Efficient Multi-Query Bi-Objective Search via Contraction Hierarchies

Han Zhang¹, Oren Salzman², Ariel Felner³, T. K. Satish Kumar¹, Carlos Hernández Ulloa⁴, Sven Koenig³

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 a 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 3x 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.

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×. 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.

**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 \(G = (S,E)\), 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 out-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 \((src(e), tar(e'), c(e) + c(e'))\) 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 \(G_{ch} = (S, E_{ch})\) 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* \(lv(s) = i\), 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_min^2(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_min^2(s(n)) ≤ g_2(n) ∧ g_min^2(s_goal) ≤ f_2(n) then continue
9         g_min^2(s(n)) ← g_2(n)
10        if s(n) = s_goal then
11           add n to Sols
12           continue
13 for e ∈ out(s(n)) do
14         n′ ← node at tar(e) with g(n′) = g(n) + c(e)
15         if g_min^2(s(n′)) ≤ g_2(n′) ∧ g_min^2(s_goal) ≤ f_2(n′) then continue
16         add n′ to Open
17 return Sols

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 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 g-value of n, denoted as g(n), is the cost of this path, and the f-value of n is f(n) = g(n) + 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 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 g(n_root) = 0, and parent(n_root) = None (Lines 3-4). At each iteration, BOA* pops a node n with the lexicographically smallest 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 g(n′) ≤ 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 g(n′) ≤ 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 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)) ≤ g_2(n) and g_2^min(s_goal) ≤ f_2(n), respectively. If n is not pruned and s(n) = s_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), g(n′) = g(n) + 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 : \( G = (S,E), L \)
1 \( S_{ch} \leftarrow S; E_{ch} \leftarrow \{ \} \)
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             if \( \text{witness}_\text{search}(G, u, v; c(e) + c(e')) \) then
7                 add shortcut\((u, v; c(e) + c(e'))\)
8     add all edges incident on \( s \) to \( E_{ch} \)
9     remove \( s \) from \( S \) and all edges incident on \( s \) from \( E \)
10    add all remaining edges in \( E \) to \( E_{ch} \)
11 return \( G_{ch} = (S_{ch}, E_{ch}) \);
12
13 Function \( \text{witness}_\text{search}(G, u, v, p) \):
14     \( \pi \leftarrow \) a path from \( u \) to \( v \) whose cost dominates \( p \), or \( \text{none} \)
15     if no such path exists
16     return \( \text{true} \)
17 Function addShortcut\((e, v_{\text{sec}})\):
18     remove edges parallel to \( e_{\text{sec}} \) whose costs are weakly dominated by \( c(e_{\text{sec}}) \) from \( E \)
19     add \( e_{\text{sec}} \) to \( E \)
```

This algorithm works 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 \( \text{witness}_\text{search} \) to determine if there exists a path (witness) from \( \text{src}(e) \) to \( \text{tar}(e') \) whose cost dominates \( c(e) + c(e') \) (Lines 4-8). We omit the pseudocode of \( \text{witness}_\text{search} \) since it is based on BOA* with the following modifications:

- **Termination:** Once a witness is found, \( \text{witness}_\text{search} \) terminates and returns \( \text{true} \).
- **Pruning:** \( \text{witness}_\text{search} \) 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 \( s_{\text{goal}} \), 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 \( s_{\text{goal}} \) for each objective individually. This proves to be too time-consuming to do every time \( \text{witness}_\text{search} \) is invoked. Thus, we terminate the backward search once \( \text{src}(e) \) is expanded. Subsequently, the heuristic value for any state \( s \) is set to the minimum path cost from \( \text{tar}(e') \) to \( s \) on the reverse graph if \( s \) has been expanded or the minimum path cost from \( \text{tar}(e') \) to \( \text{src}(e) \) on the reverse graph otherwise. The resulting heuristic function is consistent.

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

After contracting \( L \) states, Alg. 2 returns a CH \( G_{\text{ch}} = (S_{\text{ch}}, E_{\text{ch}}) \) whose states \( S_{\text{ch}} \) consist of all states of the input graph and whose edges \( E_{\text{ch}} \) 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 \( G_{\text{ch}} \) 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 \( \text{witness}_\text{search} \) can be
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 \( \Pi \) 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 \text{witness}\_\text{search}\_\text{batch} to determine which paths in \( \Pi \) need to result in shortcuts (Line 7). Function \text{witness}\_\text{search}\_\text{batch} returns a subset of \( \Pi \), denoted as \( \Pi_{sc} \), which consists of all paths in \( \Pi \) that are not weakly dominated by any path from \( u \) to \( v \) that does not traverse \( s \).

Function \text{witness}\_\text{search}\_\text{batch}, like \text{witness}\_\text{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 \( \Pi \) 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 \text{witness}\_\text{search}\_\text{batch} removes path dominated by other paths in \( \Pi \) from \( \Pi \) 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 \text{witness}\_\text{search}\_\text{batch} also uses variable \( \Pi_{sc} \) (Line 16), initialized to \( \emptyset \), to store the paths that need to result in shortcuts.

Function \text{witness}\_\text{search}\_\text{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 \text{Open} with \( f_1(n) > f_1(q.top()) \), in which case the algorithm pops \( q.top() \) and adds it to \( \Pi_{sc} \) (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 \( f(n) \). 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 \( f(n) \), 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 \( \Pi_{sc} \)). 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 \text{witness}\_\text{search}\_\text{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 \text{Open} becomes empty, the algorithm adds all remaining paths in \( q \) to \( \Pi_{sc} \) and returns \( \Pi_{sc} \) (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 \text{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*. BOA* (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 полит for the input query (s_start, s_goal).  полит 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 полит to find a Pareto frontier. We denote  полит as 〈S = S^u \cup S^d, E〉, where S^u consists of all states that can be reached from s_start via an upward path and S^d consists of all states that can reach s_goal via a downward path. S^u and S^d are computed by running a depth-first search on G_ch and its inverse graph, respectively. E consists of (i) all upward edges with source states in S^u and (ii) all downward edges with target states in S^d.

Example 2. Figure 1c shows the search graph  полит constructed for query 〈C, D〉 and the CH in Figure 1b. State set S^u consists of C and E, and state set S^d 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 〈C, D〉, there are only two Pareto-optimal paths π_1 = 〈(C, E, (5, 5)), (A, E, (3, 5)), (E, B, (2, 2)), (B, D, (3, 3))〉 with a cost of (13, 15) and π_2 = 〈(C, E, (12, 9)), (E, B, (2, 2)), (B, D, (3, 3))〉 with a cost of (17, 14) in the graph in Figure 1a. These two paths have the same costs as paths π'_1 = 〈(C, E, (8, 10)), (E, D, (5, 5))〉 and π'_2 = 〈(C, E, (12, 9)), (E, D, (5, 5))〉 in the constructed search graph  полит, 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'}, ... , 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^{min}_{2}(s,s') to denote the minimum c_2-value of all edges from s to s', that is, c_2(e^{i}_{s,s'}). We highlight the major changes of Alg. 4 over

<table>
<thead>
<tr>
<th>Algorithm 4: BOA* with Partial Expansion</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> G = (S, E), s_start, s_goal, h</td>
</tr>
<tr>
<td>for s ∈ S do</td>
</tr>
<tr>
<td>g^{min}_{2}(s) ← \infty</td>
</tr>
<tr>
<td>if parent(n) \neq None then</td>
</tr>
<tr>
<td>generate_next(parent(n), s(n), idz(n) + 1)</td>
</tr>
<tr>
<td>if g_2(n) ≥ g^{min}<em>{2}(s(n)) \lor f_2(n) ≥ g^{min}</em>{2}(s_goal) then</td>
</tr>
<tr>
<td>continue</td>
</tr>
<tr>
<td>g^{min}_{2}(s(n)) ← g_2(n)</td>
</tr>
<tr>
<td>if s(n) = s_goal then</td>
</tr>
<tr>
<td>add n to Sols</td>
</tr>
<tr>
<td>continue</td>
</tr>
<tr>
<td>for s' ∈ out_neighbor(s(n)) do</td>
</tr>
<tr>
<td>generate_next(n, s', 1)</td>
</tr>
<tr>
<td>return Sols</td>
</tr>
</tbody>
</table>

*19 Function generate_next(n, s, i): |
*20 if g_2(n) + c^{min}_{2}(s(n), s) ≥ g^{min}_{2}(s) \lor g_2(n) + c^{2}_{s(n), s} + h_2(s) ≥ g^{min}_{2}(s_goal) then |
*21 return |
*22 for j = i, i + 1... m_{s,s'} do |
*23 n' ← new node at s with g(n') = g(n) + c_{s(n), s}, |
*24 idx(n') = j, and parent(n') = n |
*25 if g_2(n') ≥ g^{min}_{2}(s) \lor f_2(n') ≥ g^{min}_{2}(s_goal) then |
*26 continue |
*27 add n' to Open |
*28 return
BOA* by using “*” before line numbers. When expanding a node $n$, the algorithm uses $\text{generate\_next}$ for each out-neighbor $s$ of $s(n)$ (Line 17). Function $\text{generate\_next}$ 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 this child node, that is, $g_2(n) + c^2_{\text{min}}(s(n), s)$, is larger than or equal to $g_2^\ast(s)$ and if the minimum $f_2$-value of these child node, that is, $g_2(n) + c^2_{\text{min}}(s(n), s) + h_2(s)$, is larger than or equal to $f_2^\ast(s)$.

If such a node exists, $\text{generate\_next}$ iterates over the edges $e^1_{s,s'}, e^2_{s,s'}, \ldots e^m_{s,s'}$ until it finds the first edge that results in an undominated child node $n'$. Function $\text{generate\_next}$ then adds $n'$ to $\text{Open}$ and returns (Lines 22-27). For each node $n$, the algorithm uses $\text{idx}(n)$ to record the index of the edge that was used to generate it. When $n$ is popped from $\text{Open}$ and is not the root node, the algorithm calls $\text{generate\_next}$ to generate the next undominated child node of $\text{parent}(n)$ at state $s(n)$, if one exists (Line 9). When iterating over the edges from $s(\text{parent}(n))$ to $s(n)$, $\text{generate\_next}$ starts with the edge with index $\text{idx}(n) + 1$ because all edges with smaller indices have already been iterated over in the previous calls of $\text{generate\_next}$ for $\text{parent}(n)$. 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. 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 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 $\psi$, 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 $\text{basic}$ and $\text{batched}$, respectively. We also implement the approach proposed by Storandt (2012), referred to as $\text{support-point}$. For each in-neighbor $s'$ and out-neighbor $s''$ of state $s$, the witness search of $\text{support-point}$ runs a series of single-step queries.

![Table 1: Experimental results for different contraction approaches and contraction ratios on the NE and LKS maps.](image-url)
objective searches from \( s' \) to \( s'' \). Each single-objective search is parameterized by a \( \lambda \)-value and finds a path \( \pi' \) that minimizes \( \lambda c_1(\pi') + (1 - \lambda) c_2(\pi') \). For every 2-hop path \( \pi \) from \( s' \) via \( s \) to \( s'' \), if the witness search finds a path whose cost dominates \( c(\pi) \), then \( \pi \) 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 \( \lambda \)-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 \( c \) and \( c' \) denote the path cost found with \( \lambda _1 \) and \( \lambda _2 \), 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 \( \text{BOA}^* \) 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, \( \text{BOA}^* \) with CHs produced by support-point has a larger average query time than \( \text{BOA}^* \) 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: \( \text{BOA}^* \), \( \text{BOA}^* \) with CHs (+CH), and \( \text{BOA}^* \) 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 \( \text{BOA}^*+\text{CH} \) and \( \text{BOA}^*+\text{CH}+\text{p} \) expand the same number of nodes, \( \text{BOA}^*+\text{CH}+\text{p} \) generates fewer nodes and hence has smaller average query times. This demonstrates that many nodes inserted into Open by \( \text{BOA}^*+\text{CH} \) are later pruned when popped from Open.

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

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
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


