Bi-Objective Search with Bi-Directional A*

Saman Ahmadi[1]
Department of Data Science and AI, Monash University, Clayton, Australia
CSIRO Data61, Canberra, Australia

Guido Tack
Department of Data Science and AI, Monash University, Clayton, Australia

Daniel Harabor
Department of Data Science and AI, Monash University, Clayton, Australia

Philip Kilby
CSIRO Data61, Canberra, Australia

Abstract

Bi-objective search is a well-known algorithmic problem, concerned with finding a set of optimal solutions in a two-dimensional domain. This problem has a wide variety of applications such as planning in transport systems or optimal control in energy systems. Recently, bi-objective A*-based search (BOA*) has shown state-of-the-art performance in large networks. This paper develops a bi-directional and parallel variant of BOA*, enriched with several speed-up heuristics. Our experimental results on 1,000 benchmark cases show that our bi-directional A* algorithm for bi-objective search (BOBA*) can optimally solve all of the benchmark cases within the time limit, outperforming the state of the art BOA*, bi-objective Dijkstra and bi-directional bi-objective Dijkstra by an average runtime improvement of a factor of five over all of the benchmark instances.

2012 ACM Subject Classification Computing methodologies → Search methodologies; Theory of computation → Shortest paths

Keywords and phrases Bi-objective search, heuristic search, shortest path problem

Digital Object Identifier 10.4230/LIPIcs.ESA.2021.3

Related Version *Previous Version*: `https://arxiv.org/abs/2105.11888`

Funding Research at Monash University is supported by the Australian Research Council (ARC) under grant numbers DP190100013 and DP200100025 as well as a gift from Amazon.

1 Introduction

Bi-objective search aims at finding a set of non-dominated, Pareto-optimal solutions in a domain with two objectives [2]. It has a wide range of real-world applications, such as planning routes for maritime transportation based on both the fuel consumption and the total risk of the vehicle route [18], or energy efficient paths for electric vehicles with arrival time considerations [13]. When the underlying system is a network, the problem is finding a set of paths between two points that are not dominated by other solution paths.

A comparison of traditional approaches to the bi-objective one-to-all shortest path problem, such as the label correcting algorithm in [15], the label setting approach in [6], and the adaptation of a near shortest path procedure in [1], was presented in [11]. These label-based approaches have been extended in several recent papers. A generalisation of Dijkstra's algorithm and its bi-directional counterpart (for the one-to-one variant) to the bi-objective problem was presented in [12] by utilising the pruning strategies of [5] to avoid

[1] Corresponding Author

© Saman Ahmadi, Guido Tack, Daniel Harabor, and Philip Kilby;
licensed under Creative Commons License CC-BY 4.0
29th Annual European Symposium on Algorithms (ESA 2021).
Editors: Petra Mutzel, Rasmus Pagh, and Grzegorz Herman; Article No. 3; pp. 3:1–3:15
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

expanding unpromising paths during the search. The results show that the state-of-the-art bi-objective Dijkstra algorithm can outperform the bounded label setting approach in [10] and the depth-first search-based *Pulse* algorithm in [5] on large-size instances.

Another recent work on point-to-point bi-objective search is the Bi-Objective A* search scheme (BOA*) in [17]. BOA* is a standard A* heuristic search that leverages the fast dominance checking procedure of [9] for multi-objective search. In contrast to eager dominance checking approaches, as in [12], BOA* lazily postpones dominance checking for newly generated nodes until their expansion. The experimental results in [17] on a set of large instances show that the efficient dominance checking helps BOA* to perform better than the bi-objective Dijkstra algorithm of [12] and other best-first search approaches such as the label-setting multi-objective search NAMOA* of [8] and its improved version with a dimensionality reduction technique called NAMOA*$_{\text{dr}}$ [9].

In this paper, we present Bi-Objective Bi-directional A* (BOBA*), a bi-directional extension of the BOA* algorithm that is easy to parallelise, uses different objective orders and includes several new heuristics to speed up the search. Our experiments on a set of 1,000 large test cases from the literature show that BOBA* can solve all of the cases to optimality, outperforming the state-of-the-art algorithms in both runtime and memory requirement.

2 Background and Notation

For a directed bi-objective graph $G = (S, E)$ with a finite set of states S and a set of edges $E \subseteq S \times S$, the point-to-point bi-objective search problem is to find the set of Pareto-optimal solution paths from $start \in S$ to $goal \in S$ that are not dominated by any solution for both objectives. Every edge $e \in E$ has two non-negative attributes accessed via the cost function $\mathbf{cost} : E \rightarrow \mathbb{R}^+ \times \mathbb{R}^+$. A path is a sequence of states $s_i \in S$ with $i \in \{1, \dots, n\}$. The cost of path $p = \{s_1, s_2, s_3, \dots, s_n\}$ is then defined as the sum of corresponding attributes on all the edges constituting the path as $\mathbf{cost}(p) = \sum_{i=1}^{n-1} \mathbf{cost}(s_i, s_{i+1})$. Following the standard notation in the heuristic search literature, we define our search objects to be *nodes*. A node x is a tuple that contains a state $s(x) \in S$; a value $\mathbf{g}(x)$ which measures the cost of a concrete path from the *start* state to state $s(x)$; a value $\mathbf{f}(x)$ which is an estimate of the cost of a complete path from *start* to *goal* via $s(x)$; and a reference $parent(x)$ which indicates the parent of node x. We perform a systematic search by *expanding* nodes in best-first order. Each expansion operation *generates* a set of successor nodes, each denoted $Succ(s(x))$, which are added into an *Open* list. The *Open* list sorts the nodes according to their **f**-values in an ascending order, for the purpose of further expansion.

As with other A*-based algorithms, we compute **f**-values using a consistent and admissible heuristic function $\mathbf{h} : S \rightarrow \mathbb{R}^+ \times \mathbb{R}^+$ [7]. In other words, $\mathbf{f}(x) = \mathbf{g}(x) + \mathbf{h}(s)$ where $\mathbf{h}(s)$ is a lower bound on the cost of paths from state s to *goal*. Moreover, in bi-objective search, the cost function has two components which means that every (boldface) cost function is a tuple, eg. $\mathbf{f} = (f_1, f_2)$ or $\mathbf{h} = (h_1, h_2)$ and all operations are considered element-wise.

▶ **Definition 1.** *A heuristic function* $\mathbf{h}$ *is consistent if we have* $\mathbf{h}(s) \leq \mathbf{cost}(s, t) + \mathbf{h}(t)$ *for every edge* $(s, t) \in E$. *It is also admissible iff* $\mathbf{h}(s) \leq \mathbf{cost}(p)$ *for every* $s \in S$ *and the optimal path* p *from state* s *to the goal state.*

▶ **Definition 2.** *For every pair of nodes* (x, y) *associated with the same state* $s(x) = s(y)$, *node* y *is dominated by* x *if we have* $g_1(x) < g_1(y)$ *and* $g_2(x) \leq g_2(y)$ *or if we have* $g_1(x) = g_1(y)$ *and* $g_2(x) < g_2(y)$. *Node* x *weakly dominates* y *if* $g_1(x) \leq g_1(y)$ *and* $g_2(x) \leq g_2(y)$.

Bi-objective A*. The Bi-Objective A* (BOA*) algorithm [17] first obtains its heuristic function $\mathbf{h}$ using two basic one-to-all searches on the reversed graph. BOA* can then establish lower bounds on the cost of complete paths or $\mathbf{f}$-values using the admissible heuristic $\mathbf{h}$. Although either of the two objectives can potentially play the key role in the bi-objective setting, standard BOA* usually chooses the first objective in the (f_1, f_2) order. The search then expands all the promising nodes based on their cost estimates so as to ensure the node with the lexicographically smallest $\mathbf{f}$-value is explored first. The algorithm terminates when there is no node in *Open* while keeping all the non-dominated nodes associated with the *goal* state in the solution set *Sol*. The main steps of the standard BOA* algorithm can be found in Algorithm 2, scripted with normal line numbers (without asterisk *) in black.

▶ **Theorem 3.** *BOA* computes a set of cost-unique Pareto-optimal solution paths [17].*

BOA* utilises an efficient strategy to check nodes for their dominance, originally employed in [9] for multi-objective search. The idea is simple yet powerful. Let us assume A* explores the graph in the (f_1, f_2) order, that is, nodes are extracted based on their f_1-value in order (with tie-breaking on f_2-values). Meanwhile, x and y are two nodes associated with the same state or $s(x) = s(y)$ in the *Open* list where x is going to be expanded first, i.e., we have $f_1(x) \leq f_1(y)$. Since both nodes have used the same heuristic value as $h_1(s(x)) = h_1(s(y))$ to determine their cost estimate f_1, we can conclude $g_1(x) \leq g_1(y)$. Therefore, the second node will be dominated by the first node if $g_2(x) \leq g_2(y)$ as shown in [9] in detail. BOA* takes advantage of this dimension reduction technique by systematically keeping track of the g_2-value of the last non-dominated node using $g_2^{min}(s(x))$ via line 11 of Algorithm 2.

BOA* can also prune some of the dominated nodes during the expansion with a similar reasoning via line 28 of Algorithm 2. This is done by comparing the newly generated node of a state and the last expanded node of the state against their secondary costs g_2. Furthermore, BOA* prunes unpromising nodes based on their cost estimate to the *goal* state, which is known as pruning by bound. Given $g_2^{min}(goal)$ as the upper bound of the secondary cost, partial paths will be pruned if the cost estimate of their complete paths to *goal* on g_2 is greater than that of the last solution already stored in $g_2^{min}(goal)$. Interested readers are referred to the standard BOA* algorithm in [17] for the detailed proof discussion.

Challenges. Lazy dominance checking in BOA* slows down the operations in the *Open* list and consumes more space. In contrast to the costly linear dominance checking approach where new nodes are checked against all of the previously generated nodes associated with a state before their insertion into the *Open* list, BOA* may add a node for which we have an unexpanded dominant node in *Open*. Thus, the search generates more nodes (using extra memory), and the *Open* list will inevitably be longer. Moreover, BOA* is only able to search the graph in one direction and with a specific objective ordering, whereas there can be cases with better performance on the reverse objective ordering as shown in [17]. Our preliminary experiments also reveal that searching backwards (from *goal* to *start*) may lead to significant improvements in the overall runtime. There are also some inefficiencies in BOA* which can be addressed with extra considerations. As an example, for the simple graph in Figure 2, BOA* needs to expand all intermediate states for each individual solution, despite the fact that some of them are not offering any alternative (non-dominated) path to *goal* (eg. s_2).

3 Bi-objective Bi-directional A* Search

Recent improvements in bi-directional heuristic search have introduced new techniques to reduce the number of necessary node expansions, such as Near-Optimal Bi-directional Search in [16] and Dynamic Vertex Cover Bi-directional Search in [14]. Given the single-objective

Algorithm 1 Bi-Objective Bidirectional A* (BOBA*) High-level.

Input: A problem instance (G, **cost**, s_{start}, s_{goal})
Output: A set of cost-unique Pareto-optimal solutions
1 **do in parallel**
2 $h'_1, ub'_2 \leftarrow$ **cost**-bounded A* from s_{start} to s_{goal} on G in (f_1, f_2) order
3 $h_2, ub_1 \leftarrow$ **cost**-bounded A* from s_{goal} to s_{start} on Reversed(G) in (f_2, f_1) order
4 **do in parallel**
5 $h'_2, ub'_1 \leftarrow$ **cost**-bounded A* from s_{start} to s_{goal} on G in (f_2, f_1) order
6 $h_1, ub_2 \leftarrow$ **cost**-bounded A* from s_{goal} to s_{start} on Reversed(G) in (f_1, f_2) order
7 **do in parallel**
8 $Sol \leftarrow$ BOA*$_{\text{enh}}$ for (G, **cost**, s_{start}, s_{goal}) with heuristics $(\mathbf{h}, \mathbf{ub}, \mathbf{h}')$ in (f_1, f_2) order
9 $Sol' \leftarrow$ BOA*$_{\text{enh}}$ for (Rev(G), **cost**, s_{goal}, s_{start}) with heuristics $(\mathbf{h}', \mathbf{ub}', \mathbf{h})$ in (f_2, f_1) order
10 **return** $Sol + Sol'$

nature of the conventional shortest path problem, none of the existing front-to-end or front-to-front algorithms can practically tackle the bi-objective shortest path problem without incorporating necessary modifications. Moreover, those algorithms are not necessarily efficient for the bi-objective search as obtaining the solutions' cost would no longer be possible in $O(1)$ time. In the conventional bi-objective setting where both searches work on the same objective, every state offers a set of non-dominated nodes (partial paths) in each direction, and handling frontier collisions (obtaining all of the complete *start-goal* joined paths of the state) would be an exhaustive process which can outweigh the speed-up achieved by expanding fewer nodes. Our preliminary experiments also confirmed that the conventional front-to-end bi-directional search with an efficient partial paths coupling approach can potentially generate fewer nodes but shows poor performance compared to the unidirectional search scheme BOA*.

We now present our contributions to the problem by explaining our Bi-Objective Bi-directional A* search (BOBA*). BOBA* employs two complementary (enhanced) uni-directional BOA* to search the solution space in both (forward and backward) directions with different objective orders ((f_1, f_2) and (f_2, f_1)). Therefore, since the algorithm does not perform partial paths coupling, we do not need to handle frontier collisions. In other words, each uni-directional BOA* is allowed to explore the entire graph towards the opposite end for each individual solution. The high level structure of BOBA* is given in Algorithm 1. BOBA* first obtains the preliminary heuristics and then performs two individual searches that explore the graph in both directions concurrently. The output will then be the aggregation of solutions found in each search routine. To avoid searching for the same cost-optimum paths in both directions, BOBA* always chooses different orders for each direction. Figure 1(Left) depicts the way Pareto-optimal solutions are found based on two searches in the two orders. Initial solutions (sol_{init}) at both ends are typically the minimum cost paths already obtained via the heuristic searches for each objective. These cost-optimum paths can also initialise the global upper bounds (ub_1, ub_2) needed by the pruning by bound strategies in BOA*. The upper bounds are updated (always decreasing) during the search every time a valid solution is found, and sol_{last} is the last solution for which we have had $f_1 < ub_1$ and $f_2 < ub_2$.

▶ **Definition 4.** *For every state $s \in S$, $\mathbf{ub}(s)$ is the upper bound on* **cost** *of complementary paths from state s to goal, eg., $ub_1(s)$ denotes the upper bound on $cost_1$.*

▶ **Definition 5.** *A path/node/state x is invalid if its estimated costs $\mathbf{f}(x)$ are not in the search global upper bounds (ub_1, ub_2), i.e., x is invalid if $f_1(x) \geq ub_1$ or $f_2(x) \geq ub_2$.*

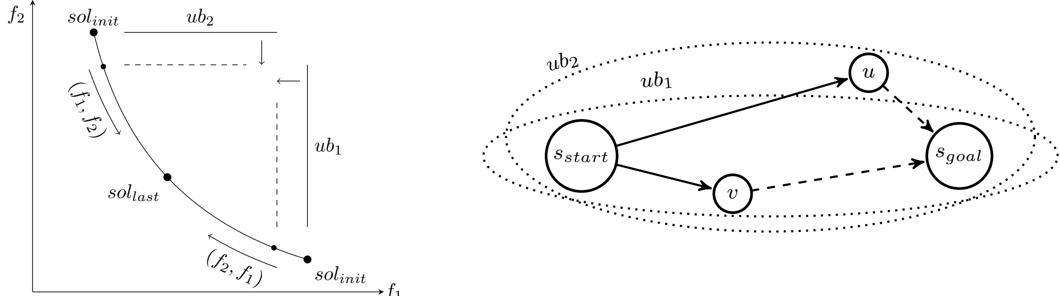


Figure 1 Left: Objective orders, bounds and Pareto-optimal solutions. Right: Schematic of states outside or inside of upper bounds. State u is out of bounds for f_1 and will be discarded in f_2 search.

3.1 Preliminary Heuristics

BOBA* requires both lower and upper bounds on the costs of complementary paths for each direction via four individual searches. In each search, we calculate a state's upper bound to be the cost of the optimum path using the non-primary objective. For example, the optimum path to state s for the first objective sets both $h_1(s)$ and $ub_2(s)$ (here $ub_2(s)$ is the cost of the path using the second objective). BOA* traditionally uses two runs of Dijkstra's algorithm to initialise lower bounds. For difficult cases, this initialisation time is usually outweighed by the main search time, but there can be simple cases where the total time of these heuristic searches dominates the main search time, especially in large instances. As a more efficient initialisation approach, we replace Dijkstra's algorithm with cost-bounded A* (or cost-bounded Dijkstra without heuristics), as formally stated in Lemma 6 and shown in lines 2-6 of Algorithm 1.

▶ **Lemma 6.** *The preliminary A* search on f_1 (or f_2) can terminate before expanding a state with $f_1 > ub_1$ (or $f_2 > ub_2$).*

Proof. Assume that a forward BOA* is intended and, therefore, the corresponding heuristics (via two backward searches) are required. If we start with two simple backward A* searches (one for each objective), each optimum *start-goal* path gives us two bounds as $(h_1(s_{start}), ub_2(s_{start}))$ and $(h_2(s_{start}), ub_1(s_{start}))$. Now, given $h_1(s_{start})$ and $h_2(s_{start})$ as the global lower bounds on f_1 and f_2-values respectively, we will have $f_1 \geq h_1(s_{start})$ and $f_2 \geq h_2(s_{start})$ for every *start-goal* path. Therefore, any state with a cost estimate of $f_1 > ub_1(s_{start})$ in the A* search on the first objective, and similarly $f_2 > ub_2(s_{start})$ in the search on the second objective, will be dominated by one of the optimum solutions. On the other hand, since A* expands states in an increasing order of f-values, each heuristic search can terminate early with the first out-of-bound state, guaranteeing that all paths via unexplored states are already dominated. ◀

Algorithm 1 shows the parallel computation of all necessary heuristics in BOBA* in two phases. In the first phase (lines 2-3), we can execute our cost-bounded A* using any admissible heuristic for the primary objective (f_1 or f_2) and with tie-breaking on the secondary objective (f_2 or f_1). Note that the upper bounds are unknown prior to the searches in phase one, i.e., we initially have $ub_1 = ub_2 = \infty$, but we can update our global upper bounds as soon as we establish the optimal solution in each search. The initialisation step of BOBA* can be further improved for the heuristic searches in the opposite direction in phase two (lines 5-6). Once the necessary heuristics in one direction have been obtained, the heuristic search

in the opposite direction can use the lower bounds obtained from the first round as more informed heuristics. That is, the second phase of our cost-bounded A* searches are normally executed faster. Moreover, the opposite search in the second round can take advantage of the reduced search space resulting from the first round, delivering better quality heuristics without needing to expand already invalidated (out-of-bound) states. Lemma 8 states this technique more formally.

▶ **Example 7.** State v in Figure 1 (right) is within the upper bound of both objectives and will be expanded in the opposite direction. However, state u is observed out of bounds for the first objective (but within the bound of the second objective) and will then be discarded if it is going to be expanded in the second round of our heuristic searches. Note that violating at least one objective's upper bound is enough to mark nodes (or states) invalid.

▶ **Lemma 8.** *In the preliminary A* search on f_1 (or f_2), states with an estimated cost of $f_2 > ub_2$ (or $f_1 > ub_1$) are not part of any solution path.*

Proof. States with $f_2 > ub_2$ are dominated by the optimum path obtained for the first objective. This means that unexplored states with an estimated cost of $f_2 > ub_2$ are all invalid. Therefore, the following search on f_1 can ignore expanding such states knowing that no non-dominated solution can be found via invalid states. The same reasoning is valid for the reverse order. ◀

3.2 Bi-directional Search

BOBA* performs two enhanced BOA* concurrently, one from each direction. Algorithm 2 shows the details of our first enhanced BOA* algorithm used in BOBA* (forward search in the (f_1, f_2) order). Lines scripted in black are from the standard BOA* and the red lines with an asterisk (*) next to line numbers are our proposed enhancements. To be consistent with the BOA* notation, we obtain the latest global upper bounds from g_{min} values, i.e., we have $g_2^{min}(s_{goal}) = ub_2$ and $g_1^{min}(s_{start}) = ub_1$. This is because the forward search on (f_1, f_2) updates $g_2^{min}(s_{goal})$ for every solution, whereas the backward search on (f_2, f_1) simultaneously updates $g_1^{min}(s_{start})$. We also add a pruning criterion to discard nodes violating the primary upper bound $g_1^{min}(s_{start})$ in line 29. To achieve the backward search, we simply reverse the search direction and the objective ordering. For example, instead of $g_2^{min}(s_{goal})$ and $h_1(s(x))$ in Algorithm 2 we will have $g_1^{min}(s_{start})$ and $h'_2(s(x))$ respectively (the backward search establishes its f-values using $\mathbf{h}'$). Note that each search has an independent *Open* list. Now we describe our contributions to the individual searches of BOBA* followed by their formal presentation in Lemmas 10-12.

Early solution update. This strategy allows the search to update the secondary upper bound and establish a tentative solution before reaching the *goal* state. This is done via line 17 of Algorithm 2 by coupling nodes with their complementary shortest path to *goal*. If the joined path is valid, the corresponding node is then temporarily added to the solution set knowing that solution nodes with a state other than s_{goal} (or $s(x) \neq s_{goal}$) must be joined with their complementary shortest path. This strategy can be further improved by not expanding nodes for which we have a unique non-dominated complementary path. This heuristic is incorporated in line 22 and is formalised in Lemma 10.

Secondary heuristic tuning. Bi-directional search provides our algorithm with a great opportunity to further improve the quality of the preliminary heuristics. Since the main search of BOBA* has more information about non-dominated paths to states and constantly updates upper bounds, there can be more outliers that our preliminary heuristic searches are

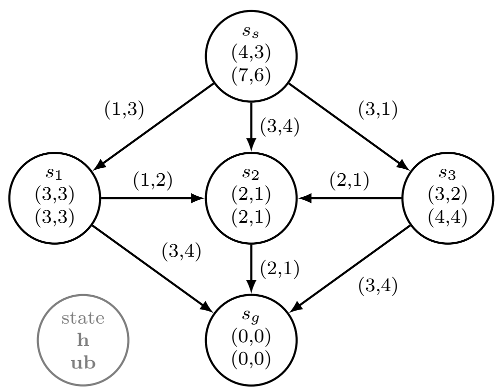
Algorithm 2 Enhanced forward Bi-Objective A* (BOA*$_{\text{enh}}$) in (f_1, f_2) objective ordering.

Inputs: A problem instance (G, **cost**, s_{start}, s_{goal}) and heuristics ($\mathbf{h}$, **ub**, $\mathbf{h}'$)
Output: A set of cost-unique Pareto-optimal solutions

1 $Open \leftarrow \emptyset, Sol \leftarrow \emptyset$
2 $g_1^{min}(s) \leftarrow g_2^{min}(s) \leftarrow \infty$ for each $s \in S$
3 $x \leftarrow$ new node with $s(x) = s_{start}$
4 $\mathbf{g}(x) \leftarrow (0,0)$, $\mathbf{f}(x) \leftarrow (h_1(s_{start}), h_2(s_{start}))$, $parent(x) \leftarrow Null$
5 Add x to $Open$
6 **while** $Open \neq \emptyset$ **do**
7   Remove a node x with the lexicographically smallest (f_1, f_2) values from $Open$
8*   **if** $f_1(x) \geq g_1^{min}(s_{start})$ **then break**
9   **if** $g_2(x) \geq g_2^{min}(s(x))$ ***or*** $f_2(x) \geq g_2^{min}(s_{goal})$ **then continue**
10*   **if** $g_2^{min}(s(x)) = \infty$ **then** $h_1'(s(x)) \leftarrow g_1(x)$
11     $g_2^{min}(s(x)) \leftarrow g_2(x)$
12   **if** $s(x) = s_{goal}$ **then**
13* $z \leftarrow$ last node in Sol
14*   **if** $(z \neq Null$ ***and*** $f_1(z) = f_1(x))$ **then** Remove z from Sol
15   Add x to Sol
16   **continue**
17*   **if** $g_2(x) + ub_2(s(x)) < g_2^{min}(s_{goal})$ **then**
18* $g_2^{min}(s_{goal}) \leftarrow g_2(x) + ub_2(s(x))$
19* $z \leftarrow$ last node in Sol
20*   **if** $(z \neq Null$ ***and*** $f_1(z) = f_1(x))$ **then** Remove z from Sol
21*   Add x to Sol
22*   **if** $h_1(s(x)) = ub_1(s(x))$ **then continue**
23   **for** *all* $t \in Succ(s(x))$ **do**
24         $y \leftarrow$ new node with $s(y) = t$
25         $\mathbf{g}(y) \leftarrow \mathbf{g}(x) + \mathbf{cost}(s(x), t)$
26         $\mathbf{f}(y) \leftarrow \mathbf{g}(y) + \mathbf{h}(t)$
27         $parent(y) \leftarrow x$
28   **if** $g_2(y) \geq g_2^{min}(t)$ ***or*** $f_2(y) \geq g_2^{min}(s_{goal})$ **then continue**
29*   **if** $f_1 \geq g_1^{min}(s_{start})$ **then continue**
30   Add y to $Open$
31 **return** Sol

not aware of. Therefore, benefiting from the main property of BOA* (finding non-dominated nodes in order), we can tune our findings over the preliminary searches and empower the pruning by bound strategy of the concurrent search in the opposite direction. This tuning is done in $O(1)$ time by updating the secondary heuristics of the reverse direction via line 10 of Algorithm 2. Note that h_1' denotes the secondary heuristic in the backward search where BOBA* uses f_2 as its primary cost. We discuss the correctness of this technique in Lemma 12.

▶ **Example 9.** We explain these strategies by just running the forward search of BOBA* for the graph in Figure 2 and iterations in Table 2. In the first iteration, the forward search explores the node associated with the start state s_s. Since the primary (heuristic) cost-optimum path from s_s is initially valid, the search immediately updates its secondary upper bound via the early solution update strategy by setting $g_2^{min}(s_g) = 6$ and adds the node into the Sol set with costs $(4, 6)$. During the s_s expansion, we notice that the extended path for state s_1 is invalid ($f_2(y) \geq g_2^{min}(s_g)$ or $3 + 3 \geq 6$). Therefore, the partial path to s_1 is pruned meaning that s_2 is no longer reachable via its primary cost optimum path. Nodes generated for states s_2 and s_3, however, are successfully added to $Open$. In the second iteration, the algorithm picks the node associated with s_2 (with higher priority). Now, since this is the first time we see s_2 being expanded, and since future visits will always have higher costs (via s_3 with $g_1 = 5$ for example), we can update the lower bound of reaching



It.	*Open* list $(s(x)$, $\mathbf{g}$(x), $\mathbf{f}$(x))	*Sol* found	Update $g_2^{min}(s_g)$
1	↑(s_s, (0,0), (4,3))	(4,6)	$\infty \to 6$
2	↑(s_2, (3,4), (5,5)) (s_3, (3,1), (6,3))	(5,5)	$6 \to 5$
3	↑(s_3, (3,1), (6,3))		
4	↑(s_2, (5,2), (7,3))	(7,3)	$5 \to 3$
5	empty		

parent arrays	s_s	s_1	s_2	s_3	s_g
`par_state`	[Null]		[s_s, s_3]	[s_s]	
`par_path_id`	[Null]		[1, 1]	[1]	

Figure 2 Left: An example graph with **cost** on the edges, and with (state, **h**, **ub**) inside the nodes. Right: Status of the *Open* list, new solution (*Sol*) and secondary upper bound $g_2^{min}(s_g)$ in every iteration (It.) for the forward search on the (f_1, f_2) ordering. Symbol ↑ beside nodes denotes the expanded min-cost node. The second table shows the status of the parent arrays of the states when the search terminates.

s_2 from s_s knowing that all possible shorter paths have already been invalidated. This is done by updating $h'_1(s_2) = 3$. Note that from the preliminary heuristics, we already had $h'_1(s_2) = 2$ (lower bound from s_s to s_2). After this update, the backward search would have better quality secondary lower bounds and can effectively prune more nodes (the backward primary heuristic is h'_2). We skip the backward search for now and continue with our forward expansions. As coupling the node (associated with s_2) with its (complementary) primary cost-optimum path yields a valid complete path, the search updates its secondary upper bound and temporarily adds the node to the *Sol* set with costs $(5, 5)$. The search also skips expanding s_2 as it does not offer any non-dominated path to s_g. In the third iteration, the node associated with state s_3 is picked. This time, s_3 is expanded since coupling does not yield valid path. During the s_3 expansion, the search finds s_g invalid but adds s_2 into *Open*. In the fourth iteration, the node associated with s_2 is the only node in *Open* which reveals the final solution with costs $(7, 3)$, again with the early solution update strategy. This last solution also verifies that the temporary solution found in the second iteration is now a valid non-dominated solution, since the primary cost of the last solution is larger than that of the second solution $(5 < 7)$.

Now we formally prove the correctness of the presented techniques as follows.

▶ **Lemma 10.** *At every iteration, if $g_2(x) + ub_2(s(x)) < g_2^{min}(s_{goal})$, the next solution has a primary cost of $f_1(x)$ and a secondary cost of at most $g_2(x) + ub_2(s(x))$. Node x is also a terminal node if $h_1(s(x)) = ub_1(s(x))$.*

Proof. If the joined path is valid (its secondary cost is within the bounds), expanding nodes on the complementary shortest path will definitely navigate us to s_{goal} with a valid secondary cost as they offer the same f_1-value. This means we can efficiently update the secondary upper bound earlier assuming that a potential solution path is already established. Therefore, valid joined paths determine the primary cost f_1 of the next solution along with setting a new upper bound for the secondary cost f_2. Furthermore, given the secondary cost as a tie-breaker in the preliminary heuristic searches, states with $h_1(s(x)) = ub_1(s(x))$ would only offer one complementary path optimum for both objectives. As none of the states on the

complementary path would offer an alternative non-dominated path to s_{goal}, the search can save time by not expanding such terminal nodes. Therefore, nodes with $h_1(s(x)) = ub_1(s(x))$ are terminal nodes if they appear on any solution path. ◀

The early solution update strategy above guarantees that the primary cost f_1 of the next solution is determined by the joined path, but this does not apply to its secondary cost. E.g., we might see two consecutive temporary solutions with the same f_1-value but different (sequentially) valid secondary costs. Therefore, the search needs to make sure that the previously added solution is not dominated by the next potential solution, and if it is dominated, it must be removed from the non-dominated solution set Sol. We address this matter in $O(1)$ time by checking our last (temporary) solution against new solutions for dominance as shown in lines 13-14 and 19-20 of Algorithm 2. This pruning is formally stated in the following Lemma 11.

▶ **Lemma 11.** *Given z and x as the last and new temporary solution nodes respectively, node z represents a non-dominated solution if $f_1(z) < f_1(x)$. The temporary solution node z is dominated by x if $f_1(z) = f_1(x)$.*

Proof. Since the secondary cost of the new solution x is already checked to be smaller than that of the last solution z stored in $g_2^{min}(s_{goal})$, we have $f_2(x) < g_2^{min}(s_{goal})$ if $s(x) = s_{goal}$ or $g_2(x) + ub_2(s(x)) < g_2^{min}(s_{goal})$ if $s(x) \neq s_{goal}$. On the other hand, since x is the new potential solution and the search explores nodes in an increasing order of f_1-values, we must have $f_1(z) \leq f_1(x)$. Therefore, if $f_1(z) < f_1(x)$, we can see that the temporary solution z is now a non-dominated solution. Otherwise, if $f_1(z) = f_1(x)$, the last solution z is dominated by the new solution x because the new solution offers a lower secondary cost. ◀

We now show the correctness of the heuristic tuning approach in BOBA*.

▶ **Lemma 12.** *The secondary heuristic tuning maintains the correctness of A* heuristics.*

Proof. BOBA* expands partial paths in the increasing order of f-values. This means the first expanded node of every state is guaranteed to have the minimum valid primary cost g_1 in each search direction, and all of the following valid nodes will have a larger primary cost. Moreover, since BOBA* uses different objective ordering for its searches, updated lower bounds in one direction represent the secondary heuristics of the other direction. Therefore, we can guarantee that the updated secondary heuristic is still admissible as there will not be any min-cost path to states better than what their first expanded node presents. Furthermore, the tuning strategy only updates the secondary heuristics of the opposite search, i.e., $h'_1(s(x))$ in the forward and $h_2(s(x))$ in the backward search. Therefore, the preliminary primary heuristics $h_1(s(x))$ and $h'_2(s(x))$ are unchanged and the A* searches are correct. ◀

Considering the correctness of the enhancements presented above, we now show the correctness of our BOBA* algorithm.

▶ **Theorem 13.** *BOBA* returns a set of cost-unique non-dominated solution paths.*

Proof. BOBA* executes two enhanced BOA* searches concurrently, each capable of finding all of the solutions. Therefore, we just need to show the correctness of the stopping criteria. Each (enhanced) BOA* searches the primary objective's domain in the increasing order of f-values and continually shrinks the secondary objective's domain every time a valid solution is found. Furthermore, since BOBA* shares the upper bounds between its searches, each search can terminate with the first node violating the main objective's upper bound (and

consequently other unexplored nodes with larger f-values in *Open*) knowing that the rest of the objective's domain has already been investigated by the concurrent search (see Figure 1). Therefore, the aggregation of the solutions found in each search yields a complete set of cost-unique non-dominated solutions. ◀

4 Practical Considerations

As BOA* enumerates all non-dominated paths, the size of *Open* can grow exponentially over the course of search. Furthermore, the huge number of nodes in difficult cases may result in major memory issues. For instance, for one particular case in our experiments BOA* generates two billion nodes. We now present two techniques to handle search nodes more efficiently.

More efficient *Open* list. To achieve faster operations in our *Open* lists, since the lower and upper bounds on the f-values of the nodes in BOBA* are known prior to its main searches, we replace the conventional heap-based lists with fixed-size bucket lists without tie-breaking [3]. In contrast to other problems where the bucket list is regularly resized and the list is sparsely populated, for the significant number of (cost-bounded) nodes in our problem we expect to see almost all of the buckets filled. Note that the search may also expand dominated nodes if they are not extracted in a lexicographical order (i.e., nodes are sorted based on their primary cost only), but BOBA* can still obtain cost-unique solutions via the dominance checks incorporated in lines 13-14 and 19-20 of Algorithm 2 as formally stated in the following Lemma 14.

▶ **Lemma 14.** *BOBA* is able to obtain cost-unique solutions even without tie-breaking in its Open lists.*

Proof. Let z and x be two solution nodes where $f_1(z) = f_1(x)$ and z is dominated by x. Without any tie-breaking, the search may temporarily add dominated node z to the solution set first. In the next iterations, when x is extracted, the search performs a quick dominance check by comparing the f_1-value of the new node x against that of the previous solution z and substitutes the dominated solution with the new solution x if $f_1(z) = f_1(x)$, as already shown in the early solution update strategy and Lemma 11 in detail. Therefore, BOBA* computes cost-unique non-dominated solutions even without tie-breaking. ◀

Memory efficient backtracking. Creating nodes is necessary to appropriately navigate the search to valid solution paths. Each new node occupies a constant amount of memory and conventionally contains essential information about paths such as costs and also back-pointers for solution path construction. Considering the difficulty of the problem and the significant number of generated nodes, we suggest a more memory efficient approach for the solution path construction in BOBA*. Since BOBA* only expands nodes once, we propose to recycle the memory used to store heavy processed nodes, while storing their backtracking information in other compact data structures. This technique results in a major reduction in memory use as part of the nodes' information would no longer be required for backtracking. We explain our compact approach using an example from Figure 2. Assume that in the second iteration of the algorithm, we want to store the backtracking information of the node corresponding to s_2 with s_s as the parent state. To this end, we keep two (initially empty) dynamic arrays for each state: one to store the parent state of the node `par_state`, and another to look up the corresponding path index in the parent state `par_path_id`. For our example, since the first

path to s_2 is derived from the first non-dominated path of s_s, we store this sequence in s_2 as `par_state[1]=`s_s and `par_path_id[1]=1`. Similarly, for the second expansion of s_2 with s_3 as the parent in the fourth iteration, we update s_2 arrays, this time with `par_state[2]=`s_3 and `par_path_id[2]=1`. Figure 2 also shows the situation of our parent arrays when the forward search terminates. As a further optimisation, we can store the index of incoming edges (which are usually very small integers) instead of parent states. We will investigate the impacts of this compression on memory usage in the following section.

5 Empirical Study and Analysis

We compare our BOBA* with recent algorithms designed for the bi-objective search problem. The selected algorithms are the bi-objective variants of Dijkstra's algorithm (Dij) and bi-directional Dijkstra (Bi-Dij) from [12], and bi-objective A* (BOA*) from [17]. We use all seven benchmark instances of [12] which include 700 random *start-goal* pairs from the large road networks in the 9th DIMACS challenge [4] with (*distance*, *time*) as objectives. To further challenge the algorithms, we used the competition's random pair generator to design an additional set of 300 test-cases for the larger networks in the DIMACS instances: E, W and CTR (100 cases each) with up to 15 M nodes and 33 M edges. The details of the instances can be found in [4].

Implementation. We implemented our BOBA* algorithms based on a parallel framework using two cores in C++ and used the C implementations of the Dij, Bi-Dij and BOA* algorithms kindly provided to us by their authors. We also fixed the scalability and tie-breaking issues in the standard BOA* algorithm before running the experiments. All code was compiled with O3 optimisation settings using the GCC7.4 compiler. Our codes are publicly available[2]. We ran the 1,000 experiments on an Intel Xeon E5-2660V3 processor running at 2.6 GHz and with 128 GB of RAM, under the SUSE Linux 12.4 environment and with a one-hour timeout.

BOA* analysis. The search in BOA* can be performed in different directions and objective orders, resulting in four variants. We also consider the virtual best version of the four, called BOA*$_{\text{best}}$ (essentially assuming an oracle that could select the best variant). Figure 3 is a cactus plot comparing the performance of all BOA* variants including the virtual best, showing how many instances can be solved in a given time (the plot only shows the longest running 300 instances). Backward BOA* in the (f_1, f_2) order (in green) is the weakest variant, but the other variants perform quite similarly, and it is difficult to declare a clear winner. The performance of the virtual variant BOA*$_{\text{best}}$ shows that an ideally-tuned BOA* can be up to two times better than its standard version on average, but is still unable to solve 15 cases to optimality within the time limit (see Table 2).

Memory. We investigate the impact of our compact approach for the solution paths construction in BOBA*. Table 1 compares the memory usage of our proposed compact approach against the conventional backtracking approach on part of the benchmark instances. In order to measure the overall space requirement of the main search, we ignore the memory required for graph construction, shared libraries and heuristics, allocated prior to the search. The results show that BOBA* can solve all of the instances with both approaches within

[2] `https://bitbucket.org/s-ahmadi`

Figure 3 Performance of the BOA* variants.

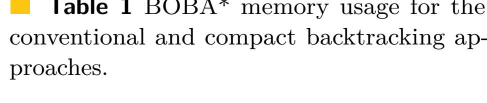


Table 1 BOBA* memory usage for the conventional and compact backtracking approaches.

Inst.	Saving Approach	Mem. (MB) Avg.	Mem. (MB) Max
NE	Conv.	307	7618
	Compact	**61**	**1186**
CAL	Conv.	326	5421
	Compact	**62**	**1009**
LKS	Conv.	5585	54331
	Compact	**955**	**9411**
E	Conv.	5836	62963
	Compact	**999**	**10895**
W	Conv.	5877	79602
	Compact	**925**	**10498**
CTR	Conv.	15835	99108
	Compact	**2662**	**21749**

the time limit, but the compact approach runs slightly faster and is five times more efficient on average in terms of memory. For the most difficult case in the experiments, the required memory of the compact approach can be as low as 21 GB (allocating 15M nodes with recycling) where the conventional approach needs 96 GB (allocating 1B nodes). Note that both approaches nearly expand the same number of nodes to solve the cases to optimality.

BOBA* performance. We compare the performance of our parallel BOBA* algorithm with the state-of-the-art Dij, Bi-Dij and BOA* algorithms from the literature. Table 2 shows the summary of experimental results for the 100 cases of each instance. For unsolved cases, we generously assume a runtime of one hour (the timeout). We also report the average memory usage of the main search of each algorithm over solved cases, ignoring the space allocated for their initialisation phase. The results in Table 2 show that the standard BOA* algorithm runs faster, needs less memory compared to both Dij and Bi-Dij algorithms and solves more instances. However, our new BOBA* outperforms BOA* in all of the instances, showing an (arithmetic) average speed up of 16 over all of the individual cases. For the average runtime of all instances, BOBA* is around five times faster than BOA*. We also compare the algorithms' performance over the solved instances for both CPU time and memory usage in Figure 4. As shown for both metrics, BOBA* delivers superior performance to its competitors by solving all of the instances to optimality within the time limit and with a maximum memory usage of 21 GB, compared to the nearly full (128 GB) memory usage of other algorithms in difficult instances. BOBA* also shows a massive speed up in the easy cases due to its efficient initialisation phase. It can solve 282 cases before BOA* solves its easiest case. Moreover, the figure shows that BOBA* completes the task eight times more efficiently in terms of memory than BOA* on average. Note that because of the difficulties in reporting the memory usage, we allow 1 MB tolerance in our experiments.

Multi-threading. We investigate the impact of multi-threading in BOBA* by running the (unmodified) algorithm on a single core instead of two cores, allowing decisions on scheduling of the threads to be made by the operating system. We compare single-core BOBA*$_{1c}$ with the virtual best variant BOA*$_{best}$ and our BOBA* with two cores in Table 2. The results show a slowdown of around 1.8 compared to parallel BOBA*, but it still outperforms

Table 2 Number of solved cases ($|S|$), runtime (in seconds) and average memory usage (Mem.) of algorithms over instances (Ins.). Memory in MB for the main search over solved cases.

			Runtime (s)			Mem.			Runtime (s)			Mem.
Alg.	Ins.	$\|S\|$	Min	Avg.	Max	Avg.	Ins.	$\|S\|$	Min	Avg.	Max	Avg.
BiDij	NY	100	0.53	0.92	6.66	21	CAL	98	4.04	168.58	3600.00	1206
Dij		100	0.31	1.47	16.98	75		100	2.73	88.93	1105.57	4283
BOA*		100	0.11	0.22	1.70	9		100	0.89	24.50	538.40	893
BOA*$_{\text{best}}$		100	0.11	0.16	0.65	6		100	0.89	8.63	187.04	498
BOA*$_{\text{enh}}$		100	0.01	0.12	0.67	2		100	0.02	6.69	147.44	67
BOBA*$_{1\text{c}}$		100	0.01	0.11	0.59	7		100	0.01	6.08	116.10	97
BOBA*		100	**0.00**	**0.08**	**0.40**	2		100	**0.00**	**3.75**	**64.80**	62
BiDij	BAY	100	0.61	1.32	11.96	35	LKS	69	6.12	1610.14	3600.00	2597
Dij		100	0.36	2.01	19.71	107		81	4.13	936.23	3600.00	11394
BOA*		100	0.13	0.38	4.10	19		89	1.30	528.12	3600.00	4854
BOA*$_{\text{best}}$		100	0.13	0.23	1.26	13		100	1.28	224.42	3500.80	9374
BOA*$_{\text{enh}}$		100	0.01	0.19	1.20	3		100	0.02	97.05	1077.96	787
BOBA*$_{1\text{c}}$		100	0.01	0.19	1.08	10		100	0.02	129.64	1488.41	1123
BOBA*		100	**0.00**	**0.13**	**0.86**	2		100	**0.00**	**69.68**	**812.17**	955
BiDij	COL	100	0.84	6.84	147.55	118	E	64	8.08	1611.10	3600.00	2223
Dij		100	0.52	6.27	111.81	348		82	5.48	1034.61	3600.00	16156
BOA*		100	0.19	1.20	20.53	77		89	1.72	552.64	3600.00	5299
BOA*$_{\text{best}}$		100	0.18	0.58	7.03	49		98	1.72	293.27	3600.00	8701
BOA*$_{\text{enh}}$		100	0.01	0.54	10.46	7		100	0.02	110.69	1684.94	850
BOBA*$_{1\text{c}}$		100	0.02	0.42	5.50	21		100	0.02	143.08	1818.63	1160
BOBA*		100	**0.00**	**0.34**	**6.58**	6		100	**0.00**	**75.94**	**952.32**	999
BiDij	FLA	100	2.11	51.49	1088.49	808	W	69	14.38	1585.11	3600.00	3476
Dij		100	1.37	52.34	1048.67	2630		74	10.04	1220.44	3600.00	12722
BOA*		100	0.48	6.42	153.07	276		94	3.14	416.94	3600.00	7705
BOA*$_{\text{best}}$		100	0.48	3.22	36.32	202		98	3.14	253.85	3600.00	8043
BOA*$_{\text{enh}}$		100	0.01	2.05	34.47	22		100	0.04	93.16	1792.57	784
BOBA*$_{1\text{c}}$		100	0.01	2.06	27.98	43		100	0.04	130.81	1834.67	1134
BOBA*		100	**0.00**	**1.31**	**19.86**	25		100	**0.02**	**70.41**	**971.67**	925
BiDij	NE	99	3.31	181.67	3600.00	1367	CTR	48	40.41	2666.66	3600.00	4904
Dij		100	2.18	68.41	1306.04	3281		51	29.29	2163.50	3600.00	16149
BOA*		100	0.73	16.83	332.36	587		77	8.46	1124.03	3600.00	9828
BOA*$_{\text{best}}$		100	0.70	10.51	332.01	533		89	8.46	745.16	3600.00	12418
BOA*$_{\text{enh}}$		100	0.02	4.79	97.25	49		100	0.03	340.50	2953.12	2178
BOBA*$_{1\text{c}}$		100	0.02	5.71	154.51	82		98	0.03	461.07	3600.00	2644
BOBA*		100	**0.00**	**3.41**	**90.01**	61		100	**0.02**	**246.01**	**2496.95**	2662

BOA*$_{\text{best}}$, solving more instances and showing an (arithmetic) average speed-up of six over all of the individual cases. Note that this virtual best version BOA*$_{\text{best}}$ does not exist, and the results are based on the best timings obtained via four individual runs of the standard BOA* algorithm.

Enhanced BOA*. To measure the contributions of our improvements to the uni-directional bi-objective search, we analyse the performance of the enhanced variant BOA*$_{\text{enh}}$ with the speed-up techniques above. This variant is obtained by switching off the backward search of BOBA*. Based on the results given in Table 2, BOA*$_{\text{enh}}$ outperforms BOA*$_{\text{best}}$ in almost all of the cases and shows a comparable performance to BOBA*$_{1\text{c}}$, solving a few more cases in the CTR map and using less memory on average. Comparing the maximum runtime over instances, we can see that BOBA*$_{1\text{c}}$ is faster than BOA*$_{\text{enh}}$ in half of the instances (maps NY, BAY, COL, FLA and CAL). Nonetheless, given the results in Table 2, BOBA* is still superior to BOA*$_{\text{enh}}$ showing a speed-up factor of 1.5 on average.

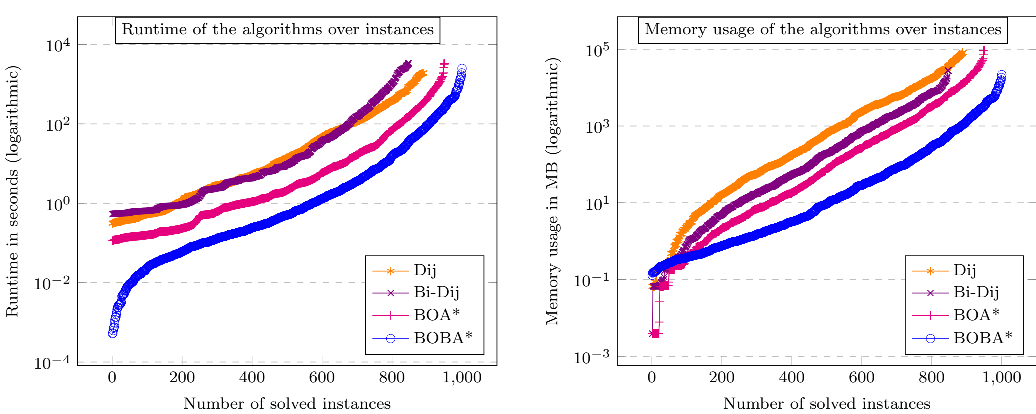


Figure 4 Cactus plots of algorithms' performance. Left: Runtime. Right: Search memory usage.

Bucket vs. heap. We found BOBA* with the bucket-based *Open* list around 1.8 times faster than BOBA* with heap for the same set of instances on average. Nonetheless, BOBA* with heap is still 2.2 times faster than standard heap-based BOA* (average over instances).

6 Conclusion

This paper introduced BOBA*, a bi-directional version of the state-of-the-art BOA* algorithm for bi-objective search. Our new algorithm explores the graph from both (forward and backward) directions in different objective orders in parallel. We enrich BOBA* with more efficient approaches for both the initial heuristic procedure and the solution path construction. We also present several speed up strategies to enhance BOBA*'s searches in various scenarios. Our experiments show that BOBA* outperforms the state-of-the-art algorithms in both runtime and memory use, solving all of the 1,000 benchmark cases to optimality in one hour timeout. Furthermore, compared to BOA*, BOBA* is five times faster and needs eight times less memory on average. Additional experiments reveal that the single-core version of BOBA* is around 1.8 times slower than the parallel version but still superior to the virtual best variant of BOA* and shows a comparable performance to BOA* enhanced with the speed-up strategies of this study.

References

1 W. Matthew Carlyle and R. Kevin Wood. Near-shortest and k-shortest simple paths. *Networks*, 46(2):98–109, 2005. doi:10.1002/net.20077.

2 Kalyanmoy Deb. Multi-objective optimization. In *Search methodologies*, pages 403–449. Springer, 2014.

3 Eric V. Denardo and Bennett L. Fox. Shortest-route methods: 1. reaching, pruning, and buckets. *Oper. Res.*, 27(1):161–186, 1979. doi:10.1287/opre.27.1.161.

4 DIMACS. 9th dimacs implementation challenge - shortest paths, 2005. URL: http://users.diag.uniroma1.it/challenge9.

5 Daniel Duque, Leonardo Lozano, and Andrés L. Medaglia. An exact method for the biobjective shortest path problem for large-scale road networks. *Eur. J. Oper. Res.*, 242(3):788–797, 2015. doi:10.1016/j.ejor.2014.11.003.

6 F Guerriero and R Musmanno. Label correcting methods to solve multicriteria shortest path problems. *Journal of optimization theory and applications*, 111(3):589–613, 2001.

7 Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. Syst. Sci. Cybern.*, 4(2):100–107, 1968. doi:10.1109/TSSC.1968.300136.

8 Enrique Machuca and Lawrence Mandow. Multiobjective heuristic search in road maps. *Expert Syst. Appl.*, 39(7):6435–6445, 2012. doi:10.1016/j.eswa.2011.12.022.

9 Francisco Javier Pulido, Lawrence Mandow, and José-Luis Pérez-de-la-Cruz. Dimensionality reduction in multiobjective shortest path search. *Comput. Oper. Res.*, 64:60–70, 2015. doi:10.1016/j.cor.2015.05.007.

10 Andrea Raith. Speed-up of labelling algorithms for biobjective shortest path problems. In *Proceedings of the 45th annual conference of the ORSNZ , 29-30 Nov 2010, Auckland, New Zealand*, page 313–322. Operations Research Society of New Zealand, 2010. URL: http://hdl.handle.net/2292/9789.

11 Andrea Raith and Matthias Ehrgott. A comparison of solution strategies for biobjective shortest path problems. *Comput. Oper. Res.*, 36(4):1299–1331, 2009. doi:10.1016/j.cor.2008.02.002.

12 Antonio Sedeño-Noda and Marcos Colebrook. A biobjective dijkstra algorithm. *Eur. J. Oper. Res.*, 276(1):106–118, 2019. doi:10.1016/j.ejor.2019.01.007.

13 Liang Shen, Hu Shao, Ting Wu, William HK Lam, and Emily C Zhu. An energy-efficient reliable path finding algorithm for stochastic road networks with electric vehicles. *Transportation Research Part C: Emerging Technologies*, 102:450–473, 2019.

14 Shahaf S. Shperberg, Ariel Felner, Nathan R. Sturtevant, Solomon Eyal Shimony, and Avi Hayoun. Enriching non-parametric bidirectional search algorithms. In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*, pages 2379–2386. AAAI Press, 2019. doi:10.1609/aaai.v33i01.33012379.

15 Anders J. V. Skriver and Kim Allan Andersen. A label correcting approach for solving bicriterion shortest-path problems. *Comput. Oper. Res.*, 27(6):507–524, 2000. doi:10.1016/S0305-0548(99)00037-4.

16 Nathan R. Sturtevant and Ariel Felner. A brief history and recent achievements in bidirectional search. In Sheila A. McIlraith and Kilian Q. Weinberger, editors, *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, pages 8000–8007. AAAI Press, 2018. URL: https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/17232.

17 Carlos Hernández Ulloa, William Yeoh, Jorge A. Baier, Han Zhang, Luis Suazo, and Sven Koenig. A simple and fast bi-objective search algorithm. In J. Christopher Beck, Olivier Buffet, Jörg Hoffmann, Erez Karpas, and Shirin Sohrabi, editors, *Proceedings of the Thirtieth International Conference on Automated Planning and Scheduling, Nancy, France, October 26-30, 2020*, pages 143–151. AAAI Press, 2020. URL: https://aaai.org/ojs/index.php/ICAPS/article/view/6655.

18 Aphrodite Veneti, Angelos Makrygiorgos, Charalampos Konstantopoulos, Grammati E. Pantziou, and Ioannis A. Vetsikas. Minimizing the fuel consumption and the risk in maritime transportation: A bi-objective weather routing approach. *Comput. Oper. Res.*, 88:220–236, 2017. doi:10.1016/j.cor.2017.07.010.