Efficient Multi-Query Bi-Objective Search via Contraction Hierarchies

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

[1] University of Southern California
[2] Technion - Israel Institute of Technology
[3] Ben-Gurion University
[4] Universidad San Sebastian
zhan645@usc.edu, osalzman@cs.technion.ac.il, felner@bgu.ac.il, tkskwork@gmail.com, carlos.hernandez@uss.cl, skoenig@usc.edu

Abstract

Contraction Hierarchies (CHs) have been successfully used as a preprocessing technique in single-objective graph search for finding shortest paths. However, only a few existing works on utilizing CHs for bi-objective search exist, and none of them uses CHs to compute Pareto frontiers. This paper proposes an CH-based approach capable of efficiently computing Pareto frontiers for bi-objective search along with several speedup techniques. Specifically, we propose a new preprocessing approach that computes CHs with fewer edges than the existing preprocessing approach, which reduces both the preprocessing times (up to $3\times$ in our experiments) and the query times. Furthermore, we propose a partial-expansion technique, which dramatically speeds up the query times. We demonstrate the advantages of our approach on road networks with 1 to 14 million states. The longest preprocessing time is less than 6 hours, and the average speedup in query times is roughly two orders of magnitude compared to BOA*, a state-of-the-art single-query bi-objective search algorithm.

Introduction

The task of bi-objective search is to find paths from a given start vertex to a given goal vertex in a graph whose edges are annotated with two costs. Each cost corresponds to a different cost metric, such as travel time, travel distance, risk, etc., and the different objectives in the search problem correspond to the minimization of different cost metrics. Bi-objective search is important for many real-world applications, including route planning for power lines considering economic and ecological impacts (Bachmann et al. 2018), inspecting regions of interest with robots considering motion cost and coverage (Fu et al. 2019; Fu, Salzman, and Alterovitz 2021), and transporting hazardous materials considering travel distance and risk (Bronfman et al. 2015).

There does not necessarily exist a single path from the start vertex to the goal vertex that simultaneously optimizes both objectives. Therefore, we are typically interested in finding a set of *undominated* paths. A path π dominates a path π' iff π is not worse than π' on any cost metric and is better than π' on at least one cost metric. In this paper, we are interested in finding all undominated paths, referred to as the *Pareto frontier*.

Copyright © 2023, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

There are many bi-objective search algorithms for computing the Pareto frontier, with recent examples like BOA* (Ulloa et al. 2023), A-BOA* (Zhang et al. 2022b), T-MDA (Maristany de las Casas et al. 2021), and BOBA* (Ahmadi et al. 2021). These algorithms are single-query and only consider solving a single problem instance on a given graph. However, in many real-world applications, the search algorithm needs to solve multiple problem instances on the same graph. It is common practice for such a multi-query setting to speed up query times via preprocessing techniques.

A well-studied preprocessing technique for single-objective search is Contraction Hierarchies (CHs) (Geisberger et al. 2008). In single-objective search, a CH is a hierarchical graph that assigns a level number to each vertex in the input graph and adds additional edges (known as shortcuts) to the input graph so that the shortest path from a given start vertex to a given goal vertex can be found by searching through the space of only up-down paths (paths with first increasing and then decreasing level numbers). Similarly, in bi-objective search, a CH needs to retain the property that the Pareto frontier can be computed by considering only up-down paths.

To our best knowledge, CHs have been used in graphs with two costs but never to compute the entire Pareto frontier. Specifically, Storandt (2012) proposed a CH-based approach to solving the constrained shortest-path problem. Its preprocessing algorithm computes shortcuts heuristically, which avoids the computational cost of computing the exact shortcuts but can add unnecessary shortcuts.

This paper proposes a CH-based approach to computing Pareto frontiers for bi-objective search and several speedup techniques that leverage recent algorithmic advances. Specifically, we show how to use BOA* (Ulloa et al. 2023) to compute shortcuts. This alternative approach allows us to compute only the necessary shortcuts, reducing both the preprocessing and query times. Furthermore, we observe that CHs in bi-objective search often contain a large number of edges, which slow down a search algorithm in the query phase. Consequently, we propose a (general) partial-expansion technique, which dramatically reduces the query time by reducing the number of unnecessary generated search nodes.

Our experimental results show that our preprocessing ap-

proach can reduce the number of shortcuts by up to 15% with less runtime than Storandt (2012). In one scenario, the speedup over Storandt (2012) is more than $3\times$. We demonstrate the advantages of our approach on road networks with 1 to 14 million states. The longest preprocessing time is less than 6 hours, and the average speedup in query times is roughly two orders of magnitude compared to BOA*, a state-of-the-art single-query bi-objective search algorithm.

Related Work

Preprocessing techniques have been used extensively in multi-query single-objective search. Examples other than CHs include true distance heuristics (Sturtevant et al. 2009), embedding in Euclidean spaces (Cohen et al. 2018), and sub-goal graphs (Uras, Koenig, and Ulloa 2013). None of them has been generalized to bi-objective search. One of the few existing works on computing Pareto frontiers with preprocessing techniques is multi-criteria SHARC (Delling and Wagner 2009). However, it has been demonstrated only on small road networks with less than 80,000 vertices and is not immediately scalable to larger graphs. This is partly because its preprocessing algorithm needs to compute single-vertex-to-all-vertices Pareto frontiers, which require a large amount of memory for large road networks.

A few existing works other than Storandt (2012) apply CHs to problems that consider graphs with two costs but differ from our approach (Geisberger, Kobitzsch, and Sanders 2010; Funke and Storandt 2013; Baum et al. 2015). Both Geisberger, Kobitzsch, and Sanders (2010) and Funke and Storandt (2013) use different weighted combinations of the costs to map a bi-objective search problem to several single-objective search problems. The resulting CHs cannot be used to find paths on the Pareto frontier that do not minimize any weighted combination of the costs. Baum et al. (2015) apply CHs to a constrained shortest-path problem that considers charging, recuperation, and battery capacity of electric vehicles. The agent (vehicle) has a fixed battery capacity and can charge at stations. The problem objective is to minimize the total travel time (including the time for charging) while satisfying that the battery never gets empty.

Terminology

We use boldface font to denote pairs. For a pair $\mathbf{p}$, we use p_i, $i \in \{1, 2\}$, to denote its i-th component. The addition of two pairs $\mathbf{p}$ and $\mathbf{p}'$ is defined as $\mathbf{p} + \mathbf{p}' = (p_1 + p'_1, p_2 + p'_2)$. $\mathbf{p} \preceq \mathbf{p}'$ denotes that $p_1 \leq p'_1$ and $p_2 \leq p'_2$. In this case, we say that $\mathbf{p}$ *weakly dominates* $\mathbf{p}'$. $\mathbf{p} \prec \mathbf{p}'$ denotes that $\mathbf{p} \preceq \mathbf{p}'$ and there exists an $i \in \{1, 2\}$ for which $p_i < p'_i$. In this case, we say that $\mathbf{p}$ *(strictly) dominates* $\mathbf{p}'$.

A *(bi-objective) graph* is a tuple $G = \langle S, E \rangle$, where S is a finite set of *states*, and E is a finite set of *directed edges*. Each *edge* $e = \langle u, v, \mathbf{c} \rangle$ is a tuple consisting of a *source state* $u \in S$, a *target state* $v \in S$, and a *cost* $\mathbf{c} \in \mathbb{R}^2_{>0}$. We use $src(e)$, $tar(e)$, and $\mathbf{c}(e)$ to denote the source state, the target state, and the cost of e, respectively. We use $in(s) = \{e \mid e \in E, tar(e) = s\}$ and $out(s) = \{e \mid e \in E, src(e) = s\}$ to denote the in- and out-edges of state s, respectively. We allow G to contain *parallel edges*, that is, edges from the same source state to the same target state. State s' is an *in-neighbor* (resp. *out-neighbor*) of state s iff there exists an edge from s' to s (resp. from s to s'). We use $in_nbr(s)$ and $out_nbr(s)$ to denote the sets of all in- and out-neighbors of state s, respectively.

A *path* from state s to state s' is a sequence of edges $\pi = [e_1, e_2 \dots e_\ell]$ with $src(e_1) = s$, $tar(e_\ell) = s'$, and $tar(e_j) = src(e_{j+1})$ for all $j \in \{1, 2 \dots \ell - 1\}$. $\mathbf{c}(\pi) = \sum_{j=1}^{\ell} \mathbf{c}(e_j)$ denotes the cost of path π. 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 *query* $q = \langle s_{\text{start}}, s_{\text{goal}} \rangle$ consists of a start state s_{start} and a goal state s_{goal}. A path is a *solution* to q iff it is from s_{start} to s_{goal}. A solution is Pareto-optimal iff it is not dominated by any other solution. In this paper, we are interested in finding a *(cost-unique) Pareto frontier*, that 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* $\mathbf{h} : S \rightarrow \mathbb{R}^2_{\geq 0}$ provides a lower bound on the cost from any given state s to the goal state. We assume that the heuristic function $\mathbf{h}$ is *consistent*, that is, $\mathbf{h}(s_{\text{goal}}) = \mathbf{0}$ and $\mathbf{h}(s) \preceq \mathbf{c}(e) + \mathbf{h}(s')$ for all $e = \langle s, s', \mathbf{c} \rangle \in E$.

Algorithmic Background

In this section, we review CHs for single-objective search (where there is only one cost to minimize) and BOA*.

CHs for Single-Objective Search

Since we consider only single-objective search in this section, we use a scalar $c(e)$ to denote the cost of edge e.

Given a single-objective graph $G = \langle S, E \rangle$, a CH is computed by performing *contractions* on the states in S one by one according to a given state ordering. Contracting a state s removes it and its incident edges (i.e., both in- and our-edges) from the graph while preserving the minimum-path cost between any pair of states in the remaining graph. To do so, before removing s and its incident edges, the preprocessing algorithm iterates through every pair of in-edge e and out-edge e' of s. It runs a so-called *witness search* to determine if there is a path (witness) from $src(e)$ to $tar(e')$ in the current graph that does not traverse state s and whose cost is smaller than or equal to $c(e) + c(e')$. The witness search can be implemented with any shortest-path algorithm, such as Dijkstra's algorithm. If the algorithm does not find a witness, a new edge $\langle src(e), tar(e'), c(e) + c(e') \rangle$ that *bridges* edges e and e', called a *shortcut*, is added to the graph to preserve the minimum path-cost from $src(e)$ to $tar(e')$. Generating a CH does not require contracting all states. Let L denote the number of states to contract, determined by a user. After the first L states are contracted, a CH $G_{\text{ch}} = \langle S, E_{\text{ch}} \rangle$ is created. The state set S is the one of the input graph G, and the edge set E_{ch} consists of the edges in E (including the ones that were removed during contraction) and all shortcuts. In case there are parallel edges, only the minimum-cost one is kept. The i-th contracted state s is assigned a *level number* of $lvl(s) = i$, and all uncontracted states (also called the *core* of the CH) are assigned level numbers of $L + 1$.

Algorithm 1: BOA*

```
   Input  : G = ⟨S, E⟩, s_start, s_goal, h
1  for s ∈ S do
2  |  g_2^min(s) ← ∞
3  n_root ← new node at s_start with g(n_root) = (0, 0) and
     parent(n_root) = None
4  initialize Open and add n_root to it
5  Sols ← ∅
6  while Open ≠ ∅ do
7  |  n ← Open.pop()
8  |  if g_2^min(s(n)) ≤ g_2(n) ∨ g_2^min(s_goal) ≤ f_2(n) then
9  |  |  continue
10 |  g_2^min(s(n)) ← g_2(n)
11 |  if s(n) = s_goal then
12 |  |  add n to Sols
13 |  |  continue
14 |  for e ∈ out(s(n)) do
15 |  |  n' ← new node at tar(e) with g(n') = g(n) + c(e)
   |  |    and parent(n') = n
16 |  |  if g_2^min(s(n')) ≤ g_2(n') ∨ g_2^min(s_goal) ≤ f_2(n') then
17 |  |  |  continue
18 |  |  add n' to Open
19 return Sols
```

An edge from state u to state v is an *upward edge* (resp. a *downward edge*) iff $lvl(u) \leq lvl(v)$ (resp. $lvl(u) > lvl(v)$). A path is an *upward path* (resp. a *downward path*) iff it consists of only upward edges (resp. downward edges). A path $\pi = [e_1, e_2 \ldots e_\ell]$ is an *up-down path* iff there exists a j such that edges $e_i, i \leq j$, are all upward edges and edges $e_i, i > j$, are all downward edges. The following theoretical result is rephrased from Lemma 1 in Geisberger et al. (2008).

Theorem 1. *For any pair of states u and v, there exists an up-down path from u to v in G_{ch} with the minimum-path cost from u to v in the input graph G.*

Given a query, a minimum-cost up-down solution in G_{ch} can be computed efficiently using a modified bidirectional Dijkstra's algorithm (Geisberger et al. 2008) or a modified A* algorithm (Harabor and Stuckey 2018). Then, the up-down solution can be unpacked into a minimum-cost solution in G by recursively replacing each shortcut with the two edges that it bridges.

Different CHs can be obtained from the same input graph using different state orderings for contraction. The ordering plays an important role in both the preprocessing and query times of the resulting CH and is usually determined with heuristics that take into account the number of shortcuts to add if a state is contracted and the number of incident edges of that state (Geisberger et al. 2008).

BOA*

BOA* (Ulloa et al. 2023) is a best-first bi-objective search algorithm. The inputs to BOA* are a graph G, a start state s_{start}, a goal state s_{goal}, and a (consistent) heuristic function $\mathbf{h}$. The output of BOA* is a Pareto frontier of solutions. In BOA*, a node n represents a path from s_{start} to some end state $s(n)$. We say that node n is a node *at* state $s(n)$. The $\mathbf{g}$-value of n, denoted as $\mathbf{g}(n)$, is the cost of this path, and the $\mathbf{f}$-value of n is $\mathbf{f}(n) = \mathbf{g}(n) + \mathbf{h}(s(n))$. We use $parent(n)$ to denote the *parent* of n, which is either a node or $None$. BOA* generalizes A* to bi-objective search but, unlike A*, needs to maintain several nodes, each with its own $\mathbf{g}$-value, at the same state.

BOA*, outlined in Alg. 1, maintains a priority queue $Open$, which contains the frontier of the search tree (the generated but not yet expanded nodes), and a set of nodes $Sols$, which contains nodes at s_{goal} and represents the Pareto-optimal solutions that it has found so far. For each state s, BOA* uses $g_2^{\min}(s)$ to store the minimum g_2-value of all expanded nodes at state s. In the beginning, BOA* initializes $Open$ with a *root node* n_{root} at state s_{start} with $\mathbf{g}(n_{\text{root}}) = (0, 0)$ and $parent(n_{\text{root}}) = None$ (Lines 3-4). At each iteration, BOA* pops a node n with the lexicographically smallest $\mathbf{f}$-value from $Open$. BOA* then performs the following *dominance checks* for n:

1. It checks if an expanded node n' at $s(n)$ with $\mathbf{g}(n') \preceq \mathbf{g}(n)$ exists. If so, any solution found via n is weakly dominated by some solution found via n'.
2. It checks if a node n' in $Sols$ with $\mathbf{g}(n') \preceq \mathbf{f}(n)$ exists. If so, any solution found via n is weakly dominated by the solution that n' represents.

In either case, BOA* prunes n (Lines 8-9). Since BOA* uses a consistent heuristic function and always pops nodes from $Open$ with lexicographically increasing $\mathbf{f}$-values, it does not need to check g_1- or f_1-values during the dominance checks. Therefore, these two checks can be done efficiently by checking if $g_2^{min}(s(n)) \leq g_2(n)$ and $g_2^{min}(s_{\text{goal}}) \leq f_2(n)$, respectively. If n is not pruned and $s(n) = s_{\text{goal}}$, BOA* adds n to $Sols$ (Lines 11-13). Otherwise, for each out-edge e of $s(n)$, BOA* expands n and generates a child node n' with $s(n') = tar(e)$, $\mathbf{g}(n') = \mathbf{g}(n) + \mathbf{c}(e)$, and $parent(n') = n$ (Lines 14-18). BOA* also performs dominance checks for n' (Lines 16-17) and adds n' to $Open$ if it is not pruned (Line 18). BOA* returns $Sols$ when $Open$ becomes empty.

CHs for Bi-Objective Search: The Preprocessing Phase

Like a CH in single-objective search, a CH in bi-objective search is built by contracting one state at a time in the input graph G until contracting L states. Contracting a state s removes it and its incident edges from G while preserving at least one Pareto frontier between any pair of states in the remaining graph. Each combination of an input edge and an output edge of s is a shortcut candidate. The pre-processing algorithm needs to determine whether to add a shortcut for each candidate. To do so, we propose two approaches to building a CH: the basic approach and the batched approach. While the basic approach runs a witness search for every shortcut candidate individually, the batched approach groups the candidates for parallel shortcuts (i.e. shortcuts from the same source state to the same target state) into a batch and uses a single witness search to test all of them at once, dramatically reducing the preprocessing times. Importantly,

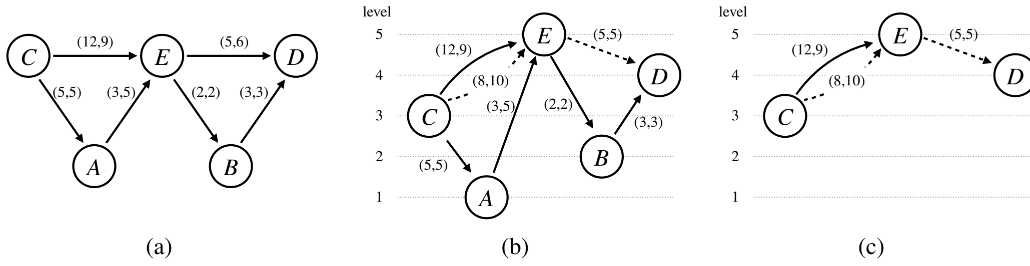

Figure 1: An example of CHs in bi-objective search. (a) The input graph, whose edges are labeled with their costs. (b) The CH created after contracting all states in alphabetic order of their names. Dashed edges depict shortcuts. (c) The search graph $\tilde{G}$ constructed for a query with start state C and goal state D.

Algorithm 2: The Preprocessing Algorithm – Basic

Input : $G = \langle S, E \rangle, L$
1 $S_{\text{ch}} \leftarrow S; E_{\text{ch}} \leftarrow \{\}$
2 **while** $|S_{ch}| - |S| < L$ **do**
3 $s \leftarrow$ choose the next state to contract
4 **for** $e \in in(s)$ **do**
5 **for** $e' \in out(s)$ **do**
6 $u \leftarrow src(e); v \leftarrow tar(e')$
7 **if** $witness_search(G, u, v, \mathbf{c}(e) + \mathbf{c}(e'))$ **then**
8 $add_shortcut(\langle u, v, \mathbf{c}(e) + \mathbf{c}(e') \rangle)$
9 add all edges incident on s to E_{ch}
10 remove s from S and all edges incident on s from E
11 add all remaining edges in E to E_{ch}
12 **return** $G_{\text{ch}} = \langle S_{\text{ch}}, E_{\text{ch}} \rangle$;
13 **Function** $witness_search(G, u, v, \mathbf{p})$**:**
14 $\pi \leftarrow$ a path from u to v whose cost dominates $\mathbf{p}$, or $none$ if no such path exists
15 **return** $true$ if $\pi = none$, otherwise $false$
16 **Function** $add_shortcut(e_{sc})$**:**
17 remove edges parallel to e_{sc} whose costs are weakly dominated by $\mathbf{c}(e_{\text{sc}})$ from E
18 add e_{sc} to E

in contrast to the witness search of Storandt (2012), these two approaches use exact witness search algorithms and add fewer shortcuts to the CH.

The Basic Approach

Our basic approach to building a CH is outlined in Alg. 2. Here, when contracting a state s, for every pair of in-edge e and out-edge e' of s, it uses $witness_search$ to determine if there exists a path (witness) from $src(e)$ to $tar(e')$ whose cost dominates $\mathbf{c}(e) + \mathbf{c}(e')$ (Lines 4-8). We omit the pseudocode of $witness_search$ since it is based on BOA* with the following modifications:

- **Termination**: Once a witness is found, $witness_search$ terminates and returns $true$.
- **Pruning**: $witness_search$ prunes any node n if (i) $f_1(n) > c_1(e) + c_1(e')$ or (ii) $f_2(n) > c_2(e) + c_2(e')$ as any solution found via n cannot be a witness.
- **Heuristic computation**: When planning a path to some goal state s_{goal}, Ulloa et al. (2023) propose to run a (single-objective) backward search with Dijkstra's algorithm on the reverse graph of G to compute the heuristic function to s_{goal} for each objective individually. This proves to be too time-consuming to do every time $witness_search$ is invoked. Thus, we terminate the backward search once $src(e)$ is expanded. Subsequently, the heuristic value for any state s is set to the minimum path cost from $tar(e')$ to s on the reverse graph if s has been expanded or the minimum path cost from $tar(e')$ to $src(e)$ on the reverse graph otherwise. The resulting heuristic function is consistent.

A shortcut $e_{\text{sc}} := \langle src(e), tar(e'), \mathbf{c}(e) + \mathbf{c}(e') \rangle$ is added to the graph if $witness_search$ does not find a witness (Line 8). Additionally, we remove all those edges parallel to e_{sc} whose costs are weakly dominated by $\mathbf{c}(e_{\text{sc}})$ (Line 17) as such edges are not needed to preserve any Pareto frontier.

After contracting L states, Alg. 2 returns a CH $G_{\text{ch}} = \langle S_{\text{ch}}, E_{\text{ch}} \rangle$ whose states S_{ch} consist of all states of the input graph and whose edges E_{ch} consist of all edges incident on the contracted states before they are removed from the input graph (Line 9) and all remaining edges (Line 11).

Example 1. *Figure 1a shows an example of a bi-objective input graph. States are contracted in alphabetic order of their names, and L is* 5*:*

- *State A is contracted, and a shortcut from C to E with cost $(8, 10)$ is added to the graph.*
- *State B is contracted, and a shortcut from E to D with cost $(5, 5)$ is added to the graph. The edge from E to D with cost $(5, 6)$ is removed since it is weakly dominated by the added shortcut.*
- *States C, D, and E are contracted in order. No shortcuts are created.*

Figure 1b shows the CH G_{ch} after the contractions. There are two parallel edges between states C and E, whose costs are not weakly dominated by each other.

The Batched Approach

As demonstrated in Example 1, a contraction can add parallel edges to the remaining graph. When contracting a state, for different combinations of its parallel in-edges and its parallel out-edges, the search effort in $witness_search$ can be

Algorithm 3: The Preprocessing Algorithm – Batched

```
    Input  : G = ⟨S, E⟩, L
1   S_ch ← S; E_ch ← {}
2   while |S_ch| − |S| < L do
3       s ← choose the next state to contract
4       for u ∈ in_nbr(s) do
5           for v ∈ out_nbr(s) do
6               Π ← all 2-hop paths from u to v that traverse s
7               Π_sc = witness_search_batch(G, u, v, s, Π)
8               for π ∈ Π_sc do
9                   add_shortcut(⟨u, v, c(π)⟩)
10      add all edges incident on s to E_ch
11      remove s from S and all edges incident on s from E
12  add all remaining edges in E to E_ch
13  return G_ch = ⟨S_ch, E_ch⟩;
14  Function witness_search_batch(G = ⟨S, E⟩, u, v, s, Π):
*15     q ← sort Π in a lexicographically increasing order of path
          costs after removing paths dominated by other paths in Π
          and, if several paths have the same cost, keeping only one
          of them
*16     Π_sc ← ∅
17      for s' ∈ S do
18          g_2^min(s') ← ∞
19      n_root ← new node at u with g(n_root) = (0, 0)
20      initialize Open and add n_root to it
21      while Open ≠ ∅ ∧ q ≠ ∅ do
22          n ← Open.pop()
*23         while q ≠ ∅ ∧ f_1(n) > c_1(q.top()) do
*24             pop q.top() from q and add it to Π_sc
*25         if q = ∅ then break
*26         if g_2^min(s(n)) ≤ g_2(n) ∨ c_2(q.top()) < f_2(n) then
*27             continue
28          g_2^min(s(n)) ← g_2(n)
29          if s(n) = v then
*30             while q ≠ ∅ ∧ f_2(n) ≤ c_2(q.top()) do
*31                 pop the top path from q
*32             if q = ∅ then break
33              continue
34          for e ∈ out(s(n)) do
35              if tar(e) = s then
36                  continue
37              n' ← new node at tar(e) with
                  g(n') = g(n) + c(e)
*38             if g_2^min(s(n')) ≤ g_2(n') ∨ c_2(q.top()) < f_2(n')
                  then
*39                 continue
40              add n' to Open
41      add all remaining paths in q to Π_sc
42      return Π_sc
```

duplicated. Our batched approach, outlined in Alg. 3, attempts to eliminate such duplicated search effort. Specifically, for every pair of in-neighbor u and out-neighbor v of s, the algorithm finds all 2-hop paths Π from u to v that traverse s, that is, all paths consisting of an in-edge e' of s with $src(e) = u$ and an out-edge e'' of s with $tar(e') = v$. It then uses a single run of $witness_search_batch$ to determine which paths in Π need to result in shortcuts (Line 7). Function $witness_search_batch$ returns a subset of Π, denoted as Π_{sc}, which consists of all paths in Π that are not weakly dominated by any path from u to v that does not traverse s.

Function $witness_search_batch$, like $witness_search$, is based on BOA* (Lines 14-40). However, here the changes are not straightforward, and we thus highlight the major changes by using "*" before line numbers in its pseudo-code. The changes include (i) initializing variables (Lines 15-16), (ii) deciding if a path in Π should result in a shortcut (Lines 23-24 and 30-31), and (iii) pruning nodes (Lines 26-27 and 38-39). We now elaborate on each change.

During the initialization, Function $witness_search_batch$ removes path dominated by other paths in Π from Π and, if several paths have the same cost, keeps only one of them. It sorts the remaining paths in lexicographically increasing order of their costs and inserts them into a priority queue q (Line 15). Since paths in q do not dominate each other and have unique costs, their c_1 costs monotonically increase, and their c_2 costs monotonically decrease. Intuitively, q contains the shortcut candidates that need to be checked. Function $witness_search_batch$ also uses variable Π_{sc} (Line 16), initialized to $\emptyset$, to store the paths that need to result in shortcuts.

Function $witness_search_batch$ then runs a BOA*-like search from the source state u. The next difference from BOA* occurs when a node n is popped from $Open$ with $f_1(n) > f_1(q.top())$, in which case the algorithm pops $q.top()$ and adds it to Π_{sc} (Lines 23-24). The algorithm does this because any solutions that can be found via n will only have c_1-values larger than or equal to the f_1-value of n and hence cannot weakly dominate any path in q. Note that $q.top()$ has the smallest c_1-value in q.

When the algorithm finds a solution node n (Line 29), it removes all paths in q whose costs are weakly dominated by $\mathbf{f}(n)$. We have $f_1(n) \leq c_1(q.top())$ (otherwise, the algorithm cannot get out of the while-loop on Lines 23-24) and thus $f_1(n) \leq c_1(\pi)$ for every $\pi \in q$. Therefore, to check if there is a path in q whose cost is weakly dominated by $\mathbf{f}(n)$, the algorithm only needs to check $q.top()$, which has the largest c_2-value. If $f_2(n) \leq c_2(q.top())$, the algorithm has found a path (represented by n) that weakly dominates $q.top()$ and hence pops $q.top()$ from q (without adding it to Π_{sc}). The algorithm repeats this process until q becomes empty or $f_2(n) \leq c_2(q.top())$ does not hold (Lines 30-31).

Function $witness_search_batch$ also has different dominance checks from BOA*. It prunes node n if $c_2(q.top()) < f_2(n)$ because, in this case, no path from u to v found via n weakly dominates any path in q. Note that, in the case of $c_2(q.top()) = f_2(n)$, it is still possible for a path from u to v found via n to weakly dominate $q.top()$. Finally, when q or $Open$ becomes empty, the algorithm adds all remaining paths in q to Π_{sc} and returns Π_{sc} (Line 42).

CHs for Bi-Objective Search: The Query Phase

In this section, we describe how we combine CHs with BOA* in the query phase. Additionally, we describe a simple-yet-effective partial-expansion technique that reduces the query time by reducing the number of nodes inserted into $Open$.

Constructing Search Graphs

The query phase relies on the up-down property of CHs. That is, for any path π from state u to state v in the input graph G, there exists an up-down path from state u to state v in the CH G_{ch} that weakly dominates π. Therefore, a Pareto frontier can be found by searching through the space of only up-down paths in G_{ch}.

While it is customary to use bi-directional Dijkstra's algorithm over CHs in the query phase of single-objective search with one direction considering only upward paths and the other direction considering only downward paths, the analogue for bi-objective search requires careful examination. One such algorithm is Bi-directional Bi-objective Dijkstra's algorithm (Sedeño-Noda and Colebrook 2019). However, Ulloa et al. (2023) show that it is less efficient than BOA*. BOBA* (Ahmadi et al. 2021) is another bi-objective search algorithm that utilizes two simultaneous bi-objective searches, one from the source and one from the target. However, the search in each direction is independent of the other one and hence cannot focus on only upward or downward paths, respectively.

Our approach is to first build a *search graph* $\tilde{G}$ for the input query $\langle s_{\text{start}}, s_{\text{goal}}\rangle$. $\tilde{G}$ is a subgraph of G_{ch} and consists of all up-down paths from s_{start} to s_{goal}. Then, we can run any bi-objective search algorithm (here, we use BOA*) on $\tilde{G}$ to find a Pareto frontier. We denote $\tilde{G}$ as $\langle \tilde{S} = S^{\uparrow} \cup S^{\downarrow}, \tilde{E}\rangle$, where $S^{\uparrow}$ consists of all states that can be reached from s_{start} via an upward path and $S^{\downarrow}$ consists of all states that can reach s_{goal} via a downward path. $S^{\uparrow}$ and $S^{\downarrow}$ are computed by running a depth-first search on G_{ch} and its inverse graph, respectively. $\tilde{E}$ consists of (i) all upward edges with source states in $S^{\uparrow}$ and (ii) all downward edges with target states in $S^{\downarrow}$.

Example 2. *Figure 1c shows the search graph $\tilde{G}$ constructed for query $\langle C, D\rangle$ and the CH in Figure 1b. State set $S^{\uparrow}$ consists of C and E, and state set $S^{\downarrow}$ consists of D and E. The edge set of the search graph only contains the upward edges from C to E and the downward edge from E to D.*

For query $\langle C, D\rangle$, there are only two Pareto-optimal paths $\pi_1 = [\langle C, A, (5,5)\rangle, \langle A, E, (3,5)\rangle, \langle E, B, (2,2)\rangle, \langle B, D, (3,3)\rangle]$ with a cost of $(13, 15)$ and $\pi_2 = [\langle C, E, (12,9)\rangle, \langle E, B, (2,2)\rangle, \langle B, D, (3,3)\rangle]$ with a cost of $(17, 14)$ in the graph in Figure 1a . These two paths have the same costs as paths $\pi'_1 = [\langle C, E, (8,10)\rangle, \langle E, D, (5,5)\rangle]$ and $\pi'_2 = [\langle C, E, (12,9)\rangle, \langle E, D, (5,5)\rangle]$ in the constructed search graph $\tilde{G}$, respectively. Paths π'_1 and π'_2 can be found with BOA, and paths π_1 and π_2 can then be obtained by unpacking paths π'_1 and π'_2, respectively.*

Partial Expansions

In a CH for bi-objective search, there can be many (up to several hundred in our experiments) parallel edges from a state s to another state s' due to contractions. When expanding a node at state s, BOA* generates child nodes for all edges from s to s', which may be unnecessary if some of these child nodes are pruned later. Therefore, we propose a "lazy" variant of BOA* that utilizes partial expansions to avoid generating all child nodes in many cases by generating them one by one, as needed. The idea of partial expansions comes from single-objective search (Felner et al. 2012), where it keeps track of the child node to generate next for each expanded node. We adapt this idea to keep track of the child node to generate next for each pair of an expanded node n and one of the out-neighbors of $s(n)$. This enables the algorithm to identify quickly whether all child nodes at the out-neighbor can be pruned without checking all corresponding out-edges.

BOA* with partial expansions, outlined in Alg. 4, requires that, for any two states s and s', all edges from s to s' that are dominated by other edges from s to s' are removed, and, if several edges have the same cost, only one of them is kept. The remaining edges are sorted in order of lexicographically increasing costs. These changes (removing and sorting edges) are done in the preprocessing phase. We use $m_{s,s'}$ to denote the number of edges from s to s' and $e^1_{s,s'}, e^2_{s,s'} \ldots e^{m_{s,s'}}_{s,s'}$ to denote the sequence of these edges when sorted in the lexicographical order. We say that i is the *index* of edge $e^i_{s,s'}$. Additionally, we use $c_2^{\min}(s,s')$ to denote the minimum c_2-value of all edges from s to s', that is, $c_2(e^{m_{s,s'}}_{s,s'})$. We highlight the major changes of Alg. 4 over

Algorithm 4: BOA* with Partial Expansion

```
    Input : G = ⟨S, E⟩, s_start, s_goal, h
1   for s ∈ S do
2   |  g2^min(s) ← ∞
3   n_root ← new node at s_start with g(n_root) = (0, 0) and
      parent(n_root) = None
4   Initialize Open and add n_root to it
5   Sols ← ∅
6   while Open ≠ ∅ do
7   |  n ← Open.pop()
*8  |  if parent(n) ≠ None then
*9  |  |  generate_next(parent(n), s(n), idx(n) + 1)
10  |  if g2(n) ≥ g2^min(s(n)) ∨ f2(n) ≥ g2^min(s_goal) then
11  |  |  continue
12  |  g2^min(s(n)) ← g2(n)
13  |  if s(n) = s_goal then
14  |  |  add n to Sols
15  |  |  continue
16  |  for s' ∈ out_neighbor(s(n)) do
*17 |  |  generate_next(n, s', 1)
18  return Sols
*19 Function generate_next(n, s, i):
*20 |  if g2(n) + c2^min(s(n), s) ≥ g2^min(s)∨
        g2(n) + c2^min(s(n), s) + h2(s) ≥ g2^min(s_goal) then
*21 |  |  return
*22 |  for j = i, i + 1 ... m_{s(n),s} do
*23 |  |  n' ← new node at s with g(n') = g(n) + c(e^j_{s(n),s}),
          idx(n') = j, and parent(n') = n
*24 |  |  if g2(n') ≥ g2^min(s)∨ f2(n') ≥ g2^min(s_goal) then
*25 |  |  |  continue
*26 |  |  add n' to Open
*27 |  |  return
```

BOA* by using "*" before line numbers. When expanding a node n, the algorithm uses $generate_next$ for each out-neighbor s of $s(n)$ (Line 17). Function $generate_next$ first checks if there is any undominated child node at s can be generated (Lines 20-21). This is done by checking if the minimum g_2-value of these child node, that is, $g_2(n) + c_2^{\min}(s(n), s)$, is larger than or equal to $g_2^{min}(s)$ and if the minimum f_2-value of these child node, that is, $g_2(n) + c_2^{\min}(s(n), s) + h_2(s)$, is larger than or equal to $f_2^{min}(s_{\text{goal}})$. If such a node exists, $generate_next$ iterates over the edges $e^1_{s,s'}, e^2_{s,s'} \dots e^{m_{s,s'}}_{s,s'}$ until it finds the first edge that results in an undominated child node n'. Function $generate_next$ then adds n' to $Open$ and returns (Lines 22-27). For each node n, the algorithm uses $idx(n)$ to record the index of the edge that was used to generate it. When n is popped from $Open$ and is not the root node, the algorithm calls $generate_next$ to generate the next undominated child node of $parent(n)$ at state $s(n)$, if one exists (Line 9). When iterating over the edges from $s(parent(n))$ to $s(n)$, $generate_next$ starts with the edge with index $idx(n) + 1$ because all edges with smaller indices have already been iterated over in the previous calls of $generate_next$ for $parent(n)$. The rest of Alg. 4 is the same as BOA*.

Experimental Results

In this section, we evaluate our CH-based approach on road networks from the 9th DIMACS Implementation Challenge: Shortest Path.[1] The two cost metrics are travel distance and time from the DIMACS data set. For each road network, we use the same 100 queries used by Ahmadi et al. (2021). The runtime limit for each query is 30 minutes. We implemented all algorithms in C++ on a common code base[2] and ran all experiments on a MacBook Pro with an M1 Pro CPU and 32GB of memory.

To order states for contraction (Line 3 in Alg. 2 and 3), we assign a priority $\psi(s)$ to each state s and contract the lowest-priority state at each iteration. To define ψ, we use $\kappa(s)$ to denote the ratio of the number of shortcuts to add when contracting s and the number of edges incident on s. Furthermore, we use $\eta(s)$ to denote the *height* of a state s to be one plus the height of the highest state with an upward edge to s or a value of one if no such state exists. Intuitively, contracting states with small heights leads to a more even contraction across the graph. In our implementation, we set $\psi(s) := 10 \cdot \kappa(s) + \eta(s)$. We also implemented the lazy-update scheme (Geisberger et al. 2008), which recalculates the priority of a state when it is popped from the priority queue and reinserts it into the priority queue if its priority becomes higher than the second-lowest priority.

Comparing different contraction approaches and contraction ratios: We start by evaluating the impact of different contraction ratios (i.e., percentages of states to contract, which is captured by L in Alg. 2 and 3) and different preprocessing approaches. Here, we use the NE road network (1.5M states and 3.9M edges) and the LKS road network (2.8M states and 6.9M edges), two medium-sized maps from the DIMACS dataset, and set a time limit of 24 hours for the preprocessing phase.

We evaluate the contraction ratios 99%, 99.5%, 99.95%, and 100% and three preprocessing approaches: The basic and batched approaches are referred to as basic and batched, respectively. We also implement the approach proposed by Storandt (2012), referred to as support-point. For each in-neighbor s' and out-neighbor s'' of state s, the witness search of support-point runs a series of single-

Algorithm	Preprocessing t_{prep}	Preprocessing $\lvert E_{\text{ch}}\rvert$	Query #exp	Query $t^{\text{+CH}}_{\text{BOA*}}$	Query $t^{\text{+CH+p}}_{\text{BOA*}}$
NE					
contracting 99% of states					
support-point	6min	8.0M	265K	3.93(100)	2.56
basic	8min	8.0M	262K	3.92(100)	2.48
batched	7min	8.0M	262K	3.82(100)	2.41
contracting 99.5% of states					
support-point	8min	8.1M	145K	3.78(100)	1.81
basic	13min	8.1M	142K	3.20(100)	1.62
batched	10min	8.1M	141K	**3.14**(100)	1.52
contracting 99.95% of states					
support-point	37min	9.2M	40K	6.10(100)	0.71
basic	3hr53min	8.8M	35K	3.62(100)	**0.51**
batched	21min	8.8M	36K	3.80(100)	**0.51**
contracting 100% of states					
support-point	2hr53min	11.8M	38K	19.92(100)	0.89
basic	timeout				
batched	1hr46min	10.3M	**32K**	11.67(100)	0.64
LKS					
contracting 99% of states					
support-point	14min	14.2M	1,155K	25.96(100)	14.87
basic	19min	14.2M	1,158K	24.41(100)	13.90
batched	16min	14.2M	1,156K	25.32(100)	15.27
contracting 99.5% of states					
support-point	23min	14.6M	604K	19.35(100)	8.80
basic	41min	14.6M	607K	20.32(100)	9.54
batched	24min	14.6M	613K	**17.86**(100)	8.42
contracting 99.95% of states					
support-point	3hr34min	18.7M	167K	78.11 (87)	5.56
basic	timeout				
batched	1hr08min	16.5M	163K	39.27 (96)	**4.35**
contracting 100% of states					
support-point	timeout				
basic	timeout				
batched	10hr08min	21.1M	**161K**	137.86 (84)	5.41

Table 1: Experimental results for different contraction approaches and contraction ratios on the NE and LKS maps. We report the preprocessing times t_{prep}, the numbers of edges $|E_{\text{ch}}|$ in each CH, the average numbers of expanded nodes #exp, the average query times (in seconds) $t^{\text{+CH}}_{\text{BOA*}}$ for BOA* with CH (but without partial expansions), with the number of solved instances in parenthesis, and the average query time (in seconds) $t^{\text{+CH+p}}_{\text{BOA*}}$ for BOA* with CH and partial expansions (here, all algorithms solved all instances). For each road network, the average runtimes are taken over instances that are solved by both BOA* and BOA* with partial expansion for all CHs.

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

[2] https://github.com/HanZhang39/Bi-Objective-Contraction-Hierarchy.

objective searches from s' to s''. Each single-objective search is parameterized by a λ-value and finds a path π' that minimizes $\lambda c_1(\pi') + (1-\lambda)c_2(\pi')$. For every 2-hop path π from s' via s to s'', if the witness search finds a path whose cost dominates $\mathbf{c}(\pi)$, then π does not result in a shortcut. Otherwise, a shortcut is added. Adding the shortcut may be unnecessary but does not affect the correctness of the query phase. We use a sequence of three λ-values $[\lambda_1, \lambda_2, \lambda_3]$ as described in Storandt (2012), with $\lambda_1 = 0$, $\lambda_2 = 1$, and $\lambda_3 = (c_2 - c'_2)/(c_2 - c'_2 + c_1 - c'_1)$, where $\mathbf{c}$ and $\mathbf{c}'$ denote the path cost found with λ_1 and λ_2, respectively.

Our results, summarized in Table 1, show that, for the same contraction ratio, the CHs produced by basic and batched have similar numbers of edges. However, basic needs much more preprocessing time because its number of witness searches increases dramatically with the contraction ratio. CHs produced by support-point have the largest number of edges because of the unnecessary shortcuts it adds, which also cause support-point to have a larger preprocessing time than batched for larger ($\geq 99.95\%$) contraction ratios. The results also show that contracting the last 0.05% of the states requires a large preprocessing time and results in a large number of additional edges.

With larger contraction ratios, the number of expanded nodes in the query phase decreases. In contrast, the average query time of BOA* with CHs increases because a large number of edges in the CH slow down the search algorithm. With the addition of partial expansions, the number of expanded nodes does not change, but the query times are reduced by up to a factor of 20. For the same contraction ratios, BOA* with CHs produced by support-point has a larger average query time than BOA* with CHs produced by batched due to the unnecessary edges that support-point adds.

Comparing different approaches in the query phase: We evaluate the scalability of our CH-based approach and the speedups it enables in the query phase. Here, we used seven road networks, whose numbers of states range from 1 million to 14 million, together with the batched approach and a contraction ratio of 99.95% (varying the contraction ratio and the ordering scheme is left to future work). For every road network, the number of edges in the CH is smaller than $2.5\times$ the number of edges in the input graph.

We evaluate three algorithms for the query phase: BOA*, BOA* with CHs (+CH), and BOA* combined CHs and partial expansion (+CH+p). The results are summarized in Table 2. All average and maximum values are taken over the instances solved by all three algorithms. The numbers of generated nodes are the numbers of nodes inserted into $Open$ (i.e., nodes that reach Line 18 of Alg. 1 or Line 26 of Alg. 4). We see dramatic reductions in the numbers of expanded nodes with CHs. While BOA*+CH and BOA*+CH+p expand the same number of nodes, BOA*+CH+p generates fewer nodes and hence has smaller average query times. This demonstrates that many nodes inserted into $Open$ by BOA*+CH are later pruned when popped from $Open$.

Figures 2a and 2b show the query times of BOA*+CH+p compared to BOA* and BOA*+CH on individual instances, respectively. The diagonal dashed lines and the numbers along them denote the minimum, median, and maximum speedups of BOA*+CH+p among instances solved by both algorithms. The query times of BOA*+CH+p are always smaller than those of BOA*, with a minimum speedup of 13 times and a maximum speedup of 1268 times.

Algorithm	#solved	t_{avg}	t_{max}	#exp	#gen
FLA (1.1M states and 2.7M edges)					
t_{prep}: 6min			$\lvert E_{\text{ch}}\rvert$: 5.5M		
BOA*	100	22.43	662.59	6,106K	7,507K
BOA*+CH	100	0.36	10.44	14K	151K
BOA*+CH+p	100	0.08	1.39	14K	45K
NE (1.5M states and 3.9M edges)					
t_{prep}: 21min			$\lvert E_{\text{ch}}\rvert$: 8.8M		
BOA*	100	56.05	1729.45	12,578K	16,281K
BOA*+CH	100	3.80	109.33	36K	939K
BOA*+CH+p	100	0.51	12.68	36K	211K
CAL (1.9M states and 4.7M edges)					
t_{prep}: 13min			$\lvert E_{\text{ch}}\rvert$: 9.4M		
BOA*	99	61.19	1617.97	14,923K	18,679K
BOA*+CH	100	1.62	43.12	28K	479K
BOA*+CH+p	100	0.29	5.70	28K	139K
LKS (2.8M states and 6.9M edges)					
t_{prep}: 1hr08min			$\lvert E_{\text{ch}}\rvert$: 16.5M		
BOA*	78	208.57	1631.61	46,928K	59,342K
BOA*+CH	96	23.94	192.44	117K	5,109K
BOA*+CH+p	100	2.83	24.51	117K	918K
E (3.6M states and 8.8M edges)					
t_{prep}: 42min			$\lvert E_{\text{ch}}\rvert$: 18.9M		
BOA*	79	269.10	1770.45	55,099K	69,055K
BOA*+CH	98	18.52	164.92	101K	3,306K
BOA*+CH+p	100	1.86	21.08	101K	659K
W (6.3M states and 15.2M edges)					
t_{prep}: 48min			$\lvert E_{\text{ch}}\rvert$: 29.6M		
BOA*	81	228.58	1784.56	60,605K	73,867K
BOA*+CH	100	4.24	27.98	100K	1,276K
BOA*+CH+p	100	0.80	4.28	100K	370K
CTR (14.1M states and 34.3M edges)					
t_{prep}: 5hr48min			$\lvert E_{\text{ch}}\rvert$: 77.5M		
BOA*	37	403.84	1634.94	87,751K	106,364K
BOA*+CH	83	28.31	223.18	165K	6,294K
BOA*+CH+p	100	3.01	19.91	165K	1,076K

Table 2: Experimental results for the query phases of different algorithms on different road networks. For each road network, we report the preprocessing times t_{prep} and the numbers of edges $|E_{\text{ch}}|$ in the CH. For each algorithm, we report the numbers of instances solved (#solved) within 30 minutes, the average (t_{avg}) and maximal (t_{max}) query times in seconds, and the average numbers of expanded (#exp) and generated (#gen) nodes.

Our experimental results also show that solving a bi-objective search instance directly can be more time-consuming than building a CH and solving it. This is because the runtime of solving a bi-objective search instance can be exponential in $|S|$ (because the size of the Pareto frontier can be exponential in $|S|$ (Ehrgott 2005; Breugem, Dollevoet, and van den Heuvel 2017)). This is in striking contrast to single-objective search, as a single-objective

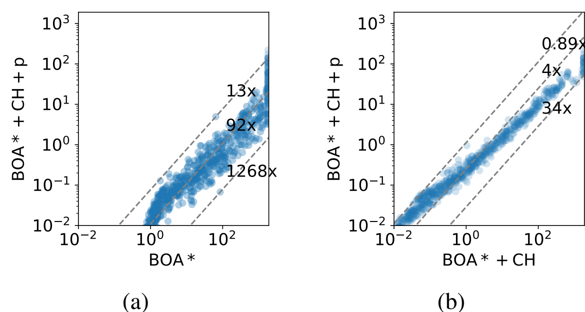

Figure 2: Query times (in seconds) on individual road network instances. The x-axes in (a) and (b) correspond to the runtimes of BOA* and BOA*+CH, respectively, while the y-axes correspond to the runtimes of BOA*+CH+p.

search instance can be solved in $O(|S| + |E|)$, while building a CH requires checking at least each contracted state and its incident edges. Therefore, building a CH and solving a single-objective search instance cannot be more efficient than solving the instance directly. Overall, our results show that CHs enable search algorithms to solve bi-objective search instances with far less computation time and memory in road networks.

Conclusions and Future Work

In this paper, we proposed a CH-based approach for efficiently computing Pareto frontiers for bi-objective search. Additionally, we proposed speedup techniques for both the preprocessing and query phases that are specifically designed for bi-objective search. Our experimental results demonstrated the scalability of our approaches to large road networks and orders-of-magnitude speedups in the query phase with all techniques combined. Interesting directions for future work include generalizing these speedup techniques to graphs with more than two costs and utilizing CHs to compute approximate Pareto frontiers (Zhang et al. 2022a; Rivera, Baier, and Ulloa 2022).

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. The research was also supported by the United States-Israel Binational Science Foundation (BSF) under grant number 2021643 and Centro Nacional de Inteligencia Artificial CENIA, FB210017, 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 any government.

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.

Baum, M.; Dibbelt, J.; Gemsa, A.; Wagner, D.; and Zündorf, T. 2015. Shortest Feasible Paths with Charging Stops for Battery Electric Vehicles. In *SIGSPATIAL International Conference on Advances in Geographic Information Systems*, 1–10.

Breugem, T.; Dollevoet, T.; and van den Heuvel, W. 2017. Analysis of FPTASes for the Multi-Objective Shortest Path Problem. *Computers & Operations Research*, 78: 44–58.

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.

Cohen, L.; Uras, T.; Jahangiri, S.; Arunasalam, A.; Koenig, S.; and Kumar, T. K. S. 2018. The FastMap Algorithm for Shortest Path Computations. In *International Joint Conferences on Artificial Intelligence (IJCAI)*, 1427–1433.

Delling, D.; and Wagner, D. 2009. Pareto Paths with SHARC. In *International Symposium on Experimental Algorithms (SEA)*, 125–136.

Ehrgott, M. 2005. *Multicriteria Optimization (2nd ed.)*. Springer.

Felner, A.; Goldenberg, M.; Sharon, G.; Stern, R.; Beja, T.; Sturtevant, N.; Schaeffer, J.; and Holte, R. 2012. Partial-Expansion A* with Selective Node Generation. In *AAAI Conference on Artificial Intelligence (AAAI)*, 471–477.

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.

Funke, S.; and Storandt, S. 2013. Polynomial-Time Construction of Contraction Hierarchies for Multi-Criteria Objectives. In *Workshop on Algorithm Engineering and Experiments (ALENEX)*, 41–54.

Geisberger, R.; Kobitzsch, M.; and Sanders, P. 2010. Route Planning with Flexible Objective Functions. In *Workshop on Algorithm Engineering and Experiments (ALENEX)*, 124–137.

Geisberger, R.; Sanders, P.; Schultes, D.; and Delling, D. 2008. Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In *International Workshop on Experimental and Efficient Algorithms*, 319–333.

Harabor, D.; and Stuckey, P. 2018. Forward Search in Contraction Hierarchies. In *Symposium on Combinatorial Search (SOCS)*, 55–62.

Maristany de las Casas, P.; Kraus, L.; Sedeño-Noda, A.; and Borndörfer, R. 2021. Targeted Multiobjective Dijkstra Algorithm. https://arxiv.org/abs/2110.10978.

Rivera, N.; Baier, J. A.; and Ulloa, C. H. 2022. Subset Approximation of Pareto Regions with Bi-objective A*. In *AAAI Conference on Artificial Intelligence (AAAI)*, 10345–10352.

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

Storandt, S. 2012. Route Planning for Bicycles—Exact Constrained Shortest Paths Made Practical via Contraction Hierarchy. In *International Conference on Automated Planning and Scheduling (ICAPS)*.

Sturtevant, N.; Felner, A.; Barrer, M.; Schaeffer, J.; and Burch, N. 2009. Memory-Based Heuristics for Explicit State Spaces. In *International Joint Conferences on Artificial Intelligence (IJCAI)*, 609–614.

Ulloa, C. H.; Yeoh, W.; Baier, J. A.; Zhang, H.; Suazo, L.; Koenig, S.; and Salzman, O. 2023. Simple and Efficient Bi-Objective Search Algorithms via Fast Dominance Checks. *Artificial intelligence*, 314: 103807.

Uras, T.; Koenig, S.; and Ulloa, C. H. 2013. Subgoal Graphs for Optimal Pathfinding in Eight-Neighbor Grids. In *International Conference on Automated Planning and Scheduling (ICAPS)*, 224–232.

Zhang, H.; Salzman, O.; Kumar, T. K. S.; Felner, A.; Ulloa, C. H.; and Koenig, S. 2022a. A* pex: Efficient Approximate Multi-Objective Search on Graphs. In *International Conference on Automated Planning and Scheduling (ICAPS)*, 394–403.

Zhang, H.; Salzman, O.; Kumar, T. K. S.; Felner, A.; Ulloa, C. H.; and Koenig, S. 2022b. Anytime Approximate Bi-Objective Search. In *Symposium on Combinatorial Search (SOCS)*, 199–207.