Enhanced Multi-Objective A* Using Balanced Binary Search Trees

Zhongqiang Ren[1], Richard Zhan[1], Sivakumar Rathinam[2], Maxim Likhachev[1] and Howie Choset[1]

[1]Carnegie Mellon University, 5000 Forbes Ave., Pittsburgh, PA 15213, USA
[2]Texas A&M University, College Station, TX 77843-3123.
{zhongqir, rzhan, mlikhach, choset}@andrew.cmu.edu, srathinam@tamu.edu

Abstract

This work addresses a Multi-Objective Shortest Path Problem (MO-SPP) on a graph where the goal is to find a set of Pareto-optimal solutions from a start node to a destination in the graph. A family of approaches based on MOA* have been developed to solve MO-SPP in the literature. Typically, these approaches maintain a "frontier" set at each node during the search process to keep track of the non-dominated, partial paths to reach that node. This search process becomes computationally expensive when the number of objectives increases as the number of Pareto-optimal solutions becomes large. In this work, we introduce a new method to efficiently maintain these frontiers for multiple objectives by incrementally constructing balanced binary search trees within the MOA* search framework. We first show that our approach correctly finds the Pareto-optimal front, and then provide extensive simulation results for problems with three, four and five objectives to show that our method runs faster than existing techniques by up to an order of magnitude.

Introduction

Given a graph with non-negative scalar edge costs, the well-known shortest path problem (SPP) requires computing a minimum-cost path from the given start node to a destination node in the graph. This work considers the so-called Multi-Objective Shortest Path Problem (MO-SPP) (Loui 1983; Stewart and White 1991; Mandow and De La Cruz 2008), which generalizes SPP by associating each edge with a non-negative cost vector (of constant length), where each component of the vector corresponds to an objective to be minimized. MO-SPP arises in many applications including hazardous material transportation (Erkut, Tjandra, and Verter 2007), robot design (Xu et al. 2021), and airport departure runway scheduling (Montoya, Rathinam, and Wood 2013).

In the presence of multiple conflicting objectives, in general, no (single) feasible path can simultaneously optimize all the objectives. Therefore, the goal of MO-SPP is to find a Pareto-optimal set (of solution paths), whose cost vectors form the so-called Pareto-optimal front. A path is Pareto-optimal (also called non-dominated) if no objective of the path can be improved without deteriorating at least one of the other objectives. MO-SPP is computationally hard, even for two objectives (Hansen 1980).

To solve MO-SPP, several multi-objective A* (MOA*)-like planners (Stewart and White 1991; Mandow and De La Cruz 2008; Ulloa et al. 2020; Goldin and Salzman 2021; Ahmadi et al. 2021) have been developed to compute the exact or an approximated Pareto-optimal front. In MO-SPP, there are in general, multiple non-dominated partial solution paths between the start and any other node in the graph, and MOA* planners memorize, select and expand these non-dominated paths at each node during the search. When a new partial solution path π is found to reach a node v, the path π needs to be compared with all previously found non-dominated paths that reach v to check for dominance: verify whether the accumulated cost vector along π is dominated by the accumulated cost vector of any existing paths. These dominance checks are computationally expensive, especially when there are many non-dominated paths at a node, as it requires numerous cost vector comparisons (Pulido, Mandow, and Pérez-de-la Cruz 2015).

Recently, fast dominance check techniques (Pulido, Mandow, and Pérez-de-la Cruz 2015; Ulloa et al. 2020) were developed under the framework of MOA*-like search to expedite these dominance checks. Among them, the Bi-objective A* (BOA*) (Ulloa et al. 2020) achieves around an order of magnitude speed up compared to the existing MOA*-like search. Recently, BOA* has been further improved in (Goldin and Salzman 2021) and (Ahmadi et al. 2021). However, BOA* as well as its improved versions are limited to handle two objectives only. In this work, we provide a new approach to perform dominance checks relatively fast that can handle an arbitrary number of objectives for MOA*-like search.

Specifically, building on the existing fast dominance check techniques (Pulido, Mandow, and Pérez-de-la Cruz 2015; Ulloa et al. 2020), we develop a new method called Enhanced Multi-Objective A* (EMOA*) that uses a balanced binary search tree (BBST) to store the non-dominated partial solution paths at each node. The key ideas are: (i) the BBST can be *incrementally* constructed during the MOA* search, which makes it computationally efficient to maintain; (ii) the BBST is organized using the lexicographic order between cost vectors, which can *guide* the dominance checks and expedite the computation; (iii) the devel-

Copyright © 2022, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

oped BBST-based method is compatible with existing dominance check approaches, which allows EMOA* to also include the existing techniques to speed up the computation of the Pareto-optimal front. We also show that the existing BOA* (Ulloa et al. 2020) is a special case of EMOA* when there are only two objectives. We analyze the runtime complexity of the proposed method and show that EMOA* can find all cost-unique Pareto-optimal solutions. To verify our approach, we run massive tests to compare EMOA* with three baselines (NAMOA*-dr and two extensions of BOA*) in various maps with three, four and five objectives. Our results show that EMOA* achieves up to an order of magnitude speed-up compared to all the baselines on average, and is particularly advantageous for problem instances that have a large number of Pareto-optimal solutions.

Problem Description

Let $G = (V, E, \vec{c})$ denote a graph with vertex set V and edge set E, where each edge $e \in E$ is associated with a non-negative cost vector $\vec{c}(e) \in (\mathbb{R}^+)^M$ with M being a positive integer and $\mathbb{R}^+$ being the set of non-negative real numbers. Let $\pi(v_1, v_\ell)$ denote a path connecting $v_1, v_\ell \in V$ via a sequence of vertices $(v_1, v_2, \dots, v_\ell)$ in G, where v_k and v_{k+1} are connected by an edge $(v_k, v_{k+1}) \in E$, for $k = 1, 2, \dots, \ell - 1$. Let $\vec{g}(\pi(v_1, v_\ell))$ denote the cost vector corresponding to the path $\pi(v_1, v_\ell)$, which is the sum of the cost vectors of all edges present in the path, $i.e.$ $\vec{g}(\pi(v_1, v_\ell)) = \Sigma_{k=1,2,\dots,\ell-1}\vec{c}(v_k, v_{k+1})$. To compare any two paths, we compare the cost vector associated with them using the dominance relation (Ehrgott 2005):

Definition 1 (Dominance) *Given two vectors a and b of length M, a dominates b (denoted as $a \succeq b$) if and only if $a(m) \le b(m)$, $\forall m \in \{1, 2, \dots, M\}$, and $a(m) < b(m)$, $\exists m \in \{1, 2, \dots, M\}$.*

If a does not dominate b, this non-dominance is denoted as $a \not\succeq b$. Any two paths $\pi_1(u, v), \pi_2(u, v)$, for two vertices $u, v \in V$, are non-dominated (with respect to each other) if the corresponding cost vectors do not dominate each other.

Let v_o, v_d denote the start and destination vertices respectively. The set of all non-dominated paths between v_o and v_d is called the *Pareto-optimal* set. A maximal subset of the Pareto-optimal set, where any two paths in this subset do not have the same cost vector is called a *cost-unique* Pareto-optimal set. This paper considers the problem that aims to compute a cost-unique Pareto-optimal set.

Preliminaries

Basic Concepts

Let $l = (v, \vec{g})$ denote a *label*[1], which is a tuple of a vertex $v \in V$ and a cost vector $\vec{g}$. A label represents a partial solution path from v_o to v with cost vector $\vec{g}$. To simplify notations, given a label l, let $v(l), \vec{g}(l)$ denote the vertex and the cost vector contained in label l respectively. A label l is said to be dominated by (or is equal to) another label l' if $v(l) = v(l')$ and $\vec{g}(l) \succeq \vec{g}(l')$ (or $\vec{g}(l) = \vec{g}(l')$).

Let $\vec{h}(v), v \in V$ denote a *consistent* heuristic vector of vertex v that satisfies $\vec{h}(v) \le \vec{h}(u) + \vec{c}(u, v), \forall u, v \in V$. Additionally, let $\vec{f}(l) := \vec{g}(l) + \vec{h}(v(l))$, and let OPEN denote a priority queue of labels, where labels are prioritized by their corresponding $\vec{f}$-vectors in *lexicographic* order.

Finally, let $\alpha(u), u \in V$ denote the *frontier* set *at* vertex u, which stores all non-dominated labels l *at* vertex u (i.e. $v(l) = u$). Intuitively, each label $l \in \alpha(u), u \in V$ identifies a non-dominated (partial solution) path from v_o to u. See Fig. 1 (a) for an illustration. For presentation purposes, we also refer to $\alpha(v_d)$ as $\mathcal{S}$, the solution set, which is the frontier set at the destination vertex, and each label in $\mathcal{S}$ identifies a cost-unique Pareto-optimal solution (path).

[1]To identify a partial solution path, different names such as nodes (Ulloa et al. 2020), states (Ren, Rathinam, and Choset 2021; Ren et al. 2022) and labels (Martins 1984; Sanders and Mandow 2013), have been used in the multi-objective path planning literature. This work uses "labels" to identify partial solution paths.

Algorithm 1: Search Framework

```
1:  l_o ← (v_o, 0⃗)
2:  Add l_o to OPEN
3:  α(v) ← ∅, ∀v ∈ V
4:  while OPEN ≠ ∅ do
5:      l ← OPEN.pop()
6:      if FrontierCheck(l) or SolutionCheck(l) then
7:          continue                    ▷ Current iteration ends
8:      UpdateFrontier(l)
9:      if v(l) = v_d then
10:         continue                    ▷ Current iteration ends
11:     for all v' ∈ GetSuccessors(v(l)) do
12:         l' ← (v', g⃗(l) + c⃗(v, v')), parent(l') ← l
13:         f⃗(l') ← g⃗(l') + h⃗(v(l'))
14:         if FrontierCheck(l') or SolutionCheck(l') then
15:             continue        ▷ Move to the next successor.
16:         Add l' to OPEN
17: return α(v_d)               ▷ α(v_d) is also referred to as S
```

Search Framework

To begin with, we reformulate BOA* (Ulloa et al. 2020) as a general search framework as shown in Alg. 1, and explain the running process. We then provide a technical review of the existing algorithms NAMOA*-dr (Pulido, Mandow, and Pérez-de-la Cruz 2015) and BOA* (Ulloa et al. 2020), and discuss how our EMOA* differs from them.

As shown in Alg. 1, to initialize (lines 1-3), an initial label $l_o = (v_o, \vec{0})$ is created, and is added to OPEN. Additionally, the frontier sets at all vertices are initialized to be empty sets. In each search iteration (lines 5-16), the label with the lexicographically minimum $\vec{f}$-vector is popped from OPEN and is denoted as l in Alg. 1. This label l is then (line 6) checked for dominance via the following two procedures:

- First, l is compared with labels in $\alpha(v(l))$ in procedure *FrontierCheck* to verify if there exists a label in $\alpha(v(l))$ that dominates (or is equal to) l.
- Second, l is compared with labels in $\mathcal{S}$ in procedure *SolutionCheck* to verify if there exists a label l^* in $\mathcal{S}$

such that $g(l^*)$ dominates (or is equal to) $\vec{f}(l)$. Note that $\vec{f}(l^*) = \vec{g}(l^*)$ as $\vec{h}(v(l^*)) = \vec{h}(v_d) = \vec{0}$.

If l is dominated in either *FrontierCheck* or *SolutionCheck*, l is discarded and the current search iteration ends, because l cannot lead to a cost-unique Pareto-optimal solution. Otherwise ($i.e.$ l is non-dominated in both *FrontierCheck* and *SolutionCheck*), l is used to update the frontier set $\alpha(v(l))$ (line 8): procedure *UpdateFrontier* first removes all the existing labels in $\alpha(v(l))$ that are dominated by l, and then adds l into $\alpha(v(l))$. Note that this includes the case when $v(l) = v_d$, where $\alpha(v_d)$ ($i.e.$ $\mathcal{S}$) is updated, which means a solution path from v_o to v_d is found. After *UpdateFrontier*, label l is verified whether $v(l) = v_d$ (line 9). If $v(l) = v_d$, the current search iteration ends (line 10); Otherwise, l is expanded, as explained in the next paragraph.

To expand a label l ($i.e.$ to expand the partial solution path represented by label l), for each successor vertex v' of $v(l)$, a new label $l' = (v', \vec{g}(l) + \vec{c}(v, v'))$ is generated, which represents a new path from v_o to v' via $v(l)$ by extending l ($i.e.$ the path represented by l). The parent pointer $parent(l')$ is set to l, which helps reconstruct a solution path for each label in $\alpha(v_d)$ after the algorithm terminates. Then (line 14), *FrontierCheck* and *SolutionCheck* are invoked on label l' to verify if l' is dominated and should be discarded or not. (Note that the dominance checks are needed at both line 6 and 14, which is explained in the next subsection.) If l' is non-dominated, l' is added to OPEN for future expansion.

Finally (line 17), the search process terminates when OPEN is empty, and returns $\alpha(v_d)$, a set of labels, each of which represents a cost-unique Pareto-optimal solution path. Additionally, the cost vectors of labels in $\alpha(v_d)$ form the entire Pareto-optimal front of the given problem instance.

Brief Summary of NAMOA*-dr and BOA*

NAMOA*-dr (Pulido, Mandow, and Pérez-de-la Cruz 2015) is a multi-objective search algorithm that can handle an arbitrary number of objectives. The main differences between NAMOA*-dr and Alg. 1 can be summarized in the following two points. *First*, unlike Alg. 1, in NAMOA*-dr, when a new label l' is generated during the expansion, l' will be used for dominance checks against *existing* labels in OPEN and frontier sets[2] to remove labels that are dominated by l (which happens between line 15 and 16 in Alg. 1 and is not shown in Alg. 1 for presentation purposes). These checks are called "eager checks" (Ulloa et al. 2020). With eager checks, each popped label from OPEN is guaranteed to be non-dominated, and lines 6-8 in Alg. 1 are thus skipped in NAMOA*-dr. *Second*, a key idea in NAMOA*-dr is that, with (i) consistent heuristics and (ii) an OPEN list where labels are lexicographically prioritized, the first component of the cost vectors can be ignored in some of the dominance checks. This idea is referred to as the "dimensionality reduction" (Pulido, Mandow, and Pérez-de-la Cruz 2015), which helps in speeding up the dominance checks in NAMOA*-dr.

[2]NAMOA*-dr maintains two frontier sets $G_{op}(v)$ (open) and $G_{cl}(v)$ (closed) at each vertex $v \in V$, and we refer the reader to (Ulloa et al. 2020; Pulido, Mandow, and Pérez-de-la Cruz 2015) for more details.

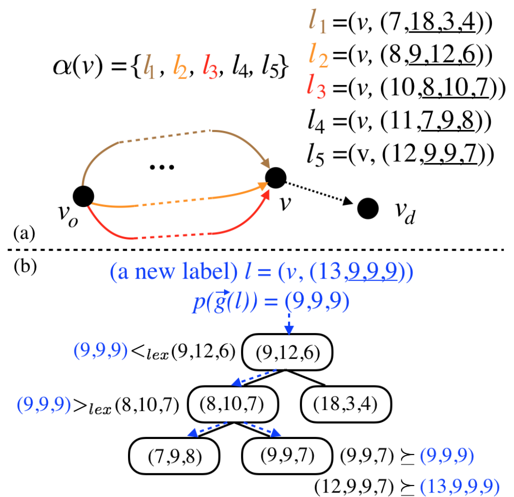


Figure 1: Fig. (a) shows the frontier set $\alpha(v)$ at some vertex v. There are five labels in $\alpha(v)$. The underlined three numbers of each $\vec{g}$-vector indicate the corresponding projected vector $p(\vec{g})$. Fig. (b) shows the corresponding balanced binary search tree. The keys of the nodes in this tree forms the non-dominated subset of the projected vectors. The dashed blue arrows show the sequence of tree nodes that are traversed when running the *FrontierCheck* procedure (Alg. 2). The projected vector $(9, 9, 7)$ in the tree dominates the input vector $(9, 9, 9)$, which indicates that the new label l (in blue) with $\vec{g}$-vector $(13, 9, 9, 9)$ is dominated and should be discarded in EMOA* search.

BOA* (Ulloa et al. 2020) leverages the aforementioned dimensionality reduction, and introduces the idea of "lazy checks", which avoids the eager checks as in NAMOA*-dr. Specifically, BOA* follows the search process as shown in Alg. 1, where lines 6-8 in Alg. 1 help defer the eager checks related to a label l until l is going to be expanded. With the help of this lazy check technique, BOA* guarantees that all dominance checks can be performed in constant time for a bi-objective problem which leads to speeding up the overall search process. Specifically, the three key procedures *FrontierCheck*, *SolutionCheck* and *UpdateFrontier* as shown in Alg. 1 can be conducted in constant time in BOA*.

Our approach EMOA* also follows the same framework as shown in Alg. 1, and inherits the ideas of dimensionality reduction and lazy check. However, EMOA* realizes the three key procedures by incrementally building balanced binary search trees, which can handle an arbitrary number of objectives (Fig. 1). This leads to up to an order of magnitude speed-up in comparison with the existing baselines.

Enhanced Multi-Objective A*

This section shows how EMOA* realizes *FrontierCheck*, *SolutionCheck* and *UpdateFrontier* by leveraging balanced binary search trees (BBST). We begin by introducing a few definitions and then elaborate the BBST-based procedures.

The Check and Update Problems

Definition 2 (Dominance Check (DC) Problem) *Given a set B of K-dimensional non-dominated vectors and a new K-dimensional vector b, the DC problem aims to verify whether there exists a vector $b' \in B$ such that $b' \leq b$ (i.e. b' is component-wise no larger than b, which is equivalent to $b' \succeq b$ or $b' = b$).*

Definition 3 (Non-Dominated Set Update (NSU) Problem) *Given a set B of K-dimensional non-dominated vectors and a new K-dimensional vector b that is non-dominated by any vectors in B, the NSU problem computes $ND(B \bigcup \{b\})$, (i.e. the non-dominated subset of $B \bigcup \{b\}$).*

The relationship between the aforementioned three procedures and these two problems can be described as follows:

- In *FrontierCheck*, given a new label l and the frontier set $\alpha(v(l))$, an equivalent DC problem can be generated with input $b = \vec{g}(l)$ and $B = \{\vec{g}(l')|l' \in \alpha(v(l))\}$.
- Similarly, in *SolutionCheck*, given a new label l and the frontier set $\alpha(v(l))$, an equivalent DC problem can be generated with $b = \vec{f}(l)$ and $B = \{\vec{g}(l')|l' \in \alpha(v_d)\}$.
- Finally, in *UpdateFrontier*, given a new label l and the frontier set $\alpha(v(l))$, an equivalent NSU problem can be generated with $b = \vec{g}(l)$ and $B = \{\vec{g}(l')|l' \in \alpha(v(l))\}$.

From now on, we focus on how to quickly solve DC and NSU problems. Note that, a *baseline* method that solves the DC problem runs a for-loop over each vector $b' \in B$ and check if $b' \leq b$, which takes $O(|B|K)$ time. A baseline method that solves the NSU problem requires two steps: (i) filter B by removing from B all vectors that are dominated by b, and (ii) add b into B. Here, a naive method for step (i) runs a for-loop over set B to remove all dominated vectors and takes $O(|B|K)$ time, and step (ii) takes constant time. Consequently, the overall time complexity is $O(|B|K)$.

Balanced Binary Search Trees (BBSTs)

To efficiently solve the DC and NSU problems, our method leverages the BBST data structure. As a short review, let n denote a node[3] within a binary search tree (BST) with the following attributes:

- Let $n.height$ denote the height of node n, which is the number of edges along the longest downwards path between n and a leaf node. A leaf node has a height of zero.
- Let $n.key$ denote the key of n, which is a K-dimensional vector in this work. To compare two nodes, their keys are compared by *lexicographic* order.
- Let $n.left$ and $n.right$ denote the left child and the right child of n respectively, which represent the left sub-tree and the right sub-tree respectively.
- We say $n = NULL$ if n does not exist in the BST. For example, if n is a leaf node, then $n.left = NULL$ and $n.right = NULL$.

[3]For the rest of this work, for presentation purposes, the term "vertex" is associated with the graph G and the term "node" is associated with the balanced binary search tree.

Algorithm 2: *Check*(n, b)

```
1: INPUT: n is a node in an AVL-tree and b is a vector
2: if n = NULL then
3:     return false
4: if n.key ≤ b then
5:     return true
6: if b <_lex n.key then
7:     return Check(n.left, b)
8: else                                  ▷ i.e. b >_lex n.key
9:     if Check(n.left, b) then          ▷ Removed in TOA*
10:        return true                   ▷ Removed in TOA*
11:    return Check(n.right, b)
```

In this work, we limit our focus to the AVL-tree, one of the most famous balanced BSTs. An AVL-tree has the following property: for any node n within an AVL-tree, let $d(n) := n.left.height - h.right.height$ denote the difference between the height of the left and the right child node, then AVL-tree is called "balanced" if $d(n) \in \{-1, 0, 1\}$. To maintain balance at insertion or deletion of nodes, an AVL-tree invokes the so-called "rotation" operations when $d(n) \leq -2$ or $d(n) \geq 2$ and the tree is always balanced. Consequently, given an AVL-tree of size N ($i.e.$ containing N nodes), the height of the root is bounded by $O(logN)$.

BBST-Based Check Method

Given a set B of non-dominated vectors, let $\mathcal{T}_B$ denote an AVL-tree that stores all vectors in B as the keys of tree nodes. Now, given a new vector b, the DC problem can be solved via Alg. 2, which traverses the tree recursively while running dominance comparison.

Specifically, Alg. 2 is invoked with *Check*$(\mathcal{T}_B.root, b)$, where $\mathcal{T}_B.root$ denotes the root of the tree and b is the input vector to be checked. As a base case (line 2), if the input node n is $NULL$, the algorithm terminates and returns false, which means b is non-dominated. When the input node is not $NULL$, b is checked for dominance against $n.key$ and returns true if $n.key \leq b$. Otherwise, the algorithm verifies if b is lexicographically smaller than (denoted as $<_{lex}$) $n.key$.

- (Case-1) If $b <_{lex} n.key$, there is no need to traverse the right sub-tree from n, since any node in the right sub-tree of n cannot be component-wise no larger than b, and the algorithm (recursively) invokes itself to traverse the left sub-tree for dominance checks.
- (Case-2) Otherwise ($i.e.$ $b >_{lex} n.key$), the algorithm first invokes itself to traverse the left sub-tree (line 9) and then the right sub-tree (line 11) for dominance checks. Note that, in this case, both child nodes need recursive traversal to ensure correctness.

BBST-Based Update Method

Similarly, the NSU problem with input B (in the form of a correpsonding BBST $\mathcal{T}_B$) and a non-dominated vector b, can be solved by (i) invoking Alg. 3 to remove nodes with dominated keys from the tree $\mathcal{T}_B$ and (ii) insert the input (non-dominated) vector b into the tree. Here, step (ii) is a regular

Algorithm 3: *Filter*(n, b)

1: INPUT: n is a node in an AVL-tree and b is a vector
2: **if** $n = NULL$ **then**
3: **return** $NULL$
4: **if** $b >_{lex} n.key$ **then**
5: $n.right \leftarrow$*Filter*$(n.right, b)$
6: **else**
7: $n.left \leftarrow$*Filter*$(n.left, b)$
8: $n.right \leftarrow$*Filter*$(n.right, b)$
9: **if** $b \succeq n.key$ **then**
10: **return** *AVL-Delete*(n)
11: *AVL-balancing*() when needed.

AVL-tree insertion operation, which takes $O(log|B|)$ time, and we will focus on step (i) in the ensuing paragraphs.

For step (i), Alg. 3 is invoked with Filter($\mathcal{T}_B.root, b$), where $\mathcal{T}_B.root$ denotes the root of the tree and b is the input non-dominated vector. As shown in Alg. 3, as a base case (line 2), if the input node is $NULL$, the algorithm terminates and returns a $NULL$. When the input node n is not $NULL$, the algorithm verifies whether $b >_{lex} n.key$.

- (Case-1) If $b >_{lex} n.key$, there is no need to filter the left sub-tree of n (since any node in the left sub-tree of n must be non-dominated by b) and the algorithm recursively invokes itself to traverse the right sub-tree for filtering.
- (Case-2) Otherwise (i.e. $b <_{lex} n.key$)[4], the algorithm first invokes itself to traverse the left sub-tree (line 9) and then the right sub-tree (line 11) for filtering. Note that, in this case, both child nodes need to be traversed for further dominance check to ensure correctness.

At the end (line 9-10), $n.key$ is checked for dominance against b. If $n.key$ is dominated, n is removed from the tree. The tree is also processed to ensure that the resulting tree is still balanced (line 11). These are all common operations related to AVL-trees. In the worst case, the entire tree is traversed and all nodes in the tree are recursively deleted (from the leaves to the root), which takes $O(|B|K)$ time.

Remark. Theoretically, both Alg. 2 and 3 runs in $O(|B|K)$ time in the worst case, which is the same as the aforementioned baseline approaches ($i.e.$ running a for-loop over B). However, as shown in the result section, these BBST-based methods can solve the DC and NSU problems much more efficiently in practice. The intuitive reason behind such efficiency is that, the AVL-tree is organized based on the lexicographic order, which can provide guidance when traversing the tree for dominance checks. As a result, only a small portion of the tree is traversed. Finally, note that the method in this section does not put any restriction on K.

EMOA* with BBST-Based Check and Update

This section presents how to use the BBST-based algorithms (Alg. 2, 3) within the framework of Alg. 1. Specifically, EMOA* leverages the idea of dimensionality reduction, which can expedite the BBST-based check and update. EMOA* has the property that, during the search process, the sequence of labels being expanded at the same vertex has non-decreasing f_1 values, where f_1 represents the first component of the $\vec{f}$-vector of a label. This property is caused by the fact that heuristics are consistent and all labels are selected from OPEN by lexicographic order of their $\vec{f}$-vectors. Additionally, since all labels at the same vertex v have the same $\vec{h}$-vector, the sequence of labels being expanded at the same vertex also has non-decreasing g_1 values, where g_1 represents the first component of the $\vec{g}$.

To simplify presentation, let $p : \mathbb{R}^M \rightarrow \mathbb{R}^{M-1}$ denote a *projection function* that removes the first component from the input vector. During the EMOA* search, when a new label l is generated, for *FrontierCheck*, we only need to do dominance comparison between $p(\vec{g}(l))$ and $p(\vec{g}(l')), \forall l' \in \alpha(v(l))$, instead of comparing $\vec{g}(l)$ with $\vec{g}(l'), \forall l' \in \alpha(v(l))$. Consequently, in EMOA*, for each vertex $v \in V$, a BBST $\mathcal{T}_B$ as aforementioned is constructed with $B = ND(\{p(\vec{g}(l')), \forall l' \in \alpha(v)\})$. In other words, the key of each node in $\mathcal{T}_B$ is a non-dominated projected cost vector of a label in $\alpha(v)$.

To realize *FrontierCheck* for a label l that is extracted from OPEN (line 6 in Alg. 1), Alg. 2 is invoked with $b = p(\vec{g}(l))$ and n being the root node of the tree $\mathcal{T}_B$. We provide a toy example for *FrontierCheck* in Fig. 1. Similarly, for *SolutionFilter* (line 6 in Alg. 1), Alg. 2 is invoked with $b = p(\vec{f}(l))$ and n being the root node of the tree $\mathcal{T}_{B'}$ with $B' = ND(\{p(\vec{g}(l')), \forall l' \in \alpha(v_d)\})$ ($i.e.$ the set of all non-dominated projected vectors of labels in the frontier set at the destination node). During the search, when a label l is extracted from OPEN and is used to update the frontier set in procedure *UpdateFrontier* (line 8 in Alg. 1), Alg. 3 is first invoked with $\vec{b} = p(\vec{g}(l))$ and n being the root node of the tree $\mathcal{T}_B$ where $B = ND(\{p(\vec{g}(l')), \forall l' \in \alpha(v(l))\})$. Then, $\vec{b} = p(\vec{g}(l))$ is added to $\mathcal{T}_B$.

In summary, to realize procedures *FrontierCheck*, *SolutionCheck* and *UpdateFrontier*, only the projected vectors of labels are needed, instead of the original vectors.

Generalization of BOA*

EMOA* generalizes BOA* in the following sense. When $M = 2$, for any cost vector $\vec{g}$ in a label, the projected vector $p(\vec{g})$ is of length one and is thus a scalar value. In this case, the AVL-tree corresponding to $\alpha(v)$ of any vertex $v \in V$ in EMOA* becomes a singleton tree: a tree with a single root node $\mathcal{T}_B.root$. The key value of $\mathcal{T}_B.root$ is the minimum value of $g_2(l)$ among all labels $l \in \alpha(v)$, which is the same as the auxiliary variable g_2^{min} introduced at each vertex in BOA*. Solving a DC problem requires only a scalar comparison between $\mathcal{T}_B.root.key$ and the scalar $p(\vec{g})$, the projected cost vector of the label selected from OPEN in each search iteration. Clearly, this scalar comparison takes constant time. Additionally, the *UpdateFrontier* in EMOA* requires simply assigning the scalar $p(\vec{g})$ to $\mathcal{T}_B.root.key$ ($i.e.$ g^2_{min}), which also takes constant time. Therefore, BOA* is a special case of EMOA* when $M = 2$.

[4]Note that it's impossible to have $b = n.key$: Within the EMOA* algorithm (Alg. 1), *UpdateFrontier* is always invoked after *FrontierCheck*. if $b = n.key$, *FrontierCheck* removes it and *UpdateFrontier* will not be invoked (line 6-8 in Alg. 1).

	BOA*	TOA*	EMOA*
M	$=2$	$=3$	≥ 2
Check	Constant Time	$O(log\|B\|)$	$O(\|B\|(M-1))$
Update	Constant Time	$O(\|B\|)$	$O(\|B\|(M-1))$

Table 1: Runtime complexity of related methods. BOA* is a special case of EMOA* when $M = 2$, and TOA* is an improved version of EMOA* when $M = 3$.

Tri-Objective A* (TOA*)

When $M = 3$, EMOA* can be further improved to achieve better theoretic runtime complexity when running Alg. 2. We name this improved algorithm TOA* (Tri-Objective A*). TOA* differs from EMOA* as follows: line 9-10 in Alg. 2 are removed. In other words, when $M = 3$, each projected vector b as well as the key of all nodes in the tree $\mathcal{T}_B$ have length $(M - 1) = 2$. In this case, during the computation of Alg. 2, when $b >_{lex} n.key$ ($i.e.$ line 8 in Alg. 2), there is no need to further traverse the left sub-tree.

Theorem 1 *Given b, a two dimensional vector, and an arbitrary node n in $\mathcal{T}_B$, if (i) $n.key \nleq b$ and (ii) $b >_{lex} n.key$, then the key of any nodes in the left sub-tree of n cannot dominate b.*

Please find the proof in the appendix.

In TOA*, the modified Alg. 2 traverses the tree either to the left sub-tree (when $b <_{lex} n.key$) or to the right sub-tree (when $b >_{lex} n.key$), which leads to a time complexity of $O(log|B|)$ (note that M is a constant here). We say that TOA* is an improved version of EMOA* with $M = 3$ since the theoretic computational complexity is improved. Finally, we summarize the computational complexity of the *Check* and *Update* procedures in BOA* (Ulloa et al. 2020) and our algorithms (TOA* and EMOA*) in Table 1.

Analysis of EMOA*

To save space, we provide the proofs in the appendix.

Lemma 1 *When expanding a label l, each successor label l' has f_1 value no smaller than the f_1 value of l.*

Lemma 2 *During the search of Alg. 1, the sequence of extracted labels from OPEN has non-decreasing f_1 values.*

Corollary 1 *During the search, the sequence of extracted and expanded labels at a specific vertex has non-decreasing f_1 and g_1 values.*

Corollary 2 *During the search, for a label l that is extracted from OPEN, we have $\vec{g}(l') \leq \vec{g}(l), l' \in \alpha(v(l))$ if and only if $p(\vec{g}(l')) \leq p(\vec{g}(l)), l' \in \alpha(v(l))$.*

Theorem 2 *EMOA* computes a maximal set of cost-unique Pareto-optimal paths connecting v_o and v_d at termination.*

Numerical Results

Baselines and Implementation

To verify the performance of EMOA* and TOA*, we introduce three baselines for comparison. The **first** baseline is NAMOA*-dr (Pulido, Mandow, and Pérez-de-la Cruz 2015), which is an algorithm in the literature that can handle an arbitrary number of objectives.

We propose a **second** baseline, which is an extension of BOA* to handle more than two objectives (hereafter referred to as ext-BOA* for simplicity). Specifically, the *FrontierCheck*, *SolutionCheck* and *UpdateFrontier* procedures are implemented with a naive for-loop as aforementioned in section "The Check and Update Problems". Furthermore, at each node v, a list $Q(v)$ that consists of the non-dominated projected $\vec{g}$-vectors of the labels in the frontier set $\alpha(v)$ is introduced, and those three procedures run for-loops to conduct dominance comparisons between the projected vector of the input label and each of the project vectors in $Q(v)$.

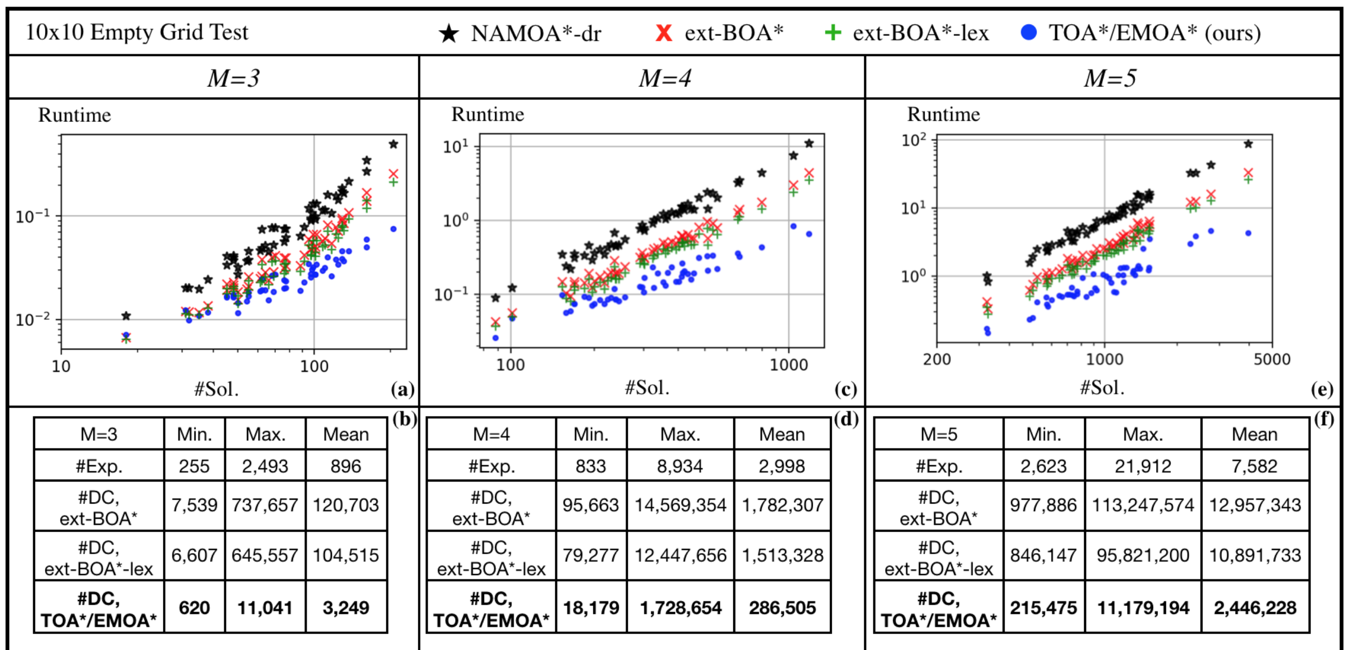
We propose a **third** baseline, which is an "optimized" version of ext-BOA* and is referred to as ext-BOA*-lex, where $Q(v)$ at each node v is further sorted with the lexicographic order from the minimum ($i.e.$ lex. min.) to the maximum ($i.e.$ lex. max.). In *FrontierCheck* (and *SolutionCheck*), to check whether an input label l is dominated or not, the procedure loops from the lex. min. to the lex. max. of $Q(v)$ (and $Q(v_d)$) and stops at the first vector in $Q(v)$ (and $Q(v_d)$) that dominates $p(\vec{g}(l))$ (and $p(\vec{f}(l))$ respectively). Similarly, in *UpdateFrontier*, to filter $Q(v)$ with an input non-dominated label l, the procedure first loops from the lex. max. to the lex. min. and stops at the first vector in the list that is lexicographically less than $p(\vec{g}(l))$. Then *UpdateFrontier* finds the right place to insert $p(\vec{g}(l))$ into $Q(v)$ to ensure that $Q(v)$ is still lexicographically sorted. We call it an optimized version as we take the view that, by running a for-loop over a lexicographically sorted list, the for-loop may stop earlier, and thus the overall search is expedited.

We implement all algorithms in C++ and test on a Ubuntu 20.04 laptop with an Intel Core i7-11800H 2.40GHz CPU and 16 GB RAM without multi-threading.[5] For a M-objective problem, the heuristic vectors are computed by running M backwards Dijkstra search from v_d: the m-th Dijkstra search ($m = 1, 2, \ldots, M$) uses edge cost values $c_m(e), \forall e \in E$ ($i.e.$ the m-th component of the cost vector $\vec{c}(e)$ of all edges). The time to compute heuristics is negligible in comparison with the overall runtime, and we report the runtime of the algorithm that excludes the time for computing heuristics. Finally, note that EMOA* with $M = 2$ is the same as BOA* (Ulloa et al. 2020), which has been investigated.

Experiment 1: Empty Map with $M = 3, 4, 5$

We begin by testing TOA* ($M = 3$) and EMOA* ($M = 4, 5$) against three baselines in a small obstacle-free four-connected grid of size 10×10 with v_o locating at the lower left corner and v_d locating at the upper right corner. Each component of the edge cost vector is randomly sampled from the integers within $[1, 10]$, which follows the convention in (Pulido, Mandow, and Pérez-de-la Cruz 2015), and 50 instances are generated. We plot the the runtime (vertical axis) against the number of cost-unique Pareto-optimal solutions (horizontal axis) for each instance in Fig. 2.

[5] Our software is at https://github.com/wonderren/public_emoa



M=3	Min.	Max.	Mean
#Exp.	255	2,493	896
#DC, ext-BOA*	7,539	737,657	120,703
#DC, ext-BOA*-lex	6,607	645,557	104,515
#DC, TOA*/EMOA*	**620**	**11,041**	**3,249**

M=4	Min.	Max.	Mean
#Exp.	833	8,934	2,998
#DC, ext-BOA*	95,663	14,569,354	1,782,307
#DC, ext-BOA*-lex	79,277	12,447,656	1,513,328
#DC, TOA*/EMOA*	**18,179**	**1,728,654**	**286,505**

M=5	Min.	Max.	Mean
#Exp.	2,623	21,912	7,582
#DC, ext-BOA*	977,886	113,247,574	12,957,343
#DC, ext-BOA*-lex	846,147	95,821,200	10,891,733
#DC, TOA*/EMOA*	**215,475**	**11,179,194**	**2,446,228**

Figure 2: Performance comparison between TOA*/EMOA* (ours) and the baselines (NAMOA*-dr, ext-BOA*, ext-BOA*-lex) in an empty 10×10 map. The runtime (seconds) of each instance is visualized against the number of cost-unique Pareto-optimal solutions of the instance in Fig (a,c,e). TOA*/EMOA*, ext-BOA* and ext-BOA*-lex all follows Alg. 1 and have the same number of expansions (#Exp). Rows with #DC in (b,d,f) shows the number of dominance checks required by each of the algorithms. To summarize, TOA*/EMOA* reduce the number of dominance checks during the search, and run up to an order of magnitude faster than the baselines. Note that the runtime axis is in log scale.

As shown in Fig. 2 (a,c,e), our TOA*/EMOA* expedites the overall search process for up to about an order of magnitude for all $M = 3, 4, 5$ in comparison with the baselines. Additionally, the speed-up provided by our methods becomes more obvious as the number of Pareto-optimal solutions increases, which indicates that TOA*/EMOA* are particularly advantageous when the given problem instance has numerous Pareto-optimal solutions. Fig. 2 (a,c,e) also show that ext-BOA*-lex slightly expedites the search process in comparison with ext-BOA* in general, which indicates that sorting $Q(v)$ by lexicographic order can expedite dominance checks within Alg. 1. However, this expedition is negligible when comparing with our methods, which verifies the benefits of constructing balanced binary search trees to organize the frontier set at vertices.

In Fig. 2 (b,d,f), Row #Exp shows the number of expansions required by ext-BOA*, ext-BOA*-lex and TOA*/EMOA*. Note that all these three approaches follow the same workflow as shown in Alg. 1 and thus have the same number of expansions during the search. NAMOA*-dr is omitted due to its relatively high runtime. Rows with #DC show the numbers of dominance checks required by the algorithms during the search. We can observe that TOA*/EMOA* significantly reduces the number of dominance checks. Note that #DC serves only as a reference here, and its not an accurate indicator of the computational burden for the following two reasons. First, the actual implementation of dominance checks runs a for-loop over components of vectors, and this for-loop terminates without reaching the last component when non-dominance is verified. This makes each dominance check operation have varying computational efforts. Second, in TOA*/EMOA*, the component-wise scalar comparison required when traversing the balanced binary search trees is not counted as dominance checks.

Experiment 2: Three Objectives in Various Maps

We then fix $M = 3$ and test the algorithms in two (grid) maps selected from a online data set (Stern et al. 2019). We make each grid map four-connected, and sample each component of the edge cost vector randomly from the integers within $[1, 10]$.

As shown in Fig. 3, TOA* runs faster than the baselines when there are a lot of Pareto-optimal solutions (e.g. > 30). When the number of Pareto-optimal solutions is small, the runtime of TOA* is similar to or slower than ext-BOA*-lex. It indicates that for problem instances with a small number of Pareto-optimal solutions, our method may not be the best choice to solve the problem. But it's also worthwhile to note that those instances are in general not challenging, as they can be solved by either of the four approaches within 0.1 seconds.

Experiment 3: City Road Network

Finally, we evaluate TOA* and the baselines in the New York City map (a graph with 264,346 vertices and 733,846

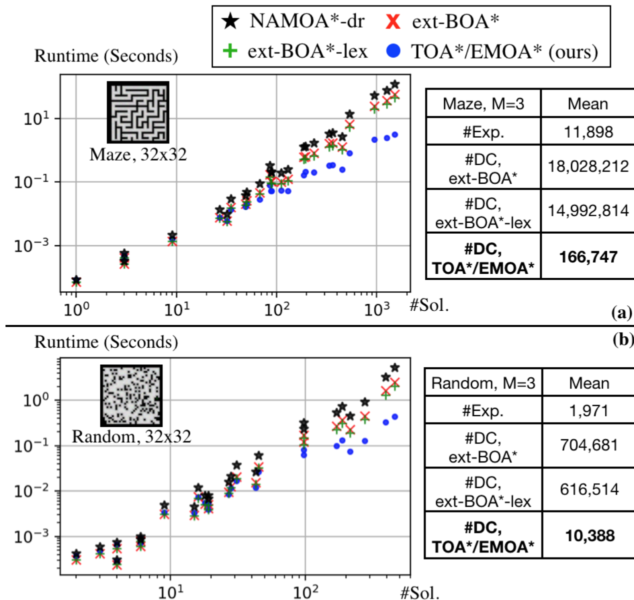


Figure 3: Comparison between TOA* (ours) and the baselines in various maps. The search time of each instance is visualized against the number of cost-unique Pareto-optimal solutions of the instance. TOA* runs up to an order of magnitude faster than the baselines due to the reduced number of dominance checks.

edges) from a online data set.[6] This data set provides distance (c_1) and travel time (c_2) for each edge. We introduce a third type of cost as follows (which is deterministic and reproducible). Let $deg(v)$ denote the degree (number of adjacent vertices) of $v \in V$, and let $deg(e) := \frac{deg(u)+deg(v)}{2}, e = (u, v) \in E$. If $deg(e) \geq 4$, $c_3(e) = 2$, otherwise $c_3(e) = 1$. The design of c_3 is motivated by hazardous material transportation (Erkut, Tjandra, and Verter 2007), where the transportation over busy edges can lead to higher risk if leakage happens, and $deg(e)$ is an indicator about how busy an edge is. We discuss the results in the caption of Table 2.

Other Related Work

MO-SPP algorithms range from exact approaches (Stewart and White 1991; Mandow and De La Cruz 2008; Ulloa et al. 2020) to approximation methods (Goldin and Salzman 2021; Perny and Spanjaard 2008; Warburton 1987), trading off solution optimality for computational efficiency. This work belongs to the category of exact approaches.

Another related work is the Kung's method (Kung, Luccio, and Preparata 1975) which addresses the following problem: Given an arbitrary set A of M dimensional vectors ($M \geq 2$), compute $ND(A)$, the non-dominated subset of A. The DC and NSU problems introduced in this work as mentioned in Def. 2 and 3 can be regarded as *incremental* versions of the problem solved by Kung's method, since the frontier set is constructed in an incremental manner during the MOA*-like search.

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

	Success/All	(*) Mean/Median/Max RT
NAMOA*-dr	16/50	25.4 / 92.9 / 539.8
ext-BOA*	17/50	11.6 / 40.1 / 222.7
ext-BOA*-lex	17/50	9.7 / 33.7 / 188.8
TOA* (ours)	**33/50**	**1.8 / 5.0 / 31.0**

Table 2: This table shows number of succeeded instances in New York City map from a online data set. Symbol (*) means the mean/median/max runtime (RT) are taken over the 16 instances where all four algorithms succeed. The mean, median and maximum number of Pareto-optimal solutions for those 16 instances are 389, 327 and 1061 respectively. The minimum runtime is omitted as there exists a trivial instance with only one Pareto-optimal solution and all algorithms terminate within a few micro-seconds. Our TOA* doubles the number of succeeded instances within a limited runtime of 600 seconds. Over the 16 solved instances, TOA* requires about 1/6 of the runtime of ext-BOA*-lex.

Conclusion

This work considers a Multi-Objective Shortest Path Problem (MO-SPP) with an arbitrary number of objectives. In this work, we observe that, during the search process of MOA*-like algorithms, the frontier set at each vertex is computed incrementally by solving the Dominance Check (DC) problem and Non-Dominated Set Update (NSU) problems iteratively. Based on this observation, we develop a balanced binary search tree (BBST)-based approach to efficiently solve the DC and NSU problems in the presence of an arbitrary number of objectives. With the help of the BBST-based methods and the existing fast dominance check techniques, we develop the Enhanced Multi-Objective A* (EMOA*), which computes all cost-unique Pareto-optimal paths for MO-SPP problems. EMOA* is a generalization of BOA*. We also develop the TOA*, an improved version of EMOA* when there are three objectives. We discuss the correctness and the computational complexity of the proposed methods, and verify them with massive tests. The numerical result shows that TOA* and EMOA* runs up to an order of magnitude faster than all the baselines, and are of particular advantage for problems with a large number of Pareto-optimal solutions.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 2120219 and 2120529. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

Ahmadi, S.; Tack, G.; Harabor, D. D.; and Kilby, P. 2021. Bi-objective Search with Bi-directional A*. In *Proceedings of the International Symposium on Combinatorial Search*, volume 12, 142–144.

Ehrgott, M. 2005. *Multicriteria optimization*, volume 491. Springer Science & Business Media.

Erkut, E.; Tjandra, S. A.; and Verter, V. 2007. Hazardous materials transportation. *Handbooks in operations research and management science*, 14: 539–621.

Goldin, B.; and Salzman, O. 2021. Approximate Bi-Criteria Search by Efficient Representation of Subsets of the Pareto-Optimal Frontier. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 31, 149–158.

Hansen, P. 1980. Bicriterion path problems. In *Multiple criteria decision making theory and application*, 109–127. Springer.

Kung, H.-T.; Luccio, F.; and Preparata, F. P. 1975. On finding the maxima of a set of vectors. *Journal of the ACM (JACM)*, 22(4): 469–476.

Loui, R. P. 1983. Optimal paths in graphs with stochastic or multidimensional weights. *Communications of the ACM*, 26(9): 670–676.

Mandow, L.; and De La Cruz, J. L. P. 2008. Multiobjective A* search with consistent heuristics. *Journal of the ACM (JACM)*, 57(5): 1–25.

Martins, E. Q. V. 1984. On a multicriteria shortest path problem. *European Journal of Operational Research*, 16(2): 236–245.

Montoya, J.; Rathinam, S.; and Wood, Z. 2013. Multiobjective departure runway scheduling using dynamic programming. *IEEE Transactions on Intelligent Transportation Systems*, 15(1): 399–413.

Perny, P.; and Spanjaard, O. 2008. Near admissible algorithms for multiobjective search. In *ECAI 2008*, 490–494. IOS Press.

Pulido, F.-J.; Mandow, L.; and Pérez-de-la Cruz, J.-L. 2015. Dimensionality reduction in multiobjective shortest path search. *Computers & Operations Research*, 64: 60–70.

Ren, Z.; Rathinam, S.; and Choset, H. 2021. Subdimensional Expansion for Multi-Objective Multi-Agent Path Finding. *IEEE Robotics and Automation Letters*, 6(4): 7153–7160.

Ren, Z.; Rathinam, S.; Likhachev, M.; and Choset, H. 2022. Multi-Objective Path-Based D* Lite. *IEEE Robotics and Automation Letters*, 7(2): 3318–3325.

Sanders, P.; and Mandow, L. 2013. Parallel label-setting multi-objective shortest path search. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, 215–224. IEEE.

Stern, R.; Sturtevant, N.; Felner, A.; Koenig, S.; Ma, H.; Walker, T.; Li, J.; Atzmon, D.; Cohen, L.; Kumar, T.; et al. 2019. Multi-Agent Pathfinding: Definitions, Variants, and Benchmarks. In *Symposium on Combinatorial Search*, 151158.

Stewart, B. S.; and White, C. C. 1991. Multiobjective A*. *Journal of the ACM (JACM)*, 38(4): 775–814.

Ulloa, C. H.; Yeoh, W.; Baier, J. A.; Zhang, H.; Suazo, L.; and Koenig, S. 2020. A Simple and Fast Bi-Objective Search Algorithm. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 30, 143–151.

Warburton, A. 1987. Approximation of Pareto optima in multiple-objective, shortest-path problems. *Operations research*, 35(1): 70–79.

Xu, J.; Spielberg, A.; Zhao, A.; Rus, D.; and Matusik, W. 2021. Multi-Objective Graph Heuristic Search for Terrestrial Robot Design. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*, 9863–9869. IEEE.