. Existing work on anytime single-objective search has investigated reusing search effort [43] or restarting the search from scratch [60]. We show how to reuse the search effort of A*pex despite its unique merging operations. We empirically show that A-A*pex often computes solutions that collectively approximate the Pareto frontier much better than the solutions found by LTMOA* for short runtimes.

3. To validate the hypothesis that multi-objective search algorithms can be sped up by preprocessing techniques, we introduce a preprocessing technique for multi-objective search based on Contraction Hierarchies (CHs). In single-objective search, a CH is a hierarchical graph that assigns a level number to each state in the given graph and adds additional edges (known as *shortcuts*) to the given graph so that the shortest path from a given start state to a given goal state can be found by searching through the space of only up-down paths (paths with first increasing and then decreasing level numbers). Similarly, in multi-objective search, a CH needs to retain the property that the Pareto frontier can be computed by considering only up-down paths. To the best of our knowledge, CHs have been used in graphs with two costs but never to compute the Pareto frontier. Specifically, Storandt

[69] proposed a CH-based approach for solving the constrained shortest-path problem. Its preprocessing algorithm computes shortcuts heuristically, which avoids the computational cost of computing the exact shortcuts but can add unnecessary shortcuts. Our approach leverages recent algorithmic advances to speed up the previous preprocessing algorithm for bi-objective search and work with any number of objectives. Specifically, it uses LTMOA* to compute shortcuts. This alternative approach allows us to compute only the necessary shortcuts, speeding up both the preprocessing and the search. Furthermore, we observe that CHs in multi-objective search often contain a large number of edges, which slows down the search algorithm. Consequently, we introduce a (general) partial-expansion technique, which dramatically speeds up the search algorithm by reducing the number of unnecessarily generated search nodes. We empirically show that our multi-objective CHs can speed up LTMOA* by up to two orders of magnitude.

4. To validate the hypothesis that multi-objective search algorithms can be sped up by using efficient data structures for time-consuming operations, we introduce a data structure called the *bucket array*. In bucket arrays, vectors are slotted into different predefined buckets based on their values. A multi-objective search algorithm can often determine if a bucket contains a vector that dominates a given vector without iterating over all vectors in this bucket. We exploit this property to speed up the dominance checks of multi-objective search algorithms. We empirically show that bucket arrays are beneficial in many cases. For example, in a set of problem instances with five objectives, enhancing LTMOA* with bucket arrays yields a speed-up of 4.3 times on average.

Although these contributions are based on insights gained from single-objective search algorithms, these insights cannot be trivially applied to multi-objective search due to differences between single-objective search and multi-objective search. As we will see in Chapters 3-6, the techniques we develop based on these insights are often quite different from the techniques in single-objective search based on the same insights.

1.3 Dissertation Structure

This dissertation is structured as follows: In Chapter 2, we give an overview of the multi-objective search problem and existing multi-objective search algorithm. We then introduce A*pex in Chapter 3 and A-A*pex in Chapter 4. We describe multi-objective contraction hierarchies in Chapter 5. Then, we describe bucket arrays in Chapter 6 before concluding this dissertation in Chapter 7.

Chapter 2

Background

In this chapter, we define the multi-objective search problem and the Pareto frontier in Section 2.1. We then review existing multi-objective search algorithms for computing Pareto frontiers in Section 2.2.

2.1 Problem Definition

We use **boldface** font to denote a vector $\mathbf{v}$ and v_i to denote the i-th component of it. The sum of two vectors $\mathbf{v}$ and $\mathbf{v}'$ of the same length N is defined as the vector $\mathbf{v} + \mathbf{v}' = (v_1 + v'_1, v_2 + v'_2 \dots v_N + v'_N)$. $\mathbf{v} \preceq \mathbf{v}'$ denotes that $v_i \leq v'_i$ for all $i = 1, 2 \dots N$. In this case, we say that $\mathbf{v}$ *weakly dominates* $\mathbf{v}'$. $\mathbf{v} \prec \mathbf{v}'$ denotes that $\mathbf{v} \preceq \mathbf{v}'$ and there exists an $i \in \{1, 2 \dots N\}$ with $v_i < v'_i$. In this case, we say that $\mathbf{v}$ *dominates* $\mathbf{v}'$.

A *(multi-objective search) graph* is a tuple $G = \langle S, E \rangle$, where S is a finite set of *states*, and E is a finite set of *directed edges*. Each *edge* $e = \langle u, v, \mathbf{c} \rangle$ is a tuple consisting of a *source state* $u \in S$, a *target state* $v \in S$, and a *cost* $\mathbf{c} \in \mathbb{R}^N_{>0}$. We use $src(e)$, $tar(e)$, and $\mathbf{c}(e)$ to denote the source state, the target state, and the cost of e, respectively. The cost of an edge is a vector of

N cost components. Each cost component of the graph corresponds to an objective to minimize. The graph is called bi-objective in case $N = 2$. We use $in(s) = \{e \in E \mid tar(e) = s\}$ and $out(s) = \{e \in E \mid src(e) = s\}$ to denote the in- and out-edges of state s, respectively. State s' is an *in-neighbor* (resp. *out-neighbor*) of state s iff there exists an edge from s' to s (resp. from s to s'). We use $in\_nbr(s)$ and $out\_nbr(s)$ to denote the sets of all in-neighbor and out-neighbor states of state s, respectively.

A *path* from state s to state s' is a sequence of edges $\pi = [e_1, e_2 \ldots e_\ell]$ with $src(e_1) = s$, $tar(e_\ell) = s'$, and $tar(e_j) = src(e_{j+1})$ for all $j = 1, 2 \ldots \ell - 1$. $s = s_{\text{start}}$ unless mentioned otherwise. We use $s(\pi)$ to denote the last state of π, that is, s'. $\mathbf{c}(\pi) = \sum_{j=1}^{\ell} \mathbf{c}(e_j)$ denotes the cost of π. Path π can be *extended* with an edge $e_{\ell+1}$ to obtain path $[e_1, e_2 \ldots e_\ell, e_{\ell+1}]$ iff $s(\pi) = src(e_{\ell+1})$. We say that a path π' *extends* path π iff π' can be obtained by extending π with a sequence of edges. Path π *dominates (resp. weakly dominates)* a path π' iff $\mathbf{c}(\pi) \prec \mathbf{c}(\pi')$ (resp. $\mathbf{c}(\pi) \preceq \mathbf{c}(\pi')$).

We allow a graph to contain *parallel edges*, that is, there can exist multiple edges (with potentially different costs) from the same source state to the same target state. However, in many examples in this and the following chapters, we consider graphs without parallel edges. In such cases, each path corresponds to a distinct sequence of states. Therefore, for ease of presentation, we often refer to a path as a sequence of states.

A *(multi-objective search) problem instance* is specified by a tuple $P = \langle G, s_{\text{start}}, s_{\text{goal}} \rangle$, where $G = \langle S, E \rangle$ is a graph, $s_{\text{start}} \in S$ is the start state, and $s_{\text{goal}} \in S$ is the goal state. The instance is called bi-objective in case $N = 2$. A *solution* is a path from s_{start} to s_{goal}. A *Pareto-optimal* solution is a solution that is not dominated by any other solution. As Pareto-optimal solutions with the same cost may exist, a typical task of multi-objective search is computing a maximal subset of all Pareto-optimal solutions where no two solutions have the same cost. We call such a set of solutions the *cost-unique Pareto frontier*, or *Pareto frontier* for short.

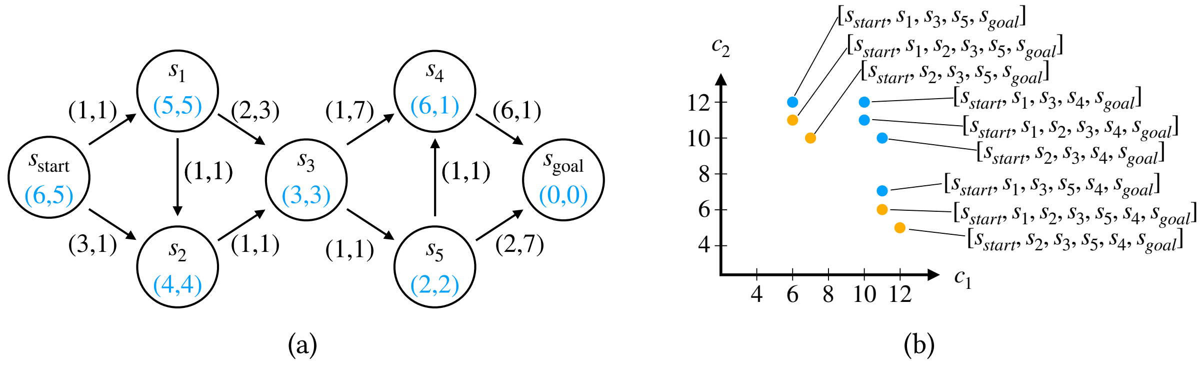
Definition 2.1. *A (cost-unique) Pareto frontier is a maximal subset of all Pareto-optimal solutions such that no two solutions have the same cost.*

A Pareto frontier provides a user with all solutions that might be desirable to them (in the absence of additional information). Different Pareto frontiers can exist for a multi-objective search problem instance. However, they are of the same size, which is equal to the number of unique costs of all Pareto-optimal solutions.

Theorem 2.1. *Consider a bi-objective search problem instance. The size of the Pareto frontier can be exponential in the number of states of the graph.*

The above theorem is rephrased from Theorem 2.1 of Breugem, Dollevoet, and Heuvel [10].

A *heuristic function* $\mathbf{h} : S \rightarrow \mathbb{R}^N_{\geq 0}$ estimates the cost of a path from any given state s to the goal state. We assume that the provided heuristic function $\mathbf{h}$ is consistent, that is, $\mathbf{h}(s_{\text{goal}}) = \mathbf{0}$ and $\mathbf{h}(src(e)) \preceq \mathbf{c}(\langle e \rangle) + \mathbf{h}(tar(e))$ for all $e \in E$. We assume that the reader is familiar with the properties of the A* search framework when used with a consistent heuristic function in single-objective search, for example, that the sequence of expanded nodes has monotonically non-decreasing f-values. A common approach to computing such consistent heuristics in existing literature [2, 5, 37, 36, 59, 87] is to use a backward search with Dijkstra's algorithm (starting from s_{goal}) to compute the minimum cost $c^*_i(s)$ from any state s to s_{goal} for the i-th objective (while ignoring all other objectives), for all $i = 1, 2 \ldots N$, and $\mathbf{h}(s) := [c^*_1(s), c^*_2(s) \ldots c^*_N(s)]$ as the heuristic function. We call this heuristic the *perfect-distance heuristic*.



s_{start} (6,5)
s_1 (5,5)
s_2 (4,4)
s_3 (3,3)
s_4 (6,1)
s_5 (2,2)
s_{goal} (0,0)
(1,1)
(3,1)
(2,3)
(1,1)
(1,1)
(1,7)
(1,1)
(1,1)
(6,1)
(2,7)
(a)
c_2
c_1
12
10
8
6
4
4
6
8
10
12
$[s_{start}, s_1, s_3, s_5, s_{goal}]$
$[s_{start}, s_1, s_2, s_3, s_5, s_{goal}]$
$[s_{start}, s_2, s_3, s_5, s_{goal}]$
$[s_{start}, s_1, s_3, s_4, s_{goal}]$
$[s_{start}, s_1, s_2, s_3, s_4, s_{goal}]$
$[s_{start}, s_2, s_3, s_4, s_{goal}]$
$[s_{start}, s_1, s_3, s_5, s_4, s_{goal}]$
$[s_{start}, s_1, s_2, s_3, s_5, s_4, s_{goal}]$
$[s_{start}, s_2, s_3, s_5, s_4, s_{goal}]$
(b)

Figure 2.1: An example bi-objective search problem instance. (a) shows the graph of this problem instance. The pair of numbers inside each state shows the value of the perfect-distance heuristic. (b) shows the costs of all solutions for this problem instance. The orange dots in (b) show the solutions in the Pareto frontier.

Example 1. *Figure 2.1 shows an example bi-objective search problem instance and the costs of its solutions. This problem instance has only one Pareto frontier because all solutions have unique costs. The Pareto frontier consists of four solutions, namely, solution* $[s_{start}, s_1, s_2, s_3, s_5, s_{goal}]$ *with cost* $(6, 11)$*, solution* $[s_{start}, s_2, s_3, s_5, s_{goal}]$ *with cost* $(7, 10)$*, solution* $[s_{start}, s_1, s_2, s_3, s_5, s_4, s_{goal}]$ *with cost* $(11, 6)$*, and solution* $[s_{start}, s_2, s_3, s_5, s_4, s_{goal}]$ *with cost* $(12, 5)$*.*

2.2 Best-First Multi-Objective Search Algorithms

In this section, we describe existing multi-objective search algorithms for computing Pareto frontiers. We begin with a general algorithmic framework called the best-first multi-objective search algorithm. We then describe dominance checks and the dimensionality-reduction technique, which is used by state-of-the-art multi-objective search algorithms to speed up dominance checks. Finally, we describe some state-of-the-art multi-objective search algorithms within the best-first multi-objective search framework.

2.2.1 The Best-First Multi-Objective Search Framework

Most existing multi-objective search algorithms conform to the best-first multi-objective search framework. A best-first multi-objective search algorithm is similar to best-first single-objective search algorithms, such as A*, but, unlike them, it needs to consider multiple search nodes—or simply called nodes—with $\mathbf{g}$-values that do not weakly dominate each other for the same state.

In a best-first multi-objective search algorithm, each node n contains a state $s(n)$ and a $\mathbf{g}$-value $\mathbf{g}(n)$. We define an $\mathbf{f}$-value for the node as $\mathbf{f}(n) = \mathbf{g}(n) + \mathbf{h}(s(n))$. Node n corresponds to a path from s_{start} to state $s(n)$ whose cost is $\mathbf{g}(n)$. Additionally, the search algorithm maintains the parent node $p(n)$ for n. The corresponding path of n extends the corresponding path of $p(n)$ with an out-edge of state $s(p(n))$ and can be constructed in reverse by following the parent nodes from n to the node that contains state s_{start}. Because the heuristic $\mathbf{h}$ is consistent, $\mathbf{f}(n)$ weakly dominates the cost of any solution that extends the corresponding path of n.

The search algorithm maintains a priority queue $Open$, which contains the generated but not yet expanded nodes, and a set of solutions $Sols$, which stores the Pareto frontier that will be returned. $Open$ is initialized with a node that contains the start state s_{start} and the $\mathbf{g}$-value $\mathbf{0}$. $Sols$ is initialized to $\emptyset$. In each iteration, the algorithm *extracts* a node from $Open$ whose $\mathbf{f}$-value is not dominated by the $\mathbf{f}$-value of any node in $Open$. The algorithm performs *dominance checks* for the extracted node to determine whether this node or any of its descendants can result in a new solution in the Pareto frontier. If not, the node is discarded. If the node is not discarded, the algorithm checks if the node contains the goal state. If so, the algorithm adds the corresponding path of the node to $Sols$. Otherwise, it expands the node by generating a new node for each of the successors of the state contained in the node. The algorithm performs dominance checks for each generated node to determine whether it or any of its descendants can result in a new solution in the Pareto frontier. If not, the generated node is discarded. Otherwise, it adds the generated node to $Open$. When $Open$ becomes empty, the algorithm terminates and returns $Sols$.

Examples of algorithms that conform to the best-first multi-objective search framework include NAMOA*-dr [54], BOA* [35], EMOA* [59], and LTMOA* [36]. They only differ in which node is extracted from $Open$ in each iteration, which information is contained in the nodes, and how the dominance checks are implemented and interleaved with the search.

2.2.2 Dominance Checks

When a multi-objective search algorithm considers a new node n (before expanding n or after extracting n from $Open$), it checks if the corresponding path of n is weakly dominated by any of the previously found paths to $s(n)$ and, if so, prunes n. Checking this condition requires comparing $\mathbf{g}(n)$ to a set of vectors, namely, the set of $\mathbf{g}$-values for all previously expanded nodes that also contain state $s(n)$. The operations for checking such conditions are modeled as *dominance checks*:

Definition 2.2. *Given a set of vectors* $\mathbf{X}$ *and a vector* $\mathbf{y}$*, the dominance check problem is the problem of checking whether there exists a vector* $\mathbf{x} \in \mathbf{X}$ *such that* $\mathbf{x} \preceq \mathbf{y}$*.*

Moreover, a multi-objective search algorithm also needs to update a set of undominated vectors $\mathbf{X}$ by first removing those vectors from it that are dominated by $\mathbf{y}$ and then adding $\mathbf{y}$. These operations are modeled as *undominated set updates*:

Definition 2.3. *Given a set of vectors* $\mathbf{X}$ *and a vector* $\mathbf{y}$*, the undominated set update problem is the problem of computing the subset of* $\mathbf{X} \cup \{\mathbf{y}\}$ *that is not dominated by any vector in* $\mathbf{X} \cup \{\mathbf{y}\}$*.*

Dominance checks and undominated set updates are often the most time-consuming parts of a multi-objective search algorithm because they are performed frequently and intrinsically require iterating over sets of vectors. Therefore, a multi-objective search algorithm needs to perform them efficiently. A straightforward implementation for these two operations is to store $\mathbf{X}$ as an array of vectors. In this case, both dominance checks and undominated set updates can be done with $O(|\mathbf{X}|)$ vector comparisons.

2.2.3 Dimensionality Reduction

The *dimensionality reduction*[1] technique [54] is a general technique for speeding up dominance checks and undominated set updates. We need to introduce additional notation before describing this technique: The *truncate function* Tr takes a vector $\mathbf{x} = [x_1, x_2 \dots x_N]$ as input and outputs $\mathbf{x}$ with its first component deleted, that is, $[x_2, x_3 \dots x_N]$. Given a set of N-dimensional vectors $\mathbf{X}$, we use $Tr(\mathbf{X}) = \{Tr(\mathbf{x}) \mid \mathbf{x} \in \mathbf{X}\}$ to denote the set of truncated vectors of $\mathbf{X}$. We use $ND(\mathbf{X}) = \{\mathbf{x} \in \mathbf{X} \mid \nexists \mathbf{x}' \in \mathbf{X}\ \ \mathbf{x}' \prec \mathbf{x}\}$ to denote the undominated subset of $\mathbf{X}$.

The dimensionality reduction technique is based on the following observations: Given a vector $\mathbf{y}$ and a set of vectors $\mathbf{X}$, if $y_1 \geq x_1$ for all $\mathbf{x} \in \mathbf{X}$, we do not need to consider the first component of any vector when performing the dominance check for $\mathbf{y}$ over $\mathbf{X}$. Instead of checking if there exists a vector $\mathbf{x} \in \mathbf{X}$ that weakly dominates $\mathbf{y}$, we only need to check if there exists an $(N-1)$-dimensional vector $\mathbf{x}' \in ND(Tr(\mathbf{X}))$ that weakly dominates $Tr(\mathbf{y})$. Doing so can greatly reduce the number of vector comparisons because $ND(Tr(\mathbf{X}))$ can be much smaller than $\mathbf{X}$. When we perform dominance checks and undominated set updates, if the given set of vectors

[1]In multi-objective search, dimensionality reduction does not have the same meaning as popularly used in machine learning and data science.

$\mathbf{X}$ and the given vector $\mathbf{y}$ always satisfy the condition that $y_1 \geq x_1$ for all $\mathbf{x} \in \mathbf{X}$, we can maintain only $ND(Tr(\mathbf{X}))$ instead of maintaining $\mathbf{X}$ explicitly. As we will see shortly, this condition often holds in best-first multi-objective search algorithms when consistent heuristics are used.

Example 2. *(adapted from Salzman et al. [63]) Consider the set* $\mathbf{X} = \{[6,2,4],[4,4,5],[2,3,6]\}$ *and note that* $\mathbf{X} = ND(\mathbf{X})$ *and* $ND(Tr(\mathbf{X})) = ND([2,4],[4,5],[3,6]) = \{[2,4]\}$. *Checking whether* $\mathbf{y} = [7,2,5]$ *is weakly dominated by a vector in* $\mathbf{X}$ *without using dimensionality reduction requires three vector comparisons. However, we can apply dimensionality reduction because* $y_1 \geq x_1$ *for all* $\mathbf{x} \in \mathbf{X}$ *and, hence, we only need one vector comparison (between* $Tr(\mathbf{y}) = [2,5]$ *and* $[2,4] \in ND(Tr(\mathbf{X}))$*) to conclude that there is a vector in* $\mathbf{X}$ *that weakly dominates* $\mathbf{y}$.

2.2.4 BOA*

In this section, we describe Bi-Objective A* (BOA*) [37]. Given a bi-objective search problem instance, BOA* computes a Pareto frontier for it. Algorithm 1 shows the pseudocode for BOA*. It initializes $Open$ with a node that contains state s_{start} and whose $\mathbf{g}$-value is $\mathbf{0}$ (Lines 1-2). This node corresponds to path $[s_{start}]$. In each iteration, BOA* *extracts* a node n from $Open$ with the *lexicographically smallest* $\mathbf{f}$-value. BOA* performs dominance checks after extracting a node from $Open$ (that is, after Line 7) and before adding a node to $Open$ (that is, before Line 18). It discards a node n if

- (Condition 1) there exists an expanded node n_{sol} that contains state s_{goal} and whose $\mathbf{g}$-value weakly dominates $\mathbf{f}(n)$ or
- (Condition 2) there exists an expanded node n' that contains state $s(n)$ and whose $\mathbf{g}$-value weakly dominates $\mathbf{g}(n)$.

Algorithm 1: BOA*

```
Input  : A bi-objective search problem instance $\langle G, s_{start}, s_{goal} \rangle$ and a consistent
         heuristic function $\mathbf{h}$
Output: A Pareto frontier
1  $n \leftarrow$ new node with $s(n) = s_{start}$, $\mathbf{g}(n) = \mathbf{0}$, and $p(n) = null$
2  $Open \leftarrow \{n\}$
3  $Sols \leftarrow \emptyset$
4  foreach $s \in S$ do
5      $g_2^{\min}(s) \leftarrow \infty$
6  while $Open \neq \emptyset$ do
7      extract a node $n$ from $Open$ with the lexicographically smallest $\mathbf{f}$-value
8      if IsDominated$(n)$ then
9          continue
10     $g_2^{\min}(s) \leftarrow g_2(n)$
11     if $s(n) = s_{goal}$ then
12         add the corresponding solution of $n$ to $Sols$
13         continue
14     for each $e \in out(s(n))$ do
15         $n' \leftarrow$ new node with $s(n') = tar(e)$, $\mathbf{g}(n') = \mathbf{g}(n) + \mathbf{c}(e)$, and $p(n') = n$
16         if IsDominated$(n')$ then
17             continue
18         add $n'$ to $Open$
19 return $Sols$
20 Function IsDominated$(n)$:
21     if $g_2^{min}(s_{goal}) \leq f_2(n)$ then
22         return true
23     if $g_2^{min}(s(n)) \leq g_2(n)$ then
24         return true
25     return false
```

Because the heuristic is required to be consistent, all solutions that can be found via node n must have costs that are weakly dominated by $\mathbf{f}(n)$. Each expanded node n_{sol} that contains state s_{goal} corresponds to a solution (that is, a path from s_{start} to s_{goal}) π_{sol} with cost $\mathbf{g}(n_{\text{sol}})$. If Condition 1 holds for such an expanded node n_{sol}, BOA* discards node n because all solutions that can be found via node n are weakly dominated by π_{sol}. If Condition 2 holds, BOA* discards node n because all solutions that can be found via node n must be weakly dominated by some solution that can be found via node n'.

A straightforward approach to checking Conditions 1 and 2 is to perform dominance checks for $\mathbf{f}(n)$ over the set of $\mathbf{g}$-values of all expanded nodes that contain state s_{goal} and $\mathbf{g}(n)$ over the set of $\mathbf{g}$-values of all expanded nodes that contain state $s(n)$, respectively. However, because BOA* always extracts a node from $Open$ with the lexicographically smallest $\mathbf{f}$-value of all nodes in $Open$ and also because the heuristic is consistent, the f_1-values of the extracted nodes of BOA* are monotonically non-decreasing. Therefore, BOA* uses dimensionality reduction to speed up dominance checks and does not check the g_1- and f_1-values. Instead of maintaining the set of $\mathbf{g}$-values of all expanded nodes, BOA* maintains its undominated truncated set, which is represented by the minimum g_2-value of all expanded nodes that contains state s and denoted as $g_2^{\min}(s)$. BOA* updates $g_2^{\min}(s)$ on Line 10 right after expanding a node that contains s. When performing dominance checks on node n, $f_1(n)$ is no smaller than the g_1-value of any expanded node n_{sol} that contains state s_{goal} because $f_1(n) \geq f_1(n_{\text{sol}})$ and $g_1(n_{\text{sol}}) = f_1(n_{\text{sol}})$. Therefore, Condition 1 holds iff $g_2^{\min}(s_{goal})$ is no larger than $f_2(n)$. Similarly, $g_1(n)$ is no smaller than the g_1-value of any expanded node n' that contains state $s(n)$ because $f_1(n) \geq f_1(n')$ and $h_1(n) = h_1(n')$. Therefore, Condition 2 holds iff $g_2^{\min}(s(n))$ is no larger than $g_2(n)$. Conditions 1 and 2 are checked on Lines 21 and 23, respectively. Both checks are done in constant time.

Let n be a node extracted by BOA* and not discarded after the dominance checks. If $s(n) = s_{goal}$, BOA* then adds the corresponding path (which is a solution) of n to $Sols$. Otherwise, BOA* *expands* n by generating a new child node for each successor of $s(n)$. When $Open$ becomes empty, the algorithm terminates and returns $Sols$. Hernández et al. [37] shows that BOA* terminates in finite time and returns $Sols$ as a Pareto frontier.

Iter	*Open* $\langle s(n), \mathbf{g}(n), \mathbf{f}(n)\rangle$	Generated $\langle s(n), \mathbf{g}(n), \mathbf{f}(n)\rangle$	Update of $g_2^{\min}(s(n))$
1	$\langle s_{start}, (0,0), (6,5)\rangle *$	$\langle s_1, (1,1), (6,6)\rangle$ $\langle s_2, (3,1), (7,5)\rangle$	$g_2^{\min}(s_{start}) = 0$
2	$\langle s_1, (1,1), (6,6)\rangle *$ $\langle s_2, (3,1), (7,5)\rangle$	$\langle s_2, (2,2), (6,6)\rangle$ $\langle s_3, (3,4), (6,7)\rangle$	$g_2^{\min}(s_1) = 1$
3	$\langle s_2, (2,2), (6,6)\rangle *$ $\langle s_3, (3,4), (6,7)\rangle$ $\langle s_2, (3,1), (7,5)\rangle$	$\langle s_3, (3,3), (6,6)\rangle$	$g_2^{\min}(s_2) = 2$
4	$\langle s_3, (3,3), (6,6)\rangle *$ $\langle s_3, (3,4), (6,7)\rangle$ $\langle s_2, (3,1), (7,5)\rangle$	$\langle s_5, (4,4), (6,6)\rangle$ $\langle s_4, (4,10), (10,11)\rangle$	$g_2^{\min}(s_3) = 3$
5	$\langle s_5, (4,4), (6,6)\rangle *$ $\langle s_3, (3,4), (6,7)\rangle$ $\langle s_2, (3,1), (7,5)\rangle$ $\langle s_4, (4,10), (10,11)\rangle$	$\langle s_4, (5,5), (11,6)\rangle$ $\langle s_{goal}, (6,11), (6,11)\rangle$	$g_2^{\min}(s_5) = 4$
6	$\langle s_3, (3,4), (6,7)\rangle *$ (discarded) $\langle s_{goal}, (6,11), (6,11)\rangle$ $\langle s_2, (3,1), (7,5)\rangle$ $\langle s_4, (4,10), (10,11)\rangle$ $\langle s_4, (5,5), (11,6)\rangle$		
7	$\langle s_{goal}, (6,11), (6,11)\rangle *$ $\langle s_2, (3,1), (7,5)\rangle$ $\langle s_4, (4,10), (10,11)\rangle$ $\langle s_4, (5,5), (11,6)\rangle$		$g_2^{\min}(s_{goal}) = 11$
8	$\langle s_2, (3,1), (7,5)\rangle *$ $\langle s_4, (4,10), (10,11)\rangle$ $\langle s_4, (5,5), (11,6)\rangle$	$\langle s_3, (4,2), (7,5)\rangle$	$g_2^{\min}(s_2) = 1$
9	$\langle s_3, (4,2), (7,5)\rangle *$ $\langle s_4, (4,10), (10,11)\rangle$ $\langle s_4, (5,5), (11,6)\rangle$	$\langle s_5, (5,3), (7,5)\rangle$ $\langle s_4, (5,9), (11,10)\rangle$	$g_2^{\min}(s_3) = 2$
10	$\langle s_5, (5,3), (7,5)\rangle *$ $\langle s_4, (4,10), (10,11)\rangle$ $\langle s_4, (5,5), (11,6)\rangle$ $\langle s_4, (5,9), (11,10)\rangle$	$\langle s_{goal}, (7,10), (7,10)\rangle$ $\langle s_4, (6,4), (12,5)\rangle$	$g_2^{\min}(s_5) = 3$
11	$\langle s_{goal}, (7,10), (7,10)\rangle *$ $\langle s_4, (4,10), (10,11)\rangle$ $\langle s_4, (5,5), (11,6)\rangle$ $\langle s_4, (5,9), (11,10)\rangle$ $\langle s_4, (6,4), (12,5)\rangle$		$g_2^{\min}(s_{goal}) = 10$
12	$\langle s_4, (4,10), (10,11)\rangle *$ (discarded) $\langle s_4, (5,5), (11,6)\rangle$ $\langle s_4, (5,9), (11,10)\rangle$ $\langle s_4, (6,4), (12,5)\rangle$		
13	$\langle s_4, (5,5), (11,6)\rangle *$ $\langle s_4, (5,9), (11,10)\rangle$ $\langle s_4, (6,4), (12,5)\rangle$	$\langle s_{goal}, (11,6), (11,6)\rangle$	$g_2^{\min}(s_4) = 5$

Iter	*Open* $\langle s(n), \mathbf{g}(n), \mathbf{f}(n)\rangle$	Generated $\langle s(n), \mathbf{g}(n), \mathbf{f}(n)\rangle$	Update of $g_2^{\min}(s(n))$
14	$\langle s_{goal}, (11,6), (11,6)\rangle *$ $\langle s_4, (5,9), (11,10)\rangle$ $\langle s_4, (6,4), (12,5)\rangle$		$g_2^{\min}(s_{goal}) = 6$
15	$\langle s_4, (5,9), (11,10)\rangle *$ (discarded) $\langle s_4, (6,4), (12,5)\rangle$		
16	$\langle s_4, (6,4), (12,5)\rangle *$	$\langle s_{goal}, (12,5), (12,5)\rangle$	$g_2^{\min}(s_4) = 4$
17	$\langle s_{goal}, (12,5), (12,5)\rangle *$		$g_2^{\min}(s_{goal}) = 5$
18	empty		

Table 2.1: Trace of $Open$, generated nodes, and $g_2^{\min}$ in each iteration of BOA* on solving the example problem instance in Figure 2.1. "$*$" marks the node that is extracted in that iteration.

Example 3. *Consider the bi-objective search problem instance in Figure 2.1. Table 2.1 shows a trace of $Open$, generated nodes, and changes to g_2^{min} in each iteration of BOA*.*

In Iteration 6, node $\langle s_3, (3,4), (6,7)\rangle$ is extracted and pruned because its truncated $\mathbf{g}$-value is no smaller than the minimum g_2-values of expanded nodes that contain state s_3. Conceptually, node $\langle s_3, (3,4), (6,7)\rangle$ corresponds to path $[s_{start}, s_1, s_3]$. Because path $[s_{start}, s_1, s_3]$ is dominated by path $[s_{start}, s_1, s_2, s_3]$ with cost $(3,3)$, it cannot be extended to any Pareto-optimal solution.

In Iterations 12 and 15, the extracted nodes are pruned because their f_2-values are no smaller than the minimum g_2-values of expanded nodes that contain state s_{goal}. Consider node $\langle s_4, (4,10), (10,11)\rangle$ pruned in Iteration 12, which corresponds to path $[s_{start}, s_1, s_2, s_3, s_4]$. The only solution that extends this path is solution $[s_{start}, s_1, s_2, s_3, s_4, s_{goal}]$, and Figure 2.1b has already shown that this solution is not Pareto-optimal.

In Iterations 7, 11, 14, and 17, BOA expands nodes that contain state s_{goal} and finds solutions. Eventually, BOA* returns $Sols$, which consists of the Pareto frontier for this problem instance.*

Algorithm 2: EMOA* and LTMOA*

```
Input  : A problem instance $\langle G, s_{start}, s_{goal} \rangle$ and a consistent heuristic function $\mathbf{h}$
Output: A Pareto frontier
1  $n \leftarrow$ new node with $s(n) = s_{start}$, $\mathbf{g}(n) = \mathbf{0}$, and $p(n) = null$
2  $Open \leftarrow \{n\}$
3  $Sols \leftarrow \emptyset$
4  foreach $s \in S$ do
5      $\mathbf{G}^{\text{tr}}_{\text{cl}}(s) \leftarrow \emptyset$
6  while $Open \neq \emptyset$ do
7      extract a node $n$ from $Open$ with the lexicographically smallest $\mathbf{f}$-value
8      if $\text{IsDominated}(n)$ then
9          continue
10     $\text{Update}(\mathbf{G}^{\text{tr}}_{\text{cl}}(s(n)), Tr(\mathbf{g}(n)))$
11     if $s(n) = s_{goal}$ then
12         add the corresponding solution of $n$ to $Sols$
13         continue
14     for each $e \in out(s(n))$ do
15         $n' \leftarrow$ new node with $s(n') = tar(e)$, $\mathbf{g}(n') = \mathbf{g}(n) + \mathbf{c}(e)$, and $p(n') = n$
16         if $\text{IsDominated}(n')$ then
17             continue
18         add $n'$ to $Open$
19 return $Sols$
20 Function $\text{IsDominated}(n)$:
21     if $\exists \mathbf{x} \in \mathbf{G}^{tr}_{cl}(s_{goal}) : \mathbf{x} \preceq Tr(\mathbf{f}(n))$ then
22         return true
23     if $\exists \mathbf{x} \in \mathbf{G}^{tr}_{cl}(s(n)) : \mathbf{x} \preceq Tr(\mathbf{g}(n))$ then
24         return true
25     return false
26 Function $\text{Update}(\mathbf{X}, \mathbf{y})$:
27     remove the vectors weakly dominated by $\mathbf{y}$ from $\mathbf{X}$
28     add $\mathbf{y}$ to $\mathbf{X}$
```

2.2.5 EMOA*, LTMOA*, and NAMOA*

In this section, we describe some existing best-first multi-objective search algorithms. We begin with Enhanced Multi-Objective A* (EMOA*) [59] and Linear-Time Multi-Objective A* (LTMOA*) [36] because these two algorithms directly follow BOA* and generalize BOA* to more than two objectives. Algorithm 2 shows the pseudocode for them. These two algorithms share the same pseudocode and are only different in how dominance checks are implemented.

The pruning conditions of EMOA* and LTMOA* generalize the pruning conditions of BOA* to more than two objectives. EMOA* and LTMOA* discard a node n if

- (Condition 1) there exists an expanded node that contains state s_{goal} and whose $\mathbf{g}$-value weakly dominates $\mathbf{f}(n)$ or
- (Condition 2) there exists an expanded node that contains state $s(n)$ and whose $\mathbf{g}$-value weakly dominates $\mathbf{g}(n)$.

In each iteration, EMOA* and LTMOA* extract a node from $Open$ with the lexicographically smallest $\mathbf{f}$-value of all nodes in $Open$. Because the heuristic is consistent, the f_1-values of the extracted nodes are monotonically non-decreasing. Therefore, EMOA* and LTMOA* apply dimensionality reduction to dominance checks and do not check the g_1- and f_1-values. Instead of maintaining the set of $\mathbf{g}$-values of all expanded nodes, EMOA* and LTMOA* maintain only the often-significantly-smaller set $\mathbf{G}^{\text{tr}}_{\text{cl}}(s)$ (where "tr" and "cl" stand for "truncated" and "closed," respectively) of undominated truncated $\mathbf{g}$-values for each state s (which is updated on Line 10). Conditions 1 and 2 are checked on Lines 21 and 23, respectively.

The difference between EMOA* and LTMOA* lies in which data structures are used to store $\mathbf{G}^{\text{tr}}_{\text{cl}}(s)$ for each state s and how Lines 21 and 23 are implemented. EMOA* uses an AVL tree to store $\mathbf{G}^{\text{tr}}_{\text{cl}}(s)$ while LTMOA* uses an array to store $\mathbf{G}^{\text{tr}}_{\text{cl}}(s)$. In the special case of three objectives, EMOA* achieves better time complexity than LTMOA* in terms of dominance checks, that is, $O(\log(|\mathbf{G}^{\text{tr}}_{\text{cl}}(s)|))$ versus $O(|\mathbf{G}^{\text{tr}}_{\text{cl}}(s)|)$. However, even in this case, in practice, the runtime overhead of AVL trees makes EMOA* less efficient than LTMOA* [36].

NAMOA* [46] and NAMOA*-dr [54], the variant of NAMOA* that uses dimensionality reduction, are two other existing algorithms. They differ from EMOA* and LTMOA* in that they also discard a node n if there exists some node in $Open$ that contains the same state as n and whose $\mathbf{g}$-value weakly dominates $\mathbf{g}(n)$. These additional checks on $Open$ make NAMOA* and NAMOA*-dr more eager than EMOA* and LTMOA* in terms of dominance checks. However, it is unclear how to apply the dimensionality reduction technique for these dominance checks (on $Open$) because the f_1-values of the generated nodes are not necessarily monotonically non-decreasing. NAMOA* and NAMOA*-dr have been shown to be less efficient than EMOA* and LTMOA* in practice [59, 36].

Among the different algorithms we have described in this section, LTMOA* has been shown to be the most efficient one in practice. Hence, we focus on LTMOA* for the rest of this dissertation.

Chapter 3

Speeding up Multi-Objective Search via Approximation

Many real-world situations admit, or even encourage, a trade-off between efficiency and solution quality. In single-objective search, researchers have investigated bounded-suboptimal search algorithms, which trade off solution quality for efficiency while still guaranteeing that the suboptimality of the reported solution is within a given suboptimality bound. Such algorithms include WA* [53], focal search [50], optimistic search [73], and EES [72]. These algorithms typically rely on different node expansion orders to guide the search and find solutions faster.

In multi-objective search, computing the Pareto frontier can be very time-consuming because its size can be exponential in the size of the graph being searched [19, 10]. Hence, researchers have proposed to compute an ε-approximate Pareto frontier instead [10, 74, 78, 31, 51], that is, a set of solutions such that any solution in the Pareto frontier is ε-dominated by some solution in the set. A solution π ε-dominates a solution π' for a given approximation factor $\varepsilon \geq 0$ iff each cost component of $c(\pi)$ is no larger than $(1+\varepsilon)$ times the corresponding cost component of $c(\pi')$. For the same ε-value, different ε-approximate Pareto frontiers can exist, and they can be of different

This chapter is based on [87].

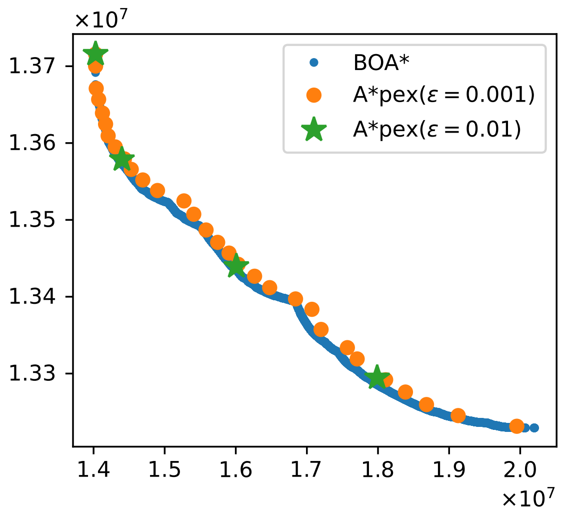


Figure 3.1: The Pareto frontier (9114 solutions), a 0.001-approximate Pareto frontier (30 solutions), and a 0.01-approximate Pareto frontier (4 solutions) computed by BOA*, A*pex with $\varepsilon = 0.001$, and A*pex with $\varepsilon = 0.01$, respectively, for a road-network problem instance with two objectives.

sizes. However, their sizes are typically much smaller than those of the Pareto frontiers, even for small approximation factors, as exemplified later. One can exploit this property to design efficient approximate multi-objective search algorithms.

Although the possibility of speeding up multi-objective search by allowing approximation of the Pareto frontier is intuitive, it remains unclear how to do so efficiently. Existing work [51] has proposed a technique that speeds up the search by pruning paths that can only result in solutions that are ε-dominated by previously computed solutions.[1] This technique can be combined with different multi-objective search algorithms, resulting in approximate variants of these algorithms. However, this technique by itself prunes only a small subset of the paths that can be pruned. PP-A* [31] is a recent approximate bi-objective search algorithm that prunes more paths by using a different data structure for representing paths. However, PP-A* only works with two objectives.

[1]Later in this chapter, we will show why pruning a path to a state s that is ε-dominated by some previously computed path to the same state s does not necessarily yield an ε-approximate Pareto frontier.

In this chapter, we introduce a new approximate multi-objective search algorithm called A*pex. A*pex extends PP-A* by generalizing the data structure used by PP-A* to (1) make PP-A* more efficient for bi-objective search and (2) generalize it to any number of objectives. Our experimental results show that A*pex can outperform state-of-the-art approximate multi-objective search algorithms by orders of magnitude in terms of runtime. They also show that the runtime of A*pex decreases as the given ε-value increases.

As an example, Figure 3.1 shows the solutions computed by different algorithms for a bi-objective search problem instance. While it takes BOA* 170 seconds to compute the Pareto frontier, it takes A*pex, our proposed algorithm, only 15 seconds and 7 seconds to compute the approximate Pareto frontiers with $\varepsilon = 0.001$ and $\varepsilon = 0.01$, respectively. For every solution π', there exists a solution π in the 0.001-approximate (resp. 0.01-approximate) Pareto frontier such that π is at most 0.1% (resp. 1%) worse than π' for all objectives, which is satisfactory for many real-world problems.

This chapter is organized as follows: We begin with the background material for our work in Section 3.1. Next, we provide a detailed description of A*pex in Section 3.2 and its correctness and completeness in Section 3.3. We then provide experimental results in Section 3.4 and our summary in Section 3.5. While this dissertation focuses on the multi-objective search problem, our previous works have also extended the algorithmic techniques behind A*pex to other problems. We end the chapter with two such extensions in Section 3.6.

3.1 Background

In this section, we formally define ε-approximate Pareto frontiers in Section 3.1.1. We then describe previous approaches for computing ε-approximate Pareto frontiers and other approaches for approximating Pareto frontiers, which are different from computing ε-approximate Pareto frontiers.

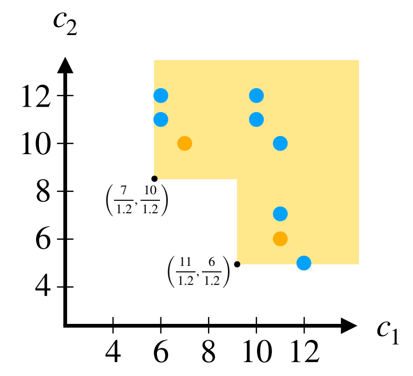
3.1.1 ε-Approximate Pareto Frontiers

Given two vectors $\mathbf{v}$ and $\mathbf{v}'$, $\mathbf{v} \preceq_\varepsilon \mathbf{v}'$ for an *approximation factor* $\varepsilon \in \mathbb{R}_{\geq 0}$ denotes that $v_i \leq (1+\varepsilon)v'_i$ for all $i = 1, 2 \ldots N$. In this case, we say that $\mathbf{v}$ *ε-dominates* $\mathbf{v}'$. $\mathbf{v}$ *ε-dominates* $\mathbf{v}'$ for any ε-value if $\mathbf{v}$ weakly dominates $\mathbf{v}'$, but the opposite is not necessarily true. We say that a path π ε-dominates another path π', denoted as $\pi \preceq_\varepsilon \pi'$, iff $\mathbf{c}(\pi) \preceq_\varepsilon \mathbf{c}(\pi')$.

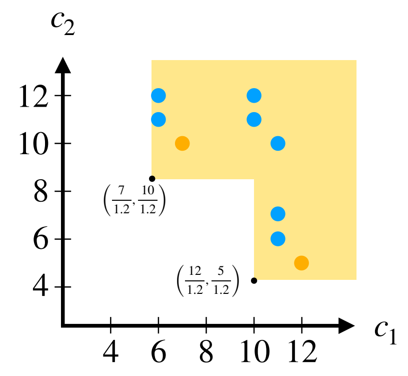
Definition 3.1. *Consider a multi-objective search problem instance and an approximation factor ε. An ε-approximate Pareto frontier is a set of solutions such that, for any Pareto-optimal solution π', there exists a solution π in the ε-approximate Pareto frontier with $\pi \preceq_\varepsilon \pi'$.*

The Pareto frontier is a 0-approximate Pareto frontier, but the opposite is not necessarily true. For example, the set of all solutions is also a 0-approximate Pareto frontier. Our task is to efficiently find an explicit representation, ideally of small size, of an ε-approximate Pareto frontier for a given instance. A multi-objective search algorithm that finds an ε-approximate Pareto frontier is also called an (ε-)*approximate multi-objective search algorithm.*

We can generalize the definitions of ε-dominance and ε-approximate Pareto frontier to allow for different approximation factors for different objectives. Our proposed algorithm, A*pex, can



(a)



(b)

Figure 3.2: Two different 0.2-approximate Pareto frontiers for the problem instance in Figure 2.1. The orange dots show the solutions in each 0.2-approximate Pareto frontier, and the shaded regions show the regions that are 0.2-dominated by the 0.2-approximate Pareto frontiers.

directly work with these generalized definitions while retaining its theoretical properties in Section 3.3. However, for ease of exposition, we will stick to the above definitions in this dissertation.

Example 4. *Figure 3.2 shows two 0.2-approximate Pareto frontiers for the problem instance in Figure 2.1. Each 0.2-approximate Pareto frontier consists of two solutions, shown by the orange dots. Each solution of the problem instance is 0.2-dominated by at least one solution in the 0.2-approximate Pareto frontier. For example, consider the 0.2-approximate Pareto frontier in Figure 3.2a. The solution with cost $(6, 11)$ is 0.2-dominated by the top-left orange solution with cost $(7, 10)$ because $7 \leq (1 + 0.2) \times 6$ and $10 \leq (1 + 0.2) \times 11$. Note that although both 0.2-approximate Pareto frontiers in this example contain only Pareto-optimal solutions, this is not required by the definition of approximate Pareto frontiers.*

Algorithm 3: The IsDominated function for LTMOA*-ε

```
1 Function IsDominated($n$):
2     if $\exists \mathbf{x} \in \mathbf{G}^{tr}_{cl}(s_{goal}) : \mathbf{x} \preceq_{\varepsilon} Tr(\mathbf{f}(n))$ then
3         return true
4     if $\exists \mathbf{x} \in \mathbf{G}^{tr}_{cl}(s(n)) : \mathbf{x} \preceq Tr(\mathbf{g}(n))$ then
5         return true
6     return false
```

3.1.2 Relaxing the Pruning Condition

Perny and Spanjaard [51] suggest computing an ε-approximate Pareto frontier by relaxing the pruning condition of a multi-objective search algorithm that computes the Pareto frontier: Instead of discarding nodes whose $\mathbf{f}$-values are weakly dominated by the cost of a previously found solution, we discard nodes whose $\mathbf{f}$-values are ε-dominated by the cost of a previously found solution. Doing so allows the search algorithm to prune more nodes and only requires a slight modification to the algorithm. For example, Algorithm 3 shows the modified IsDominated function for a variant of LTMOA* that adopts this relaxed pruning condition, called LTMOA*-ε. The only change is on Lines 2-3, where the modified IsDominated function returns true if there exists a vector in $\mathbf{G}^{\mathrm{tr}}_{\mathrm{cl}}(s_{goal})$ that ε-dominates the truncated $\mathbf{f}$-value of the given node n. Existing work has also applied this technique to BOA*, resulting in an approximate variant of BOA* called BOA*-ε [31]. The only difference between BOA* and BOA*-ε is on Line 21 of Algorithm 1, where BOA*-ε discards a node n if $g_2^{\min}(s_{goal})$ is no larger than $(1+\varepsilon)f_2(n)$.

One might consider further relaxing the pruning condition of a multi-objective search algorithm by discarding a node n if its g-value is ε-dominated by the g-value of some expanded node n' that contains state $s(n)$. For BOA*, this would require a change to Line 23 of Algorithm 1, where the algorithm now discards a node n if $g_2^{\min}(s(n))$ is no larger than $(1+\varepsilon)g_2(n)$. However,

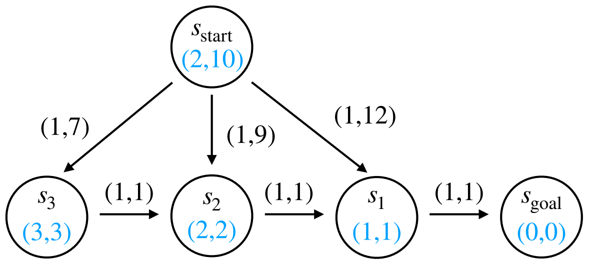


Figure 3.3: An example bi-objective search problem instance which shows that pruning a node n if its $\mathbf{g}$-value is ε-dominated by the $\mathbf{g}$-value of some expanded node n' that contains state $s(n)$ does not necessarily yield an ε-approximate Pareto frontier. The pair of numbers inside each state shows the value of the perfect-distance heuristic.

Iter	*Open* $\langle s(n), \mathbf{g}(n), \mathbf{f}(n)\rangle$	Generated $\langle s(n), \mathbf{g}(n), \mathbf{f}(n)\rangle$	Update of $g_2^{\min}(s(n))$
1	$\langle s_{start}, (0,0), (2,10)\rangle *$	$\langle s_1, (1,12), (2,13)\rangle$ $\langle s_2, (1,9), (3,11)\rangle$ $\langle s_3, (1,7), (4,10)\rangle$	$g_2^{\min}(s_{start}) = 0$
2	$\langle s_1, (1,12), (2,13)\rangle *$ $\langle s_2, (1,9), (3,11)\rangle$ $\langle s_3, (1,7), (4,10)\rangle$	$\langle s_{goal}, (2,13), (2,13)\rangle$	$g_2^{\min}(s_1) = 12$
3	$\langle s_{goal}, (2,13), (2,13)\rangle *$ $\langle s_2, (1,9), (3,11)\rangle$ $\langle s_3, (1,7), (4,10)\rangle$		$g_2^{\min}(s_{goal}) = 13$
4	$\langle s_2, (1,9), (3,11)\rangle *$ $\langle s_3, (1,7), (4,10)\rangle$	$\langle s_1, (2,10), (3,11)\rangle$ (pruned)	$g_2^{\min}(s_2) = 9$
5	$\langle s_3, (1,7), (4,10)\rangle *$	$\langle s_2, (2,8), (4,10)\rangle$ (pruned)	$g_2^{\min}(s_3) = 7$
6	empty		

Table 3.1: Trace of $Open$, generated nodes, and $g_2^{\min}$ in each iteration of Example 5 when solving the example problem instance in Figure 3.3. "$*$" marks the node that is extracted in that iteration.

the following counter-example shows that doing so does not necessarily yield an ε-approximate Pareto frontier.

Example 5. *Figure 3.3 shows an example bi-objective search problem instance. This problem instance has three solutions, namely, solution $\pi_1 = [s_{start}, s_1, s_{goal}]$ with cost $(2,13)$, solution $\pi_2 = [s_{start}, s_2, s_1, s_{goal}]$ with cost $(3,11)$, and solution $\pi_3 = [s_{start}, s_3, s_2, s_1, s_{goal}]$ with cost $(4,10)$. We consider the variant of BOA* that discards a node n if $g_2^{min}(s(n))$ is no larger than $(1+\varepsilon)g_2(n)$ and* *assume that $\varepsilon = 0.2$. Table 3.1 shows the trace of $Open$, generated nodes, and changes to g_2^{min} in each iteration of this variant of BOA*.*

In Iteration 4, node $\langle s_2, (1,9), (3,11)\rangle$ is expanded. Its child node $\langle s_1, (2,10), (3,11)\rangle$ is pruned because $g_2^{min}(s_1) = 12$ is no larger than $(1+\varepsilon)$ times the g_2-value of this child node, which is $(1+0.2)\cdot 10 = 12$. Similarly, in Iteration 5, node $\langle s_3, (1,7), (4,10)\rangle$ is expanded. Its child node $\langle s_2, (2,8), (4,10)\rangle$ is pruned because $g_2^{min}(s_2) = 9$ is no larger than $(1+\varepsilon)$ times the g_2-value of this child node, which is $(1+0.2)\cdot 8 = 9.6$.

In Iteration 3, the algorithm finds solution $\pi_1 = [s_{start}, s_1, s_{goal}]$. Eventually, it returns $Sols$, which contains only this one solution. However, $Sols$ is not a 0.2-approximate Pareto frontier because $\pi_3 = [s_{start}, s_3, s_2, s_1, s_{goal}]$ with cost $(4,10)$ is not 0.2-dominated by solution π_1 with cost $(2,13)$.

Intuitively, the algorithm goes wrong when it discards a path ($[s_{start}, s_3, s_2]$, which corresponds to node $\langle s_2, (2,8), (4,10)\rangle$) that can be extended to solution π_3 because it is 0.2-dominated by some path ($[s_{start}, s_2]$, which corresponds to node $\langle s_2, (1,9), (3,11)\rangle$) that can be extended to solution π_2 in Iteration 5. It also discards the path ($[s_{start}, s_2, s_1]$, which corresponds to node $\langle s_1, (2,10), (3,11)\rangle$) that can be extended to solution π_2 because it is 0.2-dominated by some path ($[s_{start}, s_1]$, which corresponds to node $\langle s_1, (1,12), (2,13)\rangle$) that can be extended to solution π_1 in Iteration 4. However, the ε-dominance relation is not transitive: Although π_1 0.2-dominates π_2 and π_2 0.2-dominates π_3, π_1 does not 0.2-dominate π_3. Therefore, a 0.2-approximate Pareto frontier for this problem instance needs to contain solution π_2 or π_3.

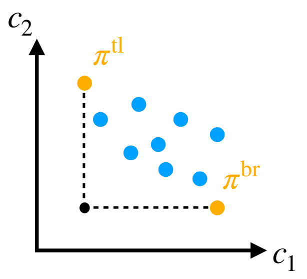


Figure 3.4: An example of path pair $\langle \pi^{\text{tl}}, \pi^{\text{br}} \rangle$ (orange) and the set of paths that it represents (orange and blue). Its g-value $(c_1(\pi^{\text{tl}}), c_2(\pi^{\text{br}}))$ (black) weakly dominates the costs of all represented paths.

3.1.3 PP-A*

PP-A* [31] is an approximate bi-objective search algorithm that finds an ε-approximate Pareto frontier for a given approximation factor ε. Similar to BOA*, PP-A* maintains an $Open$ list. Each node in $Open$ corresponds to a *path pair* $\mathcal{PP} = \langle \pi^{\text{tl}}, \pi^{\text{br}} \rangle$ (where "tl" and "br" stand for "top-left" and "bottom-right," respectively) with $s(\pi^{\text{tl}}) = s(\pi^{\text{br}})$, $c_1(\pi^{\text{tl}}) \leq c_1(\pi^{\text{br}})$, and $c_2(\pi^{\text{tl}}) \geq c_2(\pi^{\text{br}})$. The path pair contains state $s(\mathcal{PP}) = s(\pi^{\text{br}})$ and $\mathbf{g}$-value (also called apex) $\mathbf{g}(\mathcal{PP}) = (c_1(\pi^{\text{tl}}), c_2(\pi^{\text{br}}))$. Its $\mathbf{f}$-value is defined as $\mathbf{f}(\mathcal{PP}) = \mathbf{g}(\mathcal{PP}) + \mathbf{h}(s(\mathcal{PP}))$. It is *$\varepsilon$-bounded* iff $c_1(\pi^{\text{br}}) \leq (1+\varepsilon)c_1(\pi^{\text{tl}})$ and $c_2(\pi^{\text{tl}}) \leq (1+\varepsilon)c_2(\pi^{\text{br}})$ or, equivalently, iff the costs of both paths π^{tl} and π^{br} ε-dominate the $\mathbf{g}$-value of path pair $\mathcal{PP}$.

While BOA* reasons about single paths, PP-A* represents sets of paths with the same last state and similar costs as ε-bounded path pairs, which results in small numbers of path pair expansions and thus small runtimes. During the search, PP-A* maintains the following properties of a path pair and the set of paths that it represents: For an ε-bounded path pair $\langle \pi^{\text{tl}}, \pi^{\text{br}} \rangle$, the cost $\mathbf{c}(\pi^{\text{tl}}) = (c_1(\pi^{\text{tl}}), c_2(\pi^{\text{tl}}))$ of top-left path π^{tl} is the lexicographically smallest cost of all paths in this set, and the vector $(c_2(\pi^{\text{br}}), c_1(\pi^{\text{br}}))$ of bottom-right path π^{br} (called its *reverse cost*) is the lexicographically smallest such vector of all paths in this set. Also, the $\mathbf{g}$-value $(c_1(\pi^{\text{tl}}), c_2(\pi^{\text{br}}))$

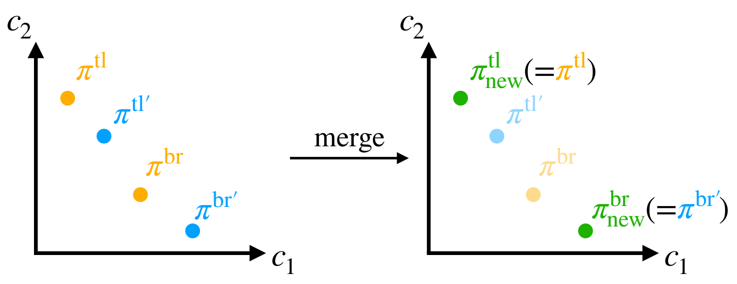


Figure 3.5: An example of merging path pairs $\langle \pi^{\text{tl}}, \pi^{\text{br}} \rangle$ (orange) and $\langle \pi^{\text{tl}'}, \pi^{\text{br}'} \rangle$ (blue) into path pair $\langle \pi^{\text{tl}}_{\text{new}}, \pi^{\text{br}}_{\text{new}} \rangle$ (green).

of the path pair weakly dominates the costs of all paths in this set (because $c_1(\pi^{\text{tl}})$ is the smallest c_1-value of all paths in this set and $c_2(\pi^{\text{br}})$ is the smallest c_2-value of them), and both the top-left and bottom-right paths ε-dominate all paths in this set [31]. See Figure 3.4 for a visualization of a path pair and the set of paths that it represents.

Any two path pairs containing the same state can be *merged* into a single path pair, where the top-left path of the merged path pair is the top-left path of the two path pairs with the lexicographically smaller cost and the bottom-right path of the merged path pair is the bottom-right path of the two path pairs with the lexicographically smaller reverse cost. See Figure 3.5 for a visualization of the outcome.

Algorithm 4 shows the pseudocode of PP-A*. It starts with a single path pair $\langle [s_{\text{start}}], [s_{\text{start}}] \rangle$ in $Open$ (Line 1). In each iteration, PP-A* extracts a path pair from $Open$ with the lexicographically smallest $\mathbf{f}$-value (Line 6). Both after extracting (that is, after Line 6) and after generating (that is, after Line 17) a path pair, PP-A* performs dominance checks with the same pruning strategy of BOA*-ε, that is, PP-A* discards a path pair $\mathcal{PP}$ if $g_2^{\min}(s_{goal})$ is no larger than $(1+\varepsilon)f_2(\mathcal{PP})$ (Line 20) or $g_2^{\min}(s(\mathcal{PP}))$ is no larger than $g_2(\mathcal{PP})$ (Line 22).

Algorithm 4: PP-A*

```
Input  : A bi-objective problem instance $\langle G, s_{start}, s_{goal}\rangle$, a consistent heuristic function
         $\mathbf{h}$, and an approximation factor $\varepsilon$
Output: An $\varepsilon$-approximate Pareto frontier
1  $Open \leftarrow \{\langle [s_{\text{start}}], [s_{\text{start}}]\rangle\}$
2  $Sols \leftarrow \emptyset$
3  foreach $s \in S$ do
4      $g_2^{\text{min}}(s) \leftarrow \infty$
5  while $Open \neq \emptyset$ do
6      extract a path pair $\mathcal{PP} = \langle \pi^{\text{tl}}, \pi^{\text{br}}\rangle$ from $Open$ with the lexicographically smallest
        $\mathbf{f}$-value
7      if IsDominated($\mathcal{PP}$) then
8          continue
9      $g_2^{\text{min}}(s(\mathcal{PP})) \leftarrow g_2(\mathcal{PP})$
10     if $s(\mathcal{PP}) = s_{goal}$ then
11         add $\pi^{\text{br}}$ to $Sols$
12         continue
13     for each $e \in out(s(\mathcal{PP}))$ do
14         $\mathcal{PP}' \leftarrow \langle \text{extend}(\pi_{\text{tl}}, e), \text{extend}(\pi_{\text{br}}, e)\rangle$
15         if IsDominated($\mathcal{PP}'$) then
16             continue
17         AddToOpen($\mathcal{PP}'$, $Open$)
18 return $Sols$
19 Function IsDominated($\mathcal{PP}$):
20     if $g_2^{min}(s_{goal}) \leq (1+\varepsilon) f_2(\mathcal{PP})$ then
21         return true
22     if $g_2^{min}(s(\mathcal{PP})) \leq g_2(\mathcal{PP})$ then
23         return true
24     return false
25 Function AddToOpen($\mathcal{PP}$):
26     for $\mathcal{PP}' \in Open[s(\mathcal{PP})]$ do
27         $\mathcal{PP}_{\text{new}} \leftarrow \text{merge}(\mathcal{PP}, \mathcal{PP}')$
28         if $\mathcal{PP}_{new}$ is $\varepsilon$-bounded then
29             remove $\mathcal{PP}'$ from $Open$
30             add $\mathcal{PP}_{\text{new}}$ to $Open$
31             return
32     add $\mathcal{PP}$ to $Open$
33     return
```

Let $\mathcal{PP}$ be a path pair extracted by PP-A* from $Open$ and not discarded after the dominance checks. PP-A* then expands $\mathcal{PP}$: If $s(\mathcal{PP}) = s_{goal}$, PP-A* adds its bottom-right path to $Sols$

(Line 11). Otherwise, PP-A* generates a child path pair $\mathcal{PP}'$ for each out-edge e of state $s(\mathcal{PP})$. The top-left and bottom-right paths of path pair $\mathcal{PP}'$ are the paths that extend the top-left and bottom-right paths, respectively, of path pair $\mathcal{PP}$ with edge e (Line 14). Let $Open[s]$ be the set of path pairs in $Open$ that contain state s. PP-A* checks on Line 17 if there exists a path pair in $Open[s(\mathcal{PP}')]$ and results in an ε-bounded path pair when merged with path pair $\mathcal{PP}'$. If so, PP-A* removes that path pair from $Open$ and then adds the merged path pair to $Open$ (Lines 29-30). Otherwise, it adds path pair $\mathcal{PP}'$ to $Open$ (Line 32). When $Open$ becomes empty, PP-A* terminates and returns $Sols$ as an ε-approximate Pareto frontier (Line 18).

3.1.4 Other Related Work

Existing work [10, 74, 78, 51] has investigated Fully Polynomial-Time Approximation Schemes (FPTAS) in the context of ε-approximate multi-objective search. The runtime complexities of these FPTAS algorithms are often polynomial in the size of the graph and $1/\varepsilon$. Unfortunately, these algorithms are still impractical for large graphs, such as road networks, that often have millions of states.

Existing work has also studied other schemes for approximating the Pareto frontier. One such scheme is to compute the set Π_{support} of all supported solutions [64, 88]. A *supported solution* is a solution that minimizes a convex combination[2] of the different objectives. A supported solution is a Pareto-optimal solution but not necessarily vice versa. Existing algorithms for computing Π_{support} include an algorithm based on Dijkstra's Algorithm [64] and an algorithm that performs a series of single-objective searches [88]. Rivera, Baier, and Hernández [61] propose another scheme for bi-objective search, which transforms the given bi-objective problem instance P to

[2]A convex combination is a linear combination where all coefficients are non-negative and sum to 1.

another bi-objective problem instance P' whose costs are two different linear combinations of the costs in P [61]. They show that the Pareto frontier for P' can be a much smaller subset of the Pareto frontier for P and hence can be much faster to compute. Unlike ε-approximate multi-objective search algorithms, these approximate schemes do not guarantee that they produce solutions that ε-dominate every Pareto-optimal solution.

3.2 A*pex

In this section, we introduce A*pex, a best-first multi-objective search algorithm that finds an ε-approximate Pareto frontier for a given approximation factor ε.

In A*pex, as in PP-A*, nodes correspond to sets of paths with the same last state and similar costs. But A*pex improves on the representation of these sets, which (1) allows for larger and thus fewer sets of paths during the search and thus results potentially in a search that is more efficient and (2) generalizes PP-A* from bi-objective search to multi-objective search with any number of objectives. PP-A* includes only one path of a path pair in the solution set, namely the bottom-right path. The top-left path is not used for this purpose. The $\mathbf{g}$-value of a path pair can be viewed as a vector whose first component is the smallest c_1-value of all paths in the set of paths that the path pair represents and whose second component is the smallest c_2-value of all paths in this set. Again, the top-left path is not used for this purpose. Therefore, in A*pex, we choose to represent a set of paths with a single representative path and a $\mathbf{g}$-value that is similar to the one of PP-A*. Having only one representative path instead of two provides flexibility. For example, the representative path can be chosen more freely than by PP-A*, as we will show in Section 3.2.2. The representation of a set of paths can now easily be generalized from two to any

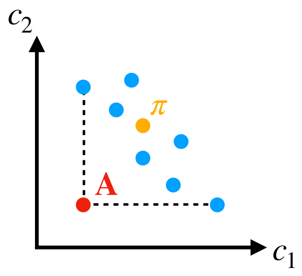


Figure 3.6: An example of apex-path pair $\mathcal{AP} = \langle \mathbf{A}, \pi \rangle$ (red and orange) and the set of paths $\Pi_{\mathcal{AP}}$ that it represents (orange and blue). Apex $\mathbf{A}$ weakly dominates the costs of all represented paths.

number of objectives by extending the $\mathbf{g}$-value from a vector of size two to a vector of a size that equals the number of objectives. The ith component of the $\mathbf{g}$-value is the smallest c_i-value of all paths in this set.

3.2.1 Apex-Path Pairs

Each node of A*pex is an ε-bounded *apex-path pair* $\mathcal{AP} = \langle \mathbf{A}, \pi \rangle$, where $\mathbf{A}$ is a N-dimensional vector and π is a *representative path* with $\mathbf{A} \preceq \mathbf{c}(\pi)$. The apex-path pair contains state $s(\mathcal{AP}) = s(\pi)$ and $\mathbf{g}$-value (called apex) $\mathbf{g}(\mathcal{AP}) = \mathbf{A}$. Its $\mathbf{f}$-value is defined as $\mathbf{f}(\mathcal{AP}) = \mathbf{g}(\mathcal{AP}) + \mathbf{h}(s(\mathcal{AP}))$. An apex-path pair $\mathcal{AP} = \langle \mathbf{A}, \pi \rangle$ is *ε-bounded* iff $\mathbf{f}(\pi) \preceq_\varepsilon \mathbf{f}(\mathcal{AP})$, where the $\mathbf{f}$-value of path π is defined as $\mathbf{f}(\pi) = \mathbf{c}(\pi) + \mathbf{h}(s(\pi))$.

In A*pex, each apex-path pair $\mathcal{AP} = \langle \mathbf{A}, \pi \rangle$ corresponds to a set of paths $\Pi_{\mathcal{AP}}$ (which includes π) with the same last state $s(\mathcal{AP})$. A*pex does not store $\Pi_{\mathcal{AP}}$ explicitly during the search. Instead, it only stores the apex $\mathbf{A} = \min_{\pi' \in \Pi_{\mathcal{AP}}} \{\mathbf{c}(\pi')\}$, which is the component-wise minimum of (and hence weakly dominates) the costs of all paths in $\Pi_{\mathcal{AP}}$, and a representative path $\pi \in \Pi_{\mathcal{AP}}$. See Figure 3.6 for a visualization of an apex-path pair and the set of paths that it represents. The following property shows that using an ε-bounded apex-path pair $\mathcal{AP}$ to represent $\Pi_{\mathcal{AP}}$ does not prevent us from finding a solution that ε-dominates any solution that extends some path in $\Pi_{\mathcal{AP}}$.

Property 3.1. *Consider an apex-path pair $\mathcal{AP} = \langle \mathbf{A}, \pi \rangle$ and the set of paths $\Pi_{\mathcal{AP}}$ that $\mathcal{AP}$ corresponds to. If $\mathcal{AP}$ is ε-bounded, every solution that extends some path $\pi' \in \Pi_{\mathcal{AP}}$ is ε-dominated by some solution that extends π.*

Proof. By the definitions of $\mathcal{AP}$ and $\Pi_{\mathcal{AP}}$, π and π' have the same last state $s(\mathcal{AP})$. Because $\mathcal{AP}$ is ε-bounded, we have

$$\begin{aligned} \mathbf{f}(\pi) &\preceq_\varepsilon \mathbf{f}(\mathcal{AP}) \\ &= \mathbf{g}(\mathcal{AP}) + \mathbf{h}(s(\mathcal{AP})) \\ &= \mathbf{A} + \mathbf{h}(s(\mathcal{AP})). \end{aligned} \tag{3.1}$$

Because $\mathbf{A}$ weakly dominates the costs of all paths in $\Pi_{\mathcal{AP}}$ (including π'), we have $\mathbf{A} \preceq \mathbf{c}(\pi')$. Combining this property and Eq. 3.1, we have

$$\begin{aligned} \mathbf{f}(\pi) &\preceq_\varepsilon \mathbf{c}(\pi') + \mathbf{h}(s(\mathcal{AP})) \\ &= \mathbf{f}(\pi'). \end{aligned} \tag{3.2}$$

Consider any solution π'_{sol} that extends π'. Because $\mathbf{h}$ is required to be consistent, we have $\mathbf{f}(\pi') \preceq \mathbf{c}(\pi'_{\text{sol}})$. Let $\boldsymbol{\delta}$ denote $\mathbf{c}(\pi'_{\text{sol}}) - \mathbf{f}(\pi')$, which is a non-negative vector. The cost of π'_{sol} can then be represented as $\mathbf{c}(\pi'_{\text{sol}}) = \mathbf{f}(\pi') + \boldsymbol{\delta}$. Consider solution π_{sol} which extends π with the path π_{ext} that extends π' to π'_{sol}. The cost of π_{ext} is

$$\begin{aligned}
\mathbf{c}(\pi_{\text{ext}}) &= \mathbf{c}(\pi'_{\text{sol}}) - \mathbf{c}(\pi') \\
&= \mathbf{f}(\pi') + \boldsymbol{\delta} - \mathbf{c}(\pi') \\
&= \mathbf{c}(\pi') + \mathbf{h}(s(\pi')) + \boldsymbol{\delta} - \mathbf{c}(\pi') \\
&= \boldsymbol{\delta} + \mathbf{h}(s(\pi')) \\
&= \boldsymbol{\delta} + \mathbf{h}(s(\mathcal{AP})).
\end{aligned}$$

The cost of π_{sol} can then be represented as

$$\begin{aligned}
\mathbf{c}(\pi_{\text{sol}}) &= \mathbf{c}(\pi) + \mathbf{c}(\pi_{\text{ext}}) \\
&= \mathbf{c}(\pi) + \boldsymbol{\delta} + \mathbf{h}(s(\mathcal{AP})) \\
&= \mathbf{f}(\pi) + \boldsymbol{\delta}.
\end{aligned}$$

Because $\boldsymbol{\delta} \preceq_\varepsilon \boldsymbol{\delta}$ (which holds because $\boldsymbol{\delta}$ is a non-negative vector) and also because $\mathbf{f}(\pi) \preceq_\varepsilon \mathbf{f}(\pi')$ (Eq. 3.2), we have $\mathbf{f}(\pi) + \boldsymbol{\delta} \preceq_\varepsilon \mathbf{f}(\pi') + \boldsymbol{\delta}$, which is equivalent to $\mathbf{c}(\pi_{\text{sol}}) \preceq_\varepsilon \mathbf{c}(\pi'_{\text{sol}})$ $\square$

We can extend an apex-path pair $\mathcal{AP} = \langle \mathbf{A}, \pi \rangle$ by an edge e. Let $\mathcal{AP}'$ denote the resulting apex-path pair. The apex of $\mathcal{AP}'$ is the sum of $\mathbf{A}$ and $\mathbf{c}(e)$, and the representative path of $\mathcal{AP}'$ is path π extended by edge e. Conceptually, apex-path pair $\mathcal{AP}'$ corresponds to the set of paths $\Pi_{\mathcal{AP}'}$

that extends every path in $\Pi_{\mathcal{AP}}$ by e. It is easy to verify that the apex of $\mathcal{AP}'$ is the component-wise minimum of the costs of the paths in $\Pi_{\mathcal{AP}'}$. The following property shows that the extend operation preserves the ε-boundedness of apex-path pairs.

Property 3.2. *Consider the apex-path pair $\mathcal{AP}' = \langle \mathbf{A}', \pi' \rangle$ resulting from extending apex-path pair $\mathcal{AP} = \langle \mathbf{A}, \pi \rangle$ with an edge e. $\mathcal{AP}'$ is ε-bounded if $\mathcal{AP}$ is ε-bounded.*

Proof. Let s and s' denote the states that $\mathcal{AP}$ and $\mathcal{AP}'$ contain, respectively. Assume that $\mathcal{AP}$ is ε-bounded, that is, it satisfies

$$\begin{aligned}\mathbf{f}(\pi) &\preceq_\varepsilon \mathbf{f}(\mathcal{AP})\\ \mathbf{f}(\pi) &\preceq_\varepsilon \mathbf{g}(\mathcal{AP}) + \mathbf{h}(s)\\ \mathbf{c}(\pi) + \mathbf{h}(s) &\preceq_\varepsilon \mathbf{A} + \mathbf{h}(s),\end{aligned}$$

and, hence,

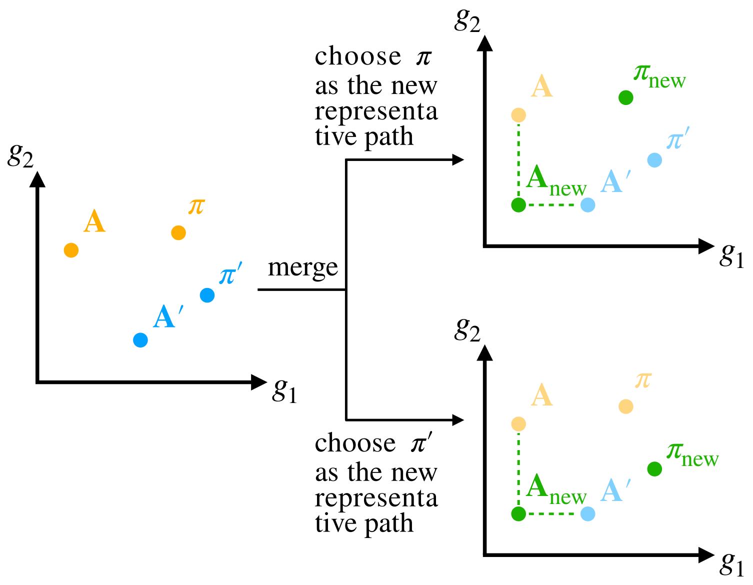
$$c_i(\pi) + h_i(s) \leq (1+\varepsilon) \cdot (A_i + h_i(s)), i = 1, 2 \ldots N. \tag{3.3}$$

Because the heuristic function $\mathbf{h}$ is consistent, we have $h_i(s) \leq c_i(e) + h_i(s')$ for all $i = 1, 2 \ldots N$. Let δ denote $c_i(e) + h_i(s') - h_i(s)$. We have $\delta \geq 0$ and hence $\delta \leq (1+\varepsilon) \cdot \delta$. By adding δ and $(1+\varepsilon) \cdot \delta$ to the left and the right sides of Eq. 3.3, respectively, we have

$$\begin{gathered}c_i(\pi) + h_i(s) + \delta \leq (1+\varepsilon) \cdot (A_i + h_i(s) + \delta)\\ c_i(\pi) + h_i(s) + c_i(e) + h_i(s') - h_i(s) \leq (1+\varepsilon) \cdot (A_i + h_i(s) + c_i(e) + h_i(s') - h_i(s))\\ c_i(\pi) + c_i(e) + h_i(s') \leq (1+\varepsilon) \cdot (A_i + c_i(e) + h_i(s'))\end{gathered}$$

Algorithm 5: The merge function for apex-path pairs

```
1 Function merge($\mathcal{AP} = \langle \mathbf{A}, \pi \rangle, \mathcal{AP}' = \langle \mathbf{A}', \pi' \rangle$):
2     $\mathbf{A}_{\text{new}} \leftarrow [min(A_1, A'_1), min(A_2, A'_2) \dots min(A_N, A'_N)]$
3     $\pi_{\text{new}} \leftarrow$ choose a path between $\pi$ and $\pi'$
4     return $\langle \mathbf{A}_{\text{new}}, \pi_{\text{new}} \rangle$
```



g_2
A
π
π'
A'
g_1
merge
choose π as the new representa tive path
choose π' as the new representa tive path
π_{new}
$\mathbf{A}_{\text{new}}$

Figure 3.7: An example of merging apex-path pairs $\langle \mathbf{A}, \pi \rangle$ (orange) and $\langle \mathbf{A}', \pi' \rangle$ (blue) into Apex-path pair $\langle \mathbf{A}_{\text{new}}, \pi_{\text{new}} \rangle$ (green).

for all $i = 1, 2 \dots N$, that is, $\mathbf{c}(\pi) + \mathbf{c}(e) + \mathbf{h}(s') \preceq_\varepsilon \mathbf{A} + \mathbf{c}(e) + \mathbf{h}(s')$. Because $\mathbf{c}(\pi') = \mathbf{c}(\pi) + \mathbf{c}(e)$ and $\mathbf{A}' = \mathbf{A} + \mathbf{c}(e)$, we have $\mathbf{f}(\pi') \preceq_\varepsilon \mathbf{f}(\mathcal{AP}')$. Thus, $\mathcal{AP}'$ is ε-bounded. □

3.2.2 Merge Operation

Similar to PP-A*, A*pex can merge two apex-path pairs that contain the same state. Algorithm 5 shows the merge function for apex-path pairs. Conceptually, merging two apex-path pairs corresponds to merging the two sets of paths that these two apex-path pairs correspond to. Hence, the apex of the merged apex-path pair is the component-wise minimum of the apexes of the two apex-path pairs (Line 2). The representative path of the merged apex-path pair is either one of the two representative paths of the two apex-path pairs (Line 3). See Figure 3.7 for a visualization of the two possible outcomes. Each of the two representative paths is considered a candidate for the representative path of the merged apex-path pair if choosing it results in an ε-bounded merged apex-path pair. Which candidate A*pex chooses on Line 3 does not affect its correctness but can affect its efficiency. We consider the following methods:

- **Random method**: A*pex randomly chooses the representative path from the candidates.

- **Lexicographically smallest reverse $\mathbf{g}$-value method**: A*pex chooses the representative path with the lexicographically smaller reverse $\mathbf{g}$-value. If this path is not a candidate, A*pex does not merge the apex-path pairs. In the bi-objective case, this method is similar to how PP-A* merges path pairs, namely by picking the bottom-right path.

- **Greedy method**: A*pex chooses the candidate π with the larger slack

  $$\min_{i=1,2\ldots N}\left\{\frac{1+\varepsilon-f_i(\pi)/f_i(\mathcal{AP})}{\varepsilon}\right\},$$

  where $\mathcal{AP}$ is the resulting merged apex-path pair. The ith component of the $\mathbf{f}$-value of the representative path could be a factor of $1+\varepsilon$ larger than the ith component of the $\mathbf{f}$-value of the apex-path pair but is only a factor of $f_i(\pi)/f_i(\mathcal{AP})$ larger. The smaller the difference between these two values, the better path π utilizes the leeway provided by the approximation factor. The difference, which can range from zero to ε, is divided by ε to normalize it and thus make the differences for different components comparable. Overall, the expression above indicates how much room is left to merge the merged apex-path pair with other apex-path pairs in the future, and maximizing it chooses the candidate that leaves more room for future merges.

3.2.3 Search Strategy

Algorithm 6 shows the pseudocode of A*pex. A*pex starts with a single apex-path pair $\langle \mathbf{0}, [s_{\text{start}}] \rangle$ in $Open$ (Line 1). In each iteration, A*pex extracts an apex-path pair from $Open$ with the lexicographically smallest $\mathbf{f}$-value (Line 7). Both after extracting an apex-path pair from $Open$ (that is, after Line 7) and before attempting to add an apex-path pair to $Open$ (that is, before Line 20), A*pex performs dominance checks. A*pex uses the dimensionality reduction technique for the dominance checks. It maintains a set $\mathbf{G}^{\text{tr}}_{\text{cl}}(s)$ for each state s that contains the undominated truncated $\mathbf{g}$-values (or, in the bi-objective case, $g_2^{\min}(s)$ for each state s that stores the minimum g_2-value) of all expanded apex-path pairs that contain state s. Additionally, A*pex maintains the set $\mathbf{C}^{\text{tr}}_{\text{sol}}$ of undominated truncated costs of $Sols$.[3] Its pruning strategy is similar to the ones of LT-MOA* and EMOA*. Let $\mathcal{AP}$ denote the apex-path pair being checked, A*pex discards apex-path pair $\mathcal{AP}$ if:

1. (Condition 1) there exists a solution in $Sols$ whose cost ε-dominates $\mathbf{f}(\mathcal{AP})$ or
2. (Condition 2) there exists an expanded apex-path pair that contains the same state as $\mathcal{AP}$ and whose $\mathbf{g}$-value weakly dominates $\mathbf{g}(\mathcal{AP})$.

[3]PP-A* does not need to maintain such an additional data structure for the following reason: In PP-A*, $g_2^{\min}(s_{goal})$ is the minimum g_2-value of all expanded path pairs that contain state s_{goal}. For each such expanded path pair, its g_2-value is equal to the c_2-value of its bottom-right path, and its bottom-right path is a solution in $Sols$. Therefore, $g_2^{\min}(s_{goal})$ is equal to the minimum c_2-value of $Sols$. For A*pex, each vector in $\mathbf{G}^{\text{tr}}_{\text{cl}}(s_{goal})$ is the truncated $\mathbf{g}$-value of an expanded apex-path pair that contains state s_{goal}, which is not necessarily equal to the truncated cost of the representative path of this apex-path pair.

Algorithm 6: A*pex

```
Input  : A problem instance $\langle G, s_{start}, s_{goal}\rangle$, a consistent heuristic function $\mathbf{h}$, and an
         approximation factor $\varepsilon$
Output: An $\varepsilon$-approximate Pareto frontier
1  $Open \leftarrow \{\langle \mathbf{0}, [s_{\text{start}}]\rangle\}$
2  $Sols \leftarrow \emptyset$
3  $\mathbf{C}^{\text{tr}}_{\text{sol}} \leftarrow \emptyset$
4  foreach $s \in S$ do
5      $\mathbf{G}^{\text{tr}}_{\text{cl}}(s) \leftarrow \emptyset$
6  while $Open \neq \emptyset$ do
7      extract an apex-path pair $\mathcal{AP} = \langle \mathbf{A}, \pi\rangle$ from $Open$ with the lexicographically
         smallest $\mathbf{f}$-value
8      if IsDominated($\mathcal{AP}$) then
9          continue
10     Update($\mathbf{G}^{\text{tr}}_{\text{cl}}(s(\mathcal{AP}))$, $Tr(\mathbf{g}(\mathcal{AP}))$)
11     if $s(\mathcal{AP}) = s_{goal}$ then
12         remove the solutions weakly dominated by $\pi$ from $Sols$
13         Update($\mathbf{C}^{\text{tr}}_{\text{sol}}$, $Tr(\mathbf{c}(\pi))$)
14         add $\pi$ to $Sols$
15         continue
16     for $e \in out(s(\mathcal{AP}))$ do
17         $\mathcal{AP}' \leftarrow \langle \mathbf{A} + \mathbf{c}(e), \text{extend}(\pi, e)\rangle$
18         if IsDominated($\mathcal{AP}'$) then
19             continue
20         AddToOpen($\mathcal{AP}'$)
21 return $Sols$
22 Function IsDominated($\mathcal{AP} = \langle \mathbf{A}, \pi\rangle$):
23     if $\exists \mathbf{x} \in \mathbf{C}^{tr}_{sol} : \mathbf{x} \preceq_\varepsilon Tr(\mathbf{f}(\mathcal{AP}))$ then
24         return true
25     if $\exists \mathbf{x} \in \mathbf{G}^{tr}_{cl}(s(\mathcal{AP})) : \mathbf{x} \preceq Tr(\mathbf{g}(\mathcal{AP}))$ then
26         return true
27     return false
28 Function AddToOpen($\mathcal{AP}$):
29     for $\mathcal{AP}' \in Open[s(\mathcal{AP})]$ do
30         $\mathcal{AP}_{\text{new}} \leftarrow \text{merge}(\mathcal{AP}, \mathcal{AP}')$
31         if $\mathcal{AP}_{new}$ is $\varepsilon$-bounded then
32             remove $\mathcal{AP}'$ from $Open$
33             add $\mathcal{AP}_{\text{new}}$ to $Open$
34             return
35     add $\mathcal{AP}$ to $Open$
36     return
```

As we will show in Section 3.3, Condition 1 holds iff there exists a vector in $\mathbf{C}_{\text{sol}}^{\text{tr}}$ that ε-dominates $Tr(\mathbf{f}(\mathcal{AP}))$ (Lemma 3.3), and Condition 2 holds iff there exists a vector in $\mathbf{G}_{\text{cl}}^{\text{tr}}(s(\mathcal{AP}))$ that weakly dominates $Tr(\mathbf{g}(\mathcal{AP}))$ (Lemma 3.4). Conditions 1 and 2 are checked on Lines 23 and 25, respectively.

When A*pex expands an apex-path pair that contains state s_{goal}, it adds the representative path of this apex-path pair to $Sols$ and updates $\mathbf{C}_{\text{sol}}^{\text{tr}}$ (Lines 13-14). When A*pex expands an apex-path pair that contains a state $s \neq s_{goal}$, it generates a child apex-path pair $\mathcal{AP}$ for each out-edge e of state s by extending the expanded apex-path pair with edge e. Assume that the child apex-path pair $\mathcal{AP}$ is not discarded after the dominance checks, and let $Open[s]$ be the set of apex-path pairs in $Open$ that contain state s. A*pex checks on Lines 29-34 if there exists an apex-path pair $\mathcal{AP}'$ in $Open[s(\mathcal{AP})]$ that results in an ε-bounded apex-path pair when merged with $\mathcal{AP}$. If so, A*pex removes $\mathcal{AP}'$ from $Open$ and then adds the merged apex-path pair to $Open$ (Lines 32-33). Otherwise, it adds $\mathcal{AP}$ to $Open$ (Line 35), just like PP-A*. When $Open$ becomes empty, A*pex terminates and returns $Sols$ as an ε-approximate Pareto frontier (Line 21).

Example 6. *We use the bi-objective search problem instance in Figure 2.1 to demonstrate how A*pex works. Because this problem instance has only two objectives, we choose to use the minimum g_2-value $g_2^{min}(s)$ of all expanded nodes for each state s in the dominance checks. We use the lexicographically smallest reverse $\mathbf{g}$-value method for choosing the representative paths and assume that $\varepsilon = 0.2$.*

*Table 3.2 shows the trace of $Open$, generated apex-path pairs, and changes to g_2^{min} in each iteration of A*pex. In the table, we use $\langle s(\mathcal{AP}), \mathbf{c}(\pi), \mathbf{f}(\mathcal{AP})\rangle$ to denote an apex-path pair $\mathcal{AP} = \langle \mathbf{A}, \pi\rangle$.*

*In Iterations 2, 3, and 5, A*pex merges the generated apex-path pairs with some apex-path pairs in $Open$. For example, in Iteration 2, A*pex generates apex path pair $\mathcal{AP} = \langle s_2, (2,2), (6,6)\rangle$*

Iter	*Open* $\langle s(\mathcal{AP}), \mathbf{c}(\pi), \mathbf{f}(\mathcal{AP})\rangle$	Generated $\langle s(\mathcal{AP}), \mathbf{c}(\pi), \mathbf{f}(\mathcal{AP})\rangle$	Update of $g_2^{\min}(s(\mathcal{AP}))$
1	$\langle s_{start}, (0,0), (6,5)\rangle *$	$\langle s_1, (1,1), (6,6)\rangle$ $\langle s_2, (3,1), (7,5)\rangle$	$g_2^{\min}(s_{start}) = 0$
2	$\langle s_1, (1,1), (6,6)\rangle *$ $\langle s_2, (3,1), (7,5)\rangle$	$\langle s_2, (2,2), (6,6)\rangle$ (merged) $\langle s_3, (3,4), (6,7)\rangle$	$g_2^{\min}(s_1) = 1$
3	$\langle \mathbf{s_2}, (\mathbf{3},\mathbf{1}), (\mathbf{6},\mathbf{5})\rangle *$ $\langle s_3, (3,4), (6,7)\rangle$	$\langle s_3, (4,2), (6,5)\rangle$ (merged)	$g_2^{\min}(s_2) = 1$
4	$\langle \mathbf{s_3}, (\mathbf{4},\mathbf{2}), (\mathbf{6},\mathbf{5})\rangle *$	$\langle s_4, (5,9), (10,10)\rangle$ $\langle s_5, (5,3), (6,5)\rangle$	$g_2^{\min}(s_3) = 2$
5	$\langle s_5, (5,3), (6,5)\rangle *$ $\langle s_4, (5,9), (10,10)\rangle$	$\langle s_{goal}, (7,10), (6,10)\rangle$ $\langle s_4, (6,4), (11,5)\rangle$ (merged)	$g_2^{\min}(s_5) = 3$
6	$\langle s_{goal}, (7,10), (6,10)\rangle *$ $\langle \mathbf{s_4}, (\mathbf{6},\mathbf{4}), (\mathbf{10},\mathbf{5})\rangle$		$g_2^{\min}(s_{goal}) = 10$
7	$\langle s_4, (6,4), (10,5)\rangle *$	$\langle s_{goal}, (12,5), (10,5)\rangle$	$g_2^{\min}(s_4) = 4$
8	$\langle s_{goal}, (12,5), (10,5)\rangle *$		$g_2^{\min}(s_{goal}) = 5$
9	empty		

Table 3.2: Trace of $Open$, generated apex-path pairs, and $g_2^{\min}$ in each iteration of A*pex when solving the example problem instance in Figure 2.1. "$*$" marks the apex-path pair that is extracted in that iteration. Boldface font marks the apex-path pairs that result from merging two apex-path pairs in the previous iteration.

and merges it with apex-path pair $\mathcal{AP}' = \langle s_2, (3,1), (7,5)\rangle$ *in* $Open$*. As shown in Figure 2.1, the heuristic for state* s_2 *is* $(4,4)$*. The* $\mathbf{g}$*-values for* $\mathcal{AP}$ *and* $\mathcal{AP}'$ *are* $(2,2)$ *and* $(3,1)$*, respectively. The new apex after merging is* $(2,1)$*, and hence the new* $\mathbf{f}$*-value is* $(6,5)$*. A*pex chooses the path with cost* $(3,1)$ *as the new representative path. The resulting apex path pair is* 0.2*-bounded because* $(3,1) + (4,4) \preceq_{0.2} (6,5)$.

*In Iterations 6 and 8, A*pex expands apex-path pairs that contain state* s_{goal} *and adds solutions with costs* $(7,10)$ *and* $(12,5)$ *to* $Sols$*, respectively. The set of solutions that A*pex returns is the* 0.2*-approximate Pareto frontier in Figure 3.2b. It takes A*pex 9 iterations to solve this problem instance, which is much fewer than the 18 iterations that it takes of BOA*, as previously shown in Example 3.*

3.3 Correctness and Completeness

In this section, we prove the completeness and correctness of A*pex. Theorem 3.1 shows that A*pex terminates in finite time and computes an ε-approximate Pareto frontier.

Lemma 3.1. *A*pex adds only ε-bounded apex-path pairs to $Open$.*

Proof. We prove this lemma by induction on iterations. On Line 1, A*pex adds apex-path pair $\langle \mathbf{0}, [s_{\text{start}}]\rangle$ to $Open$. Apex-path pair $\langle \mathbf{0}, [s_{\text{start}}]\rangle$ is ε-bounded because the $\mathbf{f}$-value of its representative path is equal to (and hence weakly dominates) its $\mathbf{f}$-value. In each iteration, A*pex extracts an apex-path pair from $Open$. Assume that this apex-path pair is ε-bounded. There are two cases where A*pex adds apex-path pairs to $Open$:

1. On Line 33, A*pex adds a merged apex-path pair $\mathcal{AP}_{\text{new}}$ to $Open$. Apex-path pair $\mathcal{AP}_{\text{new}}$ is ε-bounded because of the condition on Line 31.

2. On Line 35, A*pex adds an apex-path pair $\mathcal{AP}$ to $Open$. $\mathcal{AP}$ extends the extracted apex-path pair and is ε-bounded because of Property 3.2.

By induction, A*pex adds only ε-bounded apex-path pairs to $Open$. □

Lemma 3.2. *The sequence of extracted apex-path pairs has monotonically non-decreasing f_1-values.*

Proof. Consider an apex-path pair $\mathcal{AP}$ extracted by A*pex from $Open$. $\mathcal{AP}$ has the smallest f_1-value of all apex-path pairs in $Open$ because its $\mathbf{f}$-value is the lexicographically smallest one. When A*pex expands $\mathcal{AP}$, the f_1-value of any generated apex-path pair is no smaller than that of $\mathcal{AP}$ because the heuristic is consistent. When A*pex adds such a generated apex-path pair to

$Open$, it might merge this apex-path pair with another apex-path pair in $Open$, whose f_1-value is also no smaller than $f_1(\mathcal{AP})$. The apex-path pair resulting from the merge operation cannot have an f_1-value smaller than $f_1(\mathcal{AP})$ as well. Thus, the f_1-values of all apex-path pairs that are added to $Open$ when expanding $\mathcal{AP}$ are no smaller than $f_1(\mathcal{AP})$. Therefore, the f_1-value of the apex-path pair extracted in the next iteration are no smaller than $f_1(\mathcal{AP})$. □

Lemma 3.3. *There exists a truncated cost in $\mathbf{C}^{tr}_{sol}$ that ε-dominates the truncated $\mathbf{f}$-value of apex-path pair $\mathcal{AP}$ on Line 23 iff there exists a solution in $Sols$ whose cost ε-dominates the $\mathbf{f}$-value of apex-path pair $\mathcal{AP}$.*

Proof. Assume that there exists a solution π_{sol} in $Sols$ whose cost ε-dominates the $\mathbf{f}$-value of apex-path pair $\mathcal{AP}$. Because A*pex has added $Tr(\mathbf{c}(\pi_{\text{sol}}))$ to $\mathbf{C}^{\text{tr}}_{\text{sol}}$ on Line 13, there must exist some truncated cost $\mathbf{x}$ in $\mathbf{C}^{\text{tr}}_{\text{sol}}$ that weakly dominates $Tr(\mathbf{c}(\pi_{\text{sol}}))$, which in turn ε-dominates the truncated $\mathbf{f}$-value of apex-path pair $\mathcal{AP}$. Therefore, $\mathbf{x}$ ε-dominates the truncated $\mathbf{f}$-value of apex-path pair $\mathcal{AP}$.

Assume that there exists a truncated cost $\mathbf{x}$ in $\mathbf{C}^{\text{tr}}_{\text{sol}}$ that ε-dominates the truncated $\mathbf{f}$-value of apex-path pair $\mathcal{AP}$. Let $\mathcal{AP}_{\text{sol}} = \langle \mathbf{A}_{\text{sol}}, \pi_{\text{sol}} \rangle$ denote the apex-path pair with which A*pex reached Line 13 and added $\mathbf{x}$ to $\mathbf{C}^{\text{tr}}_{\text{sol}}$. From Line 13, we have $Tr(\mathbf{c}(\pi_{\text{sol}})) = \mathbf{x}$ and hence

$$Tr(\mathbf{c}(\pi_{\text{sol}})) \preceq_{\varepsilon} Tr(\mathbf{f}(\mathcal{AP})). \tag{3.4}$$

According to Lemma 3.1, $\mathcal{AP}_{\text{sol}}$ is ε-bounded. We have $c_1(\pi_{\text{sol}}) = f_1(\pi_{\text{sol}}) \leq (1+\varepsilon)f_1(\mathcal{AP}_{\text{sol}})$. According to Lemma 3.2 and also because $\mathcal{AP}_{\text{sol}}$ is extracted before $\mathcal{AP}$, $f_1(\mathcal{AP}_{\text{sol}}) \leq f_1(\mathcal{AP})$. Therefore, we have $c_1(\pi_{\text{sol}}) \leq (1+\varepsilon)f_1(\mathcal{AP})$. By combining this property and Eq. 3.4, we have

$\mathbf{c}(\pi_{\text{sol}}) \preceq_\varepsilon \mathbf{f}(\mathcal{AP})$. Therefore, there exists a solution π_{sol} in $Sols$ whose cost ε-dominates the $\mathbf{f}$-value of apex-path pair $\mathcal{AP}$. □

Lemma 3.4. *There exists a truncated* $\mathbf{g}$*-value in* $\mathbf{G}^{tr}_{cl}(s(\mathcal{AP}))$ *that weakly dominates the truncated* $\mathbf{g}$*-value of apex-path pair* $\mathcal{AP}$ *on Line 25 iff there exists an expanded apex-path pair* $\mathcal{AP}'$ *that contains state* $s(\mathcal{AP})$ *and whose* $\mathbf{g}$*-value weakly dominates the* $\mathbf{g}$*-value of apex-path pair* $\mathcal{AP}$*.*

Proof. Assume that there exists an expanded apex-path pair $\mathcal{AP}'$ that contains state $s(\mathcal{AP})$ and whose $\mathbf{g}$-value weakly dominates the one of apex-path pair $\mathcal{AP}$ (that is, $\mathbf{g}(\mathcal{AP}') \preceq \mathbf{g}(\mathcal{AP})$). Because A*pex has added $Tr(\mathbf{g}(\mathcal{AP}')$ to $\mathbf{G}^{\text{tr}}_{\text{cl}}(s(\mathcal{AP}))$ on Line 10, there must exist some vector $\mathbf{x}$ in $\mathbf{G}^{\text{tr}}_{\text{cl}}(s(\mathcal{AP}))$ that weakly dominates $Tr(\mathbf{g}(\mathcal{AP}))$.

Assume that there exists a truncated $\mathbf{g}$-value in $\mathbf{G}^{\text{tr}}_{\text{cl}}(s(\mathcal{AP}))$ that weakly dominates the truncated $\mathbf{g}$-value of apex-path pair $\mathcal{AP}$. Let apex-path pair $\mathcal{AP}'$ be the expanded apex-path pair that contains state $s(\mathcal{AP})$ and with which Line 10 was executed to add this truncated $\mathbf{g}$-value to $\mathbf{G}^{\text{tr}}_{\text{cl}}(s(\mathcal{AP}))$. Hence, we have $Tr(\mathbf{g}(\mathcal{AP}')) \preceq Tr(\mathbf{g}(\mathcal{AP}))$. It holds that $f_1(\mathcal{AP}') \leq f_1(\mathcal{AP})$ according to Lemma 3.2. Because $s(\mathcal{AP}) = s(\mathcal{AP}')$, we also have $g_1(\mathcal{AP}') \leq g_1(\mathcal{AP})$. Thus, the $\mathbf{g}$-value of apex-path pair $\mathcal{AP}'$ weakly dominates the one of apex-path pair $\mathcal{AP}$. □

Lemma 3.5. *If the apex of an apex-path pair weakly dominates a vector and the apex-path pair is merged with another apex-path pair, then the apex of the merged apex-path pair weakly dominates the vector as well.*

Proof. The apex of the merged apex-path pair is the component-wise minimum of the apexes of the two merged apex-path pairs. □

For the rest of this section, we introduce the following notation for ease of presentation: Given a solution π_{sol} that traverses the sequence of states $[s_1(= s_{start}), s_2 \dots s_L(= s_{goal})]$, we use $\pi_{\text{sol}}^{(l)}$, $l = 1, 2 \dots L$, to denote its prefix that traverse the first l states $[s_1, s_2 \dots s_l]$ of π_{sol}.

Lemma 3.6. *Consider any solution π_{sol} and let $[s_1, s_2 \dots s_L]$ denote the sequence of states it traverses. At the beginning of each iteration of A*pex (that is, before executing Line 7), if A*pex has expanded (that is, reached Line 11 with) an apex-path pair $\mathcal{AP}$ with $\mathbf{g}(\mathcal{AP}) \preceq \mathbf{c}(\pi_{sol}^{(j)})$ and $s(\mathcal{AP}) = s_j$ for some j, then there exists (1) an apex-path pair $\mathcal{AP}'$ in $Open$ with $\mathbf{g}(\mathcal{AP}') \preceq \mathbf{c}(\pi_{sol}^{(k)})$ and $s(\mathcal{AP}') = s_k$ for some $k > j$ or (2) a solution in $Sols$ that ε-dominates π_{sol}.*

Proof. We prove this lemma by induction on j, starting with $j = L$ and going backward. If A*pex has expanded an apex-path pair $\mathcal{AP}$ with $\mathbf{g}(\mathcal{AP}) \preceq \mathbf{c}(\pi_{\text{sol}}^{(L)})$ and $s(\mathcal{AP}) = s_L(= s_{goal})$, A*pex has added the representative path of $\mathcal{AP}$ to $Sols$ on Line 14. Because the heuristic is consistent, we have $\mathbf{h}(s(\mathcal{AP})) = \mathbf{h}(s_{goal}) = \mathbf{0}$. Because $\mathcal{AP}$ is ε-bounded (according to Lemma 3.1), the cost of its representative path ε-dominates its $\mathbf{f}$-value $\mathbf{f}(\mathcal{AP})$ (which is equal to $\mathbf{g}(\mathcal{AP})$ because $\mathbf{h}(s(\mathcal{AP})) = \mathbf{0}$) and hence ε-dominates $\mathbf{c}(\pi_{\text{sol}}^{(L)})$.[4] Because A*pex removes a solution from $Sols$ only if it is weakly dominated by a new solution (Line 12), there must exist some solution in $Sols$ that ε-dominates π_{sol} when A*pex reaches the beginning of all following iterations. Therefore, the lemma holds for $j = L$.

Assume that the lemma holds for $j = l + 1, l \leq L - 1$, and A*pex has expanded an apex-path pair $\mathcal{AP} = \langle \mathbf{A}, \pi \rangle$ with $\mathbf{g}(\mathcal{AP}) \preceq \mathbf{c}(\pi_{\text{sol}}^{(l)})$ and $s(\mathcal{AP}) = s_l$. Consider the iteration in which $\mathcal{AP}$ was expanded and the child apex-path pair $\mathcal{AP}' = \langle \mathbf{A}', \pi' \rangle$ of $\mathcal{AP}$ created on Line 17 for the lth

[4]Consider any ε-value and vectors $\mathbf{x}$, $\mathbf{y}$, and $\mathbf{z}$. If $\mathbf{x} \preceq_\varepsilon \mathbf{y}$ and $\mathbf{y} \preceq \mathbf{z}$, then $\mathbf{x} \preceq_\varepsilon \mathbf{z}$.

edge e in π_{sol}, which is from state s_l to states_{l+1}. Apex-path pair $\mathcal{AP}'$ contains state s_{l+1}, and its apex weakly dominates the cost of path $\pi_{\text{sol}}^{(l+1)}$, because

$$\begin{aligned}\mathbf{A}' =& \mathbf{A} + \mathbf{c}(e)\\ =& \mathbf{g}(\mathcal{AP}) + \mathbf{c}(e)\\ \preceq& \mathbf{c}(\pi_{\text{sol}}^{(l)}) + \mathbf{c}(e)\\ =& \mathbf{c}(\pi_{\text{sol}}^{(l+1)}).\end{aligned} \tag{3.5}$$

Because the heuristic is consistent, we have $\mathbf{c}(\pi_{\text{sol}}^{(l+1)}) + \mathbf{h}(s_{l+1}) \prec \mathbf{c}(\pi_{\text{sol}})$. Combining this and Eq. 3.5, we have

$$\begin{aligned}\mathbf{f}(\mathcal{AP}') =& \mathbf{g}(\mathcal{AP}') + \mathbf{h}(s(\mathcal{AP}'))\\ =& \mathbf{A}' + \mathbf{h}(s(\mathcal{AP}'))\\ \preceq& \mathbf{c}(\pi_{\text{sol}}^{(l+1)}) + \mathbf{h}(s(\mathcal{AP}'))\\ =& \mathbf{c}(\pi_{\text{sol}}^{(l+1)}) + \mathbf{h}(s_{l+1})\\ \preceq& \mathbf{c}(\pi_{\text{sol}}).\end{aligned} \tag{3.6}$$

We distinguish the following cases:

1. Apex-path pair $\mathcal{AP}'$ was pruned because the condition on Line 23 held. According to Lemma 3.3, there existed some solution in $Sols$ whose cost ε-dominated the $\mathbf{f}$-value of $\mathcal{AP}'$, which in turn weakly dominated the cost of π_{sol} (because of Eq. 3.6). Because A*pex removes a solution from $Sols$ only if it is weakly dominated by a new solution (Line 12),

there must exist some solution in $Sols$ that ε-dominates solution π_{sol} when A*pex reaches the beginning of all following iterations. Thus, the lemma holds for $j = l$.

2. Apex-path pair $\mathcal{AP}'$ was pruned because the condition on Line 25 held, namely, there existed a truncated $\mathbf{g}$-value in $G^T_{\text{cl}}(s(\mathcal{AP}'))$ that weakly dominated the truncated $\mathbf{g}$-value of $\mathcal{AP}'$. Then, according to Lemma 3.4, at the time $\mathcal{AP}'$ was pruned, A*pex had expanded an apex-path pair $\mathcal{AP}''$ that contains state s_{l+1} and whose apex weakly dominates the apex of $\mathcal{AP}'$ (and hence the cost of path $\pi^{(l+1)}_{\text{sol}}$). Because we assume that the lemma holds for $j = l + 1$, when A*pex reaches the beginning of all following iterations, there exists (1) an apex-path pair $\mathcal{AP}'''$ in $Open$ with $\mathbf{g}(\mathcal{AP}''') \preceq \mathbf{c}(\pi^{(k)}_{\text{sol}})$ and $s(\mathcal{AP}''') = s_k$ for some $k > l+1$ (and hence also $k > l$) or (2) a solution in $Sols$ that ε-dominates π_{sol}. Therefore, the lemma holds for $j = l$.

3. Otherwise, A*pex executed Line 20 to add apex-path pair $\mathcal{AP}'$ to $Open$, perhaps after merging it with another apex-path pair on Line 30. A*pex might merge it several (more) times with other apex-path pairs on Line 30. The apex of the resulting apex-path pair $\mathcal{AP}''$ weakly dominates $\mathbf{c}(\pi^{(l+1)}_{\text{sol}})$ because of Eq. 3.5 and Lemma 3.5. When A*pex reaches the beginning of an iteration again, we further distinguish the following subcases:

    (a) Apex-path pair $\mathcal{AP}''$ is still in $Open$. The lemma holds.

    (b) Apex-path pair $\mathcal{AP}''$ has been extracted from $Open$ and pruned because of the conditions on Lines 23 or 25 in a previous iteration. Like $\mathcal{AP}'$, $\mathcal{AP}''$ contains state s_{l+1} and its apex weakly dominates the cost of path $\pi^{(l+1)}_{\text{sol}}$. We can apply the same arguments in cases 1 and 2 for $\mathcal{AP}'$ to $\mathcal{AP}''$ and show that the lemma holds.

(c) Apex-path pair $\mathcal{AP}''$ has been extracted from $Open$ and expanded in a previous iteration. Because we assume that the lemma holds for $j = l+1$, when A*pex reaches the beginning of all following iterations, there exists (1) an apex-path pair $\mathcal{AP}'''$ in $Open$ with $\mathbf{g}(\mathcal{AP}''') \preceq \mathbf{c}(\pi_{\text{sol}}^{(k)})$ and $s(\mathcal{AP}''') = s_k$ for some $k > l+1$ (and hence also $k > l$) or (2) a solution in $Sols$ that ε-dominates π_{sol}. Therefore, the lemma holds for $j = l$.

Therefore, by induction, the lemma holds for all $j = 1, 2 \ldots L$. □

Lemma 3.7. *Consider any solution π_{sol} and let $[s_1, s_2 \ldots s_L]$ denote the sequence of states it traverses. At the beginning of each iteration of A*pex, there always exists (1) an apex-path pair $\mathcal{AP}$ in $Open$ that satisfies $\mathbf{g}(\mathcal{AP}) \preceq \mathbf{c}(\pi_{sol}^{(j)})$ and $s(\mathcal{AP}) = s_j$ for some j or (2) a solution in $Sols$ that ε-dominates π_{sol}.*

Proof. Consider the first apex-path pair that A*pex adds to $Open$, that is, apex-path pair $\mathcal{AP}_0 = \langle \mathbf{0}, [s_{\text{start}}] \rangle$. At the beginning of the first iteration of A*pex, the Lemma holds because $\mathcal{AP}_0$ satisfies $\mathbf{g}(\mathcal{AP}_0) \preceq \mathbf{c}(\pi_{\text{sol}}^{(1)})$ (because both $\mathbf{g}(\mathcal{AP}_0)$ and $\mathbf{c}(\pi_{\text{sol}}^{(1)})$ are $\mathbf{0}$) and $s(\mathcal{AP}_0) = s_1$. A*pex then expands $\mathcal{AP}_0$. The lemma holds for all future iterations according to Lemma 3.6. □

Lemma 3.8. *A*pex does not expand an apex-path pair $\mathcal{AP}$ if there exists a solution π_{sol} with $\mathbf{c}(\pi_{sol}) \prec \mathbf{f}(\mathcal{AP})$.*

Proof. Consider the case where A*pex extracts an apex-path pair $\mathcal{AP}$ from $Open$ on Line 7 and there exists a solution π_{sol} with $\mathbf{c}(\pi_{\text{sol}}) \prec \mathbf{f}(\mathcal{AP})$. We prove this lemma by showing that A*pex will not expand $\mathcal{AP}$. According to Lemma 3.7, we distinguish two cases:

1. There exists an apex-path pair $\mathcal{AP}'$ in $Open$ that satisfies $\mathbf{g}(\mathcal{AP}') \preceq \mathbf{c}(\pi_{\text{sol}}^{(j)})$ and $s(\mathcal{AP}') = s_j$ for some j. Because the heuristic is consistent, we have $\mathbf{c}(\pi_{\text{sol}}^{(j)}) + \mathbf{h}(s_j) \prec \mathbf{c}(\pi_{\text{sol}})$.

We have $\mathbf{f}(\mathcal{AP}') = \mathbf{g}(\mathcal{AP}') + \mathbf{h}(s(\mathcal{AP}')) \preceq \mathbf{c}(\pi_{\text{sol}}^{(j)}) + \mathbf{h}(s_j) \preceq \mathbf{c}(\pi_{\text{sol}})$. Hence, we have $\mathbf{f}(\mathcal{AP}') \preceq \mathbf{c}(\pi_{\text{sol}}) \prec \mathbf{f}(\mathcal{AP})$. A*pex would not extract $\mathcal{AP}$ from $Open$ while $\mathcal{AP}'$ is in $Open$ because A*pex extracts the apex-path pair with the lexicographically smallest $\mathbf{f}$-value. Therefore, this case cannot happen.

2. There exists a solution π'_{sol} in $Sols$ that ε-dominates π_{sol}. Then, $\mathbf{c}(\pi'_{\text{sol}})$ ε-dominates $\mathbf{c}(\pi_{\text{sol}})$, which in turn dominates $\mathbf{f}(\mathcal{AP})$. According to Lemma 3.3, there exists a truncated cost in $\mathbf{C}^{\text{tr}}_{\text{sol}}$ that ε-dominates the truncated $\mathbf{f}$-value of $\mathcal{AP}$. Therefore, A*pex will prune $\mathcal{AP}$ on Line 9 because of the condition on Line 23.

□

Theorem 3.1. *A*pex terminates in finite time. For any solution π_{sol}, there exists, when A*pex terminates, a solution in $Sols$ that ε-dominates solution π_{sol}.*

Proof. Consider any solution π_{sol} and any expanded apex-path pair $\mathcal{AP} = \langle \mathbf{A}, \pi \rangle$. Because A*pex generates only ε-bounded apex-path pairs (according to Lemma 3.1), $\mathbf{f}(\pi)$ must ε-dominate $\mathbf{f}(\mathcal{AP})$ (that is, $c_i(\pi)+h_i(s(\mathcal{AP})) \leq (1+\varepsilon)f_i(\mathcal{AP})$ for all $i = 1, 2 \ldots N$). According to Lemma 3.8, $\mathbf{f}(\mathcal{AP})$ is not dominated by $\mathbf{c}(\pi_{\text{sol}})$. Hence, $f_i(\mathcal{AP}) \leq c_i(\pi_{\text{sol}})$ must hold for some i. Therefore, $c_i(\pi) + h_i(s(\mathcal{AP})) \leq (1 + \varepsilon)c_i(\pi_{\text{sol}})$ must hold for some i. Because heuristic $\mathbf{h}$ is non-negative, $c_i(\pi) \leq (1 + \varepsilon)c_i(\pi_{\text{sol}})$ must hold for some i for the representative path π of any expanded apex-path pair. Because the graph is finite and has positive edge costs, one can extend a path only a finite number of times before the resulting path π' does not satisfy $c_i(\pi') \leq (1+\varepsilon)c_i(\pi_{\text{sol}})$ for all i. Thus, only a finite number of representative paths can exist for the apex-path pairs expanded by A*pex. Because A*pex never creates duplicate representative paths, there are only a finite number of apex-path pairs that A*pex can expand (and generate). A*pex terminates in finite time.

Consider the beginning of the last iteration of A*pex before it terminates, where $Open$ becomes empty. According to Lemma 3.7, for any solution π_{sol}, there exists a solution in $Sols$ that ε-dominates π_{sol}. □

3.4 Experimental Evaluation

In our experimental evaluation, we compare A*pex with BOA*, BOA*-ε, and PP-A* on problem instances with two objectives (Section 3.4.1) and A*pex with LTMOA* and LTMOA*-ε on problem instances with more than two objectives (Section 3.4.2). We implement all algorithms in C++[5] and run all experiments on a MacBook with an M1 Pro chip and 32GB of memory. We use the perfect-distance heuristic for all problem instances and algorithms. The runtime limit for solving each problem instance is five minutes.

We use different variants of A*pex with different approaches for choosing the new representative paths when merging apex-path pairs:

1. A*pex-random uses the Random method.
2. A*pex-rlex uses the Lexicographically smallest reverse $\mathbf{g}$-value method.
3. A*pex-greedy uses the Greedy method.

For each problem instance and each algorithm, we evaluate the approximation factors $\varepsilon = 0.001$, 0.01, 0.1, and 0.2.

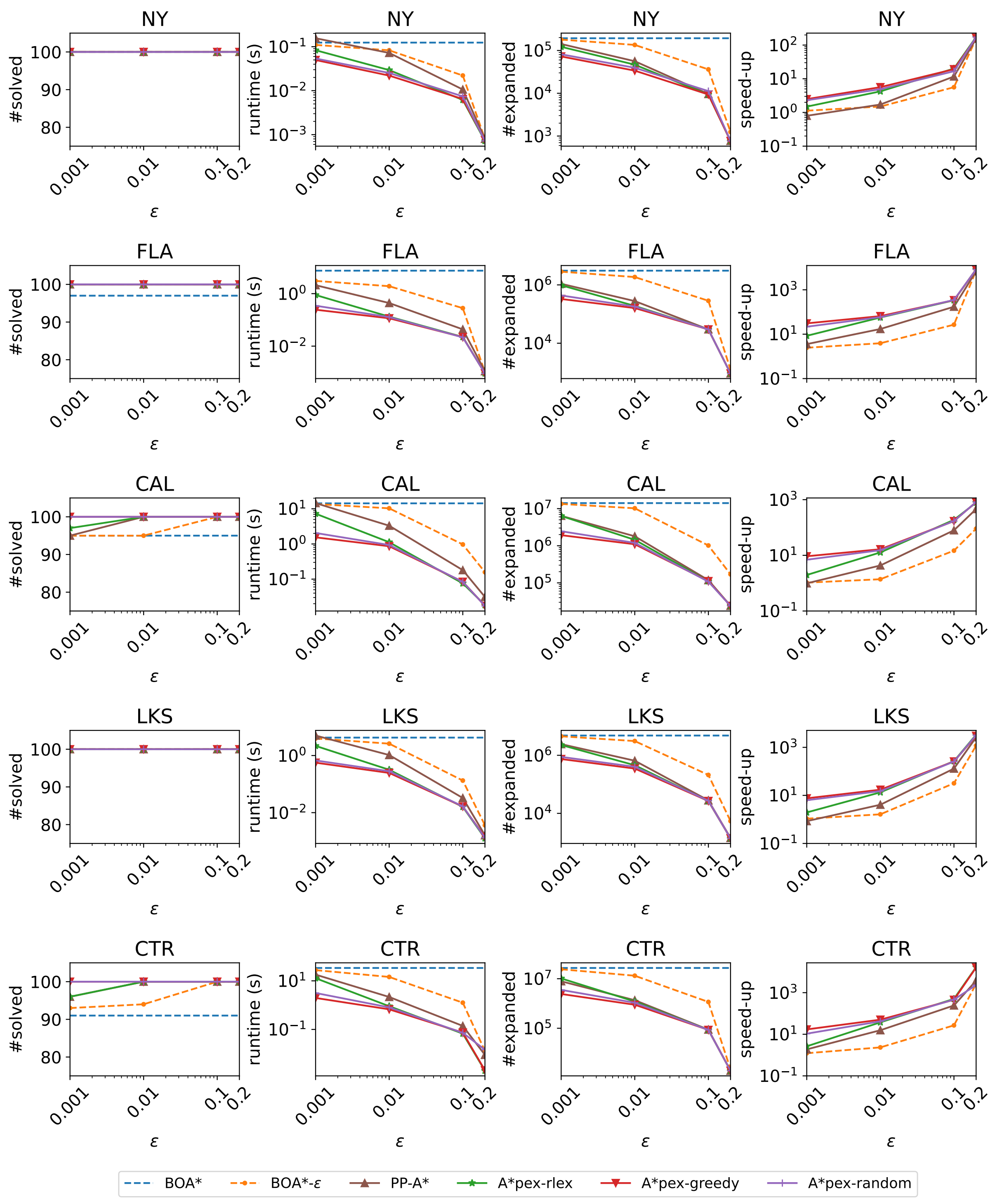


Figure 3.8: Results for BOA*, BOA*-ε, PP-A*, and different variants of A*pex on bi-objective problem instances with different approximation factors ε.

3.4.1 Problem Instances with Two Objectives

In this section, we compare different variants of A*pex with BOA*, BOA*-ε, and PP-A* on problem instances with two objectives. We use five road networks from the 9th DIMACS Implementation Challenge,[6] namely, the NY road network (264K states and 730K edges), the FLA road network (1.1M states and 2.7M edges), the CAL road network (1.9M states and 4.7M edges), the LKS road network (2.8M states and 6.9M edges), and the CTR road network (14.1M states and 34.3M edges). We use the two objectives that are available in the benchmark, namely travel time and travel distance. We use the 100 problem instances used by Ahmadi et al. [2] for each road network.

Figure 3.8 shows the numbers of solved problem instances, average runtimes (in seconds), and average numbers of node expansions of all algorithms and speed-ups of BOA*-ε, PP-A*, and all variants of A*pex over BOA* with respect to their average runtimes. All averages are calculated over the problem instances solved by all algorithms within the runtime limit for all approximation factors. All variants of A*pex have larger numbers of solved problem instances, smaller average numbers of node expansions, and smaller average runtimes than BOA*, BOA*-ε, and PP-A* in most cases. Among the different variants of A*pex, A*pex-greedy slightly outperforms the other two variants. The numbers of solved instances of A*pex-greedy are at least as high as the ones of A*pex-rlex and A*pex-random in most cases.

Figure 3.9 shows comparisons of the individual runtimes (in seconds) between A*pex-greedy and BOA*, BOA*-ε, and PP-A* on all problem instances. The x- and y-coordinates of each point show the runtimes of the baseline algorithm and A*pex-greedy, respectively, for a problem instance. Different markers represent different approximation factors ε. A*pex-greedy outperforms

[5] https://github.com/HanZhang39/MultiObjectiveSearch
[6] http://www.diag.uniroma1.it/challenge9/download.shtml

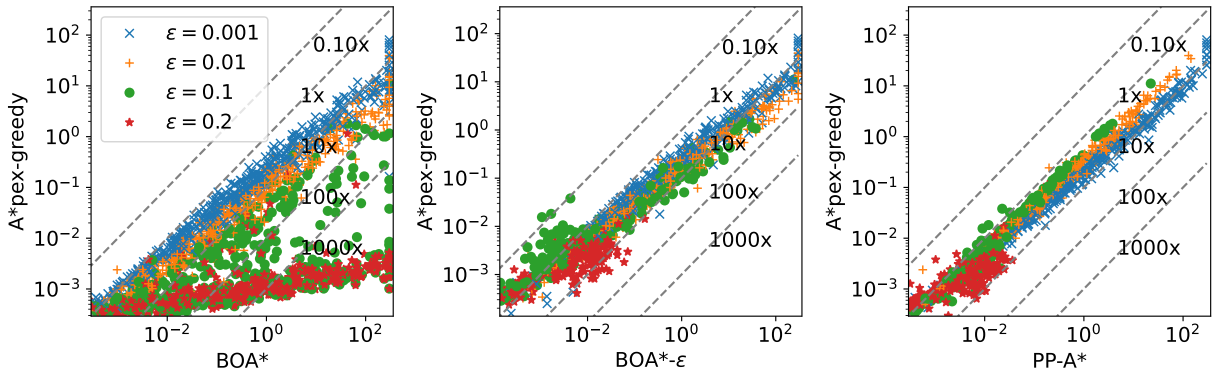


Figure 3.9: Runtime comparisons between A*pex-greedy and different algorithms on bi-objective road-network problem instances with different approximation factors.

all three baseline algorithms except on some easy problem instances where BOA*-ε and PP-A* are faster than A*pex-greedy. As the approximation factor increases, the speed-ups of A*pex-greedy over BOA* become more substantial. For all three baseline algorithms, the speed-ups of A*pex-greedy are more substantial on hard problem instances (represented by the top-right markers).

3.4.2 Problem Instances with More than Two Objectives

In this section, we compare different variants of A*pex with LTMOA* and LTMOA*-ε on problem instances with more than two objectives. We use the NY road network from the 9th DIMACS Implementation Challenge. In addition to travel time (t) and travel distance (d), we use the economic cost (m) [54], the number of edges (l) [47], and a random integer from 1 to 100 (r) [36] as the third, fourth, and fifth objectives, respectively. We use the same 100 problem instances used by Sedeño-Noda and Colebrook [65] and Ahmadi et al. [2].

Figure 3.10 shows the numbers of solved problem instances, average runtimes (in seconds), and average numbers of node expansions of all algorithms and the speed-ups of LTMOA*-ε and

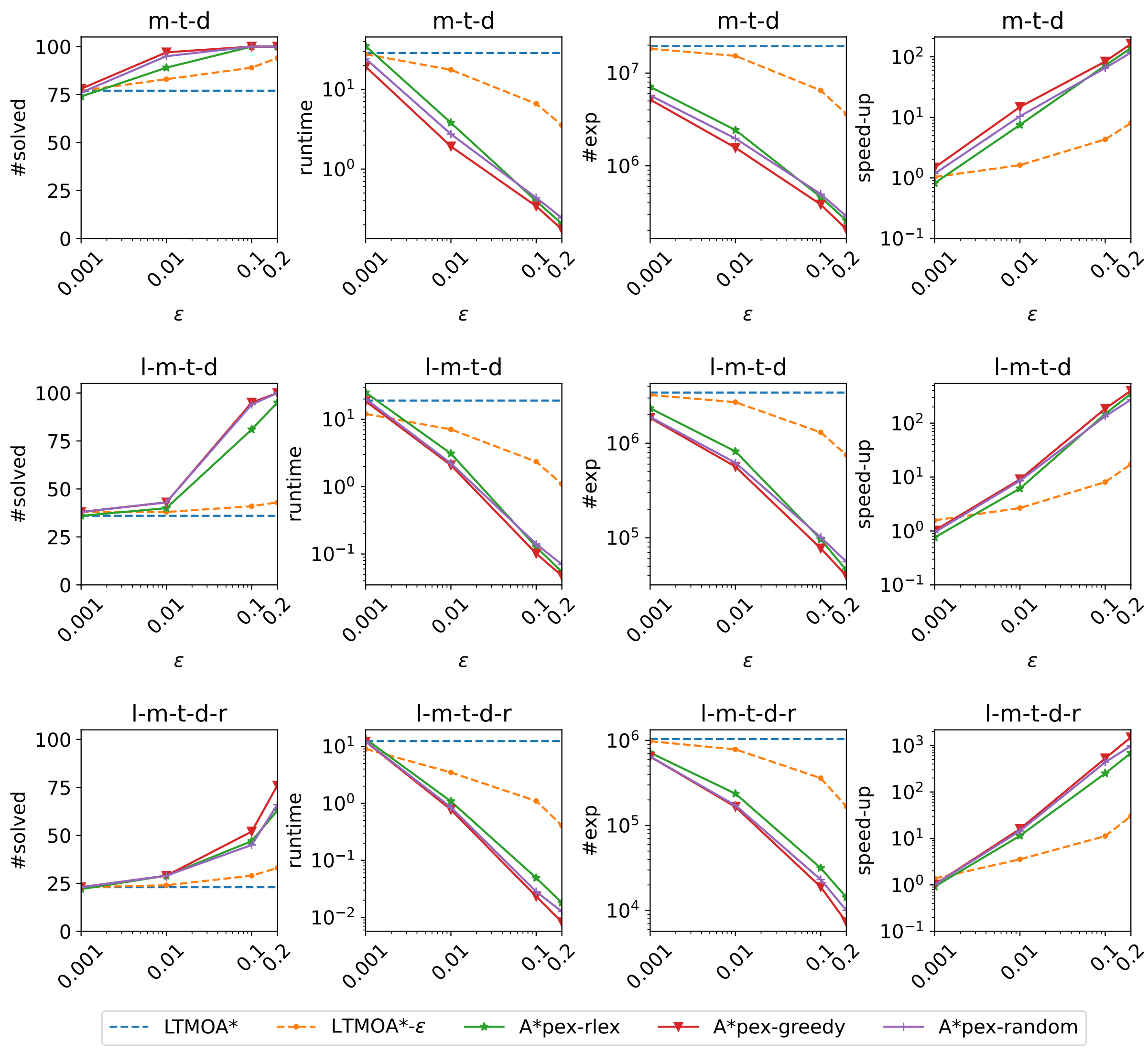


Figure 3.10: Results for LTMOA*, LTMOA*-ε, and different variants of A*pex on road-network problem instances with different objectives and approximation factors.

all variants of A*pex over LTMOA* with respect to their average runtimes. All averages are calculated over problem instances solved by all algorithms within the runtime limit for all approximation factors. Consider different combinations of numbers of objectives and approximation factors. All variants of A*pex have smaller average numbers of node expansions than LTMOA* and LTMOA*-ε for all combinations. However, the average runtimes of A*pex are larger than the ones of LTMOA* and LTMOA*-ε for $\varepsilon = 0.001$, which is due to the runtime overhead of

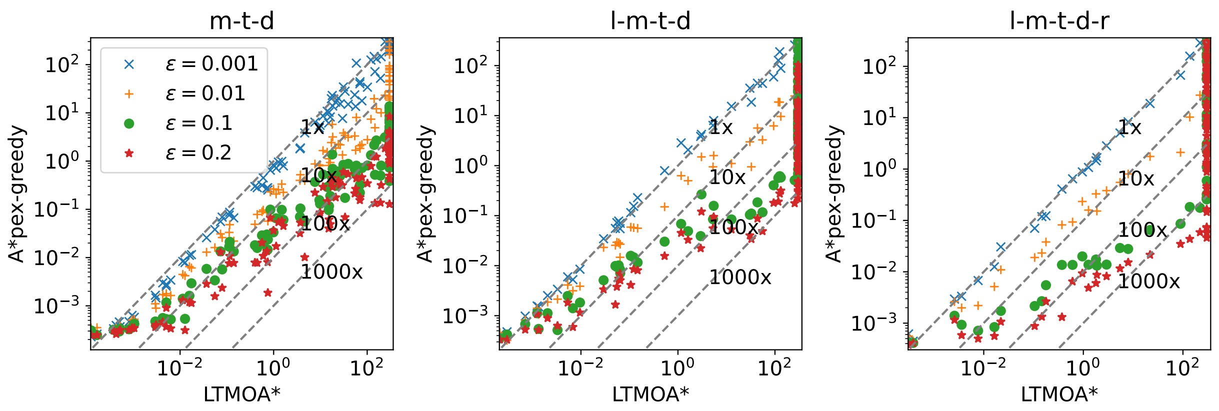


Figure 3.11: Runtimes of LTMOA* and A*pex-greedy on road-network problem instances with different objectives and approximation factors.

A*pex from iterating over *Open* and merging apex-path pairs. For all other combinations, the average runtimes of all variants of A*pex are smaller than the ones of LTMOA* and LTMOA*-ε. The speed-ups of LTMOA*-ε and different variants of A*pex over LTMOA* increase as the approximation factor ε increases. However, the speed-ups of LTMOA*-ε are much less substantial than the speed-ups of the different variants of A*pex for larger approximation factors. Similar to the results for the bi-objective problem instances, the results for the problem instances with more than two objectives show that A*pex-greedy slightly outperforms the other two A*pex variants. The numbers of solved instances of A*pex-greedy are at least as high as the ones of A*pex-rlex and A*pex-random.

Figure 3.11 shows the runtime comparisons between LTMOA* and A*pex-greedy for different numbers of objectives, and Figure 3.12 shows the runtime comparisons between LTMOA*-ε and A*pex-greedy for different numbers of objectives. On problem instances with approximation factor $\varepsilon = 0.001$, A*pex-greedy has similar runtimes as LTMOA* and LTMOA*-ε. On most problem instances with larger approximation factors, A*pex-greedy outperforms LTMOA* and LTMOA*-ε. The speed-up of A*pex-greedy over LTMOA* increases as the approximation factor increases.

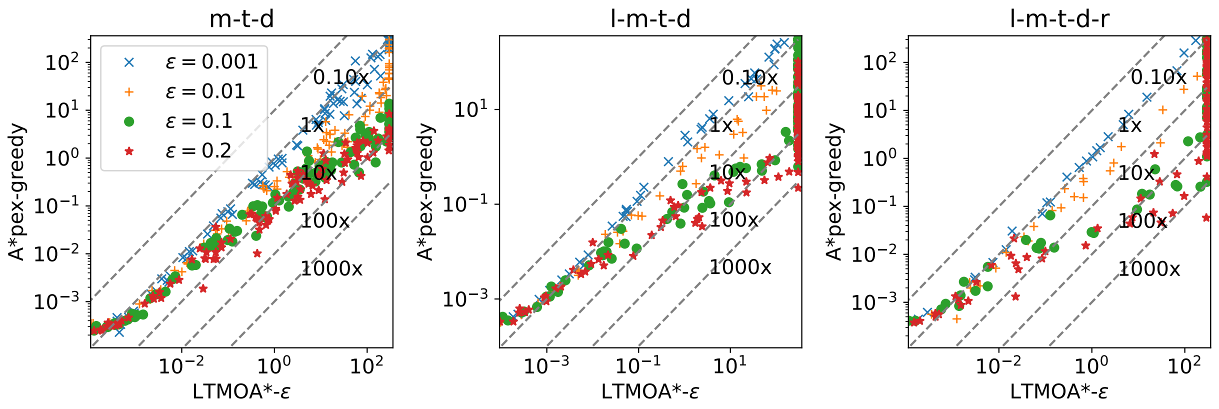


Figure 3.12: Runtimes of LTMOA*-ε and A*pex-greedy on road-network problem instances with different objectives and approximation factors.

For an approximation factor of 0.2, the speed-ups of A*pex-greedy over LTMOA* can be more than $1000\times$.

3.5 Summary

In this chapter, we introduced A*pex, an approximate multi-objective search algorithm that finds an ε-approximate Pareto frontier for a given approximation factor ε. It builds upon PP-A* but (1) makes PP-A* more efficient for bi-objective search and (2) generalizes it from two objectives to any number of objectives. We analyzed the correctness and completeness of A*pex and experimentally demonstrated its efficiency advantage over state-of-the-art multi-objective search algorithms. Our results validate the hypothesis that one can find an approximate Pareto frontier much faster than the Pareto frontier, and the runtime decreases as the given approximation factor increases.

3.6 Extensions

This chapter showed that, in multi-objective search, apex-path pairs can represent sets of paths while bounding the approximation factor of the resulting solution sets. We might be able to apply similar techniques to different problems and develop efficient approximate/suboptimal algorithms for them. This section gives a brief overview of our works on two such problems, namely, the Weight-Constrained Shortest Path (WCSP) problem (Section 3.6.1) and the Multi-Objective Multi-Agent Path Finding (MO-MAPF) problem (Section 3.6.2).

3.6.1 The Weight-Constrained Shortest Path Problem

Given a bi-objective search graph, a start state, a goal state, and a weight limit W, the Weight-Constrained Shortest Path (WCSP) problem is the problem of computing a path from the start state to the goal state that minimizes the c_1-value under the constraint that the c_2-value is no larger than W. The WCSP problem is important for many applications. In an electric vehicle domain, for example, the graph represents a road network, and the c_1- and c_2-values correspond to the driving time and the battery consumption, respectively [7]. Here, a desired route minimizes the driving time without depleting the battery. The WCSP problem is also important in the contexts of column generation [89] and vehicle routing [4]. The WCSP problem is NP-hard to solve optimally [32, 44].

A *WCSP problem instance* is specified by a tuple $P = \langle G, s_{\text{start}}, s_{\text{goal}}, W \rangle$, where G is a graph with two costs, s_{start} is the start state, s_{goal} is the goal state, and $W \in \mathbb{R}_{>0}$ is the *weight limit.* A *(WCSP) solution* π is a path from s_{start} to s_{goal} with $c_2(\pi) \leq W$. An *optimal solution* is a solution with the minimum c_1-value, denoted as c_1^*, of all solutions.

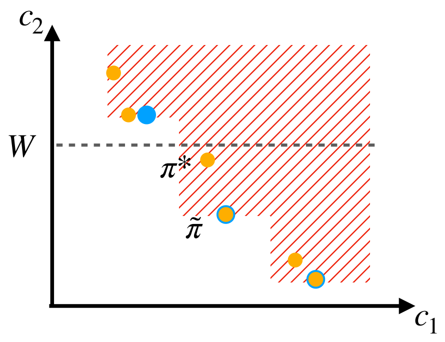


Figure 3.13: Example of the Pareto frontier (whose costs are shown by the orange dots) and an $(\varepsilon, 0)$-approximate Pareto frontier (whose costs are shown by the blue dots) for a WCSP problem instance. The shaded region shows the costs that are $(\varepsilon, 0)$-dominated by at least one blue dot. Solutions π^* and $\tilde{\pi}$ are an optimal solution and a $(1+\varepsilon)$-suboptimal solution of the WCSP instance, respectively.

We have proposed WC-A*pex [81], a bounded-suboptimal WCSP algorithm. Given a WCSP problem instance and a non-negative approximate factor ε, WC-A*pex finds a solution whose c_1-value is no larger than $(1+\varepsilon)c_1^*$. To describe WC-A*pex, we need to generalize the definition of ε-dominance and allow different approximation factors for c_1 and c_2. We say that a path π $(\varepsilon, 0)$-*dominates* another path π' iff $c_1(\pi) \leq (1+\varepsilon)c_1(\pi')$ and $c_2(\pi) \leq c_2(\pi')$. An $(\varepsilon, 0)$-*approximate Pareto frontier* $\Pi_{\varepsilon,0}$ is a set of paths from s_{start} to s_{goal} such that every path from s_{start} to s_{goal} is $(\varepsilon, 0)$-dominated by at least one path in $\Pi_{\varepsilon,0}$. The following property shows that a $(1+\varepsilon)$-suboptimal solution for a WCSP problem instance can be found in an $(\varepsilon, 0)$-approximate Pareto frontier:

Property 3.3. *Given a WCSP instance $P = \langle G, s_{start}, s_{goal}, W\rangle$ and $\varepsilon \geq 0$, any $(\varepsilon, 0)$-approximate Pareto frontier (from s_{start} to s_{goal}) $\Pi_{\varepsilon,0}$ contains a $(1+\varepsilon)$-suboptimal solution for P if a solution of P exists.*

Proof. Because a solution of P exists, there exists an optimal solution π^* of P. By the definition of an $(\varepsilon, 0)$-approximate Pareto frontier, there exists a path $\tilde{\pi} \in \Pi_{\varepsilon,0}$ with $c_1(\tilde{\pi}) \leq (1+\varepsilon) \cdot c_1(\pi^*)$ and $c_2(\tilde{\pi}) \leq c_2(\pi^*) \leq W$. We can see that $\tilde{\pi}$ is a $(1+\varepsilon)$-suboptimal solution of P. □

See Figure 3.13 for a visualization of an $(\varepsilon, 0)$-approximate Pareto frontier and a $(1+\varepsilon)$-suboptimal solution of a WCSP instance. Based on this property, WC-A*pex combines WC-A* [3], a state-of-the-art optimal WCSP algorithm, and A*pex. It uses apex-path pairs as its search nodes and guarantees that all apex-path pairs are $(\varepsilon, 0)$-bounded during its search. An apex-path pair $\langle \mathbf{A}, \pi \rangle$ is $(\varepsilon, 0)$-bounded iff $c_1(\pi) \leq (1+\varepsilon)A_1$ and $c_2(\pi) \leq A_2$. In Theorem 1 of our paper [81], we show that WC-A*pex finds a $(1+\varepsilon)$-suboptimal solution given a WCSP problem instance for which a solution exists. Our experimental [81] results show that WC-A*pex with $\varepsilon = 0.01$ (that is, with a guaranteed suboptimality of at most 1%) achieves a speed-up of up to an order of magnitude over WC-A*.

3.6.2 The Multi-Objective Multi-Agent Path Finding Problem

The Multi-Agent Path Finding (MAPF) problem [68] is the problem of finding a set of collision-free paths for a team of agents. It is related to many real-world applications [79, 48]. A (MAPF) *solution* is a set of collision-free paths, one for each agent. Computing minimum-cost solutions of MAPF problem instances is known to be NP-hard [80, 45]. The Multi-Objective Multi-Agent Path Finding (MO-MAPF) problem [56] generalizes the MAPF problem by considering multiple costs.

Most existing MO-MAPF algorithms, such as MO-M* [57], MO-CBS [56], and BB-MO-CBS [55], aim to compute the Pareto frontier. However, computing the Pareto frontier can be very time-consuming. We have introduced BB-MO-CBS-pex [77], an approximate MO-MAPF algorithm that computes an ε-approximate Pareto frontier for a given approximation factor ε. BB-MO-CBS-pex builds upon BB-MO-CBS [55], a state-of-the-art MO-MAPF algorithm, and leverages A*pex to speed up different parts of BB-MO-CBS. In our paper, we also provide two speed-up techniques for BB-MO-CBS-pex. Additionally, we introduce another approximate MO-MAPF algorithm, called BB-MO-CBS-k, which builds upon BB-MO-CBS-pex and computes up to k solutions for a given value of k. BB-MO-CBS-k is useful when it is unclear how to determine an appropriate approximation factor but the desired number of solutions is known.

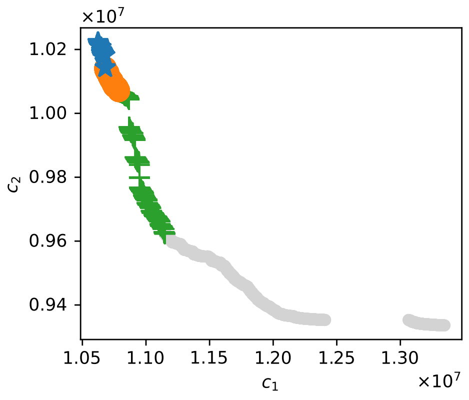
Our experimental results show that both BB-MO-CBS-pex and BB-MO-CBS-k solve significantly more instances than BB-MO-CBS for different approximation factors and k-values, respectively. Additionally, we compare BB-MO-CBS-pex with a baseline approximate variant of BB-MO-CBS and show that BB-MO-CBS-pex achieves speed-ups up to two orders of magnitude over the baseline.

Chapter 4

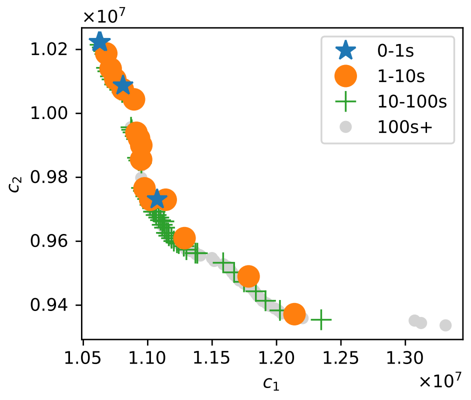
Speeding up Multi-Objective Search via Anytime Algorithms

In practice, we often have limited time available to solve a multi-objective search problem instance. Computing the Pareto frontier can be impractical in such cases. Chapter 3 has shown that an approximate multi-objective search algorithm, such as A*pex, can compute an ε-approximate Pareto frontier much faster than computing the Pareto frontier. However, it is often unclear how to specify the approximation factor with which the approximate multi-objective search algorithm makes the best use of the available time. If there is time available even after the algorithm terminates, we might want to continue computing more solutions. In this chapter, we are interested in computing solutions that collectively approximate the Pareto frontier while making use of the available time as much as possible. To achieve this, we investigate anytime approximate multi-objective search algorithms, which compute an initial approximate frontier quickly and then work on finding better and better approximate frontiers until eventually finding the entire Pareto frontier.

This chapter is based on [85].



(a) BOA*



(b) A-A*pex

Figure 4.1: The solutions found by BOA* and A-A*pex for a bi-objective search problem instance on the FLA road network. Different markers indicate different time ranges when the solutions were found.

We introduce an anytime approximate multi-objective search algorithm, called A-A*pex, which builds upon A*pex. In each iteration of its main loop, A-A*pex performs an approximate multi-objective search with an approximation factor that is smaller than the approximate factors of the previous iterations. A straightforward approach for achieving this is to restart the search from scratch for each iteration. However, this approach can be inefficient since the search effort is duplicated across different approximation factors. Therefore, we propose an approach that addresses this inefficiency by reusing previous search effort. It does sufficient bookkeeping to allow each iteration to resume the search from paths that were pruned in the previous iteration. Additionally, we propose a hybrid variant of A-A*pex which first restarts the search from scratch for each iteration and then starts to reuse its search effort in later iterations. Although existing work on anytime single-objective search has already investigated reusing search effort [43] or restarting from scratch [60], generalizing these techniques to multi-objective search is not trivial. In this chapter, we show how to reuse the search effort of A*pex despite its unique merge operations.

Existing multi-objective search algorithms, such as BOA* and LTMOA*, can also be viewed as "anytime" algorithms: We can stop these algorithms at any time and consider the solutions that they have computed so far as result. However, because these algorithms compute the solutions in lexicographically increasing order of their costs, the solutions they compute "cover" only part of the Pareto frontier and completely miss the rest. Figure 4.1 shows the solutions found by BOA* and A-A*pex for a bi-objective search problem instance on the FLA road network.[1] Different markers represent different time ranges when the solutions were found. BOA* finds solutions in a lexicographic order, that is, with increasing c_1-values and decreasing c_2-values. A-A*pex, our proposed algorithm, first finds solutions that roughly approximate the Pareto frontier and then works on finding more solutions while time allows. Given the same limited amount of time, A-A*pex can compute solutions that collectively approximate the Pareto frontier much better than those of BOA*. For example, if one is given only 10 seconds, running BOA* results in solutions that are represented by the blue and the orange markers in Figure 4.1a, which completely miss solutions with small c_2-values. On the other hand, running A-A*pex for 10 seconds results in solutions that are represented by the blue and the orange markers in Figure 4.1b, whose costs are much better distributed than those of the solutions found by BOA* within 10 seconds.

In our experimental results, we evaluate different variants of A-A*pex and show that reusing search effort in later iterations significantly reduces its runtime. We also show that A-A*pex often computes solutions that collectively approximate the Pareto frontier much better than the solutions found by state-of-the-art multi-objective search algorithms when the given time is insufficient for finding the entire Pareto frontier.

[1] http://users.diag.uniroma1.it/challenge9/download.shtml

This chapter is organized as follows: We begin by describing the related work pertinent to A-A*pex in Section 4.1. We provide a detailed description of A-A*pex in Section 4.2. We then prove its correctness and completeness in Section 4.3 before presenting our experimental results in Section 4.4 and our summary in Section 4.5.

4.1 Background

Researchers have developed various anytime single-objective search algorithms [43, 8, 67, 13]. An anytime search algorithm typically builds upon a suboptimal search algorithm. By calling the suboptimal search algorithm repeatedly (with smaller and smaller suboptimality factors), the anytime search algorithm progressively finds better and better solutions while time allows. In practice, an anytime search algorithm can often quickly compute a solution whose cost is close to optimal. Therefore, it is useful when the available time is limited or a priori unknown.

There are only a few existing works on anytime multi-objective search algorithms. In our previous work, we proposed Anytime BOA*-ε (A-BOA*-ε), an anytime bi-objective search algorithm that builds upon BOA*-ε [86]. A-BOA*-ε uses a so-called *interval* to keep track of its search progress and reuse its previous search effort. A-BOA*-ε maintains a list I of intervals. Each interval contains a set of paths. A-BOA*-ε initializes I with the interval that contains only the path $[s_{start}]$. In each iteration, it chooses an interval in I based on estimating which part of the Pareto frontier is the least approximated and removes this interval from I. It then initializes $Open$ with the paths in the chosen interval and calls a modified variant of BOA*-ε to compute new solutions. Each new solution π corresponds to a new interval, and this interval contains those paths that have been pruned because their $\mathbf{f}$-values are ε-dominated by $\mathbf{f}(\pi)$ (according to the relaxed pruning condition of BOA*-ε) although they are not weakly dominated by $\mathbf{f}(\pi)$. A-BOA*-ε then adds these new intervals in I and continues to the next iteration. A-BOA*-ε terminates and finds a Pareto frontier when every interval in I contains an empty set of paths. Our experimental results show that, given a limited amount of time, A-BOA*-ε can find solutions that collectively approximate the Pareto frontier much better than the solutions found by BOA*. However, as we have shown in Chapter 3, BOA*-ε, which A-BOA*-ε builds upon, identifies only a small subset of the paths that can be pruned and is not as efficient as A*pex in practice. It is also unclear how to generalize A-BOA*-ε to more than two objectives.

4.2 A-A*pex

As A-BOA*-ε builds upon a modified variant of BOA*-ε, A-A*pex builds upon a modified variant of A*pex. It calls this variant of A*pex with smaller and smaller ε-values to compute new solutions and eventually find the Pareto frontier. From one iteration to the next, A-A*pex can either reuse its previous search effort or restart its search from scratch. We first describe the variant of A-A*pex that reuses its previous search effort: We begin with its general search strategy in Section 4.2.1. We then describe how it reuses its previous search effort in Section 4.2.2 and a technique to improve the dominance checks for this variant of A-A*pex in Section 4.2.3. Finally, in Section 4.2.4, we describe the variant of A-A*pex that restarts its search from scratch, which is a simple algorithm that differs only in a few lines of pseudocode from the variant that reuses its previous search effort.

Algorithm 7: A-A*pex

```
Input  : A problem instance $\langle G, s_{start}, s_{goal}\rangle$, a consistent heuristic function $\mathbf{h}$, and an
         approximation factor update scheme $\mathrm{getNextEps}()$
Output: A Pareto frontier
$Pruned \leftarrow \{[s_{\text{start}}]\}$
$Sols \leftarrow \emptyset$
while Search not halted do
    $\varepsilon_{\text{curr}} \leftarrow \mathrm{getNextEps}()$
    $Open \leftarrow \emptyset$
    $Pruned' \leftarrow Pruned$; $Pruned \leftarrow \emptyset$
    for each $\pi \in Pruned'$ do
        $\mathrm{AddToOpen}(\langle \mathbf{c}(\pi), \pi\rangle)$
    $\mathrm{FindApproxPF}(\varepsilon_{\text{curr}})$
    if $Pruned = \emptyset$ then
        break
return $Sols$
Function $\mathrm{FindApproxPF}(\varepsilon_{curr})$:
    $Sols' \leftarrow$ sort solutions in $Sols$ in lexicographically increasing order of their costs
    $\mathbf{C}^{\text{tr}}_{\text{sol}} \leftarrow \emptyset$
    for each $s \in S$ do
        $\mathbf{G}^{\text{tr}}_{\text{cl}}(s) \leftarrow \emptyset$
    while $Open \neq \emptyset$ do
        extract an apex-path pair $\mathcal{AP} = \langle \mathbf{A}, \pi\rangle$ from $Open$ with the lexicographically
          smallest $\mathbf{f}$-value
        while $c_1(Sols'.top()) \leq (1 + \varepsilon_{curr}) f_1(\mathcal{AP})$ do
            $\mathrm{Update}(\mathbf{C}^{\text{tr}}_{\text{sol}}, Tr(\mathbf{c}(Sols'.top())))$
            pop $Sols'.top()$ from $Sols'$
        if IsDominated($\mathcal{AP}$) then     // IsDominated is defined in Algorithm 8
            continue
        $\mathrm{Update}(\mathbf{G}^{\text{tr}}_{\text{cl}}(s(\mathcal{AP})), Tr(\mathbf{g}(\mathcal{AP})))$
        if $s(\mathcal{AP}) = s_{goal}$ then
            remove solutions weakly dominated by $\pi$ from $Sols$
            $\mathrm{Update}(\mathbf{C}^{\text{tr}}_{\text{sol}}, Tr(\mathbf{c}(\pi)))$
            add $\pi$ to $Sols$
            continue
        for each $e \in out(s(\mathcal{AP}))$ do
            $\mathcal{AP}' \leftarrow \langle \mathbf{A} + \mathbf{c}(e), \mathrm{extend}(\pi, e)\rangle$
            if IsDominated($\mathcal{AP}'$) then
                continue
            AddToOpen($\mathcal{AP}'$)     // AddToOpen is defined in Algorithm 8
```

4.2.1 Search Strategy

Algorithm 7 shows the pseudocode of the variant of A-A*pex that reuses its previous search effort. The input to Algorithm 7 is a problem instance, a consistent heuristic $\mathbf{h}$, and an approximation factor update scheme encoded by the $\mathrm{getNextEps}$ function. A-A*pex maintains a list $Pruned$ of pruned paths, which is initialized with path $[s_{start}]$ (Line 1), and a set $Sols$ of solutions, which is initialized with the empty set (Line 2). In each iteration of its main loop (Lines 3-11), A-A*pex first calls $\mathrm{getNextEps}$ to decrease the *current approximation factor* $\varepsilon_{\mathrm{curr}}$ (Line 4). It then initializes $Open$ with the paths in $Pruned$ (Lines 5-8): A-A*pex first moves the paths from $Pruned$ to another set $Pruned'$ (Line 6) and then calls $\mathrm{AddToOpen}$ with each path in $Pruned'$. Some of these paths might be put back into $Pruned$ by $\mathrm{AddToOpen}$, which we will explain later. A-A*pex then calls $\mathrm{FindApproxPF}$ to compute an $\varepsilon_{\mathrm{curr}}$-approximate frontier (Line 9).

Lines 13-35 show the $\mathrm{FindApproxPF}$ function, which is similar to Lines 3-21 of A*pex in Algorithm 6. The $\mathrm{IsDominated}$ and $\mathrm{AddToOpen}$ functions of $\mathrm{FindApproxPF}$ are modified to identify paths that might still be extendable to Pareto-optimal solutions and add these paths to $Pruned$. We will describe these two modified functions in detail in the next section.

Like A*pex, the $\mathrm{FindApproxPF}$ function maintains a set of truncated cost vectors $\mathbf{C}^{\mathrm{tr}}_{\mathrm{sol}}$ for checking if the $\mathbf{f}$-value of a given apex-path pair is $\varepsilon_{\mathrm{curr}}$-dominated by the cost of any solution in $Sols$. Unlike A*pex, it also needs to consider the solutions computed in the previous iterations of $\mathrm{FindApproxPF}$ in $\mathbf{C}^{\mathrm{tr}}_{\mathrm{sol}}$. It first stores these solutions in $Sols'$ and sorts them in lexicographically increasing order of their costs (Line 14). After extracting an apex-path pair $\mathcal{AP}$ from $Open$, $\mathrm{FindApproxPF}$ adds to $\mathbf{C}^{\mathrm{tr}}_{\mathrm{sol}}$ the costs of those solutions in $Sols'$ whose c_1-values are no larger than $(1+\varepsilon_{\mathrm{curr}})$ times the f_1-value of $\mathcal{AP}$ and removes these solutions from $Sols'$ (Lines 20-22). As we will formally show in our theoretical results (Section 4.3), the $\mathbf{f}$-value of $\mathcal{AP}$ is $\varepsilon_{\text{curr}}$-dominated by the cost of a solution in $Sols$ iff there exists a vector in $\mathbf{C}^{\text{tr}}_{\text{sol}}$ that $\varepsilon_{\text{curr}}$-dominates $Tr(\mathbf{f}(\mathcal{AP}))$.

The FindApproxPF function initializes $\mathbf{G}^{\text{tr}}_{\text{cl}}$ on Lines 16-17 and does not reuse the truncated $\mathbf{g}$-values from previous iterations because, even if the truncated $\mathbf{g}$-value of an apex-path pair $\mathcal{AP}$ is weakly dominated by a vector $\mathbf{x}$ in $\mathbf{G}^{\text{tr}}_{\text{cl}}$ from previous iterations, $\mathbf{g}(\mathcal{AP})$ is not necessarily weakly dominated by the $\mathbf{g}$-value corresponding to $\mathbf{x}$. The $\varepsilon_{\text{curr}}$-value of the current iteration is also different from those of previous iterations. Thus, A-A*pex can expand apex-path pairs whose $\mathbf{g}$-values are weakly dominated by the $\mathbf{g}$-values of some expanded apex-path pairs containing the same states from previous iterations. However, this does not affect it computing an $\varepsilon_{\text{curr}}$-approximate frontier in each iteration because it still considers the solutions in $Sols$ from previous iterations in its dominance checks.

When FindApproxPF returns from Line 9, $Sols$ is an $\varepsilon_{\text{curr}}$-approximate frontier. If $Pruned$ is empty, FindApproxPF has not identified any paths that might still be extendable to Pareto-optimal solutions that are not in $Sols$. Thus, A-A*pex breaks from the main loop (Line 11) and returns $Sols$ as a Pareto frontier (Line 12).

4.2.2 Reusing Search Effort

A*pex gains its efficiency by merging paths and not storing all paths explicitly. However, the paths A*pex discards when pruning or merging apex-path pairs might still be extendable to Pareto-optimal solutions. One can store such representative paths and resume the search from them later. For example, consider the case where A*pex (Algorithm 6) prunes an apex-path pair $\mathcal{AP}$ because its $\mathbf{f}$-value is ε-dominated by the cost of some solution π_{sol} in $Sols$ (Line 24). The representative

Algorithm 8: The IsDominated and AddToOpen functions for A-A*pex

```
1   Function IsDominated($\mathcal{AP} = \langle \mathbf{A}, \pi \rangle$):
2       if $\exists \mathbf{x} \in \mathbf{C}^{tr}_{sol} : \mathbf{x} \preceq_{\varepsilon_{curr}} Tr(\mathbf{f}(\mathcal{AP}))$ then
+3          $\pi' \leftarrow$ the solution that corresponds to $\mathbf{x}$
+4          if not $\mathbf{c}(\pi') \preceq \mathbf{f}(\pi)$ then
+5              add $\pi$ to $Pruned$
6          return true
7       if $\exists \mathbf{x} \in \mathbf{G}^{tr}_{cl}(s(\mathcal{AP})) : \mathbf{x} \preceq Tr(\mathbf{g}(\mathcal{AP}))$ then
+8          $\pi' \leftarrow$ the representative path of the apex-path pair corresponding to $\mathbf{x}$
+9          if not $\mathbf{c}(\pi') \preceq \mathbf{c}(\pi)$ then
+10             add $\pi$ to $Pruned$
11         return true
12      return false
13  Function AddToOpen($\mathcal{AP} = \langle \mathbf{A}, \pi \rangle$):
14      for each $\mathcal{AP}' = \langle \mathbf{A}', \pi' \rangle \in Open[s(\mathcal{AP})]$ do
15          $\mathcal{AP}_{\text{new}} = \langle \mathbf{A}_{\text{new}}, \pi_{\text{new}} \rangle \leftarrow merge(\mathcal{AP}, \mathcal{AP}')$
16          if $\mathcal{AP}_{new}$ is $\varepsilon_{curr}$-bounded then
17              remove $\mathcal{AP}'$ from $Open$
18              add $\mathcal{AP}_{\text{new}}$ to $Open$
+19             $\pi_{\text{pruned}} \leftarrow \pi'$ if $\pi = \pi_{\text{new}}$ or $\pi$ otherwise
+20             if not $\mathbf{c}(\pi_{new}) \preceq \mathbf{c}(\pi_{pruned})$ then
+21                 add $\pi_{\text{pruned}}$ to $Pruned$
22              return
23      add $\mathcal{AP}$ to $Open$
24      return
```

path π of $\mathcal{AP}$ might still be extendable to a Pareto-optimal solution if $\mathbf{f}(\pi)$ (which weakly dominates the cost of any solution extending π) is not weakly dominated by $\mathbf{c}(\pi_{\text{sol}})$. Similarly, from the representative paths of the apex-path pairs pruned on Line 26 and the representative paths that are not chosen as new representative paths when merging apex-path pairs on Line 30, we can also identify paths that might still be extendable to Pareto-optimal solutions. Our technique for reusing previous search effort is based on these observations.

Algorithm 8 shows the IsDominated and AddToOpen functions for A-A*pex. We use "+" before the line numbers to indicate the changes in these two functions compared to the IsDominated and AddToOpen functions in Algorithm 6:

1. **Lines 3-5:** Each vector $\mathbf{x}$ in $\mathbf{C}^{\text{tr}}_{\text{sol}}$ corresponds to some solution in $Sols$. If the truncated $\mathbf{f}$-value of an apex-path pair $\mathcal{AP}$ is $\varepsilon_{\text{curr}}$-dominated by some vector $\mathbf{x}$ in $\mathbf{C}^{\text{tr}}_{\text{sol}}$ and the representative path π of $\mathcal{AP}$ satisfies that $\mathbf{f}(\pi)$ is not weakly dominated by the cost of the solution in $Sols$ that corresponds to $\mathbf{x}$, A-A*pex adds π to $Pruned$.

2. **Lines 8-10:** For each vector $\mathbf{x}$ in $\mathbf{G}^{\text{tr}}_{\text{cl}}(s)$ for any state s, A-A*pex maintains the representative path of the apex-path pair whose truncated $\mathbf{g}$-value equals $\mathbf{x}$ and resulted in $\mathbf{x}$ being added to $\mathbf{G}^{\text{tr}}_{\text{cl}}(s)$. If the truncated $\mathbf{g}$-value of an apex-path pair $\mathcal{AP}$ is weakly dominated by some vector $\mathbf{x}$ in $\mathbf{G}^{\text{tr}}_{\text{cl}}(s(\mathcal{AP}))$ and the representative path π of $\mathcal{AP}$ is not weakly dominated by the representative path of the apex-path pair corresponding to $\mathbf{x}$, A-A*pex adds π to $Pruned$.

3. **Lines 19-21:** When merging two apex-path pairs, one of their representative paths is chosen as the new representative path. Let π_{new} denote the chosen representative path and π_{pruned} denote the other path. If π_{pruned} is not weakly dominated by π_{new}, A-A*pex adds π_{pruned} to $Pruned$.

These three changes cover all possible places where FindApproxPF "disregards" a path. Conceptually, FindApproxPF does not need to consider these paths because any solution π_{sol} that extends such a path π is known to be $\varepsilon_{\text{curr}}$-dominated by some other solution π'_{sol}. However, if π_{sol} is not weakly dominated by π'_{sol}, it is still possible that π_{sol} is in the Pareto frontier. Hence, FindApproxPF adds π to $Pruned$.

Example 7. *We use the bi-objective search problem instance in Figure 2.1 to demonstrate how A-A*pex works. Assume that the sequence of ε_{curr}-values output by* getNextEps *begins with 0.2, and the value of ε_{curr} is divided by 10 after every iteration.*

*When A-A*pex calls* $\mathrm{FindApproxPF}$ *for the first time with* $\varepsilon_{curr} = 0.2$*, the trace of* $Open$*, generated apex-path pairs, and* g_2^{min} *is the same as the one in Table 3.2. As we have shown in Example 6, when* $\mathrm{FindApproxPF}$ *returns, there are two solutions in* $Sols$*, with costs* $(7, 10)$ *and* $(12, 5)$*, respectively. Three merge operations have happened:*

1. *In Iteration 2,* $\mathrm{FindApproxPF}$ *merges apex-path pairs* $\langle s_2, (2, 2), (6, 6)\rangle$ *and* $\langle s_2, (3, 1), (7, 5)\rangle$*. The representative path with cost* $(3, 1)$ *is chosen as the new representative path. The other representative path with cost* $(2, 2)$*, which is not weakly dominated by* $(3, 1)$*, is added to* $Pruned$*.*

2. *In Iteration 3,* $\mathrm{FindApproxPF}$ *merges apex-path pairs* $\langle s_3, (4, 2), (6, 5)\rangle$ *and* $\langle s_3, (3, 4), (6, 7)\rangle$*. The representative path with cost* $(4, 2)$ *is chosen as the new representative path. The other representative path with cost* $(3, 4)$*, which is not weakly dominated by* $(4, 2)$*, is added to* $Pruned$*.*

3. *In Iteration 5,* $\mathrm{FindApproxPF}$ *merges apex-path pairs* $\langle s_4, (6, 4), (11, 5)\rangle$ *and* $\langle s_4, (5, 9), (10, 10)\rangle$*. The representative path with cost* $(6, 4)$ *is chosen as the new representative path. The other representative path with cost* $(5, 9)$*, which is not weakly dominated by* $(6, 4)$*, is added to* $Pruned$*.*

Therefore, when $\mathrm{FindApproxPF}$ *returns, there are three paths in* $Pruned$*.*

*When A-A*pex calls* $\mathrm{FindApproxPF}$ *for the second time with* $\varepsilon_{curr} = 0.02$*,* $Open$ *is initialized with the three apex-path pairs that correspond to the three paths in* $Pruned$*. Table 4.1 shows the trace of* $Open$*, generated apex-path pairs, and* g_2^{min}*. In the table, we use tuple* $\langle s(\mathcal{AP}), \mathbf{c}(\pi), \mathbf{f}(\mathcal{AP})\rangle$ *to denote an apex-path pair* $\mathcal{AP} = \langle \mathbf{A}, \pi\rangle$*.*

Iter	*Open* $\langle s(\mathcal{AP}), \mathbf{c}(\pi), \mathbf{f}(\mathcal{AP})\rangle$	Generated $\langle s(\mathcal{AP}), \mathbf{c}(\pi), \mathbf{f}(\mathcal{AP})\rangle$	Update of $g_2^{\min}(s(x))$
1	$\langle s_2, (2,2), (6,6)\rangle *$ $\langle s_3, (3,4), (6,7)\rangle$ $\langle s_4, (5,9), (11,10)\rangle$	$\langle s_3, (3,3), (6,6)\rangle$ (merged)	$g_2^{\min}(s_2) = 2$
2	$\langle \mathbf{s_3}, (\mathbf{3}, \mathbf{3}), (\mathbf{6}, \mathbf{6})\rangle *$ $\langle s_4, (5,9), (11,10)\rangle$	$\langle s_5, (4,4), (6,6)\rangle$ $\langle s_4, (4,10), (10,11)$ (discarded)	$g_2^{\min}(s_3) = 3$
3	$\langle s_5, (4,4), (6,6)\rangle *$ $\langle s_4, (5,9), (11,10)\rangle$	$\langle s_4, (5,5), (11,6)\rangle$ (merged) $\langle s_{goal}, (6,11), (6,11)$	$g_2^{\min}(s_5) = 4$
4	$\langle s_{goal}, (6,11), (6,11)\rangle *$ $\langle \mathbf{s_4}, (\mathbf{5}, \mathbf{5}), (\mathbf{11}, \mathbf{6})\rangle$		$g_2^{\min}(s_{goal}) = 11$
5	$\langle s_4, (5,5), (11,6)\rangle *$	$\langle s_{goal}, (11,6), (11,6)\rangle$	$g_2^{\min}(s_4) = 5$
6	$\langle s_{goal}, (11,6), (11,6)\rangle *$		$g_2^{\min}(s_{goal}) = 6$
7	empty		

Table 4.1: Trace of $Open$, generated apex-path pairs, and $g_2^{\min}$ when A-A*pex calls $\mathrm{FindApproxPF}$ with $\varepsilon_{\text{curr}} = 0.02$ to solve the example problem instance in Figure 2.1. "$*$" marks the apex-path pair that is extracted in that iteration. Boldface font marks the apex-path pairs that result from merging two apex-path pairs in the previous iteration.

In Iteration 1, $\mathrm{FindApproxPF}$ *merges the generated apex-path pair* $\langle s_3, (3,3), (6,6)\rangle$ *with apex-path pair* $\langle s_3, (3,4), (6,7)\rangle$*. The representative path with cost* $(3,3)$ *is chosen as the new representative path. The other representative path with cost* $(3,4)$*, which is weakly dominated by* $(3,3)$*, is not added to* $Pruned$*.*

In Iteration 2, $\mathrm{FindApproxPF}$ *discards the generated apex-path pair* $\langle s_4, (4,10), (10,11)\rangle$ *because its* $\mathbf{f}$*-value is* ε_{curr}*-dominated by the cost* $(7,10)$ *of a solution in* $Sols$*. This solution was found when A-A*pex called* $\mathrm{FindApproxPF}$ *for the first time. The* $\mathbf{f}$*-value of the representative path of this generated apex-path pair is* $(4,10) + \mathbf{h}(s_4) = (10,11)$*, which is also weakly dominated by* $(7,10)$*. Therefore, this representative path is not added to* $Pruned$*.*

In Iteration 3, $\mathrm{FindApproxPF}$ *merges the generated apex-path pair* $\langle s_4, (5,5), (11,6)\rangle$ *with apex-path pair* $\langle s_4, (5,9), (11,10)\rangle$*. The representative path with cost* $(5,5)$ *is chosen as the new representative path. The other representative path with cost* $(5,9)$*, which is weakly dominated by* $(5,5)$*, is not added to* $Pruned$*.*

In Iterations 4 and 6, $\mathrm{FindApproxPF}$ *expands apex-path pairs that contain state* s_{goal} *and adds solutions with costs* $(6, 11)$ *and* $(11, 6)$ *to* $Sols$*. When* $\mathrm{FindApproxPF}$ *returns for the second time, there are four solutions in* $Sols$*. These four solutions consist of the Pareto frontier shown in Figure 2.1b. Because* $Pruned$ *is empty, A-A*pex terminates and returns* $Sols$*.*

4.2.3 Enhanced Dominance Checks

Although A-A*pex does not reuse the truncated $\mathbf{g}$-values from previous iterations for dominance checks, it can still prune an apex-path pair $\mathcal{AP}$ if $\mathbf{g}(\mathcal{AP})$ is weakly dominated by the cost of the representative path π' of an apex-path pair that was expanded in previous iterations and contains the same state as $\mathcal{AP}$. In such a case, the entire set of paths that $\mathcal{AP}$ corresponds to is weakly dominated by π'. One can thus enhance the dominance checks of A-A*pex by maintaining the set of undominated costs $\mathbf{C}(s)$ of the representative paths of all expanded apex-path pairs for each state s and using these sets for dominance checks. This requires changing the $\mathrm{IsDominated}$ function to check if the $\mathbf{g}$-value of the input apex-path pair $\mathcal{AP}$ is weakly dominated by any vector in $\mathbf{C}(s(\mathcal{AP}))$ and adding one line after Line 25 of Algorithm 7 to update $\mathbf{C}(s(\mathcal{AP}))$ before expanding $\mathcal{AP}$.

Although the enhanced dominance checks enable A-A*pex to prune more nodes, performing them and updating $\mathbf{C}(s)$ can be time-consuming. It is also unclear how to use the dimensionality reduction technique in these checks. In our experimental evaluation (Section 4.4), we will study whether these checks improve the efficiency of A-A*pex.

4.2.4 Restarting the Search From Scratch

Instead of reusing its previous search effort, A-A*pex can also restart the search from scratch in each iteration. This requires changes only to Lines 5-8 of Algorithm 7, where A-A*pex now initializes $Open$ with path $[s_{start}]$ instead of the paths in $Pruned$. This variant of A-A*pex cannot use the enhanced dominance checks described in Section 4.2.3 because it needs to expand apex-path pairs with the same representative paths as those expanded in the previous iterations of FindApproxPF. If we use the enhanced dominance checks in this case, the apex-path pair extracted in the first iteration, whose $\mathbf{g}$-value is $\mathbf{0}$ and whose representative path is $[s_{start}]$, would be pruned. Thus, FindApproxPF would fail to compute more solutions.

As $\varepsilon_{\text{curr}}$ decreases, FindApproxPF often returns more expanded nodes and fewer paths in $Pruned$. Hence, restarting from scratch becomes less efficient than using $Pruned$ to initialize $Open$. Let $\#_{\text{exp}}$ and $\#_{\text{pruned}}$ denote the numbers of expanded nodes and pruned paths, respectively. We propose a variant of A-A*pex, called A-A*pex-hybrid, that first restarts the search from scratch in each iteration and starts reusing its search effort when the ratio of $\#_{\text{exp}}$ and $\#_{\text{pruned}}$ in the previous iteration is larger than a given threshold. It then keeps reusing its search effort until it terminates. In the experiments, we empirically set the threshold to five based on our preliminary results.

In our preliminary study, we also tried a variant of A-A*pex that starts reusing their search effort when $\varepsilon_{\text{curr}}$ is smaller than a given threshold. However, this variant had a larger average runtime than A-A*pex-hybrid, and hence, we chose not to include it in the experiments. It is future work to study when reusing the search effort is more efficient than restarting the search from scratch.

4.3 Correctness and Completeness

This section provides theoretical results for A-A*pex. We study only the variant of A-A*pex that reuses its previous search effort because all theorems in this section trivially hold also for the variant of A-A*pex that restarts its search from scratch. Theorem 4.1 shows that A-A*pex computes an $\varepsilon_{\text{curr}}$-approximate frontier in each iteration. Theorem 4.2 shows that A-A*pex eventually computes a Pareto frontier.

Given a solution $\pi_{\text{sol}} = [s_1(= s_{start}), s_2 \dots s_L(= s_{goal})]$, we use $\pi_{\text{sol}}^{(l)}$, $l = 1, 2 \dots L$, to denote its prefix $[s_1, s_2 \dots s_l]$. We define a path π to be *l-compatible* with π_{sol} iff (i) the last state of π is s_l and (ii) $\mathbf{c}(\pi) \preceq \mathbf{c}(\pi_{\text{sol}}^{(l)})$. We define a path π to be *compatible* with π_{sol} iff there exists an l for which π is l-compatible with π_{sol}. Thus, path $[s_{start}]$ is both 1-compatible and compatible with any solution.

Lemma 4.1. *Consider any solution $\pi_{sol} = [s_1, s_2 \dots s_L]$. If* FindApproxPF *expands (that is, reaches Line 26 with) an apex-path pair $\mathcal{AP}$ whose representative path is l-compatible with π_{sol} then there exists, when* FindApproxPF *terminates, (Case 1) a path in $Pruned$ that is compatible with π_{sol} or (Case 2) a solution in $Sols$ that weakly dominates π_{sol}.*

Proof. We prove this lemma by induction on l, starting from $l = L$ and going backward. Consider the case where FindApproxPF expands an apex-path pair $\mathcal{AP}$ whose representative path π is L-compatible with π_{sol}. π is a path to s_L $(= s_{goal})$, and $\mathbf{c}(\pi) \preceq \mathbf{c}(\pi_{\text{sol}}^{(L)}) = \mathbf{c}(\pi_{\text{sol}})$ because π is L-compatible with π_{sol}. FindApproxPF then adds π to $Sols$ on Line 29. There must exist a solution in $Sols$ that weakly dominates π_{sol} when FindApproxPF terminates because it only removes a solution from $Sols$ when adding another solution that weakly dominates it (Lines 27-29). Case 2 holds.

Assume that the lemma holds for $l+1$ and consider the case where $\mathrm{FindApproxPF}$ expands an apex-path pair $\mathcal{AP}$ whose representative path π is l-compatible with π_{sol}. Consider the child apex-path pair $\mathcal{AP}_{\text{ch}} = \langle \mathbf{A}_{\text{ch}}, \pi_{\text{ch}} \rangle$ of $\mathcal{AP}$ that $\mathrm{FindApproxPF}$ generates for edge $\langle s_l, s_{l+1} \rangle$ when reaching Line 32. We have

$$\begin{aligned} \mathbf{c}(\pi_{\text{ch}}) &= \mathbf{c}(\pi) + \mathbf{c}(\langle s_l, s_{l+1} \rangle) \\ &\preceq \mathbf{c}(\pi_{\text{sol}}^{(l)}) + \mathbf{c}(\langle s_l, s_{l+1} \rangle) \\ &= \mathbf{c}(\pi_{\text{sol}}^{(l+1)}). \end{aligned} \tag{4.1}$$

Therefore, π_{ch} is $(l+1)$-compatible with π_{sol} as the last state of π_{ch} is s_{l+1}. We distinguish the following two cases:

1. $\mathrm{FindApproxPF}$ prunes $\mathcal{AP}_{\text{ch}}$ on Line 24 or 34 because the $\mathrm{IsDominated}$ function returns true. $\mathrm{IsDominated}$ returns true only when it reaches Line 6 or 11 of Algorithm 8. If π_{ch} is added to $Pruned$, Case 1 holds. If not and $\mathrm{IsDominated}$ reaches Line 6 without adding π_{ch} to $Pruned$, then there exists a solution, that is, solution π' mentioned on Line 3, whose cost weakly dominates $\mathbf{f}(\pi_{\text{ch}})$ and whose truncated cost is in $\mathbf{C}_{\text{sol}}^{\text{tr}}$. $\mathbf{f}(\pi_{\text{ch}})$ weakly dominates $\mathbf{c}(\pi_{\text{sol}})$ because $\mathbf{c}(\pi_{\text{ch}}) \preceq \mathbf{c}(\pi_{\text{sol}}^{(l+1)})$ and $\mathbf{h}$ is consistent. $\mathrm{FindApproxPF}$ adds truncated cost vectors to $\mathbf{C}_{\text{sol}}^{\text{tr}}$ only on Lines 21 and 28 for solutions in $Sols$. Therefore, π' has been in $Sols$. Because $\mathrm{FindApproxPF}$ removes a solution from $Sols$ only when adding another solution that weakly dominates it, there must exist a solution in $Sols$ that weakly dominates $\mathbf{c}(\pi_{\text{sol}})$ when $\mathrm{FindApproxPF}$ terminates. Case 2 holds. If $\mathrm{IsDominated}$ reaches Line 11 without adding π_{ch} to $Pruned$, then there exists an expanded apex-path pair (namely, the one mentioned on Line 8) that contains state s_{l+1} and whose representative path π' weakly

dominates π_{ch}. Because of Eq. 4.1, we have $\mathbf{c}(\pi') \preceq \mathbf{c}(\pi_{\text{ch}}) \preceq \mathbf{c}(\pi_{\text{sol}}^{(l+1)})$. Therefore, π' is $(l+1)$-compatible with π_{sol}. The lemma holds for l because it holds for $l+1$.

2. FindApproxPF calls AddToOpen with $\mathcal{AP}_{\text{ch}}$ on Line 35. The algorithm might merge $\mathcal{AP}_{\text{ch}}$ with other apex-path pairs before extracting the resulting apex-path pair $\mathcal{AP}''$ of these merges from $Open$ on Line 19. During these merges, a representative path is completely discarded (i.e., neither chosen as the new representative path nor added to $Pruned$) only if it is weakly dominated by the other representative path. Therefore, if no path that weakly dominates π_{ch} is added to $Pruned$ during these merges, the representative path of $\mathcal{AP}''$ must weakly dominate π_{ch} (and hence be $(l+1)$-compatible with π_{sol}). If $\mathcal{AP}''$ is pruned on Line 24, the lemma holds as we have already proved in the previous case. Otherwise, $\mathcal{AP}''$ is expanded. The lemma holds for l because it holds for $l+1$. $\square$

Lemma 4.2. *For any solution π_{sol}, when A-A*pex reaches Line 4 of Algorithm 7, there exists (Case 1) a path in $Pruned$ that is compatible with π_{sol} or (Case 2) a solution in $Sols$ that weakly dominates π_{sol}.*

Proof. We prove this lemma by induction on each time A-A*pex reaches Line 4. When A-A*pex reaches Line 4 for the first time, path $[s_{start}]$ in $Pruned$ is compatible with π_{sol}, and hence the lemma holds. Assume that A-A*pex reaches Line 4 and the lemma has held so far. If there exists a solution in $Sols$ that weakly dominates π_{sol}, there must exist a solution in $Sols$ that weakly dominates π_{sol} when A-A*pex reaches Line 4 again because A-A*pex only removes a solution from $Sols$ when adding another solution that weakly dominates it (Line 27). Otherwise, there exists a path π' in $Pruned$ that is compatible with π_{sol}. A-A*pex then calls AddToOpen with apex-path pair $\langle \mathbf{c}(\pi'), \pi' \rangle$ on Line 8 and might merge it with other apex-path pairs before FindApproxPF

extracts the resulting apex-path pair $\mathcal{AP}''$ from $Open$. As we have already proved, if the algorithm does not add a path that weakly dominates π' to $Pruned$ during these merges, the representative path π'' of $\mathcal{AP}''$ must weakly dominate π' (and hence be compatible with π_{sol}). If $\mathcal{AP}''$ is pruned on Line 24, we distinguish the following cases:

1. Path π'' is added to $Pruned$ on Line 5 or 10 of Algorithm 8. The lemma holds.

2. IsDominated returns true on Line 6 of Algorithm 8 without adding π'' to $Pruned$. Then, there exists a solution in $Sols$ whose cost weakly dominates $\mathbf{f}(\pi'')$, which in turn weakly dominates $\mathbf{c}(\pi_{\text{sol}})$ because π'' is compatible with π_{sol} and $\mathbf{h}$ is consistent. Therefore, when A-A*pex reaches Line 4 of Algorithm 7 again, there still exists a solution in $Sols$ whose cost weakly dominates $\mathbf{c}(\pi_{\text{sol}})$. Thus, the lemma holds.

3. IsDominated returns true on Line 11 of Algorithm 8 without adding π'' to $Pruned$. Then, there exists an expanded apex-path pair that contains state $s(\mathcal{AP}'')$ and whose representative path weakly dominates π'' (and hence is compatible with π_{sol}). According to Lemma 4.1, Case 1 or 2 holds when FindApproxPF terminates and hence also holds when A-A*pex reaches Line 4 of Algorithm 7 again. Thus, the lemma holds.

Otherwise, $\mathcal{AP}''$ is expanded. From Lemma 4.1, Case 1 or 2 holds when FindApproxPF terminates. Because A-A*pex reaches Line 4 of Algorithm 7 again only after FindApproxPF terminates, the lemma holds. □

Lemma 4.3. *A-A*pex adds only ε_{curr}-bounded apex-path pairs to $Open$.*

Proof. Consider the apex-path pairs for which A-A*pex calls AddToOpen on Line 8. These apex-path pairs are $\varepsilon_{\text{curr}}$-bounded because the f-values of their representative paths are equal to (and hence weakly dominate) their f-values. The AddToOpen function (Algorithm 8) either adds an apex-path pair directly to $Open$ (Line 23) or adds it to $Open$ after merging it with another apex-path pair on condition that the resulting apex-path pair is $\varepsilon_{\text{curr}}$-bounded (Line 18). Therefore, A-A*pex adds only $\varepsilon_{\text{curr}}$-bounded apex-path pairs to $Open$ on Line 8.

A-A*pex then calls FindApproxPF on Line 9. When FindApproxPF reaches Line 19 to extract an apex-path pair from $Open$ for the first time, all apex-path pairs in $Open$ are $\varepsilon_{\text{curr}}$-bounded. The induction in the proof of Lemma 3.1 applies here. Therefore, A-A*pex adds only $\varepsilon_{\text{curr}}$-bounded apex-path pairs to $Open$ on Line 35. □

Lemma 4.4. *In each run of* FindApproxPF*, the sequence of extracted apex-path pairs has monotonically non-decreasing f_1-values.*

The proof of Lemma 3.2 applies here.

Lemma 4.5. *There exists a solution in $Sols$ whose cost ε_{curr}-dominates the $\mathbf{f}$-value of apex-path pair $\mathcal{AP}$ on Line 2 of the* IsDominated *function (Algorithm 8) if there exists a truncated cost in $\mathbf{C}^{tr}_{sol}$ that ε_{curr}-dominates the truncated $\mathbf{f}$-value of apex-path pair $\mathcal{AP}$.*

Proof. Assume that there exists a truncated cost vector $\mathbf{x}$ in $\mathbf{C}^{\text{tr}}_{\text{sol}}$ that $\varepsilon_{\text{curr}}$-dominates the truncated $\mathbf{f}$-value of some apex-path pair $\mathcal{AP}$. $\mathbf{x}$ was added to $\mathbf{C}^{\text{tr}}_{\text{sol}}$ on either Line 21 or 28 of Algorithm 7 for some solution π_{sol}. We have $Tr(\mathbf{c}(\pi_{\text{sol}})) = \mathbf{x}$ and hence

$$Tr(\mathbf{c}(\pi_{\text{sol}})) \preceq_{\varepsilon_{\text{curr}}} Tr(\mathbf{f}(\mathcal{AP})). \tag{4.2}$$

Suppose that $\mathbf{x}$ was added to $\mathbf{C}_{\text{sol}}^{\text{tr}}$ on Line 21. The IsDominated function is called by A-A*pex either on Line 23 for an apex-path pair extracted from $Open$ or on Line 33 for a generated apex-path pair. Thus, we distinguish the following two cases:

1. $\mathcal{AP}$ is an apex-path pair extracted from $Open$ on Line 19, and the IsDominated function is called on Line 23. If $\mathbf{x}$ was added before $\mathcal{AP}$ was extracted from $Open$, it was added when A-A*pex extracted another apex-path pair $\mathcal{AP}'$ from $Open$. According to the condition on Line 20, $c_1(\pi_{\text{sol}}) \leq (1+\varepsilon_{\text{curr}})f_1(\mathcal{AP}')$. From Lemma 4.4, we have $f_1(\mathcal{AP}') \leq f_1(\mathcal{AP})$, and hence, $c_1(\pi_{\text{sol}}) \leq (1+\varepsilon_{\text{curr}})f_1(\mathcal{AP})$. Otherwise, $\mathbf{x}$ was added right after $\mathcal{AP}$ was extracted from $Open$ and before A-A*pex called IsDominated for $\mathcal{AP}$. According to the condition on Line 20, $c_1(\pi_{\text{sol}}) \leq (1+\varepsilon_{\text{curr}})f_1(\mathcal{AP})$.

2. $\mathcal{AP}$ is an apex-path pair generated on Line 32 for some parent apex-path pair $\mathcal{AP}_{\text{par}}$, and the IsDominated function is called on Line 33. As we have already proved in the previous case, we have $c_1(\pi_{\text{sol}}) \leq (1+\varepsilon_{\text{curr}})f_1(\mathcal{AP}_{\text{par}})$ because $\mathbf{x}$ was added to $\mathbf{C}_{\text{sol}}^{\text{tr}}$ either before or right after $\mathcal{AP}_{\text{par}}$ was extracted from $Open$. Because the heuristic is consistent, we have $f_1(\mathcal{AP}_{\text{par}}) \leq f_1(\mathcal{AP})$. Therefore, $c_1(\pi_{\text{sol}}) \leq (1+\varepsilon_{\text{curr}})f_1(\mathcal{AP}_{\text{par}}) \leq (1+\varepsilon_{\text{curr}})f_1(\mathcal{AP})$.

In all cases, we have $c_1(\pi_{\text{sol}}) \leq (1+\varepsilon_{\text{curr}})f_1(\mathcal{AP})$. Combining this inequality and Eq. 4.2, we have $\mathbf{c}(\pi_{\text{sol}}) \preceq_{\varepsilon_{\text{curr}}} \mathbf{f}(\mathcal{AP})$.

Suppose that $\mathbf{x}$ was added to $\mathbf{C}_{\text{sol}}^{\text{tr}}$ on Line 28. Let $\mathcal{AP}_{\text{sol}} = \langle \mathbf{A}_{\text{sol}}, \pi_{\text{sol}} \rangle$ denote the apex-path pair with which A*pex reached Line 28 and added $\mathbf{x}$ to $\mathbf{C}_{\text{sol}}^{\text{tr}}$. According to Lemma 4.3, $\mathcal{AP}_{\text{sol}}$ is $\varepsilon_{\text{curr}}$-bounded. Hence, we have $c_1(\pi_{\text{sol}}) \leq (1+\varepsilon_{\text{curr}})f_1(\mathcal{AP}_{\text{sol}})$. According to Lemma 4.4 and because $\mathcal{AP}_{\text{sol}}$ is extracted before $\mathcal{AP}$, $f_1(\mathcal{AP}_{\text{sol}}) \leq f_1(\mathcal{AP})$. Therefore, we have $c_1(\pi_{\text{sol}}) \leq (1+\varepsilon_{\text{curr}})f_1(\mathcal{AP})$. Combining this inequality and Eq. 4.2, we have $\mathbf{c}(\pi_{\text{sol}}) \preceq_{\varepsilon_{\text{curr}}} \mathbf{f}(\mathcal{AP})$.

Therefore, there exists a solution in $Sols$ whose cost $\varepsilon_{\text{curr}}$-dominates the $\mathbf{f}$-value of apex-path pair $\mathcal{AP}$. $\square$

Lemma 4.6. *There exists a truncated* $\mathbf{g}$*-value in* $\mathbf{G}_{cl}^{tr}(s(\mathcal{AP}))$ *that weakly dominates the truncated* $\mathbf{g}$*-value of apex-path pair* $\mathcal{AP}$ *on Line 7 of the* IsDominated *function (Algorithm 8) iff there exists an expanded apex-path pair* $\mathcal{AP}'$ *that contains state* $s(\mathcal{AP})$ *and whose* $\mathbf{g}$*-value weakly dominates the* $\mathbf{g}$*-value of apex-path pair* $\mathcal{AP}$*.*

Lemma 4.6 is similar to Lemma 3.4 for A*pex. The proof of Lemma 3.4 builds on Lemma 3.2. After replacing Lemma 3.2 with Lemma 4.4, the proof of Lemma 3.4 applies to Lemma 4.6 as well.

Lemma 4.7. *Consider any solution* $\pi_{sol} = [s_1, s_2 \dots s_L]$*. At the beginning of each iteration of* FindApproxPF *(that is, before it executes Line 19), if the same run of* FindApproxPF *has expanded an apex-path pair* $\mathcal{AP}$ *with* $\mathbf{g}(\mathcal{AP}) \preceq \mathbf{c}(\pi_{sol}^{(j)})$ *and* $s(\mathcal{AP}) = s_j$ *for some* j*, then there exists (1) an apex-path pair* $\mathcal{AP}'$ *in* $Open$ *with* $\mathbf{g}(\mathcal{AP}') \preceq \mathbf{c}(\pi_{sol}^{(k)})$ *and* $s(\mathcal{AP}') = s_k$ *for some* $k > j$ *or (2) a solution in* $Sols$ *that* ε_{curr}*-dominates* π_{sol}*.*

Lemma 4.7 is similar to Lemma 3.6 for A*pex. The proof of Lemma 3.6 builds on Lemmas 3.1, 3.3, 3.4, and 3.5. Lemma 3.5 is about a property of apex-path pairs and hence also applies to FindApproxPF. After replacing Lemmas 3.1, 3.3, and 3.4 with Lemmas 4.3, 4.5, and 4.6, respectively, the proof of Lemma 3.4 applies to Lemma 4.7 as well.

Lemma 4.8. *Consider any solution* $\pi_{sol} = [s_1, s_2 \dots s_L]$*. At the beginning of each iteration of* FindApproxPF *(that is, before it executes Line 19), there always exists (1) an apex-path pair* $\mathcal{AP}'$ *in* $Open$ *with* $\mathbf{g}(\mathcal{AP}') \preceq \mathbf{c}(\pi_{sol}^{(j)})$ *and* $s(\mathcal{AP}') = s_j$ *for some* j *or (2) a solution in* $Sols$ *that* ε_{curr}*-dominates* π_{sol}*.*

Proof. A-A*pex reaches Line 4 before it executes FindApproxPF. From Lemma 4.2, for any solution π_{sol}, there exists a solution in $Sols$ that weakly dominates π_{sol} or a path in $Pruned$ that is compatible with π_{sol}. If π_{sol} is weakly dominated by some solution in $Sols$ when A-A*pex reaches Line 4, it will always be weakly dominated (and hence also $\varepsilon_{\text{curr}}$-dominated) by some solution in $Sols$ at the beginning of all future iterations of FindApproxPF because A-A*pex removes a solution from $Sols$ only if it is weakly dominated by a new solution. Otherwise, there exists a path π in $Pruned$ that is compatible with π_{sol} when A-A*pex reaches Line 4. Let j denote the index for which π is j-compatible with π_{sol}. When A-A*pex reaches Line 8 with this path π, it adds an apex-path pair to $Open$ that contains state s_j and whose apex is equal to and hence weakly dominates $\mathbf{c}(\pi)$, which in turn weakly dominates $\mathbf{c}(\pi_{\text{sol}}^{(j)})$. This apex-path pair might be merged several (more) times with other apex-path pairs. The resulting apex-path pair $\mathcal{AP}$ still satisfies that $\mathbf{g}(\mathcal{AP}) \preceq \mathbf{c}(\pi_{\text{sol}}^{(j)})$ and $s(\mathcal{AP}) = s_j$. Consider the beginning of an iteration of FindApproxPF. If apex-path pair $\mathcal{AP}$ has not been extracted from $Open$, the lemma holds. If apex-path pair $\mathcal{AP}$ has been extracted from $Open$ and was pruned because of the condition on Line 2 of Algorithm 8, from Lemma 4.5, there exists a solution in $Sols$ whose cost $\varepsilon_{\text{curr}}$-dominates the $\mathbf{f}$-value of $\mathcal{AP}$, which in turn weakly dominates the cost of π_{sol} because $\mathbf{f}(\mathcal{AP}) = \mathbf{g}(\mathcal{AP}) + \mathbf{h}(s_j) \preceq \mathbf{c}(\pi_{\text{sol}}^{(j)}) + \mathbf{h}(s_j) \preceq \mathbf{c}(\pi_{\text{sol}})$. Thus, the lemma holds for all future iterations. If apex-path pair $\mathcal{AP}$ has been extracted from $Open$ and was pruned because of the condition on Line 7 of Algorithm 8, from Lemma 4.6, FindApproxPF has expanded an apex-path pair that contains state s_j and whose apex weakly dominates the apex of $\mathcal{AP}$ (and hence weakly dominates the cost of $\pi_{\text{sol}}^{(j)}$). Then, the lemma holds because of Lemma 4.7. Otherwise, FindApproxPF has expanded $\mathcal{AP}$. Then, the lemma holds because of Lemma 4.7. □

Lemma 4.9. *Consider that* FindApproxPF *extracts an apex-path pair* $\mathcal{AP}$ *from* $Open$ *on Line 19 and then calls the* IsDominated *function for* $\mathcal{AP}$ *on Line 23. When the* IsDominated *function reaches Line 2 of Algorithm 8, there exists a solution in* $Sols$ *whose cost* ε_{curr}*-dominates the* $\mathbf{f}$*-value of* $\mathcal{AP}$ *iff there exists a truncated cost in* $\mathbf{C}^{tr}_{sol}$ *that* ε_{curr}*-dominates the truncated* $\mathbf{f}$*-value of* $\mathcal{AP}$.

Proof. From Lemma 4.5, there exists a solution in $Sols$ whose cost $\varepsilon_{\text{curr}}$-dominates the $\mathbf{f}$-value of apex-path pair $\mathcal{AP}$ on Line 2 of the IsDominated function (Algorithm 8) if there exists a truncated cost in $\mathbf{C}^{\text{tr}}_{\text{sol}}$ that $\varepsilon_{\text{curr}}$-dominates the truncated $\mathbf{f}$-value of apex-path pair $\mathcal{AP}$.

Assume that there exists a solution π_{sol} in $Sols$ whose cost $\varepsilon_{\text{curr}}$-dominates the $\mathbf{f}$-value of apex-path pair $\mathcal{AP}$. Because the cost of π_{sol} $\varepsilon_{\text{curr}}$-dominates the $\mathbf{f}$-value of $\mathcal{AP}$, $c_1(\pi_{\text{sol}}) \leq (1 + \varepsilon_{\text{curr}})f_1(\mathcal{AP})$. We can distinguish the following two cases for π_{sol}:

1. π_{sol} is a solution found in a previous run of FindApproxPF. It is then added to $Sols'$ on Line 14 in the current run of FindApproxPF. Because $c_1(\pi_{\text{sol}}) \leq (1 + \varepsilon_{\text{curr}})f_1(\mathcal{AP})$, the truncated cost of π_{sol} has already been added to $\mathbf{C}^{\text{tr}}_{\text{sol}}$ on Line 21 before FindApproxPF calls the IsDominated function for $\mathcal{AP}$.

2. π_{sol} is a solution found in the current run of FindApproxPF. The truncated cost of π_{sol} was added to $\mathbf{C}^{\text{tr}}_{\text{sol}}$ on Line 28 when FindApproxPF expanded the apex-path pair corresponding to π_{sol}.

In both cases, FindApproxPF has added $Tr(\mathbf{c}(\pi_{\text{sol}}))$ to $\mathbf{C}^{\text{tr}}_{\text{sol}}$ before it calls the IsDominated function for $\mathcal{AP}$. When the IsDominated function reaches Line 2 of Algorithm 8, there must exist some vector $\mathbf{x}$ in $\mathbf{C}^{\text{tr}}_{\text{sol}}$ that weakly dominates $Tr(\mathbf{c}(\pi_{\text{sol}}))$, which, in turn, $\varepsilon_{\text{curr}}$-dominates the truncated $\mathbf{f}$-value of apex-path pair $\mathcal{AP}$. Therefore, $\mathbf{x}$ $\varepsilon_{\text{curr}}$-dominates the truncated $\mathbf{f}$-value of apex-path pair $\mathcal{AP}$. □

Lemma 4.10. FindApproxPF *does not expand an apex-path pair* $\mathcal{AP}$ *if there exists a solution* π_{sol} *with* $\mathbf{c}(\pi_{sol}) \prec \mathbf{f}(\mathcal{AP})$.

Proof. Consider the case where FindApproxPF extracts an apex-path pair $\mathcal{AP}$ from $Open$ on Line 19 and there exists a solution π_{sol} with $\mathbf{c}(\pi_{\text{sol}}) \prec \mathbf{f}(\mathcal{AP})$. We prove the lemma by showing that FindApproxPF will not expand $\mathcal{AP}$. From Lemma 4.8, we distinguish two cases:

1. Right before FindApproxPF extracts $\mathcal{AP}$ from $Open$ on Line 19, there existed an apex-path pair $\mathcal{AP}'$ in $Open$ with $\mathbf{g}(\mathcal{AP}') \preceq \mathbf{c}(\pi^{(j)}_{\text{sol}})$ and $s(\mathcal{AP}') = s_j$ for some j. $\mathbf{f}(\mathcal{AP}') = \mathbf{g}(\mathcal{AP}') + \mathbf{h}(s(\mathcal{AP}')) \preceq \mathbf{c}(\pi_{\text{sol}})$ because $\mathbf{g}(\mathcal{AP}') \preceq \mathbf{c}(\pi^{(j)}_{\text{sol}})$ and $\mathbf{h}$ is consistent. We have $\mathbf{f}(\mathcal{AP}') \preceq \mathbf{c}(\pi_{\text{sol}}) \prec \mathbf{f}(\mathcal{AP})$, which contradicts that FindApproxPF extracts the apex-path pair with the lexicographically smallest $\mathbf{f}$-value. Therefore, this case cannot happen.

2. There exists a solution π'_{sol} in $Sols$ that $\varepsilon_{\text{curr}}$-dominates π_{sol}. Then, $\mathbf{c}(\pi'_{\text{sol}})$ $\varepsilon_{\text{curr}}$-dominates $\mathbf{c}(\pi_{\text{sol}})$ and hence $\varepsilon_{\text{curr}}$-dominates $\mathbf{f}(\mathcal{AP})$. Therefore, FindApproxPF prunes $\mathcal{AP}$ on Line 24 because the condition on Line 2 of IsDominated (Algorithm 8) holds, according to Lemma 4.9. □

Theorem 4.1. FindApproxPF *terminates in finite time. For any solution* π_{sol}*, there exists, when* FindApproxPF *terminates, a solution in* $Sols$ *that* ε_{curr}*-dominates* π_{sol}.

Proof. Consider any solution π_{sol} and any expanded apex-path pair $\mathcal{AP} = \langle \mathbf{A}, \pi \rangle$. Because FindApproxPF generates only $\varepsilon_{\text{curr}}$-bounded apex-path pairs (Lemma 4.3), $\mathbf{f}(\pi)$ must $\varepsilon_{\text{curr}}$-dominate $\mathbf{f}(\mathcal{AP})$ (that is, $c_i(\pi) + h_i(s(\mathcal{AP})) \leq (1+\varepsilon_{\text{curr}})f_i(\mathcal{AP})$ for all i). $c_i(\pi) + h_i(s(\mathcal{AP})) \leq (1+\varepsilon_{\text{curr}})c_i(\pi_{\text{sol}})$ must hold for some i, because, otherwise, $(1+\varepsilon_{\text{curr}})c_i(\pi_{\text{sol}}) < c_i(\pi) + h_i(s(\mathcal{AP})) \leq (1+\varepsilon_{\text{curr}})f_i(\mathcal{AP})$ holds for all i and thus $\mathbf{c}(\pi_{\text{sol}}) \prec \mathbf{f}(\mathcal{AP})$, which contradicts that $\mathbf{f}(\mathcal{AP})$ is not dominated by $\mathbf{c}(\pi_{\text{sol}})$ from Lemma 4.10. Because heuristic $\mathbf{h}$ is non-negative, $c_i(\pi) \leq (1 + \varepsilon_{\text{curr}})c_i(\pi_{\text{sol}})$ must hold for some i for the representative path π of any expanded apex-path pair. Because the graph is finite and has positive edge costs, one can extend a path only a finite number of times before the resulting path π no longer satisfies $c_i(\pi) \leq (1 + \varepsilon_{\text{curr}})c_i(\pi_{\text{sol}})$ for all i. Thus, there are only a finite number of representative paths that FindApproxPF can expand (and generate). Consequently, FindApproxPF terminates in finite time.

Consider the beginning of the last iteration of FindApproxPF before it terminates, where $Open$ becomes empty. From Lemma 4.8, for any solution π_{sol}, there exists a solution in $Sols$ that $\varepsilon_{\text{curr}}$-dominates π_{sol}. Thus, the theorem holds. □

Theorem 4.2. *A-A*pex terminates when ε_{curr} becomes sufficiently small, and $Sols$ is then a cost-unique Pareto frontier.*

Proof. As we have shown in the proof of Theorem 4.1, A-A*pex can generate only a finite number of representative paths. Thus, there exists some ε' such that, when $\varepsilon_{\text{curr}}$ becomes smaller than ε', two apex-path pairs can be merged only if one of the representative paths weakly dominates the other, and only this representative path can be chosen as the new representative path. Therefore, the apex of any apex-path pair generated by A-A*pex must be equal to the cost of its representative path. The AddToOpen function cannot reach Line 21 because the new representative path π_{new} always weakly dominates the other representative path π_{pruned} on Line 20.

Assume that IsDominated reaches Line 9 with $\varepsilon_{\text{curr}} < \varepsilon'$. Path π mentioned on Line 9 is the representative path of apex-path pair $\mathcal{AP}$. Because $\varepsilon_{\text{curr}} < \varepsilon'$, we have $\mathbf{g}(\mathcal{AP}) = \mathbf{c}(\pi)$. Because $\varepsilon_{\text{curr}} < \varepsilon'$, the truncated cost of path π' mentioned on Line 9 is equal to the truncated $\mathbf{g}$-value of apex-path pair $\mathcal{AP}'$ whose representative path is π'. Thus, the truncated cost of path π' is also equal to cost vector $\mathbf{x}$ mentioned on Line 7. From the condition on Line 7, $\mathbf{x}$ weakly dominates $Tr(\mathbf{g}(\mathcal{AP}))$. Therefore,

$$Tr(\mathbf{c}(\pi')) = \mathbf{x} \preceq Tr(\mathbf{g}(\mathcal{AP})) = Tr(\mathbf{c}(\pi)). \tag{4.3}$$

From Lemma 4.4, $\mathrm{FindApproxPF}$ extracts apex-path pairs from Open with monotonically non-decreasing f_1-values. Because $\mathcal{AP}'$ is extracted from Open before $\mathcal{AP}$, we have $f_1(\mathcal{AP}') \leq f_1(\mathcal{AP})$. Also, because $s(\mathcal{AP}) = s(\mathcal{AP}')$, we have $g_1(\mathcal{AP}') \leq g_1(\mathcal{AP})$. For the same reason as for $\mathbf{g}(\mathcal{AP}) = \mathbf{c}(\pi)$, we have $\mathbf{g}(\mathcal{AP}') = \mathbf{c}(\pi')$. Therefore, we have $c_1(\pi') \leq c_1(\pi)$. Combining this with Eq. 4.3 yields $\mathbf{c}(\pi') \preceq \mathbf{c}(\pi)$. Because of the condition on Line 9, $\mathrm{IsDominated}$ cannot reach Line 10.

When $\varepsilon_{\text{curr}} < \varepsilon'$, because $\mathrm{FindApproxPF}$ can generate only a finite number of representative paths and because the $\mathbf{g}$-value of an apex-path pair is always equal to the cost of its representative path, the number of different $\mathbf{f}$-values of generated apex-path pairs is finite. Therefore, there exists some $\varepsilon'' \leq \varepsilon'$ such that, when $\varepsilon_{\text{curr}}$ becomes smaller than ε'', a solution $\varepsilon_{\text{curr}}$-dominates the $\mathbf{f}$-value of an apex-path pair only if it weakly dominates this $\mathbf{f}$-value. When $\mathrm{IsDominated}$ reaches Line 3 with $\varepsilon_{\text{curr}} < \varepsilon''$, from Lemma 4.5, $\mathbf{c}(\pi')$ $\varepsilon_{\text{curr}}$-dominates $\mathbf{f}(\mathcal{AP})$, where $\mathcal{AP} = \langle \mathbf{A}, \pi \rangle$ is the input apex-path pair. Because $\varepsilon_{\text{curr}} < \varepsilon''$, we have $\mathbf{f}(\mathcal{AP}) = \mathbf{f}(\pi)$ and $\mathbf{c}(\pi') \preceq \mathbf{f}(\mathcal{AP})$. Therefore, $\mathbf{c}(\pi') \preceq \mathbf{f}(\pi)$. Because of the condition on Line 4, $\mathrm{IsDominated}$ cannot reach Line 5.

Put together, when $\varepsilon_{\text{curr}}$ becomes sufficiently small (that is, smaller than ε''), $\mathrm{FindApproxPF}$ cannot reach Line 5, 10, or 21 in Algorithm 8. Therefore, Pruned stays empty. When $\mathrm{FindApproxPF}$ returns, A-A*pex reaches Line 11 in Algorithm 7 and then terminates.

When $Pruned$ becomes empty, from Lemma 4.2, for any solution π_{sol}, there exists a solution in $Sols$ that weakly dominates π_{sol}. Because FindApproxPF adds a solution to $Sols$ only if it is not $\varepsilon_{\text{curr}}$-dominated (and hence not weakly dominated) by any solution in $Sols$ and removes all solutions that are weakly dominated by it from $Sols$, two different solutions in $Sols$ do not weakly dominate each other. Thus, $Sols$ is a Pareto frontier. $\square$

4.4 Experimental Evaluation

In our experimental evaluation, we first compare different variants of A-A*pex (Section 4.4.2). We then compare A-A*pex with state-of-the-art multi-objective search algorithms (Section 4.4.3).

We use the NY road network from the 9th DIMACS Implementation Challenge, which has 264K states and 730K edges. The NY road network has two objectives available in the benchmark, namely travel distance (d) and travel time (t). We use the economic cost (m) [54], the number of edges (l) [47], and a random integer from 1 to 100 (r) [36] as the third, fourth, and fifth objectives, respectively. We use the same 100 problem instances used by Sedeño-Noda and Colebrook [65] and Ahmadi et al. [2].

We implement all algorithms in C++[2] and run all experiments on a MacBook with an M1 Pro chip and 32GB of memory. The runtime limit for solving each problem instance is five minutes. For A-A*pex, the sequence of ε-values output by getNextEps begins with 0.1 and is divided by η after every iteration, where η is a predetermined parameter. One can develop more complicated schemes for decreasing the ε-value, which we leave to future work.

[2]https://github.com/HanZhang39/MultiObjectiveSearch

4.4.1 Metrics

To evaluate the quality of solutions that a multi-objective search algorithm computes during its search, we introduce a metric called the *approximation error*, which we will formally define shortly. The approximation error of a set of solutions measures how “well” this set approximates the Pareto frontier. The smaller the approximation error, the better.

We define the *dominance factor* of a solution π over another solution π' as

$$\mathrm{DF}(\pi, \pi') = \max\left(\max_{i=1,2\ldots N}\left\{\frac{c_i(\pi)}{c_i(\pi')} - 1\right\}, 0\right),$$

which measures how “well” π approximates π'. $\mathrm{DF}(\pi, \pi')$ is the smallest ε-value that satisfies $\pi \preceq_\varepsilon \pi'$. For a set of solutions Π, we define the *approximation error* of Π over a solution π' as

$$e(\Pi, \pi') = \min_{\pi \in \Pi} \mathrm{DF}(\pi, \pi').$$

Roughly speaking, we find a path π in Π that approximates π' the best and compute the corresponding dominance factor. We have $e(\Pi, \pi') = 0$ iff $\exists \pi \in \Pi, \pi \preceq \pi'$. Let Π^* denote the Pareto frontier. We define the *approximation error* of a set of solutions Π as

$$e(\Pi) = \max_{\pi \in \Pi^*} e(\Pi, \pi). \tag{4.4}$$

$e(\Pi)$ is the smallest ε-value for which Π is an ε-approximate frontier.

In our experiments, there are problem instances where no algorithm finds the entire Pareto frontier within the runtime limit. When computing the approximation error using Eq. 4.4 in these cases, we use the undominated set of the solutions found by all algorithms as a substitute for Π^*.

We are interested in finding a set of solutions with a small approximation error. More specifically, we focus on the anytime behavior of a search algorithm, that is, its ability to quickly reduce the approximation error over time.

4.4.2 Comparing Different Variants of A-A*pex

We compare different variants of A-A*pex on the first 50 problem instances on the NY road network with three objectives (m-t-d). These variants of A-A*pex are:

1. A-A*pex-reuse always reuses its search effort and is our baseline variant of A-A*pex.
2. A-A*pex-reuse-enh always reuses its search effort and also uses the enhanced dominance checks.
3. A-A*pex-restart always restarts its search from scratch.
4. A-A*pex-hybrid initially restarts its search from scratch and reuses its search effort in later iterations. It also uses enhanced dominance checks once it reuses its search effort.

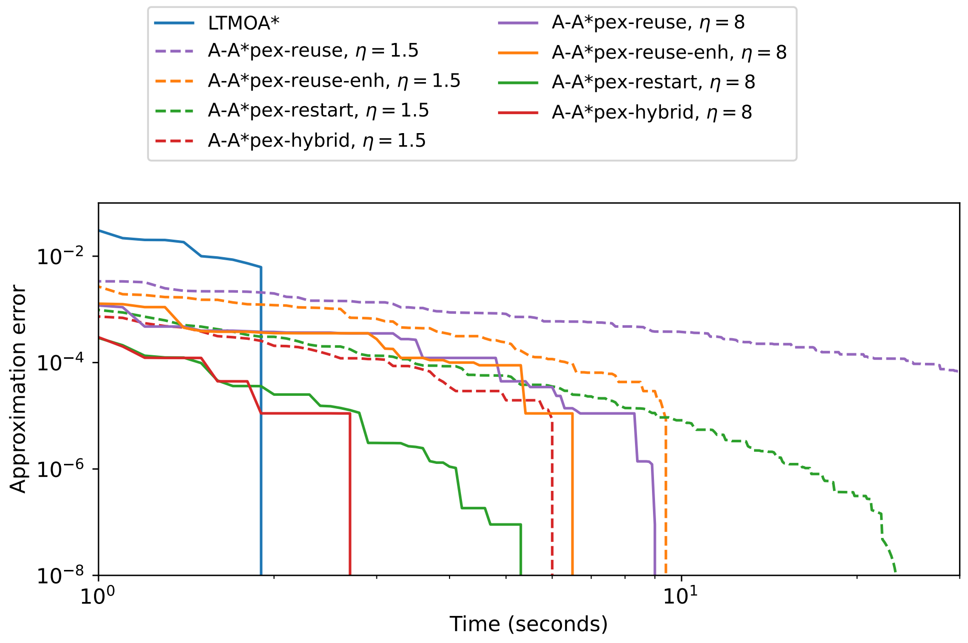
We evaluate each variant of A-A*pex for $\eta \in \{1.5, 2, 4, 8\}$. We also evaluate LTMOA*.

Table 4.2 shows the numbers of solved problem instances (i.e., the numbers of problem instances for which an algorithm finds the entire Pareto frontier within the runtime limit), average runtimes (in seconds), and average numbers of expanded nodes for all algorithms. All averages are calculated over those problem instances that all algorithms solve. LTMOA* has the largest

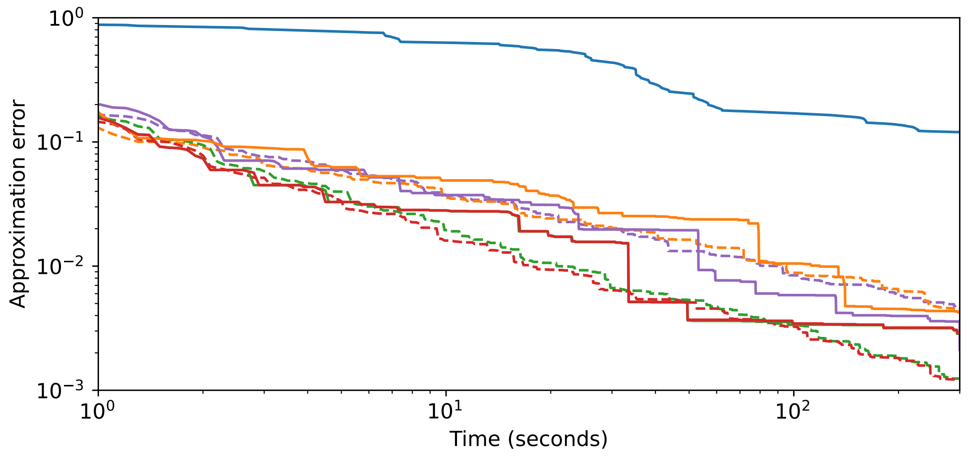
	#solved	t_{avg}	#exp
LTMOA*	47	0.31	331K
A-A*pex-reuse $\eta = 1.5$	35	6.31	2,636K
A-A*pex-reuse-enh, $\eta = 1.5$	**40**	1.18	**423K**
A-A*pex-restart, $\eta = 1.5$	38	3.61	5,539K
A-A*pex-hybrid, $\eta = 1.5$	**40**	**0.84**	1,208K
A-A*pex-reuse $\eta = 2$	36	3.51	1,779K
A-A*pex-reuse-enh, $\eta = 2$	40	1.04	**396K**
A-A*pex-restart, $\eta = 2$	39	2.00	3,417K
A-A*pex-hybrid, $\eta = 2$	**41**	**0.65**	878K
A-A*pex-reuse $\eta = 4$	38	1.75	1,121K
A-A*pex-reuse-enh, $\eta = 4$	40	0.78	**378K**
A-A*pex-restart, $\eta = 4$	40	1.11	1,866K
A-A*pex-hybrid, $\eta = 4$	**42**	**0.59**	753K
A-A*pex-reuse $\eta = 8$	39	1.19	908K
A-A*pex-reuse-enh, $\eta = 8$	40	0.81	**367K**
A-A*pex-restart, $\eta = 8$	**42**	0.78	1,307K
A-A*pex-hybrid, $\eta = 8$	**42**	**0.41**	515K

Table 4.2: Numbers of solved problem instances, average runtime, and average numbers of expanded nodes for different algorithms on 50 problem instances on the NY road network with three objectives.

number of solved problem instances and the smallest average runtime and number of expanded nodes, which is unsurprising since LTMOA* does not have the overhead of running a sequence of searches that A-A*pex has. The average runtime of each variant of A-A*pex decreases as η increases because larger values of η result in fewer iterations of A-A*pex. Adding enhanced dominance checks decreases the average runtime of A-A*pex-reuse and results in the smallest numbers of node expansions of all A-A*pex variants. With a small η-value, e.g., 1.5 and 2, A-A*pex-restart has a larger runtime than A-A*pex-reuse-enh because restarting from scratch with small ε-values for the later iterations of A-A*pex is time-consuming. For all η-values, A-A*pex-hybrid outperforms the other three A-A*pex variants in terms of the number of solved problem instances and average runtime.



(a) The 35 problem instances solved by all algorithms.



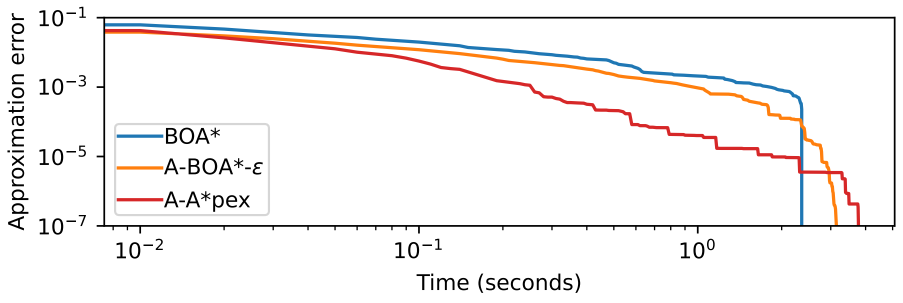
(b) The other 15 problem instances.

Figure 4.2: Approximation error as a function of the runtime of different algorithms on the NY road network with three objectives.

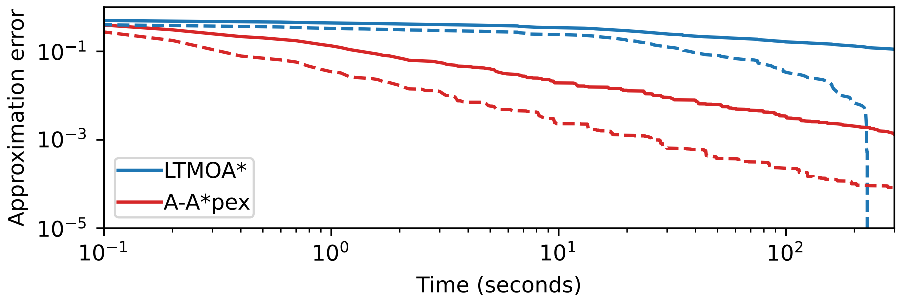
Figure 4.2 shows the approximation error as a function of the runtime for LTMOA* and all A-A*pex variants with $\eta = 1.5$ and $\eta = 8$. We use only two η-values to keep the figure clean. We divide the problem instances into two groups, namely the problem instances solved by all algorithms (Figure 4.2a) and the other problem instances (Figure 4.2b). The approximation error of each algorithm is averaged over all problem instances in a group. A-A*pex-reuse and A-A*pex-reuse-enh have larger approximation errors than A-A*pex-restart and A-A*pex-hybrid in the beginning of the search for both η-values, which shows that restarting the search from scratch is more efficient in the earlier iterations. In Figure 4.2a, the approximation errors of A-A*pex-reuse, A-A*pex-reuse-enh, and A-A*pex-hybrid quickly drop in the later iterations, which shows that reusing search effort is more efficient in the later iterations. Therefore, by mixing these two techniques, A-A*pex-hybrid often finds the Pareto frontier faster than the other variants of A-A*pex. Figure 4.2b shows that, for all variants of A-A*pex, using a smaller η-value of 1.5 (dashed lines) leads to more frequent updates than using a η-value of 8 (solid lines) and hence results in a better anytime behavior for difficult problem instances. For example, in Figure 4.2b, if we stop A-A*pex-hybrid with $\eta = 1.5$ and $\eta = 8$ at any point in time, the solution set found by A-A*pex-hybrid with $\eta = 1.5$ is much more likely to have a smaller approximation error. In Figure 4.2b, all variants of A-A*pex have a smaller approximation error than LTMOA* for the entire five minutes of runtime for both η-values. Although we expect LTMOA* to find Pareto frontiers faster than A-A*pex if the runtime limit is sufficiently high, all variants of A-A*pex often compute solution sets with approximation errors smaller than 0.01 faster than LTMOA*.

4.4.3 Comparing with the State of the Art

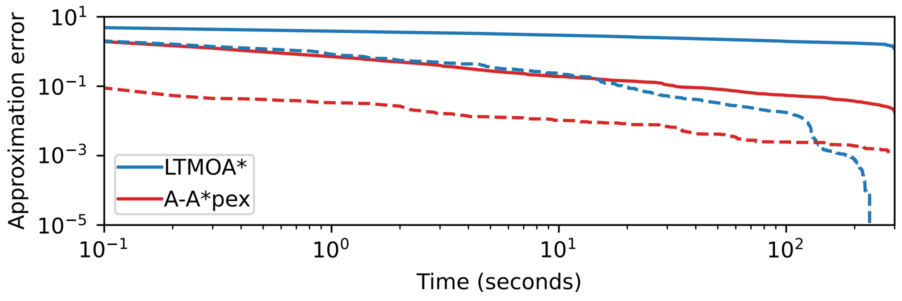
We compare the hybrid variant of A-A*pex with $\eta = 4$ with the state-of-the-art multi-objective search algorithms BOA* [37] and A-BOA*-ε [86] on problem instances with two objectives and LTMOA* on problem instances with more than two objectives.



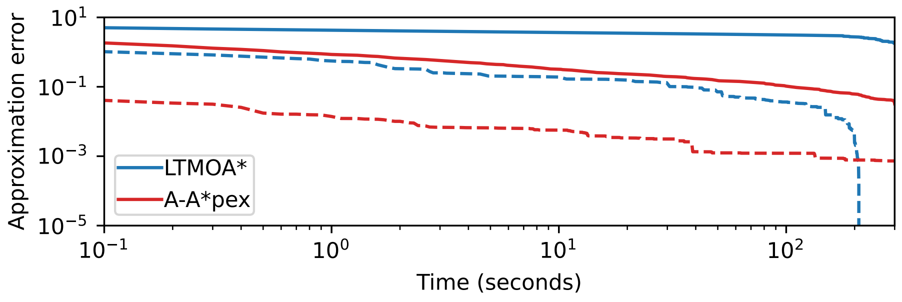
(a) NY, t-d



(b) NY, m-t-d



(c) NY, l-m-t-d



(d) NY, l-m-t-d-r

Figure 4.3: Anytime behaviors of different algorithms on problem instances with different numbers of objectives. Each plot shows the approximation error as a function of the runtime for each algorithm over all 100 problem instances (solid line) and over only those problem instances solved by at least one algorithm (dashed line).

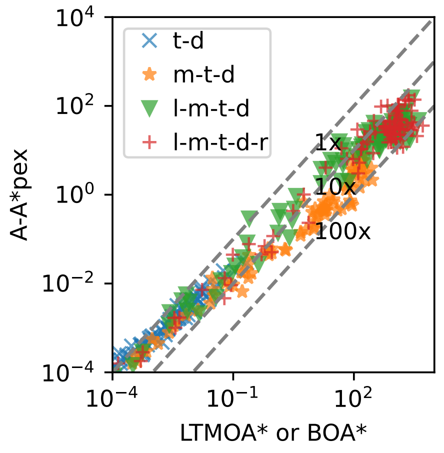


Figure 4.4: AUCs for LTMOA* (or BOA* for two objectives) and A-A*pex on all problem instances.

Figure 4.3 shows the results for different objectives. We use solid lines for all problem instances and dashed lines for those problem instances whose entire Pareto frontiers are computed within the runtime limit. Figure 4.3a contains only solid lines because the entire Pareto frontier for every problem instance is computed within the runtime limit. In all cases, A-A*pex reduces the approximation error faster than the other algorithms in the beginning of the search. Because LTMOA* and BOA* compute solutions in lexicographically increasing order of their costs, they can exactly "cover" part of the Pareto frontier while completely missing the rest during most of their searches, which explains their high approximation errors at the beginning. This behavior is undesirable since it results in high approximation errors for small runtime limits. When a sufficient runtime is provided, LTMOA* and BOA* find the Pareto frontier faster than A-A*pex and hence have smaller approximation errors than A-A*pex. However, this happens only after the approximation error becomes smaller than 0.01, even smaller than 0.001 in many cases.

We also compute the *Area Under the Curve* (AUC) of the approximation error for each problem instance P and algorithm A, formally defined as $\text{AUC}_A(P) = \int_0^{t^{\text{limit}}} e(t)$, where t^{limit} is the runtime limit of five minutes and e is the approximation error as a function of the runtime. We compare

A-A*pex with LTMOA* (or BOA* for two objectives) as the baseline. Figure 4.4 shows the results. The numbers along the dashed lines denote how many times the AUC of A-A*pex is smaller than that of the baseline. For problem instances that are more difficult to solve (represented by points in the top-right corners), A-A*pex always has smaller AUCs than the baseline. For problem instances with three or more objectives, A-A*pex has up to $100\times$ smaller AUCs than LTMOA*.

4.5 Summary

In this chapter, we introduced A-A*pex, an anytime approximate multi-objective search algorithm that builds upon A*pex. A-A*pex runs A*pex repeatedly to compute better and better approximate Pareto frontiers while time allows. In each iteration, A-A*pex can either reuse its previous search effort or restart its search from scratch. Our experimental results showed that a hybrid variant of A-A*pex, which restarts its search from scratch at first and then reuses its search effort in later iterations, often results in the best performance. Our experimental results also validated the hypothesis that anytime multi-objective search algorithms can find a set of solutions that collectively approximates the Pareto frontier better than the one found by existing multi-objective search algorithms when the given time is insufficient for finding the entire Pareto frontier.

Chapter 5

Speeding up Multi-Objective Search via Contraction Hierarchies (CHs)

So far, we have considered multi-objective search algorithms only in the context of solving a single problem instance. However, for many real-world applications, one has to solve multiple problem instances on the same environment graph. For example, for a route-planning application, one might have to find the shortest paths for different start and goal locations on the same road network. Such a setting is called a *multi-query* setting, and each problem instance is called a *query*. In multi-query single-objective search, it is common practice to speed up the search via *preprocessing* techniques: In a single preprocessing phase, a preprocessing algorithm processes a given graph[1] and builds auxiliary data for query-time use. In the query phase, different problem instances are given to a query algorithm, which often uses the auxiliary data to solve the given problem instances orders of magnitude faster than solving them directly.

This chapter is based on [83].

[1]The algorithms proposed in this chapter require that the graph is given in an explicit form (that is, the sets of states and edges are explicitly given). However, the algorithms proposed in the rest of this dissertation do not have this restriction.

A well-studied preprocessing technique for single-objective search is Contraction Hierarchies (CHs) [28]. A CH is a hierarchical graph that assigns a level number to each state in the input graph and adds additional edges (known as *shortcuts*) to the input graph so that the shortest path from a given start state to a given goal state can be found by searching through the space of only up-down paths (paths that traverse states with first increasing and then decreasing level numbers).

In this chapter, we generalize CHs to multi-objective search. We leverage existing multi-objective search algorithms, namely, LTMOA* and BOA*, in both the preprocessing and query phases. The resulting CHs retain the property that the Pareto frontier can be computed by considering only up-down paths.

Unlike a CH for single-objective search, which needs to maintain only one minimum-cost edge from a source state to a target state, a CH for multi-objective search might have to maintain several (parallel) edges (with different cost vectors). Maintaining a large number of parallel edges can slow down both the preprocessing and query algorithms. Hence, to speed up the preprocessing algorithm, we leverage LTMOA* and BOA* to determine which edges in a given set of parallel edges need to be added to the CH as shortcuts. We call this approach the batched approach because it checks the given set of parallel edges in one batch. Compared to our basic approach, which checks each parallel edge individually, the batched approach reduces the preprocessing time substantially. To speed up the query algorithm, we propose a (general) partial-expansion technique, which reduces the query time by reducing the number of unnecessarily generated search nodes. Our experimental results demonstrate the scalability of our CH-based approach to large road networks and orders-of-magnitude speed-ups in the query phase when all techniques are combined.

This chapter is organized as follows: We begin with describing the background materials for our work in Section 5.1. Then, we describe CHs for multi-objective search in Section 5.2 and prove their correctness in Section 5.3. We end the chapter with our experimental results in Section 5.4 and a summary in Section 5.5.

5.1 Background

We first describe CHs for single-objective search (Section 5.1.1). We then review existing work that uses CHs in graphs with two costs but for tasks different from computing the Pareto frontier (Section 5.1.2) and existing work on other preprocessing techniques (Section 5.1.3).

5.1.1 CHs for Single-Objective Search

In this section, we describe CHs for single-objective search. Since we consider only single-objective search in this section, we use a scalar $c(e)$ to denote the cost of edge e.

Given a single-objective search graph $G = \langle S, E \rangle$, a CH is computed by performing *contractions* on the states in S one by one according to a given state ordering. Contracting a state s removes it and its incident edges (that is, both its in- and out-edges) from the graph while preserving the minimum path cost between any pair of states in the remaining graph. To do so, before removing s and its incident edges, the preprocessing algorithm iterates through every pair of in-edge e and out-edge e' of s. It runs a so-called *witness search* to determine if there is a path, also called a *witness*, from $src(e)$ to $tar(e')$ in the current graph that does not traverse state s and whose cost is smaller than or equal to $c(e) + c(e')$. The witness search can be implemented with any shortest-path algorithm, such as Dijkstra's algorithm. If the algorithm does not find a witness, a new edge $\langle src(e), tar(e'), c(e) + c(e') \rangle$ that *bridges* edges e and e', called a *shortcut*, is added to the graph to preserve the minimum path cost from $src(e)$ to $tar(e')$. Generating a CH does not require contracting all states. Let L denote the number of states to contract, determined by a user. After the first L states have been contracted, a CH $G_{\text{CH}} = \langle S, E_{\text{CH}} \rangle$ is created. The state set S is the one of the input graph G, and the edge set E_{CH} consists of the edges in E (including the ones that were removed during contraction) and all shortcuts. In case there are parallel edges, only the minimum-cost one is kept. The i-th contracted state s is assigned a *level number* of $lvl(s) = i$, and all uncontracted states (also called the *core* of the CH) are assigned level numbers of $L + 1$.

An edge from state u to state v is an *upward edge* (resp. a *downward edge*) iff $lvl(u) \leq lvl(v)$ (resp. $lvl(u) > lvl(v)$). A path is an *upward path* (resp. a *downward path*) iff it consists of only upward edges (resp. downward edges). A path $\pi = [e_1, e_2 \dots e_\ell]$ is an *up-down path* iff there exists a j such that all edges e_i with $i \leq j$ are upward edges and all edges e_i with $i > j$ are downward edges. The following theorem is rephrased from Lemma 1 in Geisberger et al. [28].

Theorem 5.1. *For any pair of states u and v, there exists an up-down path from u to v in G_{CH} with the minimum-path cost from u to v in the input graph G.*

Given a query, a minimum-cost up-down solution on G_{CH} can be computed efficiently using a modified bidirectional Dijkstra's algorithm [28] or a modified A* algorithm [34]. Then, the up-down solution can be unpacked into a minimum-cost solution on G by recursively replacing each shortcut with the two edges that it bridges.

Different CHs can be obtained from the same input graph using different state orderings for contraction. The state ordering plays an important role for both the preprocessing and query times of the resulting CH and is usually determined with heuristics that take into account the number of shortcuts to add if a state is contracted and the number of incident edges of that state [28].

Example 8. *Figure 5.1 shows an example of CH for single-objective search. The input graph is the graph in Figure 2.1 but with only the first cost of each edge. States are contracted in the order of* $[s_2, s_5, s_1, s_4, s_{start}, s_{goal}, s_3]$*, and* L *is* 7*:*

- *State* s_2 *is contracted. We do not need to add a shortcut for path* $[s_{start}, s_2, s_3]$ *with cost* 4 *because of path* $[s_{start}, s_1, s_3]$ *with cost* 3*. Similarly, we do not need to add a shortcut for path* $[s_1, s_2, s_3]$ *with cost* 2 *because of path* $[s_1, s_3]$ *with cost* 2*. Therefore, no shortcut is created.*

- *State* s_5 *is contracted, and a shortcut from* s_3 *to* s_{goal} *with cost* 3 *is added to the graph because there is no other path from* s_3 *to* s_{goal} *with a cost smaller than or equal to the cost* 3 *of path* $[s_3, s_5, s_{goal}]$*.*

- *State* s_1 *is contracted, and a shortcut from* s_{start} *to* s_3 *with cost* 3 *is added to the graph because there is no other path from* s_{start} *to* s_3 *with a cost smaller than or equal to the cost* 3 *of path* $[s_{start}, s_1, s_3]$*.*

- *States* s_4*,* s_{start}*,* s_{goal}*, and* s_3 *are contracted in order. No shortcuts are created.*

Figure 5.1h shows the resulting CH. The minimum-cost path between any two states can be found by searching only up-down paths. For example, the only up-down path from s_{start} *to* s_{goal} *is path* $[s_{start}, s_3, s_{goal}]$ *with cost* 6*. After unpacking this path (that is, replacing the shortcuts from* s_{start} *to* s_3 *and from* s_3 *to* s_{goal} *with paths* $[s_{start}, s_1, s_3]$ *and* $[s_3, s_5, s_{goal}]$*, respectively), we obtain the minimum-cost path* $[s_{start}, s_1, s_3, s_5, s_{goal}]$ *from* s_{start} *to* s_{goal} *in the input graph. Different state*

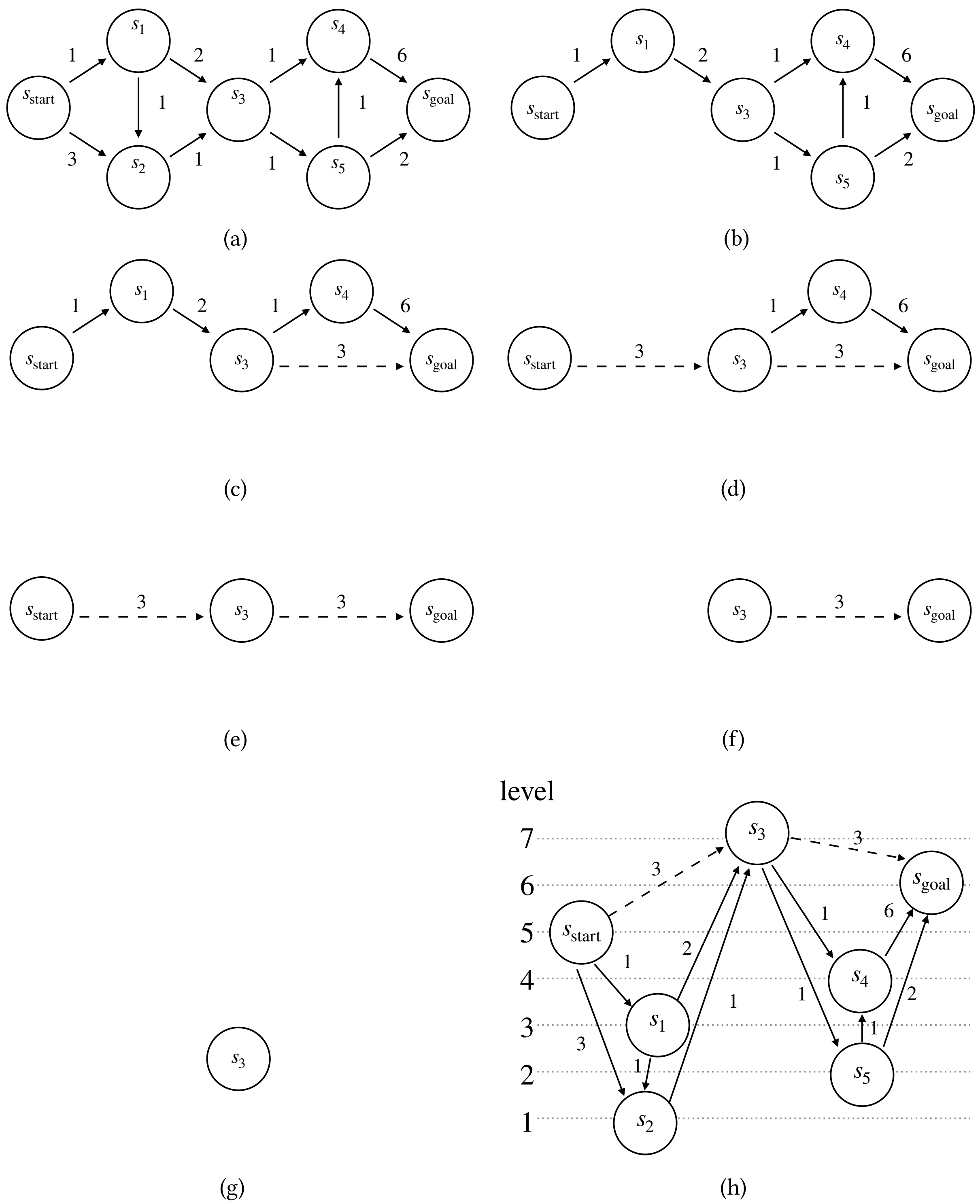


Figure 5.1: An example CH for single-objective search. (a) shows the input graph, which is the graph in Figure 2.1 but with only the first cost of each edge. (b-g) show the steps for contracting the states in the order of $[s_2, s_5, s_1, s_4, s_{start}, s_{goal}, s_3]$. (h) shows the resulting CH. Dashed edges are the shortcuts added during contraction.

orderings for contraction can result in different CHs. For example, if we contract state s_1 first, we need to add a shortcut from s_{start} to s_2 with cost 2, which is not in the CH in Figure 5.1h.

5.1.2 CHs in Graphs with Multiple Costs

To the best of our knowledge, there are only a few existing works that use CHs on graphs with multiple costs. However, even they have not investigated the task of computing the entire Pareto frontier [69, 27, 25, 7]. Most of them focus only on two costs. Specifically, Storandt [69] proposed a CH-based approach for solving the Weight Constrained Shortest-Path (WCSP) problem. Its preprocessing algorithm computes shortcuts heuristically—which avoids the burden of computing the exact shortcuts—but can add unnecessary shortcuts. Both Geisberger, Kobitzsch, and Sanders [27] and Funke and Storandt [25] use different weighted combinations of the costs to map a multi-objective search problem to several single-objective search problems. However, the resulting CHs cannot be used to find all paths on the Pareto frontier, particularly those that do not minimize any weighted combination of the costs. Baum et al. [7] apply CHs to a constrained shortest-path problem that considers charging, recuperation, and the battery capacity of electric vehicles. The vehicle has a fixed battery capacity and can charge at stations. The problem objective is to minimize the total travel time (including the time for charging) while ensuring that the battery never gets empty.

5.1.3 Other Preprocessing Techniques

Preprocessing techniques have been used extensively in single-objective search. Examples other than CHs include true distance heuristics [70], embedding in Euclidean spaces [14], and sub-goal graphs [76]. Only a few of them have been generalized to multi-objective search.

One of the existing works on computing Pareto frontiers with preprocessing techniques is multi-criteria SHARC [17]. However, it has been demonstrated only on small road networks with less than 80,000 states and is not immediately scalable to larger graphs. This is partly because its preprocessing algorithm needs to compute the Pareto frontier from one state to every other state, which requires a large amount of memory for large road networks.

Our previous work [82] generalizes Differential Heuristics (DHs) [30, 70], a class of true distance heuristics for single-objective search, to bi-objective search. We call the resulting technique Bi-Objective Differential Heuristics (BO-DHs). During the preprocessing phase, a set of landmark states L are selected. Then, for each landmark state $\ell \in L$, the preprocessing algorithm computes the Pareto frontier from ℓ to every other state and stores these Pareto frontiers in a lookup table. During the query phase, a multi-value heuristic is computed using a generalized triangle inequality rule. A multi-value heuristic [29] uses a set of vectors to estimate the Pareto frontier from any given state to the goal state, which can be potentially more informed than using a single vector as the heuristic for each state. Our experimental results show that BO-DHs can reduce the number of node expansions and the runtime of NAMOA* by up to an order of magnitude. However, BO-DHs remain impractical for speeding up state-of-the-art bi-objective search algorithms because it is unclear how to integrate multi-value heuristics with the dimensionality reduction technique.

Because the preprocessing algorithm for BO-DHs also needs to compute the Pareto frontier from one state to every other state, BO-DHs are also not immediately scalable to larger graphs.

5.2 CHs for Multi-Objective Search

In this section, we introduce CHs for multi-objective search. We first describe the preprocessing algorithm in Section 5.2.1 and then the query algorithm in Section 5.2.2.

5.2.1 Preprocessing Algorithm

Like a CH in single-objective search, a CH in multi-objective search is built by contracting one state at a time in the input graph G until contracting L states. Contracting a state s removes it and its incident edges from G while preserving at least one Pareto frontier between any pair of states in the remaining graph. Each combination of an input edge and an output edge of s is a shortcut candidate. The preprocessing algorithm needs to determine whether to add a shortcut for each candidate. We propose two approaches for doing so: the *basic* approach and the *batched* approach. The basic approach runs a witness search for each shortcut candidate individually, and the batched approach groups the candidates for parallel shortcuts (that is, shortcuts from the same source state to the same target state) into a batch and uses a single witness search to test all of them at once to reduce the preprocessing time. In contrast to the witness search of Storandt [69], these two approaches use exact witness search algorithms and add fewer shortcuts to the CH.

Algorithm 9: The Basic Preprocessing Algorithm

```
Input  : A graph $G = \langle S, E \rangle$ and a number $L$ of states to contract
Output: A CH
1  $S_{\text{CH}} \leftarrow S$; $E_{\text{CH}} \leftarrow \{\}$
2  while $|S_{CH}| - |S| < L$ do
3      $s \leftarrow$ choose the next state to contract
4      for $e \in in(s)$ do
5          for $e' \in out(s)$ do
6              $u \leftarrow src(e)$; $v \leftarrow tar(e')$
7              if $\text{WitnessSearch}(G, u, v, \mathbf{c}(e) + \mathbf{c}(e')) = \mathit{false}$ then
8                  $\text{AddShortcut}(\langle u, v, \mathbf{c}(e) + \mathbf{c}(e') \rangle)$
9      add all edges incident on $s$ to $E_{\text{CH}}$
10     remove $s$ from $S$ and all edges incident on $s$ from $E$
11 add all remaining edges in $E$ to $E_{\text{CH}}$
12 return $G_{\text{CH}} = \langle S_{\text{CH}}, E_{\text{CH}} \rangle$;
13 Function $\text{WitnessSearch}(G, u, v, \mathbf{p})$:
14     $\pi \leftarrow$ a path from $u$ to $v$ whose cost dominates $\mathbf{p}$, or $\mathit{none}$ if no such path exists
15     return $\mathit{true}$ if $\pi = \mathit{none}$, otherwise $\mathit{false}$
16 Function $\text{AddShortcut}(e_{sc})$:
17     remove edges parallel to $e_{\text{sc}}$ whose costs are weakly dominated by $\mathbf{c}(e_{\text{sc}})$ from $E$
18     add $e_{\text{sc}}$ to $E$
```

5.2.1.1 Basic Approach

Algorithm 9 shows our basic approach to building a CH. When contracting a state s, for every pair of in-edge e and out-edge e' of s, it uses WitnessSearch to determine if there exists a path (witness) from $src(e)$ to $tar(e')$ whose cost dominates $\mathbf{c}(e) + \mathbf{c}(e')$ (Lines 4-8). We omit the pseudocode of WitnessSearch since it is based on LTMOA* (or BOA* for the bi-objective case) with $src(e)$ and $tar(e')$ as the start and goal states, respectively. We make the following modifications:

- **Termination**: Once a witness is found, WitnessSearch terminates and returns true. Otherwise, when Open becomes empty, WitnessSearch terminates and returns false.
- **Pruning**: WitnessSearch prunes any node n if $f_i(n) > c_i(e) + c_i(e')$ holds for any i since any solution found via such a node n cannot be a witness.

- **Heuristic computation**: Computing the perfect-distance heuristic is too time-consuming to do every time $\mathrm{WitnessSearch}$ is invoked. Thus, when we run Dijkstra's algorithm (which starts from $tar(e')$) to compute the heuristic, we terminate Dijkstra's algorithm once $src(e)$ is expanded. Subsequently, the heuristic value for any state s is set to the minimum path cost from $tar(e')$ to s on the reverse graph if s has been expanded or the minimum path cost from $tar(e')$ to $src(e)$ on the reverse graph otherwise. The resulting heuristic function is consistent.

A shortcut $e_{\text{sc}} := \langle src(e), tar(e'), \mathbf{c}(e) + \mathbf{c}(e')\rangle$ is added to the graph if $\mathrm{WitnessSearch}$ does not find a witness (Line 8). Additionally, we remove all those edges parallel to e_{sc} whose costs are weakly dominated by $\mathbf{c}(e_{\text{sc}})$ (Line 17) since such edges are not needed to preserve any Pareto frontier.

After contracting L states, Algorithm 9 returns a CH $G_{\text{CH}} = \langle S_{\text{CH}}, E_{\text{CH}}\rangle$ whose states S_{CH} consist of all states of the input graph and whose edges E_{CH} consist of all edges incident on the contracted states before they are removed from the input graph (Line 9) and all remaining edges (Line 11).

Example 9. *Figure 5.2 shows an example CH for the bi-objective graph in Figure 2.1. Different from our previous examples, we need to consider parallel edges in this example. Therefore, we refer to a path by its sequence of edges to avoid ambiguity. Each edge e is denoted by a tuple $\langle src(e), tar(e), \mathbf{c}(e)\rangle$. As in Example 8, states are contracted in the order of $[s_2, s_5, s_1, s_4, s_{start}, s_{goal}, s_3]$, and L is 7:*

- *State s_2 is contracted. A shortcut from s_1 to s_3 with cost $(2, 2)$ is added to the graph because there is no other path from s_1 to s_3 that dominates path $[\langle s_1, s_2, (1, 1)\rangle, \langle s_2, s_3, (1, 1)\rangle]$. The edge from s_1 to s_3 with cost $(2, 3)$ is removed because its cost is dominated by $(2, 2)$. A shortcut*

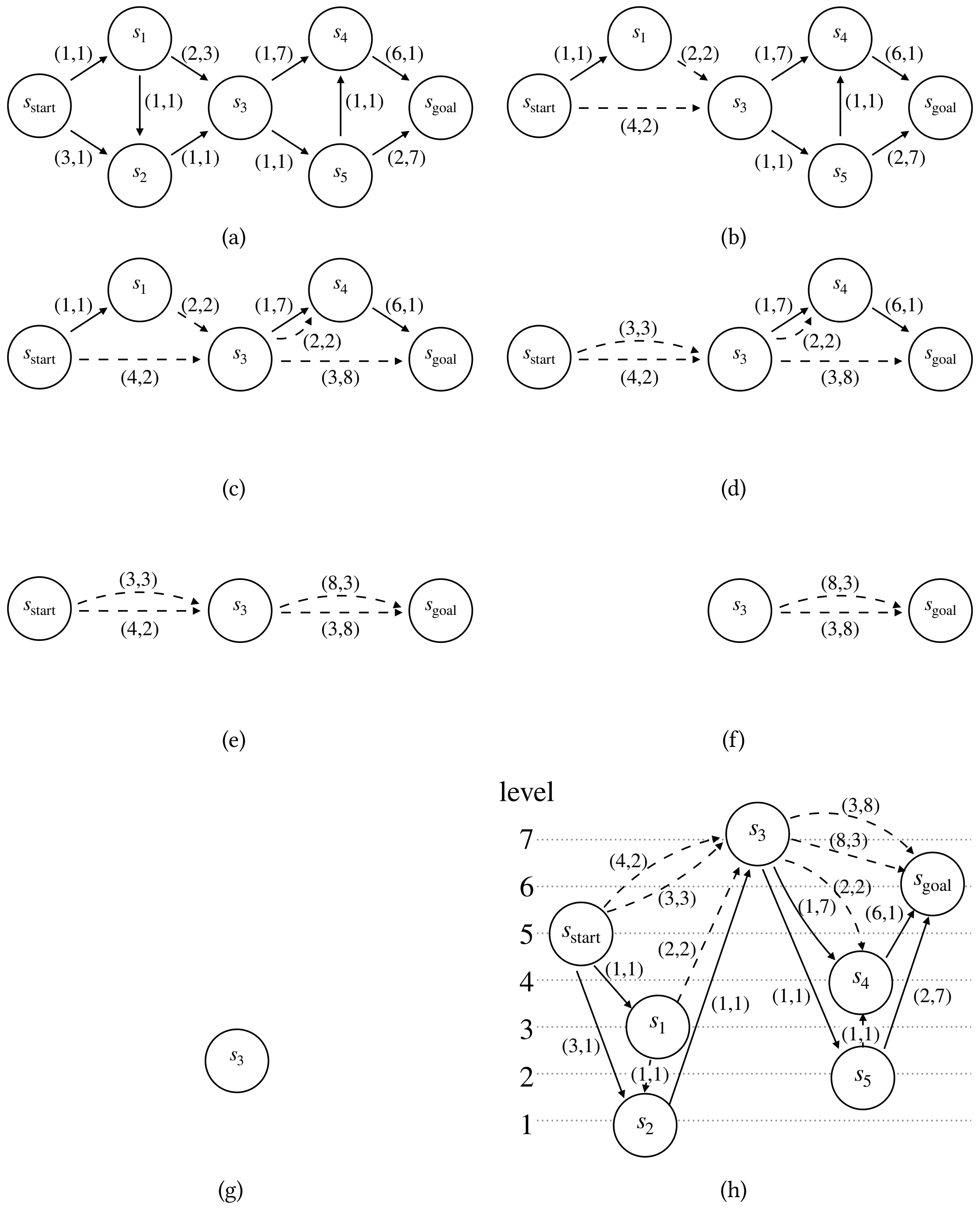


Figure 5.2: An example CH for multi-objective search. (a) shows the input graph, which is the graph in Figure 2.1. (b-g) show the steps for contracting the states in the order of $[s_2, s_5, s_1, s_4, s_{start}, s_{goal}, s_3]$. (h) shows the resulting CH. Dashed edges are the shortcuts added during the contraction.

from s_{start} to s_3 with cost $(4,2)$ is added to the graph because there is no other path from s_{start} to s_3 that dominates path $[\langle s_{start}, s_2, (3,1)\rangle, \langle s_2, s_3, (1,1)\rangle]$.

- *State s_5 is contracted. A shortcut from s_3 to s_4 with cost $(2,2)$ is added to the graph because there is no other path from s_3 to s_4 that dominates path $[\langle s_3, s_5, (1,1)\rangle, \langle s_5, s_4, (1,1)\rangle]$. A shortcut from s_3 to s_{goal} with cost $(3,8)$ is added to the graph because there is no other path from s_3 to s_{goal} that dominates path $[\langle s_3, s_5, (1,1)\rangle, \langle s_5, s_{goal}, (2,7)\rangle]$.*

- *State s_1 is contracted. A shortcut from s_{start} to s_3 with cost $(3,3)$ is added to the graph because there is no other path from s_{start} to s_3 that dominates path $[\langle s_{start}, s_1, (1,1)\rangle, \langle s_1, s_3, (2,2)\rangle]$.*

- *State s_4 is contracted. A shortcut from s_3 to s_{goal} with cost $(8,3)$ is added to the graph because there is no other path from s_3 to s_{goal} that dominates path $[\langle s_3, s_4, (2,2)\rangle, \langle s_3, s_{goal}, (6,1)\rangle]$.*

- *States s_{start}, s_{goal} and s_3 are contracted in order. No shortcuts are created.*

This example shows that a CH for multi-objective search can contain parallel edges even if the input graph does not contain one. For instance, the resulting CH contains two parallel edges from s_{start} to s_3, none of whose costs weakly dominate the others.

5.2.1.2 Batched Approach

As demonstrated in Example 9, a contraction can add parallel edges to the remaining graph. When a state is contracted, the search effort in WitnessSearch can be duplicated for different combinations of its parallel in-edges and its parallel out-edges. Our batched approach, outlined in Algorithm 10, reduces such duplicated search effort. Specifically, for every pair of in-neighbor u and out-neighbor v of s, the algorithm finds all two-hop paths Π from u to v that traverse s,

Algorithm 10: The Batched Preprocessing Algorithm

```
   Input  : A graph $G = \langle S, E \rangle$ and a number $L$ of states to contract
   Output: A CH
1  $S_{\text{CH}} \leftarrow S; E_{\text{CH}} \leftarrow \{\}$
2  while $|S_{CH}| - |S| < L$ do
3      $s \leftarrow$ choose the next state to contract
4      for $u \in in\_nbr(s)$ do
5          for $v \in out\_nbr(s)$ do
6              $\Pi \leftarrow$ all two-hop paths from $u$ to $v$ that traverse $s$
7              $\Pi_{\text{sc}} = \text{WitnessSearchBatch}(G, u, v, s, \Pi)$
8              for $\pi \in \Pi_{sc}$ do
9                  $\text{AddShortcut}(\langle u, v, \mathbf{c}(\pi) \rangle)$
10     add all edges incident on $s$ to $E_{\text{CH}}$
11     remove $s$ from $S$ and all edges incident on $s$ from $E$
12 add all remaining edges in $E$ to $E_{\text{CH}}$
13 return $G_{\text{CH}} = \langle S_{\text{CH}}, E_{\text{CH}} \rangle$;
```

that is, all paths consisting of an in-edge e' of s with $src(e) = u$ and an out-edge e'' of s with $tar(e') = v$. It then uses a single run of $\text{WitnessSearchBatch}$ to determine which paths in Π should result in shortcuts (Line 7). Function $\text{WitnessSearchBatch}$ returns a subset Π_{sc} of Π such that (1) no path in Π_{sc} is weakly dominated by any path from u to v that does not traverse s, (2) no path in Π_{sc} is dominated by any other path in Π, and (3) no two paths in Π_{sc} have the same cost.

Function $\text{WitnessSearchBatch}$, like WitnessSearch, is based on LTMOA*, or BOA* for the bi-objective case. Algorithm 11 shows the pseudocode for $\text{WitnessSearchBatch}$ that is based on LTMOA*. We omit the pseudocode for $\text{WitnessSearchBatch}$ that is based on BOA* because it is similar to Algorithm 11, with only differences in the dominance checks. We highlight the major changes of Algorithm 11 over LTMOA* by using "*" before line numbers in its pseudocode. The changes include (1) initializing variables (Lines 4-10), (2) deciding if a path in Π should result in a

Algorithm 11: Witness Search for the Batched Preprocessing Algorithm

```
1   Function WitnessSearchBatch($G = \langle S, E\rangle, u, v, s, \Pi$):
2       $n \leftarrow$ new node with $s(n) = u$ and $\mathbf{g}(n) = \mathbf{0}$
3       $Open \leftarrow \{n\}$
*4      remove paths dominated by other paths in $\Pi$ and, if several paths have the same
          cost, keep only one of them
*5      $N_{\text{candidate}} \leftarrow \emptyset$
*6      for each $\pi \in \Pi$ do
*7          $n \leftarrow$ new node with $s(n) = v$ and $\mathbf{g}(n) = \mathbf{c}(\pi)$
*8          add $n$ to $Open$ and $N_{\text{candidate}}$
*9      $\Pi_{\text{sc}} \leftarrow \emptyset$
*10     $\mathbf{c}^{\text{UB}} \leftarrow$ component-wise maximum of the costs of all paths in $\Pi$
11      for each $s \in S$ do
12          $\mathbf{G}^{\text{tr}}_{\text{cl}}(s) \leftarrow \emptyset$
13      while $Open \neq \emptyset$ do
14          extract a node $n$ from $Open$ with the lexicographically smallest $\mathbf{f}$-value, breaking
              ties in favor of nodes that are not in $N_{\text{candidate}}$
15          if IsDominated($n$) then
16              continue
17          Update($\mathbf{G}^{\text{tr}}_{\text{cl}}(s(n))$, $Tr(\mathbf{g}(n))$)
18          if $s(n) = v$ then
*19             if $n \in N_{candidate}$ then
*20                 add the corresponding path of $n$ to $\Pi_{\text{sc}}$
21              continue
22          for each $e \in out(s(n))$ do
*23             if $tar(e) = s$ then
*24                 continue
25              $n' \leftarrow$ new node with $s(n') = tar(e)$ and $\mathbf{g}(n') = \mathbf{g}(n) + \mathbf{c}(e)$
26              if IsDominated($n'$) then
27                  continue
28              add $n'$ to $Open$
29      return $\Pi_{sc}$
30  Function IsDominated($n$):
31      if $\exists \mathbf{x} \in \mathbf{G}^{tr}_{cl}(v) : \mathbf{x} \preceq Tr(\mathbf{f}(n))$ then
32          return true
33      if $\exists \mathbf{x} \in \mathbf{G}^{tr}_{cl}(s(n)) : \mathbf{x} \preceq Tr(\mathbf{g}(n))$ then
34          return true
*35     if $\exists i \in \{1, 2 \ldots N\} : f_i(n) > c^{UB}_i$ then
*36         return true
37      return false
```

shortcut (Lines 19-20), and (3) pruning nodes (Lines 23-24 and 35-36). We now elaborate on each change.

During the initialization, $\mathrm{WitnessSearchBatch}$ removes all paths dominated by other paths in Π from Π and, if several paths have the same cost, keeps only one of them. After removing these paths, Π contains the shortcut candidates that need to be checked. For each path $\pi \in \Pi$, $\mathrm{WitnessSearchBatch}$ creates a node n that contains state v and whose $\mathbf{g}$-value is $\mathbf{c}(\pi)$ and adds n to Open (Lines 7-8). These nodes are also stored in a set of nodes $N_{\text{candidate}}$ (Line 8). $\mathrm{WitnessSearchBatch}$ then initializes variable Π_{sc} to $\emptyset$ (Line 9). Additionally, it computes the component-wise maximum $\mathbf{c}^{\text{UB}}$ of the costs of all paths in Π (Line 10), which is later used in $\mathrm{IsDominated}$ for pruning nodes.

$\mathrm{WitnessSearchBatch}$ then runs an LTMOA*-like search from the source state u. Like LTMOA*, $\mathrm{WitnessSearchBatch}$ extracts a node with the lexicographically smallest $\mathbf{f}$-value from Open. Unlike LTMOA*, it breaks ties in favor of nodes that are not in $N_{\text{candidate}}$. Therefore, if there exists a path π' from u to v that does not traverse state s and has the same cost as some path $\pi \in \Pi$, $\mathrm{WitnessSearchBatch}$ finds π' before expanding the node created for π on Line 7. The next difference from LTMOA* occurs when a node n that contains the target state v is expanded. $\mathrm{WitnessSearchBatch}$ checks if n is in $N_{\text{candidate}}$. If so, n is a node created on Line 7 for some path $\pi \in \Pi$. If there is a path from u to v that does not traverse s and weakly dominates π, $\mathrm{WitnessSearchBatch}$ should have found this path before expanding n and would prune n because of the condition on Line 31 of $\mathrm{IsDominated}$. Therefore, by contradiction, π is not weakly dominated by any path from u to v that does not traverse s and hence should be added to Π_{sc}.

Function $\mathrm{WitnessSearchBatch}$ also has a different pruning strategy than LTMOA*. It prunes any nodes that contain state s (Lines 23-24) because it does not consider paths that traverse state

s. WitnessSearchBatch also prunes a node n if $f_i(n) > c_i^{\text{UB}}$ for any i (Lines 35-36) because, in this case, no path from u to v found via n weakly dominates any path in Π. Finally, when $Open$ becomes empty, the algorithm terminates and returns Π_{sc} (Line 29).

5.2.2 Query Algorithm

In this section, we describe how we combine CHs with multi-objective search algorithms in the query phase. Additionally, we describe a simple yet effective partial-expansion technique that reduces the query time by reducing the number of nodes inserted into $Open$.

5.2.2.1 Constructing Search Graphs

The query phase relies on the up-down property of CHs. That is, for any path π from state u to state v in the input graph G, there exists an up-down path from state u to state v in the CH G_{CH} that weakly dominates π (Theorem 5.2). Therefore, a Pareto frontier can be found by searching through the space of only up-down paths in G_{CH}.

While it is customary to use bi-directional Dijkstra's algorithm over CHs in the query phase of single-objective search with one direction considering only upward paths and the other direction considering only downward paths, the analog for multi-objective search requires careful examination. Bi-directional bi-objective Dijkstra's algorithm [65] is an algorithm that generalizes bi-directional Dijkstra's algorithm to bi-objective search. However, Hernández et al. [37] show that it is less efficient than BOA*. BOBA* [2] is another bi-objective search algorithm that utilizes two simultaneous bi-objective searches, one from the source and one from the target. However, the search in each direction is independent of the other one and hence cannot focus on only upward or downward paths, respectively.

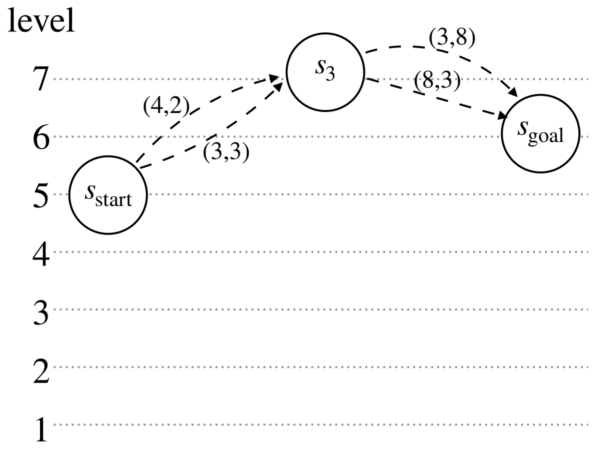


Figure 5.3: The search graph created for the query from s_{start} to s_{goal} and the CH in Figure 5.2h.

Our approach is to first build a *search graph* $\tilde{G}$ for given s_{start} and s_{goal}. $\tilde{G}$ is a subgraph of G_{CH} and consists of all up-down paths from s_{start} to s_{goal}. Then, we can run any multi-objective search algorithm (we use BOA* for problem instances with two objectives and LTMOA* for problem instances with more than two objectives) on $\tilde{G}$ to find a Pareto frontier. We denote $\tilde{G}$ by $\langle \tilde{S} = S^{\uparrow} \cup S^{\downarrow}, \tilde{E} \rangle$, where $S^{\uparrow}$ consists of all states that can be reached from s_{start} via an upward path and $S^{\downarrow}$ consists of all states that can reach s_{goal} via a downward path. $S^{\uparrow}$ and $S^{\downarrow}$ are computed by running a depth-first search on G_{CH} and its inverse graph, respectively. $\tilde{E}$ consists of (1) all upward edges with source states in $S^{\uparrow}$ and (2) all downward edges with target states in $S^{\downarrow}$.

Example 10. *Figure 5.3 shows the search graph $\tilde{G}$ constructed for the query from s_{start} to s_{goal} and the CH in Figure 5.2h. State set $S^{\uparrow}$ consists of s_{start} and s_3, and state set $S^{\downarrow}$ consists of s_3 and s_{goal}. The edge set of the search graph contains only the upward edges from s_{start} to s_3 and the downward edges from s_3 to s_{goal}.*

In search graph $\tilde{G}$, there are four paths from s_{start} to s_{goal}, namely, path $[\langle s_{start}, s_3, (3,3)\rangle, \langle s_3, s_{goal}, (3,8)\rangle]$ with a cost of $(6,11)$, path $[\langle s_{start}, s_3, (4,2)\rangle, \langle s_3, s_{goal}, (3,8)\rangle]$ with a cost of

$(7, 10)$*, path* $[\langle s_{start}, s_3, (3, 3)\rangle, \langle s_3, s_{goal}, (8, 3)\rangle]$ *with a cost of* $(11, 6)$*, and path* $[\langle s_{start}, s_3, (4, 2)\rangle,$ $\langle s_3, s_{goal}, (8, 3)\rangle]$ *with a cost of* $(12, 5)$*. These four paths exactly correspond to the four Pareto-optimal paths shown in Figure 2.1b. One can use BOA* to find these four paths in* $\tilde{G}$ *and then obtain the Pareto frontier for the original problem instance by unpacking these paths.*

5.2.2.2 Partial Expansions

In a CH for multi-objective search, there can be many (up to several hundred in our experiments) parallel edges from a state s to another state s' due to contractions. When expanding a node that contains state s, existing multi-objective search algorithms generate child nodes for all edges from s to s', which may be unnecessary if some of these child nodes are pruned later. Therefore, we propose "lazy" variants of LTMOA* and BOA* that utilize partial expansions to reduce the number of generated child nodes in many cases by generating them one by one, as needed. The idea of partial expansions comes from single-objective search [22], where a search algorithm keeps track of the child node to generate next for each expanded node. We adapt this idea to keep track of the child node to generate next for each pair of an expanded node n and one of the out-neighbors of $s(n)$. This enables the algorithm to identify quickly whether all child nodes for an out-neighbor state can be pruned without checking all corresponding out-edges.

Algorithm 12 shows LTMOA* with partial expansions. We omit the pseudocode for BOA* with partial expansions because it is similar to Algorithm 12 with the only differences being in the dominance checks. Algorithm 12 requires that, for any two states s and s', all edges from s to s' that are dominated by other edges from s to s' are removed, and, if several edges have the same cost, only one of them is kept. The remaining edges are sorted in order of lexicographically increasing costs. These changes (removing and sorting edges) are done in the preprocessing

Algorithm 12: LTMOA* with Partial Expansions

```
    Input  : A problem instance $\langle G, s_{start}, s_{goal}\rangle$ and a consistent heuristic function $\mathbf{h}$
    Output: A Pareto frontier
1   $n \leftarrow$ new node with $s(n) = s_{start}$, $\mathbf{g}(n) = \mathbf{0}$, and $p(n) = None$
2   $Open \leftarrow \{n\}$
3   $Sols \leftarrow \emptyset$
4   for each $s \in S$ do
5       $\mathbf{G}^{\mathrm{tr}}_{\mathrm{cl}}(s) \leftarrow \emptyset$
6   while $Open \neq \emptyset$ do
7       extract a node $n$ from $Open$ with the lexicographically smallest $\mathbf{f}$-value
*8      if $p(n) \neq None$ then
*9          GenerateNext($parent(n), s(n), idx(n) + 1$)
10      if IsDominated($n$) then
11          continue
12      Update($\mathbf{G}^{\mathrm{tr}}_{\mathrm{cl}}(s(n))$, $Tr(\mathbf{g}(n))$)
13      if $s(n) = s_{goal}$ then
14          add the corresponding solution of $n$ to $Sols$
15          continue
16      for $s' \in out\_nbr(s(n))$ do
*17         GenerateNext($n, s', 1$)
18  return $Sols$

*19 Function GenerateNext($n, s, i$):
*20     $\mathbf{g}^{\min} = \mathbf{g}(n) + \mathbf{c}^{\min}(s(n), s)$
*21     if $\exists \mathbf{x} \in \mathbf{G}^{tr}_{cl}(s_{goal}) : \mathbf{x} \preceq Tr(\mathbf{g}^{min} + \mathbf{h}(s))$ or $\exists \mathbf{x} \in \mathbf{G}^{tr}_{cl}(s) : \mathbf{x} \preceq Tr(\mathbf{g}^{min})$ then
*22         return
*23     for $j = i, i+1 \ldots m_{s(n),s}$ do
*24         $n' \leftarrow$ new node with $s(n') = s$, $\mathbf{g}(n') = \mathbf{g}(n) + \mathbf{c}(e^{j}_{s(n),s})$, $idx(n') = j$, and
             $p(n') = n$
*25         if IsDominated($n'$) then
*26             continue
*27         add $n'$ to $Open$
*28         return
*29     return
```

phase. We use $m_{s,s'}$ to denote the number of edges from s to s' and $[e^1_{s,s'}, e^2_{s,s'} \ldots e^{m_{s,s'}}_{s,s'}]$ to denote the sequence of these edges sorted in lexicographical order. We say that i is the *index* of edge $e^i_{s,s'}$. Additionally, we use $\mathbf{c}^{\min}(s, s')$ to denote the component-wise minimum of the costs of all edges from s to s' and $p(n)$ to store the parent node of a node n.

We highlight the major changes of Algorithm 12 over LTMOA* by using "*" before the line numbers. When expanding a node n, the algorithm uses $\mathrm{GenerateNext}$ for each out-neighbor s of $s(n)$ (Line 17). On Lines 21-22, $\mathrm{GenerateNext}$ first checks (1) if the component-wise minimum $\mathbf{f}$-value of these child nodes is weakly dominated by the $\mathbf{f}$-value of any expanded node that contains state s_{goal} and (2) if the component-wise minimum $\mathbf{g}$-value of these child nodes is weakly dominated by the $\mathbf{g}$-value of any expanded node that contains state s. Let $\mathbf{g}^{\min} = \mathbf{g}(n)+\mathbf{c}^{\min}(s(n), s)$ be the component-wise minimum $\mathbf{g}$-value of these child nodes. Because LTMOA* uses dimensionality reduction, checking (1) and (2) can be done by checking $\exists \mathbf{x} \in \mathbf{G}^{\mathrm{tr}}_{\mathrm{cl}}(s_{goal}) : \mathbf{x} \preceq Tr(\mathbf{g}^{\min}+\mathbf{h}(s))$ and $\exists \mathbf{x} \in \mathbf{G}^{\mathrm{tr}}_{\mathrm{cl}}(s) : \mathbf{x} \preceq Tr(\mathbf{g}^{\min})$, respectively . If (1) or (2) holds, one can conclude that all child nodes of n that contain state s will be pruned. Therefore, $\mathrm{GenerateNext}$ returns without adding any node to $Open$ (Line 22). Otherwise, $\mathrm{GenerateNext}$ iterates over all edges $[e^1_{s,s'}, e^2_{s,s'} \dots e^{m_{s,s'}}_{s,s'}]$ until it finds the first edge that results in an undominated child node n'. Function $\mathrm{GenerateNext}$ then adds n' to $Open$ and returns (Lines 23-28). For each node n, the algorithm uses $idx(n)$ to record the index of the edge that was used to generate it.

When a node n is extracted from $Open$ and n is not the root node, the algorithm calls $\mathrm{GenerateNext}$ to generate the next undominated child node of $p(n)$ that contains state $s(n)$, if one exists (Line 9). When iterating over the edges from $s(p(n))$ to $s(n)$, $\mathrm{GenerateNext}$ starts with the edge with index $idx(n) + 1$ because all edges with smaller indices have already been iterated over in the previous calls of $\mathrm{GenerateNext}$ for $p(n)$ and $s(n)$. The rest of Algorithm 12 is the same as LTMOA*.

Example 11. *Consider the search graph in Figure 5.3. Assume that we use the perfect-distance heuristic. The heuristics for states s_{start}, s_3, and s_{goal} are $(6, 5)$, $(3, 3)$, and $(0, 0)$, respectively. The*

Iter	*Open* $\langle s(n), \mathbf{g}(n), \mathbf{f}(n)\rangle$	Generated $\langle s(n), \mathbf{g}(n), \mathbf{f}(n)\rangle$	Update of $g_2^{\min}(s(n))$
1	$\langle s_{start}, (0,0), (6,5)\rangle *$	$\langle s_3, (3,3), (6,6)\rangle$	$g_2^{\min}(s_{start}) = 0$
2	$\langle s_3, (3,3), (6,6)\rangle *$	$\langle s_{goal}, (6,11), (6,11)\rangle$ $\langle s_3, (4,2), (7,5)\rangle$	$g_2^{\min}(s_3) = 3$
3	$\langle s_{goal}, (6,11), (6,11)\rangle *$ $\langle s_3, (4,2), (7,5)\rangle$	$\langle s_{goal}, (11,6), (11,6)\rangle$	$g_2^{\min}(s_{goal}) = 11$
4	$\langle s_3, (4,2), (7,5)\rangle *$ $\langle s_{goal}, (11,6), (11,6)\rangle$	$\langle s_{goal}, (7,10), (7,10)\rangle$	$g_2^{\min}(s_3) = 2$
5	$\langle s_{goal}, (7,10), (7,10)\rangle *$ $\langle s_{goal}, (11,6), (11,6)\rangle$	$\langle s_{goal}, (12,5), (12,5)\rangle$	$g_2^{\min}(s_{goal}) = 10$
6	$\langle s_{goal}, (11,6), (11,6)\rangle *$ $\langle s_{goal}, (12,5), (12,5)\rangle$		$g_2^{\min}(s_{goal}) = 6$
7	$\langle s_{goal}, (12,5), (12,5)\rangle *$		$g_2^{\min}(s_{goal}) = 5$
8	empty		

Table 5.1: Trace of $Open$, generated nodes, and $g_2^{\min}$ in each iteration of Example 11. "$*$" marks the node that is extracted in that iteration.

edges from s_{start} to s_3 are sorted in the order of $[\langle s_{start}, s_3, (3,3)\rangle, \langle s_{start}, s_3, (4,2)\rangle]$, and the edges from s_3 to s_{goal} are sorted in the order of $[\langle s_3, s_{goal}, (3,8)\rangle, \langle s_3, s_{goal}, (8,3)\rangle]$. Table 5.1 shows a trace of $Open$, generated nodes, and changes to g_2^{min} in each iteration of BOA with partial expansions:*

1. *In Iteration 1, the algorithm expands node $\langle s_{start}, (0,0), (6,5)\rangle$. Although s_{start} has two out-edges, the algorithm generates only child node $\langle s_3, (3,3), (6,6)\rangle$ for edge $\langle s_{start}, s_3, (3,3)\rangle$.*

2. *In Iteration 2, the algorithm expands node $\langle s_3, (3,3), (6,6)\rangle$. The algorithm first generates the second child node $\langle s_3, (4,2), (7,5)\rangle$ of node $\langle s_{start}, (0,0), (6,5)\rangle$ using edge $\langle s_{start}, s_3, (4,2)\rangle$. It then generates child node $\langle s_{goal}, (6,11), (6,11)\rangle$ of node $\langle s_3, (3,3), (6,6)\rangle$ for edge $\langle s_3, s_{goal}(3,8), \rangle$.*

3. *In Iteration 3, the algorithm expands node $\langle s_{goal}, (6,11), (6,11)\rangle$ and adds a solution with cost $(6,11)$ to $Sols$. The algorithm then generates the second child node $\langle s_{goal}, (11,6), (11,6)\rangle$ of $\langle s_3, (3,3), (6,6)\rangle$ for edge $\langle s_3, s_{goal}, (8,3)\rangle$.*

4. *In Iteration 4, the algorithm expands node* $\langle s_3, (4,2), (7,5)\rangle$ *and generates its child node* $\langle s_{goal}, (7,10), (7,10)\rangle$ *for edge* $\langle s_3, s_{goal}, (3,8)\rangle$.

5. *In Iteration 5, the algorithm expands node* $\langle s_{goal}, (7,10), (7,10)\rangle$ *and adds a solution with cost* $(7,10)$ *to* $Sols$*. The algorithm then generates the second child node* $\langle s_{goal}, (12,5), (12,5)\rangle$ *of* $\langle s_{start}, (4,2), (7,5)\rangle$ *for edge* $\langle s_3, s_{goal}, (8,3)\rangle$.

6. *In Iterations 6-7, the algorithm expands the remaining nodes in* $Open$ *and finds two solutions with costs* $(11,6)$ *and* $(12,5)$*, respectively.*

The algorithm terminates when $Open$ *becomes empty in Iteration 8 and returns* $Sols$ *as the Pareto frontier from* s_{start} *to* s_{goal}*. Despite using partial expansions, the search algorithm still expands nodes in order of lexicographically increasing* $\mathbf{f}$*-values.*

5.3 Correctness

In this section, we prove the up-down property of multi-objective CHs. We study only the batched preprocessing algorithm, but the lemmas and theorem in this section also hold for the basic preprocessing algorithm after modifying the proof of Lemma 5.1.

We use s_i and $G_{(i)} = \langle S_{(i)}, E_{(i)}\rangle$ to denote the i-th contracted state and the remaining graph after the first $i-1$ states have been contracted, respectively. Specifically, $G_{(1)}$ is the same as the input graph G, and $G_{(L+1)}$ is the core of the CH.

Lemma 5.1. *For any* i *and* j *with* $1 \leq i \leq j \leq L+1$*, two states* u *and* v *that are in both* $S_{(i)}$ *and* $S_{(j)}$*, and a path* π *from* u *to* v *in* $G_{(i)}$*, there exists a path from* u *to* v *in* $G_{(j)}$ *that weakly dominates* π.

Proof. Considering any given i, we prove this lemma by induction on j. The lemma trivially holds when $j = i$. Assume that the lemma holds for $j \leq l-1$, and consider a path π from u to v in $G_{(l)}$ with both states u and v also in $S_{(l)}$. Because the lemma holds for $j = l-1$, there exists a path $\pi_{(l-1)}$ from u to v in $G_{(l-1)}$ that weakly dominates π. If $\pi_{(l-1)}$ does not traverse state $s_{(l-1)}$, that is, the $(l-1)$-th contracted state, it is not affected by the $(l-1)$-th contraction and still in $G_{(l)}$. Thus, the lemma holds for $j = l$. Otherwise, $\pi_{(l-1)}$ traverses state $s_{(l-1)}$. Because u and v are in $S_{(l)}$, $s_{(l-1)}$ cannot be u or v. Therefore, $\pi_{(l-1)}$ traverses state $s_{(l-1)}$ with edges e and e' such that $tar(e) = src(e') = s_{(l-1)}$. When Algorithm 10 contracts state $s_{(l-1)}$ and reaches Line 7 with Π containing the two-hop path π_{sc} that bridges e and e', we distinguish the following cases:

1. There exists a path π' from $src(e)$ to $tar(e')$ in $G_{(l-1)}$ that does not traverse $s_{(l-1)}$ and whose cost weakly dominates $\mathbf{c}(e) + \mathbf{c}(e')$. Because π' does not traverse $s_{(l-1)}$, it is still in $G_{(l)}$. We can construct a path from u to v in $G_{(l)}$ that weakly dominates $\pi_{(l-1)}$ by substituting edges e and e' in $\pi_{(l-1)}$ with π'. The resulting path weakly dominates π, and hence the lemma holds.

2. There does not exist a path π' from $src(e)$ to $tar(e')$ in $G_{(l-1)}$ that does not traverse $s_{(l-1)}$ and whose cost weakly dominates $\mathbf{c}(e) + \mathbf{c}(e')$. Because π_{sc} is in Π and weakly dominates itself, there must exist some paths in Π whose costs weakly dominate $\mathbf{c}(\pi_{\text{sc}}) = \mathbf{c}(e) + \mathbf{c}(e')$. From these paths, we can always find a path π'' that is not dominated by any other path in Π. π'' is not weakly dominated by any path from u to v that does not traverse s as well. Consider the set of paths Π_{sc} returned by $\text{WitnessSearchBatch}$. Π_{sc} is a maximal subset of Π such that (1) no path in Π_{sc} is weakly dominated by any path from u to v that does not traverse s, (2) no path in Π_{sc} is dominated by any other path in Π, and (3) no two paths in Π_{sc}

have the same cost. There must exist some path in Π_{sc} that has the same cost as π'' because, otherwise, Π_{sc} is not a maximal subset. Therefore, a shortcut edge from $src(e)$ to $tar(e')$ whose cost weakly dominates $\mathbf{c}(e)+\mathbf{c}(e')$ is added to $E_{(l)}$ (Line 9). We can construct a path from u to v in $G_{(l)}$ whose cost weakly dominates $\mathbf{c}(\pi_{(l-1)})$ by substituting edges e and e' in $\pi_{(l-1)}$ with this shortcut edge. This path weakly dominates π, and hence the lemma holds.

Therefore, the lemma holds for all j. □

Lemma 5.2. *For any i with $1 \leq i \leq L+1$, states u and v that are both in $S_{(i)}$, and a path π from u to v that is in $G_{(i)}$, there exists an up-down path from u to v in G_{CH} that weakly dominates π.*

Proof. We first assume that π is a simple path and will later show that the lemma still holds without this assumption.

We prove this lemma by induction on i from $L+1$ to 1. When $i = L+1$, $G_{(i)}$ is the core of the CH. Every path π in $G_{(i)}$ traverses states with the same level number and hence is also an up-down path in G_{CH}. Because π weakly dominates itself, the lemma holds for $i = L+1$.

Assume that the lemma holds for $i \geq l+1$, $l \leq L$, and consider any path π from u to v that is in $S_{(l)}$. We distinguish three cases:

1. Neither state u or v is $s_{(l)}$. Both u and v remain in $S_{(l+1)}$ after the l-th contraction. From Lemma 5.1, there exists a path π' from u to v in $G_{(l+1)}$ that weakly dominates π. Because the lemma holds for $i = l+1$, π' is weakly dominated by some up-down path from u to v in G_{CH}. Therefore, π is also weakly dominated by this up-down path.

2. State u is $s_{(l)}$. Let e_1 denote the first edge of π and $\pi_{[1:]}$ denote the path obtained from π by removing the first edge. $\pi_{[1:]}$ is a path from $tar(e_1)$ to v. Because π is a simple path, only its

first state is $s_{(l)}$. Path $\pi_{[1:]}$ is not affected by the l-th contraction and hence is also in $G_{(l+1)}$. Since the lemma holds for $i \geq l+1$, there exists an up-down path π' from $tar(e_1)$ to v in G_{CH} that weakly dominates $\pi_{[1:]}$. Edge e_1 is added to E_{CH} when $s_{(l)}$ is contracted, and it is an upward edge because its source $u = s_{(l)}$ is contracted before its target. We can obtain an up-down path π_{ud} in G_{CH} from u to v by joining e_1 and π', which weakly dominates π because

$$\begin{aligned}\mathbf{c}(\pi_{\text{ud}}) &= \mathbf{c}(e_1) + \mathbf{c}(\pi') \\ &\preceq \mathbf{c}(e_1) + \mathbf{c}(\pi_{[1:]}) \\ &= \mathbf{c}(\pi).\end{aligned}$$

3. State v is $s_{(l)}$. Let e_{-1} denote the last edge of π, and $\pi_{[:-1]}$ denote the path obtained from π by removing the last edge. $\pi_{[:-1]}$ is a path from u to $src(e_{-1})$. Because π is a simple path, only its last state is $s_{(l)}$. Path $\pi_{[:-1]}$ is not affected by the l-th contraction and hence is also in $G_{(l+1)}$. Since the lemma holds for $i \geq l+1$, there exists an up-down path π' from u to $src(e_{-1})$ in G_{CH} that weakly dominates $\pi_{[:-1]}$. Edge e_{-1} is added to E_{CH} when $s_{(l)}$ is contracted, and it is a downward edge since its target $v = s_{(l)}$ is contracted before

its source. We can obtain an up-down path π_{ud} in G_{CH} from u to v by joining π' and e_{-1}, which weakly dominates π because

$$\begin{aligned}\mathbf{c}(\pi_{\text{ud}}) &= \mathbf{c}(\pi') + \mathbf{c}(e_{-1}) \\ &\preceq \mathbf{c}(\pi_{[:-1]}) + \mathbf{c}(e_{-1}) \\ &= \mathbf{c}(\pi).\end{aligned}$$

Therefore, the lemma holds for $i = l$ and hence holds for all i, $1 \leq i \leq L + 1$.

Now we consider the case that π is not a simple path. There must exist a simple path π' in $G_{(i)}$ that weakly dominates π. Because the lemma holds for simple paths, there exists an up-down path from u to v in G_{CH} that weakly dominates π' and hence also weakly dominates π. □

Theorem 5.2. *For any path π from u to v in G, there exists an up-down path from u to v that weakly dominates π in G_{CH}.*

Proof. Theorem 5.2 is a special case of Lemma 5.2 when $i = 1$. □

5.4 Experimental Evaluation

In this section, we evaluate our CH-based approach on road networks from the 9th DIMACS Implementation Challenge: Shortest Path.[2] We implemented all algorithms in C++ on a common code base[3] and ran all experiments on a MacBook Pro with an M1 Pro CPU and 32GB of memory.

[2]`http://users.diag.uniroma1.it/challenge9/download.shtml`.
[3]`https://github.com/HanZhang39/Bi-Objective-Contraction-Hierarchy`.

To order states for contraction (Line 3 in Algorithms 9 and 10), we assign a priority $\psi(s)$ to each state s and contract the lowest-priority state at each iteration. $\psi(s)$ is defined as a linear combination of ordering heuristics that are used by existing work [28, 34]. Specifically, we use $\kappa(s)$ to denote the ratio of the number of shortcuts to add when contracting s and the number of edges incident on s. Furthermore, we use $\eta(s)$ to denote the *height* of a state s to be one plus the height of the highest state with an upward edge to s or a value of one if no such state exists. Intuitively, choosing states with small heights to contract next leads to a more even contraction across the graph. In our implementation, we set $\psi(s) := 10 \cdot \kappa(s) + \eta(s)$. We also implemented the lazy-update scheme [28], which recalculates the priority of a state when it is popped from the priority queue and reinserts it into the priority queue if its priority has become higher than the second-lowest priority.

5.4.1 Problem Instances with Two Objectives

In this section, we evaluate our CH-based approach on problem instances with two objectives. We use the two objectives that are available in the DIMACS benchmark, namely travel time (t) and travel distance (d). For each road network, we use the 100 queries used by Ahmadi et al. [2]. The time limit for solving each query is 30 minutes.

5.4.1.1 Comparing Different Preprocessing Algorithms

We start by evaluating the impact of different contraction ratios (that is, percentages of states to contract, which is captured by L in Algorithms 9 and 10) and different preprocessing approaches. We also evaluate the resulting CHs in the query phase. Here, we use the NE road network (1.5M

Algorithm	Preprocessing t_{prep}	$\lvert E_{\text{CH}}\rvert$	Query #exp	$t_{\text{BOA}^*}^{+\text{CH}}$	$t_{\text{BOA}^*}^{+\text{CH+p}}$
NE					
contracting 99% of states					
support-point	6min	8.0M	265K	3.93(100)	2.56
basic	8min	8.0M	262K	3.92(100)	2.48
batched	7min	8.0M	262K	3.82(100)	2.41
contracting 99.5% of states					
support-point	8min	8.1M	145K	3.78(100)	1.81
basic	13min	8.1M	142K	3.20(100)	1.62
batched	10min	8.1M	141K	**3.14**(100)	1.52
contracting 99.95% of states					
support-point	37min	9.2M	40K	6.10(100)	0.71
basic	3hr53min	8.8M	35K	3.62(100)	**0.51**
batched	21min	8.8M	36K	3.80(100)	**0.51**
contracting 100% of states					
support-point	2hr53min	11.8M	38K	19.92(100)	0.89
basic	timeout				
batched	1hr46min	10.3M	**32K**	11.67(100)	0.64
LKS					
contracting 99% of states					
support-point	14min	14.2M	1,155K	25.96(100)	14.87
basic	19min	14.2M	1,158K	24.41(100)	13.90
batched	16min	14.2M	1,156K	25.32(100)	15.27
contracting 99.5% of states					
support-point	23min	14.6M	604K	19.35(100)	8.80
basic	41min	14.6M	607K	20.32(100)	9.54
batched	24min	14.6M	613K	**17.86**(100)	8.42
contracting 99.95% of states					
support-point	3hr34min	18.7M	167K	78.11 (87)	5.56
basic	timeout				
batched	1hr08min	16.5M	163K	39.27 (96)	**4.35**
contracting 100% of states					
support-point	timeout				
basic	timeout				
batched	10hr08min	21.1M	**161K**	137.86 (84)	5.41

Table 5.2: Experimental results for different contraction approaches and contraction ratios on the NE and LKS road networks. We report the preprocessing times t_{prep}, the numbers of edges $|E_{\text{CH}}|$ in each CH, the average numbers of expanded nodes #exp, the average query times (in seconds) $t_{\text{BOA}^*}^{+\text{CH}}$ for BOA* with CH (but without partial expansions), with the number of solved instances in parentheses, and the average query time (in seconds) $t_{\text{BOA}^*}^{+\text{CH+p}}$ for BOA* with CH and partial expansions (here, all algorithms solved all instances). For each road network, the average runtimes are calculated over all instances that are solved by both BOA* and BOA* with partial expansions for all CHs.

states and 3.9M edges) and the LKS road network (2.8M states and 6.9M edges), two medium-sized maps from the DIMACS dataset, and set a time limit of 12 hours for the preprocessing phase.

We evaluate the contraction ratios 99%, 99.5%, 99.95%, and 100% and three preprocessing approaches, namely the basic approach, the batched approach, and the preprocessing approach proposed by Storandt [69], referred to as the *support-point* approach. For each in-neighbor s' and out-neighbor s'' of state s, the witness search of the support-point approach runs a series of single-objective searches from s' to s''. Each single-objective search is parameterized by a λ-value, $\lambda \in [0, 1]$, and finds a path π' that minimizes $\lambda c_1(\pi') + (1 - \lambda)c_2(\pi')$. For every 2-hop path π from s' via s to s'', if the witness search finds a path whose cost dominates $\mathbf{c}(\pi)$, then π does not result in a shortcut. Otherwise, a shortcut is added. Adding the shortcut may be unnecessary but does not affect the correctness of the query phase. We use a sequence of three λ-values $[\lambda_1, \lambda_2, \lambda_3]$ as described in Storandt [69], with $\lambda_1 = 0$, $\lambda_2 = 1$, and $\lambda_3 = (c_2 - c'_2)/(c_2 - c'_2 + c_1 - c'_1)$, where $\mathbf{c}$ and $\mathbf{c}'$ denote the path cost found with λ_1 and λ_2, respectively.

Our results, summarized in Table 5.2, show that the CHs produced by the basic and batched approaches have similar numbers of edges for the same contraction ratio. However, the basic approach needs much more preprocessing time because its number of witness searches increases dramatically with the contraction ratio. CHs produced by the support-point approach have the largest number of edges because of the unnecessary shortcuts it adds. The unnecessary shortcuts also cause the support-point approach to have a larger preprocessing time than the batched approach for larger ($\geq 99.95\%$) contraction ratios. The results also show that contracting the last 0.05% of the states requires a large preprocessing time and results in a large number of additional edges.

With increasing contraction ratios, the number of expanded nodes in the query phase decreases. In contrast, the average query time of BOA* with CHs increases because a large number of edges in the CH slows down the search algorithm. With the addition of partial expansions, the number of expanded nodes does not change, but the query times are reduced by up to a factor of 20. For the same contraction ratios, BOA* with CHs produced by the support-point approach has a larger average query time than BOA* with CHs produced by the batched approach due to the unnecessary edges that the support-point approach adds.

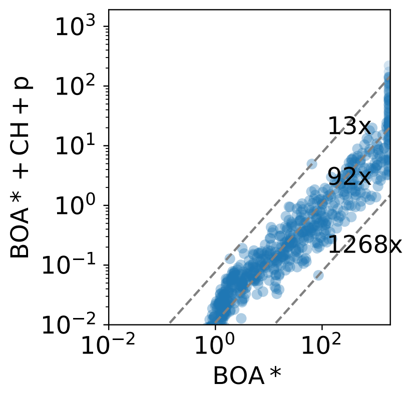
5.4.1.2 Comparing Different Query Algorithms

We evaluate the scalability of our CH-based approach and the speed-ups it enables on road networks of different sizes. Here, we use seven road networks, whose numbers of states range from 1 million to 14 million, together with the batched approach and a contraction ratio of 99.95%. For every road network, the number of edges in the CH is smaller than $2.5\times$ the number of edges in the input graph. For all seven road networks, the preprocessing times are less than 6 hours.

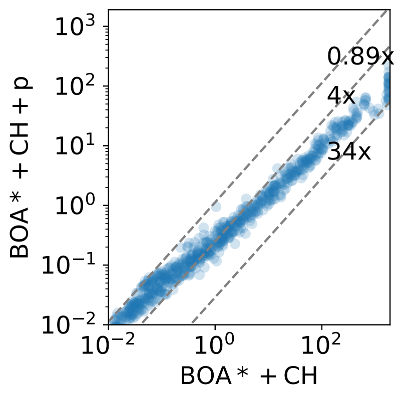
We evaluate three algorithms for the query phase, namely BOA*, BOA* with CHs (+CH), and BOA* with CHs and partial expansions (+CH+p). The results are summarized in Table 5.3. All average and maximum values are calculated over the instances solved by all three algorithms. The numbers of generated nodes are the numbers of nodes inserted into $Open$. We see dramatic reductions in the numbers of expanded nodes when comparing BOA*+CH or BOA*+CH+p to BOA*. While BOA*+CH and BOA*+CH+p expand the same number of nodes, BOA*+CH+p generates fewer nodes and hence has smaller average and maximal query times and larger numbers of solved instances. This demonstrates that many nodes inserted into $Open$ by BOA*+CH are later pruned when extracted from $Open$.

Algorithm	#solved	t_{avg}	t_{max}	#exp	#gen
FLA (1.1M states and 2.7M edges)					
t_{prep}: 6min			$\lvert E_{\text{CH}}\rvert$: 5.5M		
BOA*	100	22.43	662.59	6,106K	7,507K
BOA*+CH	100	0.36	10.44	14K	151K
BOA*+CH+p	100	**0.08**	**1.39**	14K	**45K**
NE (1.5M states and 3.9M edges)					
t_{prep}: 21min			$\lvert E_{\text{CH}}\rvert$: 8.8M		
BOA*	100	56.05	1,729.45	12,578K	16,281K
BOA*+CH	100	3.80	109.33	36K	939K
BOA*+CH+p	100	**0.51**	**12.68**	36K	**211K**
CAL (1.9M states and 4.7M edges)					
t_{prep}: 13min			$\lvert E_{\text{CH}}\rvert$: 9.4M		
BOA*	99	61.19	1,617.97	14,923K	18,679K
BOA*+CH	100	1.62	43.12	28K	479K
BOA*+CH+p	100	**0.29**	**5.70**	28K	**139K**
LKS (2.8M states and 6.9M edges)					
t_{prep}: 1hr08min			$\lvert E_{\text{CH}}\rvert$: 16.5M		
BOA*	78	208.57	1,631.61	46,928K	59,342K
BOA*+CH	96	23.94	192.44	117K	5,109K
BOA*+CH+p	**100**	**2.83**	**24.51**	117K	**918K**
E (3.6M states and 8.8M edges)					
t_{prep}: 42min			$\lvert E_{\text{CH}}\rvert$: 18.9M		
BOA*	79	269.10	1,770.45	55,099K	69,055K
BOA*+CH	98	18.52	164.92	101K	3,306K
BOA*+CH+p	**100**	**1.86**	**21.08**	101K	**659K**
W (6.3M states and 15.2M edges)					
t_{prep}: 48min			$\lvert E_{\text{CH}}\rvert$: 29.6M		
BOA*	81	228.58	1,784.56	60,605K	73,867K
BOA*+CH	100	4.24	27.98	100K	1,276K
BOA*+CH+p	100	**0.80**	**4.28**	100K	**370K**
CTR (14.1M states and 34.3M edges)					
t_{prep}: 5hr48min			$\lvert E_{\text{CH}}\rvert$: 77.5M		
BOA*	37	403.84	1,634.94	87,751K	106,364K
BOA*+CH	83	28.31	223.18	165K	6,294K
BOA*+CH+p	**100**	**3.01**	**19.91**	165K	**1,076K**

Table 5.3: Experimental results for the query phases of different algorithms on different road networks. For each road network, we report the preprocessing times t_{prep} and the numbers of edges $|E_{\text{CH}}|$ in the CH. For each algorithm, we report the numbers of instances solved (#solved) within 30 minutes, the average (t_{avg}) and maximal (t_{max}) query times in seconds, and the average numbers of expanded (#exp) and generated (#gen) nodes.



(a)



(b)

Figure 5.4: Runtimes of different query algorithms on bi-objective road-network problem instances across all seven road networks.

Figures 5.4a and 5.4b show the runtimes (in seconds) of BOA*+CH+p compared to BOA* and BOA*+CH on individual instances, respectively. The x-axes in Figures 5.4a and 5.4b correspond to the runtimes of BOA* and BOA*+CH, respectively, while the y-axes correspond to the runtimes of BOA*+CH+p. In each figure, the diagonal dashed lines and the numbers labeling them correspond to the minimum, median, and maximum speed-ups of BOA*+CH+p calculated over all instances solved by both algorithms that are compared. The query times of BOA*+CH+p are always smaller than those of BOA*, with a minimum speed-up of 13 times and a maximum speed-up of $1{,}268$ times.

Our experimental results also show that solving a bi-objective search problem instance directly can be more time-consuming than building a CH and solving it. This is so because the runtime of solving a bi-objective search problem instance can be exponential in $|S|$ (because the size of the Pareto frontier can be exponential in $|S|$ [19, 10]). This is in contrast to single-objective search, since a single-objective search instance can be solved in $O(|S|+|E|)$ time, while building a CH requires checking at least each contracted state and its incident edges. Therefore, building a CH and solving a single-objective search instance cannot be more efficient than solving the instance directly. Overall, our results show that CHs enable bi-objective search algorithms to solve problem instances on road networks with far less computation time and memory.

5.4.2 Problem Instances with More than Two Objectives

We compare LTMOA*, LTMOA* with CHs (+CH), and LTMOA* with CHs and partial expansion (+CH+p) on problem instances with more than two objectives. We use the NY road network with three, four, and five objectives. In addition to travel time (t) and travel distance (d), we use the economic cost (m) [54], the number of edges (l) [47], and a random integer from 1 to 100 (r) [36] as the third, fourth, and fifth objectives, respectively. We use the 100 problem instances used by Sedeño-Noda and Colebrook [65] and Ahmadi et al. [2]. The preprocessing algorithm terminates when it reaches the contraction ratio of 99.95% or a time limit of 12 hours. The time limit for solving each query is 5 minutes.

The results are summarized in Table 5.4. For all three numbers of objectives, the preprocessing algorithm reaches the time limit of 12 hours. We report the contraction ratios of the resulting CHs. As the number of objectives increases, the contraction ratio decreases. Table 5.4 also shows the results for different query algorithms. All average and maximum values are calculated over the instances solved by all three algorithms. The numbers of generated nodes are the numbers of nodes inserted into $Open$. Similar to the results for the bi-objective case, LTMOA*+CH and LTMOA*+CH+p solve more instances than LTMOA* and have smaller average numbers of expanded nodes. Compared to LTMOA*+CH, LTMOA*+CH+p generates fewer nodes and hence has smaller

Algorithm	#solved	t_{avg}	t_{max}	#exp	#gen
NY (m-t-d)					
contraction ratio: 99.91%			$\lvert E_{\text{CH}}\rvert$: 2.1M		
LTMOA*	73	34.91	241.29	17,481K	18,749K
LTMOA*+CH	81	23.37	229.70	290K	6,819K
LTMOA*+CH+p	**96**	**2.84**	**23.88**	290K	**1,093K**
NY (l-m-t-d)					
contraction ratio: 99.58%			$\lvert E_{\text{CH}}\rvert$: 2.5M		
LTMOA*	36	25.53	254.91	3,996K	4,707K
LTMOA*+CH	38	8.48	110.05	157K	1,533K
LTMOA*+CH+p	**40**	**4.54**	**50.20**	157K	**391K**
NY (l-m-t-d-r)					
contraction ratio: 98.95%			$\lvert E_{\text{CH}}\rvert$: 8.5M		
LTMOA*	23	16.11	219.68	1,389K	1,590K
LTMOA*+CH	24	8.44	94.42	74K	2,194K
LTMOA*+CH+p	**26**	**2.57**	**36.53**	74K	**180K**

Table 5.4: Experimental results for different query algorithms on the NY road network with different numbers of objectives. For each number of objectives, we report the contraction ratio after preprocessing and the number of edges $|E_{\text{CH}}|$ in the CH. For each algorithm, we report the numbers of instances solved (#solved) within 5 minutes, the average (t_{avg}) and maximal (t_{max}) query times in seconds, and the average numbers of expanded (#exp) and generated (#gen) nodes.

average query times. The speed-ups of LTMOA*+CH+p over LTMOA* in average runtimes for problem instances with three, four, and five objectives are $12.3\times$, $5.6\times$, and $6.3\times$, respectively.

Figure 5.5 shows the runtime comparisons between LTMOA*+CH+p and LTMOA* for different numbers of objectives. The x-axes and the y-axes correspond to the runtimes of LTMOA* and LTMOA*+CH+p, respectively. The diagonal dashed lines and the numbers labeling them correspond to the minimum, median, and maximum speed-ups of LTMOA*+CH+p over LTMOA* calculated over all instances solved by both algorithms. The query times of LTMOA*+CH+p are smaller than those of LTMOA* for most problem instances, with a maximum speed-up of more than an order of magnitude.

In general, the improvement of CHs is more substantial on problem instances with fewer numbers of objectives. Existing work on single-objective CHs [1] has shown that CHs perform

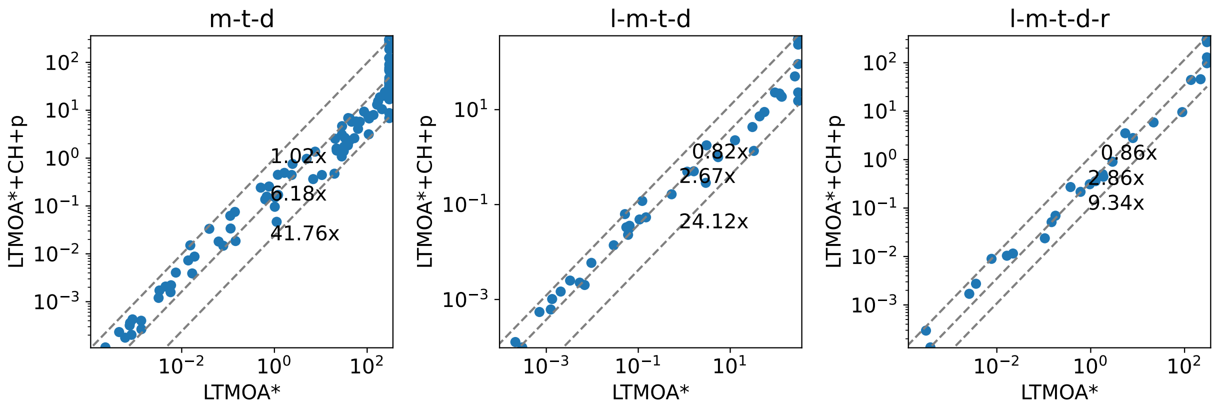


Figure 5.5: Runtimes of LTMOA*+CH+p and LTMOA* on road-network problem instances with different numbers of objectives.

better for graphs where many minimum-cost paths traverse a smaller set of “more important” states. This might explain why the speed-up of LTMOA*+CH+p decreases as the number of objectives increases because, intuitively, the set of “more important” states becomes larger as the number of objectives increases.

5.5 Summary

In this chapter, we generalized CHs to multi-objective search and introduced our CH-based approach for efficiently computing Pareto-frontiers. We proposed speed-up techniques for both the preprocessing and query phases of CHs that are specifically designed for multi-objective search. Our experimental results demonstrated the scalability of our approaches to large road networks with two objectives and orders-of-magnitude speed-ups in the query phase with all techniques combined. Our experimental results demonstrated the speed-ups of our approaches on problem instances with more than two objectives. These results validate the hypothesis that one can speed up multi-objective search algorithms via preprocessing techniques.

Chapter 6

Speeding up Multi-Objective Search via Data Structures for Efficient Dominance Checks

Existing multi-objective search algorithms, such as NAMOA*, EMOA*, and LTMOA*, perform dominance checks during the search to determine if a node can be pruned. For example, before expanding a node n, EMOA* and LTMOA* check if $Tr(\mathbf{g}(n))$ is weakly dominated by some vector in $\mathbf{G}^{\text{tr}}_{\text{cl}}(s(n))$, where $\mathbf{G}^{\text{tr}}_{\text{cl}}(s)$ is the set of undominated truncated $\mathbf{g}$-values of all expanded nodes that contain state s, and, if so, prune n. After expanding n, they also need to perform undominated set updates on $\mathbf{G}^{\text{tr}}_{\text{cl}}(s(n))$ to remove those vectors from $\mathbf{G}^{\text{tr}}_{\text{cl}}(s(n))$ that are weakly dominated by $Tr(\mathbf{g}(n))$ and add $Tr(\mathbf{g}(n))$ instead. Dominance checks and undominated set updates are performed frequently and intrinsically require iterating over sets of vectors. Therefore, they need to be performed efficiently. The differences in the runtimes of NAMOA*, EMOA*, and LTMOA* stems from how they implement dominance checks and interleave dominance checks with the search.

This chapter is based on [84].

In this chapter, we adapt well-known bucket-based data structures to our specific context of storing undominated vectors and performing dominance checks. We call them *bucket arrays* because vectors are slotted into different predefined buckets based on their values. The search algorithm can often determine if a bucket contains a vector that dominates a given vector without iterating over all vectors in the bucket.

Although LTMOA* propounds the use of arrays in practice, it can conceivably be used with other data structures that store undominated vectors for performing dominance checks. Not all of these data structures have been investigated so far. In this chapter, we evaluate LTMOA* not only with arrays but also with ND trees [39], a data structure that has been shown to generally outperform other data structures for maintaining sets of undominated vectors with respect to runtime.

This chapter is organized as follows: We begin with the background materials for our work in Section 6.1. Next, we provide a detailed description of bucket arrays in Section 6.2. We then provide the experimental results in Section 6.3 and our summary in Section 6.4.

6.1 Background

We first describe in Section 6.1.1 the data structures used by existing multi-objective search algorithms for dominance checks. Existing work has proposed several other data structures that could be used in multi-objective search but have so far been studied only in other contexts. We describe these data structures in Section 6.1.2.

6.1.1 Data Structures in Existing Multi-Objective Search Algorithms

Ren et al. [59] propose EMOA*, which uses AVL trees to store $\mathbf{G}^{\mathrm{tr}}_{\mathrm{cl}}(s)$ for each state s. Each vector in $\mathbf{G}^{\mathrm{tr}}_{\mathrm{cl}}(s)$ is stored in a tree node. All vectors stored in the left subtree of a tree node are lexicographically smaller than the vector stored in the tree node, and all vectors stored in the right subtree of the tree node are lexicographically larger than the vector stored in the tree node. When performing a dominance check, that is, checking if a given vector $\mathbf{v}$ is weakly dominated by some vector in a set of vectors $\mathbf{X}$, the algorithm starts at the root node of the AVL tree that stores $\mathbf{X}$ and traverses the AVL tree recursively. When reaching a node that stores a vector $\mathbf{v}'$ that is lexicographically no smaller than $\mathbf{v}$, the algorithm does not traverse its right subtree because the vectors stored in the right subtree are lexicographically larger than $\mathbf{v}'$ and hence cannot weakly dominate $\mathbf{v}$. Similarly, when performing an undominated set update, that is, removing from a set of vectors $\mathbf{X}$ the vectors that are dominated by a given vector $\mathbf{v}$ and adding $\mathbf{v}$ to $\mathbf{X}$, the algorithm starts at the root node of the AVL tree that stores $\mathbf{X}$ and traverses the AVL tree recursively to remove vectors that are dominated by $\mathbf{v}$. When reaching a node that contains a vector $\mathbf{v}'$ that is lexicographically no larger than $\mathbf{v}$, the algorithm does not traverse its left subtree because the vectors stored in the left subtree are lexicographically smaller than $\mathbf{v}'$ and hence cannot be dominated by $\mathbf{v}$. After removing all nodes storing dominated vectors, the algorithm adds a tree node containing $\mathbf{v}$ to the AVL tree and performs a rebalance operation to minimize the depth of the AVL tree. Intuitively, AVL trees should result in fewer vector comparisons for dominance checks and undominated set updates, but the time complexities of both operations remain $O(|\mathbf{G}^{\mathrm{tr}}_{\mathrm{cl}}(s)|)$. However, Ren et al. [59] show that the time complexity of dominance checks with AVL trees can be improved to $O(\log(|\mathbf{G}^{\mathrm{tr}}_{\mathrm{cl}}(s)|))$ when EMOA* is used to solve multi-objective search problem instances with three objectives.

Hernández et al. [36] propose using simple data structures, like arrays and linked lists, to store $\mathbf{G}^{\mathrm{tr}}_{\mathrm{cl}}(s)$ for LTMOA*. When performing dominance checks or undominated set updates, the search algorithm simply iterates over the entire array or linked list. Hernández et al. [36] show experimentally that using arrays to store $\mathbf{G}^{\mathrm{tr}}_{\mathrm{cl}}(s)$ results in $2.5\times$, $7.9\times$, and $13.5\times$ speed-ups over EMOA* in problem instances with three, four, and five objectives, respectively. They also show that, although AVL trees allow EMOA* to perform fewer vector comparisons, AVL trees have a larger runtime overhead than arrays because of the tree traversal and rebalance operations. Finally, they show that LTMOA* with linked lists slightly outperforms EMOA* but is outperformed by LTMOA* with arrays in terms of average runtime.

6.1.2 Data Structures in Other Related Work

Existing work has proposed several data structures for efficient dominance checks and undominated set updates in contexts that are different from multi-objective search, such as for multi-objective evolutionary algorithms [18]. Such data structures are known as *Pareto archives* and have yet to be evaluated for multi-objective search algorithms. They include quadtrees [49], M-fronts [18], and ND trees [39]. Among them, ND trees have been shown to be generally more efficient than the other data structures [39].

An ND tree is parameterized with a branching factor b and a maximum number L of vectors in each leaf node. Each non-leaf node of an ND tree has up to b child nodes but does not contain any vectors. Yet, each node n of an ND tree corresponds to a set of vectors $V(n)$. If n is a leaf node,

$V(n)$ is defined as the set of vectors that n contains. Otherwise, $V(n)$ is defined as the union of the sets $V(n')$ of all child nodes n' of n. Let $\mathbf{z}^*(n)$ and $\mathbf{z}_*(n)$ be the component-wise minimum and component-wise maximum of the vectors in $V(n)$, respectively. The ND tree maintains two vectors $\tilde{\mathbf{z}}^*(n)$ (with $\tilde{\mathbf{z}}^*(n) \preceq \mathbf{z}^*(n)$) and $\tilde{\mathbf{z}}_*(n)$ (with $\tilde{\mathbf{z}}_*(n) \succeq \mathbf{z}_*(n)$) to approximate $\mathbf{z}^*(n)$ and $\mathbf{z}_*(n)$, respectively.

When performing dominance checks for a given vector $\mathbf{v}$ over an ND tree, the algorithm starts at the root node and traverses the ND tree recursively. The algorithm often does not need to traverse the entire ND tree. For example, when reaching a node n whose $\tilde{\mathbf{z}}^*(n)$ does not weakly dominate $\mathbf{v}$, it does not need to check the subtree rooted at n because no vector in $V(n)$ weakly dominates $\mathbf{v}$. when reaching a node n whose $\tilde{\mathbf{z}}_*(n)$ weakly dominates $\mathbf{v}$, the algorithm returns true immediately without checking the subtree rooted at n because all vectors in $V(n)$ weakly dominate $\tilde{\mathbf{z}}_*(n)$ and hence weakly dominate $\mathbf{v}$.

When performing undominated set updates for a given vector $\mathbf{v}$ over an ND tree, the algorithm first removes all vectors that are dominated by $\mathbf{v}$ from the ND tree. It starts at the root node and traverses the ND tree recursively. Similar to the algorithm for dominance checks, it often does not need to traverse the entire ND tree. For example, when reaching a node n whose $\tilde{\mathbf{z}}^*(n)$ is dominated by $\mathbf{v}$, it can immediately remove the entire subtree rooted at n (without checking any of its nodes) because $\mathbf{v}$ dominates all vectors in $V(n)$. After removing all dominated vectors, the algorithm inserts $\mathbf{v}$ into the ND tree. It starts at the root node, greedily selects a child node n (that minimizes the Euclidean distance between $\mathbf{v}$ and the middle point of $\tilde{\mathbf{z}}^*(n)$ and $\tilde{\mathbf{z}}_*(n)$), and repeats this process until it reaches a leaf node n'. It then adds $\mathbf{v}$ to $V(n')$. If $|V(n')|$ becomes larger than L, the algorithm partitions $V(n')$ into b subsets and creates a child node of n' for each subset (that contains the subset).

Algorithm 13: Dominance Checks for Bucket Arrays

```
Input  : A bucket array $\mathbb{B} = [B_1, B_2 \dots B_n]$ and a vector $\mathbf{v}$
Output: Whether there exists a vector in $\mathbb{B}$ that weakly dominates $\mathbf{v}$
1 for each $B \in \mathbb{B}$ do
2     if $\mathbf{I}(B)$ does not weakly dominate $\mathbf{I}(\mathbf{v})$ then
3         continue
4     else if $I_i(B) < I_i(v)$ for all $i = 1, 2 \dots N-1$ then
5         return true
6     else if $\exists \mathbf{v}' \in B\ \mathbf{v}' \preceq \mathbf{v}$ then
7         return true
8 return false
```

6.2 Bucket Arrays

In this section, we describe the bucket array, a data structure for storing $\mathbf{G}^{\text{tr}}_{\text{cl}}(s)$, and how the `IsDominated` and `Update` functions work with it. Let N denote the number of objectives. Hence, the length of each vector in $\mathbf{G}^{\text{tr}}_{\text{cl}}(s)$ is $N-1$. A bucket array is an array of buckets, where each *bucket* contains an array of vectors (of length $N-1$). Each component of a vector in this bucket is within a predefined range. Let δ denote the *step* of value, which is a parameter of the bucket array. The *index* of a vector $\mathbf{v}$ is defined to be $\mathbf{I}(\mathbf{v}) = [\lfloor v_1/\delta \rfloor, \lfloor v_2/\delta \rfloor \dots \lfloor v_{N-1}/\delta \rfloor]$. A bucket B contains vectors with the same index, denoted as $\mathbf{I}(B)$. Thus, every vector $\mathbf{v}$ in bucket B satisfies

$$I_i(B) \cdot \delta \leq v_i < (I_i(B) + 1) \cdot \delta, \quad i = 1, 2 \dots N-1. \tag{6.1}$$

All buckets in a bucket array have different indices and are not empty. Like a node in an ND tree, we can easily estimate the component-wise minimum and component-wise maximum of all vectors in a bucket. As we will show shortly, this helps us to develop efficient algorithms for dominance checks and undominated set updates. Moreover, by storing buckets in an array instead of a tree, we avoid the runtime overhead of tree operations.

6.2.1 Dominance Checks

Algorithm 13 shows the pseudocode for dominance checks of a given vector $\mathbf{v}$ over a bucket array $\mathbb{B} = [B_1, B_2 \ldots B_n]$. Each bucket $B \in \mathbb{B}$ maintains information that indicates the range of vectors that it contains. This information can be used to speed up the dominance checks. Algorithm 13 iterates over each bucket B in the bucket array. We distinguish the following cases:

(Case 1) $\mathbf{I}(B)$ does not weakly dominate $\mathbf{I}(\mathbf{v})$ (Line 2). We have $I_i(B) > I_i(\mathbf{v})$ for some i. Because $I_i(B)$ and $I_i(\mathbf{v})$ are both integers, we have $I_i(B) \geq I_i(\mathbf{v}) + 1$. Consider any vector $\mathbf{u} \in B$. From Eq. 6.1, we have $v_i < (I_i(\mathbf{v}) + 1) \cdot \delta \leq I_i(B) \cdot \delta \leq u_i$. Therefore, $\mathbf{u}$ does not weakly dominate $\mathbf{v}$. The algorithm does not need to check any vector in B (Line 3) because no vector in B can weakly dominate $\mathbf{v}$.

(Case 2) $\mathbf{I}(B)$ satisfies $I_i(B) < I_i(\mathbf{v})$ for all $i = 1, 2 \ldots N-1$ (Line 4). We have $I_i(B)+1 \leq I_i(\mathbf{v})$for all i because $I_i(B)$ and $I_i(\mathbf{v})$ are both integers. Consider any vector $\mathbf{u} \in B$. From Eq. 6.1, we have $u_i < (I_i(B) + 1) \cdot \delta \leq I_i(\mathbf{v}) \cdot \delta \leq v_i$ for all i. Therefore, $\mathbf{u}$ weakly dominates $\mathbf{v}$. The algorithm returns $true$ immediately (Line 5) because there exists a vector that weakly dominates $\mathbf{v}$.

(Case 3) Otherwise, the algorithm iterates over the vectors in B and returns $true$ if it finds a vector that weakly dominates $\mathbf{v}$ (Lines 6-7).

Algorithm 13 needs to check the vectors inside a bucket only in the last case. Therefore, using bucket arrays can reduce the number of vector comparisons that need to be performed.

Example 12. *Consider the set of truncated* $\mathbf{g}$*-values* $\mathbf{G}^{tr}_{cl}(s) = \{\mathbf{v}_1 = [10, 260], \mathbf{v}_2 = [15, 220], \mathbf{v}_3 = [75, 160], \mathbf{v}_4 = [85, 140], \mathbf{v}_5 = [110, 80]\}$ *in LTMOA*. Assuming* $\delta = 100$*, the* *bucket array for storing* $\mathbf{G}^{tr}_{cl}(s)$ *has three buckets, namely, bucket* B_1 *containing* $\{\mathbf{v}_1, \mathbf{v}_2\}$ *with* $\mathbf{I}(B_1) = [0, 2]$*, bucket* B_2 *containing* $\{\mathbf{v}_3, \mathbf{v}_4\}$ *with* $\mathbf{I}(B_2) = [0, 1]$*, and bucket* B_3 *containing* $\{\mathbf{v}_5\}$ *with* $\mathbf{I}(B_3) = [1, 0]$.

Assume that LTMOA extracts a node from* $Open$ *that contains state* s *and whose truncated* $\mathbf{g}$*-value is* $\mathbf{v}_6 = [180, 70]$*. We have* $\mathbf{I}(\mathbf{v}_6) = [1, 0]$*. When performing the dominance checks, Algorithm 13 first checks buckets* B_1 *and* B_2*. Because Case 1 holds for both* B_1 *and* B_2*, Algorithm 13 does not need to check any vectors in* B_1 *and* B_2*. Algorithm 13 then checks* B_3*. Because Case 3 trivially holds for* B_3*, Algorithm 13 checks if* $\mathbf{v}_5$ *in* B_3 *weakly dominates* v_6*. Since this is not the case, Algorithm 13 eventually reaches Line 8 and returns* $false$*.*

In this example, Algorithm 13 needs four vector comparisons (including three vector comparisons between the indexes of buckets and $\mathbf{I}(\mathbf{v})$*) for the dominance check. If* $\mathbf{G}^{tr}_{cl}(s)$ *were stored in an array, we would need five vector comparisons. Although using the bucket array in this example saves only one vector comparison, bucket arrays can substantially reduce the number of vector comparisons in practice, as we will show in the experimental evaluation.*

6.2.2 Undominated Set Updates

Algorithm 14 shows the pseudocode for undominated set updates with bucket arrays. Given a bucket array $\mathbb{B}$ and a vector $\mathbf{v}$, Algorithm 14 adds $\mathbf{v}$ to $\mathbb{B}$ and removes all vectors from $\mathbb{B}$ that are dominated by $\mathbf{v}$. It first iterates over all buckets in $\mathbb{B}$ to check if there exists a bucket B whose index is $\mathbf{I}(\mathbf{v})$. If so, the algorithm adds $\mathbf{v}$ to B (Line 2). Otherwise, it creates a new bucket with index $\mathbf{I}(\mathbf{v})$, adds $\mathbf{v}$ to this bucket, and adds this bucket to $\mathbb{B}$ (Lines 4-6).

Algorithm 14: Undominated Set Updates for Bucket Arrays

```
Input  : A bucket array $\mathbb{B} = [B_1, B_2 \ldots B_n]$ and a vector $\mathbf{v}$
Output: Updated $\mathbb{B}$
1  if $\exists B \in \mathbb{B}\ \mathbf{I}(B) = \mathbf{I}(\mathbf{v})$ then
2      add $\mathbf{v}$ to $B$
3  else
4      $B \leftarrow$ a new bucket with $\mathbf{I}(B) = \mathbf{I}(\mathbf{v})$
5      add $\mathbf{v}$ to $B$
6      add $B$ to $\mathbb{B}$
7  for each $B \in \mathbb{B}$ do
8      if $\mathbf{I}(\mathbf{v})$ does not weakly dominate $\mathbf{I}(B)$ then
9          continue
10     else if $I_i(\mathbf{v}) < I_i(B)$ for all $i = 1, 2 \ldots N-1$ then
11         remove $B$ from $\mathbb{B}$
12     else
13         remove vectors that are dominated by $\mathbf{v}$ from $B$
14         if $B$ is empty then
15             remove $B$ from $\mathbb{B}$
16 return $\mathbb{B}$
```

The algorithm then iterates over each bucket B in $\mathbb{B}$ to remove all vectors that are dominated by $\mathbf{v}$. We distinguish the following cases:

(Case 1) $\mathbf{I}(\mathbf{v})$ does not weakly dominate $\mathbf{I}(B)$ (Line 8). We have $I_i(B) < I_i(\mathbf{v})$ for some i. Because $I_i(B)$ and $I_i(\mathbf{v})$ are both integers, we have $I_i(B) + 1 \leq I_i(\mathbf{v})$. Consider any vector $\mathbf{u} \in B$. From Eq. 6.1, we have $u_i < (I_i(B) + 1) \cdot \delta \leq I_i(\mathbf{v}) \cdot \delta \leq v_i$. Therefore, $\mathbf{u}$ is not weakly dominated by $\mathbf{v}$. The algorithm does not need to check any vector in B (Line 9) because none of these vectors can be dominated by $\mathbf{v}$.

(Case 2) $I_i(\mathbf{v}) < I_i(B)$ for all $i = 1, 2 \ldots N-1$ (Line 10). We have $I_i(\mathbf{v}) + 1 \leq I_i(B)$ for all i because $I_i(B)$ and $I_i(\mathbf{v})$ are both integers. Consider any vector $\mathbf{u} \in B$. From Eq. 6.1, we have $v_i < (I_i(\mathbf{v}) + 1) \cdot \delta \leq I_i(B) \cdot \delta \leq u_i$ for all i. Therefore, $\mathbf{v}$ dominates $\mathbf{u}$. The algorithm removes B from the bucket array without needing to check any vector in B because all of its vectors are dominated by $\mathbf{v}$ (Line 11). Removing B from the bucket array can be done in constant time by first switching B with the last bucket in the bucket array and then removing B from the back of the bucket array.

(Case 3) Otherwise, the algorithm iterates over the vectors in B and removes the ones that are dominated by $\mathbf{v}$ (Line 13). If B becomes empty, the algorithm removes B from the bucket array (Lines 14-15).

Like Algorithm 13, Algorithm 14 needs to check the vectors inside a bucket only in the last case.

Example 13. *Continue Example 12 and assume that LTMOA* expands the node that contains state s and whose truncated $\mathbf{g}$-value is $\mathbf{v_6} = [180, 70]$. Algorithm 14 then adds $\mathbf{v_6}$ to the bucket array (that stores $\mathbf{G}^{tr}_{cl}(s)$) and removes all vectors from the bucket array that are dominated by $\mathbf{v_6}$.*

Algorithm 14 first iterates over the bucket array and finds that bucket B_3, which contains vector $\mathbf{v}_5 = [110, 80]$, has the same index as $\mathbf{v_6}$. Therefore, it adds $\mathbf{v_6}$ to B_3. Algorithm 14 then removes all vectors from the bucket array that are dominated by $\mathbf{v_6}$: Because Case 1 holds for both B_1 and B_2, Algorithm 14 does not need to check any vectors in B_1 and B_2. It then checks B_3. Because Case 3 holds for B_3, Algorithm 14 checks if $\mathbf{v}_5$ is dominated by $\mathbf{v}_6$ and, since this is not the case, does not remove it from the bucket array.

6.3 Experimental Evaluation

In this section, we evaluate LTMOA* with different data structures for storing vectors on problem instances with three to five objectives. We evaluate LTMOA* with arrays (+array), ND trees (+NDTree), and bucket arrays (+bucket). We implemented all variants of LTMOA* in C++.[1] We

[1]https://github.com/HanZhang39/MultiObjectiveSearch

δ	#solved	t_{avg}	t_{max}	avg #comp	max #comp
1,000	23	12.04	213.68	1,508M	24,825M
10,000	**27**	2.72	43.02	440M	7,063M
20,000	**27**	1.68	24.96	**318M**	**5,061M**
50,000	26	**1.24**	**17.36**	370M	5,924M
100,000	26	1.29	18.58	568M	9,145M
200,000	26	1.64	23.91	862M	12,691M

Table 6.1: Numbers of solved problem instances (#solved), average and maximum runtimes (t_{avg} and t_{max}, both in seconds), and average and maximum numbers of vector comparisons (avg and max #comp) for LTMOA*+bucket with different δ-values on problem instances with four objectives (l-m-t-d).

obtained the original ND tree implementation from Jaszkiewicz and Lust [39] and integrated it into our code base. The runtime limit for solving each problem instance was five minutes.

We use the NY road network (264,346 states and 733,846 edges) from the 9th DIMACS Implementation Challenge: Shortest Path.[2] The NY road network has two objectives, namely travel distance (d) and travel time (t), available in the benchmark. Additionally, we use the economic cost (m) [54], the number of edges (l) [47], and a random integer between 1 and 100 (r) [36] as the third, fourth, and fifth objectives, respectively. We use the 100 pairs of start and goal states used by Sedeño-Noda and Colebrook [65] and Ahmadi et al. [2] for our problem instances.

6.3.1 Comparing Bucket Arrays with Different Parameters

We first compare LTMOA*+bucket with different δ-values on the first 30 problem instances of the NY road network with four objectives (l-m-t-d). We evaluate six δ-values: 1,000, 10,000, 20,000, 50,000, 100,000, and 200,000. A smaller δ-value results in fewer vectors in each bucket and hence more buckets in a bucket array. With a too small δ-value,each bucket in a bucket array can contain only one vector. With a too large δ-value, a bucket array can contain only one bucket

[2]http://www.diag.uniroma1.it/challenge9/download.shtml

that contains all the vectors. In either case, using bucket arrays will not reduce the number of vector comparisons for dominance checks. Therefore, choosing a suitable δ-value is important.

Table 6.1 shows the numbers of solved problem instances, average and maximum runtimes, and average and maximum numbers of vector comparisons for LTMOA*+bucket with different δ-values. All averages and maximums are calculated over instances that are solved by all algorithms. Our results show that the average and maximum runtimes of LTMOA*+bucket decrease as the δ-value increases from $1{,}000$ to $50{,}000$ and then increase as the δ-value increases from $50{,}000$ to $200{,}000$. We observe similar trends in the number of solved instances and vector comparisons with the differences that the best performances are achieved with δ-values other than $50{,}000$. For the rest of this chapter, we use $\delta = 20{,}000$ for LTMOA*+bucket because it results in the highest number of solved problem instances and the smallest number of vector comparisons.

6.3.2 Comparing Different Data Structures

In this section, we compare LTMOA*+array, LTMOA*+NDTree, and LTMOA*+bucket on problem instances with different numbers of objectives. For LTMOA*+NDTree, we ran experiments with all combinations of $b \in \{5, 10, 20\}$ and $L \in \{20, 40, 80\}$ on the 30 problem instances used in the previous section. We then chose the parameter combination with the smallest runtime, namely $b = 5$ and $L = 20$. As we have explained in the previous section, we use $\delta = 20{,}000$ for LTMOA*+bucket.

Table 6.2 shows the results for the different variants of LTMOA*. Although ND trees result in fewer vector comparisons than arrays in many cases, they do not improve the runtime of LT-MOA* due to the runtime overhead. Compared to both LTMOA*+array and LTMOA*+NDTree,

Algorithm	#solved	t_{avg}	$t_{\max}$	avg #comp	max #comp
3 objectives (m-t-d)					
LTMOA*+array	**77**	7.27	**59.50**	1,855M	12,942M
LTMOA*+NDTree	61	43.75	258.80	2,431M	11,333M
LTMOA*+bucket	**77**	**6.81**	**59.50**	**758M**	**5,907M**
4 objectives (l-m-t-d)					
LTMOA*+array	38	20.54	123.71	21,267M	133,766M
LTMOA*+NDTree	36	44.07	298.38	4,231M	17,770M
LTMOA*+bucket	**39**	**8.09**	**51.62**	**2,074M**	**12,543M**
5 objectives (l-m-t-d-r)					
LTMOA*+array	23	12.07	133.70	18,766M	235,553M
LTMOA*+NDTree	22	16.12	206.54	8,198M	21,658M
LTMOA*+bucket	**27**	**2.82**	**32.05**	**1,469M**	**17,265M**

Table 6.2: Numbers of solved problem instances (#solved), average and maximum runtimes (t_{avg} and $t_{\max}$, both in seconds), and average and maximum numbers of vector comparisons (avg and max #comp) for LTMOA* with different data structures on problem instances with different numbers of objectives.

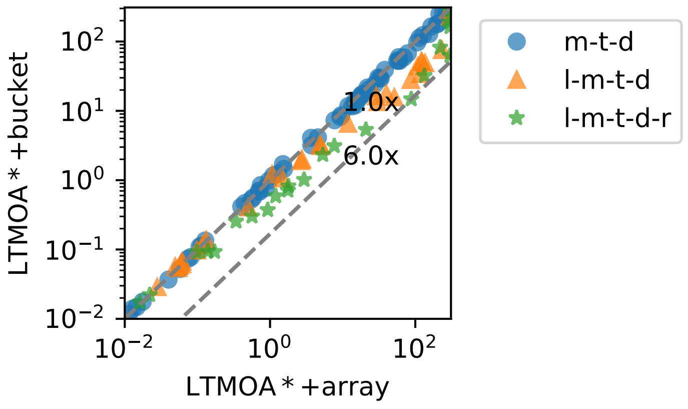


Figure 6.1: Runtimes (in seconds) of LTMOA*+array versus LTMOA*+bucket on individual problem instances. The dashed diagonal lines correspond to different speed-ups.

LTMOA*+bucket performs fewer vector comparisons. LTMOA*+bucket also has the smallest average runtimes on problem instances with three, four, and five objectives. The speed-ups of LTMOA*+bucket over LTMOA*+array in average runtimes for problem instances with three, four, and five objectives are $1.06\times$, $2.54\times$, and $4.28\times$, respectively, which shows that bucket arrays become more beneficial to the runtime performance as the number of objectives increases.

Figure 6.1 shows the runtimes of LTMOA*+array and LTMOA*+bucket on individual problem instances. We use different markers for different numbers of objectives. The x- and the y-axis correspond to the runtimes of LTMOA*+array and LTMOA*+bucket, respectively. The two diagonal dashed lines and the numbers along them denote the $1\times$ speed-up (that is, LTMOA*+array and LTMOA*+bucket have the same runtime) and the maximum speed-up of LTMOA*+bucket. LTMOA*+bucket runs faster than LTMOA* on most problem instances with four or five objectives. The speed-up over LTMOA*+array is up to 6 times.

6.4 Summary

In this chapter, we proposed bucket arrays, a data structure for storing undominated vectors and performing dominance checks in multi-objective search algorithms. In a bucket array, vectors are slotted into predefined buckets based on their values. We can often perform dominance checks and undominated set updates, which are two important operations of multi-objective search algorithms, over a bucket array without iterating over the vectors in each bucket. Therefore, bucket arrays can reduce the number of vector comparisons that multi-objective search algorithms need to perform. For example, we empirically showed that enhancing LTMOA* with bucket arrays yields a speed-up of $4.3\times$ on average for a set of problem instances with five objectives. Our experimental results validate the hypothesis that multi-objective search algorithms can be sped up by using data structures for efficient dominance checks.

Chapter 7

Conclusions

Multi-objective search is important for many applications. Existing work has generalized best-first (single-objective) search algorithms, such as A*, to multi-objective search. However, the size of the Pareto frontier can be exponential in the size of the graph being searched, which often makes existing multi-objective search algorithms very time-consuming. Multi-objective search algorithms also need to perform dominance checks frequently, which further slows down the search. Therefore, existing multi-objective search algorithms do not scale well to large graphs or many objectives.

In this dissertation, we investigated techniques that have been used to speed up single-objective search algorithms and showed that one can develop speed-up techniques for multi-objective search algorithms based on the ideas behind them. More specifically:

1. In Chapter 3, we investigated speeding up multi-objective search by trading off solution quality with efficiency. We introduced A*pex, an approximate multi-objective search algorithm that finds an ε-approximate Pareto frontier for a given approximation factor ε. A*pex builds upon PP-A* but (1) makes PP-A* more efficient for bi-objective search and (2) generalizes it from two objectives to any number of objectives. We proved the correctness and

completeness of A*pex. Our experimental results showed the efficiency advantage of A*pex over state-of-the-art multi-objective search algorithms.

2. In Chapter 4, we investigated speeding up multi-objective search by using anytime search. We introduced A-A*pex, an anytime approximate multi-objective search algorithm that builds upon A*pex. A-A*pex runs A*pex iteratively to compute better and better approximate Pareto frontiers as long as time allows. In each iteration, A-A*pex can either reuse its previous search effort or restart its search from scratch. Our experimental results showed that A-A*pex computes approximate frontiers with smaller approximation errors than state-of-the-art multi-objective search algorithms for short runtimes.

3. In Chapter 5, we investigated speeding up multi-objective search by using preprocessing techniques. We introduced a Contraction-Hierarchy(CH)-based approach for efficiently computing Pareto frontiers for multi-objective search. Additionally, we proposed speed-up techniques for both the preprocessing and query phases that are specifically designed for multi-objective search. Our experimental results showed the scalability of our approaches to large road networks and orders-of-magnitude speed-ups in the query phase when all speed-up techniques are combined.

4. In Chapter 6, we investigated speeding up multi-objective search by using efficient data structures for dominance checks. We adapted bucket-based data structures to our specific context of storing undominated vectors and performing dominance checks, resulting in a new data structure, called bucket array. In a bucket array, vectors are slotted into different predefined buckets based on their values. We can often perform dominance checks and undominated set updates, which are two important operations in multi-objective search

algorithms, without iterating over the vectors in each bucket of a bucket array. Therefore, bucket arrays reduce the number of vector comparisons that multi-objective search algorithms need to perform. Our experimental results showed that enhancing LTMOA* with bucket arrays can yield a speed-up of $4.3\times$ on average for problem instances with five objectives.

Therefore, we validated the hypothesis that multi-objective search algorithms can be sped up by applying insights gained from single-objective search algorithms after proper generalization.

More importantly, these insights are not trivially applied to multi-objective search. The techniques we develop based on these insights are often quite different from the techniques in single-objective search based on the same insights: A*pex and A-A*pex use apex-path pairs and merging operations, which are not in any single-objective search algorithm. The partial expansion technique for our CH-based query algorithm exploits parallel edges in the CHs for multi-objective search, which are not in the CHs for single-objective search. Finally, we use bucket arrays to speed up dominance checks, an operation that single-objective search algorithms do not perform.

An interesting direction for future work is to combine the algorithms or techniques proposed in this dissertation. Our preliminary study shows that some of such combinations require careful design: (1) When combining A*pex (or A-A*pex) with CHs, it is unclear how to combine A*pex with the partial expansion technique (Section 5.2.2.2). This requires a careful design of the node generation process to perform both the merge operations of apex-path pairs and the partial expansions. It is also unclear if one can speed up the preprocessing phase for CHs if one is interested in only computing ε-approximate Pareto frontiers for some given ε-value in the query phase. (2)

We have tried A*pex with bucket arrays, but the preliminary results show that the speed-up of bucket arrays is much less substantial than in LTMOA*. This is because, in A*pex, iterating over apex-path pairs in $Open$ for merging (Lines 29-34 of Algorithm 6) can be more time-consuming than dominance checks. Currently, bucket arrays do not consider this part. It is interesting to investigate if bucket arrays (or other data structures) can be used to speed up this part of A*pex as well.

Additionally, we list some techniques that have been used to speed up single-objective search algorithms and can be explored for multi-objective search:

1. **Parallel search:** Researchers have proposed several parallel (single-objective) search algorithms, including Parallel Retracting A* (PRA*) [21], Hash Distributed A* (HDA*) [40], and Parallel A* for Slow Expansions (PA*SE) [52]. Some existing work has investigated the parallelization of multi-objective search algorithms. However, this direction still needs to be explored further. Ahmadi et al. [2] proposed a bi-directional bi-objective search algorithm BOBA*, which is based on BOA*. BOBA* runs two bi-objective searches in parallel, one from the start state and one from the goal state. Consequently, this approach of parallelization can make use of only two cores. Our recent work [75] also proposed a technique to parallelize dominance checks at the instruction level. We use Single Instruction/Multiple Data (SIMD) instructions to perform vector comparisons in parallel and speed up LTMOA* by up to 7 times. However, this approach does not make use of multiple cores.

2. **Incremental search:** Single-objective search algorithms that reuse information from prior searches of similar problem instances to speed up the search include incremental search algorithms, such as D* Lite [42], Adaptive A* [41], and Generalized Adaptive A* [71]. Existing

work [58] has generalized D* Lite to multi-objective search. However, their experimental results only compared their proposed algorithms to NAMOA* (which does not use dimensionality reduction), and the empirical improvement in runtime is not substantial. A direction for future work is to generalize other incremental search algorithms to multi-objective search, such as Adaptive A*, which uses a very different approach from D* Lite.

3. **Suboptimal search:** Existing bounded-suboptimal (single-objective) search algorithms, including WA* [53], focal search [50], and EES [72], do not expand nodes in a best-first order but still guarantee to find bounded-suboptimal solutions. The node expansion orders of these algorithms can guide the search and are often key to finding solutions faster. As we have shown, the techniques used by A*pex to speed up multi-objective search are very different from those of bounded-suboptimal search algorithms. A*pex still expands nodes in a best-first order. Intuitively, expanding nodes in a best-first order is not required for finding an approximate Pareto frontier. Thus, different node expansion orders might be able to guide multi-objective search algorithms, such as A*pex, to find solutions faster, which requires a careful design of a good node expansion order, for example, one that is compatible with techniques like dimensionality reduction.

Bibliography

[1] Ittai Abraham, Amos Fiat, Andrew V Goldberg, and Renato F Werneck. "Highway Dimension, Shortest Paths, and Provably Efficient Algorithms". In: *ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 2010, pp. 782–793.

[2] Saman Ahmadi, Guido Tack, Daniel Harabor, and Philip Kilby. "Bi-Objective Search with Bi-Directional A*". In: *Symposium on Combinatorial Search (SoCS)*. 2021, pp. 142–144.

[3] Saman Ahmadi, Guido Tack, Daniel Harabor, and Philip Kilby. "Enhanced Methods for the Weight Constrained Shortest Path Problem: Constrained Path Finding Meets Bi-Objective Search". In: *arXiv:2207.14744* (2022).

[4] Saman Ahmadi, Guido Tack, Daniel Harabor, and Philip Kilby. "Vehicle Dynamics in Pickup-and-Delivery Problems Using Electric Vehicles". In: *International Conference on Principles and Practice of Constraint Programming (CP)*. 2021, p. 11.

[5] Saman Ahmadi, Guido Tack, Daniel Harabor, and Philip Kilby. "Weight Constrained Path Finding with Bidirectional A*". In: *Symposium on Combinatorial Search (SoCS)*. 2022, pp. 2–10.

[6] Daniel Bachmann, Fritz Bökler, Jakob Kopec, Kira Popp, Björn Schwarze, and Frank Weichert. "Multi-Objective Optimisation Based Planning of Power-Line Grid Expansions". In: *ISPRS International Journal of Geo-Information* 7.7 (2018), p. 258.

[7] Moritz Baum, Julian Dibbelt, Andreas Gemsa, Dorothea Wagner, and Tobias Zündorf. "Shortest Feasible Paths with Charging Stops for Battery Electric Vehicles". In: *SIGSPATIAL International Conference on Advances in Geographic Information Systems*. 2015, pp. 1–10.

[8] Jur van den Berg, Rajat Shah, Arthur Huang, and Kenneth Y. Goldberg. "Anytime Nonparametric A*". In: *AAAI Conference on Artificial Intelligence (AAAI)*. 2011, pp. 105–111.

[9] Nicola Beume, Boris Naujoks, and Michael Emmerich. “SMS-EMOA: Multiobjective Selection Based on Dominated Hypervolume”. In: *European Journal of Operational Research* 181.3 (2007), pp. 1653–1669.

[10] Thomas Breugem, Twan Dollevoet, and Wilco van den Heuvel. “Analysis of FPTASes for the Multi-Objective Shortest Path Problem”. In: *Computers & Operations Research* 78 (2017), pp. 44–58.

[11] Andrés Bronfman, Vladimir Marianov, Germán Paredes-Belmar, and Armin Lüer-Villagra. “The Maximin HAZMAT Routing Problem”. In: *European Journal of Operational Research* 241.1 (2015), pp. 15–27.

[12] Ethan Burns, Matthew Hatem, Michael Leighton, and Wheeler Ruml. “Implementing Fast Heuristic Search Code”. In: *Symposium on Combinatorial Search (SoCS).* 2012, pp. 25–32.

[13] Liron Cohen, Matias Greco, Hang Ma, Carlos Hernández, Ariel Felner, T. K. Satish Kumar, and Sven Koenig. “Anytime Focal Search with Applications”. In: *International Joint Conferences on Artificial Intelligence (IJCAI).* 2018, pp. 1434–1441.

[14] Liron Cohen, Tansel Uras, Shiva Jahangiri, Aliyah Arunasalam, Sven Koenig, and T. K. Satish Kumar. “The FastMap Algorithm for Shortest Path Computations”. In: *International Joint Conferences on Artificial Intelligence (IJCAI).* 2018, pp. 1427–1433.

[15] Mansoor Davoodi. “Bi-Objective Path Planning Using Deterministic Algorithms”. In: *Robotics and Autonomous Systems* 93 (2017), pp. 105–115.

[16] Kalyanmoy Deb and Himanshu Jain. “An Evolutionary Many-Objective Optimization Algorithm using Reference-Point-Based Nondominated Sorting Approach, Part I: Solving Problems with Box Constraints”. In: *IEEE Transactions on Evolutionary Computation* 18.4 (2013), pp. 577–601.

[17] Daniel Delling and Dorothea Wagner. “Pareto Paths with SHARC”. In: *International Symposium on Experimental Algorithms (SEA).* 2009, pp. 125–136.

[18] Martin Drozdik, Youhei Akimoto, Hernán Aguirre, and Kiyoshi Tanaka. “Computational Cost Reduction of Nondominated Sorting Using the M-Front”. In: *IEEE Transactions on Evolutionary Computation* 19.5 (2014), pp. 659–678.

[19] Matthias Ehrgott. *Multicriteria Optimization (2nd ed.)* Springer, 2005.

[20] Michael TM Emmerich and André H Deutz. “A Tutorial on Multiobjective Optimization: Fundamentals and Evolutionary Methods”. In: *Natural Computing* 17 (2018), pp. 585–609.

[21] Matthew Evett, James Hendler, Ambuj Mahanti, and Dana Nau. “PRA*: Massively Parallel Heuristic Search”. In: *Journal of Parallel and Distributed Computing* 25.2 (1995), pp. 133–143.

[22] Ariel Felner, Meir Goldenberg, Guni Sharon, Roni Stern, Tal Beja, Nathan Sturtevant, Jonathan Schaeffer, and Robert Holte. "Partial-Expansion A* with Selective Node Generation". In: *AAAI Conference on Artificial Intelligence (AAAI).* 2012, pp. 471–477.

[23] Mengyu Fu, Alan Kuntz, Oren Salzman, and Ron Alterovitz. "Toward Asymptotically-Optimal Inspection Planning via Efficient Near-Optimal Graph Search". In: *Robotics: Science and Systems (RSS).* 2019.

[24] Mengyu Fu, Oren Salzman, and Ron Alterovitz. "Computationally-Efficient Roadmap-Based Inspection Planning via Incremental Lazy Search". In: *IEEE International Conference on Robotics and Automation (ICRA).* 2021, pp. 7449–7456.

[25] Stefan Funke and Sabine Storandt. "Polynomial-Time Construction of Contraction Hierarchies for Multi-Criteria Objectives". In: *Workshop on Algorithm Engineering and Experiments (ALENEX).* 2013, pp. 41–54.

[26] Virginie Gabrel and Daniel Vanderpooten. "Enumeration and Interactive Selection of Efficient Paths in a Multiple Criteria Graph for Scheduling an Earth Observing Satellite". In: *European Journal of Operational Research* 139.3 (2002), pp. 533–542.

[27] Robert Geisberger, Moritz Kobitzsch, and Peter Sanders. "Route Planning with Flexible Objective Functions". In: *Workshop on Algorithm Engineering and Experiments (ALENEX).* 2010, pp. 124–137.

[28] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. "Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks". In: *International Workshop on Experimental and Efficient Algorithms.* 2008, pp. 319–333.

[29] Florian Geißer, Patrik Haslum, Sylvie Thiébaux, and Felipe Trevizan. "Admissible Heuristics for Multi-Objective Planning". In: *International Conference on Automated Planning and Scheduling (ICAPS).* 2022, pp. 100–109.

[30] Andrew V Goldberg and Chris Harrelson. "Computing the Shortest Path: A* Search Meets Graph Theory." In: *ACM-SIAM Symposium on Discrete Algorithms (SODA).* 2005, pp. 156–165.

[31] Boris Goldin and Oren Salzman. "Approximate Bi-Criteria Search by Efficient Representation of Subsets of the Pareto-Optimal Frontier". In: *International Conference on Automated Planning and Scheduling (ICAPS).* 2021, pp. 149–158.

[32] Gabriel Y Handler and Israel Zang. "A Dual Algorithm for the Constrained Shortest Path Problem". In: *Networks* 10.4 (1980), pp. 293–309.

[33] Eric A Hansen and Rong Zhou. "Anytime Heuristic Search". In: *Journal of Artificial Intelligence Research* 28 (2007), pp. 267–297.

[34] Daniel Harabor and Peter Stuckey. “Forward Search in Contraction Hierarchies”. In: *Symposium on Combinatorial Search (SoCS).* 2018, pp. 55–62.

[35] Carlos U. Hernandez, William Yeohz, Jorge A. Baier, Han Zhang, Luis Suazoy, and Sven Koenig. “A Simple and Fast Bi-Objective Search Algorithm”. In: *International Conference on Automated Planning and Scheduling (ICAPS).* 2020, pp. 143–151.

[36] Carlos Hernández, William Yeoh, Jorge A Baier, Ariel Felner, Oren Salzman, Han Zhang, Shao-Hung Chan, and Sven Koenig. “Multi-Objective Search via Lazy and Efficient Dominance Checks”. In: *International Joint Conferences on Artificial Intelligence (IJCAI).* 2023, pp. 7223–7230.

[37] Carlos Hernández, William Yeoh, Jorge A. Baier, Han Zhang, Luis Suazo, Sven Koenig, and Oren Salzman. “Simple and Efficient Bi-Objective Search Algorithms via Fast Dominance Checks”. In: *Artificial Intelligence* 314 (2023), p. 103807.

[38] Daniel Jackson, H Estler, and Derek Rayside. “The Guided Improvement Algorithm for Exact, General-Purpose, Many-Objective Combinatorial Optimization”. In: *CSAIL Technical Reports* (2009).

[39] Andrzej Jaszkiewicz and Thibaut Lust. “ND-Tree-Based Update: a Fast Algorithm for the Dynamic Nondominance Problem”. In: *IEEE Transactions on Evolutionary Computation* 22.5 (2018), pp. 778–791.

[40] Akihiro Kishimoto, Alex Fukunaga, and Adi Botea. “Scalable, Parallel Best-First Search for Optimal Sequential Planning”. In: *International Conference on Automated Planning and Scheduling (ICAPS).* 2009, pp. 201–208.

[41] Sven Koenig and Maxim Likhachev. “A New Principle for Incremental Heuristic Search: Theoretical Results.” In: *International Conference on Automated Planning and Scheduling (ICAPS).* 2006, pp. 402–405.

[42] Sven Koenig and Maxim Likhachev. “Fast Replanning for Navigation in Unknown Terrain”. In: *IEEE Transactions on Robotics* 21.3 (2005), pp. 354–363.

[43] Maxim Likhachev, Geoffrey J Gordon, and Sebastian Thrun. “ARA*: Anytime A* with Provable Bounds on Sub-Optimality”. In: *Advances in Neural Information Processing Systems (NIPS).* 2003.

[44] Dean H Lorenz and Danny Raz. “A Simple Efficient Approximation Scheme for the Restricted Shortest Path Problem”. In: *Operations Research Letters* 28.5 (2001), pp. 213–219.

[45] Hang Ma, Craig Tovey, Guni Sharon, T. K. Satish Kumar, and Sven Koenig. “Multi-Agent Path Finding with Payload Transfers and the Package-Exchange Robot-Routing problem”. In: *AAAI Conference on Artificial Intelligence (AAAI).* 2016, pp. 3166–3173.

[46] Lawrence Mandow and José Luis Pérez De La Cruz. “A New Approach to Multiobjective A* Search”. In: *International Joint Conferences on Artificial Intelligence (IJCAI)*. 2005, pp. 218–223.

[47] Pedro Maristany de las Casas, Luitgard Kraus, Antonio Sedeño-Noda, and Ralf Borndörfer. “Targeted Multiobjective Dijkstra Algorithm”. In: *Networks* 82.3 (2023), pp. 277–298.

[48] Robert Morris, Corina S Pasareanu, Kasper Luckow, Waqar Malik, Hang Ma, T. K. Satish Kumar, and Sven Koenig. “Planning, Scheduling and Monitoring for Airport Surface Operations”. In: *AAAI-16 Workshop on Planning for Hybrid Systems*. 2016.

[49] Sanaz Mostaghim and Jürgen Teich. “Quad-Trees: A Data Structure for Storing Pareto Sets in Multiobjective Evolutionary Algorithms with Elitism”. In: *Evolutionary Multiobjective Optimization: Theoretical Advances and Applications*. Springer, 2005, pp. 81–104.

[50] Judea Pearl and Jin H Kim. “Studies in Semi-Admissible Heuristics”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 4 (1982), pp. 392–399.

[51] Patrice Perny and Olivier Spanjaard. “Near Admissible Algorithms for Multiobjective Search”. In: *European Conference on Artificial Intelligence (ECAI)*. 2008, pp. 490–494.

[52] Mike Phillips, Maxim Likhachev, and Sven Koenig. “PA*SE: Parallel A* for Slow Expansions”. In: *International Conference on Automated Planning and Scheduling (ICAPS)*. 2014, pp. 208–216.

[53] Ira Pohl. “Heuristic Search Viewed as Path Finding in a Graph”. In: *Artificial Intelligence* (1970), pp. 193–204.

[54] Francisco-Javier Pulido, Lawrence Mandow, and José-Luis Pérez-de-la-Cruz. “Dimensionality Reduction in Multiobjective Shortest Path Search”. In: *Computers & Operations Research* 64 (2015), pp. 60–70.

[55] Zhongqiang Ren, Jiaoyang Li, Han Zhang, Sven Koenig, Sivakumar Rathinam, and Howie Choset. “Binary Branching Multi-Objective Conflict-Based Search for Multi-Agent Path Finding”. In: *International Conference on Automated Planning and Scheduling (ICAPS)*. 2023, pp. 361–369.

[56] Zhongqiang Ren, Sivakumar Rathinam, and Howie Choset. “A Conflict-Based Search Framework for Multiobjective Multiagent Path Finding”. In: *IEEE Transactions on Automation Science and Engineering* 20.2 (2022), pp. 1262–1274.

[57] Zhongqiang Ren, Sivakumar Rathinam, and Howie Choset. “Subdimensional Expansion for Multi-Objective Multi-Agent Path Finding”. In: *IEEE Robotics and Automation Letters* 6.4 (2021), pp. 7153–7160.

[58] Zhongqiang Ren, Sivakumar Rathinam, Maxim Likhachev, and Howie Choset. “Multi-Objective Path-Based D* Lite”. In: *IEEE Robotics and Automation Letters* 7.2 (2022), pp. 3318–3325.

[59] Zhongqiang Ren, Richard Zhan, Sivakumar Rathinam, Maxim Likhachev, and Howie Choset. “Enhanced Multi-Objective A* Using Balanced Binary Search Trees”. In: *Symposium on Combinatorial Search (SoCS).* Vol. 15. 1. 2022, pp. 162–170.

[60] Silvia Richter, Jordan Thayer, and Wheeler Ruml. “The Joy of Forgetting: Faster Anytime Search via Restarting”. In: *International Conference on Automated Planning and Scheduling (ICAPS).* Vol. 20. 2010, pp. 137–144.

[61] Nicolás Rivera, Jorge A Baier, and Carlos Hernández. “Subset Approximation of Pareto Regions with Bi-Objective A*”. In: *AAAI Conference on Artificial Intelligence (AAAI).* 2022, pp. 10345–10352.

[62] Diederik M Roijers and Shimon Whiteson. *Multi-Objective Decision Making*. Springer, 2017.

[63] Oren Salzman, Ariel Felner, Carlos Hernandez, Han Zhang, Shao-Hung Chan, and Sven Koenig. “Heuristic-Search Approaches for the Multi-Objective Shortest-Path Problem: Progress and Research Opportunities”. In: *International Joint Conferences on Artificial Intelligence (IJCAI).* 2023, pp. 6759–6768.

[64] Antonio Sedeno-Noda and Andrea Raith. “A Dijkstra-Like Method Computing All Extreme Supported Non-Dominated Solutions of the Biobjective Shortest Path Problem”. In: *Computers & Operations Research* 57 (2015), pp. 83–94.

[65] Antonio Sedeño-Noda and Marcos Colebrook. “A Biobjective Dijkstra Algorithm”. In: *European Journal of Operational Research* 276.1 (2019), pp. 106–118.

[66] Takehide Soh, Mutsunori Banbara, Naoyuki Tamura, and Daniel Le Berre. “Solving Multiobjective Discrete Optimization Problems with Propositional Minimal Model Generation”. In: *International Conference on Principles and Practice of Constraint Programming (CP).* 2017, pp. 596–614.

[67] Roni Stern, Ariel Felner, Jur van den Berg, Rami Puzis, Rajat Shah, and Ken Goldberg. “Potential-Based Bounded-Cost Search and Anytime Non-Parametric A*”. In: *Artificial Intelligence* 214 (2014), pp. 1–25.

[68] Roni Stern, Nathan R. Sturtevant, Dor Atzmon, Thayne Walker, Jiaoyang Li, Liron Cohen, Hang Ma, T. K. Satish Kumar, Ariel Felner, and Sven Koenig. “Multi-Agent Pathfinding: Definitions, Variants, and Benchmarks”. In: *Symposium on Combinatorial Search (SoCS).* 2019, pp. 151–158.

[69] Sabine Storandt. “Route Planning for Bicycles—Exact Constrained Shortest Paths Made Practical via Contraction Hierarchy”. In: *International Conference on Automated Planning and Scheduling (ICAPS).* 2012, pp. 234–242.

[70] Nathan Sturtevant, Ariel Felner, Max Barrer, Jonathan Schaeffer, and Neil Burch. “Memory-Based Heuristics for Explicit State Spaces”. In: *International Joint Conferences on Artificial Intelligence (IJCAI).* 2009, pp. 609–614.

[71] Xiaoxun Sun, Sven Koenig, and William Yeoh. “Generalized Adaptive A*.” In: *Autonomous Agents and MultiAgent Systems (AAMAS).* 2008, pp. 469–476.

[72] Jordan Tyler Thayer and Wheeler Ruml. “Bounded Suboptimal Search: A Direct Approach Using Inadmissible Estimates”. In: *International Joint Conferences on Artificial Intelligence (IJCAI).* 2011, pp. 674–679.

[73] Jordan Tyler Thayer and Wheeler Ruml. “Faster than Weighted A*: An Optimistic Approach to Bounded Suboptimal Search.” In: *International Conference on Automated Planning and Scheduling (ICAPS).* 2008, pp. 355–362.

[74] George Tsaggouris and Christos D. Zaroliagis. “Multiobjective Optimization: Improved FPTAS for Shortest Paths and Non-Linear Objectives with Applications”. In: *Theory of Computing Systems* 45.1 (2009), pp. 162–186.

[75] Carlos Hernández Ulloa, Han Zhang, Sven Koenig, Ariel Felner, and Oren Salzman. “Efficient Set Dominance Checks in Multi-Objective Shortest-Path Algorithms via Vectorized Operations”. In: *Symposium on Combinatorial Search (SoCS).* 2024, pp. 208–212.

[76] Tansel Uras, Sven Koenig, and Carlos Hernández Ulloa. “Subgoal Graphs for Optimal Pathfinding in Eight-Neighbor Grids”. In: *International Conference on Automated Planning and Scheduling (ICAPS).* 2013, pp. 224–232.

[77] Fangji Wang, Han Zhang, Sven Koenig, and Jiaoyang Li. “Efficient Approximate Search for Multi-Objective Multi-Agent Path Finding”. In: *International Conference on Automated Planning and Scheduling (ICAPS).* 2024, pp. 613–622.

[78] Arthur Warburton. “Approximation of Pareto Optima in Multiple-Objective, Shortest-Path Problems”. In: *Operations Research* 35.1 (1987), pp. 70–79.

[79] Peter R Wurman, Raffaello D’Andrea, and Mick Mountz. “Coordinating Hundreds of Cooperative, Autonomous Vehicles in Warehouses”. In: *AI Magazine* 29.1 (2008), pp. 9–20.

[80] Jingjin Yu and Steven M LaValle. “Structure and Intractability of Optimal Multi-Robot Path Planning on Graphs”. In: *AAAI Conference on Artificial Intelligence (AAAI).* 2013, pp. 1443–1449.

[81] Han Zhang, Oren Salzman, Ariel Felner, T. K. Satish Kumar, and Sven Koenig. "Bounded-Suboptimal Weight-Constrained Shortest-Path Search via Efficient Representation of Paths". In: *International Conference on Automated Planning and Scheduling (ICAPS)*. 2024, pp. 680–688.

[82] Han Zhang, Oren Salzman, Ariel Felner, T. K. Satish Kumar, Shawn Skyler, Carlos Hernández Ulloa, and Sven Koenig. "Towards Effective Multi-Valued Heuristics for Bi-Objective Shortest-Path Algorithms via Differential Heuristics". In: *Symposium on Combinatorial Search (SoCS)*. 2023, pp. 101–109.

[83] Han Zhang, Oren Salzman, Ariel Felner, TK Satish Kumar, Carlos Hernández Ulloa, and Sven Koenig. "Efficient Multi-Query Bi-Objective Search via Contraction Hierarchies". In: *International Conference on Automated Planning and Scheduling (ICAPS)*. 2023, pp. 452–461.

[84] Han Zhang, Oren Salzman, Ariel Felner, TK Satish Kumar, Carlos Hernández Ulloa, and Sven Koenig. "Speeding Up Dominance Checks in Multi-Objective Search: New Techniques and Data Structures". In: *Symposium on Combinatorial Search (SoCS)*. 2024, pp. 228–232.

[85] Han Zhang, Oren Salzman, Ariel Felner, Carlos Hernández Ulloa, and Sven Koenig. "A-A*pex: Efficient Anytime Approximate Multi-Objective Search". In: *Symposium on Combinatorial Search (SoCS)*. 2024, pp. 179–187.

[86] Han Zhang, Oren Salzman, T. K. Satish Kumar, Ariel Felner, Carlos Hernández, and Sven Koenig. "Anytime Approximate Bi-Objective Search". In: *Symposium on Combinatorial Search (SoCS)*. 2022, pp. 199–207.

[87] Han Zhang, Oren Salzman, T. K. Satish Kumar, Ariel Felner, Carlos Hernández Ulloa, and Sven Koenig. "A* pex: Efficient Approximate Multi-Objective Search on Graphs". In: *International Conference on Automated Planning and Scheduling (ICAPS)*. 2022, pp. 394–403.

[88] Yuli Zhang, Zuo-Jun Max Shen, and Shiji Song. "Parametric Search for the Bi-Attribute Concave Shortest Path Problem". In: *Transportation Research Part B: Methodological* 94 (2016), pp. 150–168.

[89] Xiaoyan Zhu and Wilbert E Wilhelm. "A Three-Stage Approach for the Resource-Constrained Shortest Path as a Sub-Problem in Column Generation". In: *Computers & Operations Research* 39.2 (2012), pp. 164–178.