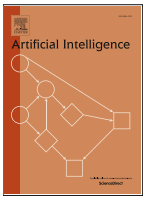
Artificial Intelligence 357 (2026) 104542

Contents lists available at ScienceDirect

Artificial Intelligence

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



Approximate multi-objective search

Han Zhang [a], Oren Salzman [b], T. K. Satish Kumar [a], Ariel Felner [c], Carlos Hernández Ulloa [d,*], Sven Koenig [e]

[a] University of Southern California, Los Angeles, USA
[b] Technion - Israel Institute of Technology, Haifa, Israel
[c] Ben-Gurion University, Beer Sheva, Israel
[d] Universidad San Sebastian, Santiago de Chile, Chile
[e] University of California, Irvine, USA

A B S T R A C T

In multi-objective search, we consider a graph whose edges are annotated with multiple cost components. A typical task is to compute the Pareto frontier, i.e., the set of all undominated paths from a given start state to a given goal state of the graph. However, the size of the Pareto frontier can be exponential in the size of the graph, and computing the entire Pareto frontier can be very time-consuming. Therefore, in this paper, we study how to find an approximate frontier for a user-provided approximation factor. Such a frontier can be significantly smaller than the Pareto frontier, enabling the design of efficient approximate multi-objective search algorithms. We present such an algorithm called A*pex, which uses an efficient, albeit approximate, representation of paths with similar costs to compute an approximate frontier. During the search, it avoids storing all paths explicitly, thereby reducing the search effort. We show that A*pex can be used as an algorithmic building block to solve additional problems by presenting (1) an anytime variant of A*pex, which computes an initial approximate frontier quickly and then works to find better approximate frontiers until eventually finding the entire Pareto frontier and (2) a variant of A*pex for approximately solving the Weight-Constrained Shortest Path (WCSP) problem, a problem that is closely related to multi-objective search. For evaluation, we use road networks from the 9th DIMACS Implementation Challenge: Shortest Path for evaluation. Our experimental results show that A*pex and its WCSP variant can outperform respective state-of-the-art algorithms by orders of magnitude in terms of runtime. We also show that the anytime variant of A*pex often computes better approximate frontiers than state-of-the-art algorithms, given a limited runtime.

1. Introduction and related work

In multi-objective search, we are given a graph (either implicitly or explicitly), a start state, and a goal state. Each edge in the graph is annotated with a cost vector consisting of multiple cost components. A solution π is a path from the start state to the goal state, and the cost of π, denoted as $c(\pi)$, is the component-wise sum of the costs of the edges traversed by π. Solution π *dominates* another solution π' if each cost component of $c(\pi)$ is no larger than the corresponding cost component of $c(\pi')$ and one of them is smaller. A typical task is to find the Pareto frontier, that is, the set of all solutions that are not dominated by any other solutions. This task generalizes the task of single-objective search to find all cost-minimal paths from the start state to the goal state. It is important for many real-world applications [1], including route planning for trucks, robots, and power lines [2] as well as inspecting regions of interest with robots [3,4]. For example, transporting hazardous material requires one to trade off between multiple costs for each street, such as its length and the number of residents that would be exposed to the hazardous material in case of a traffic accident [5].

* Corresponding author.
E-mail addresses: zhan645@usc.edu (H. Zhang), osalzman@cs.technion.ac.il (O. Salzman), tkskwork@gmail.com (T.K.S. Kumar), felner@bgu.ac.il (A. Felner), carlos.hernandez@uss.cl (C. Hernández Ulloa), svenk@uci.edu (S. Koenig).

https://doi.org/10.1016/j.artint.2026.104542
Received 18 December 2024; Received in revised form 16 April 2026; Accepted 18 April 2026
Available online 25 April 2026

However, the size of the Pareto frontier can be exponential in the size of the graph being searched [6,7], which makes computing the entire Pareto frontier often time-consuming. Researchers have therefore proposed to find an approximate frontier instead [7–11], that is, a set of solutions such that any solution in the Pareto frontier is ε-dominated by some solution in this set. A solution π ε-dominates another solution π' for some $\varepsilon \geq 0$ if each cost component of $\boldsymbol{c}(\pi)$ is no larger than $(1+\varepsilon)$ times the corresponding cost component of $\boldsymbol{c}(\pi')$, where ε is a user-provided approximation factor that can be different for different cost components. Different approximate frontiers (of different sizes) can exist for a given approximation factor. Still, their sizes can be much smaller than that of the Pareto frontier. This property can be exploited to create approximate multi-objective search algorithms that are efficient and produce small solution sets.

The first contribution of this paper is an approximate multi-objective search algorithm called A*pex. A*pex uses a novel data structure for representing paths with similar costs. It avoids storing all paths explicitly during the search. As multiple paths are grouped and represented by one single path, this results in an algorithm that is much more computationally efficient. A*pex builds on PP-A* [10], our previous algorithm, which uses a different data structure for representing paths and works only with two objectives. A*pex can work with any number of objectives and also improves the efficiency of PP-A*in bi-objective search. Our previous experimental results [12] have shown that A*pex also achieved average speed-ups of more than two times over PP-A*in bi-objective problem instances.

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

The second contribution of this paper is an anytime approximate multi-objective search algorithm called Anytime-A*pex (A-A*pex). From iteration to iteration, A-A*pex calls A*pex with a sequence of decreasing approximation factors to compute better and better approximate frontiers. Between iterations, A-A*pex can either reuse its previous search efforts or restart the search from scratch. We propose a technique to reuse the previous search efforts of A*pex by resuming the search from the paths that are pruned in the previous iteration but still have the potential to be extended to Pareto-optimal solutions. Additionally, we propose a hybrid variant of A-A*pex which first restarts the search for each iteration and starts reusing search efforts for later iterations. Existing work on anytime single-objective search has investigated reusing search effort [13] or restarting from scratch [14]. In this paper, we show how to reuse the search effort of A*pex despite its unique merge operations.

A problem closely related to multi-objective search is the Weight-Constrained Shortest-Path (WCSP) problem. In the WCSP problem, we consider graphs with two cost components, and the task is to compute a solution that minimizes the first cost component and whose second cost component is no larger than a given limit, called the *weight limit*. Several techniques of multi-objective search algorithms can be carried over to the WCSP domain. In fact, WC-A* [15], a state-of-the-art WCSP algorithm draws inspiration from multi-objective search algorithms. WC-A*and its bi-directional variant, WC-BA* [16], have been shown to outperform previous state-of-the-art algorithms, Bi-pulse [17] and RC-BDA* [18], by up to two orders of magnitude on road networks [15,16]. However, solving the WCSP problem optimally is NP-hard and hence can be time-consuming [19,20]. Many real-world situations admit—or even encourage—a trade-off between efficiency and a suboptimality bound for the path cost. A bounded suboptimal WCSP algorithm computes a $(1+\varepsilon)$-suboptimal path for a user-specified ε. A $(1+\varepsilon)$-suboptimal path has a cost within $(1+\varepsilon)$ times the minimum path cost and a weight that is no larger than the weight limit.

Our third contribution is a bounded suboptimal WCSP algorithm called WC-A*pex, which builds upon A*pex. There are only a few existing works on solving the WCSP problem bounded-suboptimally.

Cabrera et al. [17] suggested a straightforward method to convert an optimal WCSP algorithm to a bounded suboptimal one, that is, to terminate the search immediately after the cost of the incumbent solution (i.e., the best solution that an algorithm has found thus far) is proven to be within the given suboptimality bound [17]. Other works on bounded suboptimal WCSP algorithms [20,21] are typically based on fully polynomial-time approximation schemes (FPTAS), whose runtime is polynomial in the size of the graph and $1/\varepsilon$. Unfortunately, these algorithms are still impractical for large graphs, such as road networks, that often have millions of states.

In our experimental results, conducted on road networks from the 9th DIMACS Implementation Challenge: Shortest Path for evaluation, we first demonstrate the efficiency advantage of A*pex over an approximate baseline algorithm derived from LTMOA* [22], a state-of-the-art multi-objective search algorithm. A*pex obtains average speed-ups of more than an order of magnitude than the baseline algorithm in many scenarios. We then compare A-A*pex with LTMOA* because both algorithms eventually find the Pareto-optimal frontier. Although LTMOA* is more efficient than A-A*pex in terms of computing the entire Pareto-optimal frontier, A-A*pex often computes better approximate frontiers than LTMOA* when a limited runtime is given, which demonstrates the usefulness of A-A*pex in solving difficult multi-objective search problem instances. Finally, we compare WC-A*pexwith a bounded-suboptimal baseline algorithm derived from WC-A*using the same method provided by Cabrera et al. [17], called WC-A*-ε. Our experimental results show that WC-A*pexachieves up to an order-of-magnitude speed-up over WC-A*-ε.

Multi-objective search in the broader context of multi-objective optimization. In this paper we focus on the multi-objective search (MOS) problem which can be seen as a variant of the multi-objective optimization (MOO) problem (see, e.g., [23–28]). It is important to highlight the similarities and differences between the two fields.

MOO and MOS share a foundational connection, both dealing with problems that involve optimizing multiple, often conflicting, objectives. Thus, both are interested in computing the Pareto-optimal frontier of solutions (or an approximation or a subset of it). However, in MOO, solutions can be seen as points in some cost space that satisfy given constraints, while in MOS, solutions are paths. Put simply, in MOO we want a set of values to a set of variables that obey some constraints. In MOS we want full paths from a start to a goal.

Thus, in contrast to MOS, which builds upon search algorithms such as A*, MOO builds upon local and global optimization methods such as genetic algorithms [29–31], particle swarm optimization [32], and simulated annealing [33].

It is noteworthy that some of the problems we consider in our work have similar, though not identical, analogous problems in MOO that may be confused. Specifically, we use the acronym WCSP to refer to the Weight-Constrained Shortest Path, while in the MOO community this can refer to the Multi-Objective Weighted Constraint Satisfaction Problem (a generalization of a constraint satisfaction problem). For more details on this problem and solution approaches, see e.g., [34] and references within.

Paper organization. We start by introducing the necessary terminology for the paper as well as formally defining the problems we will address in this paper (Section 2). We then continue by providing the necessary algorithmic background, which will be used in our proposed algorithms (Section 3). Moving to the key contributions of the paper, we start by presenting A*pex, our best-first multi-objective search algorithm for approximating the Pareto frontier for any user-provided approximation factor (Section 4). We then move on to describe A-A*pex, an anytime approximate multi-objective search algorithm that builds upon A*pex (Section 5). We finish the algorithmic contributions to propose WC-A*pex, which generalizes A*pex to compute bounded-suboptimal solutions for WCSP problem instances (Section 6). We conclude the paper by presenting experimental results for A*pex, A-A*pex, and WC-A*pex(Section 7) and some concluding remarks (Section 8).

2. Terminology and problem definitions

Boldface font denotes vectors or vector functions. v_i denotes the ith component of vector or vector function $\boldsymbol{v}$. The sum of two vectors $\boldsymbol{v}$ and $\boldsymbol{v'}$ of the same length N is defined as $\boldsymbol{v}+\boldsymbol{v'} = [v_1+v'_1, v_2+v'_2 \dots v_N+v'_N]$. $\boldsymbol{v} \preceq \boldsymbol{v'}$ denotes that $v_i \le v'_i$ for all $i = 1,2 \dots N$. In this case, we say that $\boldsymbol{v}$ *weakly dominates* $\boldsymbol{v'}$. $\boldsymbol{v} \prec \boldsymbol{v'}$ denotes that $\boldsymbol{v} \preceq \boldsymbol{v'}$ and there exists an $i \in \{1,2 \dots N\}$ with $v_i < v'_i$. In this case, we say that $\boldsymbol{v}$ *dominates* $\boldsymbol{v'}$. $\boldsymbol{v} \preceq_\varepsilon \boldsymbol{v'}$ for an *approximation factor* (or, more precisely, vector of approximation factors) $\boldsymbol{\varepsilon} = [\varepsilon_1, \varepsilon_2 \dots \varepsilon_N] \in \mathbb{R}^N_{\ge 0}$ denotes that $v_i \le (1+\varepsilon_i) \cdot v'_i$ for all $i = 1,2 \dots N$. In this case, we say that $\boldsymbol{v}$ $\boldsymbol{\varepsilon}$*-dominates* $\boldsymbol{v'}$. The *truncate function* Tr takes a vector as input and outputs the vector with its first component deleted.

A *graph* is a tuple $G = \langle S, E, \boldsymbol{c}\rangle$, where S is a finite set of *states* and $E \subseteq S \times S$ is a finite set of *edges*. $\texttt{succ}(s) = \{s' \in S : \langle s, s'\rangle \in E\}$ denotes the successors of state s. Cost function $\boldsymbol{c} : E \to \mathbb{R}^N_{>0}$ maps an edge to its cost, which is a vector of its N cost components. The graph is called a *weighted graph* or *single-objective* when $N = 1$, *multi-objective* when $N \ge 2$ and also *bi-objective* for the specific case that $N = 2$.

A *path* from state s_1 to state s_ℓ is a sequence of states $\pi = [s_1, s_2 \dots s_\ell]$ with $\langle s_j, s_{j+1}\rangle \in E$ for all $j = 1,2 \dots \ell-1$. $s_1 = s_{\text{start}}$ (to be defined shortly) unless mentioned otherwise. We use $s(\pi)$ to denote the last state of π, that is, s_ℓ. $\boldsymbol{c}(\pi) = \sum_{j=1}^{\ell-1} \boldsymbol{c}(\langle s_j, s_{j+1}\rangle)$ denotes the cost of path π. It can be *extended* with an edge $\langle s_\ell, s_{\ell+1}\rangle$ to obtain path $[s_1, s_2 \dots s_\ell, s_{\ell+1}]$. Path π *dominates (resp. weakly dominates and* ε*-dominates)* a path π' iff $\boldsymbol{c}(\pi) \prec \boldsymbol{c}(\pi')$ (resp. $\boldsymbol{c}(\pi) \preceq \boldsymbol{c}(\pi')$ and $\boldsymbol{c}(\pi) \preceq_\varepsilon \boldsymbol{c}(\pi')$).

A *multi-objective search problem instance* is specified by a tuple $P = \langle G, s_{\text{start}}, s_{\text{goal}}\rangle$, where G is a multi-objective graph, $s_{\text{start}} \in S$ is the start state, and $s_{\text{goal}} \in S$ is the goal state. The instance is called bi-objective in case $N = 2$. A *solution* is a path from s_{start} to s_{goal}. A *Pareto-optimal* solution is a solution that is not dominated by any other solution. The *Pareto frontier* is the set of all Pareto-optimal solutions. An $\boldsymbol{\varepsilon}$*-approximate frontier* is a set of solutions such that, for any Pareto-optimal solution π', there exists a solution π in the $\boldsymbol{\varepsilon}$-approximate frontier with $\pi \preceq_\varepsilon \pi'$. The Pareto frontier is a $\mathbf{0}$-approximate frontier but not necessarily vice versa. For example, the set of all solutions is also a $\mathbf{0}$-approximate frontier. A multi-objective search algorithm that finds an $\boldsymbol{\varepsilon}$-approximate frontier is also called an ($\boldsymbol{\varepsilon}$-)*approximate multi-objective search algorithm*.

A *WCSP problem instance* is specified by a tuple $P = \langle G, s_{\text{start}}, s_{\text{goal}}, W\rangle$, where G is a graph with two cost components, s_{start} is the start state, s_{goal} is the goal state, and $W \in \mathbb{R}_{>0}$ is the *weight limit*. A *(WCSP) solution* π is a path from s_{start} to s_{goal} with $c_2(\pi) \le W$. We say that P is *feasible* if a solution exists. An *optimal solution* is a solution with the minimum c_1-value, denoted as c_1^*, of all solutions. Given a non-negative ε, a solution π is $(1+\varepsilon)$*-suboptimal* if $c_1(\pi) \le (1+\varepsilon) \cdot c_1^*$. A WCSP algorithm that finds a $(1+\varepsilon)$-suboptimal solution is a *bounded-suboptimal WCSP algorithm*.

A heuristic function $\boldsymbol{h} : S \to \mathbb{R}^N_{\ge 0}$ provides a lower bound on the cost of any path from any given state s to the goal state s_{goal}. We assume that the provided heuristic function $\boldsymbol{h}$ is consistent, that is, $\boldsymbol{h}(s_{\text{goal}}) = \mathbf{0}$ and $\boldsymbol{h}(s) \preceq \boldsymbol{c}(\langle s, s'\rangle) + \boldsymbol{h}(s')$ for all $\langle s, s'\rangle \in E$. It is a common practice in existing literature [12,16,35,36] to use Dijkstra's algorithm (starting from s_{goal}) to compute the minimum cost $c_i^*(s)$ from any state s to s_{goal} for the ith objective (while ignoring other objectives), $i = 1,2 \dots N$, and $\boldsymbol{h}(s) := [c_1^*(s), c_2^*(s) \dots c_N^*(s)]$ as the heuristic function. We call these heuristic functions *perfect-distance heuristics*. Importantly, computing the perfect-distance heuristics requires running N instances of Dijkstra's algorithm, one for each of the N objectives.

3. Algorithmic background

In this section, we first review the best-first multi-objective search framework because almost all state-of-the-art multi-objective search algorithms, including our new ones, are based on it. Specifically, we describe LTMOA* and WC-A*in detail. For additional background, see Salzman et al. [37].

3.1. The best-first multi-objective search framework

In a best-first multi-objective search algorithm, each search node n contains a state $s(n)$ corresponding to a path from s_{start} to state $s(n)$. Each node stores a $\boldsymbol{g}$-value $\boldsymbol{g}(n)$ and its $\boldsymbol{f}$-value is defined as $\boldsymbol{f}(n) = \boldsymbol{g}(n) + \boldsymbol{h}(s(n))$. A best-first multi-objective search algorithm is similar to best-first single-objective search algorithms, such as A*, but, most differently, it needs to consider multiple nodes (with $\boldsymbol{g}$-values that do not dominate each other) for the same state. It maintains a priority queue OPEN, which contains the generated but not yet expanded nodes. OPEN is initialized with a node that contains the start state s_{start} and the $\boldsymbol{g}$-value $\mathbf{0}$. At each iteration, the algorithm *extracts* a node from OPEN with undominated $\boldsymbol{f}$-value of all nodes in OPEN. The algorithm performs *dominance checks* for the extracted node to determine whether this node or any of its descendants have the potential to be in the solution set. If not, the node is discarded. If the node contains the goal state, the algorithm adds the node to the solution set. Otherwise, it expands the node by generating a new node for each of the successors of the state contained in the node. The algorithm performs dominance checks for each generated node to determine whether the generated node or any of its descendants have the potential to be in the solution set. If not, it discards the generated node. Otherwise, it adds the generated node to OPEN. When OPEN becomes empty, the algorithm terminates and returns the solution set.

Examples of algorithms that conform to the best-first multi-objective search framework include NAMOA*dr [38], BOA* [39], EMOA* [40], LTMOA* [22], and PP-A* [10]. They only differ in which node is extracted from OPEN, which information is contained in the nodes, and how the dominance checks are interleaved with the search and implemented.

3.2. LTMOA*

Algorithm 1: LTMOA*.

Input : A problem instance $\langle G, s_{\text{start}}, s_{\text{goal}} \rangle$ and a consistent heuristic function $\mathbf{h}$
Output: A Pareto frontier
$n \leftarrow$ new node with $s(n) = s_{\text{start}}$ and $\mathbf{g}(n) = \mathbf{0}$
OPEN $\leftarrow \{n\}$
SOLS $\leftarrow \emptyset$
foreach $s \in S$ **do**
  $\boldsymbol{G}^{\text{tr}}_{\text{cl}}(s) \leftarrow \emptyset$
while *OPEN* $\neq \emptyset$ **do**
  extract a node n from OPEN with the lexicographically smallest $\boldsymbol{f}$-value
  if $isDominated(n)$ **then**
    continue
  update$(\boldsymbol{G}^{\text{tr}}_{\text{cl}}(s(n)), Tr(\mathbf{g}(n)))$
  if $s(n) = s_{\text{goal}}$ **then**
    add the corresponding solution of n to SOLS
    continue
  for each $s' \in succ(s(n))$ **do**
    $n' \leftarrow$ new node with $s(n') = s'$ and $\mathbf{g}(n') = \mathbf{g}(n) + \mathbf{c}(\langle s, s' \rangle)$
    if $isDominated(n')$ **then**
      continue
    add n' to OPEN
return SOLS
Function $isDominated(n)$**:**
  if $\exists \mathbf{x} \in \boldsymbol{G}^{\text{tr}}_{\text{cl}}(s_{\text{goal}}) : \mathbf{x} \preceq Tr(\mathbf{f}(n))$ **then**
    return ***true***
  if $\exists \mathbf{x} \in \boldsymbol{G}^{\text{tr}}_{\text{cl}}(s(n)) : \mathbf{x} \preceq Tr(\mathbf{g}(n))$ **then**
    return ***true***
  return ***false***
Function $update(\mathbf{X}, \mathbf{y})$**:**
  remove the vectors weakly dominated by $\mathbf{y}$ from $\mathbf{X}$
  add $\mathbf{y}$ to $\mathbf{X}$

Algorithm 1 shows the pseudo-code for LTMOA*. Following the general framework outlined in Section 3.1, in LTMOA*, a node n corresponds to a path from s_{start} to state $s(n)$ with cost $\boldsymbol{g}(n)$. The $\boldsymbol{f}$-value of n is defined as $\boldsymbol{f}(n) = \boldsymbol{g}(n) + \boldsymbol{h}(s(n))$. Additionally, LTMOA* maintains the parent node $p(n)$ for n, and hence the corresponding path of n can be constructed in reverse by following the parent nodes from $s(n)$ to s_{start}. LTMOA* initializes OPEN with a node on state s_{start}, which corresponds to path $[s_{\text{start}}]$ and whose $\boldsymbol{g}$-value is

0. In each iteration, LTMOA* *extracts* a node n from OPEN with the *lexicographically smallest* $\boldsymbol{f}$-value. LTMOA* performs dominance checks after extracting (that is, after Line 7) and before generating (that is, before Line 18) a node n. LTMOA* discards a node n if one of the following conditions hold:

- $\mathbf{C1_{LTMOA*}}$ there exists an expanded node on state s_{goal} (which corresponds to a solution in SOLS) whose $\boldsymbol{g}$-value weakly dominates $\boldsymbol{f}(n)$.
- $\mathbf{C2_{LTMOA*}}$ there exists an expanded node on state $s(n)$ whose $\boldsymbol{g}$-value weakly dominates $\boldsymbol{g}(n)$.

Because LTMOA* extracts a node from OPEN with the lexicographically smallest $\boldsymbol{f}$-value of all nodes in OPEN and also because the heuristic is consistent, LTMOA* satisfies that the f_1-values of extract nodes are monotonically non-decreasing. Therefore, LTMOA* does not need to check g_1 or f_1- values for the dominance checks. Instead of maintaining the set of all $\boldsymbol{g}$-values, it maintains only the often significantly smaller set $\boldsymbol{G}^{\text{tr}}_{\text{cl}}(s)$ of undominated truncated $\boldsymbol{g}$-values for each state s (which is updated on Line 10). This technique is called *dimensionality reduction* [38] and has been used in other multi-objective search algorithms.

Let n be a node extracted by LTMOA* and not discarded after the dominance checks. If $s(n) = s_{\text{goal}}$, LTMOA* then adds the corresponding path (which is a solution) of n to SOLS. Otherwise, LTMOA* *expands* n by generating a new child node n' for each successor of $s(n)$. When OPEN becomes empty, the algorithm terminates and returns SOLS.

Perny et al. [11] suggests computing an $\boldsymbol{\epsilon}$-approximate frontier by relaxing Condition $\mathbf{C1_{LTMOA*}}$ of the dominance checks and discarding a node if its $\boldsymbol{f}$-value is $\boldsymbol{\epsilon}$-dominated by the $\boldsymbol{g}$-value of an expanded node on state s_{goal}. We adopt this approach to obtain a baseline algorithm LTMOA*-$\boldsymbol{\epsilon}$. In LTMOA*-$\boldsymbol{\epsilon}$, we change Line 21 in Algorithm 1 to test if $\mathbf{x} \preceq_{\epsilon} Tr(\mathbf{f}(n))$.

3.3. WC-A*

WC-A*computes an optimal solution for an input WCSP instance. In WC-A*, each node n is a path from s_{start} to $s(n)$ with cost $\boldsymbol{g}(n)$. It only maintains at most one *incumbent solution*, that is, the best solution that has been found, in the solution set. Recall that, in WCSP, a solution is a path from s_{start} to s_{goal} whose c_2-value is not larger than W. WC-A*also uses the dimensionality reduction technique. Because the graph in a WCSP instance has only two cost components, a truncated $\boldsymbol{g}$-value is equivalent to a scalar number, namely the g_2-value. The set of undominated truncated $\boldsymbol{g}$-values for a state s is equivalent to the smallest g_2-value of all expanded nodes on state s, and dominance checks can be done in constant time. WC-A*also discards a node n if its g_2-value is not smaller than the smallest g_2-value of all expanded nodes on state $s(n)$, which is equivalent to Condition $\mathbf{C1_{LTMOA*}}$. Additionally it discards a node if its f_2-value is larger than the weight limit W or its f_1-value is not smaller than the c_1-value of the incumbent solution. Since WC-A*extracts nodes with monotonically non-decreasing f_1-values, it terminates (and returns the incumbent solution) once the minimum f_1-value in OPEN is not smaller than the one of the incumbent solution.

The heuristic computation with Dijkstra's algorithm can also obtain the minimum-c_1 and minimum-c_2 paths from any state s to s_{goal}. We call these paths the *complementary paths* of s. When generating a node n, WC-A*tries to update the incumbent solution with better ones by connecting the corresponding path of n with the complementary paths of $s(n)$.

WC-A*can be converted to a bounded suboptimal WCSP algorithm by terminating the algorithm when the minimum f_1-value in OPEN is no smaller than the f_1-value of the incumbent solution divided by $(1+\epsilon)$. We denote this variant of WC-A*as WC-A*-ϵ.

It is noteworthy that Ahmadi et al. [16] propose WC-BA*, a bi-directional variant of WC-A*that runs two WC-A*searches (one starting from s_{start} and the other starting from s_{goal}) concurrently. We omit empirical comparisons with WC-BA*in this paper because [15] has shown that WC-BA*does not dominate WC-A*in their experimental results and, in fact, has larger average runtime in several scenarios.

4. A*pex

In this section, we describe A*pex, a best-first multi-objective search algorithm that finds an $\boldsymbol{\epsilon}$-approximate frontier for a user-provided approximation factor ϵ. While algorithms such as LTMOA* reason about single paths, A*pex reasons about sets of paths with the same last state and similar costs as $\boldsymbol{\epsilon}$-bounded apex-path pairs, which results in small numbers of apex-path pair expansions and thus small runtimes.

Each node of A*pex is a so-called *apex-path pair* $\mathcal{AP} = \langle \boldsymbol{A}, \pi \rangle$, where $\boldsymbol{A}$ is an N-dimensional vector called the node's apex and the path π is called the node's *representative path*. For any apex-path pair $\mathcal{AP} = \langle \boldsymbol{A}, \pi \rangle$, the apex will dominate the cost of the representative path. Namely, $\boldsymbol{A} \preceq \mathbf{c}(\pi)$. The state $s(\mathcal{AP})$ of $\mathcal{AP}$ will be defined as $s(\mathcal{AP}) := s(\pi)$ (recall that $s(\pi)$ is the last state of path π). The $\mathbf{g}$-value and $\mathbf{f}$-value of $\mathcal{AP}$ are defined as $\mathbf{g}(\mathcal{AP}) := \boldsymbol{A}$ and $\mathbf{g}(\mathcal{AP}) := \boldsymbol{A} + \mathbf{h}(s(\mathcal{AP}))$. Finally, we say that $\mathcal{AP}$ is ϵ-bounded iff $\mathbf{f}(\pi) \preceq_{\epsilon} \mathbf{f}(\mathcal{AP})$ (recall that $\mathbf{f}(\pi) := \mathbf{c}(\pi) + \mathbf{h}(s(\pi))$). As we will see, all nodes in A*pex will be $\boldsymbol{\epsilon}$-bounded.

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

Property 1. *Consider an apex-path pair $\mathcal{AP} = \langle \boldsymbol{A}, \pi \rangle$ and the set of paths $\Pi_{\mathcal{AP}}$ that $\mathcal{AP}$ corresponds to. If $\mathcal{AP}$ is ϵ-bounded, every solution that extends some path $\pi' \in \Pi_{\mathcal{AP}}$ is ϵ-dominated by some solution that extends π.*

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

$$\begin{aligned}\mathbf{f}(\pi) &\preceq_\epsilon \mathbf{f}(\mathcal{AP}) \\ &= \boldsymbol{A} + \mathbf{h}(s(\mathcal{AP})).\end{aligned} \tag{1}$$

Because $\boldsymbol{A}$ weakly dominates the costs of all paths in $\Pi_{\mathcal{AP}}$ (including π'), we have $\boldsymbol{A} \preceq \mathbf{c}(\pi')$. Combining this and Eq. (1), we have

$$\begin{aligned}\mathbf{f}(\pi) &\preceq_\epsilon \mathbf{c}(\pi') + \mathbf{h}(s(\mathcal{AP})) \\ &= \mathbf{f}(\pi').\end{aligned} \tag{2}$$

Consider any solution π'_{sol} that extends π'. Because $\mathbf{h}$ is required to be consistent, we have $\mathbf{f}(\pi') \preceq \mathbf{c}(\pi'_{\text{sol}})$. Let $\boldsymbol{\delta}$ denote $\mathbf{c}(\pi'_{\text{sol}}) - \mathbf{f}(\pi')$, which is a non-negative vector. The cost of π'_{sol} can then be represented as $\mathbf{c}(\pi'_{\text{sol}}) = \mathbf{f}(\pi') + \boldsymbol{\delta}$. Consider solution π_{sol} which extends π with the same path, denoted as π_{ext}, that extends π' to π'_{sol}. The cost of π'_{ext} can be represented as $\mathbf{c}(\pi_{\text{ext}}) = \mathbf{c}(\pi'_{\text{sol}}) - \mathbf{c}(\pi') = \boldsymbol{\delta} + \mathbf{h}(s(\mathcal{AP}))$. The cost of π'_{sol} can then be represented as $\mathbf{c}(\pi_{\text{sol}}) = \mathbf{c}(\pi) + \mathbf{c}(\pi_{\text{ext}}) = \mathbf{f}(\pi) + \boldsymbol{\delta}$. Because $\boldsymbol{\delta} \preceq_\epsilon \boldsymbol{\delta}$ (which holds because $\boldsymbol{\delta}$ is a non-negative vector) and also because $\mathbf{f}(\pi) \preceq_\epsilon \mathbf{f}(\pi')$ (Eq. 2), we have $\mathbf{f}(\pi) + \boldsymbol{\delta} \preceq_\epsilon \mathbf{f}(\pi') + \boldsymbol{\delta}$, which is equivalent to $\mathbf{c}(\pi_{\text{sol}}) \preceq_\epsilon \mathbf{c}(\pi'_{\text{sol}})$. □

Algorithm 2: The extend and merge operations on apex-path pairs.

1 **Function** *extend*$(\mathcal{AP} = \langle \boldsymbol{A}, \pi \rangle, e)$**:**
2 $\quad \pi' \leftarrow$ path π extended by edge e
3 $\quad$ **return** $\langle \boldsymbol{A} + \boldsymbol{c}(e), \pi' \rangle$
4 **Function** *merge*$(\mathcal{AP} = \langle \boldsymbol{A}, \pi \rangle, \mathcal{AP}' = \langle \boldsymbol{A}', \pi' \rangle)$**:**
5 $\quad \boldsymbol{A}_{\text{new}} \leftarrow [\min(A_1, A'_1), \min(A_2, A'_2), \dots, \min(A_N, A'_N)]$
6 $\quad \pi_{\text{new}} \leftarrow$ choose a path between π and π'
7 $\quad$ **return** $\langle \boldsymbol{A}_{\text{new}}, \pi_{\text{new}} \rangle$

Before we explain how apex-path pairs are used, let us define the operations on apex-path pairs: The first operation we consider is *extending* an apex-path pair $\mathcal{AP} = \langle \boldsymbol{A}, \pi \rangle$ by an edge e (Lines 1-3 of Algorithm 2). Let $\mathcal{AP}'$ denote the resulting apex-path pair. The apex of $\mathcal{AP}'$ is the sum of $\boldsymbol{A}$ and $\mathbf{c}(e)$, and the representative path of $\mathcal{AP}'$ is path π extended by edge e. Conceptually, apex-path pair $\mathcal{AP}'$ corresponds to the set of paths $\Pi_{\mathcal{AP}'}$ that extends every path in $\Pi_{\mathcal{AP}}$ by e. It is easy to verify that the apex of $\mathcal{AP}'$ is the component-wise minimum of the costs of the paths in $\Pi_{\mathcal{AP}'}$. The following property shows that the extend operation preserves the ϵ-boundedness.

Property 2. *Consider the resulting apex-path pair $\mathcal{AP}' = \langle \boldsymbol{A}', \pi' \rangle$ of extending apex-path pair $\mathcal{AP} = \langle \boldsymbol{A}, \pi \rangle$ with an edge e. $\mathcal{AP}'$ is ϵ-bounded if $\mathcal{AP}$ is ϵ-bounded.*

Proof. Let s and s' denote the states that $\mathcal{AP}$ and $\mathcal{AP}'$ contain, respectively. Assume that $\mathcal{AP}$ is ϵ-bounded, that is, it satisfies $\mathbf{c}(\pi) + \mathbf{h}(s) \preceq_\epsilon \boldsymbol{A} + \mathbf{h}(s)$, and hence,

$$c_i(\pi) + h_i(s) \le (1 + \epsilon_i) \cdot (A_i + h_i(s)), i = 1, 2 \dots N. \tag{3}$$

Because the heuristic function $\mathbf{h}$ is consistent, we have $h_i(s) \le c_i(e) + h_i(s')$ for all i. Set $\delta := c_i(e) + h_i(s') - h_i(s)$ and note that $\delta \ge 0$ and hence $\delta \le (1 + \epsilon_i) \cdot \delta$. By adding the latter inequality to Eq. (3), we have $c_i(\pi) + c_i(e) + h_i(s') \le (1 + \epsilon_i) \cdot (A_i + c_i(e) + h_i(s'))$ for all i, that is, $\mathbf{c}(\pi) + \mathbf{c}(e) + \mathbf{h}(s') \preceq_\epsilon \boldsymbol{A} + \mathbf{c}(e) + \mathbf{h}(s')$. Because $\mathbf{c}(\pi') = \mathbf{c}(\pi) + \mathbf{c}(e)$ and $\boldsymbol{A}' = \boldsymbol{A} + \mathbf{c}(e)$, we have $\mathbf{f}(\pi') \preceq_\epsilon \mathbf{f}(\mathcal{AP}')$. Thus, $\mathcal{AP}'$ is ϵ-bounded. □

The second operation is *merging* two apex-path pairs that contain the same state (Lines 4-6 of Algorithm 2). Conceptually, merging two apex-path pairs corresponds to merging the two sets of paths that these two apex-path pairs correspond to. Hence, the apex of the merged apex-path pair is the component-wise minimum of the apexes of the two apex-path pairs (Line 5). The representative path of the merged apex-path pair is either one of the two representative paths of the two apex-path pairs (Line 6). See Fig. 1 for a visualization of the two possible outcomes. Each of the two representative paths is considered a candidate for the merged apex-path pair if choosing it results in an ϵ-bounded merged apex-path pair. Which candidate A*pex chooses does not affect its correctness but can affect its efficiency. We therefore consider these different methods:

M1 Random method: A*pex randomly chooses the representative path from the candidates.

M2 Lexicographically smallest reverse g-value method: A*pex chooses the representative path with the lexicographically smaller reverse g-value. If this path is not a candidate, A*pex does not merge the apex-path pairs. If there are only two cost components, this method is similar to how PP-A*merges path pairs, namely by picking the bottom-right path.

M3 Greedy method: A*pex chooses the candidate π with the larger slack

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

where $\mathcal{AP}$ is the resulting merged apex-path pair. The ith component of the $\boldsymbol{f}$-value of the representative path could be a factor of $1 + \epsilon_i$ larger than the ith component of the $\boldsymbol{f}$-value of the apex-path pair but is only a factor of $f_i(\pi)/f_i(\mathcal{AP})$ larger. The

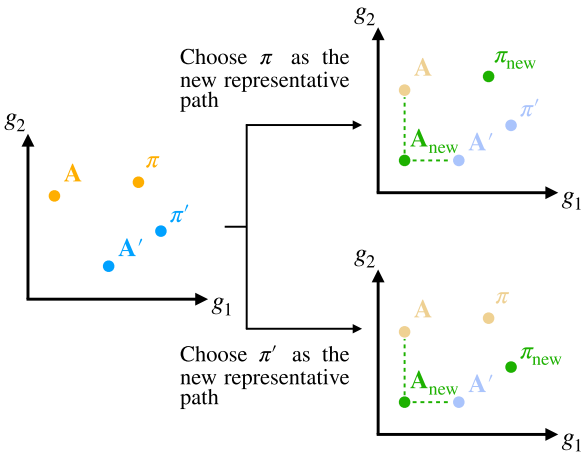


Fig. 1. An example of merging apex-path pairs $\langle \boldsymbol{A}, \pi \rangle$ (orange) and $\langle \boldsymbol{A}', \pi' \rangle$ (blue) into Apex-path pair $\langle \boldsymbol{A}_{\text{new}}, \pi_{\text{new}} \rangle$ (green). Here, when we depict an apex-path pair $\langle \boldsymbol{A}, \pi \rangle$, we use a two-dimensional point to depict the g-value of $\boldsymbol{A}$ and of π and we use their names to distinguish these two points. . (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

smaller the difference between these two values, the better path π utilizes the leeway provided by the approximation factor. The difference, which can range from zero to ϵ_i, is divided by ϵ_i to normalize it and thus make the differences for different components comparable. Overall, the expression above indicates how much room is left to merge the merged apex-path pair with other apex-path pairs in the future, and maximizing it chooses the candidate that leaves more room for future merges.

During its search, A*pex generates only ϵ-bounded apex-path pairs because of Property 2 and also because it merges two apex-path pairs only when the resulting apex-path pair is ϵ-bounded.

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

- **C1$_{\text{A*pex}}$** there exists a solution in SOLS whose cost ϵ-dominates the $\mathbf{f}$-value of $\mathcal{AP}$ (Line 23) or
- **C2$_{\text{A*pex}}$** there exists an expanded apex-path pair that contains the same state as $\mathcal{AP}$ and whose $\mathbf{g}$-value weakly dominates $\mathbf{g}(\mathcal{AP})$ (Line 25).

Condition **C1$_{\text{A*pex}}$** holds iff there exists a vector in $\mathbf{C}^{\text{tr}}_{\text{sol}}$ that ϵ-dominates $Tr(\mathbf{f}(\mathcal{AP}))$, and Condition **C2$_{\text{A*pex}}$** holds iff there exists a vector in $\boldsymbol{G}^{\text{tr}}_{\text{cl}}(s(\mathcal{AP}))$ that weakly dominates $Tr(\mathbf{g}(\mathcal{AP}))$. Conditions **C1$_{\text{A*pex}}$** and **C2$_{\text{A*pex}}$** are checked on Lines 23 and 25, respectively.

When A*pex expands an apex-path pair that contains state s_{goal}, it adds the representative path of this apex-path pair to SOLS and updates $\mathbf{C}^{\text{tr}}_{\text{sol}}$ (Lines 13-14). When A*pex expands an apex-path pair that contains state $s \neq s_{\text{goal}}$, it generates a child apex-path pair $\mathcal{AP}$ for each state $s' \in \texttt{succ}(s)$ of state s by extending the expanded apex-path pair with edge (s, s')). Consider that the child apex-path pair $\mathcal{AP}$ is not discarded after the dominance checks, and let OPEN$[s]$ be the set of apex-path pairs in OPEN that contains state s. A*pex checks on Lines 18-23 if there exists an apex-path pair $\mathcal{AP}'$ in OPEN$[s(\mathcal{AP})]$ that results in an ϵ-bounded apex-path pair when merged with $\mathcal{AP}$. If so, A*pex removes $\mathcal{AP}'$ from OPEN and then adds the merged apex-path pair to OPEN (Lines 21-22). Otherwise, it adds $\mathcal{AP}$ to OPEN (Line 24). When OPEN becomes empty, A*pex terminates and returns SOLS as an ϵ-approximate Pareto frontier (Line 21 and formally proven in Theorem 1).

5. A-A*pex

In this section, we describe A-A*pex, an anytime approximate multi-objective search algorithm that builds upon A*pex. For simplicity, we assume that A-A*pex uses the same approximate factor for all objectives.[1] Therefore, in this section, we denote $\boldsymbol{\epsilon} = [\epsilon, \epsilon \dots \epsilon]$ simply as ϵ. From iteration to iteration, A-A*pex calls A*pex with a sequence of decreasing ϵ-values to compute better and better approximate frontiers. Between iterations, A-A*pex can either reuse previous search efforts or restart the search from

[1] There is no technical restriction forcing us to take this assumption. It is only introduced to avoid the need to handle cost scaling and to simplify the exposition.

Algorithm 3: A*pex.

Input : A problem instance $\langle G, s_{\text{start}}, s_{\text{goal}}\rangle$, a consistent heuristic function $\mathbf{h}$, and an approximation factor $\boldsymbol{\epsilon}$
Output: An ϵ-approximate Pareto frontier

1 OPEN $\leftarrow \{\langle \mathbf{0}, [s_{\text{start}}]\rangle\}$
2 SOLS $\leftarrow \emptyset$
3 $\mathbf{C}^{\text{tr}}_{\text{sol}} \leftarrow \emptyset$
4 **foreach** $s \in S$ **do**
5 $\quad \boldsymbol{G}^{\text{tr}}_{\text{cl}}(s) \leftarrow \emptyset$
6 **while** *OPEN* $\neq \emptyset$ **do**
7 $\quad$ extract an apex-path pair $\mathcal{AP} = \langle \boldsymbol{A}, \pi\rangle$ from OPEN with the lexicographically smallest $\boldsymbol{f}$-value
8 $\quad$ **if** $isDominated(\mathcal{AP})$ **then**
9 $\quad\quad$ **continue**
10 $\quad$ update$(\boldsymbol{G}^{\text{tr}}_{\text{cl}}(s(\mathcal{AP})), Tr(\mathbf{g}(\mathcal{AP})))$
11 $\quad$ **if** $s(\mathcal{AP}) = s_{goal}$ **then**
12 $\quad\quad$ remove the solutions weakly dominated by π from SOLS
13 $\quad\quad$ update$(\mathbf{C}^{\text{tr}}_{\text{sol}}, Tr(\mathbf{c}(\pi)))$
14 $\quad\quad$ add π to SOLS
15 $\quad\quad$ **continue**
16 $\quad$ **for** $s' \in succ(s(\mathcal{AP}))$ **do**
17 $\quad\quad$ $\mathcal{AP}' \leftarrow$ extend$(\mathcal{AP}, \langle s, s'\rangle)$
18 $\quad\quad$ **if** $isDominated(\mathcal{AP}')$ **then**
19 $\quad\quad\quad$ **continue**
20 $\quad\quad$ addToOpen$(\mathcal{AP}')$
21 **return** SOLS
22 **Function** $isDominated(\mathcal{AP} = \langle \boldsymbol{A}, \pi\rangle)$**:**
23 $\quad$ **if** $\exists \mathbf{x} \in \mathbf{C}^{tr}_{sol} : \mathbf{x} \preceq_{Tr(\epsilon)} Tr(\mathbf{f}(\mathcal{AP}))$ **then**
24 $\quad\quad$ **return** ***true***
25 $\quad$ **if** $\exists \mathbf{x} \in \boldsymbol{G}^{\text{tr}}_{\text{cl}}(s(\mathcal{AP})) : \mathbf{x} \preceq Tr(\mathbf{g}(\mathcal{AP}))$ **then**
26 $\quad\quad$ **return** ***true***
27 $\quad$ **return** ***false***
28 **Function** $addToOpen(\mathcal{AP})$**:**
29 $\quad$ **for** $\mathcal{AP}' \in$ *OPEN*$[s(\mathcal{AP})]$ **do**
30 $\quad\quad$ $\mathcal{AP}_{\text{new}} \leftarrow$ merge$(\mathcal{AP}, \mathcal{AP}')$
31 $\quad\quad$ **if** $\mathcal{AP}_{new}$ *is* ϵ*-bounded* **then**
32 $\quad\quad\quad$ remove $\mathcal{AP}'$ from OPEN
33 $\quad\quad\quad$ add $\mathcal{AP}_{\text{new}}$ to OPEN
34 $\quad\quad\quad$ **return**
35 $\quad$ add $\mathcal{AP}$ to OPEN
36 $\quad$ **return**
37 **Function** $update(\mathbf{X}, \mathbf{y})$**:**
38 $\quad$ remove the vectors weakly dominated by $\mathbf{y}$ from $\mathbf{X}$
39 $\quad$ add $\mathbf{y}$ to $\mathbf{X}$

scratch. We first describe the variant of A-A*pex that reuses search efforts. Then, we describe the restarting variant, which only differs in a few lines of pseudo-code.

Algorithm 4 shows the pseudocode of the variant of A-A*pex that reuses its previous search effort. The input to Algorithm 4 is a problem instance, a consistent heuristic $\mathbf{h}$, and an approximation factor update scheme encoded by the `getNextEps` function. Here we use a simple scheme wherein the sequence of ϵ-values output by `getNextEps` began with $\epsilon_{\text{init}} = 0.1$ and was divided by η after every iteration, where $\eta \geq 1$ was a predetermined parameter, see also Section 7. However, any monotonically decreasing sequence converging to 0 can be used.

A-A*pex maintains a list PRUNED of pruned paths, which is initialized with path $[s_{\text{start}}]$ (Line 1), and a set SOLS of solutions, which is initialized as an empty set (Line 2). In each iteration of its main loop (Lines 3-11), A-A*pex first calls `getNextEps` to decrease the *current approximation factor* ϵ_{curr} (Line 4). It then initializes OPEN with the paths in PRUNED (Lines 5-8): A-A*pex first moves the paths from PRUNED to another set PRUNED′ (Line 6) and then calls `addToOpen` with each of the paths in PRUNED′. Some of these

Algorithm 4: A-A*pex.

Input : A problem instance $\langle G, s_{\text{start}}, s_{\text{goal}} \rangle$, a consistent heuristic function $\mathbf{h}$, and an approximation factor update scheme `getNextEps()`
Output: A Pareto frontier

1 PRUNED $\leftarrow \{[s_{\text{start}}]\}$
2 SOLS $\leftarrow \emptyset$
3 **while** *Search not halted* **do**
4 $\quad \varepsilon_{\text{curr}} \leftarrow$ `getNextEps()`
5 $\quad$ OPEN $\leftarrow \emptyset$
6 $\quad$ PRUNED$'$ $\leftarrow$ PRUNED; PRUNED $\leftarrow \emptyset$
7 $\quad$ **for each** $\pi \in$ *PRUNED*$'$ **do**
8 $\quad\quad$ `addToOpen`$(\langle \mathbf{c}(\pi), \pi \rangle)$
9 $\quad$ `findApproxPF`$(\varepsilon_{\text{curr}})$
10 $\quad$ **if** *PRUNED* $= \emptyset$ **then**
11 $\quad\quad$ **break**
12 **return** *SOLS*
13 **Function** *findApproxPF*(ε_{curr})**:**
14 $\quad$ SOLS$'$ $\leftarrow$ sort SOLS in the lexicographically increasing order of their costs
15 $\quad \mathbf{C}^{\text{tr}}_{\text{sol}} \leftarrow \emptyset$
16 $\quad$ **foreach** $s \in S$ **do**
17 $\quad\quad \boldsymbol{G}^{\text{tr}}_{\text{cl}}(s) \leftarrow \emptyset$
18 $\quad$ **while** *OPEN* $\neq \emptyset$ **do**
19 $\quad\quad$ extract an apex-path pair $\mathcal{AP} = \langle \boldsymbol{A}, \pi \rangle$ from OPEN with the lexicographically smallest $\boldsymbol{f}$-value
20 $\quad\quad$ **while** $c_1(\textit{SOLS}'.top()) \leq (1 + \varepsilon_{curr}) \cdot f_1(\mathcal{AP})$ **do**
21 $\quad\quad\quad$ `update`$(\mathbf{C}^{\text{tr}}_{\text{sol}}, Tr(\mathbf{c}(\text{SOLS}'.top())))$
22 $\quad\quad\quad$ pop SOLS$'.top()$ from SOLS$'$
23 $\quad\quad$ **if** *isDominated*$(\mathcal{AP})$ **then**
24 $\quad\quad\quad$ **continue**
25 $\quad\quad$ `update`$(\boldsymbol{G}^{\text{tr}}_{\text{cl}}(s(\mathcal{AP})), Tr(\mathbf{g}(\mathcal{AP})))$
26 $\quad\quad$ **if** $s(\mathcal{AP}) = s_{goal}$ **then**
27 $\quad\quad\quad$ remove solutions weakly dominated by π from SOLS
28 $\quad\quad\quad$ `update`$(\mathbf{C}^{\text{tr}}_{\text{sol}}, Tr(\mathbf{c}(\pi)))$
29 $\quad\quad\quad$ add π to SOLS
30 $\quad\quad\quad$ **continue**
31 $\quad\quad$ **for** $e \in succ(s(\mathcal{AP}))$ **do**
32 $\quad\quad\quad \mathcal{AP}' \leftarrow$ `extend`$(\mathcal{AP}, e)$
33 $\quad\quad\quad$ **if** *isDominated*$(\mathcal{AP}')$ **then**
34 $\quad\quad\quad\quad$ **continue**
35 $\quad\quad\quad$ `addToOpen`$(\mathcal{AP}')$
36 $\quad$ **Function** *getNextEps()***:**
37 $\quad\quad$ **if** *first time called* **then**
38 $\quad\quad\quad$ **return** $\varepsilon_{\text{init}}$
39 $\quad\quad$ **return** $\varepsilon_{\text{curr}}/\eta$
40 $\quad$ **Function** *update*$(\mathbf{X}, \mathbf{y})$**:**
41 $\quad\quad$ remove the vectors weakly dominated by $\mathbf{y}$ from $\mathbf{X}$
42 $\quad\quad$ add $\mathbf{y}$ to $\mathbf{X}$

paths might be put back into PRUNED by `addToOpen`, which we will explain shortly. A-A*pex then calls `findApproxPF` to compute an $\varepsilon_{\text{curr}}$-approximate frontier (Line 9).

Lines 13-35 show the `findApproxPF` function, which is similar to Lines 3-21 of A*pex in Algorithm 3. The `isDominated` and `addToOpen` functions for `findApproxPF` are modified to identify paths that might still be extendable to Pareto-optimal solutions and add these paths to PRUNED. We will describe these two modified functions in detail in the next section.

Like A*pex, the `findApproxPF` function maintains a set of truncated cost vectors $\mathbf{C}^{\text{tr}}_{\text{sol}}$ for checking if the $\mathbf{f}$-value of a given apex-path pair is $\varepsilon_{\text{curr}}$-dominated by the cost of any solution in SOLS. Unlike A*pex, it also needs to consider the solutions computed in the previous iterations of `findApproxPF` in $\mathbf{C}^{\text{tr}}_{\text{sol}}$. It first stores these solutions in SOLS$'$ and sorts them in the lexicographically increasing

order of their costs (Line 14). After extracting an apex-path pair $\mathcal{AP}$ from OPEN, `findApproxPF` adds to $\mathbf{C}^{\text{tr}}_{\text{sol}}$ the costs of those solutions in SOLS′ whose c_1-values are no larger than $(1+\varepsilon_{\text{curr}})$ times the f_1-value of $\mathcal{AP}$ and removes these solutions from SOLS′ (Lines 20-22). The $\mathbf{f}$-value of $\mathcal{AP}$ is $\varepsilon_{\text{curr}}$-dominated by the cost of a solution in SOLS iff there exists a vector in $\mathbf{C}^{\text{tr}}_{\text{sol}}$ that $\varepsilon_{\text{curr}}$-dominates $Tr(\mathbf{f}(\mathcal{AP}))$.

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

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

Algorithm 5: The isDominated and addToOpen functions for A-A*pex.

```
1  Function isDominated(AP = ⟨A, π⟩):
2      if ∃x ∈ C^tr_sol : x ⪯_{ε_curr} Tr(f(AP)) then
+3         π′ ← the solution that corresponds to x
+4         if not c(π′) ⪯ f(π) then
+5             add π to PRUNED
6          return true
7      if ∃x ∈ G^tr_cl(s(AP)) : x ⪯ Tr(g(AP)) then
+8         π′ ← the representative path of the apex-path pair corresponding to x
+9         if not c(π′) ⪯ c(π) then
+10            add π to PRUNED
11         return true
12     return false
13 Function addToOpen(AP = ⟨A, π⟩):
14     for AP′ = ⟨A′, π′⟩ ∈ OPEN[s(AP)] do
15         AP_new = ⟨A_new, π_new⟩ ← merge(AP, AP′)
16         if AP_new is ε_curr-bounded then
17             remove AP′ from OPEN
18             add AP_new to OPEN
+19            π_pruned ← π′ if π = π_new or π otherwise
+20            if not c(π_new) ⪯ c(π_pruned) then
+21                add π_pruned to PRUNED
22             return
23     add AP to OPEN
24     return
```

Algorithm 5 shows the `isDominated` and `addToOpen` functions for A-A*pex. We use "+" before the line numbers to indicate the changes in these two functions:

1. **Lines 3-5:** If the truncated $\mathbf{f}$-value of an apex-path pair $\mathcal{AP}$ is $\varepsilon_{\text{curr}}$-dominated by some vector $\mathbf{x}$ in $\mathbf{C}^{\text{tr}}_{\text{sol}}$, the $\mathbf{f}$-value of $\mathcal{AP}$ is $\varepsilon_{\text{curr}}$-dominated by some solution π' in SOLS. If the representative path π of $\mathcal{AP}$ also satisfies that $\mathbf{f}(\pi)$ is not weakly dominated by $\mathbf{c}(\pi')$, A-A*pex adds π to PRUNED.
2. **Lines 8-10:** For each vector $\mathbf{x}$ in $\boldsymbol{G}^{\text{tr}}_{\text{cl}}(s)$ for any state s, A-A*pex maintains the representative path of the apex-path pair whose truncated $\mathbf{g}$-value equals $\mathbf{x}$ and resulted in $\mathbf{x}$ being added to $\boldsymbol{G}^{\text{tr}}_{\text{cl}}(s)$. If the truncated $\mathbf{g}$-value of an apex-path pair $\mathcal{AP}$ is weakly dominated by some vector $\mathbf{x}$ in $\boldsymbol{G}^{\text{tr}}_{\text{cl}}(s(\mathcal{AP}))$ and the representative path π of $\mathcal{AP}$ is not weakly dominated by the representative path of the apex-path pair corresponding to $\mathbf{x}$, A-A*pex adds π to PRUNED.
3. **Lines 19-21:** When merging two apex-path pairs, one of their representative paths is chosen as the new representative path. Let π_{new} denote the chosen representative path and π_{pruned} denote the other. If π_{pruned} is not weakly dominated by π_{new}, A-A*pex adds π_{pruned} to PRUNED.

These three changes cover all possible places where `findApproxPF` "disregards" a path. Conceptually, `findApproxPF` does not need to consider these paths because any solution π_{sol} that extends such a path π is known to be $\varepsilon_{\text{curr}}$-dominated by some other solution π'_{sol}. However, if π_{sol} is not weakly dominated by π'_{sol}, it is still possible that π_{sol} is in the Pareto frontier. Hence, `findApproxPF` adds π to PRUNED. When $\varepsilon_{\text{curr}}$ becomes sufficiently small, the truncated $\mathbf{g}$-or $\mathbf{f}$-value of apex-path pair is $\varepsilon_{\text{curr}}$-dominated by some vector

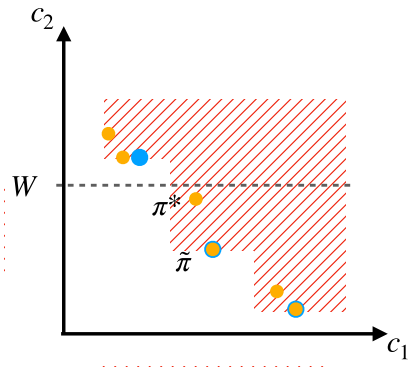


Fig. 2. Example of the Pareto frontier (whose costs are shown by the orange dots) and an $(\varepsilon, 0)$-approximate frontier(whose costs are shown by the blue dots) from the start state to the goal state of a WCSP instance, respectively. The shaded region shows the costs that are $(\varepsilon, 0)$-dominated by at least one blue dot. Note that all orange dots are within the shaded region. Solutions π^* and $\tilde{\pi}$ are an optimal solution and a $(1+\varepsilon)$-suboptimal solution of the WCSP instance, respectively, with $c(\tilde{\pi}) \preceq_{(\varepsilon,0)} c(\pi^*)$. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

only when it is weakly dominated by this vector. Also, when merging two apex-path pairs, the chosen representative path also always weakly dominates the other representative path. Therefore, PRUNED becomes empty, and A-A*pex returns SOLS, which is a Pareto frontier.

Enhanced dominance checks. Although A-A*pex does not reuse the truncated **g**-values from previous iterations for dominance checks, it can still prune an apex-path pair $\mathcal{AP}$ if $\mathbf{g}(\mathcal{AP})$ is weakly dominated by the cost of the representative path π' of an apex-path pair that was expanded in previous iterations and contains the same state as $\mathcal{AP}$. In such a case, the entire set of paths that $\mathcal{AP}$ corresponds to is weakly dominated by π' despite the $\varepsilon_{\text{curr}}$-value. One can thus enhance the dominance checks of A-A*pex by maintaining the set of undominated costs $\mathbf{C}(s)$ of the representative paths of all expanded apex-path pairs for each state s and using these for dominance checks. This requires changes only inside the `isDominated` function to check if the **g**-value of the input apex-path pair $\mathcal{AP}$ is weakly dominated by any vector in $\mathbf{C}(s(\mathcal{AP}))$ and adding one line after Line 25 of Algorithm 4 to update $\mathbf{C}(s(\mathcal{AP}))$ before expanding $\mathcal{AP}$. Although the enhanced dominance checks enable A-A*pex to prune more nodes, performing these checks and updating $\mathbf{C}(s)$ can be time-consuming. It is also unclear how to use the dimensionality reduction technique in these checks. In our experimental evaluation (Section 7.2.2), we will study if adding these checks improves the efficiency of A-A*pex or not.

A-A*pex with restarting. Instead of reusing its previous search effort, A-A*pex can also restart the search from scratch in each iteration. This requires changes only to Lines 5-8 of Algorithm 4, where A-A*pex now initializes OPEN with path $[s_{\text{start}}]$ instead of the paths in PRUNED. This variant of A-A*pex cannot use the enhanced dominance checks because it needs to expand apex-path pairs with the same representative paths as those expanded in the previous iterations of `findApproxPF`.

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

6. WC-A*pex

In this section, we propose WC-A*pex, which generalizes A*pex to computing a $(1+\varepsilon)$-suboptimal solution for a WCSP problem instance and a user-provided ε-value. We first use the following observation to show the connection between a bounded-suboptimal WCSP algorithm and an approximate bi-objective search algorithm.

Observation 1. *For a feasible WCSP instance $P = \langle G, s_{start}, s_{goal}, W\rangle$ and $\varepsilon \geq 0$, any $(\varepsilon, 0)$-approximate frontier (from s_{start} to s_{goal}) Π contains a $(1+\varepsilon)$-suboptimal solution of P.*

Proof. Let π^* denote an optimal solution of P. By definition, there exists a path $\pi \in \Pi$ with $c(\pi) \preceq_{(\varepsilon,0)} c(\pi^*)$ (i.e., $c_1(\pi) \leq (1+\varepsilon) \cdot c_1(\pi^*)$ and $c_2(\pi) \leq c_2(\pi^*) \leq W$). We can see that π is a $(1+\varepsilon)$-suboptimal solution of P. □

See Fig. 2 for a visualization of an ε-approximate frontier and a $(1+\varepsilon)$-suboptimal solution of a WCSP instance.

Observation 1 shows that a $(1+\varepsilon)$-suboptimal solution of a WCSP instance can be found in a corresponding $(\varepsilon, 0)$-approximate frontier. This motivates us to propose WC-A*pex, which can be viewed as a modified A*pex with $\boldsymbol{\varepsilon} = (\varepsilon, 0)$ and additional prunings. Note that we use boldface $\boldsymbol{\varepsilon}$ and regular ε to distinguish between the approximation factor that A*pex would use and the user-provided parameter for the WCSP problem. Since the second component of $\boldsymbol{\varepsilon}$ is set to 0, when merging two apex-path pairs $\langle \boldsymbol{A}, \pi\rangle$

and $\langle \boldsymbol{A}', \pi' \rangle$ with $c_2(\pi) < c_2(\pi')$, WC-A*pexcannot choose π' as the new representative path. Otherwise, the resulting apex path pair is not ϵ-bounded. In addition to dominance checks, WC-A*pexalso prunes nodes whose f_2-values are larger than W.

Algorithm 6: WC-A*pex.

Input : A WCSP problem instance $\langle G, s_{\text{start}}, s_{\text{goal}}, W \rangle$, a consistent heuristic function **h**, and a non-negative ϵ-value
Output: A $(1+\epsilon)$-suboptimal solution

1 OPEN $\leftarrow \{\langle \mathbf{0}, [s_{\text{start}}] \rangle\}$
2 **for each** $s \in S$ **do**
3 $\quad g_2^{\min}(s) \leftarrow \infty$
4 **while** *OPEN* $\neq \emptyset$ **do**
5 $\quad$ extract an apex-path pair $\mathcal{AP} = \langle \boldsymbol{A}, \pi \rangle$ from OPEN with the lexicographically smallest $\boldsymbol{f}$-value
6 $\quad$ **if** $g_2(\mathcal{AP}) \geq g_2^{min}(s(\mathcal{AP})) \vee f_2(\mathcal{AP}) > W$ **then**
7 $\quad\quad$ **continue**
8 $\quad g_2^{\min}(s(\mathcal{AP})) \leftarrow g_2(\mathcal{AP})$
9 $\quad$ **if** $s(\mathcal{AP}) = s_{goal}$ **then**
10 $\quad\quad$ **return** π
11 $\quad$ **for** $s' \in succ(s(\mathcal{AP}))$ **do**
12 $\quad\quad \mathcal{AP}' \leftarrow \texttt{extend}(\mathcal{AP}, \langle s, s' \rangle)$
13 $\quad\quad$ **if** $g_2(\mathcal{AP}) \geq g_2^{min}(s') \vee f_2(\mathcal{AP}') > W$ **then**
14 $\quad\quad\quad$ **continue**
15 $\quad\quad \texttt{addToOpen}(\mathcal{AP}')$
16 **return** *None*
17 **Function** *addToOpen*$(\mathcal{AP})$**:**
18 $\quad$ **for** $\mathcal{AP}' \in$ *OPEN*$[s(\mathcal{AP})]$ **do**
19 $\quad\quad \mathcal{AP}_{\text{new}} \leftarrow \texttt{merge}(\mathcal{AP}, \mathcal{AP}')$
20 $\quad\quad$ **if** $\mathcal{AP}_{new}$ *is* $(\epsilon, 0)$*-bounded* **then**
21 $\quad\quad\quad$ remove $\mathcal{AP}'$ from OPEN
22 $\quad\quad\quad$ add $\mathcal{AP}_{\text{new}}$ to OPEN
23 $\quad\quad\quad$ **return**
24 $\quad$ add $\mathcal{AP}$ to OPEN
25 $\quad$ **return**

Algorithm 6 shows the pseudocode of WC-A*pex. It starts with a single apex-path pair $\langle \mathbf{0}, [s_{\text{start}}] \rangle$ in OPEN (Line 1). Different from A*pex, WC-A*pexdoes not maintain SOLS because it only needs to compute one solution. At each iteration, WC-A*pexextracts an apex-path pair from OPEN with the lexicographically smallest $\boldsymbol{f}$-value (Line 5). Because the WCSP problem only considers two objectives, a truncated $\boldsymbol{g}$-value corresponds to and thus can be represented by a single number. The set of undominated truncated $\boldsymbol{g}$-values of the expanded apex-path pairs that contains some state s can be in turn represented by the smallest g_2-value of these expanded apex-path pairs, denoted as $g_2^{\min}(s)$. Both after extracting (that is, after Line 5) and before generating (that is, before Line 15) an apex-path pair $\mathcal{AP}$ with state s, WC-A*pexdiscards $\mathcal{AP}$ if there exists an expanded apex-path pair that contains the same state as $\mathcal{AP}$ and whose $\boldsymbol{g}$-value weakly dominates $\boldsymbol{g}(AP)$, which can be done by checking if $g_2(\mathcal{AP}) \geq g_2^{\min}(s(\mathcal{AP}))$. Additionally, WC-A*pexdiscards $\mathcal{AP}$ if $f_2(\mathcal{AP}) > W$. In this case, the representative path of $\mathcal{AP}$ cannot be extended to a solution (whose c_2-value needs to be no larger than W). WC-A*pexterminates when it finds a solution (Line 10) or OPEN becomes empty (Line 16), in which case, there is no solution for the given WCSP instance.

7. Experimental results

In this section, we report experimental results for A*pex, A-A*pex, and WC-A*pex. We implemented all algorithms in C++ and ran all experiments on a MacBook with an M1 Pro-chip and 32GB of memory. For all experiments, we set a runtime limit of five minutes.

7.1. Evaluation of A*pex

7.1.1. Problem instances with two objectives

We compare different variants of A*pex with BOA*, BOA*-ϵ, and PP-A*in problem instances with two objectives. We use five road networks from the 9th DIMACS Implementation Challenge,[2] which are the NY road network (264K states and 730K edges), the

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

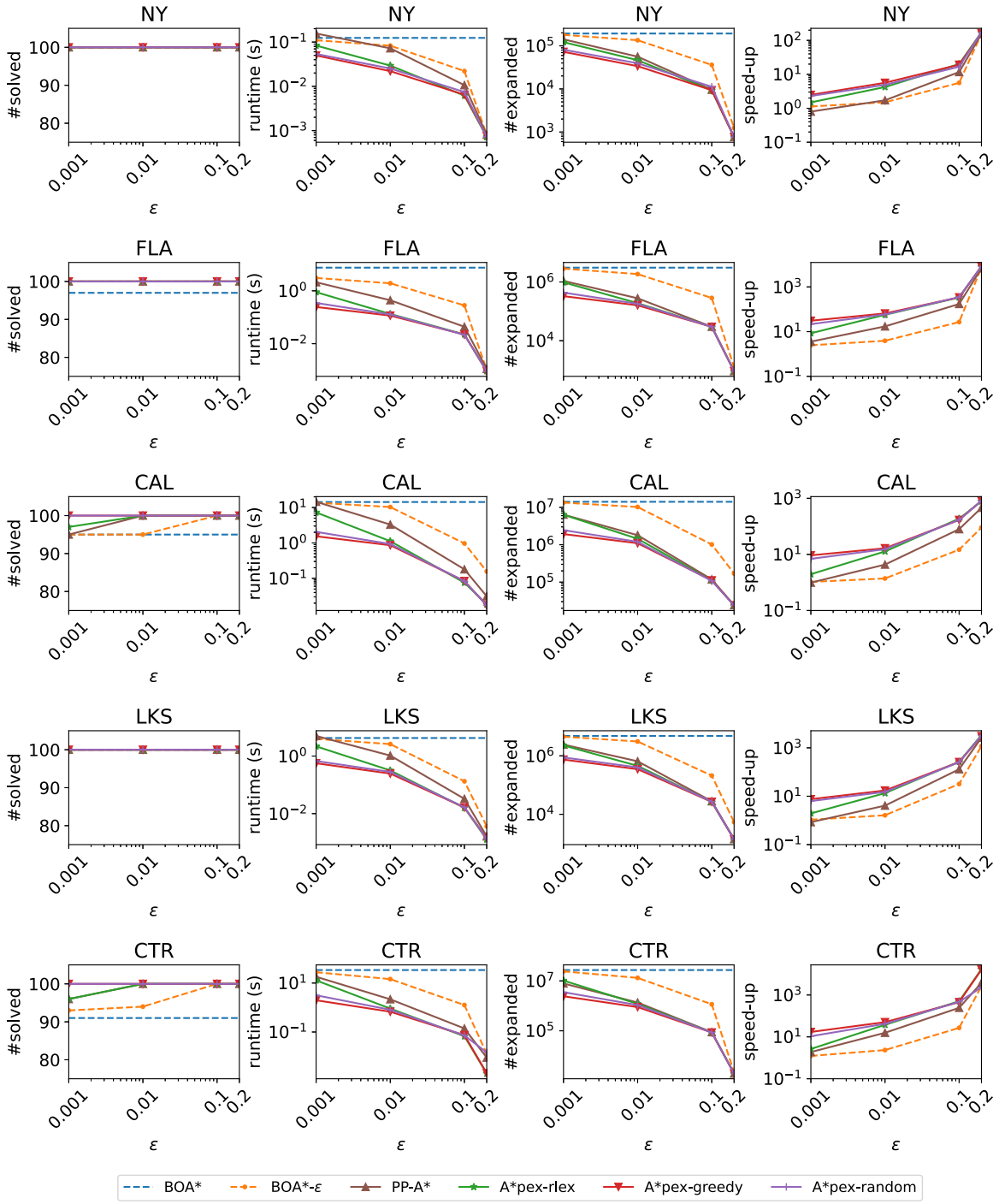


Fig. 3. Results for BOA*, BOA*-ϵ, PP-A*, and different variants of A*pex on bi-objective problem instances with different approximation factors.

FLA road network (1.1M states and 2.7M edges), the CAL road network (1.9M states and 4.7M edges), the LKS road network (2.8M states and 6.9M edges), and the CTR road network (14.1M states and 34.3M edges). We use the two objectives that are available in the benchmark, namely travel time (t) and travel distance (d). We use the same 100 problem instances used by Ahmadi et al. [35] for each road network.

Fig. 3 shows the numbers of solved problem instances, average runtimes (in seconds), and average numbers of node expansions of all algorithms and speed-ups of BOA*-ϵ, PP-A*, and all variants of A*pex over BOA* in average runtimes. All average values are taken over problem instances solved by all algorithms within the runtime limit for all ϵ-values. All variants of A*pex have larger

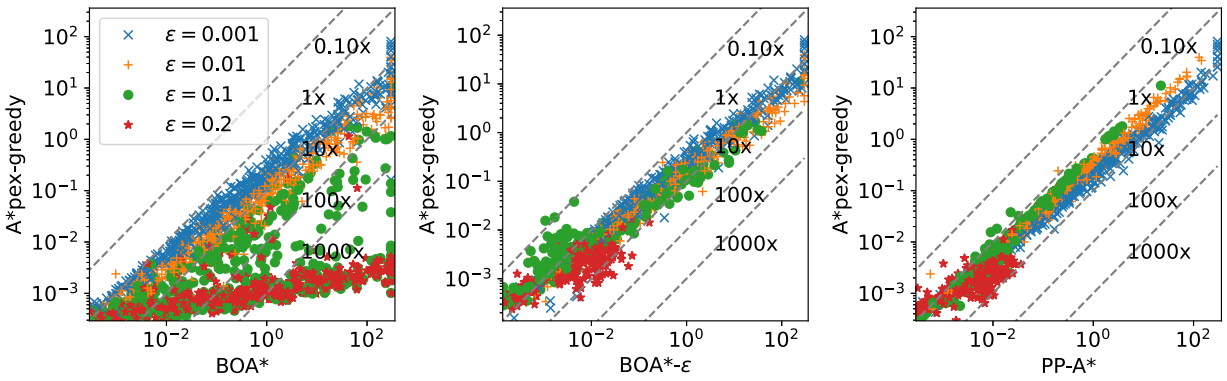


Fig. 4. Runtime comparisons between A*pex-greedy and different algorithms on bi-objective road-network problem instances with different approximation factors.

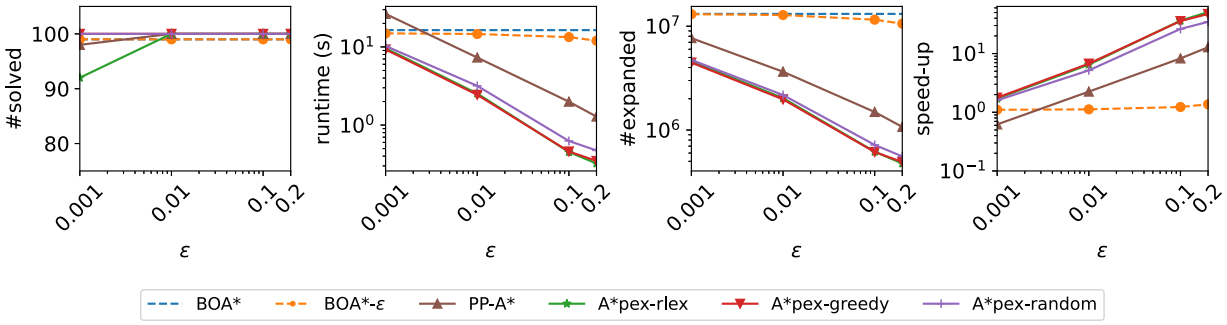


Fig. 5. Results for BOA*, BOA*-ϵ, PP-A*, and different variants of A*pex on NY map with travel time and the economic cost objectives and different approximation factors.

numbers of solved problem instances, smaller average numbers of node expansions, and smaller average runtime than BOA*, BOA*-ϵ, and PP-A*in most cases. Among different variants of A*pex, A*pex-greedy slightly outperforms the other two variants. The numbers of solved instances of A*pex-greedy are at least as high as the ones of A*pex-rlex and A*pex-random in most cases.

Fig. 4 shows the comparisons of individual runtime (in seconds) between A*pex-greedy and different algorithms in all problem instances. The x- and y-coordinates of each point show the runtimes of the baseline algorithm and A*pex-greedy, respectively, for a problem instance. Different markers represent different ε-values. A*pex-greedy outperforms all three algorithms except in some easy problem instances where BOA*-$\epsilon$and PP-A*are faster than A*pex-greedy. As the ε-value increases, the speed-ups of A*pex-greedy over BOA* become more substantial. In all three cases, the speed-ups of A*pex-greedy are more substantial in hard problem instances (represented by those markers in the top-right).

To complement, Fig. 5 shows results with weakly correlated objectives. The objective are travel time and the economic cost (fuel cost + road toll). The pearson correlation between objectives is 0.16. Similar of usign travel time and and travel distance, all variants of A*pex have larger numbers of solved problem instances, smaller average numbers of node expansions, and smaller average runtime than BOA*, BOA*-ϵ, and PP-A*in most cases. Only with $\varepsilon = 0.0001$ A*pex-greedy-rlex solved less problems.

7.1.2. Problem instances with more than two objectives

In this section, we compare different variants of A*pex with LTMOA* and LTMOA*-ϵ in problem instances with more than two objectives. We use the NY road network from the 9th DIMACS Implementation Challenge. In addition to travel time (t) and travel distance (d), we use the economic cost (m) [38], the number of edges (l) [41], and a random integer from 1 to 100 (r) [22] as the third, fourth, and fifth objectives, respectively. We use the same 100 problem instances used by Sedeño-Noda and Colebrook [42] and Ahmadi et al. [35].

Fig. 6 shows the numbers of solved problem instances, average runtimes (in seconds), and average numbers of node expansions of all algorithms and the speed-ups of LTMOA*-ϵ and different variants of A*pex over LTMOA* in average runtime. All average values are taken over problem instances solved by all algorithms within the runtime limit for all ε-values. All variants of A*pex have smaller average numbers of node expansions than LTMOA* and LTMOA*-ϵ in all cases. However, the average runtimes of A*pex are larger than the ones of LTMOA* and LTMOA*-ϵ for $\varepsilon = 0.001$, which is due to the runtime overhead of A*pex in iterating over OPEN and merging apex-path pairs. In all other cases, the average runtimes of all A*pex variants are smaller than the ones of LTMOA* and LTMOA*-ϵ. The speed-ups of LTMOA*-ϵ and different variants of A*pex over LTMOA* increase as the ε-value increases. However, the speed-ups of LTMOA*-ϵ are much less substantial than the speed-ups of different variants of A*pex for larger ε-values. Similar

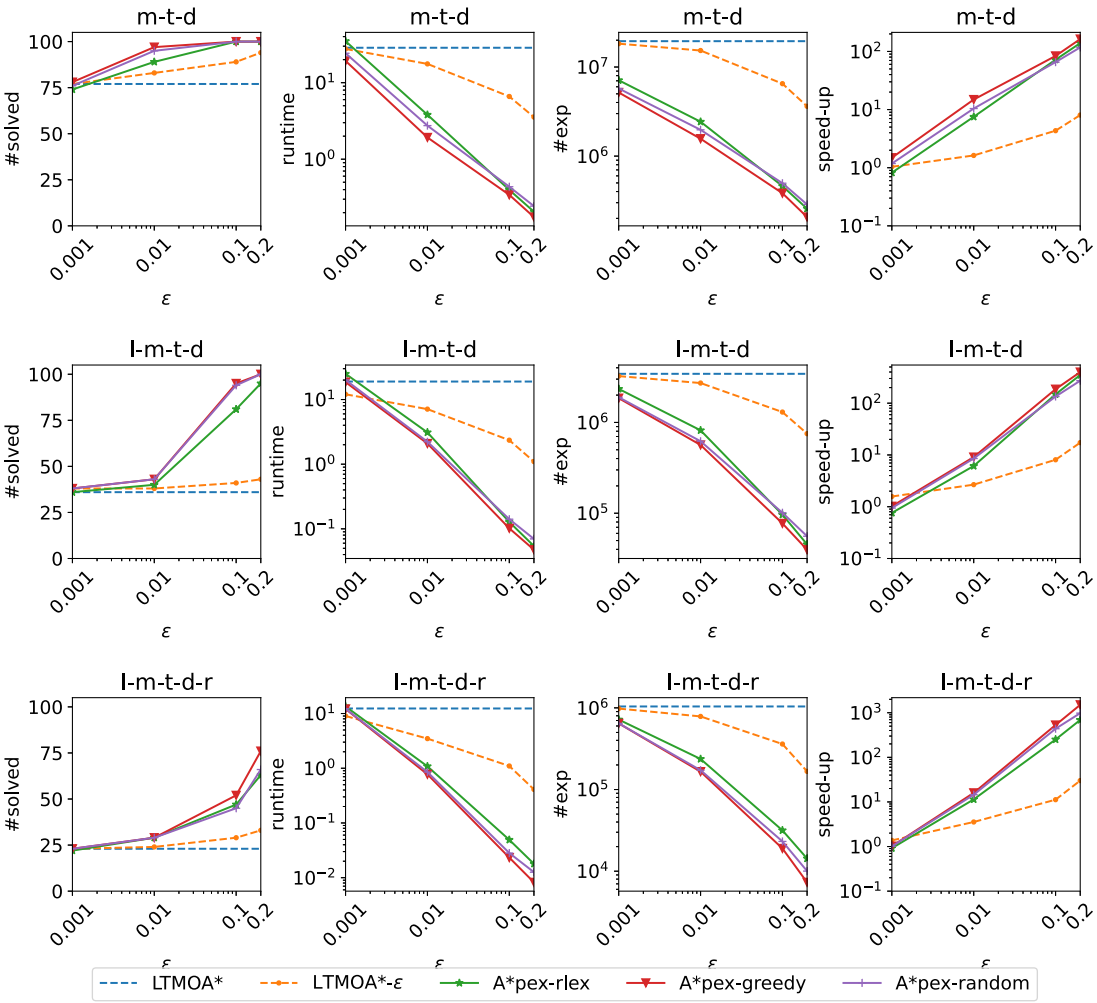


Fig. 6. Results for LTMOA*, LTMOA*-ϵ, and different variants of A*pex on road-network problem instances with different objectives and approximation factors.

to the results for bi-objective problem instances, the results for problem instances with more than two objectives show that A*pex-greedy slightly outperforms the other two A*pex variants. The numbers of solved instances of A*pex-greedy are at least as high as the ones of A*pex-rlex and A*pex-random in all cases.

Fig. 7 shows the runtime comparisons between LTMOA* and A*pex-greedy for different numbers of objectives, and Fig. 8 shows the runtime comparisons between LTMOA*-ϵ and A*pex-greedy for different numbers of objectives. In problem instances with $\epsilon = 0.001$, A*pex-greedy has similar runtimes as LTMOA* and LTMOA*-ϵ. In problem instances with other ϵ-values, A*pex-greedy outperforms LTMOA* and LTMOA*-ϵ in most problem instances. The speed-up of A*pex-greedy over LTMOA* increases as ϵ-value increases. With an approximation factor of 0.2, the speed-ups of A*pex-greedy over LTMOA* are up to more than 1000×.

Finally, Fig. 9 shows the number of solution found over problem instances solved by all algorithms within the runtime limit for all ϵ-values. In all cases, the number of solutions decrease when the ϵ-value increases for all the approximate algorithms.

While LTMOA*-ϵ is slower and performs more expansions than the different variants of A*pex, LTMOA*-ϵ obtains fewer solutions than the different variants of A*pex. The reason for the difference in the number of solutions is that A*pex does not necessarily produce solutions on the Pareto frontier while LTMOA*-ϵ does as LTMOA*-ϵ only prunes paths according to the already found solutions. This may cause more approximate solutions to be found by A*pex (though both variants produce solutions that ϵ-approximate the true Pareto frontier).

7.1.3. Maps with random edge costs

As costs in road networks can exhibit correlation (e.g., between travel time and travel distance) [43], in this section we extend the comparison of A*pex with LTMOA* and LTMOA*-ϵ presented in Section 7.1.2 to maps with random edge costs. Following [44],

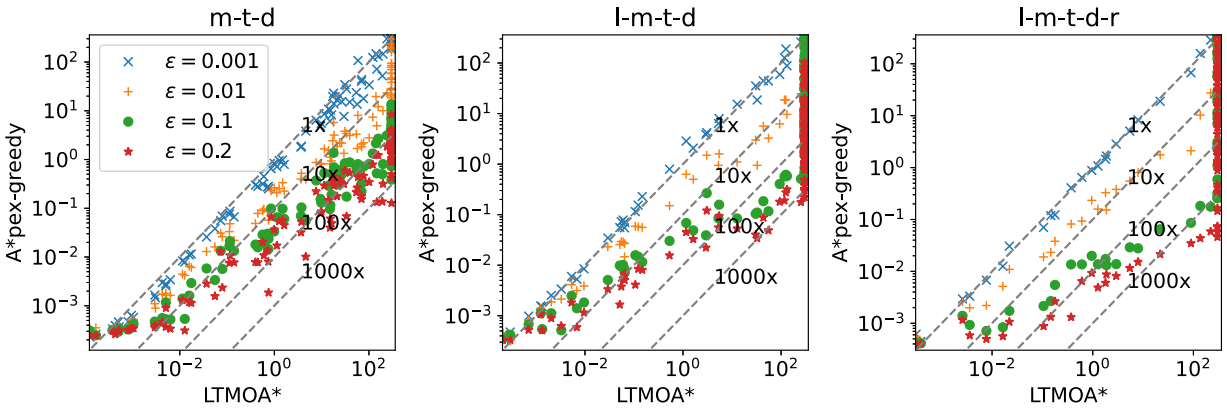


Fig. 7. Runtimes of LTMOA* and A*pex-greedy on road-network problem instances with different objectives and approximation factors.

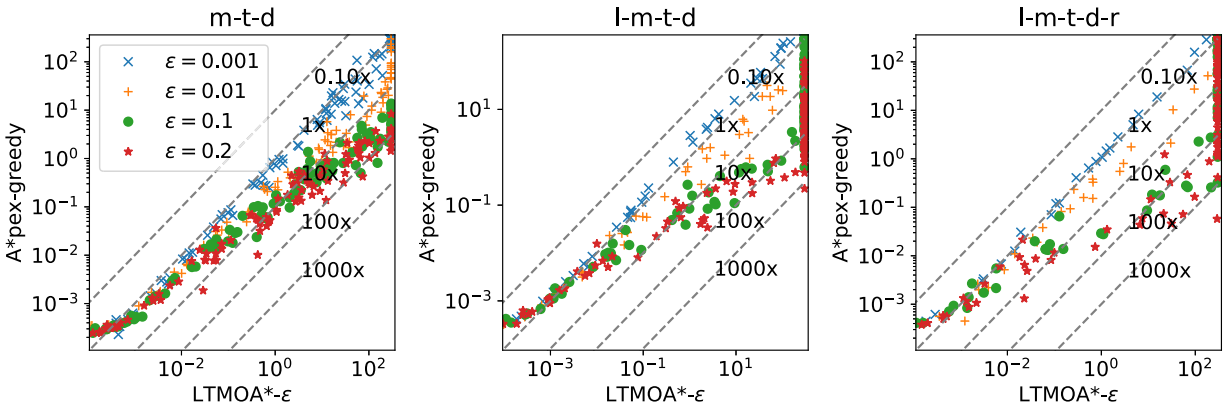


Fig. 8. Runtimes of LTMOA*-ε and A*pex-greedy on road-network problem instances with different objectives and approximation factors.

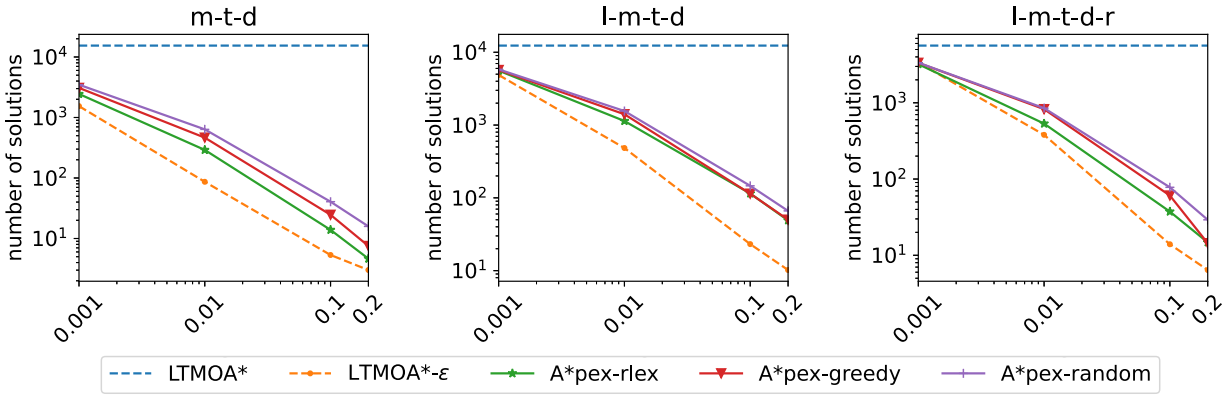


Fig. 9. Number of solutions of LTMOA*, LTMOA*-ε, and different variants of A*pex on road-network problem instances with different objectives (m-t-d, l-m-t-d, and l-m-t-d-r) and approximation factors..

we use the COL road network (436K states and 1.01M edges) from the 9th DIMACS Implementation Challenge, which is larger than the NY road network. We used two, three, four, and five objectives in this map.

Specifically, and in contrast to the edge weights for the NY road network used in Section 7.1.2, we use random integer costs for all edges and all cost components: Given two integers $l < h$, let rnd(l, h) denote a function choosing an integer uniformly at random in the range $[l, h]$. For the first cost component c_1 of each edge e, we set $c_1(e) := \text{rnd}(1, 10000)$. For the i'th cost component c_1 of each edge e (with $i = 2, \ldots, N$ we set $c_i(e) := \rho \cdot c_1(e) + \sqrt{1-\rho^2} \cdot \text{rnd}(1, 10000)$, where ρ is a parameter that tunes the amount of correlation with c_1. In our experiments, we use $\rho = 0.0001$ such that c_i ensures very small correlation with c_1. Finally, we generated 100 random problem instances for each number of objectives.

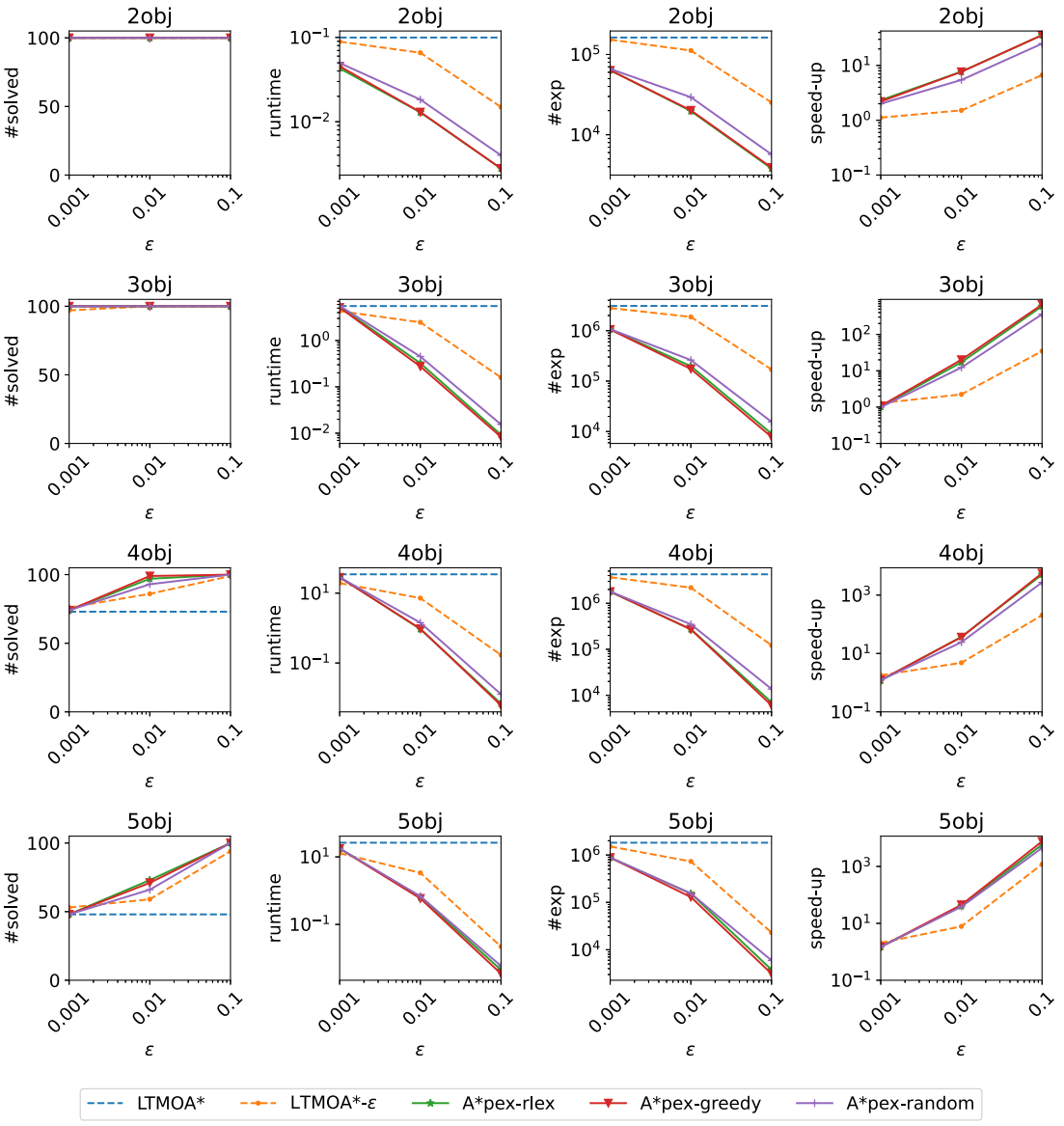


Fig. 10. Results for LTMOA*, LTMOA*-ϵ, and different variants of A*pex on road-network problem with uncorrelated random edge costs with different numbers of objectives and approximation factors.

Fig. 10 shows the numbers of solved problem instances, average runtimes (in seconds), and average numbers of node expansions of all algorithms and the speed-ups of LTMOA*-ϵ and different variants of A*pex over LTMOA* in average runtime. All average values are taken over problem instances solved by all algorithms within the runtime limit of five minutes for all ϵ-values. All variants of A*pex have a smaller average number of node expansions than LTMOA* and LTMOA*-ϵ in all cases. However, for three to five objectives, the average runtime of A*pex is larger than LTMOA*-ϵ and similar to LTMOA* for $\epsilon = 0.001$. This is due to the runtime overhead of A*pex in iterating over OPEN and merging apex-path pairs. In all other cases, the average runtime of all A*pex variants is smaller than those of LTMOA* and LTMOA*-ϵ (including the results for two objectives, which is consistent with the results of Fig. 3). For all numbers of objectives, the behavior of the algorithms is similar to the previous results: The speed-ups of LTMOA*-ϵ and different variants of A*pex over LTMOA* increase as the value of ϵ increases. However, the speed-ups of LTMOA*-ϵ are much less substantial than the speed-ups of the different variants of A*pex for larger ϵ-values. The results also show that A*pex-greedy slightly outperforms the other two A*pex variants. The number of solved instances of A*pex-greedy is at least as high as the one of A*pex-rlex and slightly better than number of solved instances of A*pex-random.

In summary, similar to the results shown in previous sections, all versions of A*pex outperform LTMOA* and LTMOA*-ϵ, and A*pex-greedy is slightly better than the other versions of A*pex in maps with uncorrelated random edge costs.

Table 1
Results for different algorithms on 50 problem instances on the NY road network with three objectives.

	$\#_{solved}$	time (s)	$\#_{exp}$
LTMOA*	47	0.31	331K
A-A*pex-reuse $\eta = 1.5$	35	6.31	2636K
A-A*pex-reuse-enh, $\eta = 1.5$	**40**	1.18	**423K**
A-A*pex-restart, $\eta = 1.5$	38	3.61	5539K
A-A*pex-hybrid, $\eta = 1.5$	**40**	**0.84**	1208K
A-A*pex-reuse $\eta = 2$	36	3.51	1779K
A-A*pex-reuse-enh, $\eta = 2$	40	1.04	**396K**
A-A*pex-restart, $\eta = 2$	39	2.00	3417K
A-A*pex-hybrid, $\eta = 2$	**41**	**0.65**	878K
A-A*pex-reuse $\eta = 4$	38	1.75	1121K
A-A*pex-reuse-enh, $\eta = 4$	40	0.78	**378K**
A-A*pex-restart, $\eta = 4$	40	1.11	1866K
A-A*pex-hybrid, $\eta = 4$	**42**	**0.59**	753K
A-A*pex-reuse $\eta = 8$	39	1.19	908K
A-A*pex-reuse-enh, $\eta = 8$	40	0.81	**367K**
A-A*pex-restart, $\eta = 8$	**42**	0.78	1307K
A-A*pex-hybrid, $\eta = 8$	**42**	**0.41**	515K

7.2. Evaluation of A-A*pex

We first compare different variants of A-A*pex. We then compare A-A*pex with state-of-the-art multi-objective search algorithms.

7.2.1. Metrics

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

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

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

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

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

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

$$e(\Pi) = \max_{\pi \in \Pi^*} e(\Pi, \pi). \tag{4}$$

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

In our experiments, there are problem instances where no algorithm finds the entire Pareto frontier within the runtime limit. When computing the approximation error using Eq. (4) in these cases, we use the set of undominated solutions that all algorithms computed as a substitution for Π^*.

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

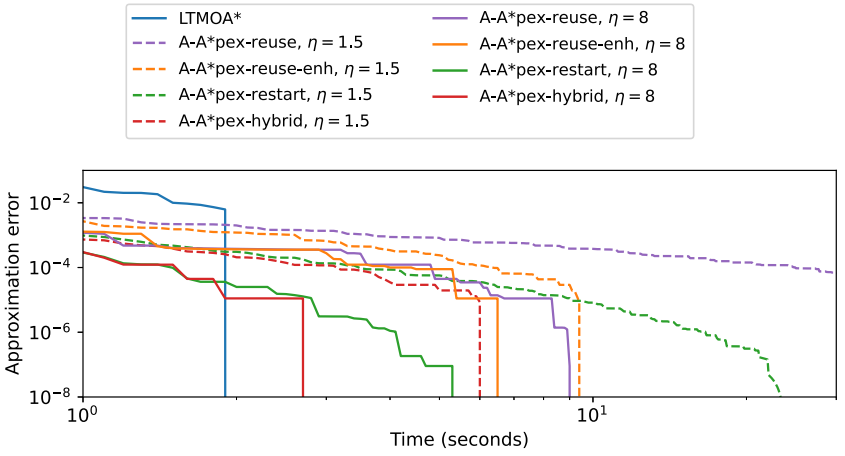
7.2.2. Comparing different variants of A-A*pex

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

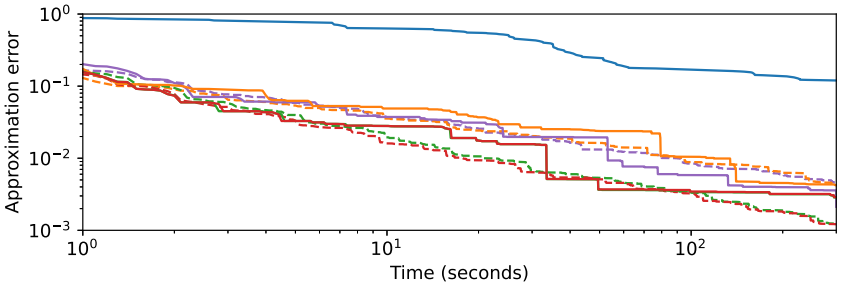
1. A-A*pex-reuse always reuses its search effort and is our baseline variant of A-A*pex.
2. A-A*pex-reuse-enh always reuses its search effort and also uses the enhanced dominance checks.
3. A-A*pex-restart always restarts the search from scratch.
4. A-A*pex-hybrid initially restarts the search from scratch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

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

Table 1 shows the numbers of solved problem instances (i.e., the number of instances for which an algorithm finds the entire Pareto frontier within the runtime limit), average runtimes (in seconds), and average numbers of expanded nodes for all algorithms. All averages are over those instances that all algorithms solve. LTMOA* has the largest number of solved instances and the smallest



(a) The 35 instances solved by all algorithms.



(b) The other 15 instances.

Fig. 11. Approximation error as a function of the runtime for different algorithms on NY with three objectives.

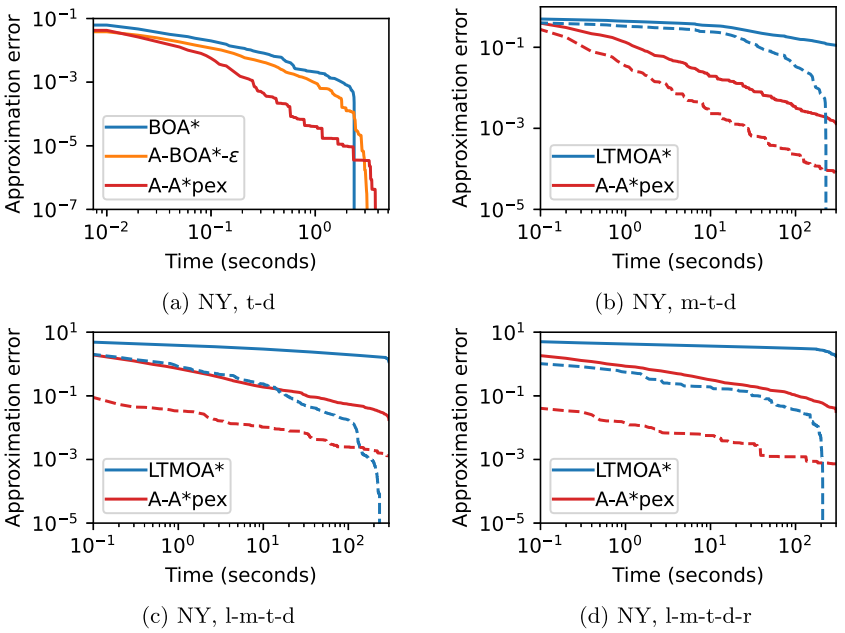
average runtime and number of expanded nodes. The average runtime of each variant of A-A*pex decreases as η increases because larger η results in fewer iterations of A-A*pex. Adding enhanced dominance checks decreases the average runtime of A-A*pex-reuse and results in the smallest node expansions of all A-A*pex variants. With a small η-value, e.g., 1.5 and 2, A-A*pex-restart has a larger runtime than A-A*pex-reuse-enh because restarting from scratch with small ε-values is time-consuming. For all η-values, A-A*pex-hybrid outperforms the other three A-A*pex variants in terms of the number of solved instances and average runtime.

Fig. 11 shows the approximation error as a function of the runtime for LTMOA* and all A-A*pex variants with $\eta = 1.5$ and $\eta = 8$. We use only two η-values to keep the figure clean. We divide the instances into two groups, namely the instances solved by all algorithms (Fig. 11a) and the other instances (Fig. 11b). The approximation error of each algorithm is averaged over all instances in a group. A-A*pex-reuse and A-A*pex-reuse-enh have larger approximation errors than A-A*pex-restart and A-A*pex-hybrid at the beginning of the search for both η-values, which shows that restarting the search from scratch is more efficient in the earlier iterations. In Fig. 11a, the approximation error of A-A*pex-reuse, A-A*pex-reuse-enh, and A-A*pex-hybrid quickly drops in the later iterations, which shows that reusing search effort is more efficient in the later iterations. Therefore, by mixing these two techniques, A-A*pex-hybrid often finds the Pareto frontier faster than the other variants of A-A*pex. Fig. 11b shows that, for all variants of A-A*pex, using a smaller η-value of 1.5 (dashed lines) leads to more frequent updates than using an η-value of 8 (solid lines) and hence results in better anytime behavior for difficult problem instances. For example, in Fig. 11b, if we stop A-A*pex-hybrid with $\eta = 1.5$ and $\eta = 8$ at any time point, the solution set found by A-A*pex-hybrid with $\eta = 1.5$ is much more likely to have a smaller approximation error. In Fig. 11b, all variants of A-A*pex have a smaller approximation error than LTMOA* for the entire five minutes of runtime for both η-values. Although we expect LTMOA* to find Pareto frontiers faster than A-A*pex if sufficient runtime is provided, all variants of A-A*pex often compute solution sets with approximation errors smaller than 0.01 faster than LTMOA*.

7.2.3. Comparing with state of the art

We compare the hybrid variant of A-A*pex with $\eta = 4$ with the state-of-the-art multi-objective search algorithms BOA* [36] and A-BOA*-ϵ [45] on problem instances with two objectives and LTMOA* on problem instances with more than two objectives. We use the same problem instances in the NY road network that are used in Sections 7.1.1 and 7.1.2.

Fig. 12 shows the results for different objectives. We use solid lines for all problem instances and dashed lines for those problem instances whose entire Pareto frontiers are computed within the runtime limit. Fig. 12a contains only solid lines because the entire Pareto frontier for every problem instance has been computed within the runtime limit. In all cases, A-A*pex reduces the approximation error faster than the other algorithms at the beginning of the search. Because LTMOA* and BOA* compute solutions



(a) NY, t-d (b) NY, m-t-d

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

Fig. 12. Anytime behaviors of different algorithms on problem instances with different objectives. Each plot shows the approximation error as a function of the runtime for each algorithm over all 100 problem instances (solid line) and over only those problem instances solved by at least one algorithm (dashed line).

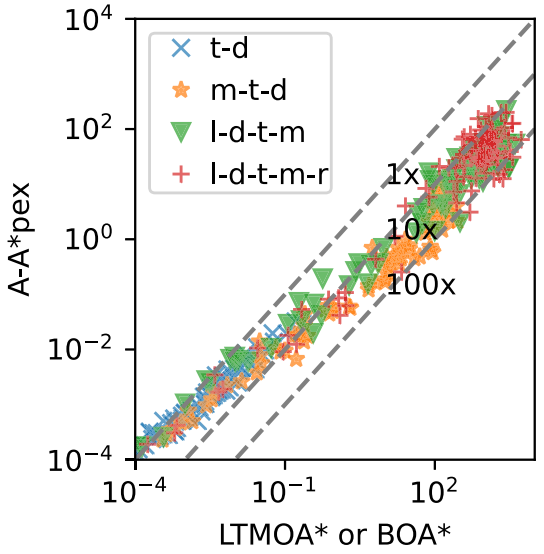


Fig. 13. AUCs for LTMOA* (or BOA* for two objectives) and A-A*pex on all problem instances.

in lexicographically increasing order of their costs, they can exactly "cover" part of the Pareto frontier while completely missing the rest during most part of the search, which explains their high approximation errors at the beginning. This behavior is undesirable from the perspective of approximating the entire Pareto frontier when a limited deliberation time is given. When a sufficient runtime is provided, LTMOA* and BOA* find the Pareto frontier faster than A-A*pex and hence have smaller approximation errors than A-A*pex. However, this happens only after the approximation error becomes smaller than 0.01, even smaller than 0.001 in many cases, which means that A-A*pex computes 0.01-approximate frontiers faster than BOA* or LTMOA*. For every solution π, there exists a solution π' in a 0.01-approximate frontier such that π is at most 1% worse than π' for any objective, which is sufficient for many real-world problems.

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

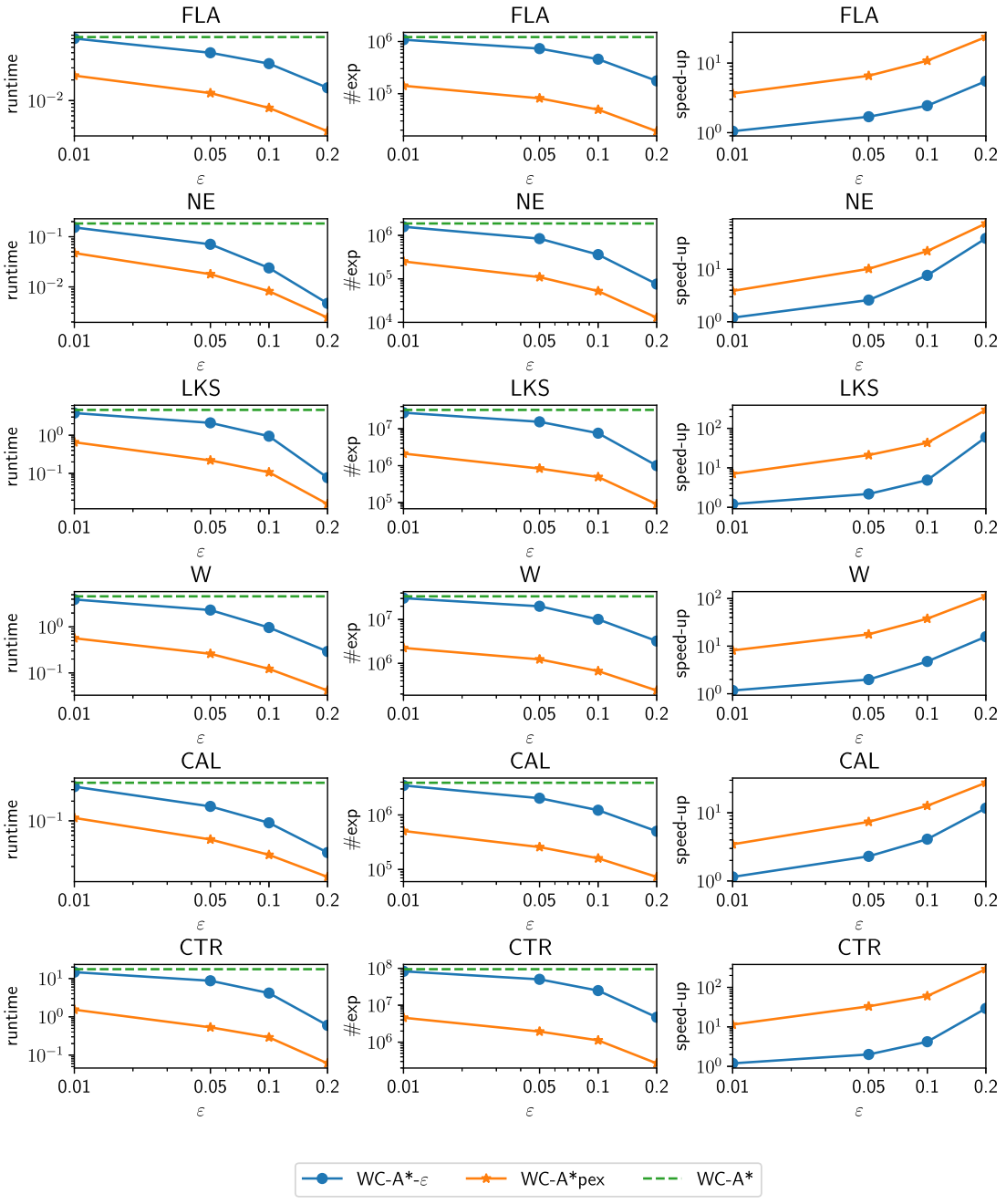


Fig. 14. Results for WC-A*and WC-A*pexon problem instances with different road networks and approximation factors.

runtime. We compare A-A*pex with LTMOA* (or BOA* for two objectives) as the baseline. Fig. 13 shows the results. The numbers along the dashed lines denote how many times the AUC of A-A*pex is smaller than that of the baseline. For instances that are more difficult to solve (the points in the top-right corners), A-A*pex always has a smaller AUC than the baseline. For problem instances with three or more objectives, A-A*pex has up to 100× smaller AUCs than LTMOA*.

*7.3. Evaluation of WC-A*pex*

We evaluate WC-A*pexwith different ε-values and compare the runtimes and numbers of node expansions of WC-A*, WC-A*-ε, and WC-A*pex.

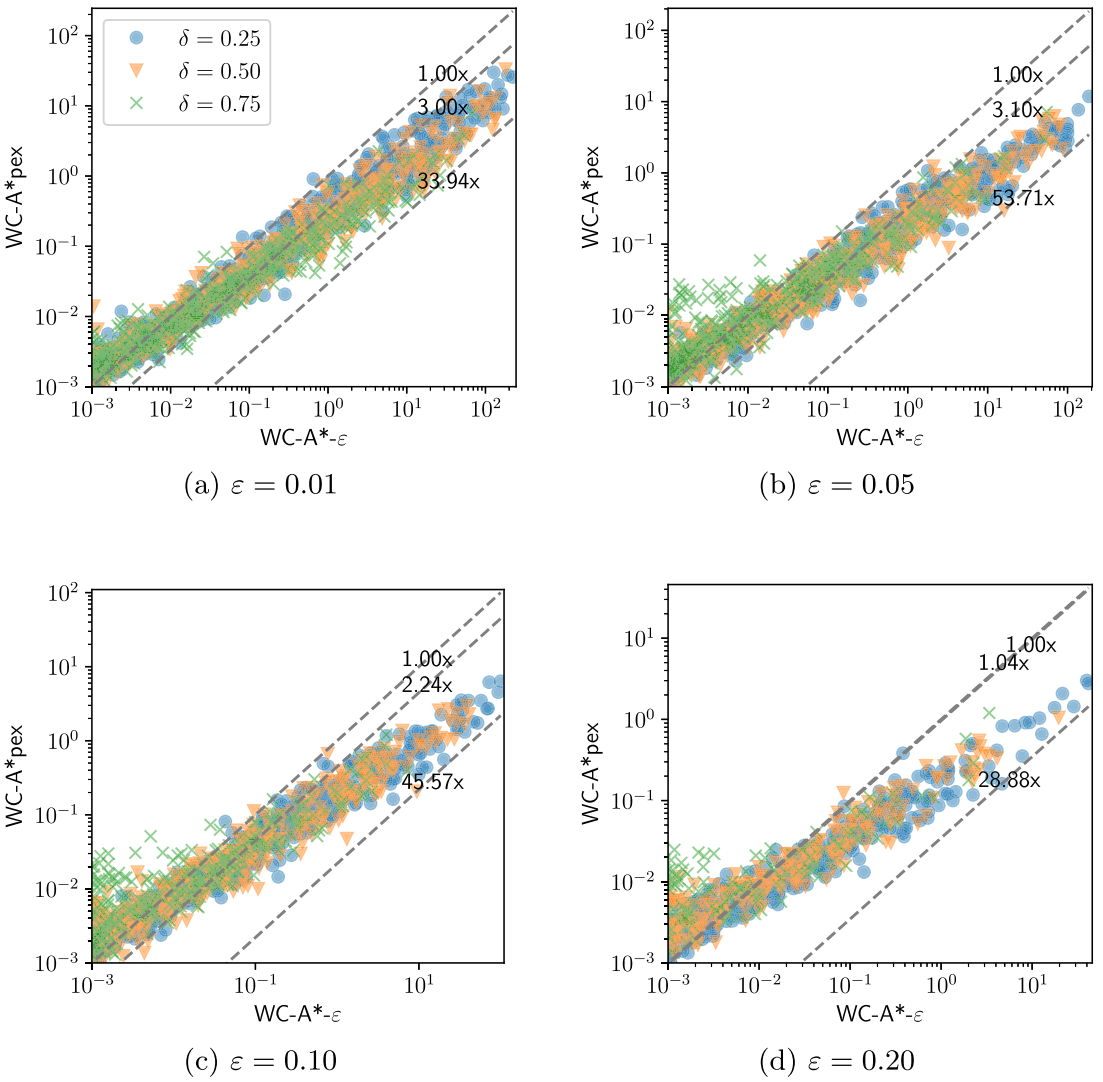


Fig. 15. Runtimes of WC-A*pexand WC-A*-ε with different suboptimality factors on all WCSP instances.

We choose seven road networks, namely FLA (1.1M states and 2.7M edges), NE (1.5M states and 3.9M edges), CAL (1.9M states and 4.7M edges), LKS (2.8M states and 6.9M edges), E (3.6M states and 8.8M edges), W (6.3M states and 15.2M edges), and CTR (14.1M states and 34.3M edges) from the DIMACS data set. The c_1- and c_2-values for each edge are its travel time and distance, respectively, both available from the DIMACS data set. Each WCSP instance thus corresponds to computing a path that is bounded-suboptimal with respect to its travel time and with its travel distance being no larger than a given limit. For each road network, we use the same 100 s_{start} and s_{goal} pairs used by Sedeño-Noda and Colebrook [42] and Ahmadi et al. [35]. Following previous work [16,17], for each s_{start} and s_{goal} pair, we generate a WCSP instance with the weight limit $W = c_2^{\text{lb}} + \delta \cdot (c_2^{\text{ub}} - c_2^{\text{lb}})$ based on a *tightness factor* δ, where c_2^{lb} and c_2^{ub} are the minimum and maximum c_2-values of all Pareto-optimal paths from s_{start} to s_{goal}, respectively. A smaller δ-value thus corresponds to a tighter weight limit. For each s_{start} and s_{goal} pair, we use the tightness factors $0.25, 0.5$, and 0.75. Therefore, we have 300 WCSP instances for each road network.

For each WCSP instance, we evaluate WC-A*-ε and WC-A*pexwith the ε-values 0.01, 0.05, and 0.1. Fig. 14 shows the average runtimes (in seconds) and the numbers of node expansions of WC-A*, WC-A*-ε, and WC-A*pexover all WCSP instances. We choose not to show the results for different tightness factors (δ-values) separately because, as we will show in Fig. 15, they do not affect the results significantly. With $\varepsilon = 0.01$, i.e., a guaranteed suboptimality of at most 1%, the average speed-up of WC-A*pexover WC-A*is more than $11\times$ on the largest road network (CTR). However, the average speed-up of WC-A*-ε with $\varepsilon = 0.01$ over WC-A*is only about 20% because WC-A*-ε still needs to expand a large number of nodes to prove that the incumbent solution is bounded-suboptimal. The runtimes and numbers of node expansions of WC-A*pexare always smaller than the ones of WC-A*and WC-A*-ε with the same ε-value on all road networks, which shows that merging apex-path pairs greatly reduces the runtimes and numbers of node expansions.

Fig. 15 shows the individual runtimes (in seconds) of WC-A*pexand WC-A*-ε for all WCSP instances and ε-values. We use different markers for different tightness factors δ used to generate the WCSP instances. The diagonal dashed lines and the numbers along them denote different speed-ups (1×, 10×, and the maximum speed-up) of WC-A*pexover WC-A*-ε . For different tightness factors, the trends of the speed-ups of WC-A*pexover WC-A*-ε are similar. Although WC-A*pexis slower than WC-A*-ε on easy WCSP instances (which both algorithms solve mostly within around 0.1 seconds), WC-A*pexachieves significant speed-ups over WC-A*-ε on more difficult instances (represented by the points on the top-right corners).

8. Conclusions

In this paper, we presented A*pex, a multi-objective search algorithm that finds an ε-approximate Pareto-optimal solution set for a user-provided approximation factor ε. Building upon A*pex, we present (1) A-A*pex, an anytime variant of A*pex that computes an initial approximate frontier quickly and then works to find better approximate frontiers until eventually finding the entire Pareto frontier, and (2) WC-A*pex, a variant of A*pex for approximately solving the Weight-Constrained Shortest Path (WCSP) problem. Our experimental results show that (1) A*pex and WC-A*pexcan outperform respective state-of-the-art algorithms by orders of magnitude and that (2) A-A*pex often computes better approximate frontiers than state-of-the-art algorithms, given a limited runtime.

We believe that A*pex can be used as an algorithmic building block to address other problems in multi-objective search. For example, Salzman et al. [37] identified that multi-valued heuristics, or MVH,[3] have the potential to dramatically enhance multi-objective search algorithms but efficiently computing effective MVHs is an open challenge. Running a backward search using A*pex can be a first step in that direction. Another line of work that can benefit from A*pex is in the setting where regions of the graph exhibit high correlation. If such regions are identified, then A*pex can be used to "compress' paths that cross a region. A proof of concept for this idea was already demonstrated for two objectives [43] but it is not clear how to extend this to multiple objectives. Finally, we are interested in using the algorithmic framework presented to address other multi-objective variants such as approximate multi-objective TSP [46] or search problems where the objectives admit some hierarchy [47,48].

CRediT authorship contribution statement

Han Zhang: Writing – review & editing, Writing – original draft, Methodology, Investigation, Formal analysis, Conceptualization; **Oren Salzman:** Writing – review & editing, Writing – original draft, Supervision, Methodology, Investigation, Formal analysis, Conceptualization; **T. K. Satish Kumar:** Writing – review & editing, Formal analysis, Conceptualization; **Ariel Felner:** Writing – review & editing, Supervision, Formal analysis; **Carlos Hernández Ulloa:** Writing – review & editing, Supervision, Methodology; **Sven Koenig:** Writing – review & editing, Supervision, Conceptualization.

Data availability

We will share our source in a web page code If the paper is accepted.

Declaration of competing interest

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

Acknowledgment

The research in the US was supported by the National Science Foundation (NSF) under grant number 1817189, 1837779, 1935712, 2121028, 2112533, and 2321786. The research in Israel was supported by the Israel Science Foundation (ISF) 909/23, the United States-Israel Binational Science Foundation (BSF) grant 2019703, and 2021643, and by the Israeli Ministry of Science & Technology grants number 3-16079 and 3-17385. Finally, the research in Chile supported by the National Center for Artificial Intelligence CENIA FB210017, Financiamiento Basal ANID, and the Centro Ciencia & Vida FB210008, Financiamiento Basal ANID.

Appendix A

A.1. Theoretical results for A*pex

In this section, we prove the completeness and correctness of A*pex. Theorem 1 shows that A*pex terminates in finite time and computes an ε-approximate Pareto frontier.

Lemma 1. *A*pex adds only ε-bounded apex-path pairs to* OPEN.

[3] Roughly speaking, an MVH associates multiple heuristic functions with each state.

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

1. On Line 22, A*pex adds a merged apex-path pair $\mathcal{AP}_{\text{new}}$ to OPEN. Apex-path pair $\mathcal{AP}_{\text{new}}$ is ϵ-bounded because of the condition on Line 20.
2. On Line 24, A*pex adds an apex-path pair $\mathcal{AP}$ to OPEN. $\mathcal{AP}$ extends the extracted apex-path pair and is ϵ-bounded because of Property 2.

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

Lemma 2. *The sequence of extracted apex-path pairs has monotonically non-decreasing f_1-values.*

Proof. Consider an apex-path pair $\mathcal{AP}$ extracted by A*pex from OPEN. $\mathcal{AP}$ has the smallest f_1-value of all apex-path pairs in OPEN because its $\mathbf{f}$-value is the lexicographically smallest one. Since the f_1-value of an apex-path pair is no smaller than that of its parent apex-path pair (because the heuristic is consistent) and an apex-path pair resulting from a merge operation cannot have an f_1-value smaller than the f_1-values of both of the merged apex-path pairs, the f_1-values of apex-path pairs that are added to OPEN when expanding $\mathcal{AP}$ can only be no larger than $f_1(\mathcal{AP})$. Therefore, the f_1-value of the apex-path pair extracted in the next iteration can only be no larger than $f_1(\mathcal{AP})$. □

Lemma 3. *There exists a truncated cost in $\mathbf{C}^{tr}_{sol}$ that ϵ-dominates the truncated $\mathbf{f}$-value of some apex-path pair $\mathcal{AP}$ on Line 23 iff there exists a solution in SOLS whose cost ϵ-dominates the $\mathbf{f}$-value of apex-path pair $\mathcal{AP}$.*

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

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

$$Tr(\mathbf{c}(\pi_{\text{sol}})) \preceq_{\epsilon} Tr(\mathbf{f}(\mathcal{AP})). \tag{A.1}$$

According to Lemma 1, $\mathcal{AP}_{\text{sol}}$ is ϵ-bounded. Hence, we have $c_1(\pi_{\text{sol}}) \leq (1+\epsilon) f_1(\mathcal{AP}_{\text{sol}})$. According to Lemma 2 and also because $\mathcal{AP}_{\text{sol}}$ is extracted before $\mathcal{AP}$, $f_1(\mathcal{AP}_{\text{sol}}) \leq f_1(\mathcal{AP})$. Therefore, we have $c_1(\pi_{\text{sol}}) \leq (1+\epsilon) f_1(\mathcal{AP})$. Combining this and Eq. (A.1), we have $\mathbf{c}(\pi_{\text{sol}}) \preceq_{\epsilon} \mathbf{f}(\mathcal{AP})$. Therefore, there exists solution π_{sol} in SOLS whose cost ϵ-dominates the $\mathbf{f}$-value of apex-path pair $\mathcal{AP}$. □

Lemma 4. *There exists a truncated $\mathbf{g}$-value in $\boldsymbol{G}^{\text{tr}}_{\text{cl}}(s(\mathcal{AP}))$ that weakly dominates the truncated $\mathbf{g}$-value of some apex-path pair $\mathcal{AP}$ on Line 25 iff there exists an expanded apex-path pair $\mathcal{AP}'$ that contains state $s(\mathcal{AP})$ and whose $\mathbf{g}$-value weakly dominates the $\mathbf{g}$-value of apex-path pair $\mathcal{AP}$.*

Proof. Assume that there exists an expanded apex-path pair $\mathcal{AP}'$ that contains state $s(\mathcal{AP})$ and whose $\mathbf{g}$-value weakly dominates the one of apex-path pair $\mathcal{AP}$ (that is, $\mathbf{g}(\mathcal{AP}') \preceq \mathbf{g}(\mathcal{AP})$). Because A*pex has added $Tr(\mathbf{g}(\mathcal{AP}'))$ to $\boldsymbol{G}^{\text{tr}}_{\text{cl}}(s(\mathcal{AP}))$ on Line 13, there must exist some vector $\mathbf{x}$ in $\boldsymbol{G}^{\text{tr}}_{\text{cl}}(s(\mathcal{AP}))$ that weakly dominates $Tr(\mathbf{g}(\mathcal{AP}))$.

Assume tha there exists a truncated $\mathbf{g}$-value in $\boldsymbol{G}^{\text{tr}}_{\text{cl}}(s(\mathcal{AP}))$ that weakly dominates the truncated $\mathbf{g}$-value of apex-path pair $\mathcal{AP}$. Let apex-path pair $\mathcal{AP}'$ be the expanded apex-path pair that contains state $s(\mathcal{AP})$ and with which Line 10 was executed to add this truncated $\mathbf{g}$-value to $\boldsymbol{G}^{\text{tr}}_{\text{cl}}(s(\mathcal{AP}))$. Hence, we have $Tr(\mathbf{g}(\mathcal{AP}')) \preceq Tr(\mathbf{g}(\mathcal{AP}))$. It holds that $f_1(\mathcal{AP}') \leq f_1(\mathcal{AP})$ according to Lemma 2. Also because $s(\mathcal{AP}) = s(\mathcal{AP}')$, we have $g_1(\mathcal{AP}') \leq g_1(\mathcal{AP})$. Thus, the $\mathbf{g}$-value of apex-path pair $\mathcal{AP}'$ weakly dominates the one of apex-path pair $\mathcal{AP}$. □

Lemma 5. *If the apex of an apex-path pair weakly dominates a vector and the apex-path pair is merged with another apex-path pair, then the apex of the merged apex-path pair weakly dominates the vector as well.*

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

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

Lemma 6. *Consider any solution π_{sol} and let $[s_1, s_2 \dots s_L]$ denote the sequence of states it traverses. At the beginning of each iteration of A*pex (that is, before executing Line 7), if A*pex has expanded (that is, reaching Line 11 with) an apex-path pair $\mathcal{AP}$ with $\mathbf{g}(\mathcal{AP}) \preceq \mathbf{c}(\pi^{(j)}_{sol})$ and $s(\mathcal{AP}) = s_j$ for some j, then there exists (1) an apex-path pair $\mathcal{AP}'$ in OPEN with $\mathbf{g}(\mathcal{AP}') \preceq \mathbf{c}(\pi^{(k)}_{sol})$ and $s(\mathcal{AP}') = s_k$ for some $k > j$ or (2) a solution in SOLS that ϵ-dominates π_{sol}.*

Proof. We prove this lemma by induction on j, starting with $j = L$ and going backward. If A*pex has expanded an apex-pair $\mathcal{AP}$ with $\mathbf{g}(\mathcal{AP}) \preceq \mathbf{c}(\pi^{(L)}_{\text{sol}})$ and $s(\mathcal{AP}) = s_L(= s_{\text{goal}})$, A*pex has added the representative path of $\mathcal{AP}$ to SOLS on Line 14. Because $\mathcal{AP}$ is ϵ-bounded (according to Lemma 1), the cost of its representative path ϵ-dominates its $\mathbf{f}$-value $\mathbf{f}(\mathcal{AP})$ (which is equal to $\mathbf{g}(\mathcal{AP})$ because $\mathbf{h}(s(\mathcal{AP})) = \mathbf{0}$) and hence ϵ-dominates $\mathbf{c}(\pi^{(L)}_{\text{sol}})$. Because A*pex only removes a solution from SOLS if it is weakly dominated by a new

solution (Line 12), there must exist some solution in SOLS that ϵ-dominates π_{sol} when A*pex reaches the beginning of all following iterations. Therefore, the lemma holds for $j = L$.

Assume that the lemma holds for $j = l + 1, l \leq L - 1$, and also assume that A*pex has expanded an apex-path pair $\mathcal{AP}$ with $\mathbf{g}(\mathcal{AP}) \preceq \mathbf{c}(\pi^{(l)}_{\text{sol}})$ and $s(\mathcal{AP}) = s_l$. Consider the iteration in which $\mathcal{AP}$ was expanded and the child apex-path pair $\mathcal{AP}' = \langle \boldsymbol{A}', \pi' \rangle$ of $\mathcal{AP}$ created on Line 17 for the lth edge e in π_{sol}, which is from state s_l to states_{l+1}. Apex-path pair $\mathcal{AP}'$ contains state s_{l+1}, and its apex weakly dominates the cost of path $\pi^{(l+1)}_{\text{sol}}$, because

$$\begin{aligned} \boldsymbol{A}' &= \boldsymbol{A} + \mathbf{c}(e) \\ &\preceq \mathbf{c}(\pi^{(l)}_{\text{sol}}) + \mathbf{c}(e) \\ &= \mathbf{c}(\pi^{(l+1)}_{\text{sol}}). \end{aligned} \tag{A.2}$$

We distinguish the following cases:

1. Apex-path pair $\mathcal{AP}'$ was pruned because the condition on Line 23 held. According to Lemma 3, there existed some solution in SOLS whose cost ϵ-dominated the $\mathbf{f}$-value of $\mathcal{AP}'$, which in turn weakly dominated the cost of π_{sol} (because the heuristic is consistent). There must exist some solution in SOLS that ϵ-dominates solution π_{sol} when A*pex reaches the beginning of all following iterations. Thus the lemma holds for $j = l$.
2. Apex-path pair $\mathcal{AP}'$ was pruned because the condition on Line 25 held, namely, there existed a truncated $\mathbf{g}$-value in $G^T_{\text{cl}}(s(\mathcal{AP}'))$ that weakly dominated the truncated $\mathbf{g}$-value of $\mathcal{AP}'$. Then, according to Lemma 4, A*pex had expanded an apex-path pair $\mathcal{AP}''$ that contained state s_{l+1} and whose apex weakly dominated the apex of $\mathcal{AP}'$ (and hence the cost of path $\pi^{(l+1)}_{\text{sol}}$). The lemma holds for $j = l$ because it holds for $j = l + 1$.
3. Otherwise, A*pex executed Line 20 to add apex-path pair $\mathcal{AP}'$ to OPEN, perhaps after merging it with another apex-path pair on Line 19. A*pex might merge it several (more) times with other apex-path pairs on Line 19. The apex of the resulting apex-path pair weakly dominates $\mathbf{c}(\pi^{(l+1)}_{\text{sol}})$ because of Eq. (A.2) and Lemma 5. When A*pex reaches the beginning of an iteration again, the lemma holds if this apex-path pair is still in OPEN. Otherwise, this apex-path pair has been extracted from OPEN in a previous iteration. If this apex-path pair was pruned because of the condition on Lines 23 or 25, as we have already proved, the lemma holds. Otherwise, this apex-path pair has been expanded, and hence the lemma holds for $j = l$ because it holds for $j = l + 1$.

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

Lemma 7. *Consider any solution π_{sol} and let $[s_1, s_2 \ldots s_L]$ denote the sequence of states it traverses. At the beginning of each iteration of A*pex, there always exists (1) an apex-path pair $\mathcal{AP}$ in OPEN that satisfies $\mathbf{g}(\mathcal{AP}) \preceq \mathbf{c}(\pi^{(j)}_{sol})$ and $s(\mathcal{AP}) = s_j$ for some j or (2) a solution in SOLS that ϵ-dominates π_{sol}.*

Proof. Consider the first apex-path pair that A*pex adds to OPEN, that is, apex-path pair $\mathcal{AP}_0 = \langle \mathbf{0}, [s_{\text{start}}] \rangle$. At the beginning of the first iteration of A*pex, the Lemma holds because $\mathcal{AP}_0$ satisfies $\mathbf{g}(\mathcal{AP}_0) \preceq \mathbf{c}(\pi^{(1)}_{\text{sol}})$ (because both $\mathbf{g}(\mathcal{AP}_0)$ and $\mathbf{c}(\pi^{(1)}_{\text{sol}})$ are $\mathbf{0}$) and $s(\mathcal{AP}_0) = s_1$. A*pex then expands $\mathcal{AP}_0$. The lemma holds for all future iterations according to Lemma 6. □

Lemma 8. *A*pex does not expand an apex-path pair $\mathcal{AP}$ if there exists an solution π_{sol} with $\mathbf{c}(\pi_{sol}) \prec \mathbf{f}(\mathcal{AP})$.*

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

1. There exists an apex-path pair $\mathcal{AP}'$ in OPEN that satisfies $\mathbf{g}(\mathcal{AP}') \preceq \mathbf{c}(\pi^{(j)}_{\text{sol}})$ and $s(\mathcal{AP}') = s_j$ for some j. $\mathbf{f}(\mathcal{AP}') = \mathbf{g}(\mathcal{AP}') + \mathbf{h}(s(\mathcal{AP}')) \preceq \mathbf{c}(\pi_{\text{sol}})$ because $\mathbf{g}(\mathcal{AP}') \preceq \mathbf{c}(\pi^{(j)}_{\text{sol}})$ and $\mathbf{h}$ is consistent. Hence, we have $\mathbf{f}(\mathcal{AP}') \preceq \mathbf{c}(\pi_{\text{sol}}) \prec \mathbf{f}(\mathcal{AP})$. A*pex would not extract $\mathcal{AP}$ from OPEN while $\mathcal{AP}'$ is in OPEN because this contradicts that A*pex extracts the apex-path pair with the lexicographically smallest $\mathbf{f}$-value. Therefore, this case cannot happen.
2. There exists a solution π'_{sol} in SOLS that ϵ-dominates π_{sol}. $\mathbf{c}(\pi'_{\text{sol}})$ ϵ-dominates $\mathbf{c}(\pi_{\text{sol}})$, which in turn dominates $\mathbf{f}(\mathcal{AP})$. Therefore, A*pex will prune $\mathcal{AP}$ on Line 9 because of the condition on Line 23 and also according to Lemma 3.

□

Theorem 1. *A*pex terminates in finite time and for any solution π_{sol}, there exists, when A*pex returns, a solution in SOLS that ϵ-dominates solution π_{sol}.*

Proof. Consider any solution π_{sol} and any expanded apex-path pair $\mathcal{AP} = \langle \boldsymbol{A}, \pi \rangle$. Because A*pex generates only ϵ-bounded apex-path pairs (according to Lemma 1), $\mathbf{f}(\pi)$ must ϵ-dominate $\mathbf{f}(\mathcal{AP})$ (that is, $c_i(\pi) + h_i(s(\mathcal{AP})) \leq (1 + \epsilon) f_i(\mathcal{AP})$ for all i). According to Lemma 8, $\mathbf{f}(\mathcal{AP})$ is not dominated by $\mathbf{c}(\pi^{(1)}_{\text{sol}})$, and hence $f_i(\mathcal{AP}) \leq c_i(\pi_{\text{sol}})$ must hold for some i. Therefore, $c_i(\pi) + h_i(s(\mathcal{AP})) \leq (1 + \epsilon) c_i(\pi_{\text{sol}})$ must hold for some i. Because heuristic $\mathbf{h}$ is non-negative, $c_i(\pi) \leq (1 + \epsilon) c_i(\pi_{\text{sol}})$ must hold for some i for the representative path π of any expanded apex-path pair. Because the graph is finite and has positive edge costs, one can extend a path for finite times before the resulting path π does not satisfy $c_i(\pi) < (1 + \epsilon) c_i(\pi_{\text{sol}})$ for all i. Thus, there are only a finite number of representative paths that A*pex can expand (and generate). A*pex returns in finite time.

Consider the beginning of the last iteration of A*pex before it returns, where OPEN becomes empty. According to Lemma 7, for any solution π_{sol}, there exists a solution in SOLS that ϵ-dominates π_{sol}. □

A.2. Theoretical results for A-A*pex

This section provides theoretical results for A-A*pex. We study only the variant of A-A*pex that reuses its previous search effort because all theorems in this section trivially hold for the variant of A-A*pex that restarts the search from scratch. Theorem 2 shows that A-A*pex computes an ϵ_{curr}-approximate frontier in each iteration. Theorem 3 shows that A-A*pex eventually computes a Pareto frontier.

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

Lemma 9. *Consider any solution π_{sol} and let $[s_1, s_2 \dots s_L]$ denote the sequence of states it traverses. If `findApproxPF` expands (that is, reaches Line 26 with) an apex-path pair $\mathcal{AP}$ whose representative path is l-compatible with π_{sol} for some l then there exists, when `findApproxPF` terminates, (Case 1) a path in* PRUNED *that is compatible with π_{sol} or (Case 2) a solution in* SOLS *that weakly dominates π_{sol}.*

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

Assume that the lemma holds for $l+1$ and consider the case where `findApproxPF` expands an apex-path pair $\mathcal{AP}$ whose representative path π is l-compatible with π_{sol}. Consider the child apex-path pair $\mathcal{AP}_{\text{ch}} = \langle \boldsymbol{A}_{\text{ch}}, \pi_{\text{ch}} \rangle$ of $\mathcal{AP}$ that `findApproxPF` generates for edge $\langle s_l, s_{l+1} \rangle$ when reaching Line 32. π_{ch} is $(l+1)$-compatible with π_{sol} because the last state of π_{ch} is s_{l+1} and $\mathbf{c}(\pi_{\text{ch}}) = \mathbf{c}(\pi) + \mathbf{c}(\langle s_l, s_{l+1} \rangle) \preceq \mathbf{c}(\pi_{\text{sol}}^{(l)}) + \mathbf{c}(\langle s_l, s_{l+1} \rangle) = \mathbf{c}(\pi_{\text{sol}}^{(l+1)})$. We distinguish the following two cases:

1. `findApproxPF` prunes $\mathcal{AP}_{\text{ch}}$ on Line 24 or 34 because the `isDominated` function returns true. `isDominated` returns true only when it reaches Line 6 or 11 of Algorithm 5. If π_{ch} is added to PRUNED, Case 1 holds. If not and `isDominated` reaches Line 6 without adding π_{ch} to PRUNED, then there exists a solution, that is, solution π' mentioned on Line 3, whose cost weakly dominates $\mathbf{f}(\pi_{\text{ch}})$ and whose truncated cost is in $\mathbf{C}_{\text{sol}}^{\text{tr}}$. $\mathbf{f}(\pi_{\text{ch}})$ weakly dominates $\mathbf{c}(\pi_{\text{sol}})$ because $\mathbf{c}(\pi_{\text{ch}}) \preceq \mathbf{c}(\pi_{\text{sol}}^{(l+1)})$ and $\mathbf{h}$ is consistent. `findApproxPF` adds truncated cost vectors to $\mathbf{C}_{\text{sol}}^{\text{tr}}$ only on Lines 21 and 28 for solutions in SOLS. Therefore, π' has been in SOLS. Because `findApproxPF` removes a solution from SOLS only when adding another solution that weakly dominates it, there must exist a solution in SOLS that weakly dominates $\mathbf{c}(\pi_{\text{sol}})$ when `findApproxPF` terminates. Case 2 holds. If `isDominated` reaches Line 11 without adding π_{ch} to PRUNED, then there exists an expanded apex-path pair (namely, the one mentioned on Line 8) that contains state s_{l+1} and whose representative path π' weakly dominates π_{ch} (and hence is $(l+1)$-compatible with π_{sol}). The lemma holds for l because it holds for $l+1$.
2. `findApproxPF` calls `addToOpen` with $\mathcal{AP}_{\text{ch}}$. The algorithm might merge $\mathcal{AP}_{\text{ch}}$ with other apex-path pairs before extracting the resulting apex-path pair of these merges, denoted as $\mathcal{AP}''$, from OPEN. During these merges, a representative path is completely discarded (i.e., neither chosen as the new representative path nor added to PRUNED) only if it is weakly dominated by the other representative path. Therefore, if no path that weakly dominates π_{ch} is added to PRUNED during these merges, the representative path of $\mathcal{AP}''$ must weakly dominate π_{ch} (and hence is $(l+1)$-compatible with π_{sol}). If $\mathcal{AP}''$ is pruned on Line 24, the lemma holds as we have already proved. Otherwise, $\mathcal{AP}''$ is expanded. The lemma holds for l because it holds for $l+1$.

□

Lemma 10. *For any solution π_{sol}, when A-A*pex reaches Line 4 of Algorithm 4, there exists (Case 1) a path compatible with π_{sol} in* PRUNED *or (Case 2) a solution in* SOLS *that weakly dominates π_{sol}.*

Proof. We prove this lemma by induction. When A-A*pex reaches Line 4 for the first time, path $[s_{\text{start}}]$ in PRUNED is compatible with π_{sol}, and hence the lemma holds. Assume that A-A*pex reaches Line 4 and the lemma has held so far. If there exists a solution in SOLS that weakly dominates π_{sol}, there must exist a solution in SOLS that weakly dominates π_{sol} when A-A*pex reaches Line 4 again because A-A*pex only removes a solution from SOLS when adding another solution that weakly dominates it (Line 27). Otherwise, there exists a path π' in PRUNED that is compatible with π_{sol}. A-A*pex then calls `addToOpen` with apex-path pair $\langle \mathbf{c}(\pi'), \pi' \rangle$ on Line 8 and might merge it with other apex-path pairs before `findApproxPF` extracts the resulting apex-path pair $\mathcal{AP}''$ from OPEN. As we have already proved, if the algorithm does not add a path that weakly dominates π' to PRUNED during these merges, the representative path π'' of $\mathcal{AP}''$ must weakly dominate π' (and hence is compatible with π_{sol}). If $\mathcal{AP}''$ is pruned on Line 24, we can distinguish the following cases:

1. Path π'' is added to PRUNED. The lemma holds.
2. `isDominated` returns true on Line 6 without adding π'' to PRUNED. There exists a solution in SOLS whose cost weakly dominates $\mathbf{f}(\pi'')$, which in turn weakly dominates $\mathbf{c}(\pi_{\text{sol}})$ because π'' is compatible with π_{sol} and $\mathbf{h}$ is consistent. Therefore, when A-A*pex reaches Line 4 again, there still exists a solution in SOLS whose cost weakly dominates $\mathbf{c}(\pi_{\text{sol}})$. The lemma holds.
3. `isDominated` returns true on Line 11 without adding π'' to PRUNED. There exists an expanded apex-path pair that contains state $s(\mathcal{AP}'')$ and whose representative path weakly dominates π'' (and hence is compatible with π_{sol}). According to Lemma 9, Case 1 or 2 holds when `findApproxPF` terminates and hence also holds when A-A*pex reaches Line 4 again. The lemma holds.

Otherwise, $\mathcal{AP}''$ is expanded. From Lemma 9, the lemma holds. □

Lemma 11. *A-A*pex adds only ϵ_{curr}-bounded apex-path pairs to OPEN.*

Proof. Consider the apex-path pairs that A-A*pex calls `addToOpen` with on Line 8. These apex-path pairs are ϵ_{curr}-bounded because the f-values of their representative paths are equal to (and hence weakly dominates) their f-values. The `addToOpen` function (Algorithm 5) either adds an apex-path pair directly to OPEN (Line 23) or merges it with another apex-path pair on condition that the resulting apex-path pair is ϵ_{curr}-bounded (Line 18). Therefore, A-A*pex adds only ϵ_{curr}-bounded apex-path pairs to OPEN on Line 8.

A-A*pex then calls `findApproxPF` on Line 9. When `findApproxPF` reaches Line 19 to extract an apex-path pair from OPEN for the first time, all apex-path pair in OPEN are ϵ_{curr}-bounded. The same induction in the proof for Lemma 1 applies here. Therefore, A-A*pex adds only ϵ_{curr}-bounded apex-path pairs to OPEN on Line 35. □

Lemma 12. *In each run of `findApproxPF`, the sequence of extracted apex-path pairs has monotonically non-decreasing f_1-values.*

The same proof for Lemma 2 applies here.

Lemma 13. *There exists a solution in SOLS whose cost ϵ_{curr}-dominates the* **f***-value of some apex-path pair $\mathcal{AP}$ on Line 2 of the `isDominated` function (Algorithm 5) if there exists a truncated cost in $\mathbf{C}^{tr}_{sol}$ that ϵ_{curr}-dominates the truncated* **f***-value of apex-path pair $\mathcal{AP}$.*

Proof. Assume that there exists a truncated cost vector, denoted as $\mathbf{x}$, in $\mathbf{C}^{\text{tr}}_{\text{sol}}$ that ϵ_{curr}-dominates the truncated $\mathbf{f}$-value of some apex-path pair $\mathcal{AP}$. $\mathbf{x}$ was added to $\mathbf{C}^{\text{tr}}_{\text{sol}}$ on either Line 21 or 28 of Algorithm 4 for some solution. Let π_{sol} denote this solution. We have $Tr(\mathbf{c}(\pi_{\text{sol}})) = \mathbf{x}$ and hence

$$Tr(\mathbf{c}(\pi_{\text{sol}})) \preceq_{\epsilon_{\text{curr}}} Tr(\mathbf{f}(\mathcal{AP})). \tag{A.3}$$

Suppose that $\mathbf{x}$ was added to $\mathbf{C}^{\text{tr}}_{\text{sol}}$ on Line 21. We distinguish the following two cases for $\mathcal{AP}$:

1. $\mathcal{AP}$ is an apex-path pair extracted from OPEN. According to the condition on Line 20, $c_1(\pi_{\text{sol}}) \leq (1+\epsilon_{\text{curr}}) f_1(\mathcal{AP})$.
2. $\mathcal{AP}$ is an apex-path pair generated on Line 32 for some parent apex-path pair $\mathcal{AP}'$. Because the heuristic is consistent, we have $f_1(\mathcal{AP}') \leq f_1(\mathcal{AP})$. According to the condition on Line 20, $c_1(\pi_{\text{sol}}) \leq (1+\epsilon_{\text{curr}}) f_1(\mathcal{AP}') \leq (1+\epsilon_{\text{curr}}) f_1(\mathcal{AP})$.

In both cases, we have $c_1(\pi_{\text{sol}}) \leq (1+\epsilon_{\text{curr}}) f_1(\mathcal{AP})$. Combining this and Eq. (A.3), we have $\mathbf{c}(\pi_{\text{sol}}) \preceq_{\epsilon_{\text{curr}}} \mathbf{f}(\mathcal{AP})$.

Suppose that $\mathbf{x}$ was added to $\mathbf{C}^{\text{tr}}_{\text{sol}}$ on Line 28. Let $\mathcal{AP}_{\text{sol}} = \langle \boldsymbol{A}_{\text{sol}}, \pi_{\text{sol}} \rangle$ denote the apex-path pair with which A*pex reached Line 28 and added $\mathbf{x}$ to $\mathbf{C}^{\text{tr}}_{\text{sol}}$. According to Lemma 11, $\mathcal{AP}_{\text{sol}}$ is ϵ_{curr}-bounded. Hence, we have $c_1(\pi_{\text{sol}}) \leq (1+\epsilon_{\text{curr}}) f_1(\mathcal{AP}_{\text{sol}})$. According to Lemma 12 and also because $\mathcal{AP}_{\text{sol}}$ is extracted before $\mathcal{AP}$, $f_1(\mathcal{AP}_{\text{sol}}) \leq f_1(\mathcal{AP})$. Therefore, we have $c_1(\pi_{\text{sol}}) \leq (1+\epsilon_{\text{curr}}) f_1(\mathcal{AP})$. Combining this and Eq. (A.3), we have $\mathbf{c}(\pi_{\text{sol}}) \preceq_{\epsilon_{\text{curr}}} \mathbf{f}(\mathcal{AP})$.

Therefore, there exists solution π_{sol} in SOLS whose cost ϵ_{curr}-dominates the $\mathbf{f}$-value of apex-path pair $\mathcal{AP}$. □

Lemma 14. *There exists a truncated* **g***-value in $\boldsymbol{G}^{\text{tr}}_{\text{cl}}(s(\mathcal{AP}))$ that weakly dominates the truncated* **g***-value of some apex-path pair $\mathcal{AP}$ on Line 2 of the `isDominated` function (Algorithm 5) iff there exists an expanded apex-path pair $\mathcal{AP}'$ that contains state $s(\mathcal{AP})$ and whose* **g***-value weakly dominates the* **g***-value of apex-path pair $\mathcal{AP}$.*

Lemma 14 is similar to Lemma 4 for A*pex. The proof for Lemma 4 builds upon Lemma 2. After replacing Lemma 2 with Lemma 12, the same proof for Lemma 4 applies for Lemma 14.

Lemma 15. *Consider any solution $\pi_{sol} = [s_1, s_2 \dots s_L]$. At the beginning of each iteration of `findApproxPF` (that is, before executing Line 19), if the same run of `findApproxPF` has expanded an apex-path pair $\mathcal{AP}$ that satisfies $\mathbf{g}(\mathcal{AP}) \preceq \mathbf{c}(\pi^{(j)}_{sol})$ and $s(\mathcal{AP}) = s_j$ for some j, then there exists (1) an apex-path pair $\mathcal{AP}'$ in OPEN that satisfies $\mathbf{g}(\mathcal{AP}') \preceq \mathbf{c}(\pi^{(k)}_{sol})$ and $s(\mathcal{AP}') = s_k$ for some $k > j$ or (2) a solution in SOLS that ϵ_{curr}-dominates π_{sol}*

Lemma 15 is similar to Lemma 6 for A*pex. The proof for Lemma 6 builds upon Lemmas 1, 3, 4, and 5. Lemma 5 is about the property of apex-path pairs and hence applies for `findApproxPF`, too. After replacing Lemmas 1, 3, and 4 with Lemmas 11, 13, and 14, respectively, the same proof for Lemma 4 applies for Lemma 15.

Lemma 16. *Consider any solution $\pi_{sol} = [s_1, s_2 \dots s_L]$. At the beginning of each iteration of `findApproxPF` (that is, before executing Line 19), there always exists (1) an apex-path pair $\mathcal{AP}'$ in OPEN that satisfies $\mathbf{g}(\mathcal{AP}') \preceq \mathbf{c}(\pi^{(j)}_{sol})$ and $s(\mathcal{AP}') = s_j$ for some j or (2) a solution in SOLS that ϵ_{curr}-dominates π_{sol}.*

Proof. A-A*pex reaches Line 4 before executing `findApproxPF`. From Lemma 10, for any solution π_{sol}, there exists a solution in SOLS that weakly dominates π_{sol} or a path in PRUNED that is compatible with π_{sol}. If π_{sol} is weakly dominated by some solutions in SOLS on Line 4, it will always be weakly dominated (and hence also ϵ_{curr}-dominated) by some solutions in SOLS at the beginning of each iteration of `findApproxPF`. If there exists a path π in PRUNED that is compatible with π_{sol} on Line 4. Let j denote the index for which π is j-compatible with π_{sol}. When A-A*pex reaches Line 8 with this path π, it adds an apex-path pair that contains state s_j and whose apex is equal to and hence weakly dominates $\mathbf{c}(\pi)$, which in turn weakly dominates $\mathbf{c}(\pi^{(j)}_{\text{sol}})$, to OPEN. This apex-path pair might be merged several (more) times with other apex-path pairs. The resulting apex-path pair $\mathcal{AP}$ will still satisfy that $\mathbf{g}(\mathcal{AP}) \preceq \mathbf{c}(\pi^{(j)}_{\text{sol}})$ and $s(\mathcal{AP}) = s_j$. Consider the beginning of an iteration of `findApproxPF`. If apex-path pair $\mathcal{AP}$ has not been extracted from OPEN, the lemma holds. If apex-path pair $\mathcal{AP}$ has been extracted from OPEN and was pruned because of the condition on Line 2 of Algorithm 5, from Lemma 13, there exists a solution in SOLS whose cost ϵ_{curr}-dominates the $\mathbf{f}$-value of $\mathcal{AP}$, which in turn weakly dominates

the cost of π_{sol} because $\mathbf{f}(\mathcal{AP}) = \mathbf{g}(\mathcal{AP}) + \mathbf{h}(s_j) \preceq \mathbf{c}(\pi_{\text{sol}}^{(j)}) + \mathbf{h}(s_j)$ and $\mathbf{h}$ is consistent. Thus, the lemma holds. If apex-path pair $\mathcal{AP}$ has been extracted from OPEN and was pruned because of the condition on Line 7 of Algorithm 5, from Lemma 14, `findApproxPF` has expanded an apex-path pair that contains state s_l and whose apex weakly dominates the apex of $\mathcal{AP}$ (and hence weakly dominates the cost of $\pi_{\text{sol}}^{(l)}$). The lemma holds because of Lemma 15. Otherwise, `findApproxPF` has expanded $\mathcal{AP}$. The lemma holds because of Lemma 15. □

Lemma 17. *Consider an apex-path pair $\mathcal{AP}$ that `findApproxPF` extracts from OPEN on Line 19. There exists a solution in SOLS whose cost ϵ_{curr}-dominates the $\mathbf{f}$-value of $\mathcal{AP}$ on Line 2 of the `isDominated` function (Algorithm 5) iff there exists a truncated cost in $\mathbf{C}_{sol}^{tr}$ that ϵ_{curr}-dominates the truncated $\mathbf{f}$-value of $\mathcal{AP}$.*

Proof. From Lemma 13, there exists a solution in SOLS whose cost ϵ_{curr}-dominates the $\mathbf{f}$-value of some apex-path pair $\mathcal{AP}$ on Line 7 of the `isDominated` function (Algorithm 5) if there exists a truncated cost in $\mathbf{C}_{\text{sol}}^{\text{tr}}$ that ϵ_{curr}-dominates the truncated $\mathbf{f}$-value of apex-path pair $\mathcal{AP}$.

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

1. Solutionπ_{sol} is found by a previous run of `findApproxPF`. π_{sol} has been added to SOLS′ on Line 14. Because $c_1(\pi_{\text{sol}}) \leq (1+\epsilon_{\text{curr}}) f_1(\mathcal{AP})$, the truncated cost of π_{sol} has then been added to $\mathbf{C}_{\text{sol}}^{\text{tr}}$ on Line 21.
2. Solution π_{sol} is found by the current run of `findApproxPF`. The truncated cost of π_{sol} has been added to $\mathbf{C}_{\text{sol}}^{\text{tr}}$ on Line 28.

In both cases, `findApproxPF` has added $Tr(\mathbf{c}(\pi_{\text{sol}}))$ to $\mathbf{C}_{\text{sol}}^{\text{tr}}$. There must exist some vector $\mathbf{x}$ in $\mathbf{C}_{\text{sol}}^{\text{tr}}$ that weakly dominates $Tr(\mathbf{c}(\pi_{\text{sol}}))$, which in turn ϵ_{curr}-dominates the truncated $\mathbf{f}$-value of apex-path pair $\mathcal{AP}$. Therefore, $\mathbf{x}$ ϵ_{curr}-dominates the truncated $\mathbf{f}$-value of apex-path pair $\mathcal{AP}$. □

Lemma 18. *`findApproxPF` does not expand an apex-path pair $\mathcal{AP}$ if there exists an solution π_{sol} with $\mathbf{c}(\pi_{sol}) \prec \mathbf{f}(\mathcal{AP})$.*

Proof. Consider the case that `findApproxPF` extracts an apex-path pair $\mathcal{AP}$ from OPEN on Line 19 and there exists an solution π_{sol} with $\mathbf{c}(\pi_{\text{sol}}) \prec \mathbf{f}(\mathcal{AP})$. We prove this lemma by showing that `findApproxPF` will not expand $\mathcal{AP}$. From Lemma 16, we can distinguish two cases:

1. There exists an apex-path pair $\mathcal{AP}'$ in OPEN that satisfies $\mathbf{g}(\mathcal{AP}') \preceq \mathbf{c}(\pi_{\text{sol}}^{(j)})$ and $s(\mathcal{AP}') = s_j$ for some j. $\mathbf{f}(\mathcal{AP}') = \mathbf{g}(\mathcal{AP}') + \mathbf{h}(s(\mathcal{AP}')) \preceq \mathbf{c}(\pi_{\text{sol}})$ because $\mathbf{g}(\mathcal{AP}') \preceq \mathbf{c}(\pi_{\text{sol}}^{(j)})$ and $\mathbf{h}$ is consistent. We have $\mathbf{f}(\mathcal{AP}') \preceq \mathbf{c}(\pi_{\text{sol}}) \prec \mathbf{f}(\mathcal{AP})$, which contradicts that `findApproxPF` extracts the apex-path pair with the lexicographically smallest $\mathbf{f}$-value. Therefore, this case cannot happen.
2. There exists a solution π'_{sol} in SOLS that ϵ_{curr}-dominates π_{sol}. $\mathbf{c}(\pi'_{\text{sol}})$ ϵ_{curr}-dominates $\mathbf{c}(\pi_{\text{sol}})$ and hence weakly dominates $\mathbf{f}(\mathcal{AP})$. Therefore, `findApproxPF` will prune $\mathcal{AP}$ on Line 24 because the condition on Line 2 of `isDominated` (Algorithm 5) holds, according to Lemma 17

□

Theorem 2. *`findApproxPF` returns in finite time. For any solution π_{sol}, there exists, when `findApproxPF` returns, a solution in SOLS that ϵ_{curr}-dominates solution π_{sol}.*

Proof. Consider any solution π_{sol} and any expanded apex-path pair $\mathcal{AP} = \langle \boldsymbol{A}, \pi \rangle$. Because `findApproxPF` generates only $\boldsymbol{\epsilon}_{\text{curr}}$-bounded apex-path pairs (Lemma 11), $\mathbf{f}(\pi)$ must $\boldsymbol{\epsilon}_{\text{curr}}$-dominate $\mathbf{f}(\mathcal{AP})$ (that is, $c_i(\pi) + h_i(s(\mathcal{AP})) \leq (1+\boldsymbol{\epsilon}_{\text{curr}}) f_i(\mathcal{AP})$ for all i). $c_i(\pi) + h_i(s(\mathcal{AP})) \leq (1+\boldsymbol{\epsilon}_{\text{curr}}) c_i(\pi_{\text{sol}})$ must hold for some i, because, otherwise, $(1+\boldsymbol{\epsilon}_{\text{curr}}) c_i(\pi_{\text{sol}}) < c_i(\pi) + h_i(s(\mathcal{AP})) \leq (1+\boldsymbol{\epsilon}_{\text{curr}}) f_i(\mathcal{AP})$ holds for all i, which contradicts that $\mathbf{f}(\mathcal{AP})$ is not dominated by $\mathbf{c}(\pi_{\text{sol}})$ from Lemma 18. Because heuristic $\mathbf{h}$ is non-negative, $c_i(\pi) \leq (1+\boldsymbol{\epsilon}_{\text{curr}}) c_i(\pi_{\text{sol}})$ must hold for some i for the representative path π of any expanded apex-path pair. Because the graph is finite and has positive edge costs, one can extend a path for finite times before the resulting path π does not satisfy $c_i(\pi) < (1+\boldsymbol{\epsilon}_{\text{curr}}) c_i(\pi_{\text{sol}})$ for all i. Thus, there are only a finite number of representative paths that `findApproxPF` can expand (and generate). `findApproxPF` returns in finite time.

Consider the beginning of the last iteration of `findApproxPF` before it returns, where OPEN becomes empty. From Lemma 16, for any solution π_{sol}, there exists a solution in SOLS that $\boldsymbol{\epsilon}_{\text{curr}}$-dominates π_{sol}. Thus, the theorem holds. □

Theorem 3. *Assuming that the sequence of ϵ's used by A-A*pex tends in the limit to zero, A-A*pex terminates, and SOLS is then a cost-unique Pareto frontier.*

Proof. Recall that the number of solution in any Pareto frontier $\mathcal{PF}$ is finite, there exists an approximation factor $\epsilon_{\mathcal{PF}} > 0$ such that the only $\epsilon_{\mathcal{PF}}$-approximation to $\mathcal{PF}$ is exactly $\mathcal{PF}$. As the sequence of ϵ's used by A-A*pex tends in the limit to zero, it will reach some approximation factor $\epsilon_{\text{final}} \leq \epsilon_{\mathcal{PF}}$. In that iteration A-A*pex will terminate and SOLS will contain the Pareto frontier $\mathcal{PF}$. □

References

[1] O. Salzman, C. Hernández, A. Felner, S. Koenig, Multi-objective search: algorithms, applications, and emerging directions, in: Proceedings of the AAAI Conference on Artificial Intelligence, 40 (2026), 40990–40999.

[2] D. Bachmann, F. Bökler, J. Kopec, K. Popp, B. Schwarze, F. Weichert, Multi-objective optimisation based planning of power-line grid expansions, ISPRS Int. J. Geo-Information 7 (7) (2018) 258.

[3] M. Fu, A. Kuntz, O. Salzman, R. Alterovitz, Toward asymptotically-optimal inspection planning via efficient near-optimal graph search, in: Robotics: Science and Systems (RSS), 2019.

[4] M. Fu, O. Salzman, R. Alterovitz, Computationally-efficient roadmap-based inspection planning via incremental lazy search, in: IEEE International Conference on Robotics and Automation (ICRA), 2021, pp. 7449–7456.
[5] A. Bronfman, V. Marianov, G. Paredes-Belmar, A. Lüer-Villagra, The Maximin HAZMAT routing problem, Eur. J. Oper. Res. 241 (1) (2015) 15–27.
[6] M. Ehrgott, Multicriteria Optimization (2nd Ed.), Springer, 2005.
[7] T. Breugem, T. Dollevoet, W. van den Heuvel, Analysis of FPTASes for the multi-objective shortest path problem, Comput. Oper. Res. 78 (2017) 44–58.
[8] G. Tsaggouris, C.D. Zaroliagis, Multiobjective optimization: improved FPTAS for shortest paths and non-linear objectives with applications, Theory Comput. Syst. 45 (1) (2009) 162–186.
[9] A. Warburton, Approximation of pareto optima in multiple-objective, shortest-path problems, Oper. Res. 35 (1) (1987) 70–79.
[10] B. Goldin, O. Salzman, Approximate bi-criteria search by efficient representation of subsets of the pareto-optimal frontier, in: International Conference on Automated Planning and Scheduling (ICAPS), 2021, pp. 149–158.
[11] P. Perny, O. Spanjaard, Near admissible algorithms for multiobjective search, in: European Conference on Artificial Intelligence (ECAI), 2008, pp. 490–494.
[12] H. Zhang, O. Salzman, T.K.S. Kumar, A. Felner, C.H. Ulloa, S. Koenig, A* pex: efficient approximate multi-objective search on graphs, in: International Conference on Automated Planning and Scheduling (ICAPS), 2022, pp. 394–403.
[13] M. Likhachev, G.J. Gordon, S. Thrun, ARA*: Anytime A* with provable bounds on sub-optimality, in: Advances in Neural Information Processing Systems, 2003.
[14] S. Richter, J. Thayer, W. Ruml, The joy of forgetting: faster anytime search via restarting, in: Proceedings of the International Conference on Automated Planning and Scheduling, 20 (2010), 137–144.
[15] S. Ahmadi, G. Tack, D. Harabor, P. Kilby, Enhanced Methods for the Weight Constrained Shortest Path Problem: Constrained Path Finding Meets Bi-objective Search, 2022. in: arXiv: 2207.14744
[16] S. Ahmadi, G. Tack, D. Harabor, P. Kilby, Weight constrained path finding with bidirectional A*, in: Symposium on Combinatorial Search (SoCS), 2022, pp. 2–10.
[17] N. Cabrera, A.L. Medaglia, L. Lozano, D. Duque, An exact bidirectional pulse algorithm for the constrained shortest path, Networks 76 (2) (2020) 128–146.
[18] B.W. Thomas, T. Calogiuri, M. Hewitt, An exact bidirectional A* approach for solving resource-constrained shortest path problems, Networks 73 (2) (2019) 187–205.
[19] G.Y. Handler, I. Zang, A dual algorithm for the constrained shortest path problem, Networks 10 (4) (1980) 293–309.
[20] D.H. Lorenz, D. Raz, A simple efficient approximation scheme for the restricted shortest path problem, Oper. Res. Lett. 28 (5) (2001) 213–219.
[21] F. Ergun, R. Sinha, L. Zhang, An improved FPTAS for restricted shortest path, Inf. Process. Lett. 83 (5) (2002) 287–291.
[22] C. Hernández, W. Yeoh, J.A. Baier, A. Felner, O. Salzman, H. Zhang, S.-H. Chan, S. Koenig, Multi-objective search via lazy and efficient dominance checks, in: IJCAI, 2023, pp. 7223–7230.
[23] J. Branke, K. Deb, K. Miettinen, R. Slowiński, Multiobjective Optimization: Interactive and Evolutionary Approaches, 5252, Springer Science & Business Media, 2008.
[24] K. Miettinen, Nonlinear Multiobjective Optimization, 12, Springer Science & Business Media, 2012.
[25] D.M. Roijers, S. Whiteson, Multi-Objective Decision Making, Synthesis Lectures on Artificial Intelligence and Machine Learning, Morgan & Claypool Publishers, 2017.
[26] R.T. Marler, J.S. Arora, Survey of multi-objective optimization methods for engineering, Struct. Multidiscip. Optim. 26 (6) (2004) 369–395.
[27] C.-L. Hwang, A.S. Masud, Multiple Objective Decision Making-Methods and Applications: A State-of-the-Art Survey, 164, Springer Science & Business Media, 2012.
[28] M. Emmerich, A. Deutz, A tutorial on multiobjective optimization: fundamentals and evolutionary methods, Nat. Comput. 17 (3) (2018) 585–609.
[29] K. Deb, A. Pratap, S. Agarwal, T. Meyarivan, A fast and elitist multiobjective genetic algorithm: NSGA-II, IEEE Trans. Evol. Comput. 6 (2) (2002) 182–197.
[30] K. Deb, H. Jain, An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, Part I: solving problems with box constraints, IEEE Trans. Evol. Comput. 18 (4) (2013) 577–601.
[31] Q. Zhang, H. Li, MOEA/D: a multiobjective evolutionary algorithm based on decomposition, IEEE Trans. Evol. Comput. 11 (6) (2007) 712–731.
[32] C.A.C. Coello, M.S. Lechuga, MOPSO: a proposal for multiple objective particle swarm optimization, in: Proceedings of the 2002 Congress on Evolutionary Computation. CEC'02 (Cat. No. 02TH8600), 2, IEEE, 2002, pp. 1051–1056.
[33] H. Li, D. Landa-Silva, An adaptive evolutionary multi-objective approach based on simulated annealing, Evol. Comput. 19 (4) (2011) 561–595.
[34] V.-I. Lupoaie, I.-A. Chili, M. Raschip, M.E. Breaban, An evolutionary approach for solving multi-objective WCSPs using mini-bucket elimination heuristics, in: 2021 IEEE Congress on Evolutionary Computation (CEC), IEEE, 2021, pp. 408–415.
[35] S. Ahmadi, G. Tack, D. Harabor, P. Kilby, Bi-objective search with bi-directional a*, in: Symposium on Combinatorial Search (SoCS), 2021, pp. 142–144.
[36] C. Hernández, W. Yeoh, J.A. Baier, H. Zhang, L. Suazo, S. Koenig, O. Salzman, Simple and efficient bi-objective search algorithms via fast dominance checks, Artif. Intell. 314 (2023) 103807.
[37] O. Salzman, A. Felner, C. Hernández, H. Zhang, S. Chan, S. Koenig, Heuristic-search approaches for the multi-objective shortest-path problem: progress and research opportunities, in: IJCAI, 2023, pp. 6759–6768.
[38] F.-J. Pulido, L. Mandow, J.-L. Pérez-de-la Cruz, Dimensionality reduction in multiobjective shortest path search, Comput. Oper. Res. 64 (2015) 60–70.
[39] C.U. Hernandez, W. Yeohz, J.A. Baier, H. Zhang, L. Suazoy, S. Koenig, A simple and fast bi-objective search algorithm, in: International Conference on Automated Planning and Scheduling (ICAPS), 2020, pp. 143–151.
[40] Z. Ren, R. Zhan, S. Rathinam, M. Likhachev, H. Choset, Enhanced multi-objective A* using balanced binary search trees, in: Symposium on Combinatorial Search (SoCS), 15, 2022, pp. 162–170.
[41] P. Maristany de las Casas, L. Kraus, A. Sedeño-Noda, R. Borndörfer, Targeted multiobjective Dijkstra algorithm, Networks 82 (3) (2023) 277–298.
[42] A. Sedeño-Noda, M. Colebrook, A biobjective Dijkstra algorithm, Eur. J. Oper. Res. 276 (1) (2019) 106–118.
[43] Y. Halle, A. Felner, S. Koenig, O. Salzman, A preprocessing framework for efficient approximate bi-objective shortest-path computation in the presence of correlated objectives, in: Symposium on Combinatorial Search (SoCS), 2025, pp. 65–73.
[44] Z. Ren, C. Hernández, M. Likhachev, A. Felner, S. Koenig, O. Salzman, S. Rathinam, H. Choset, EMOA*: a framework for search-based multi-objective path planning, Artif. Intell. 339 (2025) 104260.
[45] H. Zhang, O. Salzman, T.K.S. Kumar, A. Felner, C. Hernández, S. Koenig, Anytime approximate bi-objective search, in: Symposium on Combinatorial Search (SoCS), 2022, pp. 199–207.
[46] W. Li, Traveling salesman problem, in: The Traveling Salesman Problem: Optimization with the Attractor-Based Search System, Springer, 2023, pp. 9–25.
[47] K. Slutsky, D.S. Yershov, T. Wongpiromsarn, E. Frazzoli, Hierarchical multiobjective shortest path problems, in: Algorithmic Foundations of Robotics, 17 (2021), 261–276.
[48] O. Muhammetkulyyev, O. Salzman, T. Wongpiromsarn, Approximate multi-objective search under rulebooks, in: Workshop on the Algorithmic Foundations of Robotics (WAFR), 2026.