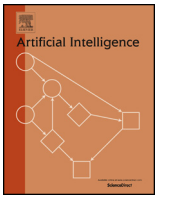
Artificial Intelligence 314 (2023) 103807

Contents lists available at ScienceDirect

Artificial Intelligence

journal homepage: www.elsevier.com/locate/artint



Simple and efficient bi-objective search algorithms via fast dominance checks

Carlos Hernández [a,*], William Yeoh [b], Jorge A. Baier [c,d], Han Zhang [e], Luis Suazo [a], Sven Koenig [e], Oren Salzman [f]

[a] Facultad de Ingeniería, Arquitectura y Diseño, Universidad San Sebastián, Bellavista 7, Santiago, 8420524, Chile
[b] Department of Computer Science and Engineering, Washington University in St. Louis, USA
[c] Departamento de Ciencia de la Computación, Pontificia Universidad Católica de Chile, Chile
[d] Millennium Institute for Foundational Research on Data, Chile
[e] Department of Computer Science, University of Southern California, USA
[f] Department of Computer Science, Technion–Israel Institute of Technology, Israel

ARTICLE INFO

Article history:
Received 18 November 2020
Received in revised form 11 July 2022
Accepted 11 October 2022
Available online 13 October 2022

Keywords:
A*
Bi-objective search
Dijkstra's algorithm
Heuristic search

ABSTRACT

Many interesting search problems can be formulated as bi-objective search problems, that is, search problems where two kinds of costs have to be minimized, for example, travel distance and time for transportation problems. Instead of looking for a single optimal path, we compute a Pareto-optimal frontier in bi-objective search, which is a set of paths in which no two paths dominate each other. Bi-objective search algorithms perform dominance checks each time a new path is discovered. Thus, the efficiency of these checks is key to performance. In this article, we propose algorithms for two kinds of bi-objective search problems. First, we consider the problem of computing the Pareto-optimal frontier of the paths that connect a given start state with a given goal state. We propose Bi-Objective A* (BOA*), a heuristic search algorithm based on A*, for this problem. Second, we consider the problem of computing one Pareto-optimal frontier for each state s of the search graph, which contains the paths that connect a given start state with s. We propose Bi-Objective Dijkstra (BOD), which is based on BOA*, for this problem. A common feature of BOA* and BOD is that all dominance checks are performed in constant time, unlike the dominance checks of previous algorithms. We show in our experimental evaluation that both BOA* and BOD are substantially faster than state-of-the-art bi-objective search algorithms.

1. Introduction

The A* algorithm [1] is at the core of many heuristic search algorithms developed for solving shortest-path problems due to its strong theoretical properties, especially when used in conjunction with consistent **h**-values. In such problems, one has to find a path from a given start state to a given goal state that minimizes the path cost. However, there are often two or more kinds of path costs in real life [2–4]. For example, government agencies that transport hazardous material need to find routes that do not only minimize the travel distance but also the risk of exposure for residents [2]. Motivated by

* Corresponding author.
E-mail addresses: carlos.hernandez@uss.cl (C. Hernández), wyeoh@wustl.edu (W. Yeoh), jabaier@ing.puc.cl (J.A. Baier), zhan645@usc.edu (H. Zhang), luis.suazo@live.cl (L. Suazo), skoenig@usc.edu (S. Koenig), osalzman@cs.technion.ac.il (O. Salzman).

https://doi.org/10.1016/j.artint.2022.103807
0004-3702/

such applications, researchers have extended A* to solve bi- and multi-objective shortest path problems where one wants to find the set of Pareto-optimal solutions from the start state to the goal state (or, synonymously, the optimal paths on the Pareto-optimal frontier), which is the set of paths that are not dominated by any path, where path p dominates path p' iff each kind of path cost of p is no larger than the corresponding kind of path cost of p' and at least one kind of path cost of p is smaller than the corresponding kind of path cost of p'.

Two such state-of-the-art extensions of A* are the *Multi-Objective A** (MOA*) [5] and *New Approach for MOA** (NAMOA*) [6] algorithms. These best-first multi-objective search algorithms differ from A* in various ways. The most relevant difference in the context of this article is that the concept of optimality is now related to (path) *dominance*. Since dominance checks are repeatedly performed throughout the execution of these algorithms, the time complexity of the checks plays a crucial role for their efficiency. For example, upon generating any search node, they need to check if the newly-found path to some state s is dominated by a previously-found path to s and, if so, discard the newly-found path. They also need to check whether a previously-found path to s is dominated by the newly-found path to s and, if so, discard the previously-found path.

NAMOA* performs all these dominance checks in a time that is linear in the size of the *Open* list and the number of paths found to a given state. Pulido et al. [7] proposed an improvement, called NAMOA*dr, that significantly improves the time complexity of some of these dominance checks to *constant* time. Unfortunately, the time complexity of other dominance checks remains *linear*.

In this article, we further improve the efficiency of the dominance checks. Our *Bi-Objective A** (BOA*) algorithm prunes dominated paths more efficiently by exploiting that there are only two kinds of path costs and that the **h**-values are consistent. It performs lazy dominance checks, that is, it does not check for dominance over nodes in *Open* when nodes are generated. This allows *all* dominance checks to be done in *constant time*, which results in a significant speedup, especially for large instances.

Moreover, we show how we can use constant-time dominance checks to solve the problem of computing the set of Pareto-optimal solutions from the start state to *all* states (and not only to the goal state). We call this algorithm *Bi-Objective Dijkstra* (BOD).[1]

Our extensive experimental results on road maps show that BOA* is faster than NAMOA*, NAMOA*dr, and the bi-objective search algorithms Bi-Objective Dijkstra [9] and Bidirectional Bi-Objective Dijkstra [9]. BOA* is especially faster in most instances, especially in the larger instances. We conclude the article by discussing how one might be able to improve and extend BOA*, including how to speed it up, find representative solutions on the Pareto-optimal frontier, find bounded-suboptimal solutions, and generalize it to problems with more than two kinds of path costs.

This article extends our ICAPS-20 publication [10] by including:

- a new *Bi-Objective Dijkstra* (BOD) algorithm,
- new detailed examples for the operation of BOA* [10] and BOD,
- new theoretical analyses for BOA* and BOD,
- new experimental results for BOA* and BOD, and
- a real-world application of BOA* for the hazardous material transport problem (HAZMAT) in Santiago, Chile.

2. Related work

Our problem of bi-objective search falls under the general problem of multi-objective optimization. Thus, we start with a general overview of the topic and then detail relevant work from the search literature.

2.1. Multi-objective optimization

Multi-objective optimization is the mathematical optimization problem involving more than one objective function. It has applications ranging from drug design (e.g., maximizing potency while minimizing synthesis costs and unwanted side effects) [11,12] to the optimization of building designs (e.g., minimizing cost, energy consumption and health hazards) [13]. In the general setting, we are given a *decision space* which is the space of all possible solutions. A solution can be evaluated using *objective functions* which are typically computable equations but might also be the results of physical experiments or computer simulations. The goal is to find a solution or set of solutions that is optimal. Here there are different notions of optimality that we will discuss shortly.

One approach to solve multi-objective optimization problems is via *scalarization techniques* [14]. Roughly speaking, in this approach the objective functions are aggregated (or reformulated as constraints), and then a constrained single-objective problem is solved. The exact method for which the multiple objectives are aggregated dictates the different solutions that can be obtained. A simple scalarization technique is to use linear weighting where non-negative weights are attached to each objective and the weighted sum of the objective functions is minimized. It can be shown that any such solution lies on

[1] The name "Dijkstra" in BOD is an homage to Edsger Dijkstra, but the algorithm is actually an adaptation of the Uniform-Cost Search algorithm and not Dijkstra's algorithm as it appears in standard textbooks. See [8] for a discussion of the subtle, yet important difference.

the Pareto-optimal frontier, regardless of the weights chosen.[2] Unfortunately, if the Pareto-optimal frontier is non convex,[3] then there may be solutions on the Pareto-optimal frontier that cannot be found using this approach, regardless of the weights chosen [15]. A different scalarization technique is called "ε-constrained". Here, one of the objectives is minimized while the rest are restricted within user-specific values. By varying these values, one can obtain different solutions on the Pareto-optimal frontier. However, it is difficult in practice to choose these values. For other scalarization techniques see, e.g., [15].

An alternative approach to solving multi-objective optimization problems is via evolutionary algorithms which use mechanisms inspired by biological evolution, such as reproduction, mutation, recombination, and selection to devise generic population-based metaheuristic optimization algorithms. In the context of multiobjective optimization these algorithms gradually approach sets of Pareto-optimal solutions. Generally speaking, the algorithms differ in the paradigms used to define the selection operators. For additional details see, e.g., [16–19].

2.2. Multi-objective shortest path

A variant of multi-objective optimization that is the focus of this article is the multi-objective shortest-path problem, an extension to the classical (single-objective) shortest-path problem. Unfortunately, the general problem is NP-hard [20] and even determining whether a path belongs to the Pareto-optimal frontier is NP-hard [21]. Moreover, the cardinality of the size of the Pareto-optimal frontier may be exponential in the number of graph vertices [22,23]. Existing methods either try to efficiently compute the Pareto-optimal frontier or to relax the problem and only compute an approximation of this set.

2.2.1. Efficient computation of the Pareto-optimal frontier

To efficiently compute the Pareto-optimal frontier, adaptations of the celebrated Dijkstra and A* algorithms [1] were suggested. The first Dijkstra-based algorithms were suggested by Hansen [24] and Martins [25] for the bi-objective and multi-objective search problems, respectively. These were later extended using different techniques from single-objective search such as bi-directional search [9,26,27] and depth-first-search (DFS) [28].

As we have already mentioned, Stewart and White III [5] introduced Multi-Objective A* (MOA*) which is a multiobjective extension of A*. The most notable difference between MOA* and A* is in maintaining the Pareto-optimal frontier to intermediate vertices. MOA* was later revised [7,6,29] and the work we present here can be considered as another revision that obtains its computational efficiency via efficient $O(1)$ dominance checks.

2.2.2. Approximating the Pareto-optimal frontier

Initial methods in computing an approximation of the Pareto-optimal frontier were directed towards devising a Fully Polynomial Time Approximation Scheme (FPTAS) [30].[4] Warburton [31] proposed a method for finding an approximate Pareto-optimal solution to the problem for any degree of accuracy using scaling and rounding techniques. Perny and Spanjaard [32] presented another FPTAS given that a finite upper bound L on the numbers of arcs of all solution-paths in the Pareto-optimal frontier is known. This requirement was later relaxed [23,33] by partitioning the space of solutions into cells according to the approximation factor and, roughly speaking, take only one solution in each grid cell. Unfortunately, the running times of FPTASs are typically polynomials of high degree, and hence they may be slower than exact approaches when applied to relatively-small instances and running them on graphs with even a moderate number of states (e.g., $\approx 10,000$) is often impractical [23].

A different approach to compute a subset of the Pareto-optimal solution is to find all extreme supported non-dominated points (i.e., the extreme points on the convex hull of the Pareto-optimal set) [34]. Taking a different approach, Legriel et al. [35] suggest a method based on satisfiability/constraint solvers. Alternatively, a simple variation of MOA*, termed MOA$^*_\varepsilon$ allows to compute an approximation of the Pareto-optimal frontier by pruning intermediate paths that are approximately dominated by already-computed solutions [32].

3. Preliminaries

A bi-objective *search graph* is a tuple $(S, E, \mathbf{c})$, where S is the finite set of *states*, $E \subseteq S \times S$ is the finite set of edges, and $\mathbf{c} : E \to \mathbb{R}^{\geq 0} \times \mathbb{R}^{\geq 0}$ is a *cost function* that associates a pair of non-negative real costs with each edge. $Succ(s) = \{t \in S \mid (s,t) \in E\}$ denotes the successors of state s. A *path* from s_1 to s_n is a sequence of states $s_1, s_2, \ldots, s_n$ such that $(s_i, s_{i+1}) \in E$ for all $i \in \{1, \ldots, n-1\}$.

Boldface font indicates pairs. p_1 denotes the first component of pair $\mathbf{p}$, and p_2 denotes its second component; that is, $\mathbf{p} = (p_1, p_2)$. The addition of two pairs $\mathbf{p}$ and $\mathbf{q}$ and the multiplication of a real-valued scalar k and a pair $\mathbf{p}$ are defined in the natural way, namely as $\mathbf{p} + \mathbf{q} = (p_1 + q_1, p_2 + q_2)$ and $k\mathbf{p} = (kp_1, kp_2)$, respectively. $\mathbf{p} \prec \mathbf{q}$ denotes that ($p_1 < q_1$ and

[2] Solutions lying on the Pareto-optimal frontier are often called "efficient solutions" by the OR community.

[3] A formal definition of convexity in this context requires some additional notation that are out of the scope of this article. For additional details see, e.g., [15].

[4] An FPTAS is an approximation scheme whose time complexity is polynomial in the input size and also polynomial in $1/\varepsilon$ where ε is the approximation factor.

$p_2 \leq q_2$) or ($p_1 \leq q_1$ and $p_2 < q_2$). In this case, we say that $\mathbf{p}$ *dominates* $\mathbf{q}$. $\mathbf{p} \leq \mathbf{q}$ denotes that $p_1 \leq q_1$ and $p_2 \leq q_2$. In this case, we say that $\mathbf{p}$ *weakly dominates* $\mathbf{q}$. $P \prec \mathbf{q}$ (resp. $P \leq \mathbf{q}$) for a set P of pairs denotes that there exists a $\mathbf{p} \in P$ such that $\mathbf{p} \prec \mathbf{q}$ (resp. $\mathbf{p} \leq \mathbf{q}$).

$\mathbf{c}(\pi) = \sum_{i=1}^{n-1} \mathbf{c}(s_i, s_{i+1})$ is the cost of path $\pi = s_1, \ldots, s_n$. $\pi \prec \pi'$ (resp. $\pi \leq \pi'$) for two paths π and π' denotes that $\mathbf{c}(\pi) \prec \mathbf{c}(\pi')$ (resp. $\mathbf{c}(\pi) \leq \mathbf{c}(\pi')$). In this case, we say that π *dominates* (resp. *weakly dominates*) π'.

A single-source search instance is defined as a tuple $P = (S, E, \mathbf{c}, s_{start})$, where $(S, E, \mathbf{c})$ is a search graph and $s_{start} \in S$ is the *start state*.

Given a single-source search instance $P = (S, E, \mathbf{c}, s_{start})$, a *Pareto-optimal solution set for a state* $s \in S$, denoted by $sols(s)$, contains every path π from the start state to s with the property that, for every other path π' from the start state to s, $\pi' \not\prec \pi$; that is, $sol(s)$ contains all non-dominated paths from the start state to s. In this article, we are interested in finding any maximal subset of the Pareto-optimal solution set such that any two solutions in the subset do not have the same cost and refer to this subset as the *cost-unique Pareto-optimal solution set*.

In many applications, we are given a *goal state* $s_{goal} \in S$ and need to find the Pareto-optimal solution set for the goal state only. In this case, similar to A*, we define $\mathbf{h}$-values $\mathbf{h} : S \rightarrow \mathbb{R}^{\geq 0} \times \mathbb{R}^{\geq 0}$. The $\mathbf{h}$-value $\mathbf{h}(s)$ estimates the cost of a path from state s to the goal state. $\mathbf{h}$ is admissible iff $\mathbf{h}(s) \leq \mathbf{c}(\pi)$ for all states s and all paths π from s to the goal state, that is, both components of $\mathbf{h}$ are admissible for the corresponding components of the cost function. Similarly, $\mathbf{h}$ is consistent iff (1) $\mathbf{h}(s_{goal}) = (0, 0)$ and (2) $\mathbf{h}(s) \leq \mathbf{c}(s, t) + \mathbf{h}(t)$ for all $(s, t) \in E$, that is, both components of $\mathbf{h}$ are consistent for the corresponding components of the cost function. We also define $\mathbf{g}$- and $\mathbf{f}$-values $\mathbf{g}$ and $\mathbf{f} : S \rightarrow \mathbb{R}^{\geq 0} \times \mathbb{R}^{\geq 0}$. The $\mathbf{g}$-value $\mathbf{g}(s)$ stores the computed cost of a path from the start state to state s, and the $\mathbf{f}$-value $\mathbf{f}(s)$ is the sum of the $\mathbf{g}$- and $\mathbf{h}$-values of state s.

4. Algorithmic background: best-first bi-objective search

In this section, we study the problem of computing a Pareto-optimal solution set for a single goal state with best-first search. In Section 6, we study the problem of computing solution sets for every state.

Open list We can compute the Pareto-optimal solution set with a modified version of A* that maintains an *Open* list, containing the frontier of the search tree (that is, the generated but not yet expanded nodes), and, optionally, a *Closed* list, containing the interior of the search tree (that is, the expanded nodes). A node is associated with a state, a $\mathbf{g}$-value, an $\mathbf{h}$-value, and an $\mathbf{f}$-value and corresponds to a path to the state of a cost that is equal to the $\mathbf{g}$-value. Different from A*, the $\mathbf{g}$-, $\mathbf{h}$-, and $\mathbf{f}$-values are tuples rather than scalars. Also different from A*, the *Open* list might contain different nodes with the same state, corresponding to different paths to the same state, since we need to compute Pareto-optimal solution sets rather than a single solution.

Node selection The algorithm repeatedly extracts a node from the *Open* list. To guarantee optimality, the $\mathbf{f}$-value of the extracted node must not be dominated by the $\mathbf{f}$-value of any node in the *Open* list.

Solution recording When the algorithm extracts a node with the goal state, the path corresponding to the node is a solution. Different from A*, the algorithm cannot terminate and return this solution since it has to compute the Pareto-optimal solution set. Thus, it checks whether this solution is dominated by a previously-found solution. If not, then it adds this solution to the solution set and removes all solutions from the solution set that are dominated by this solution. In both cases, it continues the search.

Node expansion When the algorithm extracts a node with a non-goal state, it expands the extracted node. Let the extracted node have state s. The algorithm then generates the child nodes of the extracted node, one for each successor t of s, by adding them to the *Open* list. It terminates when the *Open* list is empty and returns the solution set.

Efficiency We can improve the efficiency of the algorithm by performing the dominance checks not once it has found a solution but earlier. Assume that the algorithm has extracted a node and is about to generate a child node with state t. The $\mathbf{f}$-value of the child node is a lower bound on the costs of all solutions that complete the path that the child node corresponds to. Thus, the algorithm does not need to generate the child node if the $\mathbf{f}$-value of the child node is dominated by the $\mathbf{f}$-value (that is, the cost) of a solution in the solution set. Similarly, the algorithm does not need to generate the child node if the $\mathbf{f}$-value of the child node is dominated by the $\mathbf{f}$-value of a node with state t that has already been generated (corresponding to a path to t that has already been found). In addition, it can remove all paths to t from the *Open* list whose $\mathbf{f}$-values are dominated by the $\mathbf{f}$-value of the newly-found path to t. If t is the goal state, it also has to remove all solutions from the solution set whose $\mathbf{f}$-values (that is, costs) are dominated by the $\mathbf{f}$-value (that is, cost) of the newly-found solution.

4.1. The NAMOA* algorithm

NAMOA* [6] is a best-first multi-objective search algorithm that provides the foundation for most multi-objective search algorithms. Algorithm 1 shows its pseudocode for bi-objective search problems. It takes as input a bi-objective search problem and consistent $\mathbf{h}$-values and computes the Pareto-optimal solution set. We describe its key elements in the following.

Variables Each node in the *Open* list is a triple of the form $(s, \mathbf{g}_s, \mathbf{f}_s)$ with state s, $\mathbf{g}$-value $\mathbf{g}_s$, and $\mathbf{f}$-value $\mathbf{f}_s$ and corresponds to a path to s of cost $\mathbf{g}_s$. In addition, NAMOA* maintains parents. Different from A*, a parent is a set of $\mathbf{g}$-values

Algorithm 1: The NAMOA* algorithm.

```
   Input  : A search problem (S, E, c, s_start, s_goal) and consistent h-values h
   Output: The Pareto-optimal solution set
 1 sols ← ∅
 2 for each s ∈ S do
 3     G_op(s) ← ∅; G_cl(s) ← ∅
 4 G_op(s) ← {(0, 0)}
 5 parent((0, 0)) ← ∅
 6 Initialize Open and add (s_start, (0, 0), h(s_start)) to it
 7 while Open ≠ ∅ do
 8     Remove a node (s, g_s, f_s) from the Open list with the lexicographically smallest f-value of all nodes in the Open list
 9     Remove g_s from G_op(s) and add it to G_cl(s)
10     if s = s_goal then
11         Add g_s to sols
12         Remove all nodes (u, g_u, f_u) with f_s ≺ f_u from the Open list
13         continue
14     for each t ∈ Succ(s) do
15         g_t ← g_s + c(s, t)
16         if g_t ∈ G_op(t) ∪ G_cl(t) then
17             Add g_s to parent(g_t)
18             continue
19         if G_op(t) ∪ G_cl(t) ≺ g_t then
20             continue
21         f_t ← g_t + h(t)
22         if sols ≺ f_t then
23             continue
24         Remove all g-values g'_t from G_op(t) that are dominated by g_t and remove their corresponding nodes (t, g'_t, f'_t) from the Open list
25         Remove all g-values from G_cl(t) that are dominated by g_t
26         parent(g_t) ← {g_s}
27         Add g_t to G_op(t)
28         Add (t, g_t, f_t) to the Open list
29 return sols
```

of some of the predecessors of s (rather than a single predecessor) and is associated with $\mathbf{g}$-value $\mathbf{g}_s$ (rather than state s). Also different from A*, NAMOA* maintains two sets of $\mathbf{g}$-values for state s, namely $\mathbf{G}_{cl}(s)$, which contains the $\mathbf{g}$-values of all expanded nodes with state s, and $\mathbf{G}_{op}(s)$, which contains the $\mathbf{g}$-values of all generated but not yet expanded nodes with state s.

Node selection NAMOA* always extracts a node from the *Open* list whose $\mathbf{f}$-value is not dominated by the $\mathbf{f}$-value of any node in the *Open* list. Such a node can be identified efficiently for bi-objective search problems as a node in the *Open* list with the lexicographically smallest $\mathbf{f}$-value (f_1, f_2) of all nodes in the *Open* list (Line 8). To see why this is correct, let (f'_1, f'_2) be the $\mathbf{f}$-value of any node in the *Open* list. Then, either (1) $f_1 = f'_1$ and $f_2 \leq f'_2$ or (2) $f_1 < f'_1$. In both cases, $(f'_1, f'_2) \not\prec (f_1, f_2)$; that is, (f_1, f_2) is not dominated by the $\mathbf{f}$-value of any node in the *Open* list. Consequently, the nodes in the *Open* list should be ordered in increasing lexicographic order of their $\mathbf{f}$-values.

Solution recording When NAMOA* extracts a node with the goal state, it has found an undominated solution. In this case, it adds the $\mathbf{g}$-value of the node to the solution set and removes all nodes from the *Open* list whose $\mathbf{f}$-values are dominated by the $\mathbf{f}$-value of the node (Lines 10-13).

Node expansion When NAMOA* extracts a node with a non-goal state, it expands the extracted node $(s, \mathbf{g}_s, \mathbf{f}_s)$ by calculating its child nodes $(t, \mathbf{g}_t, \mathbf{f}_t)$, one for each successor t of state s. If it has generated a node with state t and $\mathbf{g}$-value $\mathbf{g}_t$ before, then it adds $\mathbf{g}_s$ to the parent set $parent(\mathbf{g}_t)$ (Lines 16-18) (which corresponds to recording another path to t of cost $\mathbf{g}_t$ and is necessary since NAMOA* computes the Pareto-optimal solution set rather than a single solution). In this case, it does not add the child node to the *Open* list. Neither does it add the child node to the *Open* list if $\mathbf{g}_t$ is dominated by the $\mathbf{g}$-value of a generated node with state t (Lines 19-20) (which corresponds to pruning the newly-found path to t since it is dominated by another path to t that has already been found). Neither does it add the child node to the *Open* list if the $\mathbf{f}$-value $\mathbf{f}_t$ is dominated by the $\mathbf{f}$-value (that is, $\mathbf{g}$-value and cost) of a solution in the solution set (Lines 22-23) (which corresponds to pruning the newly-found path to t since it is dominated by a solution that has already been found). Otherwise, it generates the child node by adding it to the *Open* list, adding $\mathbf{g}_t$ to $\mathbf{G}_{op}(t)$, making $\mathbf{g}_s$ the only $\mathbf{g}$-value in the parent set $parent(\mathbf{g}_t)$ (which corresponds to recording the first path to t of cost $\mathbf{g}_t$), and removing all references to paths to t from the *Open* list, $\mathbf{G}_{op}(t)$, and $\mathbf{G}_{cl}(t)$ that are dominated by the newly-found path to t (Lines 24-28). It terminates when the *Open* list is empty and returns the solution set (Line 29).

Algorithm 2: The NAMOA*dr algorithm.

```
   Input  : A search problem (S, E, c, s_start, s_goal) and consistent h-values h
   Output: The Pareto-optimal solution set
 1 sols ← ∅
 2 for each s ∈ S do
 3     G_op(s) ← ∅; G_cl(s) ← ∅; g_2^min(s) ← ∞
 4 G_op(s_start) ← {(0, 0)}
 5 parent((0, 0)) ← ∅
 6 Initialize the Open list and add (s_start, (0, 0), h(s_start)) to it
 7 while Open ≠ ∅ do
 8     Remove a node (s, g_s, f_s) from the Open list with the lexicographically smallest f-value of all nodes in the Open list
 9     Remove g_s from G_op(s) and add it to G_cl(s)
10     if g_2^min(s_goal) ≤ g_{s,2} then
11         continue /* prune s if sols ≺ g_s                                                                    */
12     if s = s_goal then
13         Add g_s to sols
14         continue
15     g_2^min(s) ← g_{s,2}
16     for each t ∈ Succ(s) do
17         g_t ← g_s + c(s, t)
18         if g_t ∈ G_op(t) ∪ G_cl(t) then
19             Add g_s to parent(g_t)
20             continue
21         if g_2^min(t) ≤ g_{t,2} or G_op(t) ≺ g_t then
22             continue
23         f_t ← g_t + h(t)
24         if g_2^min(s_goal) ≤ f_{t,2} then
25             continue
26         Remove all g-values g'_t from G_op(t) that are dominated by g_t and remove their corresponding nodes (t, g'_t, f'_t) from the Open list
27         parent(g_t) ← {g_s}
28         Add g_t to G_op(t)
29         Add (t, g_t, f_t) to the Open list
30 return sols
```

4.2. The NAMOA*dr algorithm

Some of the operations of NAMOA* are time-consuming since they perform dominance checks that involve either the **f**-values (Lines 12 and 22) or **g**-values (Lines 24-25) and require it to iterate over a number of elements proportional to $|\mathbf{G}_{op}(t)|$, $|\mathbf{G}_{cl}(t)|$, $|Open|$, or $|sols|$. Pulido et al. [7] (in short: PMP) improved NAMOA* to NAMOA*dr by observing that, if (A1) consistent **h**-values are used and (A2) the *Open* list is sorted lexicographically, then a number of NAMOA*'s dominance checks can be implemented in constant rather than linear time.

PMP observed that some dominance checks can be carried out more efficiently by reducing one of the dimensions of the vectors involved. To see how this is done, assume that we receive as input a list ν of 2-dimensional vectors $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n$, which are sorted lexicographically. We want to construct a largest subset of vectors that do not dominate each other by considering the vectors in ν one by one, in order. Assume that V is the set of non-dominated vectors that we have constructed after considering the first k vectors in ν. Vector $\mathbf{v}_{k+1}$ cannot dominate any vector in V because of the way the vectors in ν are arranged. Therefore, $\mathbf{v}_{k+1}$ is added to V if it is not dominated by any vector in V (that is, $V \not\prec \mathbf{v}_{k+1}$), and discarded otherwise. By keeping track of the minimum m of the second components of all vectors in V, the dominance check can be done in constant time since $V \prec \mathbf{v}_{k+1}$ if and only if the second component of $\mathbf{v}_{k+1}$ is smaller than m.

PMP's insight can be used to implement NAMOA* more efficiently. Since we use consistent **h**-values and the *Open* list is lexicographically ordered, each time we extract a state s from the *Open* list, the g_1-value of this state is greater or equal to the g_1-values of all previously-found paths to s. This fact can be used to implement Lines 19 and 22 of NAMOA* in constant time when checking whether the **g**-value of a state t is dominated by the **g**-value of any state in $\mathbf{G}_{cl}(t)$.

Algorithm 2 shows the pseudocode of NAMOA*dr, a bi-objective version of NAMOA* based on PMP's multi-objective search algorithm. The differences between NAMOA*dr and NAMOA* are highlighted in blue. The key conceptual difference between the algorithms is that NAMOA*dr keeps a $g_2^{\min}$-value for every state of the search graph. $g_2^{\min}(s)$ is the minimum g_2-value of a path from the start state to s that has been extracted from the *Open* list. Such a value is used to implement NAMOA*'s dominance checks of Lines 19 and 22 of Algorithm 1 in constant time. In addition, following PMP, NAMOA*dr, adds the pruning of Lines 10–11 (Algorithm 2), which discards a node just extracted from the *Open* list if its cost is dominated by the cost of a solution. This check is also implemented in constant time. Finally, NAMOA*dr returns a cost-unique Pareto-optimal set.

Algorithm 3: The Bi-Objective A* (BOA*) algorithm.

```
   Input  : A search problem (S, E, c, s_start, s_goal) and a consistent heuristic function h
   Output: A cost-unique Pareto-optimal solution set
 1 for each s ∈ S do
 2     sols(s) ← ∅
 3     g_2^min(s) ← ∞
 4 x ← new node with s(x) = s_start
 5 g(x) ← (0, 0)
 6 parent(x) ← null
 7 f(x) ← (h_1(s_start), h_2(s_start))
 8 Initialize Open and add x to it
 9 while Open ≠ ∅ do
10     Remove a node x from Open with the lexicographically smallest f-value of all nodes in Open
11     if g_2(x) ≥ g_2^min(s(x)) ∨ f_2(x) ≥ g_2^min(s_goal) then
12         continue
13     g_2^min(s(x)) ← g_2(x)
14     Add x to sols(s(x))
15     if s(x) = s_goal then
16         continue
17     for each t ∈ Succ(s(x)) do
18         y ← new node with s(y) = t
19         g(y) ← g(x) + c(s(x), t)
20         parent(y) ← x
21         f(y) ← g(y) + h(t)
22         if g_2(y) ≥ g_2^min(t) ∨ f_2(y) ≥ g_2^min(s_goal) then
23             continue
24         Add y to Open
25 return sols(s_goal)
```

5. The Bi-Objective A* (BOA*) algorithm

The improvements to NAMOA* proposed by PMP remove some, but not all, of its most time-consuming operations. Specifically, it still iterates over a number of **g**-values proportional to $|\mathbf{G}_{op}(t)|$ on Lines 21 and 26 (Algorithm 2).

In this section, we therefore describe our *Bi-Objective A** (BOA*) algorithm, a best-first bi-objective search algorithm. Our primary design objective is to perform all dominance checks in constant time. We use PMP's ideas and additional insights: (1) to avoid having to maintain the sets $\mathbf{G}_{op}(s)$ and $\mathbf{G}_{cl}(s)$ for all states s and thus not having to perform any of the eager checks on Line 26 (Algorithm 2) to remove **g**-values from these sets and (2) to make the eager check on Line 21 (Algorithm 2) more efficient by maintaining a value $g_2^{\min}(s)$ for each state s, which is the smallest g_2-value of any expanded node with state s.

A secondary design objective is to make the presentation of BOA* similar to that of modern descriptions of A*, such as those in [36], thereby making it potentially easier to understand and implement. Another design objective is to compute the cost-unique Pareto-optimal solution set rather than the Pareto-optimal solution set since it is sufficient for our purposes to compute one representative solution for all cost-identical and thus equally good solutions.

The *Open* list of BOA* contains *nodes*, which are akin to the *labels* commonly used in the operations research literature [37]. Each node x has a state $s(x)$, a **g**-value $\mathbf{g}(x)$, an **f**-value $\mathbf{f}(x)$, and a parent $parent(x)$ and corresponds to a path to $s(x)$ of cost $\mathbf{g}(x)$. The parent is a single node.

Algorithm 3 shows the pseudocode of BOA*. It takes as input a bi-objective search problem and consistent **h**-values and computes the cost-unique Pareto-optimal solution set. In each iteration, it extracts a node x from the *Open* list with the lexicographically smallest **f**-value of all nodes in the *Open* list (Line 10). It does not expand the node if its g_2-value is at least $g_2^{\min}(s(x))$ or its f_2-value is at least $g_2^{\min}(s_{goal})$ (Lines 11-12). Otherwise, it updates $g_2^{\min}(s(x))$ (Line 13) and expands the node. If $s(x)$ is the goal state, then BOA* has found an undominated solution and adds node x to the solution set *sols* (Lines 14-16). Otherwise, it calculates the child nodes of node x (Lines 18-21). It does not add a child node y to the *Open* list if its g_2-value is at least $g_2^{\min}(s(y))$ or its f_2-value is at least $g_2^{\min}(s_{goal})$ (Lines 22-23). Otherwise, it generates the child node by adding it to the *Open* list (Line 24). It terminates when the *Open* list is empty and returns the solution set (Line 25).

5.1. Relationship to the NAMOA*dr algorithm

The main difference between NAMOA*dr and BOA* is that BOA* avoids all linear-time dominance checks. Even though NAMOA*dr avoids linear-time dominance checks when checking whether the current **g**-value of node t is dominated by the **g**-value of an element in $\mathbf{G}_{cl}(t)$ or *sols*, it has to iterate through the elements in $\mathbf{G}_{op}(t)$ (Line 26, Algorithm 2) to check whether the current **g**-value of state t is dominated by the **g**-value of an element in $\mathbf{G}_{op}(t)$ before adding t to the *Open*

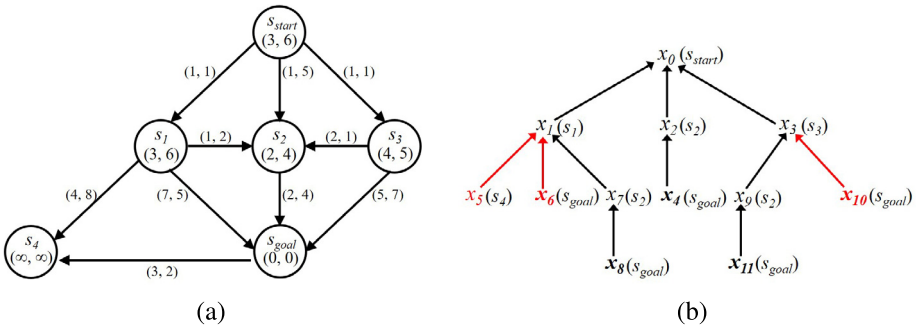


Fig. 1. (a) An example search graph, in which pairs of numbers inside each state indicate its **h**-value. (b) Search tree of Bi-Objective A* for the example graph. Red edges are pruned successors. (For interpretation of the colors in the figure(s), the reader is referred to the web version of this article.)

Table 1
States expanded by BOA* run on the graph of Fig. 1 (a). Nodes in red are pruned.

iteration	*node*	*parent*(*node*)	*s*(*node*)	**g**(*node*)	**f**(*node*)	$g_2^{\min}(s(node))$
1	x_0	*null*	s_{start}	(0, 0)	(3, 6)	0
4	x_1	x_0	s_1	(1, 1)	(4, 7)	1
2	x_2	x_0	s_2	(1, 5)	(3, 9)	5
7	x_3	x_0	s_3	(1, 1)	(5, 6)	1
3	x_4	x_2	s_{goal}	(3, 9)	(3, 9)	9
10	x_6	x_1	s_{goal}	(8, 6)	(8, 6)	
5	x_7	x_1	s_2	(2, 3)	(4, 7)	3
6	x_8	x_7	s_{goal}	(4, 7)	(4, 7)	7
8	x_9	x_3	s_2	(3, 2)	(5, 6)	2
9	x_{11}	x_9	s_{goal}	(5, 6)	(5, 6)	6

Table 2
States pruned by BOA* upon generation when run on the graph of Fig. 1 (a). Nodes in red are pruned.

iteration	*node*	*parent*(*node*)	*s*(*node*)	**g**(*node*)	**f**(*node*)	$g_2^{\min}(s(node))$
4	x_5	x_1	s_4	(5, 7)	(∞, ∞)	∞
7	x_{10}	x_3	s_{goal}	(6, 8)	(6, 8)	

list since PMP's constant-time dominance check cannot be used for these dominance checks. BOA*, instead, does not check whether a node just generated has a **g**-value that is dominated by the **g**-value of a node in the *Open* list. Instead, it delays such a check until expansion time, using the condition of Line 11 (Algorithm 3). Thus, a way of interpreting the difference between BOA* and NAMOA*dr is that some of BOA*'s dominance checks are lazy in comparison to the eager ones of NAMOA*dr.

5.2. Example

Fig. 1 (a) shows a graph with six states and ten edges, where s_{start} is the source state and s_{goal} is the goal state. We use the perfect distances as **h**-values, which can be computed with Dijkstra's algorithm. Table 1 shows the expanded states (in black) and those states pruned by Line 11 (in red), for each iteration of BOA*. For each node expansion, the table shows the node removed from the *Open* list for expansion (*node*), the parent of the node (*parent*(*node*)), the state of the node (*s*(*node*)), the **g**-values, the **f**-values and the $g_2^{\min}$ of the state. Note that all $g_2^{\min}$-values are initialized to ∞. Finally, Table 2 shows the nodes that are pruned by BOA* in Line 22, right after generation.

- In iteration 1, the node x_0 is removed from the *Open* list, and its three child nodes x_1, x_2, and x_3 are added to the *Open* list. $g_2^{\min}(s_{start})$ is updated from ∞ to 0.
- In iteration 2, node x_2 is removed from the *Open* list, and its child node x_4 is added to the *Open* list. $g_2^{\min}(s_2)$ is updated from ∞ to 5.
- In iteration 3, node x_4 is removed from the *Open* list. A non-dominated path is found because $s(x_4)$ is equal to s_{goal}. $g_2^{\min}(s_{goal})$ is updated from ∞ to 9.
- In iteration 4, node x_1 is removed from the *Open* list, and its two child nodes x_6 and x_7 are added to the *Open* list. The child node x_5 is pruned (in Line 22) because $f_2(x_5) = \infty \geq g_2^{\min}(s_{goal}) = 9$. $g_2^{\min}(s_1)$ is updated from ∞ to 1.

Algorithm 4: The Bi-Objective Dijkstra's (BOD) algorithm.

```
   Input  : A bi-objective weighted graph (S, E, c, s_start)
   Output: A cost-unique Pareto-optimal solution set for each state in S
 1 for each s ∈ S do
 2     sols(s) ← ∅
 3     g_2^min(s) ← ∞
 4 x ← new node with s(x) = s_start
 5 g(x) ← (0, 0)
 6 parent(x) ← null
 7 Initialize the Open list and add x to it
 8 while Open ≠ ∅ do
 9     Remove a node x from the Open list with the lexicographically smallest g-value of all nodes in the Open list
10     if g_2(x) ≥ g_2^min(s(x)) then
11         continue
12     g_2^min(s(x)) ← g_2(x)
13     Add x to sols(s(x))
14     for each t ∈ Succ(s(x)) do
15         y ← new node with s(y) = t
16         g(y) ← g(x) + c(s(x), t)
17         parent(y) ← x
18         if g_2(y) ≥ g_2^min(t) then
19             continue
20         Add y to the Open list
21 return sols
```

- In iteration 5, node x_7 is removed from the *Open* list, and its child node x_8 is added to the *Open* list. $g_2^{\min}(s_2)$ is updated from 5 to 3.
- In iteration 6, node x_8 is removed from the *Open* list. A non-dominated path is found because $s(x_8)$ is equal to s_{goal}. $g_2^{\min}(s_{goal})$ is updated from 9 to 7.
- In iteration 7, node x_3 is removed from the *Open* list, and its child node x_9 is added to the *Open* list. The child node x_{10} is pruned (in Line 22) because $g_2(x_{10}) = 8 \geq g_2^{\min}(s_{goal}) = 7$. $g_2^{\min}(s_3)$ is updated from ∞ to 1.
- In iteration 8, node x_9 is removed from the *Open* list, and its child node x_{11} is added to the *Open* list. $g_2^{\min}(s_2)$ is updated from 3 to 2.
- In iteration 9, node x_{11} is removed from the *Open* list. A non-dominated path is found because $s(x_{11})$ is equal to s_{goal}. $g_2^{\min}(s_{goal})$ is updated from 7 to 6.
- In iteration 10, node x_6 is removed from the *Open* list. The node is pruned (in Line 11) because $g_2(x_6) = 6 \geq g_2^{\min}(s_{goal}) = 6$.

BOA* found three solutions of cost $(3, 9)$, $(4, 7)$ and $(5, 6)$. Fig. 1 (b) shows the search tree of BOA*. Pruned nodes are in red, and the solution nodes are in boldface.

6. The Bi-Objective Dijkstra's (BOD) algorithm

In this section, we describe how BOA* can be modified to find Pareto-optimal solution sets for *every* state in the search graph. We call our algorithm Bi-Objective Dijkstra (BOD).

Uniform cost search without a goal condition, which is the kind of search that results when running A* with zero **h**-values for all states in a search graph, is equivalent to Dijkstra's algorithm. However, Felner [8] shows that implementing Dijkstra's algorithm as a uniform-cost search results in a faster implementation compared to a standard textbook implementation, when determining all single-source shortest paths.

We therefore design BOD by converting BOA* to a uniform-cost search without a goal condition. To this end, we use zero **h**-values. This results in the following modifications to Algorithm 3. First, in Line 7, we replace the **h**-values with zero. The *Open* list is thus a priority queue ordered by the **g**-values. Second, we modify the pruning conditions of Lines 11 and 22 so that they no longer refer to s_{goal}.

Fig. 2 (a) shows a graph with six states and ten edges, where s_{start} is the source state. Table 3 shows an execution trace of BOD run on that graph. For each iteration, Table 3 shows the node removed from the *Open* list for expansion (*node*), the parent of the node (*parent*(*node*)), the state of the node (s(*node*)), and the **g**-value of the node (**g**(*node*)). If *node* is pruned by Line 10, the entry is shown in red. Otherwise, the entry is shown in black, and the $g_2^{\min}$-value of the state associated with the node ($g_2^{\min}(s(node))$) after it is updated on Line 12 is also shown. Note that, when a node x is expanded (that is, it is not pruned on Line 10), a new non-dominated path from the start state to state $s(x)$ has been found.

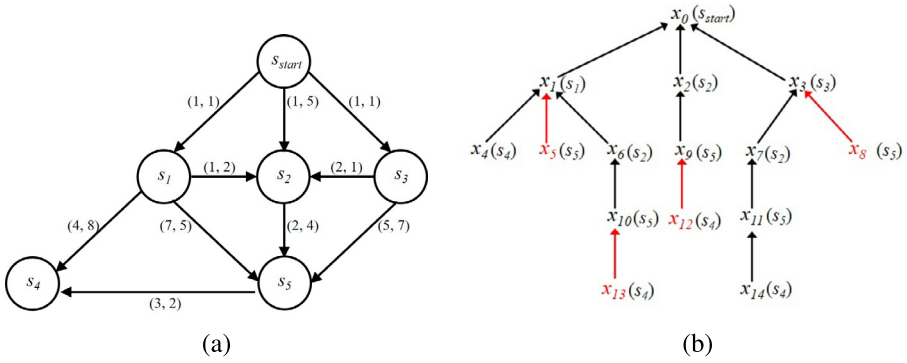


Fig. 2. (a) An example search graph, and (b) a search tree corresponding to a run of BOD on the graph. Red edges are pruned.

Table 3
An execution trace of BOD run on the graph of Fig. 2 (a). Nodes in red are pruned.

iteration	*node*	*parent*(*node*)	*s*(*node*)	**g**(*node*)	$g_2^{\min}(s(node))$
1	x_0	*null*	s_{start}	(0, 0)	0
2	x_1	x_0	s_1	(1, 1)	1
4	x_2	x_0	s_2	(1, 5)	5
3	x_3	x_0	s_3	(1, 1)	1
10	x_4	x_1	s_4	(5, 9)	9
14	x_5	x_1	s_5	(8, 6)	
5	x_6	x_1	s_2	(2, 3)	3
6	x_7	x_3	s_2	(3, 2)	2
11	x_8	x_3	s_5	(6, 8)	
7	x_9	x_2	s_5	(3, 9)	9
8	x_{10}	x_6	s_5	(4, 7)	7
9	x_{11}	x_7	s_5	(5, 6)	6
13	x_{12}	x_9	s_4	(6, 11)	
12	x_{13}	x_{10}	s_4	(7, 9)	
15	x_{14}	x_{11}	s_4	(8, 8)	8

- In iteration 1, the node x_0 is removed from the *Open* list, and its three child nodes x_1, x_2, and x_3 are added to the *Open* list. $g_2^{\min}(s_{start})$ is updated from ∞ to 0.
- In iteration 2, node x_1 is removed from the *Open* list, and its three child nodes x_4, x_5 and x_5 are added to *Open* list. $g_2^{\min}(s_1)$ is updated from ∞ to 1.
- In iteration 3, node x_3 is removed from the *Open* list, and its two child nodes x_7 and x_8 are added to the *Open* list. $g_2^{\min}(s_3)$ is updated from ∞ to 1.
- In iteration 4, node x_2 is removed from the *Open* list, and its child node x_9 is added to the *Open* list. $g_2^{\min}(s_2)$ is updated from ∞ to 5.
- In iteration 5, node x_6 is removed from the *Open* list, and its child node x_{10} is added to the *Open* list. $g_2^{\min}(s_2)$ is updated from 5 to 3.
- In iteration 6, node x_7 is removed from the *Open* list, and its child node x_{11} is added to the *Open* list. $g_2^{\min}(s_2)$ is updated from 3 to 2.
- In iteration 7, node x_9 is removed from *Open* list, and its child node x_{12} is added to the *Open* list. $g_2^{\min}(s_5)$ is updated from ∞ to 9.
- In iteration 8, node x_{10} is removed from the *Open* list, and its child node x_{13} is added to the *Open* list. $g_2^{\min}(s_5)$ is updated from 9 to 7.
- In iteration 9, node x_{11} is removed from the *Open* list, and its child node x_{14} is added to the *Open* list. $g_2^{\min}(s_5)$ is updated from 7 to 6.
- In iteration 10, node x_4 is removed from the *Open* list. The node has no children. $g_2^{\min}(s_4)$ is updated from ∞ to 9.
- In iteration 11, node x_8 is removed from the *Open* list. The node is pruned (by Line 10) because $g_2(x_8) = 8 \geq g_2^{\min}(s_5) = 6$.
- In iteration 12, node x_{13} is removed from the *Open* list. The node is pruned (by Line 10) because $g_2(x_{13}) = 9 \geq g_2^{\min}(s_4) = 9$.

- In iteration 13, node x_{12} is removed from the *Open* list. The node is pruned (by Line 10) because $g_2(x_{12}) = 11 \geq g_2^{\min}(s_4) = 9$.
- In iteration 14, node x_5 is removed from the *Open* list. The node is pruned (by Line 10) because $g_2(x_5) = 6 \geq g_2^{\min}(s_5) = 6$.
- In iteration 15, node x_{14} is removed from *Open* list. The node has no children. $g_2^{\min}(s_4)$ is updated from 9 to 8.

BOD finds the following sets of solutions:

$$sols(s_1) = \{(1, 1)\},$$
$$sols(s_2) = \{(1, 5), (2, 3), (3, 2)\},$$
$$sols(s_3) = \{(1, 1)\},$$
$$sols(s_4) = \{(5, 9), (8, 8)\},$$
$$sols(s_5) = \{(3, 9), (4, 7), (5, 6)\}.$$

Fig. 2 (b) shows the search tree of BOD for this execution. The pruned nodes are in red.

7. Theoretical results for the BOA* and BOD algorithms

We first provide theoretical results for BOA*, where we assume that the **h**-values are consistent, before providing theoretical results for BOD. We say that a node x_1 dominates (resp. weakly dominates) a node x_2 iff the **g**-value of node x_1 dominates (resp. weakly dominates) the **g**-value of node x_2.

7.1. Theoretical results for the BOA* algorithm

Lemma 1. *Each generated (or about to be generated but pruned) node x has f_1- and f_2-values that are no smaller than the f_1- and f_2-values, respectively, of its parent node p.*

Proof Sketch. Since the **h**-values are consistent, $c_1(s(p), s(x)) + h_1(s(x)) \geq h_1(s(p))$. Therefore, we get:

$$\begin{aligned} f_1(x) &= g_1(x) + h_1(s(x)) \\ &= g_1(p) + c_1(s(p), s(x)) + h_1(s(x)) \\ &\geq g_1(p) + h_1(s(p)) \\ &= f_1(p) \end{aligned}$$

The same proof strategy yields $f_2(x) \geq f_2(p)$. □

Lemma 2. *The sequences of extracted nodes and of expanded nodes have monotonically non-decreasing f_1-values.*

Proof Sketch. BOA* extracts the node from the *Open* list with the lexicographically smallest **f**-value of all nodes in the *Open* list (Line 10). This node has the smallest f_1-value of all nodes in the *Open* list. Since generated nodes that are added to the *Open* list have f_1-values that are no smaller than those of their expanded parent nodes (Lemma 1), the sequence of extracted nodes has monotonically non-decreasing f_1-values. Since nodes are expanded in the same order in which they are extracted, the sequence of expanded nodes also has monotonically non-decreasing f_1-values. □

Lemma 3. *The sequence of expanded nodes with the same state has strictly monotonically decreasing f_2-values.*

Proof Sketch. Assume for a proof by contradiction that BOA* expands node x_1 with state s before node x_2 with state s, that it expands no node with state s after node x_1 and before node x_2, and that $f_2(x_1) \leq f_2(x_2)$. Then, $g_2(x_1) + h_2(s) = f_2(x_1) \leq f_2(x_2) = g_2(x_2) + h_2(s)$. Thus, $g_2(x_1) \leq g_2(x_2)$. After node x_1 is expanded and before node x_2 is expanded, $g_2^{\min}(s) = g_2(x_1)$ (Line 13). Combining both (in)equalities yields $g_2^{\min}(s) \leq g_2(x_2)$, which is the first pruning condition on Line 11. Therefore, node x_2 is not expanded, which contradicts the assumption. □

Lemma 4. *The sequence of expanded nodes with the same state has strictly monotonically increasing f_1-values.*

Proof Sketch. Since the sequence of expanded nodes has monotonically non-decreasing f_1-values (Lemma 2), the sequence of expanded nodes with the same state also has monotonically non-decreasing f_1-values. Assume for a proof by contradiction that BOA* expands node x_1 with state s before node x_2 with state s, that it expands no node with state s after node x_1 and before node x_2, and that $f_1(x_1) = f_1(x_2)$. We distinguish two cases:

- Node x_2 is in the *Open* list when BOA* expands node x_1: When BOA* expands node x_1, node x_1 has the lexicographically smallest **f**-value of all nodes in the *Open* list. Since $f_1(x_1) = f_1(x_2)$, it follows that $f_2(x_1) \leq f_2(x_2)$, which contradicts Lemma 3.
- Node x_2 is not in the *Open* list when BOA* expands node x_1: BOA* thus generates node x_2 after it expands node x_1. Thus, there is a node x_3 in the *Open* list when BOA* expands node x_1 that is expanded after node x_1 (or is equal to it) and before node x_2 and becomes an ancestor node of node x_2 in the search tree. Since the sequence of expanded nodes has monotonically non-decreasing f_1-values (Lemma 2) and $f_1(x_1) = f_1(x_2)$, $f_1(x_1) = f_1(x_3) = f_1(x_2)$. When BOA* expands node x_1, node x_1 has the lexicographically smallest **f**-value of all nodes in the *Open* list. Since $f_1(x_1) = f_1(x_3)$, it follows that $f_2(x_1) \leq f_2(x_3)$. Since each node has an f_2-value that is no smaller than the f_2-values of its ancestor nodes (Lemma 1), $f_2(x_3) \leq f_2(x_2)$. Combining both inequalities yields $f_2(x_1) \leq f_2(x_2)$, which contradicts Lemma 3. □

Lemma 5. *Expanded nodes with the same state do not weakly dominate each other.*

Proof Sketch. Assume that BOA* expands node x_1 with state s before node x_2 with state s. Since the sequence of expanded nodes with the same state has strictly monotonically decreasing f_2-values (Lemma 3), $f_2(x_1) > f_2(x_2)$. It follows that $g_2(x_1) + h(s) = f_2(x_1) > f_2(x_2) = g_2(x_2) + h(s)$ and thus $g_2(x_1) > g_2(x_2)$. Since the sequence has strictly monotonically increasing f_1-values (Lemma 4), the same reasoning yields $g_1(x_1) < g_1(x_2)$. According to the two inequalities, nodes x_1 and x_2 do not weakly dominate each other. □

Lemma 6. *If node x_1 with state s is weakly dominated by node x_2 with state s, then each node with another state s' in the subtree of the search tree rooted at node x_1 is weakly dominated by a node with the state s' in the subtree rooted at node x_2.*

Proof Sketch. Since node x_1 is weakly dominated by node x_2, $g_1(x_2) \leq g_1(x_1)$. Assume that node x_3 is a node with state s' in the subtree of the search tree rooted at node x_1. Let the sequence of states of the nodes along a branch of the search tree from the root node to node x_1 be $s_1, \ldots, s_i$ (with $s_1 = s_{start}$ and $s_i = s$), the sequence of states of the nodes along a branch of the search tree from the root node to node x_2 be $s'_1, \ldots, s'_j$ (with $s'_1 = s_{start}$ and $s'_j = s$), and the sequence of states of the nodes along a branch of the search tree from node x_1 to node x_3 be $\pi = s_i, \ldots, s_k$ (with $s_k = s'$). Then, there is a node x_4 with state s' in the subtree rooted at node x_2 such that the sequence of states of the nodes along a branch of the search tree from the root node to node x_4 is $s'_1, \ldots, s'_j, s_{i+1}, \ldots, s_k$. Since $g_1(x_2) \leq g_1(x_1)$, it follows that $g_1(x_4) = g_1(x_2) + c_1(\pi) \leq g_1(x_1) + c_1(\pi) = g_1(x_3)$ and thus $g_1(x_4) \leq g_1(x_3)$. The same proof strategy yields $g_2(x_4) \leq g_2(x_3)$. Combining both inequalities yields that node x_3 is weakly dominated by node x_4. □

Lemma 7. *When BOA* prunes a node x_1 with state s (on Line 11 or 22) and this prevents it in the future from adding a node x_2 (with the goal state) to the solution set of state s_{goal} (on Line 14), then it can still add in the future a node with the goal state that weakly dominates node x_2 (on Line 14).*

Proof Sketch. We prove the statement by induction on the number of pruned nodes so far, including node x_1. If the number of pruned nodes is zero, then the lemma trivially holds. Now assume that the number of pruned nodes is $n + 1$ and the lemma holds for $n \geq 0$. We distinguish three cases:

- **Case 1:** BOA* prunes node x_1 on Line 11 because of the (first) pruning condition $g_2(x_1) \geq g_2^{\min}(s)$. Then, BOA* has expanded a node x_4 with state s previously such that $g_2^{\min}(s) = g_2(x_4)$ since otherwise $g_2^{\min}(s) = \infty$ and the pruning condition could not hold. Combining both (in)equalities yields $g_2(x_1) \geq g_2(x_4)$. Since $f_1(x_1) \geq f_1(x_4)$ (Lemma 2), $g_1(x_1) + h(s) = f_1(x_1) \geq f_1(x_4) = g_1(x_4) + h(s)$ and thus $g_1(x_1) \geq g_1(x_4)$. Combining both inequalities yields that node x_1 is weakly dominated by node x_4 and thus each node with state s' in the subtree rooted at node x_1, including node x_2, is weakly dominated by a node x_5 with state s' in the subtree rooted at node x_4 (Lemma 6). In case BOA* has pruned a node that prevents it in the future from adding node x_5 to the solution set of state s', then it can still add in the future a node (with state s') that weakly dominates node x_5 and thus also node x_2 (induction assumption).
- **Case 2:** BOA* prunes node x_1 on Line 11 because of the (second) pruning condition $f_2(x_1) \geq g_2^{\min}(s_{goal})$. Then, BOA* has expanded a node x_4 with the goal state previously such that $g_2^{\min}(s_{goal}) = g_2(x_4)$ since otherwise $g_2^{\min}(s_{goal}) = \infty$ and the pruning condition could not hold. Combining both (in)equalities yields that $f_2(x_1) \geq g_2(x_4)$. Since node x_1 is an ancestor node of node x_2 in the search tree, $f_2(x_2) \geq f_2(x_1)$ (Lemma 1). Combining both inequalities yields $g_2(x_2) = f_2(x_2) \geq g_2(x_4)$. Since node x_1 is an ancestor node of node x_2 in the search tree, $g_1(x_2) = f_1(x_2) \geq f_1(x_1)$ (Lemma 1). Since $f_1(x_1) \geq f_1(x_4)$ (Lemma 2), it follows that $g_1(x_2) \geq f_1(x_1) \geq f_1(x_4) = g_1(x_4)$. Combining $g_1(x_2) \geq g_1(x_4)$ and $g_2(x_2) \geq g_2(x_4)$ yields that node x_2 is weakly dominated by node x_4 (with the goal state). In case BOA* has pruned a node that prevents it in the future from adding node x_4 to the solution set of the goal state, then it can still add in the future a node (with the goal state) that weakly dominates node x_4 and thus also node x_2 (induction assumption).
- **Case 3:** BOA* prunes node x_1 on Line 22 because of the pruning condition $g_2(x_1) \geq g_2^{\min}(s)$ or $f_2(x_1) \geq g_2^{\min}(s_{goal})$. The proofs of Case (1) or Case (2), respectively, apply unchanged except that $f_1(x_1) \geq f_1(x_4)$ now holds for a different

reason. Let node x_3 be the node that BOA* expands when it executes Line 22. Combining $f_1(x_1) \geq f_1(x_3)$ (Lemma 1) and $f_1(x_3) \geq f_1(x_4)$ (Lemma 2) yields $f_1(x_1) \geq f_1(x_4)$. □

Theorem 1. *BOA* computes a cost-unique Pareto-optimal solution set.*

Proof Sketch. Let the path of a node x (and the solution of a node x with the goal state) be the sequence of states of the nodes along a branch of the search tree from the root node to node x. Then, the **g**-value of node x is the cost of the path (or the solution). Since the costs are non-negative and expanded nodes with the same state do not weakly dominate each other (Lemma 5), the paths of the expanded nodes are cycle-free. Since there are only a finite number of cycle-free paths, there are only a finite number of expanded nodes and thus only a finite number of generated nodes that are put into the *Open* list. Since one node is extracted from the *Open* list during each iteration, there are only a finite number of iterations and BOA* terminates.

Now consider any non-empty set $X(s_{goal})$ of all nodes whose solutions are Pareto-optimal solutions for the goal state. If BOA* is prevented in the future from adding a node $x_1 \in X(s_{goal})$ to the solution set $sols(s_{goal})$, then there exists a node $x_2 \in X^{x_1}_{\text{dom}}$ that will be added to $sols(s_{goal})$, where $X^{x_1}_{\text{dom}}$ is the set of all nodes with the goal state that weakly dominates node x_1 and it is guaranteed to be non-empty (Lemma 7). If this does not hold, then all nodes in $X^{x_1}_{\text{dom}}$ are pruned and no node with the goal state is added to $sols(s_{goal})$, which contradicts Lemma 7. The computed solution set $sols(s_{goal})$ is thus a superset of a cost-unique Pareto-optimal solution set P. Since BOA* can add only expanded nodes to the solution set $sols(s_{goal})$ and expanded nodes with the goal state do not weakly dominate each other (Lemma 5), the computed solution set $sols(s_{goal})$ cannot contain solutions that are not Pareto-optimal or have the same cost as other solutions in the computed solution set. Thus, it is exactly the cost-unique Pareto-optimal solution set P. □

7.2. Theoretical results for the BOD algorithm

We now provide theoretical results for BOD, which are based on the theoretical results for BOA* above. Recall that the key algorithmic differences between BOA* and BOD are the following:

1. BOA* uses arbitrary consistent **h**-values, while BOD uses zero **h**-values.
2. BOD does not have a goal state defined.
3. BOA* uses $g_2^{\min}(s_{goal})$ to prune nodes (Lines 11 and 22), while BOD does not.
4. BOA* does not generate the children of those nodes x with the goal state (Lines 15-16), while BOD does generate the children of those nodes.
5. BOA* returns the solution set $sols(s_{goal})$ for the goal state (Line 25), while BOD returns the set *sols* of solution sets for all states (Line 21).

For Lemmata 1-6, the differences above do not affect their statements or proofs. Therefore, they apply unchanged to BOD. For Lemma 7, the second difference affects its proof since it assumes that a goal state is defined in Cases 2 and 3 of the proof. Specifically, BOA* prunes node x_1 when the pruning condition $f_2(x_1) \geq g_2^{\min}(s_{goal})$ is satisfied. However, since BOD does not have a goal state defined, it does not use this pruning condition. Therefore, it's pruning condition is looser than that of BOA*, and the following Corollary 1 for BOD trivially holds.

Corollary 1. *When BOD prunes a node x_1 with state s (on Line 10 or 18) and this prevents it in the future from adding a node x_2 (with a different state s') to the solution set of state s' (on Line 13), then it can still add in the future a node (with state s') that weakly dominates node x_2 (on Line 13).*

Finally, for Theorem 1, the difference of the solution set(s) returned by BOA* and BOD affects its statement and proof. Nonetheless, we can generalize Theorem 1 to the following Theorem 2 for BOD, whose proof follows closely the one of Theorem 1.

Theorem 2. *BOD computes a cost-unique Pareto-optimal solution set for each state.*

Proof Sketch. Let the path of a node x be the sequence of states of the nodes along a branch of the search tree from the root node to node x. Then, the **g**-value of node x is the cost of the path. Since the costs are non-negative and expanded nodes with the same state do not weakly dominate each other (Lemma 5), the paths of the expanded nodes are cycle-free. Since there are only a finite number of cycle-free paths, there are only a finite number of expanded nodes and thus only a finite number of generated nodes that are put into the *Open* list. Since one node is extracted from the *Open* list during each iteration, there are only a finite number of iterations and BOD terminates.

Now consider any non-empty set $X(s)$ of all nodes whose solutions are Pareto-optimal solutions for state s. If BOD is prevented in the future from adding a node $x_1 \in X(s)$ to the solution set $sols(s)$, then there exists a node $x_2 \in X^{x_1}_{\text{dom}}$ that will be added to $sols(s)$, where $X^{x_1}_{\text{dom}}$ is the set of all nodes with state s that weakly dominates node x_1 and it is guaranteed to

be non-empty (Corollary 1). If this does not hold, then all nodes in $X^{x_1}_{\text{dom}}$ are pruned and no node with state s is added to $sols(s)$, which contradicts Corollary 1. The computed solution set $sols(s)$ is thus a superset of a cost-unique Pareto-optimal solution set $P(s)$ for state s. Since BOD can add only expanded nodes to the solution set $sols(s)$ and expanded nodes with state s do not weakly dominate each other (Lemma 5), the computed solution set $sols(s)$ cannot contain solutions that are not Pareto-optimal or have the same cost as other solutions in the computed solution set. Thus, it is exactly the cost-unique Pareto-optimal solution set $P(s)$. □

8. Empirical evaluation

In this section, we evaluate our algorithms empirically. We first describe our empirical evaluations of BOA* and BOD in Sections 8.1 and 8.2, respectively. We then demonstrate in Section 8.3 how BOA* can be applied to the real-world setting of balancing path length and safety when transporting hazardous material in a city.

8.1. The BOA* algorithm

Setup We compare BOA*, BOA* with standard linear-time dominance checking (sBOA*), NAMOA*dr [7], Bi-Objective Dijkstra (BDijkstra) [9], and Bidirectional Bi-Objective Dijkstra (BBDijkstra) [9] for finding a Pareto-optimal solution set (for the goal state). We use the C implementations of BBDijkstra and BDijkstra provided by their authors (with two pruning strategies for one to one search, namely pruning by nadir points and pruning by efficient set [9]). We implement BOA*, sBOA*, and NAMOA*dr from scratch in C using a standard binary heap for the *Open* list.[5] We run all experiments on a 3.80 GHz Intel(R) I7(R) CPU Linux computer with 64 GB of RAM. We use road maps from the 9th DIMACS Implementation Challenge: Shortest Path.[6] The cost components represent travel distances (c_1) and times (c_2). The **h**-values are the exact travel distances and times to the goal state, computed with Dijkstra's algorithm. It takes 75 milliseconds to compute the **h**-values for the largest road map. The reported runtimes include this computation. All algorithms obtain the same number of solutions for all instances used in the experiments, implying that no two Pareto-optimal solutions have the same cost.

Results We compare the runtimes of the five algorithms on the 50 instances each of four road maps from the USA used by Machuca and Mandow [38]. Table 4 shows the name of the road map, the number of states and edges of the road map, and the average number of Pareto-optimal solutions. For each algorithm, it shows the number of instances solved for a runtime limit of 3,600 seconds as well as the average, maximum, minimum, median, and standard deviation of the runtimes (in seconds). NAMOA*dr can be an order-of-magnitude faster than sBOA*. BOA* can be faster than NAMOA*dr, especially on instances with a large number of Pareto-optimal solutions. For example, BOA* is 2.2 times faster than NAMOA*dr on FL (with 739 Pareto-optimal solutions on average), while BOA* is only 1.3 times faster than NAMOA*dr on BAY (with 119 Pareto-optimal solutions on average). BOA* can also be an order-of-magnitude faster than BBDijkstra and BDijkstra. In addition, we compare the runtimes of the five algorithms on 50 random instances each of four bigger road maps, see Table 5. BOA* solves all instances on all road maps and is faster than the other algorithms, especially on instances with a large number of Pareto-optimal solutions.

We now compare the runtimes of the algorithms as a function of the difficulty of the instances on the largest road maps of Tables 4 and 5. Figs. 3 and 4 show the cumulative runtimes (in seconds) of the algorithms on instances of FL and LKS, respectively, for a runtime limit of 3,600 seconds. When an algorithm reaches the runtime limit, we use 3,600 seconds in the calculation of the runtime metric. The instances are ordered in increasing numbers of their Pareto-optimal solutions ($|sols|$). When $|sols|$ is small, the cumulative runtimes of the algorithms are similar. As $|sols|$ increases, the cumulative runtimes of the algorithms increase proportionally. The cumulative runtime of BOA* becomes orders of magnitude smaller than the ones of sBOA*, BDijkstra, and BBDijkstra and several times smaller than the cumulative runtime of NAMOA*dr.

We now compare the number of op-pruning operations of BOA* and NAMOA*dr for the dominance checks on $\mathbf{G}_{op}$. The term "op-pruning operations" was coined by PMP and describes the number of nodes checked on $\mathbf{G}_{op}$ when a node is generated. Table 6 shows the number of Pareto-optimal solutions, the ratio of generated nodes of NAMOA*dr and BOA*, the ratio of runtimes of NAMOA*dr and BOA*, and the number of op-pruning operations per generated node for NAMOA*dr on four instances of LKS. BOA* generates around 1.04 times more nodes than NAMOA*dr because BOA* can first generate nodes (and insert them into the *Open* list) and later prune them on Lines 11-12 of Algorithm 3. For Instance 1, NAMOA*dr and BOA* are about equally fast. However, for the other instances, BOA* is faster because NAMOA*dr performs more op-pruning operations as $|sols|$ increases. This advantage of BOA* demonstrates the power of BOA*, whose dominance checks run in constant time, over NAMOA*dr, whose dominance checks on $\mathbf{G}_{op}$ do not run in constant time (linear time in our implementation).

We now consider the runtime of BOA* as a function of the lexicographic ordering used for the *Open* list, namely either (f_1, f_2) or (f_2, f_1). Table 7 shows the runtime (in seconds) of BOA* with both the (f_1, f_2) and (f_2, f_1) orderings of the *Open* list on 50 instances of LKS. The ordering of the cost components has a strong influence on the runtime of BOA*. In

[5] The implementation of BOA*, NAMOA*dr and BOD is available at https://github.com/jorgebaier/BOAstar/.
[6] http://users.diag.uniroma1.it/challenge9/download.shtml.

Table 4
Runtime (in seconds) on 50 instances of the specified road map.

New York City (NY)						
264,346 states, 730,100 edges, $\|sols\|$ = 199 on average						
	solved	average	max	min	median	stdev
BOA*	50/50	0.20	1.29	0.07	0.09	0.29
NAMOA*dr	50/50	0.24	1.68	0.07	0.10	0.39
sBOA*	50/50	3.94	59.24	0.07	0.14	12.07
BDijkstra	50/50	1.68	13.98	0.10	0.48	3.07
BBDijkstra	50/50	1.15	14.72	0.15	0.26	2.55
San Francisco Bay (BAY)						
321,270 states, 794,830 edges, $\|sols\|$ = 119 on average						
	solved	average	max	min	median	stdev
BOA*	50/50	0.15	1.66	0.08	0.09	0.23
NAMOA*dr	50/50	0.20	3.28	0.08	0.09	0.46
sBOA*	50/50	1.57	53.74	0.08	0.10	7.60
BDijkstra	50/50	1.20	23.63	0.12	0.21	3.66
BBDijkstra	50/50	0.50	5.86	0.16	0.23	0.89
Colorado (COL)						
435,666 states, 1,042,400 edges, $\|sols\|$ = 427 on average						
	solved	average	max	min	median	stdev
BOA*	50/50	0.54	11.45	0.10	0.14	1.61
NAMOA*dr	50/50	1.05	31.34	0.11	0.15	4.37
sBOA*	50/50	17.04	480.18	0.11	0.27	68.64
BDijkstra	50/50	5.36	97.99	0.17	0.38	16.32
BBDijkstra	50/50	2.32	40.16	0.21	0.35	6.15
Florida (FL)						
1,070,376 states, 2,712,798 edges, $\|sols\|$ = 739 on average						
	solved	average	max	min	median	stdev
BOA*	50/50	3.33	45.71	0.27	0.36	9.28
NAMOA*dr	50/50	7.42	140.00	0.27	0.39	24.54
sBOA*	50/50	180.62	3,235.10	0.27	0.81	591.82
BDijkstra	50/50	110.89	1,851.22	0.42	1.66	320.10
BBDijkstra	50/50	46.54	886.05	0.61	1.02	148.98

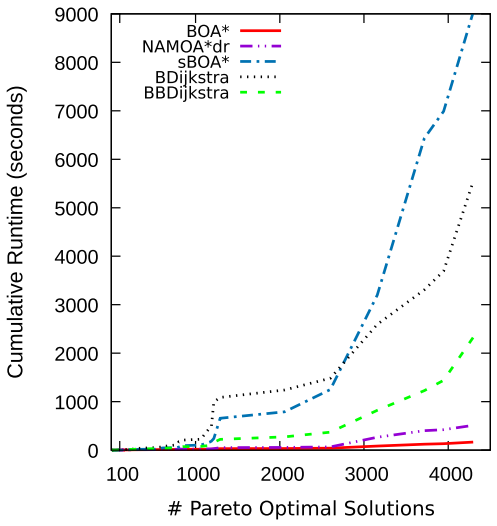


Fig. 3. Cumulative runtime (in seconds) on 50 instances of FL. The instances are ordered on the x-axis in increasing numbers of their Pareto-optimal solutions.

Table 5
Runtime (in seconds) on 50 instances of the specified road map. When an algorithm reaches the runtime limit of 3,600 seconds, we use 3,600 seconds in the calculation of the runtime metrics.

Northwest USA (NW)						
1,207,495 states, 2,840,208 edges, $\|sols\|$ = 1,051 on average						
	solved	average	max	min	median	stdev
BOA*	50/50	3.95	45.32	0.31	0.63	8.94
NAMOA*dr	50/50	9.59	121.19	0.31	0.72	24.91
sBOA*	47/50	348.76	3,600.00	0.32	4.15	946.65
BDijkstra	50/50	70.11	568.85	0.50	5.35	133.09
BBDijkstra	50/50	46.08	449.86	0.70	4.97	105.35
Northeast USA (NE)						
1,524,453 states, 3,897,636 edges, $\|sols\|$ = 1,071 on average						
	solved	average	max	min	median	stdev
BOA*	50/50	8.26	48.21	0.40	2.57	11.24
NAMOA*dr	50/50	18.33	112.46	0.41	3.34	26.87
sBOA*	49/50	516.58	3,600.00	0.41	90.01	866.15
BDijkstra	50/50	194.73	1,031.57	0.66	22.93	288.93
BBDijkstra	50/50	142.35	1,222.92	0.96	15.66	246.19
California and Nevada (CAL)						
1,890,815 states, 4,657,742 edges, $\|sols\|$ = 907 on average						
	solved	average	max	min	median	stdev
BOA*	50/50	8.16	98.35	0.52	0.86	18.20
NAMOA*dr	50/50	19.44	313.36	0.52	1.02	52.97
sBOA*	47/50	432.72	3,600.00	0.53	10.41	946.10
BDijkstra	48/50	269.24	3,600.00	0.85	8.56	762.84
BBDijkstra	50/50	85.43	1,034.39	1.22	4.81	194.86
Great Lakes (LKS)						
2,758,119 states, 6,885,658 edges, $\|sols\|$ = 6,057 on average						
	solved	average	max	min	median	stdev
BOA*	50/50	237.86	1,250.43	1.85	86.33	315.88
NAMOA*dr	44/50	815.42	3,600.00	2.73	258.29	1,119.26
sBOA*	14/50	2,852.85	3,600.00	101.49	3,600.00	1.242.49
BDijkstra	33/50	1,793.71	3,600.00	9.66	1,584.31	1,408.82
BBDijkstra	27/50	2,095.12	3,600.00	29.17	1,920.43	1,460.94

Table 6
Four instances of LKS. (1) Ratio of generated nodes (NAMOA*dr/BOA*). (2) Ratio of runtimes (NAMOA*dr/BOA*). (3) Number of *op-pruning* operations per generated node for NAMOA*dr.

Great Lakes (LKS)						
#	*start*	*goal*	$\|sols\|$	(1)	(2)	(3)
1	1,941,792	785,069	27	0.97	1.03	1.3
2	207,871	3,619	419	0.95	1.08	8.4
3	1,137,220	991,262	1,947	0.95	2.96	16.5
4	1,836,318	1,792,612	4,072	0.95	2.91	15.7

particular, BOA* is faster when its $Open$ list is ordered lexicographically according to (f_2, f_1) instead of (f_1, f_2) because it generates 10% fewer nodes (and, consequently, also performs fewer heap percolations).

8.2. The BOD algorithm

Setup We compare BOD and BDijkstra for finding a Pareto-optimal solution set for every state.

Results We compare the runtimes of both algorithms on the 50 instances each of the four road maps NY, BAY, COL, and FL used by Machuca and Mandow [38], ignoring their goal states. Table 8 shows the name of the road map, the number of states and edges of the road map, and the average number of Pareto-optimal solutions. For each algorithm, it shows the number of instances solved for a runtime limit of 3,600 seconds as well as the average, maximum, minimum, median, and standard deviation of the runtimes (in seconds). BOD can be several times faster than BDijkstra. For example, it is 6.3 times faster than BDijkstra on average on COL and 4.5 times faster than BDijkstra on average on FL.

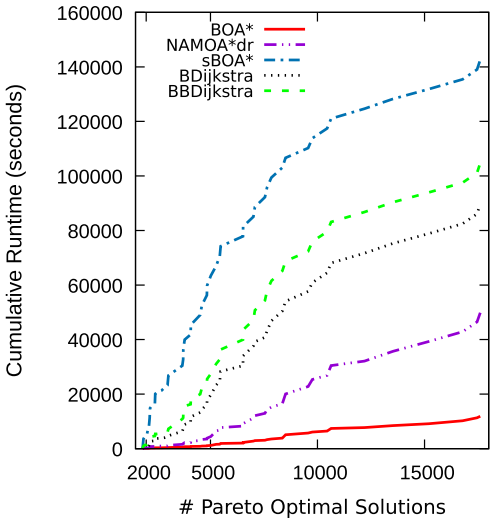


Fig. 4. Cumulative runtime (in seconds) on 50 instances of LKS. The instances are ordered on the x-axis in increasing numbers of their Pareto-optimal solutions.

Table 7
Runtime (in seconds) on 50 instances of LKS.

Great Lakes (LKS)						
	solved	average	max	min	median	stdev
BOA* (f_1, f_2)	50/50	237.86	1,250.43	1.85	86.33	315.88
BOA* (f_2, f_1)	50/50	**174.41**	**983.49**	**1.62**	**59.16**	**237.21**

Table 8
Runtime (in seconds) on 50 instances of the specified road map. When an algorithm reaches the runtime limit of 3,600 seconds, we use 3,600 seconds in the calculation of the runtime metrics.

New York City (NY)						
264,346 states, 730,100 edges, $\|sols\|$ = 166 per state on average						
	solved	average	max	min	median	stdev
BOD	50/50	15.09	35.92	6.61	13.29	7.21
BDijkstra	50/50	57.33	141.86	17.04	48.10	33.57
San Francisco Bay (BAY)						
321,270 states, 794,830 edges, $\|sols\|$ = 137 per state on average						
	solved	average	max	min	median	stdev
BOD	50/50	12.15	44.26	2.56	10.59	7.25
BDijkstra	50/50	50.14	190.51	5.34	41.81	36.89
Colorado (COL)						
435,666 states, 1,042,400 edges, $\|sols\|$ = 374 per state on average						
	solved	average	max	min	median	stdev
BOD	50/50	63.44	252.04	16.61	43.84	57.90
BDijkstra	50/50	397.42	1,802.13	73.42	292.12	432.24
Florida (FL)						
1,070,376 states, 2,712,798 edges, $\|sols\|$ = 528 per state on average						
	solved	average	max	min	median	stdev
BOD	48/50	414.43	3,600.00	56.20	259.57	668.77
BDijkstra	42/50	1,866.57	3,600.00	323.89	1,580.30	1,058.96

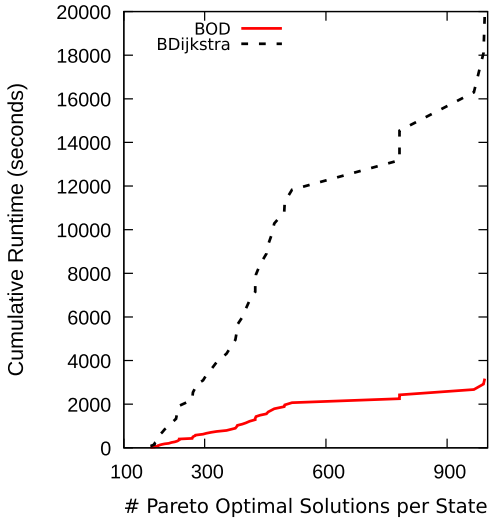


Fig. 5. Cumulative runtime (in seconds) on 50 instances of COL. The instances are ordered on the x-axis in increasing numbers of their Pareto-optimal solutions.

Table 9
Runtime (in milliseconds) on 100 instances of SCL.

Santiago Chile (SCL)						
2,212 states, 6,681 edges, $\|sols\|$ = 29.1 on average						
	solved	average	max	min	median	stdev
BOA*	100/100	0.61	2.51	0.28	0.45	0.42
NAMOA*dr	100/100	0.62	2.56	0.28	0.46	0.43
sBOA*	100/100	3.81	21.40	0.29	2.38	3.86
BDijkstra	100/100	2.94	18.18	0.65	1.66	2.95
BBDijkstra	100/100	2.45	11.05	0.66	1.63	1.91

Fig. 5 shows the cumulative runtimes (in seconds) of both algorithms on instances of COL for a runtime limit of 3,600 seconds. The instances are ordered in increasing numbers of their Pareto-optimal solutions ($|sols|$). When $|sols|$ is small, the cumulative runtimes of the algorithms are similar. As $|sols|$ increases, the cumulative runtimes of the algorithms increase proportionally. The cumulative runtime of BOD becomes orders of magnitude smaller than the one of BDijkstra since it is faster than BDijkstra on all instances.

8.3. The Hazardous material transport (HAZMAT) problem in Santiago, Chile

The purpose of this section is to demonstrate the potential of BOA* and other bi-objective search algorithms on a real-world application domain – the hazardous material (HAZMAT) transportation problem in Santiago, Chile (SCL) [2]. In this problem, we are given the road network of SCL, which spans an area of 641 square kilometers. There are 244 vulnerable locations that the transport of the HAZMAT shipments should avoid, defined to be schools with more than one thousand students. These schools have between 1,070 and nearly 4,500 students each, with a total of 386,254 students. The cost components represent travel distances (c_1) and the student population exposed to hazardous material in case of an accident (c_2) [2]. Table 9 shows the number of states and edges of the road map and, for each exposure radius, the average number of Pareto-optimal solutions as well as the average, maximum, minimum, median, and standard deviation of the runtimes (in milliseconds) for all algorithms evaluated. BOA* and NAMOA*dr compute the Pareto-optimal frontier in less than one millisecond for most instances while classical operations research models are only able to obtain a subset approximation of the Pareto-optimal frontier within several seconds [2] in this small road map, which demonstrates the potential of BOA* and bi-objective search algorithms for solving bi-objective routing problems.

8.4. An alternative to improve the linear-time dominance checking

The M3 algorithm by Bentley et al. [39] can potentially reduce the number of op-pruning operations in NAMOA*dr and, as a consequence, reduce the runtime of the algorithm. M3 was designed to obtain logarithmic-time when a new vector is checked over a set of non-dominating vectors. The main idea of M3 is to compare a new vector with the "powerful" dominators of the set first. We implemented M3 in NAMOA*dr to perform the dominance checks in $\mathbf{G}_{op}$. The results show

that the reduction of the number of op-pruning operations is only 1% in average. The reasons for the small reduction are:

- The size of $\mathbf{G}_{op}$ is small. For example, the size of $\mathbf{G}_{op}$ in the instances of Table 6 is 6.5 on average.
- $\mathbf{G}_{op}$ is a dynamic set where the nodes of the set continually change. Nodes are inserted and removed in the node generation process, and nodes are removed in the node expansion process.
- The node with smaller lexicographic order is removed of $\mathbf{G}_{op}$ first. This node is generally a powerful dominator.

We did not include results of NAMOA*dr with M3 in our evaluation since only a very small runtime improvement is observed. We conjecture that M3 could have a larger impact in the performance of NAMOA*dr in problems with more than two objectives because $\mathbf{G}_{op}$ may grow larger and M3 could also be used in $\mathbf{G}_{cl}$.

9. Conclusions and future work

We introduced a simple, yet effective approach for performing efficient, constant-time dominance checks in bi-objective search algorithms. Using this approach, we presented Bi-Objective A* (BOA*) and Bi-Objective Dijkstra (BOD), that, given some start state, efficiently compute the Pareto-optimal frontier to a single goal state and to all states, respectively. Our experimental evaluation demonstrated that our algorithms are faster than state-of-the-art algorithms such as NAMOA*dr, Bi-Objective Dijkstra, and Bidirectional Bi-Objective Dijkstra. We intend to improve and extend our algorithmic framework in future work, as we discuss in the following in the context of BOA*:

Efficient computation of the Pareto-optimal frontier via node re-ordering The cost of a solution is a pair (c_1, c_2). The c_1-values of solutions found by BOA* are strictly monotonically increasing in time, and the c_2-values are strictly monotonically decreasing in time. Thus, the first solution found by BOA* has the smallest c_1-value, and the last solution has the smallest c_2-value. If BOA* orders the *Open* list lexicographically according to (f_2, f_1) instead of (f_1, f_2), the opposite happens. BOA* might therefore run faster if it runs two BOA* instantiations in parallel, one for each ordering, and terminates when both instantiations find a solution of the same cost.

Efficient computation of the Pareto-optimal frontier via bi-directional search Bi-directional search simultaneously searches from the start state toward the goal state and from the goal state toward the start state instead of in one of these directions only, which typically speeds up the search without requiring better **h**-values. The currently best bi-directional bi-objective search algorithm is Bi-Directional Bi-Objective Dijkstra (BBDijkstra), but we showed that BOA* can be orders of magnitude faster than BBDijkstra, which is not surprising since BBDijkstra performs linear-time dominance checks, while BOA* performs constant-time dominance checks. There have been recent advances of the theory behind bi-directional single-objective search that have resulted in faster bi-directional single-objective search algorithms. It might therefore be possible to develop faster bi-directional bi-objective search algorithms based on BOA* and recent ideas for bi-directional single-objective search [40,41].

Extensions to multi-objective search BOA* might be able to find all cost-unique Pareto-optimal solutions for cost functions with more than two components if it runs several times for different permutations of the components. For example, BOA* might find a subset of the Pareto-optimal solutions if it orders the *Open* list lexicographically according to some ordering of the components. Other orderings might result in different subsets. It might therefore be possible to extend BOA* to multi-objective search if once can prove that the union of all such subsets contains exactly all cost-unique Pareto-optimal solutions.

Bounded approximations of the Pareto-optimal frontier Several of our instances have thousands of Pareto-optimal solutions. For example, one of the LKS instances has 17,606 solutions. Many of the Pareto-optimal solutions contain almost the same edges. It might therefore be possible to extend BOA* to compute an approximation of the Pareto-optimal solutions. In fact, we recently presented our preliminary results in this direction [42,43].

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

The research at Universidad San Sebastián and Pontificia Universidad Católica de Chile was supported by National Center for Artificial Intelligence CENIA FB210017, Basal ANID. The research at the University of Southern California was supported by the National Science Foundation (NSF) under grant numbers 1409987, 1724392, 1817189, 1837779, and 1935712. The research at Washington University in St. Louis was supported by NSF under grant numbers 1812619 and 1838364. The research at the Technion—Israel Institute of Technology was supported by the Israeli Ministry of Science & Technology under grant numbers 102583 and 2028142 and by the United States-Israel Binational Science Foundation (BSF) under grant

number 1018193. Finally, we thank Antonio Sedeño for sharing the source code of the BDijkstra and BBDijkstra algorithms with us.

References

[1] P.E. Hart, N. Nilsson, B. Raphael, A formal basis for the heuristic determination of minimal cost paths, IEEE Trans. Syst. Sci. Cybern. 4 (1968) 100–107.
[2] A. Bronfman, V. Marianov, G. Paredes-Belmar, A. Lüer-Villagra, The maximin HAZMAT routing problem, Eur. J. Oper. Res. 241 (2015) 15–27.
[3] M. Fu, A. Kuntz, O. Salzman, R. Alterovitz, Toward asymptotically-optimal inspection planning via efficient near-optimal graph search, in: Robotics: Science and Systems (RSS), 2019.
[4] D. Bachmann, F. Bökler, J. Kopec, K. Popp, B. Schwarze, F. Weichert, Multi-objective optimisation based planning of power-line grid expansions, ISPRS Int. J. Geo-Inf. 7 (2018) 258.
[5] B.S. Stewart, C.C. White III, Multiobjective A*, J. ACM 38 (1991) 775–814.
[6] L. Mandow, J.L.P. De La Cruz, Multiobjective A* search with consistent heuristics, J. ACM 57 (2010) 27:1–27:25.
[7] F.-J. Pulido, L. Mandow, J.-L. Pérez-de-la-Cruz, Dimensionality reduction in multiobjective shortest path search, Comput. Oper. Res. 64 (2015) 60–70.
[8] A. Felner, Position paper: Dijkstra's algorithm versus uniform cost search or a case against Dijkstra's algorithm, in: Proceedings of the Annual Symposium on Combinatorial Search, 2011.
[9] A. Sedeño-Noda, M. Colebrook, A biobjective Dijkstra algorithm, Eur. J. Oper. Res. 276 (2019) 106–118.
[10] C. Hernandez, W. Yeoh, J. Baier, H. Zhang, L. Suazo, S. Koenig, A simple and fast bi-objective search algorithm, in: Proceedings of the International Conference on Automated Planning and Scheduling, 2020, pp. 143–151.
[11] E. van der Horst, P. Marqués-Gallego, T. Mulder-Krieger, J. van Veldhoven, J. Kruisselbrink, A. Aleman, M.T. Emmerich, J. Brussee, A. Bender, A.P. IJzerman, Multi-objective evolutionary design of adenosine receptor ligands, J. Chem. Inf. Model. 52 (2012) 1713–1721.
[12] S. Rosenthal, M. Borschbach, Design perspectives of an evolutionary process for multi-objective molecular optimization, in: Proceedings of the International Conference on Evolutionary Multi-Criterion Optimization, 2017, pp. 529–544.
[13] C. Hopfe, M. Emmerich, R. Marijt, J. Hensen, Robust multi-criteria design optimization in building design, in: Proceedings of the IBPSA-England Conference on Building Simulation and Optimization, 2012, pp. 118–125.
[14] K. Miettinen, Nonlinear Multiobjective Optimization, vol. 12, Springer, 2012.
[15] M.T. Emmerich, A.H. Deutz, A tutorial on multiobjective optimization: fundamentals and evolutionary methods, Nat. Comput. 17 (2018) 585–609.
[16] K. Deb, Multi-Objective Optimization Using Evolutionary Algorithms, vol. 16, John Wiley & Sons, 2001.
[17] E. Zitzler, L. Thiele, M. Laumanns, C.M. Fonseca, V.G. Da Fonseca, Performance assessment of multiobjective optimizers: an analysis and review, IEEE Trans. Evol. Comput. 7 (2003) 117–132.
[18] J. Branke, K. Deb, K. Miettinen, R. Slowiński, Multiobjective Optimization: Interactive and Evolutionary Approaches, vol. 5252, Springer, 2008.
[19] C.A.C. Coello, G.B. Lamont, D.A. van Veldhuizen, Evolutionary Algorithms for Solving Multi-Objective Problems, vol. 5, Springer, 2007.
[20] P. Serafini, Some considerations about computational complexity for multi objective combinatorial problems, in: Recent Advances and Historical Development of Vector Optimization, Springer, 1987, pp. 222–232.
[21] C.H. Papadimitriou, M. Yannakakis, On the approximability of trade-offs and optimal access of web sources, in: Proceedings of the IEEE Symposium on Foundations of Computer Science, 2000, pp. 86–92.
[22] M. Ehrgott, Multicriteria Optimization, 2nd ed., Springer, 2005.
[23] T. Breugem, T. Dollevoet, W. van den Heuvel, Analysis of FPTASes for the multi-objective shortest path problem, Comput. Oper. Res. 78 (2017) 44–58.
[24] P. Hansen, Bicriterion path problems, in: Multiple Criteria Decision Making Theory and Application, Springer, 1980, pp. 109–127.
[25] E.Q.V. Martins, On a multicriteria shortest path problem, Eur. J. Oper. Res. 16 (1984) 236–245.
[26] S. Demeyer, J. Goedgebeur, P. Audenaert, M. Pickavet, P. Demeester, Speeding up Martins' algorithm for multiple objective shortest path problems, 4OR 11 (2013) 323–348.
[27] L. Galand, A. Ismaili, P. Perny, O. Spanjaard, Bidirectional preference-based search for multiobjective state space graph problems, in: Proceedings of the Annual Symposium on Combinatorial Search, 2013, pp. 80–88.
[28] D. Duque, L. Lozano, A.L. Medaglia, An exact method for the biobjective shortest path problem for large-scale road networks, Eur. J. Oper. Res. 242 (2015) 788–797.
[29] L. Mandow, J.L.P. De La Cruz, A new approach to multiobjective A* search, in: Proceedings of the International Joint Conference on Artificial Intelligence, 2005, pp. 218–223.
[30] V.V. Vazirani, Approximation Algorithms, Springer, 2001.
[31] A. Warburton, Approximation of Pareto optima in multiple-objective, shortest-path problems, Oper. Res. 35 (1987) 70–79.
[32] P. Perny, O. Spanjaard, Near admissible algorithms for multiobjective search, in: Proceedings of the European Conference on Artificial Intelligence, vol. 178, 2008, pp. 490–494.
[33] G. Tsaggouris, C.D. Zaroliagis, Multiobjective optimization: improved FPTAS for shortest paths and non-linear objectives with applications, Theory Comput. Syst. 45 (2009) 162–186.
[34] A. Sedeno-Noda, A. Raith, A Dijkstra-like method computing all extreme supported non-dominated solutions of the biobjective shortest path problem, Comput. Oper. Res. 57 (2015) 83–94.
[35] J. Legriel, C. Le Guernic, S. Cotton, O. Maler, Approximating the Pareto front of multi-criteria optimization problems, in: Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems, 2010, pp. 69–83.
[36] S. Edelkamp, S. Schrödl, Heuristic Search: Theory and Applications, Morgan Kaufmann, 2011.
[37] A. Raith, M. Ehrgott, A comparison of solution strategies for biobjective shortest path problems, Comput. Oper. Res. 36 (2009) 1299–1331.
[38] E. Machuca, L. Mandow, Multiobjective heuristic search in road maps, Expert Syst. Appl. 39 (2012) 6435–6445.
[39] J.L. Bentley, K.L. Clarkson, D.B. Levine, Fast linear expected-time algorithms for computing maxima and convex hulls, Algorithmica 9 (1993) 168–183.
[40] R.C. Holte, A. Felner, G. Sharon, N.R. Sturtevant, J. Chen, MM: a bidirectional search algorithm that is guaranteed to meet in the middle, Artif. Intell. 252 (2017) 232–266.
[41] V. Alcázar, P.J. Riddle, M. Barley, A unifying view on individual bounds and heuristic inaccuracies in bidirectional search, in: Proceedings of the AAAI Conference on Artificial Intelligence, 2020, pp. 2327–2334.
[42] O. Salzman, B. Goldin, 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, 2021, pp. 149–158.
[43] H. Zhang, O. Salzman, T.K.S. Kumar, A. Felner, C. Hernandez, S. Koenig, A*pex: efficient approximate multi-objective search on graphs, in: Proceedings of the International Conference on Automated Planning and Scheduling, 2022.