Improving Bi-Objective Shortest Path Search with Early Pruning

L. Mandow[a;*] **and J.L. Pérez de la Cruz**[a]

[a]Universidad de Málaga, Andalucía Tech, Dpto. de Lenguajes y Ciencias de la Computación, Málaga, España
ORCiD ID:

Abstract. Bi-objective search problems are a useful generalization of shortest path search. This paper reviews some recent contributions for the solution of this problem with emphasis on the efficiency of the dominance checks required for pruning, and introduces a new algorithm that improves time efficiency over previous proposals. Experimental results are presented to show the performance improvement using a set of standard problems over bi-objective road maps.

1 Introduction

Bi-objective search problems are a useful generalization of shortest path search. Since [6], several algorithms have been proposed for multi-objective (MO) search in general (e.g. [17] [13] [14] [19]), and bi-objective search in particular (e.g. [16] [15] [8] [5] [1][20] [9]). Ideas relevant to multi-objective search apply to bi-objective search as well. However, the special properties of bi-objective problems make it possible to devise specific and more efficient procedures for this case, turning it into a separate research subject.

The work of [13] showed the importance of the *consistency property* of heuristics in MO search, and analysed NAMOA*, an algorithm that is optimal in the number of paths explored under such property.

Although the number of explored paths is a fundamental measure in the efficiency of MO search, other important computational considerations need to be taken into account. In A*, each path to a known node can be pruned or preserved with a constant-time check on its cost. However, MO search requires dominance checks betweeen the vector costs of all paths (open or closed) reaching a given node. Additionally, newly generated paths need to be checked for dominance against the set of solution costs already found. A dominance check between a given vector and a set of vectors is potentially a computationally costly operation [2]. Several recent contributions on the runtime efficiency of multi-objective search have focused on reducing this cost, with special emphasis in the bi-objective case [14] [9].

This paper reviews these recent contributions under a common framework, and identifies a new additional technique that further improves dominance check efficiency.

Section 2 presents the problem definition and necessary notation. Sections 3 and 4 review recent contributions in efficient dominance checks for biobjective search and identify a new area of improvement. Sections 5 and 6 describe a new biobjective search algorithm and discuss some properties. Experiments on the efficiency of the new approach over road map benchmark problems are presented and discussed in sections 7 and 8. Finally, conclusions are summarized and future work is outlined.

2 Problem definition and notation

Let G be a locally finite directed weighted graph $G = (S, A, C)$, where S is a finite set of *states*; A is a set of *arcs* $A = \{(i_i, j_1), ..., (i_m, j_m)\} \subseteq S \times S$; and C is a function that assigns to each arc $(i, j) \in A$ a vector of 2 positive costs $\vec{c}(i,j) = (c^1_{ij}, c^2_{ij}) \in \mathbb{R}^{2+}$. Let a path from state s_1 to state s_k be a sequence of states $(s_1, s_2, \dots s_k)$ such that $\forall i < k \; (s_i, s_{i+1}) \in A$. Let the cost vector of a path be defined as the sum of the cost vectors of its component arcs. Bi-objective cost vectors induce a partial order preference relation $\prec$ called *dominance*,

$\forall \vec{y}, \vec{y'} \in \mathbb{R}^2 \quad \vec{y} \prec \vec{y'} \quad \Leftrightarrow \quad y_1 \leq y'_1 \wedge y_2 \leq y'_2 \wedge \vec{y} \neq \vec{y'}$.

We also define the *dominance or equality* ($\preceq$) relation,

$\forall \vec{y}, \vec{y'} \in \mathbb{R}^2 \quad \vec{y} \preceq \vec{y'} \quad \Leftrightarrow \quad \vec{y} \prec \vec{y'} \vee \vec{y} = \vec{y'}$.

Given a set of vectors X, we define $nd(X)$, the set of non-dominated vectors in X as, $nd(X) = \{\vec{x} \in X \mid \nexists \vec{y} \in X \quad \vec{y} \prec \vec{x}\}$. Let $\mathcal{P}$ be the set of all paths in the graph G, with start state $s_{start} \in S$ and goal state $s_{goal} \in S$. Let $\mathcal{X}$ be the set of their costs.

The full bi-objective shortest path problem (G, s_{start}, s_{goal}) consist in finding all paths in $\mathcal{P}$ with costs in $nd(\mathcal{X})$. This paper deals with the *cost-unique bi-objective shortest path problem* [9], which consists in finding only one path in $\mathcal{P}$ for each cost in $nd(\mathcal{X})$.

Many multi-objective best-first search algorithms use the *min lexicographic order* $\prec_L$, since the lexicographic minimum in a set is also non-dominated. This total order is defined as follows,

$\forall \vec{y}, \vec{y'} \in \mathbb{R}^2 \quad \vec{y} \prec_L \vec{y'} \; \Leftrightarrow \; y_1 < y'_1 \; \vee (y_1 = y'_1 \wedge y_2 < y'_2)$

and the preference relation $\preceq_L$ has its natural meaning,

$\forall \vec{y}, \vec{y'} \in \mathbb{R}^2 \quad \vec{y} \preceq_L \vec{y'} \; \Leftrightarrow \; \vec{y} \prec_L \vec{y'} \vee \vec{y} = \vec{y'}$

A *heuristic function* $\vec{h}(s)$ is a function that for each state s returns an estimation of the cost of non-dominated paths from state n to the goal. We say that $\vec{h}(s)$ is monotone or consistent if, for all arcs (s, s') in the graph, the following condition holds,

$\vec{h}(s) \preceq \vec{c}(s, s') + \vec{h}(s')$.

* Financiado por Plan Propio de Investigación de la Universidad de Málaga (UMA), Campus de Excelencia Internacional Andalucía Tech. Work supported by the Spanish Ministry of Science and Innovation, European Regional Development Fund (FEDER), Junta de Andalucía, and Universidad de Málaga through the research projects with reference *IRIS* PID2021-122812OB-I00, PID2021-122381OB-I00 and UMA20-FEDERJA-065. Corresponding Author Email: lmandow@uma.es. This is the pre-review version of a paper accepted and presented at the 26th European Conference on Artificial Intelligence (ECAI 2023), published in the conference proceedings (DOI: 10.3233/FAIA230452).

A common choice for a consistent biobjective heuristic is $\vec{h}(s) = (h_1^*(s), h_2^*(s))$ where optimal costs $h_1^*(s), h_2^*(s)$ are precalculated using two single-objective Dijkstra searches (one for each objective) from the goal state over the graph with reversed arcs [18]. This has also been empirically analyzed by [10]. The computational cost of its precalculation is generally quite small compared to the ensuing bi-objective search. Alternative heuristics are discussed in [11] [4].

3 Antecedents

Most current unidirectional algorithms for multi-objective search are generalizations of the A* algorithm for single-objective shortest paths [7]. Succinctly, A* builds a search tree, with root at the start state s_{start}, that keeps the best known path to each visited state. When two different paths reaching the same state are found, only the best is kept and the other is discarded (*pruned*), breaking ties arbitrarily. Each path in the tree reaches a different state s with cost $g(s)$. These states are kept in a priority queue $Open$ in increasing order of the characteristic function $f(s) = g(s) + h(s)$. States in the tree that are not present in $Open$ are said to be 'closed'. The procedure iteratively removes and *expands* the first state in $Open$, i.e. it generates its successors and adds them to the tree and $Open$ when appropriate. When $h(s)$ satisfies the consistency property A* can be shown to be optimal in its class over the number of states expanded [3]. The procedure terminates when a goal state s_{goal} is selected from $Open$.

NAMOA* [12] is a multi-objective extension of A* that shares an analogous optimality property when heuristics are consistent. In other words, NAMOA* expands the optimal number of paths [13]. This is a landmark property regarding the efficiency of multi-objective search. However, those paths generated but not explored by the algorithm are pruned on the basis of dominance checks. Dominance is a computationally costly operation. Therefore, several subsequent algorithms have improved the runtime performance of NAMOA* focusing on efficient ways to perform the necessary dominance checks. This is also the focus of our discussion.

There are several important differences between single and multi-objective search that need to be tackled by any multi-objective best-first algorithm. Firstly, there are generally many non-dominated (optimal) paths reaching each state. Therefore, each relevant path is identified by a *label*, which combines the path's terminal state with its associated vector cost. A newly found path to a state s is locally pruned if its cost is dominated by that of some previously found path to s. Likewise, a new path can prune a previously known one if both reach the same state and the former dominates the latter. In the bi-objective case, each label has the form $(s, (g_1(s), g_2(s)))$, where $g_1(s)$ and $g_2(s)$ denote the two cost functions to be minimized.

Secondly, each path or label $(s, \vec{g})$ has an evaluation cost $\vec{f} = \vec{g} + \vec{h}(s)$. In the bi-objective case $\vec{f} = (f_1, f_2)$. Throughout this paper we will use 'extended' labels $(s, \vec{g}, \vec{f})$ when necessary for the sake of clarity. However, in practice the evaluation costs may be actually stored in the label or calculated on demand. Labels are kept in an $Open$ queue, and at each iteration a non-dominated (optimal) one according to its evaluation cost $\vec{f}$ is removed and expanded.

All algorithms described in the following sections implement $Open$ as a priority queue of labels with min lexicographic order (see section 2) of evaluation costs. The reason is that the lexicographic order is a total order relationship, and the lexicographic optimum in a set is guaranteed to be non-dominated. We assume the following priority queue operations:

- $empty(queue)$: returns $true$ if the $queue$ is empty, $false$ otherwise.
- $insert(l, queue)$: inserts label l in the $queue$.
- $top(queue)$: returns the min (top) label from the $queue$, and leaves the $queue$ unchanged.
- $pop(queue)$: removes and returns the min (top) label in $queue$.
- $update(l_1, l_2, queue)$: replaces label l_1 with l_2 in the $queue$, preserving $queue$ order.

Finally, the full multi-objective search problem aims to find the set of all non-dominated solution paths. Any path with heuristic cost dominated by that of a found solution can be discarded (this is a different kind of pruning sometimes referred to as 'filtering'). Search terminates when $Open$ is empty, i.e. when all labels have been either expanded, discarded (pruned), or identified as solutions. The recent BOA* algorithm solves the simpler cost-unique problem (see section 2). We assume the same problem definition in this paper. The extension to the more general full problem is straightforward. The next subsections review the improvements in dominance check efficiency over NAMOA* proposed by two recent bi-objective algorithms: NAMOA$_{dr}$* and BOA*.

3.1 NAMOA$^*_{dr}$

NAMOA$_{dr}$* (NAMOA* with dimensionality reduction) [14] is an efficient revision of NAMOA* that assumes lexicographic ordering and consistent heuristics. Figure 1 presents a pseudocode freely adapted to the cost-unique bi-objective case. Pruning operations are highlighted with comments.

The overall idea is similar to A*. An initial label l_{start} is inserted in the $Open$ priority queue. Then, labels are iteratively selected from $Open$ on a lexicographic best-first basis according to their evaluation cost (f_1, f_2). Goal labels are recorded, and non-goal ones are expanded[1].

We describe NAMOA$^*_{dr}$ highlighting its improvements against NAMOA*. In order to carry out local pruning operations, NAMOA* keeps two local sets G_{op} and G_{cl} associated to each state s. The first keeps the labels of open paths reaching s, while the second keeps the labels of closed ones. It also keeps a set of non-dominated solution labels $Sols$. Each time a new label is generated for s, its cost is checked against $G_{op}(s)$ and $G_{cl}(s)$ for local pruning, and its heuristic cost against $Sols$ for global pruning (filtering). If the label is not pruned, then any labels it may dominate are pruned from $G_{op}(s)$ [2]. Additionally, each time a new solution label is added to $Sols$, the heuristic costs of labels in $Open$ and the $G_{op}(s)$ sets are checked against it for filtering[3]. In summary, NAMOA* performs all these pruning operations *as soon as possible*. We collectively denote this default behavior in NAMOA* as 'eager pruning'.

To be more precise, we extend the terminology of [14] and distinguish four different kinds of pruning:

- cl-pruning: when the cost of a new label $(s, \vec{g})$ is dominated by some label in $G_{cl}(s)$.
- op-pruning: when the cost of a new label $(s, \vec{g})$ is dominated by some label in $G_{op}(s)$ or, conversely, some label in $G_{op}(s)$ is dominated by such new label.

[1] The pseudocode abstracts the details of successor label generation for brevity. Given a label $l_1 = (s, (g_1, g_2), (f_1, f_2))$ and a successor state s' of s with cost $\vec{c}(s, s') = (c_1, c_2)$, then $l_2 = (s', (g_1', g_2'), (f_1', f_2'))$ is a successor label with $g_i' = g_i + c_i$, and $f_i' = g_i' + h_i(s')$ for $i = 1, 2$.

[2] Due to the consistency of the heuristic, labels in $G_{cl}(s)$ are locally non-dominated, so there is no need to check them.

[3] Again, due to consistency, no closed label can be dominated by a newly found solution label.

```
Open ← empty queue; Sols ← $\emptyset$;
Set default value $\forall s \in S$ $g_2^{min}(s) \leftarrow \infty$;
Set default value $\forall s \in S$ $G_{op}(s) \leftarrow \emptyset$;
Let $l_{start}$ be $(s_{start}, (0,0), (h_1(s), h_2(s)))$;
parent($l_{start}$) ← $null$;
insert($l_{start}$, Open);
while $\neg empty(Open)$ do
    $l_1$ ← pop(Open);
    Let $l_1$ be $(s, (g_1, g_2)(f_1, f_2))$;
    Remove $l_1$ from $G_{op}(s)$ ;
    if $f_2 \geq g_2^{min}(s_{goal})$ then
        continue;                              // lazy filter
    end
    $g_2^{min}(s) \leftarrow g_2$;
    if $s = s_{goal}$ then
        add($l_1$, Sols);
        continue;
    end
    foreach new label $l_2$ successor of $l_1$ do
        Let $l_2$ be $(s', (g'_1, g'_2), (f'_1, f'_2))$;
        if $(g'_2 \geq g_2^{min}(s')) \vee (f'_2 \geq g_2^{min}(s_{goal}))$ then
            continue;                          // eager cl-prune/filter
        end
        if $G_{op}(s') \prec (g'_1, g'_2)$ then
            continue;                          // eager op-prune
        end
        Remove from $G_{op}(s')$ all vectors dominated by
          $(g'_1, g'_2)$;                      // eager op-prune
        parent($l_2$) ← $l_1$;
        insert($l_2$, Open);
        insert($l_2$, $G_{op}(s')$);
    end
end
```

Algorithm 1: NAMOA$^*_{dr}$ algorithm, freely adapted from [14] for cost-unique bi-objective search.

- old-filter: The evaluation cost $\vec{f}$ of newly generated labels $(s, \vec{g}, \vec{f})$ is checked against known (old) solution labels in $Sols$, and discarded if found to be dominated.
- new-filter: Newly found solution labels are checked against the heuristic cost of all open labels, and those found to be dominated are pruned.

The bi-objective search algorithms discussed in this paper differ mainly in the way they implement the dominance checks needed for these four pruning operations. Therefore, we highlight these differences.

NAMOA$^*_{dr}$ applies two efficient techniques to reduce the computational cost of dominance checks in cl-pruning and filtering. These exploit the following property. When a set of non-dominated bi-dimensional vectors $\{\vec{y}\}$ is min ordered lexicographically, then (i) the sequence of y_1 values is monotonically non-decreasing, and (ii) the sequence of y_2 values is monotonically non-increasing. If all vectors are different, then the sequences are (i') strictly increasing and (ii') strictly decreasing, respectively. This follows naturally from the definition of lexicographic ordering (see section 2).

Let us assume some dominated vectors are inserted in lexicographic order in an ordered non-dominated sequence. These can be easily identified scanning the sequence in min lexicographic order. Any vector that breaks the monotonically non-increasing sequence of y_2 values is dominated.

NAMOA$^*_{dr}$ exploits this property and keeps a scalar value $g_2^{min}(s)$ for each node, equal to the minimum second cost component of all closed labels to s (i.e. that of the one most recently selected). A new label $(s, (g_1, g_2))$ can be cl-pruned if $g_2 > g_2^{min}(s)$ avoiding the need for an explicit and computationally costly full dominance comparison against $G_{cl}(s)$. This technique is called 'dimensionality reduction', since bi-dimensional vector dominance checks are reduced to constant-time uni-dimensional scalar checks [14][4]. Notice that the cl-pruning operation is still 'eager', in the sense that it is carried out as soon as possible, only with a more efficient procedure.

NAMOA$^*_{dr}$ also reduces to a great extent the cost of both new and old filtering operations. Old-filter operations can also be carried out with a constant time scalar check between the evaluation cost f_2 of each new label and the minimum g_2 of labels in $Sols$. This is again an incarnation of the same dimensionality reduction idea, since solution labels are found by the algorithm following also a min lexicographic ordering.

New-filter operations can be particularly costly, since the $Open$ set can be large. NAMOA$^*_{dr}$ applies in this case a technique called 'lazy filtering'. When a new solution label is found, no particular operation is carried out (i.e. no eager filtering checks are applied). Current open labels wait their turn in the $Open$ queue, and are eventually checked for filtering only when they reach the top of the queue. At that point the filtering operation can be carried out again with a constant time scalar check between the heuristic cost f_2 of the selected label and the minimum g_2 of labels in $Sols$. The term 'lazy' means the pruning is not made as soon as possible, but rather delayed until label selection, when it can be carried out in a more efficient way.

Finally, NAMOA$^*_{dr}$ performs eager op-pruning operations using dominance against the G_{op} sets, just as NAMOA*.

These improvements made NAMOA$^*_{dr}$ an order of magnitude faster than NAMOA* on a benchmark of bi-objective road map problems [14].

[4] Analogously, for k-objective problems, k dimensional checks are reduced to $k-1$ dimensional ones.

*3.2 BOA**

BOA* (Bi-objective A*) [8] [9] is a recent extension of A* for bi-objective search problems. A pseudocode is presented in algorithm 2 with pruning operations highlighted by comments.

Like NAMOA$^*_{dr}$, BOA* incorporates dimensionality reduction for eager cl-pruning and lazy filtering. However, BOA* introduces several additional simplifications. Since the algorithm solves the cost-unique problem (i.e. only searches for a single path for each non-dominated cost), there is no need to explicitly keep the G_{cl} sets for each node[5]. The main contribution, from the point of view of our discussion on pruning operations, is an efficient implementation of op-pruning, that eliminates the need for the G_{op} sets as well.

The elimination of the G_{op} sets means newly generated labels to a known state s cannot be compared straightaway against other currently open labels to s. Instead, all such labels are inserted into $Open$. Only when labels reach the top of $Open$, they are compared against the g_2^{min} value of their state. This is a constant time operation between two scalar values. The result is a simpler and more efficient bi-objective search procedure.

In summary, BOA* incorporates dimensionality reduction in eager cl-pruning and lazy filtering, and extends these ideas to op-pruning. We call this operation 'lazy op-pruning'. These improvements showed BOA* to be around three times faster than NAMOA$^*_{dr}$ on a benchmark set of bi-objective road map problems [9].

```
Open ← empty queue; Sols ← $\emptyset$;
Set default value $\forall s \in S\ g_2^{min}(s) \leftarrow \infty$;
Let $l_{start}$ be $(s_{start}, (0,0), (h_1(s), h_2(s)))$;
parent($l_{start}$) ← $null$;
insert($l_{start}$, Open);
while $\neg empty(Open)$ do
    $l_1$ ← pop(Open);
    Let $l_1$ be $(s, (g_1, g_2)(f_1, f_2))$;
    if $(g_2 \geq g_2^{min}(s)) \vee (f_2 \geq g_2^{min}(s_{goal}))$ then
        continue;            // lazy op-prune/filter
    end
    $g_2^{min}(s) \leftarrow g_2$;
    if $s = s_{goal}$ then
        add($l_1$, Sols);
        continue;
    end
    foreach new label $l_2$ successor of $l_1$ do
        Let $l_2$ be $(s', (g'_1, g'_2), (f'_1, f'_2))$;
        if $(g'_2 \geq g_2^{min}(s')) \vee (f'_2 \geq g_2^{min}(s_{goal}))$ then
            continue;            // eager cl-prune/filter
        end
        parent($l_2$) ← $l_1$;
        insert($l_2$, Open);
    end
end
```

Algorithm 2: BOA* algorithm, adapted from [9]

4 Computational overhead of lazy pruning

In this section we provide some insights on how the costly dominance checks between vectors and sets of vectors are eventually replaced by (apparently) constant-time scalar comparison operations in bi-objective search algorithms. We discuss in turn the different kinds of pruning operations (cl-pruning, filtering, and op-pruning).

Let us first address the case of dimensionality reduction applied to cl-pruning. As explained above, the key of this technique is the fact that best-first algorithms already need to sort open labels in some way so that nondominated ones are always selected (best-first search). Sorting with a min lexicographic order results in a double advantage. Since the sequence of labels selected from $Open$ is lexicographically monotonically non-decreasing, when a label $(s, (g_1, g_2))$ is selected, its g_1 value is already known to be as large as all those previously selected for s (and present in $G_{cl}(s)$). All that remains to be checked for dominance is the scalar constant time comparison of g_2 and $g_2^{min}(s)$, regardless the size of $G_{cl}(s)$. Therefore, this technique is virtually computationally free. The computational cost of the lexicographic ordering is actually needed for the best-first operation of the algorithm, and dimensionality reduced cl-pruning just takes advantage of it. Practically all the comparisons between vector costs required for cl-pruning have already implicitly taken place in the queue ordering process. The cl-pruning operation is still eager (i.e. carried out as soon as possible) and vastly more efficient. A similar analysis applies to dimensionality reduced eager old-filtering.

Let us now analyze the case of *lazy new-filtering* used also in NAMOA$^*_{dr}$ and BOA*. Instead of performing an eager filtering operation each time a new solution is found (i.e. checking the cost of the new solution label against all open labels), no $Open$ labels are checked at the time. The final constant time checks are performed gradually as each of them is selected from the top of the queue.

Unlike dimensionality reduced cl-pruning, lazy new-filtering is not completely free from a computational point of view. Once a label $l = (s, (g_1, g_2), (f_1, f_2))$ is selected from $Open$, the check between f_2 and $g_2^{min}(s_{goal})$ can be performed in constant time. However, if label l could have been filtered eagerly by some solution label l^* found after l entered $Open$, then l has lingered longer in $Open$ than it could have. It could have been filtered earlier (eagerly), just when l^* was found. As a result, the size of $Open$ was larger than strictly necessary in the period between the discovery of l^* and the selection of l. As a side effect, this increases the average computational cost of insertion and deletion operations in $Open$ during that period.

Finally, the lazy op-pruning operation carried out by BOA* also incurs in a similar overhead for $Open$ queue operations. Explicit dominance checking operations are replaced by more efficient implicit ordering ones inside the $Open$ queue. However, labels that could have been eagerly op-pruned against their $G_{op}(s)$ sets populate the $Open$ queue until they are eventually selected and checked with a final constant time operation.

The identification of this computational overhead of lazy pruning techniques is important, since it opens up the possibility of further improvements in the runtime of bi-objective search algorithms.

In the next section we present a new bi-objective search algorithm. This algorithm applies an alternative dimensionality reduced pruning technique that lies in between eager and lazy pruning. We call this technique 'early pruning', since it generally does not prune as soon as possible, but does not wait as much as lazy pruning either. Experimental results in section 7 will show that this new technique produces a practical reduction in runtime.

5 Algorithm EBA*

This section introduces EBA* (Early pruning Bi-objective A*), a new bi-objective search algorithm. Our proposal relies on the same assumptions of previous algorithms (NAMOA$^*_{dr}$, BOA*), i.e. use of

[5] We also applied this simplification in our pseudocode of NAMOA$^*_{dr}$ in algorithm 1 adapted to the cost-unique problem.

lexicographic ordering, and a consistent heuristic function.

The pseudocode of EBA* is included in algorithm 3. Like previous algorithms, EBA builds a search tree with root at the start state s_{start}. Found solution labels are stored in a set $Sols$. Each node s keeps a variable $g_2^{min}(s)$ with the smallest value of g_2 among expanded labels to s, and in particular $g_2^{min}(s_{goal})$ keeps the smallest value of g_2 among solutions found. These are used for efficient checking in eager old-filtering and cl-pruning respectively (dimensionality reduction). One difference between EBA and previous algorithms is the management of the priority queues of labels:

- Each state s in the tree keeps a local priority queue of unexplored labels $G_{op}(s)$, according to a min lexicographic order of cost vectors (g_1, g_2)[6].
- A single $Open$ priority queue of labels is ordered according to a min lexicographic order of evaluation vectors (f_1, f_2). This queue contains only the top label of each non-empty $G_{op}(s)$ queue. The top label in $Open$ is trivially the best among all labels in $G_{op}(s)$ queues.

```
Open ← empty queue; Sols ← ∅;
Set default value $\forall s \in S\ g_2^{min}(s) \leftarrow \infty$;
Set default value $\forall s \in S\ G_{op}(s)$ ← empty queue;
Let $l_s$ be $(s_{start}, (0,0), (h_1(s_{start}), h_2(s_{start})))$;
parent($l_s$) ← $null$;
insert($l_s$, $Open$);
while $\neg empty(Open)$ do
    $l_1$ ← popReplace(Open);
    Let $l_1$ be $(s, (g_1, g_2)(f_1, f_2))$;
    if $f_2 > g_2^{min}(s_{goal})$ then
        continue ;                        // lazy filter
    end
    $g_2^{min}(s) \leftarrow g_2$;
    if $s = s_{goal}$ then
        add($l_1$, Sols);
        continue;
    end
    foreach label $l_2$ successor of $l_1$ do
        Let $l_2$ be $(s', (g'_1, g'_2), (f'_1, f'_2))$;
        if $(g'_2 \geq g_2^{min}(s')) \wedge (f'_2 \geq g_2^{min}(s_{goal}))$ then
            continue;           // eager cl-prune/filter
        end
        parent($l_2$) ← $l_1$;
        insertReplace($l_2$, $Open$);
    end
end
```

Algorithm 3: EBA* Algorithm

The algorithm creates the start label, inserts it in $Open$ and $G_{op}(s_{start})$, and creates the search tree. Until $Open$ becomes empty, the best open label l_1 is selected (popReplace). Then lazy filtering is checked and l_1 filtered if needed. Otherwise, the g_2^{min} value of the selected state is updated. If l_1 is a goal label, then it is added to $Sols$. Otherwise, the label is expanded. For each successor label l_2 the algorithm checks for dimensionality reduced eager cl-pruning and old-filtering and the label is discarded if needed. Otherwise, it is added to the search tree and the open queues (insertReplace).

[6] Evaluation vectors (f_1, f_2) can be equivalently used since all labels in $G_{op}(s)$ reach the same state s.

```
$l \leftarrow top(Open)$;
Let $l$ be $(s, (g_1, g_2), (f_1, f_2))$ ;
$newlabel \leftarrow false$;
while $\neg newlabel \wedge \neg empty(G_{op}(s))$ do
    Let $l'$ be $(s, (g'_1, g'_2)(f'_1, f'_2)) \leftarrow$ top($G_{op}(s)$ );
    if $(g'_2 \geq g_2^{min}(s)) \vee (f'_2 \geq g_2^{min}(s_{goal}))$ then
        pop($G_{op}(s)$ );
        // early op-prune, early-filter
    else
        update($l, l', Open$);
        $newlabel \leftarrow true$;
    end
end
if $\neg newlabel$ then
    pop(Open);                              // remove $l$
end
return $l$;
```

Algorithm 4: popReplace(Open) algorithm.

```
Let $l$ be $(s, (g_1, g_2), (f_1, f_2))$;
if $empty(G_{op}(s))$ then
    insert($l$, $Open$);
    insert($l$, $G_{op}(s)$ );
else
    $l' \leftarrow top(G_{op}(s))$;
    if $l$ lexicographically better than $l'$ then
        update($l', l, Open$);
    end
    insert($l$, $G_{op}(s)$ );
end
```

Algorithm 5: insertReplace(l,Open) algorithm.

The popReplace and insertReplace operations manage the $Open$ and local G_{op} queues. Both operations are detailed in algorithms 4 and 5 respectively.

Algorithm $popReplace$ returns the best label in $Open$. Before that, it checks if another open label of its state s can be promoted to the $Open$ queue. To this end, it iteratively pops labels from $G_{op}(s)$, applying dimensionality reduced early op-pruning and old-filtering, until a suitable label is found or $G_{op}(s)$ becomes empty. Notice that this pruning operation is not eager nor lazy, as it is carried out just before a label is transferred from $G_{op}(s)$ to $Open$. If a suitable label l' is found, it is used to update $Open$. Otherwise, $G_{op}(s)$ became empty and label l is definitely removed from $Open$. After this procedure, $Open$ still has the single best label for each state with non-empty G_{op} .

The use of local $G_{op}(s)$ queues reduces the overall $Open$ size, since there is at most one label in $Open$ for each state at any given time. Additionally, op-pruning operations are carried out on a local basis. This key process prevents many labels from entering $Open$ when compared to lazy op-pruning. Labels are 'early' pruned upon reaching the top of their local $G_{op}(s)$ queue, which is generally of a small size compared to $Open$. In consequence, the overhead in local queue operations due to early op-pruning is smaller compared to the overhead in $Open$ due to lazy op-pruning operations in $BOA*$.

The $popReplace$ operation also offers the chance to perform early filtering, discarding additional labels before they even enter the $Open$ queue. More precisely, this allows labels to be efficiently checked against solutions found between the moment the label was generated and the moment it reaches the top of its local $G_{op}(s)$ queue. This is a new source of efficiency over the lazy new filtering applied by both NAMOA$^*_{dr}$ and BOA*. Labels that can be efficiently early filtered in the local $G_{op}(s)$ queues, will never enter $Open$ nor produce overhead in its operations.

Finally, the $insertReplace$ operation adds newly generated labels to the open queues. Care is taken in case a new label beats the current best of its state. In such case it replaces the previous representative for that state in $Open$.

5.1 *Example*

Let us compare the workings of BOA* and EBA* through a simple example (figure 1), with start s_0 and goal s_4. For simplicity let's assume blind search, i.e. $\forall s,\ h_1(s) = h_2(s) = 0$. Table 1 shows the lexicographically ordered content of $Open$ for both algorithms at each iteration (only state and evaluation $\vec{f}$ for each label, since for all labels $\vec{f}(s) = \vec{g}(s)$). Each label expansion for a state s updates $g_2^{min}(s)$, but only $g_2^{min}(s_2)$ is mentioned in our discussion, since it is the only value actually used for pruning in this example.

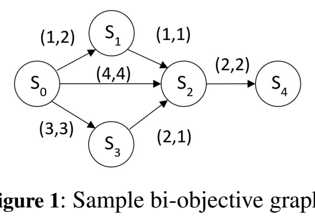

Figure 1: Sample bi-objective graph.

BOA* starts expanding label $[s_0, (0, 0)]$. At iteration 2 label $[s_1, (1, 2)]$ is expanded and successor $[s_2, (2, 3)]$ added to $Open$. Notice that label $[s_2, (4, 4)]$ is locally dominated, but no eager op-pruning is applied and both labels to s_2 coexist in $Open$. At iteration 3 $[s_2, (2, 3)]$ is selected, setting $g_2^{min}(s_2) = 3$. At iteration 4 $[s_3, (3, 3)]$ is expanded and its successor $[s_2, (5, 4)]$ eagerly cl-pruned. At iteration 5 label $[s_2, (4, 4)]$ is selected and lazy op-pruned using $g_2^{min}(s_2) = 3$. The only solution is found at iteration 6.

In EBA* all expansions result in 'popReplace', and successors are processed by 'insertReplace' using local queues. EBA* starts expanding label $[s_0, (0, 0)]$, and all three successors are added to $Open$ and their local queues $G_{op}(s_1), G_{op}(s_2), G_{op}(s_3)$ (not shown in Table 1). At iteration 1 EBA* expands $[s_1, (1, 2)]$. The local $G_{op}(s_1)$ queue becomes empty and 'insertReplace' is called over successor $[s_2, (2, 3)]$. This is added to the local $G_{op}(s_2)$ queue along with $[s_2(4, 4)]$. Since it is the best label, it becomes the only representative of s_2 in $Open$, replacing $[s_2(4, 4)]$. At iteration 3 $[s_2, (2, 3)]$ is selected, setting $g_2^{min}(s_2) = 3$, and 'popReplace' early op-prunes label $[s_2(4, 4)]$ from the local queue, which becomes empty. At iteration 4 $[s_3(3, 3)]$ is selected, and its successor $[s_2(5, 4)]$ is eagerly cl-pruned. Finally, the only solution is found at iteration 5.

This simple example illustrates how using local queues in EBA* allows for earlier op-pruning of label $[s_2(4, 4)]$ at iteration 3, effectively reducing the overall count of open labels when compared to BOA* in subsequent iterations. This in turn reduces the computational cost of any subsequent queue operations.

It.	BOA*	EBA*
1	$s(0,0)$	$s(0,0)$
2	$s_1(1,2), s_3(3,3), s_2(4,4)$	$s_1(1,2), s_3(3,3), s_2(4,4)$
3	$s_2(2,3), s_3(3,3), s_2(4,4)$	$s_2(2,3), s_3(3,3)$
4	$s_3(3,3), s_2(4,4), s_4(4,5)$	$s_3(3,3), s_4(4,5)$
5	$s_2(4,4), s_4(4,5)$	$s_4(4,5)$
6	$s_4(4,5)$	-

Table 1: $Open$ queue contents at each iteration for BOA* and EBA*.

6 Properties

We omit a full-fledged proof on the admissibility of EBA* due to space limitations. However, the admissibility of the new procedure can be easily sketched from previous results.

BOA* was shown to be admissible for the cost-unique bi-objective problem [9](theorem 1). EBA* makes the same use of dimensionality reduction in cl-pruning and lazy filtering as BOA*. The correctness of these operations was established by [14](theorem 1).

EBA* differs from BOA* in: (a) the structure of queues; (b) the early filtering operation; (c) the early op-pruning operation.

The management of the $Open$ and local G_{op} queues by $popReplace$ and $insertReplace$ guarantees that the best lexicographic open label of each state is present in $Open$. The transitivity of the lexicographic order guarantees then that the lexicographic optimal open label is always selected from $Open$. Therefore, selected and expanded labels from $Open$ have monotonically non-decreasing f_1 values, and those from the same state have monotonically increasing f_1 and monotonically decreasing f_2 values [9](lemmas 2-4).

Regarding early filtering, the monotonically decreasing sequence of $g_2^{min}(s_{goal})$ values guarantees that any label pruned by early filtering would be pruned by lazy filtering as well. The former operation only anticipates the result.

Regarding early op-pruning, all open labels for a given state s are always present at the G_{op} queue. The relative ordering of selection of local labels is unaffected by labels of other states. Therefore, the decreasing sequence of $g_2^{min}(s)$ values guarantees that both early and lazy op-pruning prune the same sets of label costs. Once again, the former operation only anticipates the result.

Therefore, given the equivalence of all pruning operations, EBA* is admissible for the cost-unique bi-objective problem under the same conditions of BOA*: non-negative arc costs and consistent heuristics.

7 Experiments

A recent contribution [9] showed BOA* to improve the runtime efficiency of NAMOA$^*_{dr}$. Therefore, we limit our experimental comparison to EBA* and BOA*.

We use a publicly available C implementation of BOA*[7]. An efficient binary heap is used for the $Open$ queue. We build our C implementation of EBA* sharing as much code as possible, and use the same kind of binary heaps for $Open$ and the $G_{op}(s)$ queues. We ran both algorithms over a set of publicly available test problems on bi-objective road maps used in [9]. These comprise eight sets of fifty problem instances, defined over different road maps. Due to space limitations we present results here for the four largest road maps (Table 2a), which include the hardest problem instances. The maps are available from the "9th DIMACS implementation challenge: shortest paths"[8]. The maps provide sets of states (locations) and arcs (roads) with cost information regarding distance (c_1) and travel time (c_2).

Experiments were run on an Intel (R) Core(TM) I7 10700K 3.8GHZ S1200 16Mb CPU with 64Gb DDR4 RAM under Ubuntu 22.04. Each process was run on a single thread.

Both algorithms perform the same number of label expansions and find the same number of solution labels over all problem instances. In other words, both explore the same portion of the state space, being the difference the efficiency of that exploration.

Table 2b presents results on the runtime of EBA* and BOA* over the instances of the different maps. Table 2c records some statistics on the number of basic heap percolation operations carried out by both algorithms. In the case of BOA* this includes percolations in $Open$, while for EBA* this includes percolations over all queues ($Open$ and the $G_{op}(s)$ queues). Table 2d presents some statistics regarding the size of the queues in both algorithms. We measure the average size of the $Open$ queue for each problem instance. The table shows the average of such values for all problem instances in each map. We also measure the maximum size of $Open$ for each instance, and report the maximum of such values for the instances of each map. We also report the maximum value of any G_{op} queue for each problem set.

8 Discussion

Runtime results for BOA* in Table 2b are consistent with those reported by [9], though our machine is somewhat slower. EBA* beats BOA* in all four sets of instances, both in average and maximum runtimes, and obtains a beneficial runtime ratio between 0.932 and 0.801 when compared to BOA*. Best performance is achieved over the hardest set (LKS). The ratio over all test sets was 0.81, i.e.a saving of 19% of the runtime taken by BOA*. Paired one-tailed Student's t-tests were carried out to validate the statistical significance of average runtime results. These provided p-values of $0.029, 9.1 \times 10^{-5}, 0.021$ and 2.9×10^{-5} for NW, NE, CAL and LKS respectively. These tests confirm the significance of improvement in average runtime for EBA* with confidence of at least $97,1\%$, and much higher for the more difficult LKS set.

The results in Table 2c provide some explanation for this better performance. The use of local $G_{op}(s)$ queues combined with early

Name	Region	States	Arcs	Avg. sols.
NW	Northwest USA	1207495	2840208	1051
NE	Northeast USA	1524453	3897636	1071
CAL	California and Nevada	1890815	4657742	907
LKS	Great Lakes	2758119	6885658	6057

(a) Road map sizes and average number of solutions in problem sets.

	Avg.	Max	Min	Med	σ
NW (Northwest USA)					
EBA*	3.79	46.35	0.34	0.67	8.55
BOA*	4.14	48.00	0.34	0.68	9.51
NE (Northeast USA)					
EBA*	8.32	47.11	0.44	2.71	11.43
BOA*	8.92	52.53	0.44	2.69	12.31
CAL (California and Nevada)					
EBA*	7.95	91.04	0.55	0.88	17.45
BOA*	8.73	106.67	0.55	0.93	19.80
LKS (Great Lakes)					
EBA*	213.46	1087.99	1.92	82.99	276.60
BOA*	266.61	1422.41	2.00	94.96	360.75

Runtime ratios EBA*/BOA*			
NW	NE	CAL	LKS
0.913	0.932	0.911	0.801

(b) Runtime statistics (in seconds) for each of the 50 instance benchmarks, and runtime ratios EBA*/BOA*.

	Avg.	Max	Min	Med	σ
NW (Northwest USA)					
EBA*	157.71	1509.23	0.02	24.10	336.89
BOA*	221.57	2023.89	0.02	32.62	469.39
NE (Northeast USA)					
EBA*	293.51	1458.35	$< 10^4$	118.52	378.41
BOA*	413.74	2091.72	$< 10^4$	172.60	528.94
CAL (California and Nevada)					
EBA*	269.62	2430.72	$< 10^4$	27.25	519.10
BOA*	378.14	3361.32	$< 10^4$	39.13	721.13
LKS (Great Lakes)					
EBA*	5557.78	24707.62	73.91	2417.70	6550.34
BOA*	7663.18	33563.67	103.58	3404.71	8959.35

# percolations ratios EBA*/BOA*			
NW	NE	CAL	LKS
0.712	0.709	0.713	0.725

(c) Heap percolation operations (in millions) for different test sets, and percolation ratios EBA*/BOA*.

	NW	NE	CAL	LKS
Average of $Open$ average sizes				
EBA*	5543.40	15529.39	11638.37	63859.85
BOA*	80949.22	268680.64	181907.00	2110226.94
Maximum of $Open$ maximum sizes				
EBA*	42705	79459	86191	234311
BOA*	1135990	1517523	2825448	12379212
Maximum of all G_{op} maximum sizes				
EBA*	1110	926	1256	5118

(d) Queue size statistics for different test sets

Table 2: Experimental data and results.

[7] https://github.com/jorgebaier/BOAstar/

[8] https://users.diag.uniroma1.it/challenge9/download.shtml

pruning in EBA* saves a sizeable amount of heap percolations, well over 8 billion in the hardest instance. Again, EBA* systematically beats BOA* in all road maps, with a substantial reduction both in average and maximum values. EBA* performs only 73.2% of the percolations carried out by BOA* over all test sets.

The results in Table 2d provide in turn some explanation for the reduced number of percolations in EBA*. The average size of $Open$ in EBA* is much smaller than in BOA* in all instance sets, and clearly an order of magnitude smaller in the hard LKS set. The maximum size of $Open$ is also at least an order of magnitude larger in BOA* when compared to EBA* in all sets. In contrast, the size of the G_{op} sets in EBA* is much smaller in size, reaching a global maximum of 5118 among all such sets in all problem instances. This means heap operations (pop, insert, and update) are carried out in EBA* in much smaller queues.

9 Conclusions and future work

This paper introduces EBA*, a new bi-objective shortest paths search algorithm with efficient pruning checks.

Dimensionality reduction is an efficient dominance checking technique that exploits the min lexicographic ordering of labels in multi-objective best-first search algorithms. Previous algorithms successfully applied this technique in a variety of eager and lazy pruning operation types.

We present an analysis highlighting the different types of pruning techniques applied by recent successful bi-objective algorithms. More precisely, we characterize pruning operations depending on their type (op-pruning, cl-pruning, old-filter, new-filter), the moment they are applied (eager, early, lazy), and their use of dimensionality reduction. Additionally, the computational overhead introduced by apparently constant-time lazy pruning techniques is identified. The new EBA* exact bi-objective algorithm is presented to reduce this overhead. EBA* incorporates previous efficient dominance check techniques, introducing early pruning, a new efficient technique that exploits the use of global and local G_{op} queues to reduce the computational overhead of lazy op-pruning and filtering. The consequence is a reduction in the number of open labels. This in turn results is a significant reduction in the computational cost of queue operations and algorithm runtime.

The admissibility of the new algorithm is discussed, given the equivalence in pruning power of early and lazy pruning, i.e. both techniques prune the same paths, though at different times during the algorithm execution. Experimental results show a consistent and significant reduction in runtime when compared to the previous BOA* algorithm over standard road map problem sets.

Future work includes deeper experimental analyses on different problem sets, the extension of early pruning to problems with more objectives, and the evaluation of EBA* in combination with bidirectional approaches for MO search. The recent work of [1] proposed a bidirectional bi-objective search framework in which two BOA* searches are run concurrently in opposite directions. This allows sharing of solutions between searches for efficient filtering. Heuristics can also be improved at runtime by exploiting information gained in the opposite search. An evaluation of EBA* in this framework is also a promising area of future research.

References

[1] Saman Ahmadi, Guido Tack, Daniel Harabor, and Philip Kilby, 'Bi-objective search with bi-directional A', in *29th Annual European Symposium on Algorithms, ESA 2021, September 6-8, 2021, Lisbon, Portugal (Virtual Conference)*, eds., Petra Mutzel, Rasmus Pagh, and Grzegorz Herman, volume 204 of *LIPIcs*, pp. 3:1–3:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, (2021).

[2] Jon Louis Bentley, Kenneth L. Clarkson, and David B. Levine, 'Fast linear expected-time algorithms for computing maxima and convex hulls', in *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms, 22-24 January 1990, San Francisco, California.*, pp. 179–187, (1990).

[3] Rina Dechter and Judea Pearl, 'Generalized best-first search strategies and the optimality of A*', *Journal of the ACM*, **32**(3), 505–536, (July 1985).

[4] F. Geißer, P. Haslum, S. Thiébaux, and F. W. Trevizan, 'Admissible heuristics for multi-objective planning', in *Proceedings of the Thirty-Second International Conference on Automated Planning and Scheduling, ICAPS 2022, Singapore (virtual), June 13-24, 2022*, eds., A. Kumar, S. Thiébaux, P. Varakantham, and W. Yeoh, pp. 100–109. AAAI Press, (2022).

[5] Boris Goldin and Oren Salzman, 'Approximate Bi-Criteria Search by Efficient Representation of Subsets of the Pareto-Optimal Frontier', in *Proceedings of the Thirty-First International Conference on Automated Planning and Scheduling (ICAPS 2021)*, pp. 149–158, (2021).

[6] P. Hansen, 'Bicriterion path problems', in *Lecture Notes in Economics and Mathematical Systems 177*, pp. 109–127. Springer, (1979).

[7] P.E. Hart, N.J. Nilsson, and B. Raphael, 'A formal basis for the heuristic determination of minimum cost paths', *IEEE Trans. Systems Science and Cybernetics SSC-4*, **2**, 100–107, (1968).

[8] C. Hernandez, W. Yeoh, J. Baier, H. Zhang, L. Suazo, and S. Koenig, 'A simple and fast bi-objective search algorithm', in *Proceedings of the 30th International Conference on Automated Planning and Scheduling (ICAPS 2020)*, pp. 143–151, (2020).

[9] Carlos Hernandez, William Yeoh, Jorge A. Baier, Han Zhang, Luis Suazo, Sven Koenig, and Oren Salzman, 'Simple and efficient bi-objective search algorithms via fast dominance checks', *Artificial Intelligence*, **314**, 103807, (2023).

[10] E. Machuca and L. Mandow, 'Multiobjective heuristic search in road maps', *Expert Syst. Appl.*, **39**(7), 6435–6445, (2012).

[11] E. Machuca and L. Mandow, 'Lower bound sets for biobjective shortest path problems', *J. Glob. Optim.*, **64**(1), 63–77, (2016).

[12] L. Mandow and J. L. Pérez de la Cruz, 'A new approach to multiobjective A* search', in *Proc. of the XIX Int. Joint Conf. on Artificial Intelligence (IJCAI'05)*, pp. 218–223, (2005).

[13] L. Mandow and J. L. Pérez de la Cruz, 'Multiobjective A* search with consistent heuristics', *Journal of the ACM*, **57**(5), 27:1–25, (2010).

[14] F. J. Pulido, L. Mandow, and J. L. Pérez de-la Cruz, 'Dimensionality reduction in multiobjective shortest path search', *Computers and Operations Research*, **64**, 60–70, (2015).

[15] Antonio Sedeño-Noda and Marcos Colebrook, 'A biobjective dijkstra algorithm', *European Journal of Operational Research*, **276**(1), 106–118, (2019).

[16] Antonio Sedeño-Noda and Andrea Raith, 'A dijkstra-like method computing all extreme supported non-dominated solutions of the biobjective shortest path problem', *Computers and Operations Research*, **57**, 83–94, (2015).

[17] B. S. Stewart and C. C. White, 'Multiobjective A*', *Journal of the ACM*, **38**(4), 775–814, (1991).

[18] Chi Tung Tung and Kim Lin Chew, 'A multicriteria Pareto-optimal path algorithm', *European Journal of Operational Research*, **62**, 203–209, (1992).

[19] Han Zhang, Oren Salzman, T. K. Satish Kumar, Ariel Felner, Carlos Hernandez, and Sven Koenig, 'A*pex: Efficient Approximate Multi-Objective Search on Graphs', in *Proceedings of the Thirty-Second International Conference on Automated Planning and Scheduling (ICAPS2022)*, pp. 394–403, (2022).

[20] Han Zhang, Oren Salzman, T. K. SatishKumar, Ariel Felner, Carlos Hernandez, and Sven Koenig, 'Anytime Approximate Bi-Objective Search', in *Proceedings of the Fifteenth International Symposium on Combinatorial Search (SoCS2022)*, pp. 199–207, (2022).