Speeding Up Dominance Checks in Multi-Objective Search: New Techniques and Data Structures

Han Zhang[1], **Oren Salzman**[2], **Ariel Felner**[3], **T. K. Satish Kumar**[1], **Carlos Hernández Ulloa**[4,5], **Sven Koenig**[1]

[1] University of Southern California
[2] Technion—Israel Institute of Technology
[3] Ben-Gurion University
[4] Universidad San Sebastián
[5] Centro Ciencia & Vida

zhan645@usc.edu, osalzman@cs.technion.ac.il, felner@bgu.ac.il, tkskwork@gmail.com, carlos.hernandez@uss.cl, skoenig@usc.edu

Abstract

In multi-objective search, given a directed graph where each edge is annotated with multiple cost metrics, a start state, and a goal state. We are interested in computing the Pareto frontier, i.e., the set of all undominated paths from the start state to the goal state. Almost all multi-objective search algorithms use dominance checks to determine if a search node can be pruned. Since dominance checks are performed in the inner loop of the multi-objective search, they are the most time-consuming part of it. In this paper, we propose (1) two novel techniques to reduce duplicate dominance checks and (2) a simple data structure that enables more efficient dominance checks. Our experimental results show that combining our proposed techniques and data structure speeds up LTMOA*, a state-of-the-art multi-objective search algorithm, by up to an order of magnitude on road network instances.

Introduction and Related Work

In multi-objective search (Ulungu and Teghem 1991; Salzman et al. 2023), we are given a directed graph, a start state, and a goal state. Each edge in the graph is annotated with a cost vector, where each component corresponds to a cost metric to minimize, such as travel time, travel distance, or economic cost. A *solution* is a path from the start state to the goal state. A solution π *dominates* another solution π' iff π is not worse than π' on any cost metric and is better than π' on at least one cost metric. A typical task of multi-objective search is to find the *Pareto frontier*, that is, all undominated solutions. Multi-objective search and the task of finding Pareto frontiers are important in many real-world application domains, including route planning for trucks, robots, and power lines (Bachmann et al. 2018) as well as inspecting regions of interest with robots (Fu et al. 2019; Fu, Salzman, and Alterovitz 2021). For example, transporting hazardous material requires one to consider trade-offs between the path length and the number of residents exposed to the hazardous material in case of a traffic accident (Bronfman et al. 2015).

Existing multi-objective search algorithms include NAMOA*dr (Pulido, Mandow, and Pérez-de-la Cruz 2015), EMOA* (Ren et al. 2022), and LTMOA* (Hernández et al. 2023). All these algorithms represent paths emanating from the start state as search nodes, or simply called *nodes*, and use *dominance checks* to determine if a path has the potential to be extended to an undominated solution. Since dominance checks are performed frequently, i.e., in the inner loop of the search, they are the most time-consuming part of it and need to be performed efficiently. However, dominance checks intrinsically require iterating over sets of vectors. For example, when generating a new path, NAMOA*dr checks if (the cost vector of) this path is dominated by (the cost vector of) any previously generated path with the same last state. The difference in the runtime performance of these algorithms stems from their difference in how they implement dominance checks and interleave dominance checks with the search. Among them, the most recent LTMOA* has been shown to outperform EMOA* and NAMOA*dr by up to an order of magnitude in terms of runtime (Hernández et al. 2023). Hence, in this paper, we focus on adding our proposed enhancements to LTMOA*, although they can be added to any of these algorithms.

Our first contribution consists of two novel techniques that speed up LTMOA* by eliminating duplicate dominance checks between a node and its parent node. Our second contribution is to adapt well-known bucket-based data structures to our specific context of storing undominated vectors and performing dominance checks. We call these *bucket arrays*: vectors are slotted into different predefined buckets based on their values. The search algorithm can often determine if a bucket contains a vector that dominates a given vector without iterating over all vectors in this bucket.

Although LTMOA* propounds the use of arrays in practice, it can conceivably be used with other data structures that store undominated vectors for performing dominance checks. Not all of these data structures have been investigated so far. In this paper, we evaluated LTMOA* not only with arrays but also with ND-trees (Jaszkiewicz and Lust 2018), a data structure that has been shown to generally outperform other data structures for maintaining sets of undominated vectors with respect to runtime.

In our experimental study, we evaluated LTMOA* with

and without our proposed enhancements on road network instances with three to five objectives. The results show that our proposed enhancements are beneficial for most problem instances. On problem instances with five objectives, they yield up to an order of magnitude speed-up.

Terminology and Problem Definition

We use **boldface** font to denote vectors and v_i to denote the i-th component of a vector $\mathbf{v}$. The addition of two vectors $\mathbf{v}$ and $\mathbf{v}'$ of the same length N is defined as $\mathbf{v} + \mathbf{v}' = [v_1 + v'_1, v_2 + v'_2 \dots v_N + v'_N]$. We say that $\mathbf{v}$ *weakly dominates* $\mathbf{v}'$, denoted as $\mathbf{v} \preceq \mathbf{v}'$, iff $v_i \leq v'_i$ for all $i = 1, 2 \dots N$. We say that $\mathbf{v}$ *dominates* $\mathbf{v}'$, denoted as $\mathbf{v} \prec \mathbf{v}'$, iff $\mathbf{v} \preceq \mathbf{v}'$ and $\mathbf{v} \neq \mathbf{v}'$. The *truncated vector* of a vector $\mathbf{v}$, denoted as $\mathrm{Tr}(\mathbf{v})$, is $\mathbf{v}$ with its first component deleted, i.e., $[v_2, v_3 \dots v_N]$.

A *(multi-objective search) graph* is a tuple $\langle S, E, \mathbf{c} \rangle$, where S is a finite set of *states* and $E \subseteq S \times S$ is a finite set of directed *edges*. $succ(s) = \{s' \in S : \langle s, s' \rangle \in E\}$ denotes the successors of state s. Cost function $\mathbf{c} : E \to \mathbb{R}^N_{\geq 0}$ maps an edge to its cost, which is a vector with N non-negative components. A *(multi-objective search) problem instance* is a tuple $P = \langle S, E, \mathbf{c}, s_{\text{start}}, s_{\text{goal}} \rangle$, where $\langle S, E, \mathbf{c} \rangle$ is a graph, $s_{\text{start}} \in S$ is the start state, and $s_{\text{goal}} \in S$ is the goal state.

A *path* from state s_1 to state s_l is a sequence of states $\pi = [s_1, s_2 \dots s_l]$ with $\langle s_i, s_{i+1} \rangle \in E$ for all $i = 1, 2 \dots l-1$. We assume $s_1 = s_{\text{start}}$ unless mentioned otherwise. $\mathbf{c}(\pi) = \sum_{i=1}^{l-1} \mathbf{c}(\langle s_i, s_{i+1} \rangle)$ denotes the *cost* of path π. Path π can be *extended* with an edge $\langle s_l, s_{l+1} \rangle$ to obtain a new path $[s_1, s_2 \dots s_l, s_{l+1}]$. Path π *dominates* (resp. *weakly dominates*) another path π' iff $\mathbf{c}(\pi) \prec \mathbf{c}(\pi')$ (resp. $\mathbf{c}(\pi) \preceq \mathbf{c}(\pi')$).

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. A *(cost-unique) Pareto frontier* is a maximal subset of all Pareto-optimal solutions such that any two solutions in the subset do not have the same cost.

A *heuristic (function)* $\mathbf{h} : S \to \mathbb{R}^N_{\geq 0}$ provides a lower bound on the cost of any path from any given state s to the goal state. We assume that the provided heuristic function $\mathbf{h}$ is consistent, that is, $\mathbf{h}(s_{\text{goal}}) = \mathbf{0}$ and $\mathbf{h}(s) \preceq \mathbf{c}(\langle s, s' \rangle) + \mathbf{h}(s')$ for all $\langle s, s' \rangle \in E$.

LTMOA*

Algorithm 1 shows the pseudo-code of LTMOA*. In LTMOA*, a node n is associated with a state $s(n)$, a $\mathbf{g}$-value $\mathbf{g}(n)$, and a parent node $p(n)$. We say that n is a node on state $s(n)$. The $\mathbf{f}$-value of n is defined as $\mathbf{f}(n) = \mathbf{g}(n) + \mathbf{h}(s(n))$. Conceptually, a node n corresponds to a path from s_{start} to $s(n)$ with cost $\mathbf{g}(n)$, and this path can be constructed in reverse by following the parent nodes from $s(n)$ to s_{start}. LTMOA* maintains a priority queue $Open$ for the generated but not expanded nodes and a set of solutions $sols$. It initializes $Open$ with a node on state s_{start} which corresponds to path $[s_{\text{start}}]$ and whose $\mathbf{g}$-value is $\mathbf{0}$.

In each iteration, LTMOA* *extracts* a node n from $Open$ with the *lexicographically smallest* $\mathbf{f}$-value. It then performs dominance checks (Lines 10-11) and prunes n if there exists

- (Condition 1) an expanded node on state $s(n)$ whose $\mathbf{g}$-value weakly dominates $\mathbf{g}(n)$ or
- (Condition 2) an expanded node on state s_{goal} whose $\mathbf{f}$-value weakly dominates $\mathbf{f}(n)$.

If node n is not pruned on Line 10, LTMOA* reaches Line 12. In this case, we say that LTMOA* *expands* n. If $s(n) = s_{\text{goal}}$, LTMOA* adds the corresponding solution of n to $sols$ (Line 14) or, if $s(n) \neq s_{\text{goal}}$, it *generates* a new child node n' for each successor of $s(n)$ (Line 17). LTMOA* also performs dominance checks for n' (Lines 20-21) before adding it to $Open$. When $Open$ becomes empty, LTMOA* terminates and returns $sols$ as a Pareto frontier (Line 23).

By exploiting the fact that the expanded nodes have lexicographically non-decreasing $\mathbf{f}$-values, LTMOA* does not need to check the g_1- and f_1-values for dominance checks. Instead of maintaining the set of $\mathbf{g}$-values of all expanded nodes for each state s, LTMOA* maintains only the often significantly smaller set $\mathbf{G}^{\text{Tr}}_{\text{cl}}(s)$ of undominated truncated $\mathbf{g}$-values. Checking Conditions 1 and 2 can be done by checking if there exists a vector in $\mathbf{G}^{\text{Tr}}_{\text{cl}}(s(n))$ that weakly dominates $\mathrm{Tr}(\mathbf{g}(n))$ and a vector in $\mathbf{G}^{\text{Tr}}_{\text{cl}}(s_{\text{goal}})$ that weakly dominates $\mathrm{Tr}(\mathbf{f}(n))$, respectively, using function `IsDominated` (Lines 24-25). For each expanded node n, LTMOA* calls function `Update` (Line 12) to remove

Algorithm 1: LTMOA*

Input : A search problem $(S, E, \mathbf{c}, s_{\text{start}}, s_{\text{goal}})$ and a consistent heuristic function $\mathbf{h}$
Output: A Pareto frontier

1 $sols \leftarrow \emptyset$
2 **for each** $s \in S$ **do**
3 | initialize $\mathbf{G}^{\text{Tr}}_{\text{cl}}(s)$
4 $n \leftarrow$ new node with $s(n) = s_{\text{start}}$
5 $\mathbf{g}(n) \leftarrow$ zero in all dimensions
6 $p(n) \leftarrow null$
7 initialize $Open$ and add n to it
8 **while** $Open \neq \emptyset$ **do**
9 | extract a node n from $Open$ with the lexicographically smallest f-value
10 | **if** `IsDominated`$(\mathrm{Tr}(\mathbf{g}(n)), \mathbf{G}^{\text{Tr}}_{\text{cl}}(s(n)))$ *or* `IsDominated`$(\mathrm{Tr}(\mathbf{f}(n)), \mathbf{G}^{\text{Tr}}_{\text{cl}}(s_{\text{goal}}))$ **then**
11 | | **continue**
12 | `Update`$(\mathbf{G}^{\text{Tr}}_{\text{cl}}(s(n)), \mathrm{Tr}(\mathbf{g}(n)))$
13 | **if** $s(n) = s_{goal}$ **then**
14 | | add n to $sols$
15 | | **continue**
16 | **for each** $s' \in succ(s(n))$ **do**
17 | | $n' \leftarrow$ new node with $s(n') = s'$
18 | | $\mathbf{g}(n') \leftarrow \mathbf{g}(n) + \mathbf{c}(\langle s(n), s' \rangle)$
19 | | $p(n') \leftarrow n$
20 | | **if** `IsDominated`$(\mathrm{Tr}(\mathbf{g}(n')), \mathbf{G}^{\text{Tr}}_{\text{cl}}(s'))$ *or* `IsDominated`$(\mathrm{Tr}(\mathbf{f}(n')), \mathbf{G}^{\text{Tr}}_{\text{cl}}(s_{\text{goal}}))$ **then**
21 | | | **continue**
22 | | add n' to $Open$
23 **return** $sols$
24 **Function** `IsDominated`$(V, \mathbf{v})$**:**
25 | **return** $\exists \mathbf{v}' \in V \; \mathbf{v}' \preceq \mathbf{v}$
26 **Function** `Update`$((V, \mathbf{v}))$**:**
27 | remove all vectors weakly dominated by $\mathbf{v}$ from V
28 | add $\mathbf{v}$ to V

all vectors that are weakly dominated by $Tr(\mathbf{g}(n))$ from $\mathbf{G}_{\text{cl}}^{\text{Tr}}(s(n))$ and then add $Tr(\mathbf{g}(n))$ to $\mathbf{G}_{\text{cl}}^{\text{Tr}}(s(n))$ (Lines 26-28). Hernández et al. (2023) propose to use arrays to implement $\mathbf{G}_{\text{cl}}^{\text{Tr}}(s)$ for each state s, and both the `IsDominated` and `Update` functions need to iterate over the vectors in $\mathbf{G}_{\text{cl}}^{\text{Tr}}(s)$, which takes time linear in the size of $\mathbf{G}_{\text{cl}}^{\text{Tr}}(s)$.

Reducing Duplicate Dominance Checks

Consider the case when LTMOA* reaches Line 10 with a node n whose parent node $p(n)$ is not null. LTMOA* needs to iterate over $\mathbf{G}_{\text{cl}}^{\text{Tr}}(s(n))$ to check if a vector in $\mathbf{G}_{\text{cl}}^{\text{Tr}}(s(n))$ weakly dominates $\text{Tr}(\mathbf{g}(n)$. However, some vectors in $\mathbf{G}_{\text{cl}}^{\text{Tr}}(s(n))$ are truncated $\mathbf{g}$-values for nodes whose parent nodes are also on state $s(p(n))$. Intuitively, LTMOA* does not need to check these vectors because it has already checked the truncated $\mathbf{g}$-values of the parent nodes of the nodes corresponding to these vectors before expanding $p(n)$. A similar observation holds for Line 20 as well. For each truncated $\mathbf{g}$-value $\mathbf{v}$ in $\mathbf{G}_{\text{cl}}^{\text{Tr}}(s(n))$, let the *parent state* of $\mathbf{v}$ refer to the state of the parent node of the node corresponding to $\mathbf{v}$. We show the following property of LTMOA*.

Property 1. *When LTMOA* checks Condition 1 for a node n whose parent node $p(n)$ is not null, a vector $\mathbf{v}$ in $\mathbf{G}_{cl}^{Tr}(s(n))$ does not weakly dominate $\text{Tr}(\mathbf{g}(n))$ if the parent state of $\mathbf{v}$ is $s(p(n))$.*

Proof. We prove this property by contradiction. Assume that such a vector $\mathbf{v}$ weakly dominates $\text{Tr}(\mathbf{g}(n))$. Let n' denote the expanded node corresponding to $\mathbf{v}$ whose parent $p(n')$ is also on $s(p(n))$. We have $\text{Tr}(\mathbf{g}(n')) \preceq \text{Tr}(\mathbf{g}(n))$. Because n' is expanded prior to n, the $\mathbf{f}$-value of n' is lexicographically no larger than the $\mathbf{f}$-value of n, and hence we have $g_1(n') \leq g_1(n)$. Put together, we have $\mathbf{g}(n') \preceq \mathbf{g}(n)$. Because $s(p(n)) = s(p(n'))$, the difference between $\mathbf{g}(p(n))$ and $\mathbf{g}(n)$ is equal to the difference between $\mathbf{g}(p(n'))$ and $\mathbf{g}(n')$, that is, $\mathbf{c}(\langle s(p(n)), s(n)\rangle)$. Therefore, we have $\mathbf{g}(p(n')) \preceq \mathbf{g}(p(n))$. If $\mathbf{g}(p(n')) \prec \mathbf{g}(p(n)), p(n')$ is expanded before $p(n)$, and $p(n)$ would have been pruned in dominance checks. Otherwise, we have $\mathbf{g}(p(n')) = \mathbf{g}(p(n))$, the later extracted one between $p(n)$ and $p(n')$ would have been pruned. In both cases, we have a contradiction because both $p(n)$ and $p(n')$ are expanded nodes. □

Based on Property 1, we propose the following technique to reduce the number of dominance checks:

Dominance-check reduction technique 1 (R1). *We partition the vectors in $\mathbf{G}_{cl}^{Tr}(s)$ into different subsets according to their parent states. When checking Condition 1, we only check those subsets of $\mathbf{G}_{cl}^{Tr}(s(n))$ whose parent states are not $s(p(n))$. When updating $\mathbf{G}_{cl}^{Tr}(s(n))$ with vector $Tr(\mathbf{g}(n))$, we add $Tr(\mathbf{g}(n))$ only to the subset corresponding to $s(p(n))$ but still remove vectors weakly dominated by $Tr(\mathbf{g}(n))$ from all subsets.*

While the R1 technique reduces the number of dominance checks for Condition 1, the following technique reduces the number of dominance checks for Condition 2:

Dominance-check reduction technique 2 (R2). *When performing dominance checks for a node n, we do not check Condition 2 if $\mathbf{f}(n) = \mathbf{f}(p(n))$.*

Intuitively, if $\mathbf{f}(n) = \mathbf{f}(p(n))$, $Tr(\mathbf{f}(n))$ is not dominated by any vector in $\mathbf{G}_{\text{cl}}^{\text{Tr}}(s_{\text{goal}})$ because, otherwise, $p(n)$ would have been pruned. However, it is possible for LTMOA* to find a solution whose cost is equal to $\mathbf{f}(n)$ between the expansion of $p(n)$ and n. Such cases can be detected by checking the last solution found by LTMOA*.

Bucket Arrays

In this section, we describe the bucket array, a data structure for storing $\mathbf{G}_{\text{cl}}^{\text{Tr}}(s)$, and how the `IsDominated` and `Update` functions work with bucket arrays. Let N denote the number of objectives, and hence the length of each vector in $\mathbf{G}_{\text{cl}}^{\text{Tr}}(s)$ is $N-1$. A bucket array is an array of buckets, where each *bucket* in turn contains an array of vectors (of length $N-1$) and each component of a vector in this bucket is within a predefined range of value. More specifically, let δ denote the *step* of value, which is a parameter for the bucket array. The *index* of a vector $\mathbf{v}$ is defined as $\mathbf{I}(\mathbf{v}) = [\lfloor v_1/\delta \rfloor, \lfloor v_2/\delta \rfloor \dots \lfloor v_{N-1}/\delta \rfloor]$. A bucket B contains vectors with the same index, denoted as $\mathbf{I}(B)$. Thus, every vector $\mathbf{v}$ in bucket B satisfies that $I_i(B) \cdot \delta \leq v_i < (I_i(B)+1) \cdot \delta$ for $i = 1, 2 \dots N-1$. All buckets in a bucket array have different indices and are not empty.

When calling `IsDominated` to determine if there is a vector in $\mathbf{G}_{\text{cl}}^{\text{Tr}}(s)$ that weakly dominates an input vector $\mathbf{v}$, the algorithm needs to iterate over each bucket B in the bucket array (that stores $\mathbf{G}_{\text{cl}}^{\text{Tr}}(s)$) with the following cases:

1. If $\mathbf{I}(B)$ does not weakly dominate $\mathbf{I}(\mathbf{v})$, then `IsDominated` does not check any vector in B because none of these vectors weakly dominates $\mathbf{v}$.
2. If $\mathbf{I}(B)$ satisfies $I_i(B) < I_i(\mathbf{v})$ for all $i = 1, 2 \dots N-1$, then `IsDominated` returns $true$ immediately because all vectors in B weakly dominate $\mathbf{v}$.
3. Otherwise, `IsDominated` checks the vectors in B and returns $true$ if it finds a vector that weakly dominates $\mathbf{v}$.

The algorithm needs to check the vectors inside a bucket only in the last case. Therefore, using bucket arrays can reduce the number of vectors that need to be checked.

When LTMOA* calls the `Update` function, it adds an input vector $\mathbf{v}$ to $\mathbf{G}_{\text{cl}}^{\text{Tr}}(s)$ and removes all vectors that are weakly dominated by $\mathbf{v}$ from $\mathbf{G}_{\text{cl}}^{\text{Tr}}(s)$. Note that $\mathbf{v}$ is not weakly dominated by any vector in $\mathbf{G}_{\text{cl}}^{\text{Tr}}(s)$ because, otherwise, LTMOA* would have reached Line 11 and continued to the next iteration. The `Update` function first adds $\mathbf{v}$ to the bucket whose index is $\mathbf{I}(\mathbf{v})$ (before that, `Update` might need to create such a bucket if it does not exist yet). It then iterates over the buckets in $\mathbf{G}_{\text{cl}}^{\text{Tr}}(s)$ with the following cases:

1. If $\mathbf{I}(B)$ is not weakly dominated by $\mathbf{I}(\mathbf{v})$, then `Update` does not update B because none of these vectors is weakly dominated by $\mathbf{v}$.
2. If $\mathbf{I}(B)$ satisfies that $I_i(v) < I_i(\mathbf{B})$ for all $i = 1, 2 \dots N-1$, `Update` removes B from the bucket array because all of its vectors are weakly dominated by $\mathbf{v}$.
3. Otherwise, `Update` iterates over the vectors in B and remove the ones that are weakly dominated by $\mathbf{v}$.

	3 objectives (d-t-m)			4 objectives (l-d-t-m)			5 objectives (l-d-t-m-r)		
	#solved	t_{avg}	#ops	#solved	t_{avg}	#ops	#solved	t_{avg}	#ops
LTMOA*	70	22.10	19727K	38	37.66	24444K	23	23.93	19078K
LTMOA*+R1	71	17.56	16267K	39	30.46	20379K	23	18.21	15858K
LTMOA*+R2	71	20.07	17041K	38	35.30	20613K	23	21.12	16120K
LTMOA*+R	71	17.45	13581K	39	29.69	16548K	23	15.94	12899K
LTMOA*+ND-Tree	59	67.35	6585K	37	56.02	7020K	22	21.93	12201K
LTMOA*+Bucket	76	11.01	1744K	40	13.54	2332K	27	4.08	1506K
LTMOA*+R+Bucket	**77**	**10.08**	**958K**	**41**	**9.51**	**1293K**	**28**	**3.31**	**886K**

Table 1: Numbers of solved problem instances (#solved), average runtimes (t_{avg}, in seconds) and average numbers of vector comparisons (#ops) for different variants of LTMOA* on problem instances with different numbers of objectives.

Similar to the three cases for the `IsDominated` function, the `Update` function needs to check the vectors inside a bucket only in the last case.

Bucket arrays (and other data structures) can be combined with the R1 and R2 techniques with minor modifications. For example, the R1 technique partitions $\mathbf{G}_{\text{cl}}^{\text{Tr}}(s)$ into different subsets. In this case, we simply store each subset in a separate bucket array.

Experimental Results

In this section, we evaluate our proposed enhancements with LTMOA* on problem instances with three to five objectives. We use the NY road network (264,346 states and 733,846 edges) from the 9th DIMACS Implementation Challenge: Shortest Path.[1] The NY road network has two objectives, namely travel distance (d) and travel time (t), available in the benchmark. Additionally, we use the economic cost (m) (Pulido, Mandow, and Pérez-de-la Cruz 2015), the number of edges (l) (Maristany de las Casas et al. 2023), and a random integer between 1 and 100 (r) (Hernández et al. 2023) as the third, fourth, and fifth objective, respectively. We use the same 100 pairs of start and goal states used by Sedeño-Noda and Colebrook (2019) and Ahmadi et al. (2021) for our problem instances. Following Hernández et al. (2023), we use the heuristic that corresponds to the exact minimum cost for each objective to reach the goal state.

We first evaluate LTMOA* with only the R1 technique (+R1), only the R2 technique (+R2), and both techniques (+R), where arrays are used to store sets of cost vectors. Then, we evaluate LTMOA* with ND-trees (+NDTree), LTMOA* with bucket arrays (+Bucket), and, finally, LTMOA* that combines R1, R2, and bucket arrays (+R+Bucket). We implemented all variants of LTMOA* in C++.[2] We obtained the original ND-Tree implementation from Jaszkiewicz and Lust (2018) and integrated it into our code base. The runtime limit for solving each problem instance was ten minutes.

An ND-Tree is parameterized by a branching factor b and a maximum number of vectors L in each leaf node. In our preliminary study, we evaluated LTMOA*+NDTree with all combinations of $b \in \{5, 10, 20\}$ and $L \in \{20, 40, 80\}$ on 30 random problem instances on NY with four objectives. We then chose the parameter combination with the smallest runtime, namely $b = 5$ and $L = 20$. Similarly, we evaluated LTMOA*+Bucket with $\delta \in \{1000, 10000, 20000, 50000, 100000\}$ on the same 30 problem instances and chose $\delta = 20000$.

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

[2] https://github.com/HanZhang39/MultiObjectiveSearch

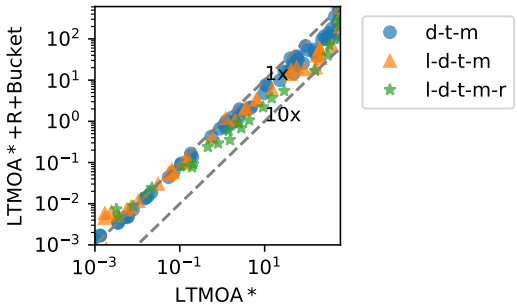

Figure 1: Runtimes (in seconds) on individual problem instances for LTMOA* versus LTMOA*+R+Bucket. The dashed diagonal lines correspond to different speed-ups.

Table 1 shows the results for the different variants of LTMOA*. All averages are taken over instances that are solved by all algorithms. For all objectives, the addition of R1 or R2 improves the average runtime of LTMOA*. For five objectives, the addition of R1 and R2 speeds up the average runtime of LTMOA* about 1.5 times. Although ND-trees reduce the numbers of vector comparisons in all cases, they improve the average runtime only for problem instances with five objectives due to their runtime overhead. However, bucket arrays improve the average runtime of LTMOA* in all cases, and LTMOA*+R+Bucket has the best runtime overall.

Figure 1 shows the runtime of LTMOA* and LTMOA*+R+Bucket on each problem instance. We use different markers for different objectives. LTMOA*+R+Bucket runs faster than LTMOA* on most problem instances except some easy ones that take less than 0.01 seconds to solve. The speed-up over LTMOA* is up to an order of magnitude.

Conclusions

Multi-objective search requires frequent dominance checks. In this paper, we facilitated efficient dominance checks via two techniques that reduce duplicate dominance checks and an effective data structure, called bucket arrays. Our experimental results showed that our proposed enhancements speed up LTMOA* by up to an order of magnitude. One direction of future work is to investigate the effect of different δ-values for the different dimensions of bucket arrays.

Acknowledgements

The research at the University of Southern California was supported by the National Science Foundation (NSF) under grant numbers 1409987, 1724392, 1817189, 1837779, 1935712, and 2112533. It was also supported by the United States-Israel Binational Science Foundation (BSF) under grant number 2021643, the Ministry of Science & Technology, Israel, under grant number 3-17385, the National Center for Artificial Intelligence CENIA FB210017, Basal ANID, and the Centro Ciencia & Vida FB210008, Financiamiento Basal ANID. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the sponsoring organizations, agencies, or governments.

References

Ahmadi, S.; Tack, G.; Harabor, D.; and Kilby, P. 2021. Bi-Objective Search with Bi-Directional A*. In *Symposium on Combinatorial Search (SoCS)*, 142–144.

Bachmann, D.; Bökler, F.; Kopec, J.; Popp, K.; Schwarze, B.; and Weichert, F. 2018. Multi-Objective Optimisation Based Planning of Power-Line Grid Expansions. *ISPRS International Journal of Geo-Information*, 7(7): 258.

Bronfman, A.; Marianov, V.; Paredes-Belmar, G.; and Lüer-Villagra, A. 2015. The Maximin HAZMAT Routing Problem. *European Journal of Operational Research*, 241(1): 15–27.

Fu, M.; Kuntz, A.; Salzman, O.; and Alterovitz, R. 2019. Toward Asymptotically-Optimal Inspection Planning via Efficient Near-Optimal Graph Search. In *Robotics: Science and Systems (RSS)*.

Fu, M.; Salzman, O.; and Alterovitz, R. 2021. Computationally-Efficient Roadmap-Based Inspection Planning via Incremental Lazy Search. In *IEEE International Conference on Robotics and Automation (ICRA)*, 7449–7456.

Hernández, C.; Yeoh, W.; Baier, J. A.; Felner, A.; Salzman, O.; Zhang, H.; Chan, S.-H.; and Koenig, S. 2023. Multi-Objective Search via Lazy and Efficient Dominance Checks. In *International Joint Conference on Artificial Intelligence (IJCAI)*, 7223–7230.

Jaszkiewicz, A.; and Lust, T. 2018. ND-Tree-Based Update: a Fast Algorithm for the Dynamic Nondominance Problem. *IEEE Transactions on Evolutionary Computation*, 22(5): 778–791.

Maristany de las Casas, P.; Kraus, L.; Sedeño-Noda, A.; and Borndörfer, R. 2023. Targeted Multiobjective Dijkstra Algorithm. *Networks*, 82(3): 277–298.

Pulido, F.-J.; Mandow, L.; and Pérez-de-la Cruz, J.-L. 2015. Dimensionality Reduction in Multiobjective Shortest Path Search. *Computers & Operations Research*, 64: 60–70.

Ren, Z.; Zhan, R.; Rathinam, S.; Likhachev, M.; and Choset, H. 2022. Enhanced Multi-Objective A* Using Balanced Binary Search Trees. In *Symposium on Combinatorial Search (SoCS)*, 162–170.

Salzman, O.; Felner, A.; Hernández, C.; Zhang, H.; Chan, S.; and Koenig, S. 2023. Heuristic-Search Approaches for the Multi-Objective Shortest-Path Problem: Progress and Research Opportunities. In *International Joint Conference on Artificial Intelligence (IJCAI)*, 6759–6768.

Sedeño-Noda, A.; and Colebrook, M. 2019. A Biobjective Dijkstra Algorithm. *European Journal of Operational Research*, 276(1): 106–118.

Ulungu, E.; and Teghem, J. 1991. Multi-Objective Shortest Path Problem: A Survey. In *Workshop on Multicriteria Decision Making: Methods–Algorithms–Applications*, 176–188.