Karlsruhe Institute of Technology

Extending, Recombining and Evaluating Contraction Hierarchy based Many-to-Many Shortest Path Algorithms

Bachelor Thesis of

Theo Wieland

At the Department of Informatics
Institute of Theoretical Informatics

Reviewers:	PD Dr. Torsten Ueckerdt
	Prof. Dr. Peter Sanders
Advisors:	Tim Zeitz, M.Sc.

Time Period: 8th December 2021 – 8th April 2022

Statement of Authorship

Ich versichere wahrheitsgemäß, die Arbeit selbstständig verfasst, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde sowie die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet zu haben.

Karlsruhe, April 5, 2022

Abstract

This thesis focuses on efficient algorithms for the one-to-many and the many-to-many shortest path problem on continental-sized road networks. Shortest path distances from a set of source locations to a set of target locations are required by many real-world applications like map matching, the vehicle routing problem or ride-sharing. To efficiently calculate shortest path distances various speed-up techniques have been developed in the past. In this work, we study algorithms that use Contraction Hierarchies to speed up shortest path queries. We systematically examine and comprehensively evaluate existing algorithms for the one-to-many and the many-to-many shortest path problem. Additionally, we propose various recombinations and extensions of existing algorithms to efficiently process multiple source and target vertices simultaneously. We introduce a new algorithm for the established bucket-based many-to-many algorithm, that efficiently populates the buckets from multiple targets simultaneously. By using our proposed algorithm, we are able to beat the state of the art in different scenarios.

Deutsche Zusammenfassung

Diese Arbeit befasst sich mit effizienten Algorithmen um kürzeste Wege in statischen, kontinentalen Straßennetzen zu berechnen. Wir beschäftigen uns dabei mit Algorithmen, die alle Distanzen von einem Startpunkt zu einer Vielzahl an Zielpunkten oder alle Distanzen von einer Vielzahl an Startpunkten zu einer Vielzahl an Zielpunkten berechnen. Die Berechnung solcher Distanzen werden in vielen Anwendungen benötigt, wie z.B. Map Matching, Tourenplanung oder in Anwendungen die effizient Mitfahrgelegenheiten zuweisen. Wir betrachten hierbei ausschließlich Algorithmen, die auf Contraction Hierarchies aufbauen, um solche kürzesten Wege effizient zu berechnen. Wir implementieren und analysieren systematisch die Effektivität vieler bereits existierender kürzeste Wege Algorithmen. Darüber hinaus schlagen wir Rekombinationen und Erweiterungen existierender Algorithmen vor, deren Wirksamkeit wir in unseren Experimenten nachweisen. Alle Rekombinationen und Erweiterungen zielen darauf ab, effizient mehrere Start- und Zielpunkte gleichzeitig zu berücksichtigen. Zuletzt schlagen wir einen alternativen Algorithmus für den etablierten bucket-based Algorithmus vor, welcher die buckets effizient befüllt. In einigen Szenarien können wir darüber hinaus mit unserem vorgeschlagenen Algorithmus, existierende Algorithmen schlagen.

Contents

1. Introduction

Determining short routes is a problem that we humans encounter almost daily. Be it the way to work, to the nearest supermarket or to a desired vacation destination. Luckily, many applications exist that calculate such shortest paths in road networks for us, i.e. Apple Maps[1], Bing Maps[2], and Google Maps[3]. Nowadays, using such applications for finding one's way around has become natural.

In graph theory, the problem of computing an optimal route between two locations is referred to as the *shortest path problem*. Although the shortest path problem has been studied extensively in the past, there has been an increased interest in algorithms that answer such shortest path queries in real time due to recent trends in web-based map services and autonomous navigation systems.

To efficiently calculate shortest path distances in road networks several speed-up techniques have been developed in the past. These speed-up techniques are usually based on a preprocessing phase that computes some auxiliary data on the road network in advance. Given that the properties of the network are static, this auxiliary data can be integrated into a shortest path query to speed up the computation.

In addition to the shortest path distance between two given locations, some applications require all pairwise shortest path distances between two sets of locations. Unfortunately, most shortest path algorithms were designed to calculate shortest path distances from a single source location to a single target location.

To utilize *path prediction* [EFH+11], one must calculate the shortest path distances from a single source location to a set of target locations in order to anticipate a trajectory, assuming that the driver always takes the optimal route. *Ride sharing* applications must efficiently match a passenger who requests a ride with one of potential many drivers offering rides. Hence, one must identify the driver who has to take the smallest detour to meet the demands of a passenger. Identifying such a driver can be done with multiple one-to-many and point-to-point shortest path queries. Furthermore, some applications rely on efficient *many-to-many* shortest path queries. For example, the *travelling salesman problem* is concerned with the following task: Given a list of cities with shortest path distances between each pair of cities, what is the length of a minimal tour that visits every city once

[1] https://www.apple.com/maps/
[2] https://www.bing.com/maps
[3] https://www.google.com/maps

and returns to the origin city? Hence, in order to solve the travelling salesman problem, one must first calculate the shortest path distances between all pairs of cities. Additionally, the *vehicle routing problem* which is a generalization of the traveling salesman problem requires the same many-to-many shortest path distances. Lastly, in order to utilize *map matching* where one must map a series of imprecise locations to an existing path in the road network, one must calculate such many-to-many shortest path distances repeatedly.

1.1. Related Work

Although optimal route planning in road networks has been an ongoing topic of research [BDG+16], there has not been much attention on the many-to-many shortest path problem. Conventional solutions for the many-to-many shortest path problem mostly answer multiple queries from each source location independently.

Dijkstra's algorithm [Dij59] was the first to solve the shortest path problem for a single source location. Although Dijkstra's algorithm can be used to solve the one-to-many shortest path problem, in comparison to more efficient speed-up techniques, Dijkstra's algorithm settles too many vertices. Especially on continental-sized road networks, Dijkstra's algorithm is not practical for most applications.

Faster shortest path queries that do not rely on a preprocessing can be achieved by goal-directed algorithms that guide the search towards the goal location. $A*$ [HNR68] is a goal-directed algorithm that uses a heuristic to efficiently guide a shortest path query towards a single target location. Although it is straightforward to guide a point-to-point shortest path query towards a given target location, guiding the search towards multiple targets is not as easy.

As previously stated, further improvements are mainly achieved by speed-up techniques that calculate some auxiliary data on the network during a preprocessing. Using such a preprocessing phase reduces the flexibility but allows for faster shortest path calculations.

Contraction Hierarchies [GSSD08] is a speed-up technique that has been successfully applied on road networks. By utilizing added shortcut edges and a hierarchy, shortest path queries can be efficiently answered. Various algorithms, like PHAST [DGNW10], RPHAST [DGW11] and the bucket-based algorithm as proposed by Knopp et al. [KSS+07], exist that use Contraction Hierarchies as their speed-up technique.

The bucket-based algorithm as proposed by Knopp et al. [KSS+07], was one of the first to specifically solve the many-to-many shortest path problem. The algorithm was initially based on Highway Hierarchies [SS06], but can be applied to any hierarchical speed-up technique like Contraction Hierarchies as stated by Delling et al. [DGW11]. The bucket-based algorithm is able to efficiently calculate all shortest path distances from a set of source vertices S to a set of target vertices T, without having to perform individual $|S| \cdot |T|$ point-to-point shortest path queries.

PHAST [DGNW10] is an extension of Contraction Hierarchies that is better suited for modern computer architectures as it accesses memory more efficiently by taking data locality into account. As stated by Delling et al. [DGNW10], PHAST is able to calculate the shortest path distances from one source location to all vertices in the graph with essentially a single linear sweep over all edges.

The RPHAST [DGW11] algorithm is currently one of the fastest algorithm to calculate shortest path distances from one source location to a set of target locations. Although Delling et al. in [DGW11] mainly considered algorithms for the one-to-many problem, they also covered algorithms for the many-to-many problem briefly. SSE RPHAST, an

algorithm that uses SIMD instructions to batch multiple sources in one RPHAST query, is currently the fastest algorithm to calculate all many-to-many shortest path distances in many scenarios. Our work builds on and extends the ideas as present by Delling et al. [DGW11], whereby we place a greater focus on the many-to-many problem.

Hub Labels [ADGW10] currently allow for the fastest point-to-point shortest path queries on continental-sized road networks after a preprocessing. To make the algorithm practical on continental-sized road networks, Hub Labels can be realized with Contraction Hierarchies [ADGW10]. Using Hub Labels to speed-up one-to-many or many-to-many shortest path queries has not received much attention. A trivial approach to use Hub Labels to calculate one-to-many or many-to-many shortest path distances is to perform multiple point-to-point shortest path queries.

Different approaches to the many-to-many problem that do not require a preprocessing phase include the clustered MSP algorithm, as proposed by Jagadeesh and Srikanthan [JS19]. The clustered MSP algorithm identifies a set of exit vertices that must be traversed by a shortest path from any clustered source to vertices outside a given range. Such exit vertices are used to speed up all many-to-many shortest path queries.

1.2. Contribution

In this thesis, we systematically examine existing Contraction Hierarchy based algorithms for the one-to-many and the many-to-many shortest path problem. Moreover, we comprehensively evaluate the performance of all algorithms in different scenarios on continental-sized road networks.

We propose various recombinations and extensions of existing algorithms to efficiently process multiple source and target vertices simultaneously. Furthermore, we propose a new algorithm that improves the bucket-based many-to-many algorithm as initially proposed by Knopp et al. [KSS$^+$07]. Based on our proposed algorithm, we are able to efficiently solve the many-to-many problem.

1.3. Outline

This thesis is structured in four chapters.

We introduce preliminaries in Chapter 2, additionally we cover the basic notation and formally define the problem that we study in this thesis. Furthermore, we discuss Contraction Hierarchies. In Chapter 3, we describe all one-to-many and many-to-many algorithms implemented for this thesis in detail. Moreover, we present various modifications as well as recombinations of existing algorithms in Chapter 3. Lastly, we present the results of our experiments in Chapter 4. We conduct a series of one-to-many and many-to-many experiments that aim to highlight the key differences between all algorithms.

2. Preliminaries

2.1. Notation

Formally, we define a directed graph $G = (V, E)$ as a tuple of vertices V and edges $E \subseteq V \times V$. Let $n := |V|$ be the number of vertices and $m := |E|$ the number of edges in the graph G.

A road network can be described by a directed graph $G = (V, E)$. Each vertex $v \in V$ represents an intersection and each edge $(u, v) \in E$ represents a road segment. Furthermore, the time needed to travel along a given road segment $(u, w) \in E$ can be described by a weight function $\omega : E \to \mathbb{R}_{\geq 0}$.

Given two vertices $s, t \in V$, we specify a path from s to t as a list of vertices $P := (v_0 = s, \ldots, v_n = t)$ where $(v_i, v_{i+1}) \in E$. The weight function can be extended to paths as follows: $\omega(P) := \sum_{i=0}^{n-1} \omega(v_i, v_{i+1})$. Let $D(s, t)$ be the minimal weight of a shortest path from s to t in G. In case there is no path from s to t, we define $D(s, t) := \infty$.

2.2. Dijkstra's Algorithm

A common approach to compute shortest paths in graphs is to use Dijkstra's algorithm [Dij59]. Dijkstra's algorithm calculates shortest paths from a single source vertex to all vertices. Given a graph $G = (V, E)$ as well as a source vertex $s \in V$, Dijkstra's algorithm grows a tree of shortest paths starting from s. The algorithm maintains a distance array $d[v]$ that stores the tentative distance from s to v. Furthermore, the algorithm maintains an array of parent vertices $p[v]$ that stores the parent of v on a shortest path from s to v. Lastly, a priority queue Q that contains *unsettled* vertices ordered by minimum distance $d[v]$ is required by the algorithm.

Initially, the distance value $d[v]$ is set to ∞ and the parent $p[v]$ to $\perp$ for all vertices $v \in V$. Moreover, the distance value $d[s]$ is set to zero and the parent vertex $p[s]$ is set to s for the source vertex s. Lastly, the priority queue Q is initialized with s.

In each iteration, a vertex v with minimum distance $d[v]$ is removed from Q and *settled* by *relaxing* every edge: The algorithm examines every edge $(v, w) \in E$ as follows: If $d[v] + \omega(v, w) < d[w]$, $d[w]$ is updated by assigning $d[v] + \omega(v, w)$ to it. Furthermore, in such a case the parent $p[w]$ is updated to v. Lastly, w is either added to Q or has its position updated according to the reduced tentative distance $d[w]$.

We refer to the *search space* as the set of all vertices settled by the algorithm.

The algorithm can be extended to stop early, by checking if all relevant target vertices have been settled. Once a vertex has been settled, its associated distance value $d[v]$ is final and holds the shortest path distance from s to v.

Although Dijkstra's algorithm has a theoretical running time of $\mathcal{O}(n \log n + m)$ by using Fibonacci heaps [FT87], it is not usable in modern routing planning applications as it is too slow on continental-sized road networks.

Algorithm 2.1: DIJKSTRA

Input: Graph $G = (V, E)$, weight function ω and a source vertex s
Data: Priority queue Q ordered by minimum distance $d[v]$
Output: tentative distances $d[v]$, shortest-path tree from s given by $p[\cdot]$

```
// Initialization
forall v ∈ V do
    d[v] ← ∞
    p[v] ← ⊥
Q.INSERT(s, 0)
d[s] ← 0
p[s] ← s
// Main loop
while Q is not empty do
    u ← Q.DELETEMIN()
    forall (u, v) ∈ E do
        if d[u] + ω(u, v) < d[v] then
            d[v] ← d[u] + ω(u, v)
            p[v] ← u
            if Q.CONTAINS(v) then
                Q.DECREASEKEY(v, d[v])
            else
                Q.INSERT(v, d[v])
```

2.3. Contraction Hierarchies

Contraction Hierarchies (CH) [GSSD08] is a speed-up technique that can be used to efficiently calculate shortest paths in road networks. The fundamental concept of a CH is to introduce a hierarchy that ranks all vertices in increasing order of importance. Based on such a ranking and a corresponding *augmented* graph G^+, shortest path queries can be answered by only settling a few vertices, compared to Dijkstra's algorithm. A ranking is based on the idea that some vertices and adjacent edges are more important than others. For example, a highway that connects two cities that are far apart can be more important than two intersections that share a connecting road inside a local neighborhood . Nevertheless, for the query algorithm to calculate correct shortest path distances, any ranking is applicable, though query times can vary with respect to different orders of importance.

2.3.1. Preprocessing

The preprocessing phase computes an *augmented* graph G^+ based on the importance of each vertex. We assume that the importance of each vertex is known in advance, otherwise

it is also possible to dynamically rank the vertices while calculating G^+. We denote the importance of each vertex v with $rank(v)$. In case a vertex v is more important than another vertex w we require that $rank(v) > rank(w)$. Lastly, $rank(\cdot)$ must define a strict total order among all vertices.

We calculate G^+ by iteratively *contracting* a vertex v, such that v has the lowest rank among all vertices that have not been contracted yet. Initially, the augmented graph $G^+ := (V, E^+)$ is equal to the original graph $G = (V, E)$. To contract v, the algorithm assumes that v is no longer present in G^+ by also preserving all shortest path distances. Hence, we add *shortcut* edges to G^+, in order to preserve shortest path distances among the remaining vertices. Shortcut edges are added based on the following condition: If for any *incoming* edge $(u, v) \in E^+$ into v and any *outgoing* edge $(v, w) \in E^+$ from v, a shortest path from u to w contains v, a shortcut edge (u, w) must be added to E^+. In doing so, we can use the added shortcut edge to bypass v during a shortest path query. The length of the added shortcut edge (u, w) is equal to $\omega(u, v) + \omega(v, w)$. The process of checking if (u, v, w) is the only shortest path from u to w is called a *witness search*. A shortcut edge is not required if the witness search finds a shorter path from u to w that does not contain v. During each vertex contraction we assume that only the vertices that have not been contracted are present in the graph. Hence, we only examine all incoming and outgoing edges that start or lead to vertices that have not yet been contracted. Furthermore, the witness search only settles vertices that have not been contracted.

2.3.2. Query

As previously stated, a shortest path query can be efficiently answered by only settling a few vertices in G^+ based on the added shortcuts and the relative rank of each vertex. Formally, we introduce the *upward* graph $G^\uparrow$ as the graph that contains exclusively edges leading to vertices with a higher rank, hence $E^\uparrow := \{(u, v) \mid (u, v) \in E^+,\ rank(u) < rank(v)\}$. Likewise, the *downward* graph $G^\downarrow$ contains exclusively edges leading to vertices with a lower rank, hence $E^\downarrow := \{(u, v) \mid (u, v) \in E^+,\ rank(u) > rank(v)\}$. G^+ is equal to both $G^\uparrow$ and $G^\downarrow$ combined.

The query algorithm uses a bidirectional version of Dijkstra's algorithm. The bidirectional query algorithm runs two instances of Dijkstra's algorithm, to settle the vertices in G^+. A *forward* search runs on $G^\uparrow$ while a *backward* search runs on $G^\downarrow$. The backward search settles vertices in $G^\downarrow$ by using Dijkstra's algorithm on the reversed graph $\overleftarrow{G}^\downarrow := (V, \overleftarrow{E}^\downarrow)$, where $\overleftarrow{E}^\downarrow := \{(w, v) \mid (v, w) \in E^\downarrow\}$. Additionally, the bidirectional search keeps track of a vertex u that minimizes $\mu = dist(s, u) + dist(u, t)$. Initially, μ is set to ∞. We denote the shortest path distance from s to u in $G^\uparrow$ with $dist(s, u)$. Likewise, we use $dist(u, t)$ to denote the shortest path distance from u to t in $G^\downarrow$. Furthermore, we want to emphasize that $dist(s, u)$ and $dist(u, t)$ may not be equal to the shortest path distance $D(s, u)$ and $D(u, t)$ in G respectively.

We now describe the bidirectional query algorithm in detail. The query algorithm alternates between settling vertices from the forward search and the backward search. While settling a vertex v, each search checks if v has also been reached by the other search. If so, the algorithm updates u and μ: If $d^\uparrow[v] + d^\downarrow[v] < \mu$, the algorithm assigns $\mu = d^\uparrow[v] + d^\downarrow[v]$ and $u = \mu$. We denote the distance array of the forward search with $d^\uparrow[v]$, hence the distance from s to v in $G^\uparrow$, as calculated by the forward search, is equal to $d^\uparrow[v]$. Likewise, $d^\downarrow[v]$ denotes the distance calculated by the backward search. $d^\downarrow[v]$ is equal to the shortest path distance from t to v in $\overleftarrow{G}^\downarrow$, which is equal to the shortest path distance from v to t in $G^\downarrow$. Lastly, the algorithm aborts as soon as the minimum distance of both priority queues is greater than μ or both priority queues are empty. The shortest path distance from s to t in G^+ is equal to μ.

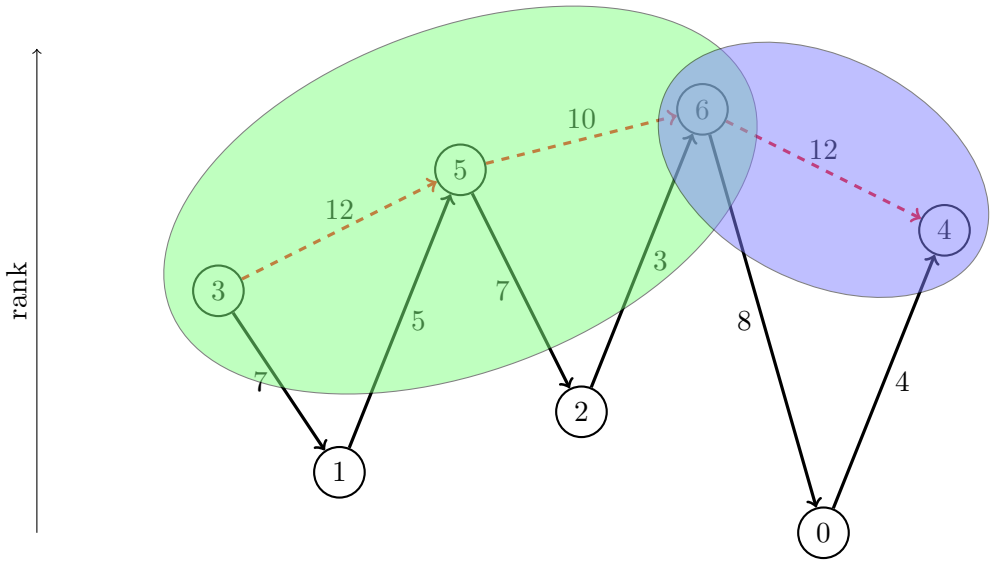


Figure 2.1.: Contraction Hierarchy query from v_3 to v_4.

We refer to Figure 2.1 for a schematic representation of a query, where v_3 is the source vertex and v_4 is the target vertex. Shortcuts that were added during the preprocessing are shown with a dashed line. Vertices visited by the forward search in $G^{\uparrow}$ from v_3 are highlighted in green. Likewise, vertices visited by the backward search in $G^{\downarrow}$ from v_4 are highlighted in blue. To calculate the shortest path distance from v_3 to v_4 we must only settle v_3, v_5, v_6 and v_4. The only edges examined are the three added shortcut edges.

Theorem 2.1. *The CH query algorithm calculates correct shortest path distances from s to t in G.*

Proof. Given that the forward search calculates only *upward-paths* and the backward search only calculates *downward-paths* we must prove that G^+ contains a shortest path from s to t that is an *upward-downward* path. Furthermore, the *upward-downward* path must be of equal length to a shortest path in G.

Let P be a shortest path from s to t in G^+. If P is an upward-downward path we have nothing to prove. Otherwise, we assume that P is not an upward-downward path. Hence, there must exist a vertex v in P such that both its predecessor u and successor w have a higher rank than v. In this case, by definition of the CH preprocessing, v was contracted before u and w. Hence, either a shortcut edge from u to w was added to G^+, or the witness search found a shortest path from u to w by only settling vertices with a higher rank than v in G^+. This contradicts the assumption that P is not an upward-downward path.

Additionally, because G^+ was constructed by only adding shortcut edges, the calculated distance is equal to the shortest path distance in the original graph G. □

2.4. Many-to-Many Problem

The *many-to-many problem* is the problem of computing all shortest path distances from a set of source vertices to a set of target vertices.

Formally, we receive a directed and weighted graph $G = (V, E)$, a non-empty set of sources $S \subseteq V$ as well as a non-empty set of targets $T \subseteq V$. The goal is to compute a distance table $\mathbb{R}^{|S| \times |T|}$ that stores the shortest path distance from each $s \in S$ to all $t \in T$ denoted

by $D[s, t]$. Naively computing all shortest path distances requires $|S| \cdot |T|$ individual point-to-point shortest path queries. Fortunately, multiple algorithms exist that compute this distance table much more efficiently. In the following work we are going to cover such algorithms that efficiently solve the many-to-many problem.

Other similar problems are the *one-to-many problem*, the *many-to-one problem* and the *point-to-point problem*. The *one-to-many problem* is the problem of calculating shortest paths from one source vertex to a set of target vertices. Likewise, the *many-to-one problem* is the problem of calculating shortest paths from multiple sources to one target vertex. Lastly, the *point-to-point problem* is the problem of calculating a shortest path from a single source vertex to a single target vertex.

3. Algorithms

In this chapter, we discuss all algorithms implemented for this thesis in detail. Furthermore, we present various modifications as well as recombinations of existing algorithms.

All algorithms are built on top of a CH preprocessed graph. We therefore assume that the CH preprocessing was executed in advance. Each algorithm is supplied with the augmented graph, $G^{\uparrow}$ and $G^{\downarrow}$. Moreover, a set of source vertices $S \subseteq V$ and a set of target vertices $T \subseteq V$ are given as input to each algorithm.

The goal of each algorithm is to compute all shortest path distances $D(s,t)$, for all $(s,t) \in S \times T$. We store all shortest path distances in a two-dimensional array denoted by $D[s,t]$. We are only interested in the distance of a shortest path and not the shortest path by itself.

3.1. Implementation

3.1.1. Graph Representation

Given that all algorithms need to efficiently traverse a graph, we need a data representation that allows efficient access to adjacent vertices and associated edge weights. We use the *adjacency array* representation that stores the graph in a pair of arrays. We refer to these arrays as the *first* array and the *arclist* array.

We assume that vertices have sequential IDs from 0 to $n-1$.

To represent $G^{\uparrow}$, we store each edge together with its associated weight in the *arclist* array. Each edge $(v,u) \in E^{\uparrow}$ is stored as a pair containing the ID of the head vertex u and the weight $\omega(v,u)$ associated with that edge. Furthermore, we sort the *arclist* array in ascending order based on the tail ID of each edge. By following this approach, all outgoing edges starting from a given vertex v are stored consecutively in the *arclist* array. Finally, we store the position of the first outgoing edge from v in the *arclist* array in $\mathit{first}[v]$. In doing so, we can iterate over the *arclist* array from $\mathit{first}[v]$ to $\mathit{first}[v+1]$ in order to access all outgoing edges from v. Additionally, $\mathit{first}[n]$ is set to m to avoid special cases.

We take a slightly modified approach to store $G^{\downarrow}$. A CH backward search in $G^{\downarrow}$ needs to efficiently access all *incoming* edges that lead to a given vertex v. Hence, we cannot use the *adjacency array* representation as is. Instead of storing $G^{\downarrow}$, we store the reversed graph $\overleftarrow{G}^{\downarrow}$, $\overleftarrow{G}^{\downarrow} := (V, \overleftarrow{E}^{\downarrow})$ in the same way as we store $G^{\uparrow}$. Edges in $\overleftarrow{G}^{\downarrow}$ are equal to: $\overleftarrow{E}^{\downarrow} := \{(w,v) \mid (v,w) \in E^{\downarrow}\}$.

To further improve memory locality and reduce the number of cache misses, vertex IDs can be permuted appropriately. We permute vertex IDs based on their *rank*, such that vertex ID = *rank*.

3.2. Buckets

3.2.1. Baseline Algorithm

The bucket-based algorithm, as proposed by Knopp et al. [KSS+07], was one of the first to efficiently answer many-to-many shortest path queries. The algorithm was initially based on Highway Hierarchies [SS06], but can be applied to any hierarchical speed-up technique like Contraction Hierarchies as stated by Delling et al. [DGW11].

The main concept behind the algorithm is to perform the CH forward and backward searches only once for each source and target vertex. Naively computing all required shortest path distances requires $|S| \cdot |T|$ forward and backward searches.

The bucket-based algorithm maintains a bucket $B(v)$ for each vertex. Initially each bucket $B(v)$ is empty, and later populated during the *target selection* phase. Each bucket can store multiple pairs, consisting of a target vertex t and the shortest path distance from v to t in $G^{\downarrow}$, denoted by $dist(v,t)$. Additionally, the bucket algorithm directly modifies the distance values for each $(s,t) \in S \times T$ in $D[s,t]$. Initially, all entries in the distance array D are set to ∞.

Buckets are populated through multiple CH backward searches during the target selection phase. We perform one CH backward search from each $t \in T$ in $G^{\downarrow}$ consecutively. For every settled vertex v, during such a backward search from t, a pair $(t, dist(v,t))$ is added to $B(v)$. Hence, once finished with all backward searches, each bucket $B(v)$ contains multiple pairs $(t, dist(v,t))$. One pair each for all targets t that can be reached from v in $G^{\downarrow}$.

During the subsequent *query phase* the information available in each bucket is used to simulate $|S| \cdot |T|$ individual shortest path queries. The query algorithm performs $|S|$ separate one-to-many queries to calculate all shortest path distances. Each one-to-many query performs a CH forward search from a different $s \in S$ in $G^{\uparrow}$ to settle vertices. While settling a vertex v, the bucket contents of the associated bucket $B(v)$ are used to update the tentative distances in D. More precisely, for each bucket entry $(t, dist(v,t)) \in B(v)$, $D[s,t]$ is updated by assigning $D[s,t] = \min\{D[s,t],\ dist(s,v) + dist(v,t)\}$.

Once the algorithm is finished with all CH forward searches, $D[s,t]$ is equal to the shortest path distance $D(s,t)$ from s to t in G, for all $(s,t) \in S \times T$.

3.2.2. Stall-on-Demand

In the following, we discuss and illustrate cases where some bucket entries can be removed without jeopardizing the correctness of the algorithm.

Because $G^{\downarrow}$ is constructed as a subgraph of G^{+} that only contains selected edges, a shortest path from v to t in $G^{\downarrow}$ might be longer than a shortest path from v to t in G^{+}. Hence, such entries where the associated distance $dist(v,t)$ is greater than the shortest path distance $D(v,t)$, from v to t in G^{+}, only require unnecessary scanning time. We apply a fast heuristic, similar to the stall-on-demand technique [SS07], to identify and remove most unnecessary bucket entries.

We check for bucket entries with unnecessary distance values by using a one-step lookahead. We implement the following procedure into the target selection phase, during which we perform a CH backward search from each $t \in T$ to populate the buckets. Let v be the

currently settled vertex with associated shortest path distance $dist(v, t)$ in $G^{\downarrow}$ during such a backward search from $t \in T$. If for any outgoing edge $(v, w) \in G^{\uparrow}$, $\omega(v, w) + d[w] < dist(v, t)$ is true, we know that $dist(v, t) > D(v, t)$. Hence, a shorter path from v to t exists in G^{+} that traverses w before reaching t instead of going from v to t without traversing w.

Figure 3.1 depicts this concept. Because vertices are settled in increasing order of minimum distance, w is settled before v during a backward search from t. Thus, $d[w] = 1$ while we settle v. Consequently, while settling v, $\omega(v, w) + d[w] < dist(v, t)$ is true. Hence, we do not have to add a bucket entry to $B(v)$. Additionally, we do not relax any edges otherwise relaxed from v, as calculated distances would be unnecessary as well. Without the stall-on-demand technique a bucket entry (t, 10) would be added to $B(v)$.

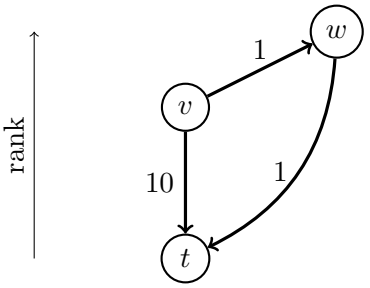


Figure 3.1.: Stall-on-Demand visualization.

3.2.3. Simultaneous Bucket Initialization

Rather than performing a separate backward search for each target $t \in T$, we propose an algorithm to simultaneously populate the buckets for all targets T at once. Our proposed algorithm only replaces the target selection phase of the bucket-based algorithm in Section 3.2.1. The query phase remains unchanged.

First, we give an outline of the main concept behind the algorithm, followed by a more detailed explanation. Additionally, we refer to Algorithm 3.1 for the pseudocode of the algorithm. The algorithm settles vertices in increasing rank order. For each settled vertex, we copy bucket entries from previously settled adjacent vertices to the current vertex. Furthermore, we merge bucket entries from all adjacent vertices by adding only one entry to the bucket of the current vertex for each discovered target. The distance associated with each target is equal to the shortest path distance to that target in $G^{\downarrow}$.

To access all previously settled adjacent vertices, we need to modify how we access $G^{\downarrow}$. Unfortunately, we cannot use the adjacency array representation to access all adjacent vertices as intended. The *first* and *arclist* array that store $G^{\downarrow}$ only allow efficient access to all incoming edges. By using the adjacency array, we can only efficiently iterate over all edges $(u, v) \in E^{\downarrow}$ where $rank(u) > rank(v)$ for a given vertex v. However, to copy bucket entries, we need to access all edges $(v, w) \in E^{\downarrow}$ where $rank(v) > rank(w)$ for a given vertex v. Instead of modifying how $G^{\downarrow}$ is stored in the *first* and *arclist* array, we propose a different approach that also reduces the number of accessed edges. By following our approach we only store selected edges, namely those edges leading to vertices that are in the combined search space as discovered by all CH backward searches from all targets T in $G^{\downarrow}$. In case we were to store the entire graph $G^{\downarrow}$ alternatively we would access more edges compared to our approach. We maintain an *adjacency list* denoted by *DownVertices*(v) to access edges as required. *DownVertices*(v) is initialized as an empty list and populated as follows: While settling a vertex v, we use the usual adjacency array representation to access all incoming edges $(u, v) \in E^{\downarrow}$. Given u, with $rank(u) > rank(v)$, we add a pair $(v, \omega(u, v))$ to *DownVertices*(u). Thus, *DownVertices*(u) can be used to access all outgoing

edges from u that lead to vertices with a lower rank. Because we settle vertices in increasing rank order, *DownVertices*(u) is fully populated as soon as we settle u.

Next we are going to elaborate on an efficient algorithm to efficiently merge all bucket entries from previously settled adjacent vertices. Moreover, we efficiently calculate the shortest path distance to each target by iterating over each adjacent bucket only once. We initially create an empty list T' and a distance array $d[t]$. T' is used to mark all targets $t \in T$ such that a path from v to t exists in $G^{\downarrow}$, where v is the vertex we currently settle. Additionally, we store the tentative shortest path distance from v to t in $d[t]$, initially $d[t]$ is set to ∞. During the initialization, we add pair $(t, 0)$ to $B(t)$ for each $t \in T$. By adding such a pair, we indicate that t can be reached from t, with a shortest path of length zero. Lastly, we initialize a priority queue Q ordered by minimum rank $rank(v)$ with all targets $t \in T$. In each iteration, a vertex v with minimum rank is removed from Q and settled. We perform the following steps to settle v: First we iterate over all edges $(u, v) \in E^{\downarrow}$ and add u to Q in case it has not been added before. Additionally, we add v to *DownVertices*(u) as previously stated. In order to prepare for copying bucket entries we need to access all buckets of vertices adjacent to v. Iterating over all entries $(w, \omega(v, w)) \in$ *DownVertices*(v) reveals all adjacent vertices. Accessing $(t, dist(w, t)) \in B(w)$ for each w points us to all available bucket entries to copy. The algorithm processes each discovered t as follows: If $t \notin T'$, t is added to T'. Moreover, $d[t] = \min\{d[t],\ \omega(v, w) + dist(w, t)\}$, no matter the previous condition. Once finished with all w, T' contains all targets t such that v has a path to t in $G^{\downarrow}$. The associated shortest path distance from v to t is stored in $d[t]$. Hence, we add a pair $(t, d[t])$ to $B(v)$ for each $t \in T'$. Finally, we reset the tentative distances by setting $d[t]$ to ∞ for all $t \in T'$. Lastly, we remove all entries from T', in order to reuse T' during the following iteration.

Retrospective Pruning

Although the simultaneous bucket initialization efficiently populates the buckets from all targets T at once, it lacks a pruning criterion. Without removing unnecessary bucket entries, a shortest path query spends too much time on scanning each bucket. Again, we use a one-step lookahead to prune bucket entries with unnecessary distance values. The idea is the same as for the stall-on-demand technique that we applied to the bucket initialization as proposed by Knopp et al. in Section 3.2.2. However, because we settle vertices in increasing rank order we must adjust our approach. By settling vertices in increasing rank order, we cannot rely on vertices with a higher rank to prune bucket entries, as they have not been initialized yet. Hence, we cannot look at outgoing upward edges to prune bucket entries directly. Instead, we prune buckets of previously settled vertices retrospectively. While settling a vertex v, we want to access all vertices u adjacent to v via an incoming edge $(u, v) \in E^{\uparrow}$, where $rank(v) > rank(u)$. Given such a vertex u, we use the tentative shortest path distances $d[t]$ to check for the following condition: If $B(u)$ contains a bucket entry $(t, dist(u, t))$ with $dist(u, t) > \omega(u, v) + d[t]$ we can remove t from $B(u)$. Because v is the vertex we currently settle, $d[t]$ is equal to the shortest path distance from v to t in $G^{\downarrow}$.

Given that we need to access $d[t]$, we first initialize $d[t]$ as mentioned in Section 3.2.3. As soon as $d[t]$ is initialized, we use $d[t]$ to prune bucket entries as stated. To efficiently access all u, with $(u, v) \in E^{\uparrow}$ where $rank(u) < rank(v)$, we maintain another adjacency list *UpVertices*(v). *UpVertices*(v) follows the same concept as *DownVertices*(v), but stores edges $(u, v) \in E^{\uparrow}$. Unfortunately, we also initialize *UpVertices*(v) for vertices that we do not necessarily settle. Hence, we must clear *UpVertices*(v) appropriately, in case we reuse the data structure for multiple many-to-many queries. We can clear *DownVertices*(v) after settling v, so that *DownVertices*(w) is empty for all $w \in V$ after the algorithm is finished.

Algorithm 3.1: Simultaneous Bucket Initialization

Input: $G^{\downarrow}$ and $G^{\uparrow}$, weight function ω and a set of target vertices $T \subseteq V$
Data: Priority queue Q ordered by minimum rank, Terminals, distance array $d[t]$, *DownVertices*(v), *UpVertices*(v)
Output: The populated buckets B(v)

```
// Initialization
forall v ∈ V do
    initialize an empty bucket B(v)
forall t ∈ T do
    Q.INSERT(t, RANK(t))
    B(t).APPEND((t, 0))
    d[t] ← ⊥

// Main loop
while Q is not empty do
    v ← Q.DELETEMIN()
    // Initialize DownVertices(u)
    forall (u, v) ∈ E↓ do
        DownVertices(u).APPEND((v, ω(u, v)))
    // Initialize UpVertices(u)
    forall (v, u) ∈ E↑ do
        UpVertices(u).APPEND((v, ω(v, u)))
    // Discover bucket entries to copy
    forall (w, ω(v, w)) ∈ DownVertices(v) do
        forall (t, dist(w, t)) ∈ B(w) do
            if d[t] ≠ ⊥ then
                Terminals.APPEND(t)
            d[t] ← min{d[t], ω(v, w) + dist(w, t)}
    // Apply the retrospective pruning algorithm
    forall (w, ω(w, v)) ∈ UpVertices(v) do
        forall (t, dist(w, t)) ∈ B(w) do
            if dist(w, t) > ω(w, v) + d[t] then
                B(w).REMOVE((t, dist(w, t)))
    // Copy bucket entries to the current vertex
    forall t ∈ Terminals do
        B(v).APPEND((t, d[t]))
        d[t] ← ⊥
    Terminals.CLEAR
```

3.2.4. Pruning Effectiveness

In this section we compare the effectiveness of the stall-on-demand technique (Section 3.2.2) against the retrospective pruning that we applied to the simultaneous bucket initialization (Section 3.2.3). Although both algorithms reduce the number of bucket entries by roughly 80 percent, in some scenarios one algorithm produces fewer bucket entries than the other. However, no algorithm is strictly better than the other. We refer to Section 4.5 for an experimental analysis of the effectiveness of different pruning algorithms.

In the following we discuss the conditions for differences in bucket sizes to occur. In both illustrations, Figure 3.2 and Figure 3.3, the two algorithms are compared against each other. Each algorithm performs the initialization for a single target vertex. In this case v_0 is the target vertex. Each vertex is displayed with its vertex ID and the contents of its bucket. We denote each bucket entry as a pair that contains the ID of the target vertex together with the associated distance to that target (notation: (*ID*, *distance*)). Vertices are ordered in increasing rank order along the vertical axis. Also, each vertex ID is equal to its rank.

In Figure 3.2 the stall-on-demand technique produces a total of two bucket entries. However, the simultaneous bucket initialization produces an additional entry that is not necessarily required. The bucket entry $(0, 11) \in B(v_2)$ is not required, as a shorter path from v_2 to v_0 is discovered by continuing the forward search until settling v_4.

Because the stall-on-demand technique settles vertices in increasing order of minimum distance, vertices are settled in the following order: Initially the algorithm starts with v_0, proceeds with v_4, and lastly settles v_1. After settling v_1, the pruning criterion is true. Hence, further edges are not relaxed. This way, v_2 is never settled. Additionally, v_3 is never settled because it has no path to v_0 by only using downward edges.

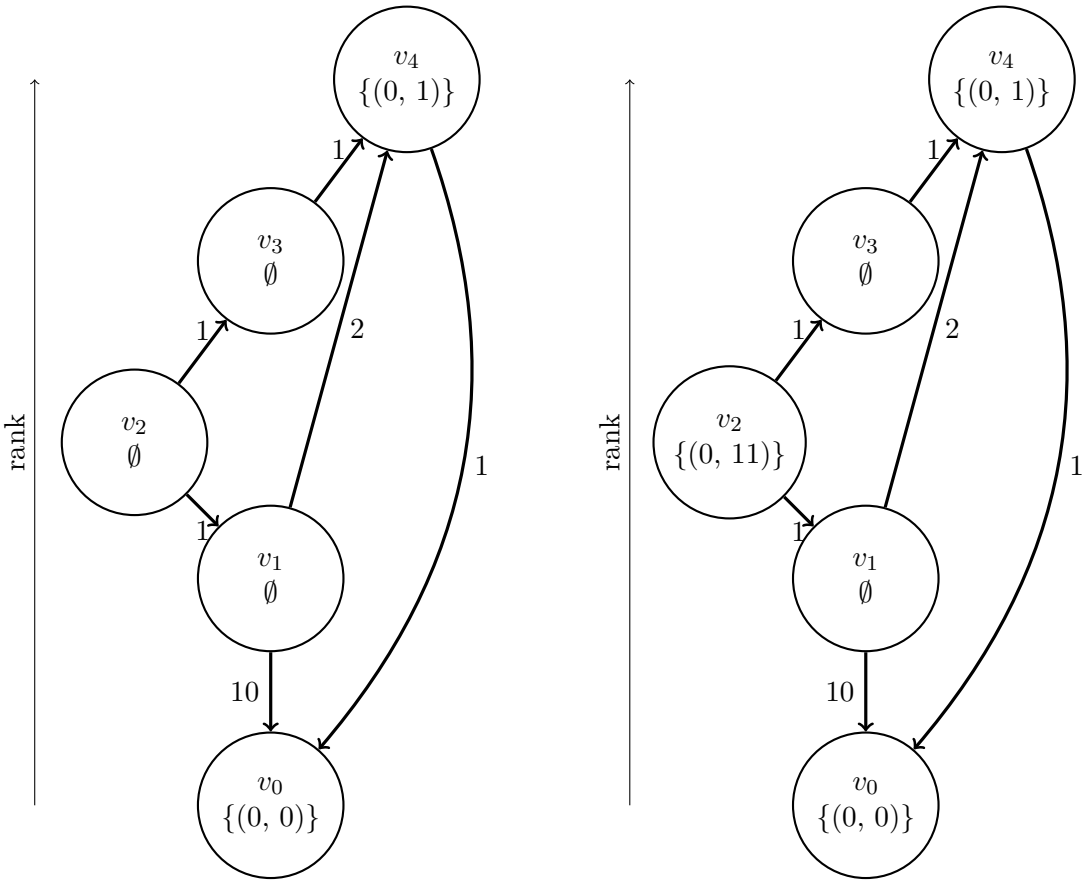
The simultaneous bucket initialization settles all vertices from v_0 to v_4 in increasing rank order. While settling the last vertex, v_4, only the buckets of v_1 and v_3 are available for pruning. This way, the associated buckets of v_1 and v_3, do not have an entry to the target. Unfortunately, v_2 keeps its unnecessary bucket entry.

Figure 3.3 demonstrates a case in which the simultaneous bucket initialization populates the buckets with fewer entries than the stall-on-demand technique.

Again, the stall-on-demand technique settles vertices in increasing order of minimum distance. Hence, vertices are settled in the following order: First v_0, then v_4 and v_2, and lastly v_1. Unfortunately, v_3 is never settled because the pruning criterion is true for v_2, thus the edge (v_2, v_3) is never relaxed. Because v_3 is never settled, the pruning criterion is false for v_1. This way the unnecessary bucket entry at v_1 is not pruned.

The simultaneous bucket initialization settles vertices from v_0 to v_4 in increasing rank order. Initially, a bucket entry $(v_0, 5)$ is added to v_1, but later pruned. Whilst settling v_3, a shorter path from v_1 to v_0 is discovered (v_1-v_3-v_2-v_0). Therefore, the algorithm removes the unnecessary entry from $B(v_1)$. Likewise, by settling v_4, respective bucket entries in v_2 and v_3 are pruned.

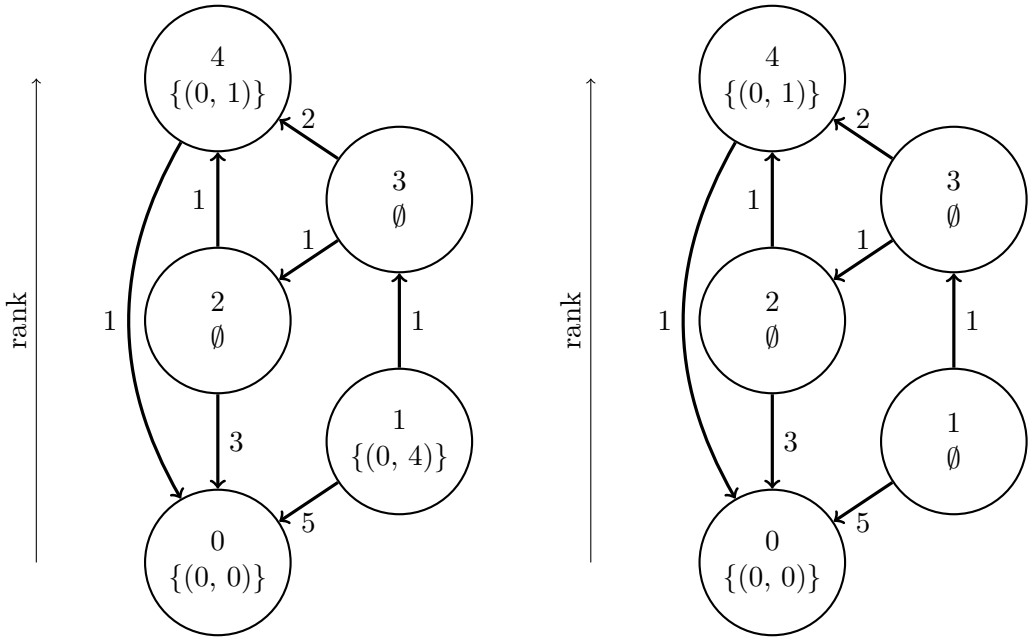
Based on our experimental evaluation in Section 4.5, the stall-on-demand technique is marginally better at removing unnecessary bucket entries. However, given that the simultaneous bucket initialization with the retrospective pruning is generally faster than the stall-on-demand technique, no clear winner can be picked. Nonetheless, both techniques are very effective in removing unnecessary bucket entries.



(a) Stall-on-Demand Initialization

(b) Simultaneous Bucket Initialization

Figure 3.2.: Buckets as populated by different algorithms.



(a) Stall-on-Demand Initialization

(b) Simultaneous Bucket Initialization

Figure 3.3.: Buckets as populated by different algorithms.

3.2.5. Simultaneous Bucket Initialization Many-to-Many

In this section we introduce a new algorithm that efficiently solves the many-to-many problem. The algorithm uses the simultaneous bucket initialization (SBI), as described in Section 3.2.3, to perform both the forward and backward search in G^+.

Initially, the algorithm follows the same concept as the bucket algorithm. Given our set of target vertices T, we use the SBI to populate buckets in $G^{\downarrow}$. We denote to the buckets initialized by the SBI in $G^{\downarrow}$ with $B_T(v)$. Hence, $B_T(v)$ contains entries $(t, dist(v,t))$, for multiple $t \in T$.

Following this *backward initialization*, we use the SBI again to populate buckets $B_S(v)$ in $G^{\uparrow}$. However, to use the SBI on $G^{\uparrow}$ we must slightly modify the algorithm. We use the SBI on $G^{\uparrow}$, by using *DownVertices*(v) to store edges $(w,v) \in E^{\uparrow}$, where `rank(v)` > `rank(w)`. The retrospective pruning must be modified analogously to consider edges in $E^{\downarrow}$ during the *forward initialization*. Lastly, we initialize the modified SBI with all sources S. Hence, $B_S(v)$ contains entries $(s, dist(s,v))$, for multiple $s \in S$. In this case, $dist(s,v)$ is equal to the shortest path from s to v in $G^{\uparrow}$. The distances stored in $B_T(v)$ are equal to the shortest path distance from v to t in $G^{\downarrow}$.

To calculate all shortest path distances, we must combine the information available in $B_S(v)$ and $B_T(v)$ for each $v \in V$. We initialize a set of vertices V', such that V' contains all vertices that we previously settled during the forward initialization as well as the backward initialization. Furthermore, V' must only contain vertices $v \in V'$, such that $B_S(v)$ is not empty and $B_T(v)$ is not empty.

Lastly, we iterate over all $v \in V'$ to calculate all shortest path distances. For each v, we iterate over all bucket entries $(s, dist(s,v)) \in B_S(v)$, each time iterating over all bucket entries $(t, dist(v,t)) \in B_T(v)$. Finally, we use $dist(s,v)$ and $dist(v,t)$ to update $D[s,t] = \min\{D[s,t],\ dist(s,v) + dist(v,t)\}$.

Once finished with all $v \in V'$, $D[s,t]$ is equal to the shortest path distance from s to t in G, for all $(s,t) \in S \times T$.

3.3. Hub Labels

3.3.1. Baseline Algorithm

Hub Labels (HL) is currently one of the fastest algorithm to calculate point-to-point shortest path distances. The HL algorithm was initially proposes by Abraham et al. [ADGW10] as a two stage algorithm. During the preprocessing, a *forward label* $L_f(v)$ and a *backward label* $L_b(v)$ is calculated for each vertex v. Each forward label $L_f(v)$ is made up of a set of vertices $u \in V$, with the associated shortest path distance $D(v,u)$ from v to u in G. Analogously, each backward label $L_b(v)$ contains a set of vertices $w \in V$, together with the associated shortest path distance $D(w,v)$ from w to v in G.

The labels must have the *cover property*, in order for the query algorithm to calculate correct shortest path distances. For every pair of vertices $(s,t) \in V \times V$, $L_f(s) \cap L_b(t)$ must contain a vertex h that is part of a shortest path from s to t. Hence, the shortest path distance from s to t is equal to $D(s,h) + D(h,t)$. Based on the cover property and an appropriate data representation, a point-to-point shortest path query can be answered very efficiently. As previously stated, the query algorithm must search for such a vertex h that minimizes $D(s,h) + D(h,t)$. By storing the labels in a consecutive memory slice sorted by their hub vertex, we can scan $L_f(s)$ and $L_b(t)$ simultaneously for h. We present the details of the query algorithm in Algorithm 3.2.

Algorithm 3.2: HUB LABELS QUERY

Input: forward label $L_f(s)$, backward label $L_b(t)$
Data: fwdIndex, bwdIndex and tentative shortest path distance μ
Output: Shortest distance from s to t

```
// Initialization
fwdIndex ← 0
bwdIndex ← 0
μ ← ∞
// Main loop
while fwdIndex < L_f(s).LEN() and bwdIndex < L_b(t).LEN() do
    (fwdHub, fwdDist) ← L_f(s)[fwdIndex]
    (bwdHub, bwdDist) ← L_b(t)[bwdIndex]
    if fwdHub == bwdHub then
        if fwdDist + bwdDist < μ then
            μ ← fwdDist + bwdDist
        fwdIndex ← fwdIndex + 1
        bwdIndex ← bwdIndex + 1
    else if fwdHub < bwdHub then
        fwdIndex ← fwdIndex + 1
    else
        bwdIndex ← bwdIndex + 1
return μ
```

3.3.2. Contraction Hierarchies

As shown by Abraham et al. [ADGW10], Hub Labels can be realized with Contraction Hierarchies in order to initialize the forward and backward labels. More precisely, we perform a CH forward search from each v in $G^{\uparrow}$ to initialize $L_f(v)$. While settling a vertex w, we add a pair $(w, dist(v, w))$ to $L_f(v)$. Backward labels are initialized analogously by a CH backward search in $G^{\downarrow}$.

We now focus on the correctness of the query algorithm. Again, we only explain the concept for the forward labels. As previously stated, the labels must have the cover property, in order for the query algorithm to calculate correct shortest path distances. However, because $G^{\uparrow}$ is created as a subgraph of G^{+} with only selected edges, a shortest path from v to w in $G^{\uparrow}$ might be longer than a shortest path from v to w in G^{+}. Nonetheless, following the correctness of a CH query, the cover property still holds true. As proven in Theorem 2.1 a shortest path from s to t in G^{+} exists that is an upward-downward path. Furthermore, let h be the highest ranked vertex, on such a shortest path from s to t. Following the correctness of a CH query, h must be settled by both the forward and backward search. Hence, h is added to both $L_f(s)$ and $L_b(t)$. Lastly, the distance from s to h $dist(s, h)$ in $G^{\uparrow}$ is equal to the shortest path distance from s to h in G^{+}. Consequentially, $L_f(s) \cap L_b(t)$ must contain a vertex h that ensures that the cover property is fulfilled.

3.3.3. Many-to-Many Hub Labels

In this section we focus on Hub Labels for the many-to-many shortest path problem. Instead of calculating the forward and backward labels during a separate preprocessing, we selectively calculate all labels while answering a many-to-many shortest path query. Given

that we only need to calculate shortest path distances from all $s \in S$ to $t \in T$, we must only initialize the labels for all source and target vertices.

To calculate all shortest path distances in $D[s, t]$ we perform the following steps: Initially, we initialize the forward labels for all $s \in S$ and the backward labels for all $t \in T$, by following the procedure as described in Section 3.3.2. However, because each label contains too many entries to allow efficient point-to-point shortest path query, we reduce the size of each label as stated in Section 3.3.4. Lastly, we perform $|S| \cdot |T|$ point-to-point shortest path queries to calculate all shortest path distances.

3.3.4. Label pruning

In this section we describe two pruning algorithms that reduce the size of each forward and backward label. Both algorithms were initially proposed by Abraham et al. [ADGW12]. First we describe an algorithm that removes most unnecessary label entries, followed by a stricter algorithm that removes all unnecessary entries.

The concept of both pruning algorithms is it to remove label entries with unnecessary distance values, that is distances that are not equal to the shortest path distance in G. As previously stated, the distance associated with each label entry is equal to the shortest path distance in either $G^{\uparrow}$ or $G^{\downarrow}$.

First, we apply the stall-on-demand technique, as previously used by the bucket-based algorithm in Section 3.2.2, to remove most unnecessary label entries. We refer to this approach as *HL partially pruned*. We briefly cover the forward case in which we initialize $L_f(v)$. For a more detailed explanation we refer to Section 3.2.2.

Let s be the vertex we want to calculate the forward label $L_f(s)$ for by using a CH forward search in $G^{\uparrow}$. While settling a vertex v, we check for the following condition: If for any incoming edge $(w, v) \in E^{\downarrow}$, $dist(s, v) > \omega(w, v) + d[w]$, we do not add v to $L_f(s)$. Furthermore, we do not have to relax any edges otherwise relaxed from v. Again, the backward labels can be initialized analogously.

We now focus on a stricter algorithm that removes all unnecessary label entries. We refer to this algorithm as *HL top down*. The *HL top down* algorithm identifies label entries with unnecessary distance values by calculating the shortest path distance in G for each entry. More precisely, all label entries $(v, dist(s, v)) \in L_f(s)$ whose associated distance $dist(s, v)$ is not equal to the shortest path distance $D(s, v)$ in G can be removed. To calculate the shortest path distances in G we use the HL query algorithm itself, as it is one of the fastest point-to-point shortest path algorithms. However, to use the HL query algorithm we must also initialize the forward and backward labels for all vertices present in the combined search space from any source or target vertex.

As before, we only describe the concept for the forward labels, backward labels can be initialized analogously. Given that we want to use the HL query algorithm to check if $dist(s, w)$ in $G^{\uparrow}$ is equal to $D(s, w)$ in G, for all $(w, dist(s, w)) \in L_f(s)$, we must also initialize the backward labels for all w. Hence, instead of only initializing the forward and backward labels exclusively for all $S \cup T$ we must initialize them for all vertices in the combined search space.

Initially, we perform $|S|$ separate forward searches to discover the combined forward search space from all sources. We refer to all vertices in this combined search space with V_S. Given V_S we initialize all labels by performing the following steps: Let v be the vertex with the highest rank in V_S, for which we have not initialized the forward and backward label. We use an iterative two-way merge to merge all $L_f(w)$ for all $(v, w) \in E^{\uparrow}$. However, by merging the labels we must increase the associated distance according to the weight of

the traversed edge $\omega(v, w)$. Lastly, we add a pair $(v, 0)$ to $L_f(v)$. Given that we initialize both forward and backward labels from top to bottom, we can use a HL query to calculate shortest path distance in G. We iterate over all label entries $(w, dist(v, w)) \in L_f(v)$: If $dist(v, w)$ is not equal to the distance as calculated by a *HL query* from v to w, we remove w from $L_f(v)$.

Once finished with all vertices in V_S, we have fully pruned forward labels.

3.4. PHAST

PHAST [DGNW10], short for parallel hardware-accelerated shortest path trees, is an extension of Contraction Hierarchies. PHAST is better suited for modern computer architectures as it accesses memory more efficiently by taking data locality into account.

PHAST calculates shortest paths from a single source to all vertices in the graph G. A PHAST query is split into two parts. First, a CH forward search starting from s in $G^{\uparrow}$ is performed. After being finished, the distance value $d[v]$ holds the shortest path distance from s to v in $G^{\uparrow}$, for each settled vertex v.

The second part of the query settles all vertices in descending rank order. For every settled vertex v all incoming edges $(u, v) \in E^{\downarrow}$ are examined. The algorithm updates $d[v]$ by assigning $d[v] = \min\{d[v],\ d[u] + \omega(u, v)\}$, for each edge $(u, v) \in E^{\downarrow}$.

Once all vertices in $G^{\downarrow}$ have been settled, $d[v]$ holds the correct shortest path distance from s to v in G.

Theorem 3.1. *PHAST computes correct shortest path distances $d[v]$, from s to v.*

Proof. As already proven in Theorem 2.1, in G^{+} there exists an upward-downward path P from s to t such that $\omega(P) = D(s, t)$. Let w be the vertex with maximal rank on such an upward-downward path P. Given that the first part of the query is the same as in a CH query, we know that $d[w]$ is equal to the shortest path distance from s to w in G. Furthermore, G^{+} was constructed to contain a shortest path from w to v by only traversing vertices in decreasing rank order. Given that the second part of a PHAST query settles vertices in decreasing rank order we must end up with correct shortest path distance $D(s, t)$, from s to v in $d[v]$. □

3.4.1. RPHAST

RPHAST [DGW11], short for restricted PHAST, is an extension of PHAST. Whereas PHAST calculates shortest paths from one source vertex to all vertices, RPHAST only calculates shortest paths from one source vertex to a selected set of target vertices T. As proposed by Delling et al. [DGW11], RPHAST adds a *target selection phase* to create a restricted graph $G_T^{\downarrow}$. $G_T^{\downarrow}$ contains exclusively the vertices and edges necessary to calculate shortest paths from any $s \in V$ to targets in T. Instead of using $G^{\downarrow}$ during the second part of a PHAST query, RPHAST uses $G_T^{\downarrow}$. Hence, RPHAST reduces the number of settled vertices and relaxed edges. The forward search, during the first part of an RPHAST query, still operates on the full $G^{\uparrow}$. Furthermore, storing $G_T^{\downarrow}$ in memory requires less space compared to $G^{\downarrow}$, thus brining data closer together and improving locality.

In order to calculate correct shortest paths, $G_T^{\downarrow}$ must contain all required vertices and edges, that is all vertices and edges visited by all CH backward searches from $t \in T$ in $G^{\downarrow}$ combined. The naive approach to calculate $G_T^{\downarrow}$ is to perform a CH backward search from each $t \in T$ in $G^{\downarrow}$ separately. During each CH backward search all settled vertices and relaxed edges can be extracted to create $G_T^{\downarrow}$.

RPHAST uses an alternative algorithm to efficiently construct $G_T^{\downarrow}$. Instead of performing an individual CH backward search for each $t \in T$, RPHAST performs a single backward search for all targets T at once. More precisely, RPHAST performs the following steps to calculate a set of necessary vertices T': T' is the set of vertices required to calculate shortest paths from any vertex to all $t \in T$ in $G^{\downarrow}$. Initially, T' and an unordered queue Q are initialized with all targets T. In each iteration an arbitrary vertex u is removed from Q and settled. If for any incoming edge $(w, u) \in E^{\downarrow}$, $w \notin T'$ is true, w is added to both T' and Q. The algorithm terminates as soon as the queue Q is empty. Lastly, $G_T^{\downarrow}$ is constructed as the subgraph of $G^{\downarrow}$, that only contains the vertices in T'. The set of required edges $E_T^{\downarrow}$ is equal to $E_T^{\downarrow} := \{(u, v) \mid (u, v) \in E^{\downarrow}, u \in T', \text{and } v \in T'\}$.

We further modify this approach to construct $G_T^{\downarrow}$ in order to increase the efficiency of a query. Although $G_T^{\downarrow}$ contains fewer vertices than $G^{\downarrow}$, the *first* array that we use to store $G_T^{\downarrow}$ has the same length as the *first* array for $G^{\downarrow}$. Because we access the *first* array with the vertex ID of each vertex, we must have an entry for every vertex ID. The *arclist* array already contains exclusively all edges that are present in $E^{\downarrow}$, thus we cannot reduce its size any further. To decrease the length of the *first* array for $G_T^{\downarrow}$, we assign restricted vertex IDs to each $v' \in T'$. We pick restricted vertex IDs consecutively from $(0, \ldots, |T'| - 1)$. Hence, by assigning restricted vertex IDs we reduce the length of the *first* array to $|T'| + 1$. To access the *first* array we use the restricted vertex ID of each vertex. Additionally, we replace vertex IDs in the *arclist* array with the restricted vertex IDs. We denote the restricted vertex ID of a vertex v with v'. In the case $v \notin T'$, v' is undefined.

We use a modified *depth-first-search* (DFS) to simultaneously assign restricted vertex IDs and create $G_T^{\downarrow}$. The modified DFS assigns restricted vertex IDs in reverse order of discovery. Initially, an empty stack $\mathcal{S}$ and an empty list T' are initialized. Furthermore, we process each target $t \in T$ as follows: If $t \notin T'$, we add t to both $\mathcal{S}$ and T'. As long as $\mathcal{S}$ is not empty, we examine $v \in \mathcal{S}$, such that v has been added to $\mathcal{S}$ the most recently. Otherwise, if $\mathcal{S}$ is empty we proceed with the next $t \in T$. If for any edge $(u, v) \in E^{\downarrow}$, $u \notin T'$, we add u to T' and $\mathcal{S}$. Otherwise, if no such u exists, we settle v. That is, we remove v from $\mathcal{S}$ and assign the next consecutive restricted vertex ID to v. Once finished with all $t \in T$, we have a set of required vertices T' together with restricted vertex IDs.

Moreover, we use the order as induced by the restricted vertex IDs to settle vertices during the second part of an RPHAST query. Our modified DFS only settles a vertex v if all u, with $(u, v) \in E^{\downarrow}$, have been settled before. Hence, by settling vertices in $G_T^{\downarrow}$ in the same order as we have settled them during the selection, we must end up with correct shortest path distances.

Lastly, we introduce a restricted distance array $d_T[v']$. The previous distance array $d[v]$ is the only array that is indexed by the non-restricted vertex ID during the second part of an RPHAST query. To avoid having to translate unnecessarily between non-restricted vertex IDs v and restricted vertex IDs v', we maintain a restricted distance array $d_T[v']$. As before, we access $d_T[v']$ with the restricted vertex ID. $d_T[v']$ only stores a distance value associated for each $v' \in T'$. Initially, $d_T[v'] = \infty$ for all $v' \in T'$. Furthermore, we initialize the distance values $d_T[v']$ during the first part of an RPHAST query. For each settled vertex v during a CH forward search from s in $G^{\uparrow}$, we check if $v \in T'$. If so, we copy the shortest path distance $dist(s, v)$, from s to v in $G^{\uparrow}$, to $d_T[v']$.

Following these adjustments, the second part of an RPHAST query operates exclusively on the restricted distance array $d_T[\cdot]$ and requires no translation between non-restricted and restricted vertex IDs. By settling vertices in the same order as they were settled by our modified DFS, we access $d_T[\cdot]$, the *first* array and the *arclist* array sequentially.

3.4.2. RPHAST Many-to-Many

We now focus on RPHAST for the many-to-many problem. To solve the many-to-many problem by using the RPHAST algorithm, we perform the following steps: Initially, we run the target selection phase once to calculate $G_T^{\downarrow}$. Given $G_T^{\downarrow}$, we perform multiple one-to-many queries to calculate all shortest path distances.

However, in order to exploit locality we can process multiple sources simultaneously. Instead of performing the first and second part of an RPHAST query separately for each $s \in S$, we perform the second part once for multiple sources simultaneously. Based on the observation that the second part settles all vertices in $G_T^{\downarrow}$ in the same order for any source, we can update the distance values for multiple sources simultaneously. As previously stated, the second part of an RPHAST query accesses all edges $(u, v) \in E_T^{\downarrow}$ to update $d[v]$. In case we calculate distances from multiple source, we can access (u, v) and $\omega(u, v)$ once to update multiple distance values.

We process a *batch* of k sources simultaneously. Hence, we pick k sources, $s_0, \ldots, s_{k-1}$, from S. We then calculate the shortest path distances from all s_i to all targets in T in one batch. In case the number of sources $|S|$ is greater than k, multiple batches can be processed to calculate all shortest path distances $D[s, t]$.

To simultaneously update the distances from k sources, we need to assign k distance values to each $v' \in T'$. We maintain k distance values, $d_{T,0}[v'], \ldots, d_{T,k-1}[v']$, for each $v' \in T'$. As initially discovered by the forward searches, $d_{T,i}[v']$ contains the shortest path distance from s_i to v in $G^{\uparrow}$. To store k distance values at each $v' \in T'$, we resize our restricted distance array $d_T[\cdot]$ to store $k \cdot |T'|$ values.

Given that all distance values $d_{T,0}[v'], \ldots, d_{T,k-1}[v']$ have been initialized appropriately, the second part proceeds as follows: As previously stated, we settle vertices in $G_T^{\downarrow}$ in the same order as they were settled by our modified DFS search. Let v' be the vertex we currently settle. Just like before, we examine all edges $(u', v') \in E_T^{\downarrow}$. However, this time we access each edge with its associated weight $\omega(u', v')$ once to update all k distance values. We update $d_{T,i}[v'] = \min\{d_{T,i}[v'],\ d_{T,i}[u'] + \omega(u', v')\}$ for all $i \in (0, \ldots, k-1)$.

Furthermore, SIMD (single instruction, multiple data) instructions can be used to execute these operations for multiple sources simultaneously.

We now discuss three methods to initialize all k distance values, $d_{T,0}[v'], \ldots, d_{T,k-1}[v']$, during the first part of an RPHAST query from multiple sources to all targets T. After one of the three methods has initialized the k distance values, the second part must be performed only once to calculate the shortest path distances.

k Separate Forward Searches

We perform the first part of an RPHAST query k times. During each iteration, we start with one of the k batched sources s_i. While settling a vertex v, we check if $v \in T'$. If so, we copy the shortest path distance $dist(s_i, v)$ to the respective entry in the restricted distance array.

Modified Simultaneous Bucket Initialization

We use the simultaneous bucket initialization, as proposed in Section 3.2.3, to initialize all k distance values for each $v' \in T'$. However, to apply the simultaneous bucket initialization we have to slightly modify the algorithm.

As initially planned, the simultaneous bucket initialization was used by the bucket-based algorithm to calculate shortest path distances in $G^{\downarrow}$. We now modify the simultaneous

bucket initialization to calculate shortest path distance in $G^{\uparrow}$. Moreover, we initialize the restricted distance values, $d_{T,0}[v'], \ldots, d_{T,k-1}[v']$ with the bucket entries in $B(v)$, if $v \in T'$. Instead of using *DownVertices*(v) to access all outgoing edges $(v,u) \in E^{\downarrow}$, where `rank(`v`)` $>$ `rank(`u`)`, we use *DownVertices*(v) to access appropriate edges in $E^{\uparrow}$. That is, *DownVertices*(v) stores all forward edges $(u,v) \in E^{\uparrow}$, where `rank(`u`)` $<$ `rank(`v`)`. Besides this modification, the algorithm follows the same concept as before. Again, while settling a vertex v we copy the current bucket entries to the restricted distance array if $v \in T'$. To efficiently copy the current bucket entries $B(v)$, we access the tentative distance array $d[s_i]$ directly. However, because the distance array $d[\cdot]$, as used by the SBI algorithm, only contains the distances to all targets that were discovered from any adjacent bucket we also have to check for the following condition: If the currently settled vertex v is equal to one of the k batched sources s_i, we must set the corresponding restricted distance value $d_{T,i}[v']$ to zero. By setting the appropriate distance value to zero, we indicate that the distance from s_i to v is equal to zero if v and s_i refer to the same vertex.

Simultaneous Rank Initialization

Lastly, we propose an algorithm to calculate shortest path distances starting from multiple sources simultaneously. This approach follows the same concept as the second part of an RPHAST query, but this time settling vertices in $G^{\uparrow}$. We settle vertices in $G^{\uparrow}$ in increasing rank order, each time updating the distance values for multiple sources.

Instead of using a bucket data structure as the simultaneous bucket initialization does, we maintain k distance values for every $v \in V$. Because we have no indication to the size of the combined forward search space from all k sources we must assign k distance values to each $v \in V$. We refer to each distance value with $d_0[v], \ldots, d_{k-1}[v]$. Storing k distance values for each $v \in V$ can be done in a one-dimensional array of length kn. However, for large n and k this might be challenging as it requires a lot of memory. In cases where not enough memory is available or practical we dynamically assign k distance values to each discovered vertex. To this end, we maintain a resizable array for each $v \in V$. Initially every list is empty and only resized to store k distance values as soon as we settle v or relax an edge leading to v. Hence, we only need to store the distance values for every settled vertex. Additionally, we can reset the lists of a settled vertex as soon as we have copied its distance values to the restricted distance array $d_T[\cdot]$.

Given an appropriate distance value representation, we initialize the distance values by performing the following steps: To settle vertices in increasing rank order we use a priority queue Q ordered by minimum rank. Initially, we populate Q with all s_i. In each iteration a vertex v with minimum rank is removed from Q and settled. We iterate over all outgoing edge $(v,u) \in E^{\uparrow}$ and add u to Q in case it has not been added before. Moreover, we update $d_i[u] = \min\{d_i[u], d_i[v] + \omega(v,u)\}$ for all $i \in (0, \ldots, k-1)$. Additionally, we check if $v \in T'$. If so, we copy $d_0[v]$, …, $d_{k-1}[v]$ to $d_{T,0}[v']$, …, $d_{T,k-1}[v']$.

3.5. Lazy RPHAST

3.5.1. Baseline Algorithm

Lazy RPHAST is an extension of RPHAST that computes shortest path distances from multiple sources to one target. The algorithm is based on the CH-Potentials heuristic, as proposed by Strasser and Zeitz [SZ21].

Instead of selecting multiple target, during the target selection phase, Lazy RPHAST selects a single target vertex. The core idea behind the Lazy RPHAST algorithm is to *lazily* calculate the forward distances from multiple sources. That is, during a forward

search from any s all calculated shortest path distances are *memoized*. As long as the target does not change, every additional shortest path query can reuse these previously computed shortest path distances. Especially when calculating shortest path distances from multiple sources that share the same forward search space, we can reuse memoized shortest path distances.

The Lazy RPHAST algorithm maintains two distance arrays to calculate and memoize shortest path distances, $B[v]$ and $F[v]$. Initially all values in $B[v]$ are set to ∞. Furthermore, the values in $F[v]$ are set to $\perp$, indicating that we have yet to calculate the shortest path distance from v to the selected target.

During the target selection phase, a single target $t \in V$ is selected. Given such a target t the distance values in $B[v]$ are initialized by a CH backward search in $G^{\downarrow}$ from t. Following this target selection phase, $B[v]$ is equal to the shortest path distance from v to t in $G^{\downarrow}$.

After the target selection phase, multiple shortest path queries, starting from any vertex $s \in V$ to the selected target, can be efficiently answered. Given such a source vertex s the algorithm uses the *calcAndMemoize* algorithm (Algorithm 3.3), to recursively calculate the shortest path distance.

The function *calcAndMemoize* performs the following steps to calculate and memoize the shortest path distance from a given vertex s to the previously selected target t: First, we check if $F[s] \neq \perp$. If so, the shortest path distance from s to t has been calculated before. Hence, the function returns $F[s]$, as $F[s]$ is equal to the shortest path distance from s to t. Otherwise, $F[s]$ is assigned $B[s]$. Additionally, we iterate over all edges $(s, v) \in E^{\uparrow}$ and update $F[s]$ by assigning $F[s] = \min\{F[s],\ \omega(s, v) + F[v]\}$. Before accessing $F[v]$, we recursively call the *calcAndMemoize* function on v, so that $F[v]$ already contains the shortest path distance from v to t.

Algorithm 3.3: ComputeAndMemoize

Input: $B[v]$: shortest path distance from v to t as initialized by the target selection phase
Input: $F[v]$: memoized distance from v to t, initially set to $\perp$
Input: s: the source vertex on a path from s to t

```
// Main loop
1 if F[s] == ⊥ then
2     F[s] ← B[s]
3     forall (s, v) ∈ E↑ do
4         F[s] ← min{F[s], ω(u, v) + ComputeAndMemoize(v)}
5 return F[s]
```

3.5.2. Batched Lazy RPHAST

In this section we propose an extension of Lazy RPHAST that is better suited to solve the many-to-many problem. We refer to this algorithm as *Batched Lazy RPHAST*. The concept behind the Batched Lazy RPHAST algorithm is the same as in the RPHAST algorithm for multiple sources, as described in Section 3.4.2. As before, we exploit locality by processing a batch of k vertices simultaneously. Because the Lazy RPHAST algorithm efficiently calculates many-to-one distances, we batch multiple targets instead of sources. We denote the k targets, that are part of the current batch, with $t_0, \ldots, t_{k-1}$.

As previously stated in Section 3.5.1, Lazy RPHAST uses the target selection phase to initialize a distance array $B[v]$ for a single target vertex t. To simultaneously process a

batch of k targets, we must initialize the distances to all k batched targets t_i in $G^{\downarrow}$. To efficiently initialize the distances to all t_i, we utilize the simultaneous bucket initialization as proposed in Section 3.2.3. Hence, by using the simultaneous bucket initialization we store the shortest path distance from $v \in V$ to t_i in a bucket $B(v)$.

Lastly, we modify the *computeAndMemoize* algorithm to calculate shortest path distances from one source to k targets. As before, $F[v]$ is initialized with $\perp$ for all $v \in V$. However, instead of storing a single distance value in $F[v]$, we store k distance values in $F[v]$. We denote to the individual shortest path distance from v to t_i with $F_i[v]$. Let s be the source vertex to calculate the distance from to all batched targets t_i. First, we check if $F[s] \neq \perp$. If so, the distances from s to all targets t_i were already calculated and memoized. Hence, the function returns all k distance values $F_i[s]$. Otherwise, we initialize $F[s]$ with k individual distance values. Initially, $F_i[s] = \infty$ for all $i \in (0, \ldots, k-1)$. Furthermore, we iterate over all entries $(t_i, dist(s, t_i)) \in B(s)$ and set $F_i[s]$ to $dist(s, t_i)$. Lastly, we access each edge $(s, v) \in E^{\uparrow}$ with its associated weight $\omega(s, v)$ once to update all $F_i[s]$ by assigning $F_i[s] = \min\{F_i[s],\ \omega(s, v) + F_i[v]\}$. Before accessing $F_i[v]$ for any i, we recursively call the *calcAndMemoize* function once on v, so that all required distance values are calculated.

Similar to SIMD RPHAST, we introduce a variant of Batched Lazy RPHAST that uses SIMD instructions to process multiple targets simultaneously.

4. Experiments

In this chapter, we evaluate all implemented algorithms. Moreover, we compare the algorithms with each other in different scenarios.

4.1. Environment

We implemented all algorithms in Rust[1] and compiled them with cargo 1.60.0-nightly using the release profile with opt-level=3. All experiments were performed on a Supermicro Superserver SYS-5018R-MR that runs openSUSE Leap[2] with version 15.3. Moreover, our benchmark machine has 128 GB (8x16 GB) DDR4 2133 MHz ECC RAM and an Intel Xeon E5-1630 v3 CPU with four cores clocked at 3.70 GHz. All experiments were executed sequentially.

Our test graph is the European road network as it was made available by PTV[3] for the 9th DIMACS Implementation Challenge [DGJ09]. The graph has about 18 million vertices and 42 million directed edges prior to the CH preprocessing. Additionally, we used RoutingKit[4] to create the augmented graph G^+.

4.2. Algorithms

We use this section to give an overview of all algorithms considered during our experiments. As described in Section 3, we assume that a CH preprocessing was performed in advance. Hence, all running times only include the time needed to calculate all shortest path distances, excluding the time required to calculate G^+.

- *Dijkstra*: Dijkstra's algorithm as described in Section 2.2. Moreover, we stop the query as soon as all target vertices T have been settled.
- *BCH*: The standard bucket-based CH algorithm as described in Section 3.2.1. Additionally, the *BCH* algorithm uses the stall-on-demand technique, as described in Section 3.2.2, to prune unnecessary bucket entries.

[1] https://www.rust-lang.org
[2] https://get.opensuse.org/leap/
[3] https://ptvgroup.com
[4] https://github.com/RoutingKit/RoutingKit

- *SBI*: The simultaneous bucket initialization algorithm as described in Section 3.2.3. The *SBI* algorithm uses the same query algorithm as the *BCH* algorithm, to perform the one-to-many queries after all buckets have been populated. Furthermore, the *SBI* algorithm applies the retrospective pruning as described in Section 3.2.3.
- *SBI Many-to-Many*: The SBI algorithm for the many-to-many problem as described in Section 3.2.5.
- *RPHAST*: The RPHAST algorithm, as described in Section 3.4.1. To compute all shortest path distances, *RPHAST* performs $|S|$ separate one-to-many queries.
  - *RPHAST batched dijkstra*, *RPHAST batched SBI*, and *RPHAST batched rank*: All three algorithms process a batch of k sources during the second part of an RPHAST query as described in Section 3.4.2. Only the first part of a batched RPHAST query varies between all three algorithms.
  - *RPHAST batched dijkstra*: Runs Dijkstra's algorithm k times as described in Section 3.4.2 to initialize the distances for k sources.
  - *RPHAST batched SBI*: Uses the SBI as described in Section 3.4.2 to initialize the distances for k sources.
  - *RPHAST batched rank*: Uses the rank initialization as described in Section 3.4.2 to initialize the distances for k sources.
  - *SIMD RPHAST*: Uses SIMD instructions to process a batch of $k = 16$ sources simultaneously. *SIMD RPHAST* uses the rank initialization as described in Section 3.4.2 to initialize the distances for k sources by also using SIMD instructions.
- *Lazy RPHAST*: The *Lazy RPHAST* algorithm as described in Section 3.5.1.
- *Batched Lazy RPHAST*: The batched variant of *Lazy RPHAST*, as described in Section 3.5.2.
- *SIMD Lazy RPHAST*: The same as *Batched Lazy RPHAST* but uses SIMD instructions to process $k = 16$ targets simultaneously.
- *HL partially pruned*: The Hub Label based algorithm for the many-to-many problem, as described in Section 3.3.3. *HL partially pruned* initializes the forward and backward labels only for the source and target vertices as describe in Section 3.3.4.
- *HL top down*: The Hub Label based algorithm for the many-to-many problem, as described in Section 3.3.3. *HL top down* initializes the forward and backward labels of all vertices in the combined search space by using the stricter pruning algorithm as described in Section 3.3.4.

4.3. One-to-Many

To model different scenarios in which one-to-many queries must be answered, we pick our source and target vertices from different subsets of V. In some cases, it might be interesting to calculate shortest path distances from one source location to a set of target locations that are in proximity to each other. For example, the distances to all restaurants or ATMs within a given city. However, in other cases, it might be interesting to calculate shortest path distances to multiple targets that are spread over a large area. For example, the distances from one source location to all city centers of all major cities in a given range.

We pick our vertices from a ball $\mathcal{B}$ of varying size. We use Dijkstra's algorithm to obtain $\mathcal{B} \subseteq V$. Initially, we pick a center vertex c uniformly at random from V and use it as the

source location for Dijkstra's algorithm. We let Dijkstra's algorithm settle vertices until the number of settled vertices reaches a predetermined amount, that is $|\mathcal{B}|$ the size of our ball. We then pick a set of distinct target vertices uniformly at random from $\mathcal{B}$. Lastly, we pick a separate source vertex either uniformly at random from $\mathcal{B}$ or V to simulate different scenarios.

4.3.1. Same Ball

We give an overview of the running times of all one-to-many algorithms in Figure 4.3. Additionally, we present the selection and query times separately in Figure 4.1 and in Figure 4.2. Each algorithm is tasked with calculating all shortest path distances from a single source vertex to a given set of target vertices T. We pick the source vertex and all target vertices from the same ball $\mathcal{B}$. Furthermore, we use different sizes for the set of target vertices. We present the running times for $|T| = 2^{14}$ in this section. Results with different target set sizes can be found in Appendix D, E and F i.e. $|T| = 2^{10} = 1024$, $|T| = 2^{12} = 4096$, and $|T| = 2^{14} = 16384$. Lastly, all experiments vary the size of the ball $|\mathcal{B}|$, from which targets are picked. Initially, $|\mathcal{B}| = |T|$, hence all targets are in proximity to each other. However, we gradually increase the size of the ball $|\mathcal{B}|$ from $|T|$ up to 2^{24}, to model scenarios where targets are spread over a large area. We perform 100 iterations per ball size $|\mathcal{B}|$ and algorithm. Each point is the average of 100 iterations. Moreover, each iteration uses a different source vertex as well as a different set of target vertices T. Selection times include the time needed to calculate $G_T^{\downarrow}$, populate the buckets or calculate the forward and backward labels. Query times include the time needed to perform the subsequent query phase. Lastly, we do not break down the running times of Dijkstra's algorithm and *Lazy RPHAST* as they have no comparable selection phase. Contrary to what we described in Section 3.5.1, we use a mirrored version of *Lazy RPHAST* that is better suited to solve the one-to-many problem.

Increasing the size of the target set $|T|$, from 2^{10} to 2^{12} and then to 2^{14}, has no effect on the relative order among all algorithms.

The running time of Dijkstra's algorithm increases steadily with the size of the ball $|\mathcal{B}|$. As *Dijkstra* is not goal-directed, the algorithm settles all vertices around the source vertex in all directions. Hence, by spreading the targets further apart, *Dijkstra* must settle more vertices in total which results in an increased running time. By doubling the size of the target set $|T|$, the running time of *Dijkstra* increases by roughly the same factor.

In the following we take a closer look on the differences between the two bucket-based algorithms. Both *BCH* and *SBI* use the same query algorithm, hence the query times of both algorithms are roughly the same. The only difference between both query algorithms is a minor disparity in the number of bucket entries per bucket. As stated in Section 3.2.4, in some scenarios one pruning algorithm produces fewer bucket entries. We refer to Section 4.5 for an experimental analysis of the average bucket sizes of both algorithms.

The bucket-based query algorithm requires almost the same time for all ball sizes $|\mathcal{B}|$, since the CH forward search settles the same amount of vertices in total no matter the size of the ball $|\mathcal{B}|$. However, as the targets are spread over a larger area, the average size of each populated bucket decreases, thus resulting in a decreased time required to scan each bucket. This phenomenon cannot be observed in our experiment where the size of the target set is fixed to 2^{10}, as the number of entries per bucket is always on a low level. Furthermore, increasing the size of the target set from 2^{10} to 2^{14} increases the average query time from 0.159 ms to 1.68 ms (only an increase of roughly 11x as opposed to 16x for the size of the target set). In comparison to the *RPHAST* query, both the *BCH* query and the *SBI* query eventually become faster as the size of the ball $|\mathcal{B}|$ increases. For smaller target set sizes $|T|$, both the *BCH* query and the *SBI* query are almost always faster than

the *RPHAST* query. However, even for $|T| = 2^{14}$ a bucket query is about 3.65 times faster than an RPHAST query when $|\mathcal{B}| = 2^{24}$ (1.14 ms for a bucket query compared to 4.16 ms for an RPHAST query).

We now focus on the selection phase of both bucket-based algorithms. The *BCH* selection algorithm performs a separate CH backward search for each target. Hence, as evident in our one-to-many experiments, the time needed to populate the buckets is not influenced by the size of the ball $|\mathcal{B}|$. Nonetheless, for larger ball sizes $|\mathcal{B}|$ the *BCH* selection time increases slightly due to worse locality. Furthermore, the *BCH* selection time increases by roughly the same factor from 21 ms to 315 ms (15x) when increasing the size of the target set from 2^{10} to 2^{14}. Contrary to the *BCH* selection algorithm, the *SBI* selection algorithm requires more time as the size of the ball $|\mathcal{B}|$ increases. Initially, when targets are in proximity to each other, the *SBI* selection algorithm is able to efficiently populate the buckets. By increasing the size of the ball $|\mathcal{B}|$ we increase the number of total unique vertices settled during the target selection. Hence, the advantage gained by visiting every vertex only once during the selection is not able to compensate for the overhead needed to merge bucket entries as targets are spread over a larger area. Moreover, without taking the early stopping criteria of the stall-on-demand technique into account, both query algorithms settle the same vertices. In extreme cases, where only a few targets are spread over a large area, the *BCH* selection is eventually faster than the *SBI* selection. We can observe this by selecting 2^{10} targets from a ball $\mathcal{B}$ that has at least 2^{22} vertices to pick from. However, for $|T| = 2^{14}$ the *SBI* algorithm is always the fastest algorithm to populate the buckets. Even for $|B| = 2^{24}$ the *SBI* selection is about 1.22 times faster than the *BCH* selection (360 ms for the *BCH* selection compared to 295 ms for the *SBI* selection).

RPHAST is about 44 times faster than Dijkstra's algorithm when comparing the total time required to calculate all one-to-many shortest path distances. Both the time needed to calculate $G_T^{\downarrow}$ and the query time depend on the size of the ball $|\mathcal{B}|$. By selecting targets from a larger ball $\mathcal{B}$, the size of the restricted graph $G_T^{\downarrow}$ increases significantly. Hence, the time needed to create $G_T^{\downarrow}$ increases. Consequentially, the query time increases as well, as more vertices must be settled. By increasing the size of the ball $|\mathcal{B}|$ from 2^{14} to 2^{24} while selecting 2^{14} targets, the number of vertices in $G_T^{\downarrow}$ increases from 18150 to 306012 on average (16.86x). As observed by Delling et al. in [DGW11], query times increase slightly less from 0.27 ms to 4.2 ms (15.03x) due to the fact that the single CH forward search in $G^{\uparrow}$ does not depend on the size of the ball $|\mathcal{B}|$. Furthermore, the selection time increases more for larger ball sizes $|\mathcal{B}|$ (from 1.30 ms to 50.92 ms when increasing the ball size $|\mathcal{B}|$ from 2^{14} to 2^{24}).

Lazy RPHAST is the fastest algorithm to calculate all one-to-many distances, when adding up selection and query times. *RPHAST* performs slightly worse due to its much slower target selection phase during which $G_T^{\downarrow}$ must be calculated during each one-to-many query. If we were to only consider the query times, *RPHAST* would be the fastest algorithm.

Lastly, we focus on the two Hub Label based algorithms. Both query algorithms perform equally well as they mainly perform the same work. Although the *HL top down* algorithm produces labels with fewer entries, its query performs slightly worse due to worse locality. We store the labels for *HL partially pruned* consecutively in memory for all source and target vertices, whereas we store them directly at each vertex $v \in V$ for *HL top down*. When considering selection times, *HL partially pruned* is much faster than *HL top down* as the algorithm only calculates the labels for the source vertex as well as all target vertices. Hence, the *HL partially pruned* selection does not depend on the size of the ball $|\mathcal{B}|$. On the other hand, the *HL top down* selection time correlates with the size of the ball $|\mathcal{B}|$, as more labels must be calculated when the targets are spread further apart from each other.

Based on these observations, *HL partially pruned* is the faster of the two algorithms, as query times are almost identical and its selection is much faster.

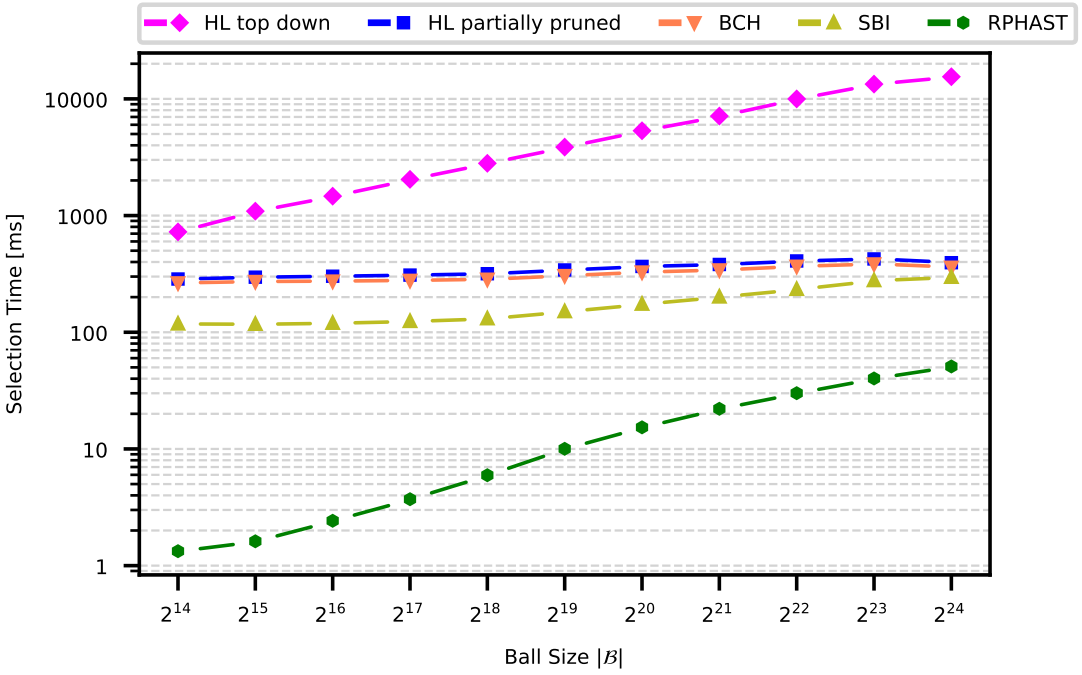


Figure 4.1.: Selection times of one-to-many algorithms with $|T| = 2^{14}$ targets picked from a ball of varying size $|\mathcal{B}|$; Source picked from the same ball.

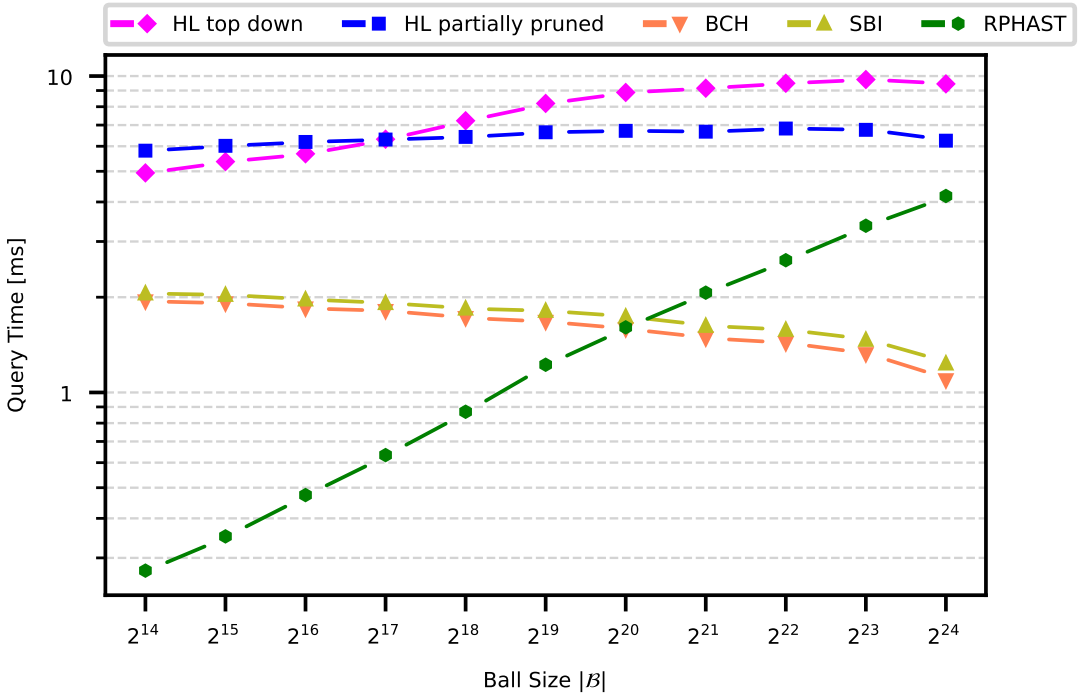


Figure 4.2.: Query times of one-to-many algorithms with $|T| = 2^{14}$ targets picked a ball of varying size $|\mathcal{B}|$; Source picked from the same ball.

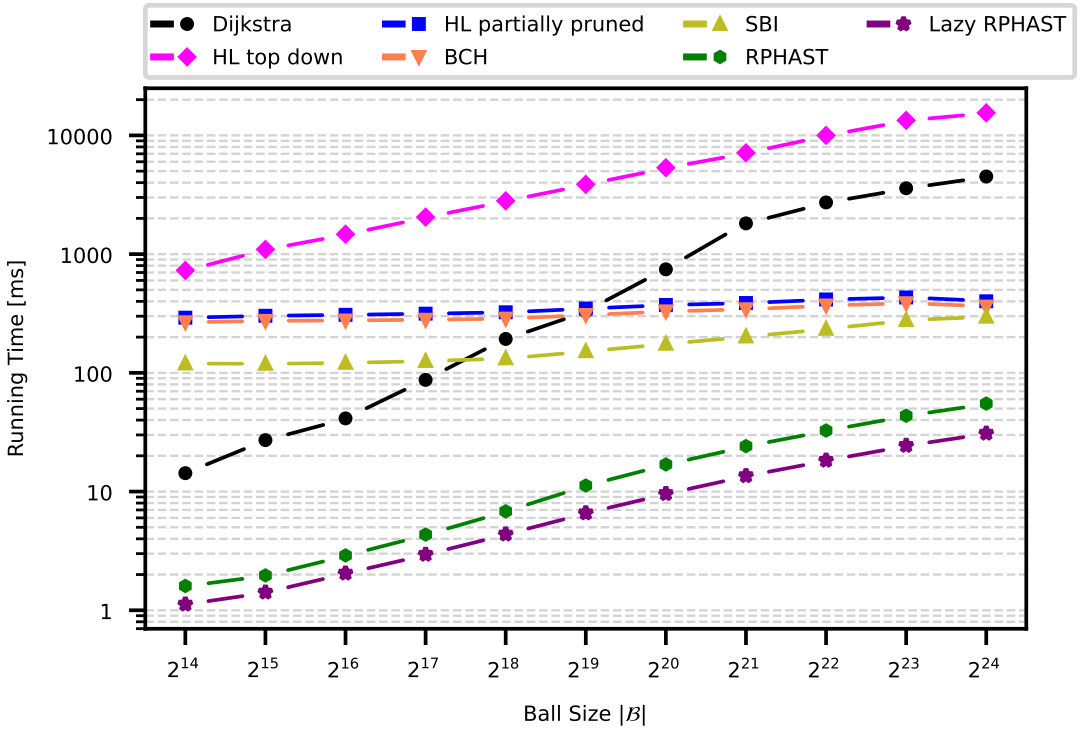


Figure 4.3.: Running times of one-to-many algorithms with $|T| = 2^{14}$ targets picked from a ball of varying size $|\mathcal{B}|$; Source picked from the same ball.

4.3.2. Different Balls

In the following, we are going to cover a slightly modified one-to-many experiment than the previous one-to-many experiment in Section 4.3.1. This time we pick our source vertex from V instead of $\mathcal{B}$. We still pick all target vertices T from a ball $\mathcal{B}$ of varying size. We refer to Appendix A, B, and C for the results. Although most algorithms perform the same under these modified conditions, there are two notably exceptions.

First, Dijkstra's algorithm performs worse. By picking the source vertex from V, Dijkstra's algorithm must settle more vertices on average until all targets T are settled. Hence, the running time increases. Furthermore, the running time is no longer dependent on the size of the ball $|\mathcal{B}|$, because the average distance from the source to the most distant $t \in T$ remains roughly the same for all ball sizes $|\mathcal{B}|$. We can only observe a small uptick in the running time by increasing the size of the ball $|\mathcal{B}|$.

Secondly, the bucket-based query algorithm (the *BCH* query and the *SBI* query) requires less time as we pick the source from V, compared to when we pick the source from the same ball $\mathcal{B}$ as the targets. As we move the source further away from all targets, we reduce the number of vertices settled during the forward search that were also settled during the selection phase. Hence, as we only examine those bucket entries of vertices visited by both searches, a query scans less bucket entries in total in this case. However, as we increase the size of the ball $|\mathcal{B}|$ the query time increases because the number of settled vertices that have a non-empty bucket increases as well. Initially, for $|B| = |T| = 2^{14}$ a bucket query is about 2.27 times faster when the source is picked from V compared to when the source is picked from $\mathcal{B}$ (0.88 ms compared to 2 ms). However, for $|B| = 2^{24}$ and $|T| = 2^{14}$ picking the source from V or $\mathcal{B}$ makes almost no difference since the size of the ball is almost equal to $|V|$. Hence, $\mathcal{B}$ contains almost all vertices present in V.

4.4. Many-to-Many

Our many-to-many experiments follow the same approach as our one-to-many experiments in Section 4.3. As before, we use Dijkstra's algorithm to obtain $\mathcal{B} \subseteq V$. We pick our set of source and target vertices from either a shared ball or two different balls, to model different scenarios. Additionally, we conduct experiments in which $S = T$, which is useful for problems related to the vehicle routing problem. In cases where we pick our source and target vertices from different balls we always use the same ball size $|\mathcal{B}|$ for both balls. Moreover, in such cases where we pick source and target vertices from separate balls, both balls may overlap. Hence, $|S \cap T| \geq 0$. Especially for large ball sizes $|\mathcal{B}|$ both balls must overlap, as there are roughly 18 million vertices in V. Besides symmetric cases in which $|S| = |T|$, we also examine asymmetric cases in which $|S| > |T|$ or $|S| < |T|$. All running times include the time needed to select the targets T, i.e. construct $G_T^{\downarrow}$, populate all buckets or calculate all labels as well as the time needed to calculate all shortest path distances. During each experiment, we perform 100 iterations per algorithm and ball size $|\mathcal{B}|$. Each iteration uses a different set of source and target vertices.

4.4.1. Symmetric Case

In the following, we conduct the same symmetric experiment with three different configurations. We pick $|S| = |T| = 2^{10}$, $|S| = |T| = 2^{12}$, and $|S| = |T| = 2^{14}$ source and target vertices from two different balls of the same size $|\mathcal{B}|$.

RPHAST

Initially, we take a detailed look on the various RPHAST variants. We refer to Figure 4.4 for a simplified overview that only contains the running times of all RPHAST variants, when we pick $|S| = |T| = 2^{14}$ source and target vertices. The relative order between all variants remains unchanged for the different symmetric configurations we considered. We present the results with different configurations, i.e. $|S| = |T| = 2^{10}$ and $|S| = |T| = 2^{12}$, in Appendix G. We use a logarithmic scale along the vertical axis to highlight the disparity between *RPHAST batched dijkstra* and both *RPHAST batched SBI* and *RPHAST batched rank*.

The running times of all RPHAST variants are dependent on the size of the ball $|\mathcal{B}|$. However, increasing the size of the ball $|\mathcal{B}|$ affects *RPHAST* the worst. The average *RPHAST* running time increases from 4.4 s to 66.72 s (15.16x), when the size of the ball $|\mathcal{B}|$ is increased from 2^{14} to 2^{24}. As previously observed during the one-to-many experiments in Section 4.3, the number of vertices in $G_T^{\downarrow}$ increases by roughly the same factor. Furthermore, by batching $k = 16$ sources during the second part of an RPHAST query, we are able to reduce the relative increase down to factor of roughly 11.02 times. Using SIMD instructions to batch $k = 16$ sources improves the running time further (*SIMD RPHAST* is about 5 times faster than *RPHAST*).

We now take a closer look on *RPHAST batched dijkstra*, *RPHAST batched SBI*, and *RPHAST batched rank* as they are mostly the same. The only difference between them is how they initialize the restricted distance array for k source vertices. Because the query time is mainly dominated by the more expensive second part, query times are roughly the same for all three variants. However, *RPHAST batched dijkstra* performs the worst out of the three variants. Especially when sources are in proximity to each other, *RPHAST batched dijkstra* settles the same vertices several times to initialize all distance values. Both *RPHAST batched SBI* and *RPHAST batched rank* are able to improve the first part of an RPHAST query by better utilizing locality. Nonetheless, with an increased ball size $|\mathcal{B}|$, the running times of all three variants converge on each other. Although *RPHAST batched SBI*

and *RPHAST batched rank* require almost the same time, *RPHAST batched SBI* requires far less memory. Especially in cases where the size of the ball $|\mathcal{B}|$ is comparatively large, *RPHAST batched SBI* requires far less memory than *RPHAST batched rank*.

Given that *RPHAST fwd dijkstra* is always dominated by *RPHAST fwd rank* as well as *RPHAST fwd SBI*, we are going to discard *RPHAST fwd dijkstra* hereafter.

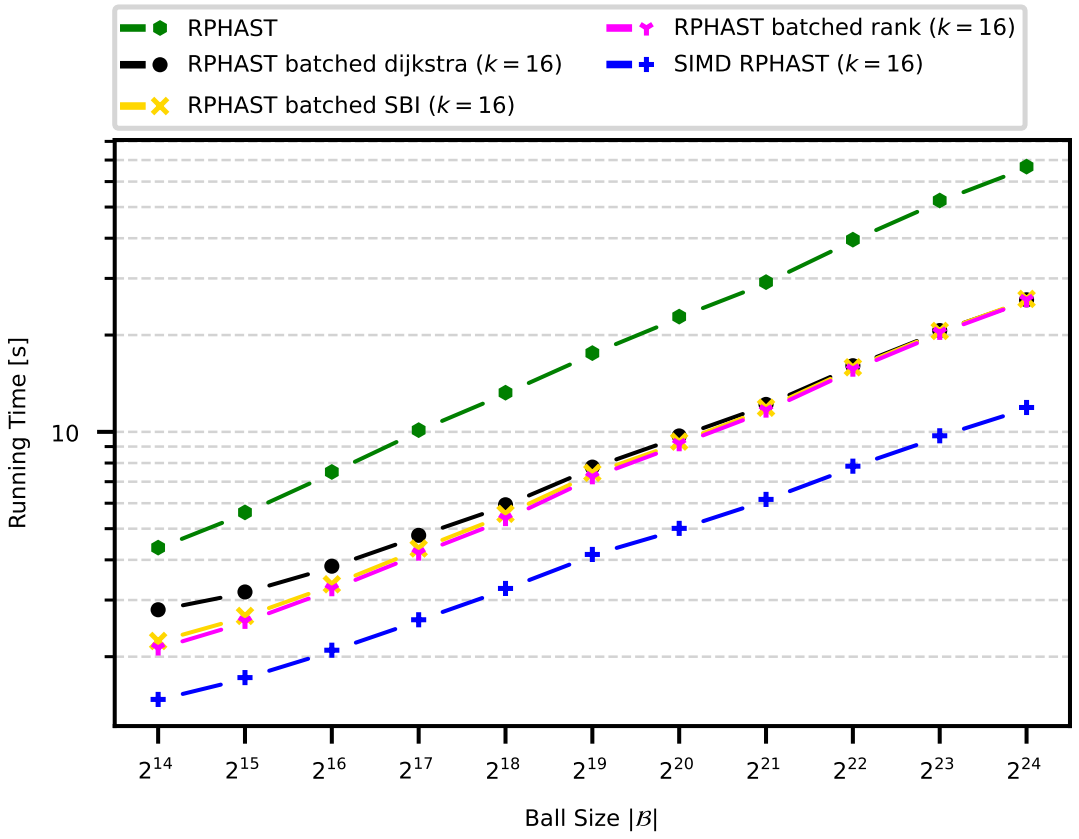


Figure 4.4.: Running times of RPHAST algorithms with $|S| = |T| = 2^{14}$ sources and targets picked from different balls of varying size $|\mathcal{B}|$.

Hub Labels

Next we are going to focus on both Hub Label based algorithms for the many-to-many problem. We refer to Figure 4.5 for a comparison between *HL partially pruned* and *HL top down*. Additionally, we refer to Appendix H for the results of experiments with different configurations.

While picking less source and target vertices, i.e. $|S| = |T| = 2^{10}$, *HL partially pruned* performs much better than *HL top down*. In this case, running times are dominated by the time needed to calculate the forward and backward labels. Hence, *HL partially pruned* performs much better as it has to calculate fewer labels. Especially by increasing the size of the ball $|\mathcal{B}|$, *HL top down* has to calculate more labels, thus leading to an increased selection time. By increasing the size of the source and target set from $|S| = |T| = 2^{10}$ to $|S| = |T| = 2^{14}$, running times are no longer dominated by the time needed to calculate the labels. Answering $|S| \cdot |T|$ point-to-point shortest path queries becomes the dominating factor. Unfortunately, even in this case *HL partially pruned* is faster on average for most ball sizes $|\mathcal{B}|$. Initially, *HL top down* is faster for small ball sizes due to the fact that its labels have slightly fewer entries. However, due to worse locality and the increased time

required to calculate the labels, *HL top down* is much slower than *HL partially pruned* in total. Nonetheless, in comparison to other many-to-many algorithms, both *HL top down* and *HL partially pruned* do not stand a chance as they require far more time to calculate all shortest path distances.

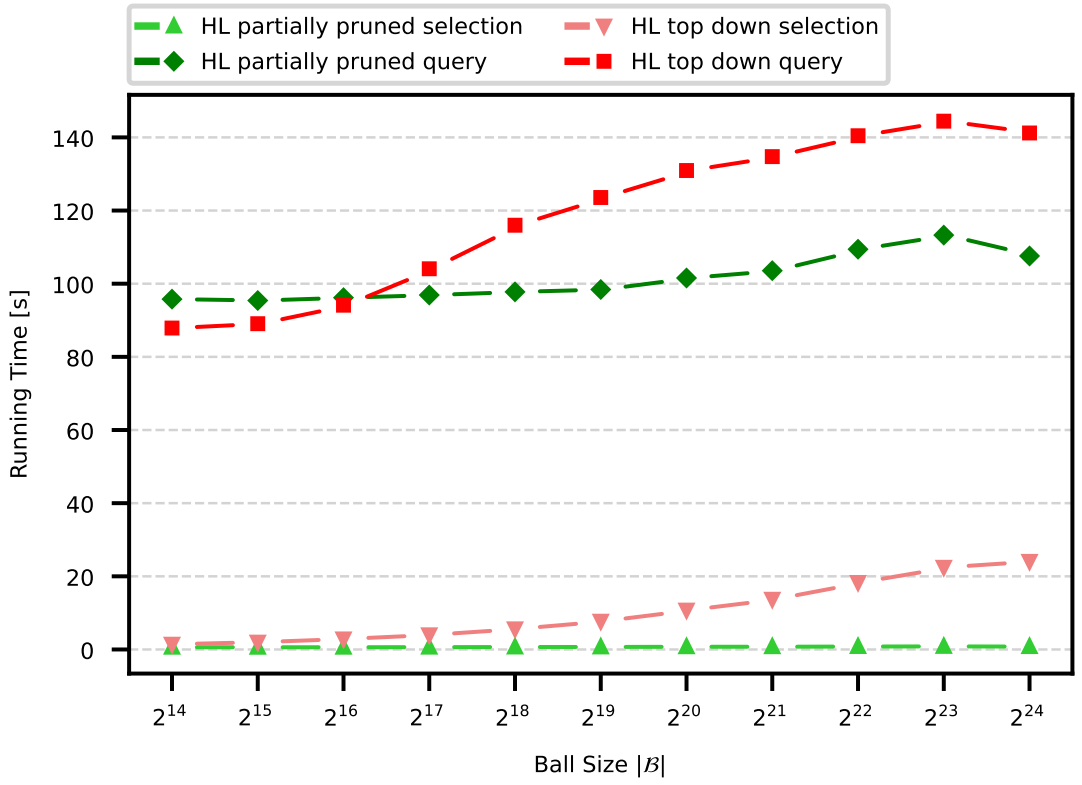


Figure 4.5.: Running times of HL algorithms with $|S| = |T| = 2^{14}$ sources and targets picked from separate balls of varying size $|\mathcal{B}|$.

Buckets

Finally, we are going to examine the different bucket-based algorithms. We refer to Figure 4.6 for the results of the many-to-many experiment in which we pick 2^{14} source and target vertices from different balls of the same size $|\mathcal{B}|$. Additionally, we refer to Appendix I for the results with different configurations.

As previously observed, during our one-to-many experiments in Section 4.3.2, the bucket-based query time increases with the size of the ball $|\mathcal{B}|$, in case source vertices are picked from a different ball than the targets. Hence, as the running time is dominated by the $|S|$ one-to-many queries, we can observe an up tick in the running time as we increase the size of the ball $|\mathcal{B}|$. Furthermore, *BCH* and *SBI* perform equivalently. The faster simultaneous bucket initialization is not able to significantly improve the running time, as it must be performed only once during a many-to-many query. Additionally, for $|S| = |T| = 2^{14}$, *BCH* is always faster than *SBI* as the stall-on-demand pruning algorithm populates the buckets with slightly fewer entries on average. In this case, where $|S| = |T| = 2^{14}$, *BCH* is about 830 ms faster than *SBI*. Moreover, *SBI Many-to-Many* which uses the SBI during the forward and backward search, is about 1.7 times faster than either bucket-based algorithm.

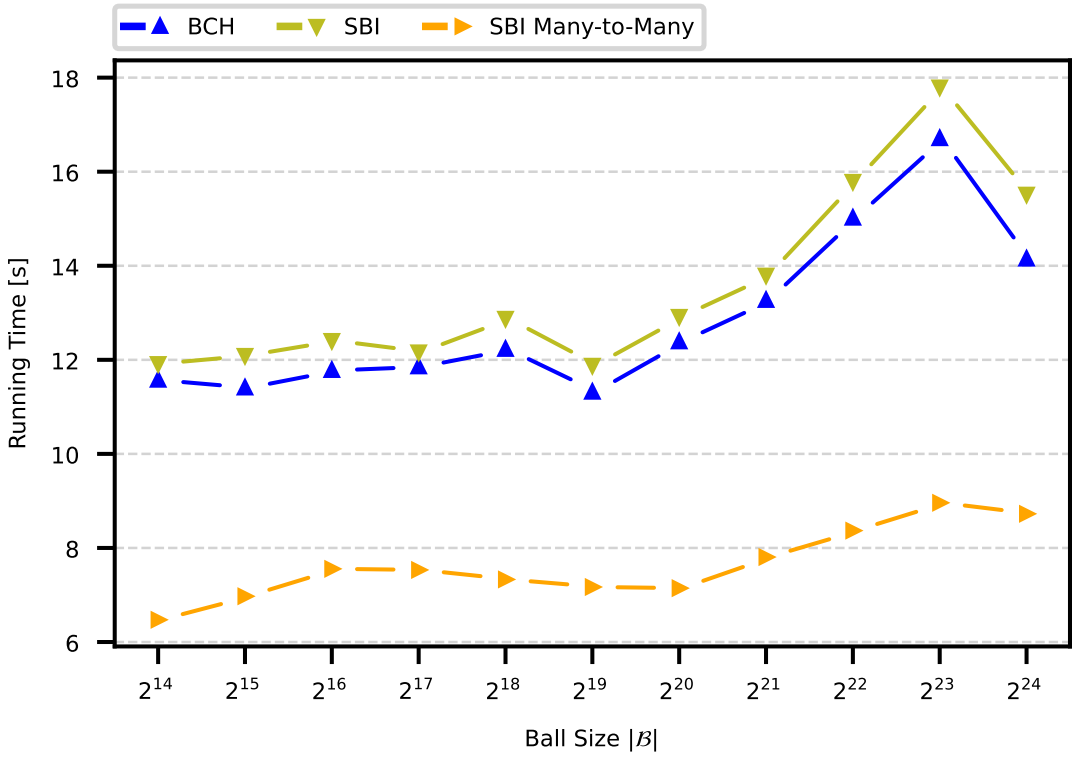


Figure 4.6.: Running times of bucket algorithms with $|S| = |T| = 2^{14}$ sources and targets picked from separate balls of varying size $|\mathcal{B}|$.

Batch Size Comparison

Before we compare the fastest algorithms in each category with each other, we first take a look on the effects of different batch sizes on the running times of selected algorithms. We conduct the following experiment with *RPHAST batched SBI*, *RPHAST batched rank*, and *Batched Lazy RPHAST*. Although *RPHAST batched SBI* and *RPHAST batched rank* are almost identical, we can observe a difference in their running times due to differences in their cache-friendliness.

To examine the effects of different batch sizes, we exponentially increase the batch size k from $k = 2$ up to $k = 2^{10}$ or $k = 2^{14}$, depending on the experiment. Furthermore, we only examine the effects of different batch sizes by using two configurations in the same setting as before, i.e. we pick source and target vertices from different balls of the same size $|\mathcal{B}|$. We perform the same experiment with $|S| = |T| = 2^{10}$ and $|S| = |T| = 2^{14}$. We refer to Appendix K, L and M for an overview of the effects that different batch sizes have on the running times of each algorithm. We rank all batch sizes k based on their average performance across all ball sizes $|\mathcal{B}|$. *#1* is associated with the batch size k for which all distances are calculated the most quickly. Supplementary, we present the range in which the running times vary for all algorithms in Figure 4.7 and in Figure 4.8.

Initially, we are able to improve the running times by increasing the batch size for *RPHAST batched SBI* and *RPHAST batched rank*. However, upon reaching some point, increasing the batch size further has an ever-increasing negative effect on the running time of each algorithm. Contrary to *RPHAST batched SBI* and *RPHAST batched rank*, we are able to improve the running time of *Batched Lazy RPHAST* by increasing the batch size up to the maximum of $k = |T|$.

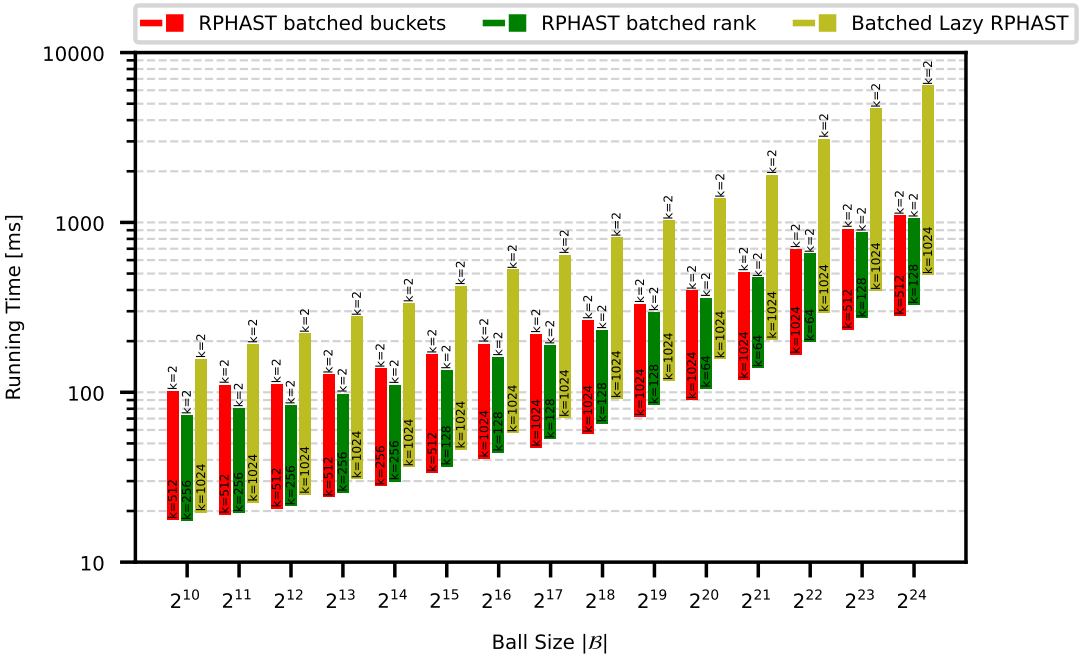


Figure 4.7.: Running times for selected algorithms with different batch sizes $k \in \{2^x \mid x \in (1, \ldots, 10)\}$. $|S| = |T| = 2^{10}$ source and target vertices picked from separate balls of varying size $|\mathcal{B}|$. Each bar represents the range in which running times fall for all batch sizes. We state the batch size k for which each algorithm calculates all shortest path distance the fastest at the bottom of each bar. Analogously, we state the batch size k for which each algorithm calculates all distances the slowest at the top of each bar.

As observed by Delling et at. [DGNW10], as we increase the batch size k we tend to evict data from the processor caches that might be useful. With growing k, we increase the memory required by the distance arrays that store all shortest path distances. Hence, as their size increases we must remove some, potential useful, data from the processor caches to make space for the data we currently access. Furthermore, varying the size of the source and target set can affect the batch size k for which each algorithm calculates all distances the fastest on average.

First we cover the results of our many-to-many experiment where $|S| = |T| = 2^{10}$. Initially, for smaller batch sizes, the average running time of *RPHAST batched SBI* and *RPHAST batched rank* improves while we increase the batch size k. By doubling the batch size from $k = 2$ to $k = 4$, both *RPHAST batched SBI* and *RPHAST batched rank* calculate all shortest path distances roughly 1.45 times faster. However, as we increase the batch size further, the advantage gained by batching multiple sources decreases, until it finally reverses. Although both algorithms only use a different algorithm to initialize the restricted distance array, their running times are affected differently by different batch sizes. *RPHAST batched SBI* is able to utilize larger batch sizes more efficiently than *RPHAST batched rank* as it uses memory more cache friendly. When picking $|S| = |T| = 2^{10}$ source and target vertices, *RPHAST batched SBI* is the fastest on average for $k = 512$. *RPHAST batched rank* is the fastest on average for a much smaller batch size of $k = 128$. Because *RPHAST batched rank* initializes k distance values for every reached vertex during the forward search, *RPHAST batched rank* allocates more memory in general compared to *RPHAST batched SBI*. Hence, *RPHAST batched rank* evicts other useful data sooner

than *RPHAST batched SBI* from the processor caches. That is why the performance of *RPHAST batched rank* reverses sooner for smaller batch sizes.

Contrary to *RPHAST batched SBI* and *RPHAST batched rank*, *Batched Lazy RPHAST* does not have a similar turning point up on which increasing the batch size has an increasingly negative effect on the running time. We are able to speed up the running time of *Batched Lazy RPHAST* by increasing the batch size further and further up to the maximum of $k = |T|$. However, for large batch sizes the advantage gained is not as significant as it is for smaller batch sizes. *Batched Lazy RPHAST* is about 9.49 times faster with $k = 1024$ compared to $k = 2$ and only 2.18 times faster with $k = 1024$ compared to $k = 16$.

Next we are going to focus on our batch size experiments in which $|S| = |T| = 2^{14}$. We give an overview for all algorithms with $|S| = |T| = 2^{14}$ in Figure 4.8.

By increasing the size of our source and target set, we further increase the memory occupied by the distance arrays of each algorithm. Hence, both *RPHAST batched SBI* and *RPHAST batched rank* perform the fastest for even smaller batch sizes k, compared to the previous experiment where $|S| = |T| = 2^{10}$. In this case, both *RPHAST batched SBI* and *RPHAST batched rank* are the fastest on average for $k = 64$.

Moreover, as stated before, *Batched Lazy RPHAST* is consistently faster for greater batch sizes. Hence, *Batched Lazy RPHAST* is the fastest on average for $k = 16384$. Because *Batched Lazy RPHAST* has no selection phase during which a fixed size distance array is allocated, memory allocation and access procedures are not cache friendly in general. Hence, there is no point upon which the running time decreases for greater batch sizes.

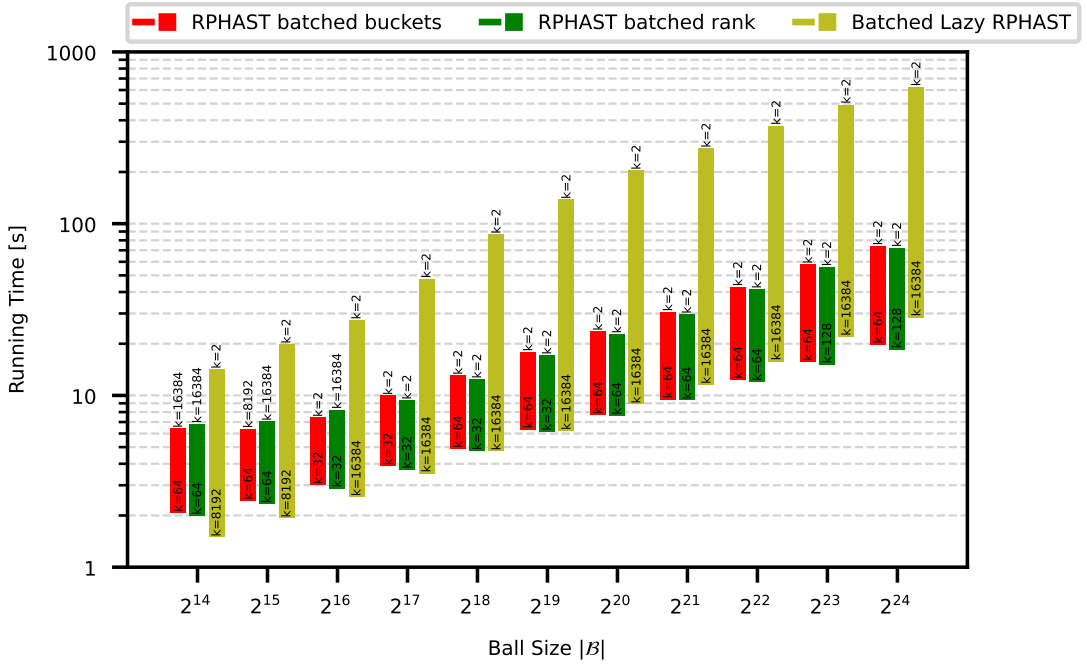


Figure 4.8.: Running times for selected algorithms with different batch sizes $k \in \{2^x \mid x \in (1, \ldots, 14)\}$. $|S| = |T| = 2^{14}$ source and target vertices picked from separate balls of varying size $|\mathcal{B}|$. Each bar represents the range in which running times fall for all batch sizes. We state the batch size k for which each algorithm calculates all shortest path distance the fastest at the bottom of each bar. Analogously, we state the batch size k for which each algorithm calculates all distances the slowest at the top of each bar.

Summary

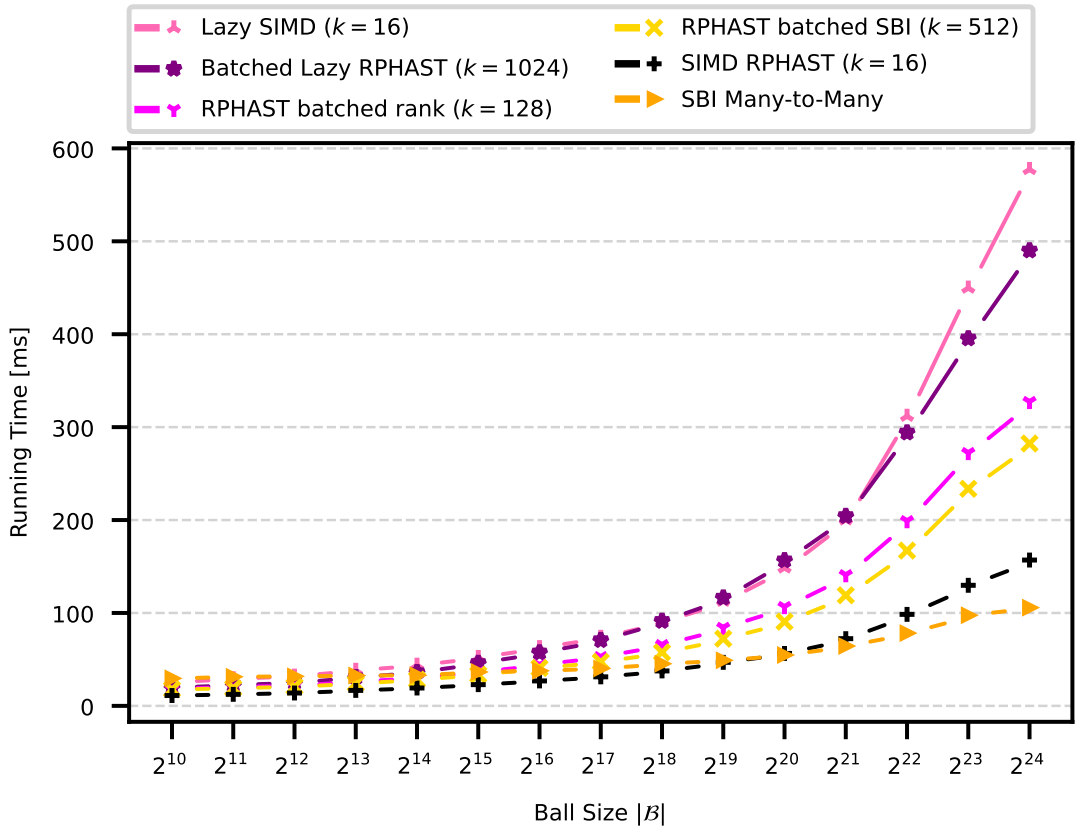


Figure 4.9.: Running times of many-to-many algorithms with $|S| = |T| = 2^{10}$ sources and targets picked from separate balls of varying size $|\mathcal{B}|$.

Lastly, we compare the fastest algorithms of each category in Figure 4.9 and in Figure 4.10. We use the batch size for which each algorithm calculates all shortest path distances the fastest, as determined in the previous paragraph. We discard both Hub Label based algorithms, as they perform worse than all other many-to-many algorithms.

First, we are going to cover the case where we pick 2^{10} source and target vertices in Figure 4.9. Initially, *Bached Lazy RPHAST* and *Lazy SIMD* perform equivalently for $|\mathcal{B}| \leq 2^{21}$. However, for $|\mathcal{B}| > 2^{21}$ *Batched Lazy RPHAST* is faster than *Lazy SIMD*. The SIMD instructions with $k = 16$ are not able to compete with the much bigger batch size of $k = 1024$ as used by *Batched Lazy RPHAST*. Nonetheless, in comparison to all other portrayed algorithms, both *Batched Lazy RPHAST* and *Lazy SIMD* require more time to calculate all shortest path distances when $|\mathcal{B}| > 2^{13}$. Additionally, *RPHAST batched SBI* with $k = 512$ is about 1.48 times faster on average than *Batched Lazy RPHAST* with $k = 1024$.

Contrary to *Lazy SIMD*, using SIMD instructions makes *SIMD RPHAST* always the fastest algorithm compared to *RPHAST batched SBI* and *RPHAST batched rank*. Moreover, *SIMD RPHAST* is about 1.58 times faster than *RPHAST batched SBI* on average.

SBI Many-to-Many is initially the slowest algorithm when $|\mathcal{B}| = 2^{14}$, however as we increase the size of the ball $|\mathcal{B}|$ the running time of *SBI Many-to-Many* increases the least in comparison to all other algorithms. Hence, *SBI Many-to-Many* is the fastest algorithm for $|\mathcal{B}| > 2^{20}$ when picking $|S| = |T| = 2^{10}$ source and target vertices. Furthermore, *SBI*

Many-to-Many is about 1.48 times faster than *SIMD RPHAST* for $|\mathcal{B}| = 2^{24}$ (157 ms for *SIMD RPHAST* compared to 106 ms for *RPHAST batched SBI*).

Finally, we shortly compare the fastest algorithms of each category when picking 2^{14} source and target vertices. We refer to Figure 4.10 for the results.

Because *Batched Lazy RPHAST* performs the fastest for $k = |T|$, increasing the size of the target set increases the disparity between *Lazy SIMD* and *Batched Lazy RPHAST* further. In this case, *Batched Lazy RPHAST* with $k = 16384$ is about 1.4 times faster on average than *Lazy SIMD* with $k = 16$.

Furthermore, initially, for smaller ball sizes $|\mathcal{B}| < 2^{18}$, *SBI Many-to-Many* requires far more time than all other algorithms to calculate all shortest path distances. However, *SBI Many-to-Many* eventually beats *SIMD RPHAST* when sources and targets are spread over a large area. We are able to observe this, when picking 2^{14} source and target vertices from a ball $\mathcal{B}$ that has at least 2^{23} vertices to pick from. In this case *SBI Many-to-Many* needs 8.73 s while *SIMD RPHAST* needs 11.91 s to calculate all many-to-many shortest path distances.

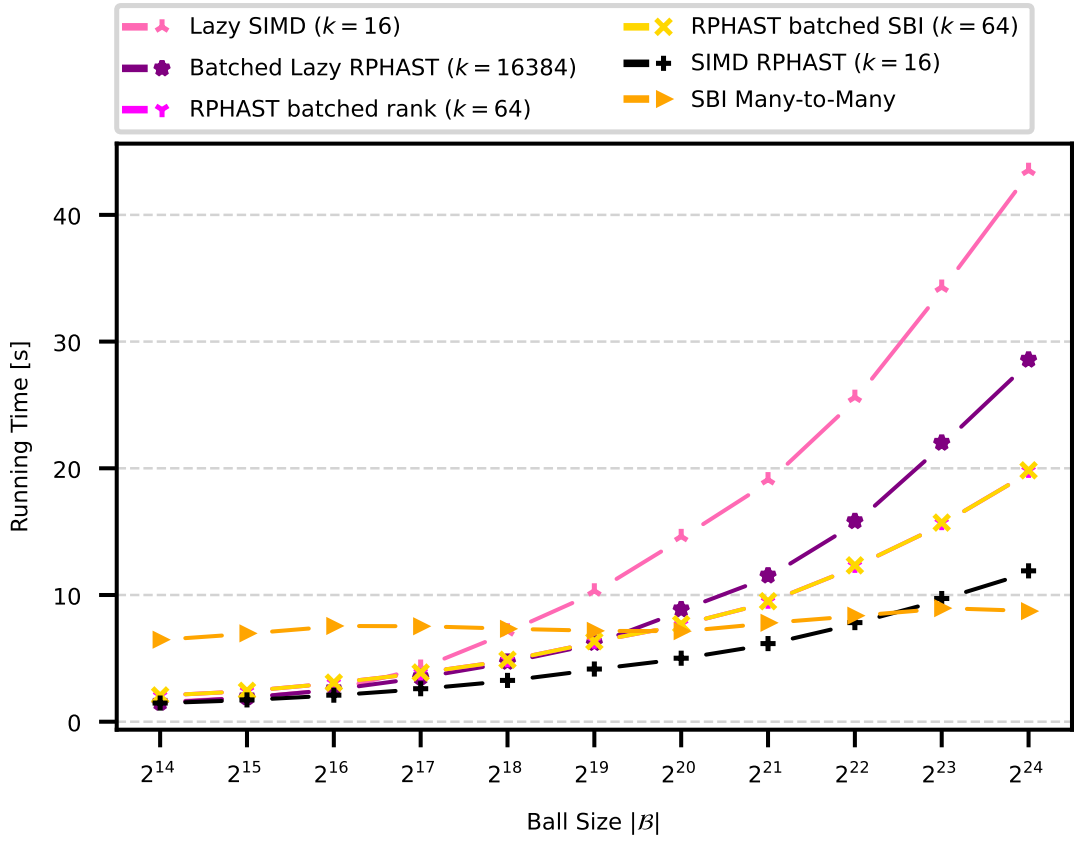


Figure 4.10.: Running times of many-to-many algorithms with $|S| = |T| = 2^{14}$ sources and targets picked from separate balls of varying size $|\mathcal{B}|$.

4.4.2. Asymmetric Cases

In this section, we focus on asymmetric cases with $|S| > |T|$ and $|S| < |T|$. For simplicity, we only consider one case each. In Figure 4.11 we display the running times of many-to-many algorithms with either $|S| < |T|$ or $|S| > |T|$, depending on which configuration can

be computed faster by the algorithm. Furthermore, we refer to Appendix J for additional figures that depict all configurations.

We categorize all algorithms into two categories, based on which configuration is better suited for the algorithm. We give an overview of assigned categories in Table 4.1.

Algorithms that are better suited to solve the many-to-many problem with $|S| < |T|$ are *RPHAST*, *BCH*, and *SBI*. All these algorithms solve the many-to-many problem by answering $|S|$ one-to-many queries. The target selection must only be performed once during a many-to-many query, hence the target selection does not influence the running time much. Consequentially, as running times are dominated by the $|S|$ one-to-many queries, these algorithms perform better when fewer sources are picked.

On the other hand, *Batched Lazy RPHAST* is the only algorithm that is better suited to solve the many-to-many problem where $|S| > |T|$. Because the running time of *Batched Lazy RPHAST* is dominated by the $|T|$ many-to-one queries. Hence, *Batched Lazy RPHAST* is faster for smaller target set sizes. Although *Batched Lazy RPHAST* is more fitted to solve the many-to-one problem, all batched RPHAST variants are able to keep up with *Batched Lazy RPHAST* when $|S| > |T|$.

Lastly we focus on *SBI Many-to-Many*. *SBI Many-to-Many* is the only algorithm that performs equally in either case. Because the algorithm is completely symmetric, i.e. it uses the SBI during the forward and backward search, *SBI Many-to-Many* is able to cope with either case equally well.

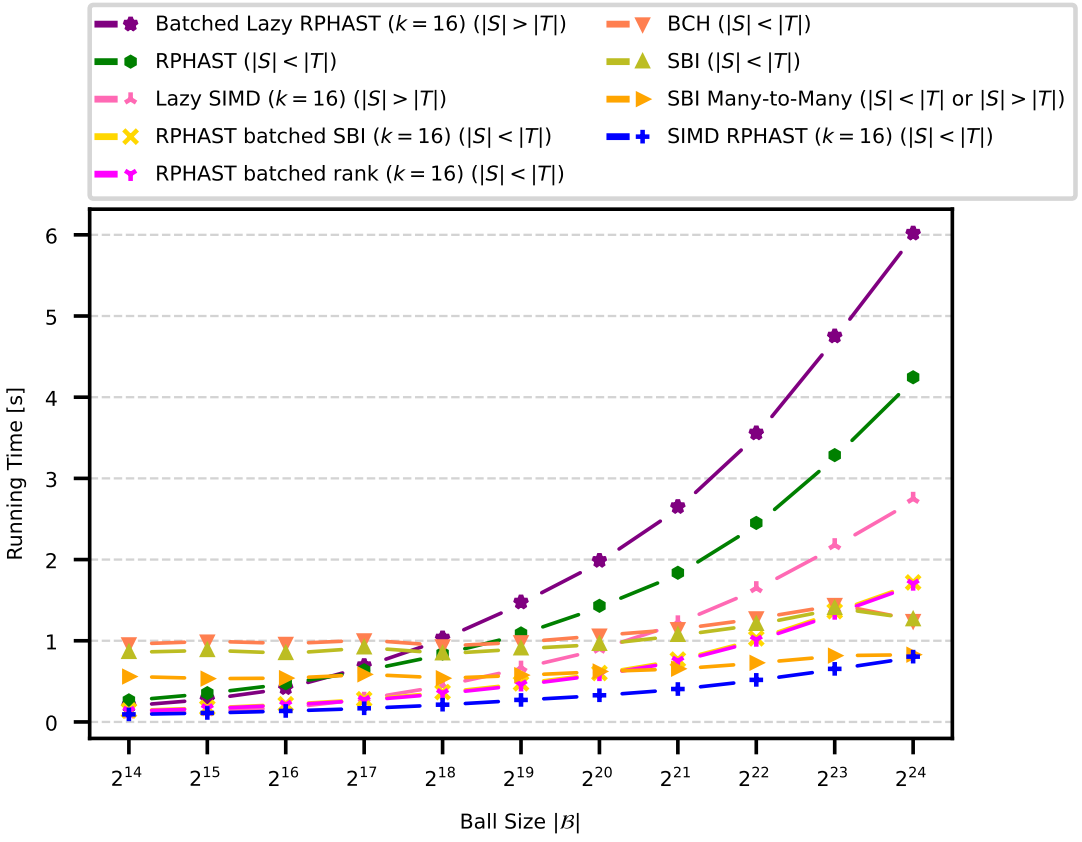


Figure 4.11.: Running times of many-to-many algorithms with either $|S| = 2^{10}$ and $|T| = 2^{14}$ or $|S| = 2^{14}$ and $|T| = 2^{10}$ vertices picked from different balls of varying size $|\mathcal{B}|$.

	$\|S\| > \|T\|$	$\|S\| < \|T\|$
BCH		✓
SBI		✓
Batched Lazy RPHAST	✓	
RHAST batched SBI		✓
RHAST batched rank		✓
RHAST batched dijkstra		✓
SIMD RPHAST		✓
SBI Many-to-Many	✓	✓

Table 4.1.: Algorithm classifications based on which configuration can be computed faster by the algorithm.

4.4.3. Sources equal Targets

Lastly, we briefly cover the special case in which $S = T$. We use three different sizes for the size of the source and target set, $|S| = |T| = 2^{10}$, $|S| = |T| = 2^{12}$, and $|S| = |T| = 2^{14}$.

In such a scenario, most algorithms perform equally compared to other scenarios in which we pick our source and target vertices from two separate balls, as we did in Section 4.4.1. Unfortunately, the bucket-based algorithms perform worse in this case. Especially for smaller ball sizes $|\mathcal{B}|$, all bucket-based algorithms require more time to calculate the shortest path distances. We present the running times of all bucket-based algorithms, when $|S| = |T| = 2^{14}$, in Figure 4.12. As the set of sources is equal to the set of targets, both the forward searches and the target selection settle almost the same vertices. Hence, most vertices visited by the forward search have populated buckets. Consequentially, running times are worse for smaller ball sizes $|\mathcal{B}|$ as the average number of entries per scanned bucket is greater compared to when targets are spread further apart, as they were in our previous experiment in Figure 4.6. Moreover, *BCH* and *SBI* are about 2.16 times faster when $|\mathcal{B}| = 2^{24}$ compared to when all vertices are in proximity to each other, i.e. $|\mathcal{B}| = 2^{14}$. The running time of *SBI Many-to-Many* decreases slightly more from 23.65 ms down to 8.71 ms, which is about 2.7 times faster.

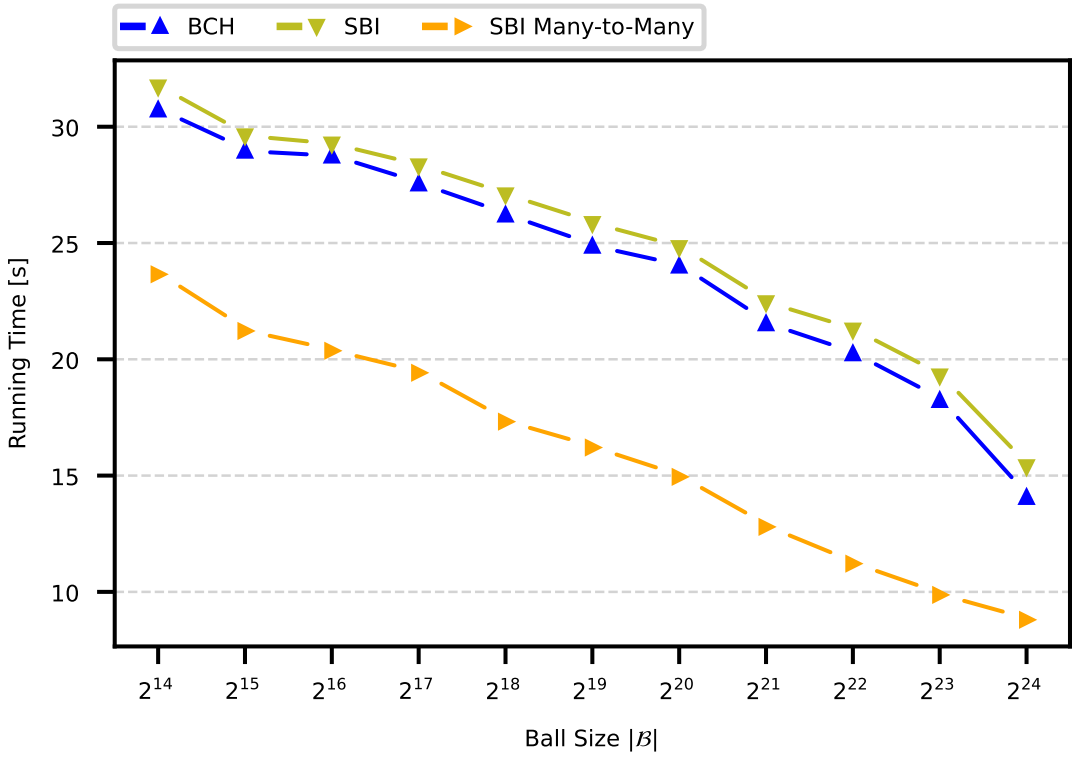


Figure 4.12.: Running times of many-to-many algorithms with 2^{14} sources and targets $S = T$ picked from a ball of varying size $|\mathcal{B}|$.

4.5. Bucket Pruning Algorithms

In the following, we compare the effectiveness of both bucket pruning algorithms. We compare the stall-on-demand technique (Section 3.2.2) against the retrospective pruning (Section 3.2.3).

As before, we use a ball $\mathcal{B} \subseteq V$ of varying size to obtain our set of target vertices. We present the number of total entries and the average number of entries per bucket as populated by each algorithm in Table 4.2. We pick 100 different sets of target vertices. Each set of target vertices T contains 2^{14} vertices picked from a different ball $\mathcal{B}$ of varying size. Furthermore, we present the average values of all 100 target sets in Table 4.2.

As evident in the table, both algorithms are able to reduce the total number of bucket entries by roughly 80 percent. Furthermore, the stall-on-demand technique is usually able to remove slightly more unnecessary bucket entries. Lastly, by spreading targets further apart, the average number of entries per populated bucket decreases while the total entry count remains the same.

	No Pruning		Stall-on-Demand (BCH)			Retrospective (SBI)		
$\lvert\mathcal{B}\rvert$	#entries [$\times 10^6$]	avg	#entries [$\times 10^6$]	avg	reduction [%]	#entries [$\times 10^6$]	avg	reduction [%]
2^{14}	10.5	579.2	1.9	107.8	-82.21	2.1	114.7	-80.32
2^{15}	9.2	353.9	1.8	72.6	-80.18	2.0	78.4	-77.93
2^{16}	9.7	267.9	1.8	52.3	-81.15	2.1	57.0	-78.79
2^{17}	9.8	197.1	1.8	38.3	-81.25	2.1	41.2	-79.13
2^{18}	10.1	153.1	1.9	29.7	-81.45	2.1	31.7	-79.32
2^{19}	10.4	117.5	1.9	22.3	-82.10	2.1	23.7	-79.85
2^{20}	10.7	92.5	1.9	17.8	-82.24	2.1	18.5	-80.08
2^{21}	10.7	72.9	1.9	14.3	-82.26	2.1	14.5	-80.15
2^{22}	11.6	60.2	1.9	11.6	-83.20	2.2	11.4	-81.20
2^{23}	11.7	47.9	2.0	9.4	-83.32	2.2	9.0	-81.36
2^{24}	10.5	34.4	1.9	7.6	-81.82	2.1	7.1	-79.59

Table 4.2.: Bucket size for different pruning algorithms; $|T| = 2^{14}$ targets picked from a ball of varying size $|\mathcal{B}|$.
All values are the average of 100 different target sets T. We denote the total number of bucket entries produced by each algorithm with *#entries*. Additionally, *avg* refers to the average number of entries per populated bucket. Hence, *avg* is equal to *#entries* divided by the number of unique vertices settled during the target selection. Lastly, *reduction [%]* refers to the relative reduction in total entries compared to if no pruning is applied.

5. Conclusion

In this thesis we implemented and evaluated various Contraction Hierarchy based shortest path algorithms for the one-to-many and the many-to-many problem. Additionally, we proposed different extensions and recombinations of existing algorithms that have proven to be successful.

By using the restricted distance array $d_{T,i}[v']$ for the second part of an RPHAST query, we are able to efficiently batch all source vertices in one batch. As determined in our experiments, increasing the batch size above $k = 16$ has a positive effect on the performance of the RPHAST algorithm. However, upon reaching some point, increasing the batch size further has an ever-increasing negative effect on the running times for some algorithms.

Moreover, we introduced new algorithms based on existing concepts that have proven to be able to keep up with existing algorithms and even beat them in some scenarios. As shown in our one-to-many experiments, Lazy RPHAST is the fastest algorithm to solve the one-to-many shortest path problem, in case the set of target vertices changes between consecutive queries.

Lastly, our simultaneous bucket initialization has proven itself as a valuable building block in different algorithms. As originally intended, the simultaneous bucket initialization is able to speed up the selection time of the established bucket-based algorithm as initially proposed by Knopp et al. [KSS+07]. Especially for the one-to-many problem when all targets are in proximity to each other, the simultaneous bucket initialization is able to improve the running time significantly. By combining the simultaneous bucket initialization with a batched RPHAST query, we are able to utilize larger batch sizes more efficiently, as the simultaneous bucket initialization requires comparatively less memory. Last but not least, our proposed *SBI Many-to-Many* algorithm that uses the simultaneous bucket initialization for the forward and backward search, is about 1.7 times faster than the original bucket-based algorithm. In some cases where source and target vertices are spread over a large area, the *SBI Many-to-Many* algorithm even beats the *SIMD RPHAST* algorithm.

Bibliography

[ADGW10] Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. A Hub-Based Labeling Algorithm for Shortest Paths on Road Networks. Technical Report MSR-TR-2010-165, Microsoft Research, 2010.

[ADGW12] Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. Hierarchical Hub Labelings for Shortest Paths. In *Proceedings of the 20th Annual European Symposium on Algorithms (ESA'12)*, volume 7501 of *Lecture Notes in Computer Science*, pages 24–35. Springer, 2012.

[BDG+16] Hannah Bast, Daniel Delling, Andrew Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F. Werneck. *Route Planning in Transportation Networks*, pages 19–80. Springer International Publishing, Cham, 2016.

[DGJ09] Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson, editors. *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, volume 74 of *DIMACS Book*. American Mathematical Society, 2009.

[DGNW10] Daniel Delling, Andrew V. Goldberg, Andreas Nowatzyk, and Renato F. Werneck. PHAST: Hardware-Accelerated Shortest Path Trees. Technical Report MSR-TR-2010-125, Microsoft Research, 2010.

[DGW11] Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. Faster Batched Shortest Paths in Road Networks. In *Proceedings of the 11th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'11)*, volume 20 of *OpenAccess Series in Informatics (OASIcs)*, pages 52–63, 2011.

[Dij59] Edsger W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1:269–271, 1959.

[EFH+11] Jochen Eisner, Stefan Funke, Andre Herbst, Andreas Spillner, and Sabine Storandt. Algorithms for Matching and Predicting Trajectories. In *Proceedings of the 13th Workshop on Algorithm Engineering and Experiments (ALENEX'11)*, pages 84–95. SIAM, 2011.

[FT87] Michael L. Fredman and Robert E. Tarjan. Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms. *Journal of the ACM*, 34(3):596–615, July 1987.

[GSSD08] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In *Proceedings of the 7th Workshop on Experimental Algorithms (WEA'08)*, volume 5038 of *Lecture Notes in Computer Science*, pages 319–333. Springer, June 2008.

[HNR68] Peter E. Hart, Nils Nilsson, and Bertram Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4:100–107, 1968.

[JS19] George R. Jagadeesh and Thambipillai Srikanthan. Fast computation of clustered many-to-many shortest paths and its application to map matching. *ACM Trans. Spatial Algorithms Syst.*, 5(3), aug 2019.

[KSS+07] Sebastian Knopp, Peter Sanders, Dominik Schultes, Frank Schulz, and Dorothea Wagner. Computing Many-to-Many Shortest Paths Using Highway Hierarchies. In *Proceedings of the 9th Workshop on Algorithm Engineering and Experiments (ALENEX'07)*, pages 36–45. SIAM, 2007.

[SS06] Peter Sanders and Dominik Schultes. Engineering Highway Hierarchies. In *Proceedings of the 14th Annual European Symposium on Algorithms (ESA'06)*, volume 4168 of *Lecture Notes in Computer Science*, pages 804–816. Springer, 2006.

[SS07] Dominik Schultes and Peter Sanders. Dynamic Highway-Node Routing. In *Proceedings of the 6th Workshop on Experimental Algorithms (WEA'07)*, volume 4525 of *Lecture Notes in Computer Science*, pages 66–79. Springer, June 2007.

[SZ21] Ben Strasser and Tim Zeitz. A Fast and Tight Heuristic for A* in Road Networks. In David Coudert and Emanuele Natale, editors, *19th International Symposium on Experimental Algorithms (SEA 2021)*, volume 190 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 6:1–6:16, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

Appendix

A. One-to-Many Selection Times - Different Balls

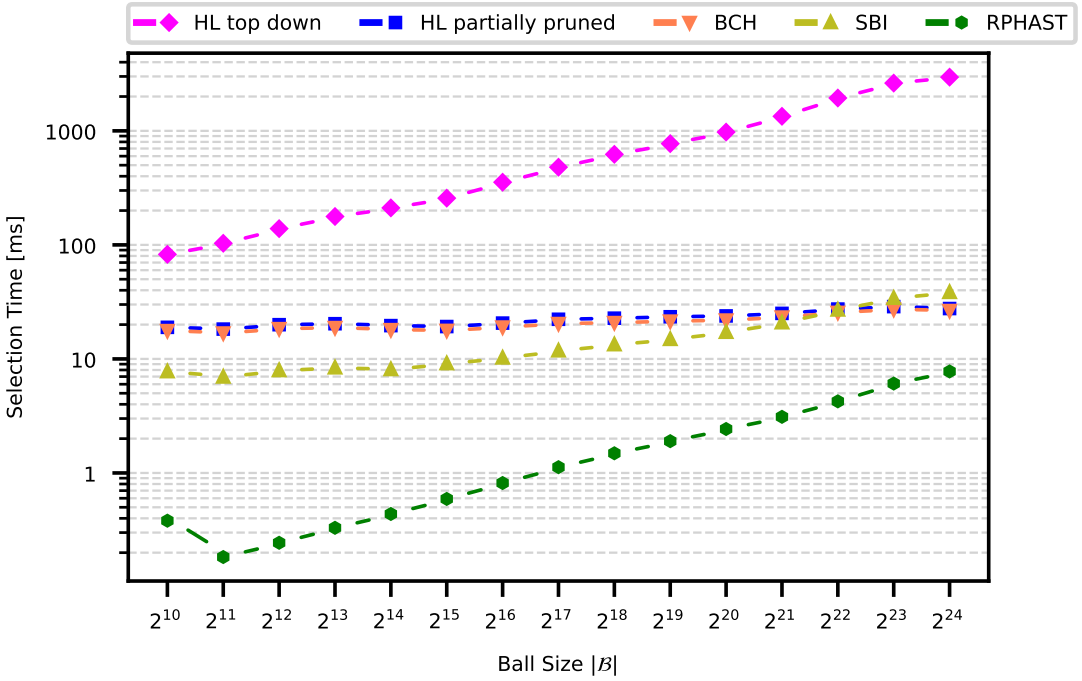


Figure A.1.: Selection times of one-to-many algorithms with $|T| = 2^{10}$ targets picked from a ball of varying size $|\mathcal{B}|$; Source picked separately from V.

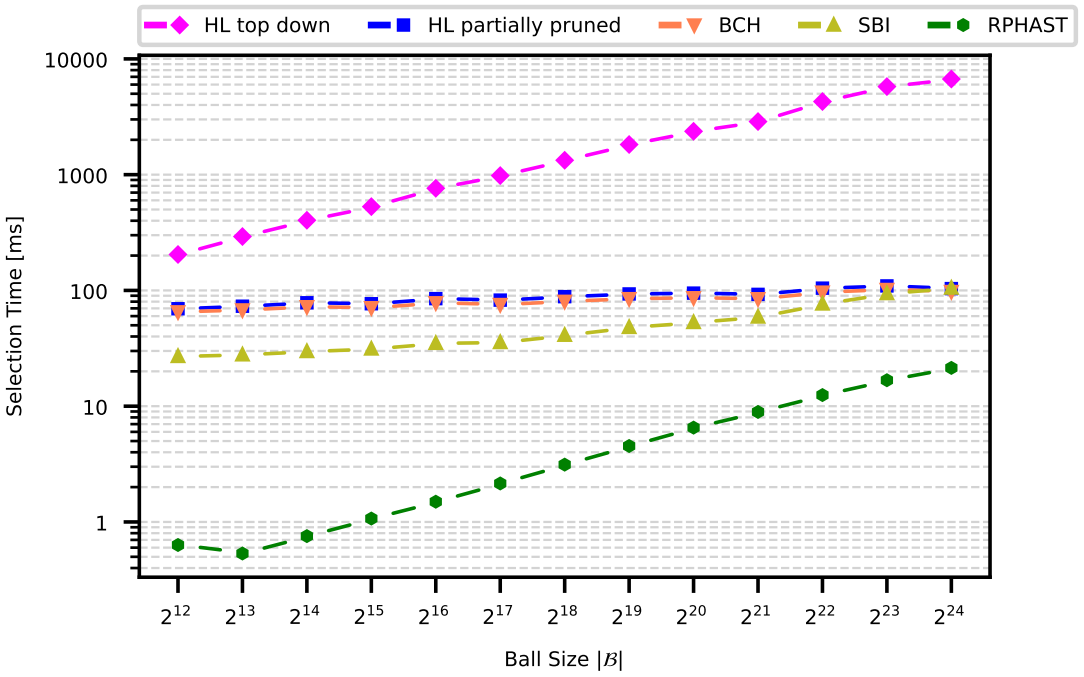


Figure A.2.: Selection times of one-to-many algorithms with $|T| = 2^{12}$ targets picked from a ball of varying size $|\mathcal{B}|$; Source picked separately from V.

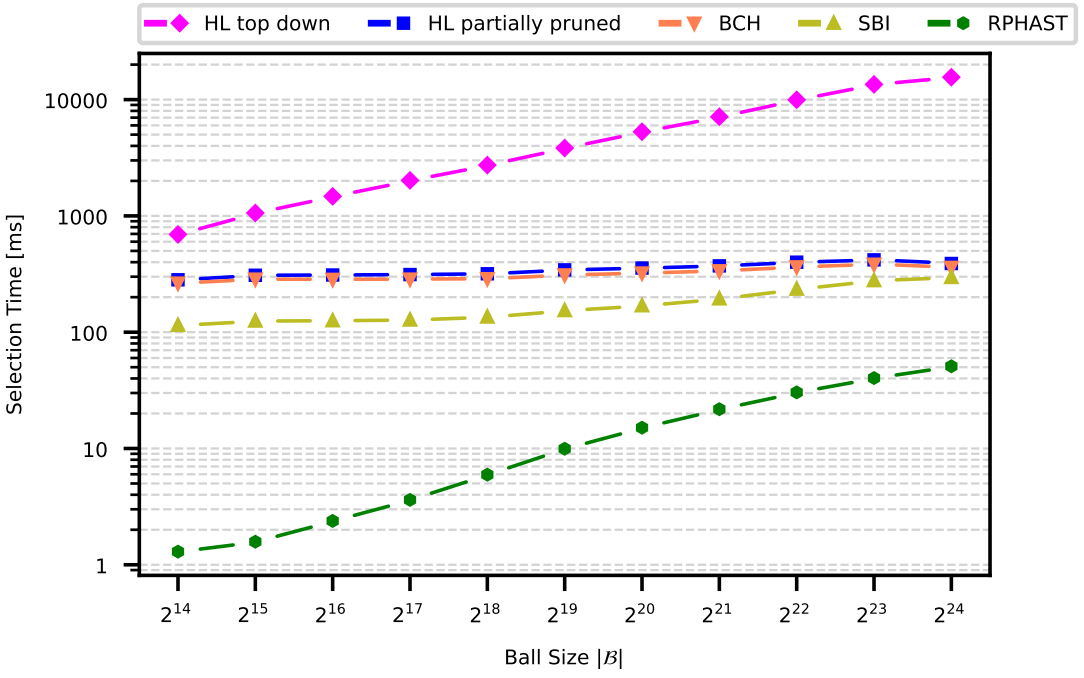


Figure A.3.: Selection times of one-to-many algorithms with $|T| = 2^{14}$ targets picked from a ball of varying size $|\mathcal{B}|$; Source picked separately from V.

B. One-to-Many Query Times - Different Balls

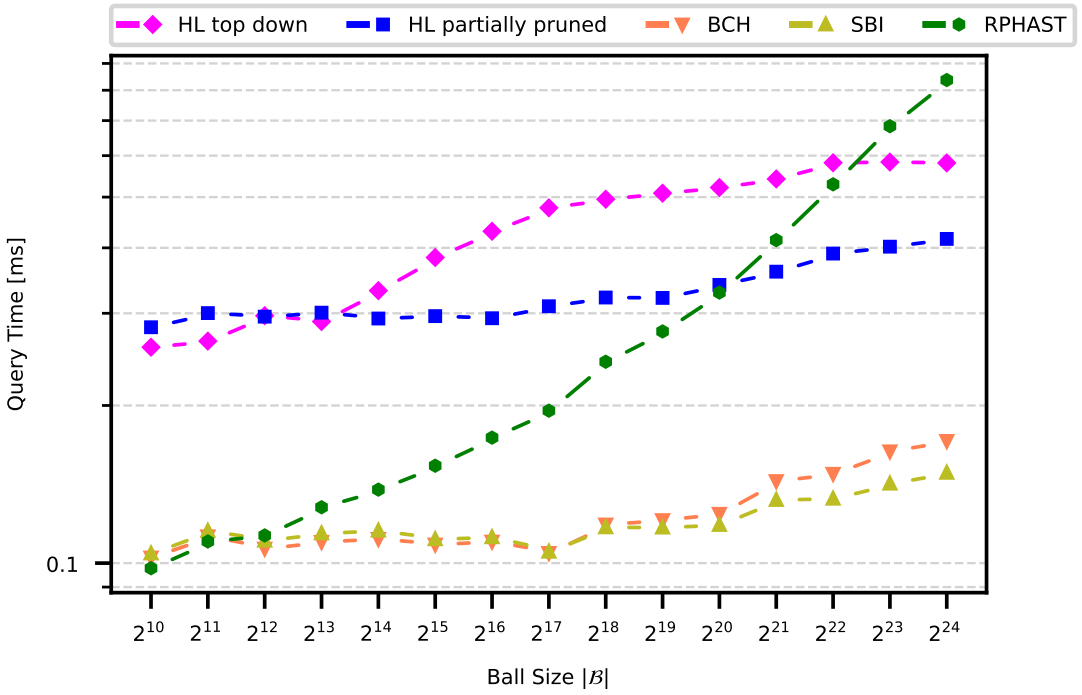


Figure B.4.: Query times of one-to-many algorithms with $|T| = 2^{10}$ targets picked from a ball of varying size $|\mathcal{B}|$; Source picked separately from V.

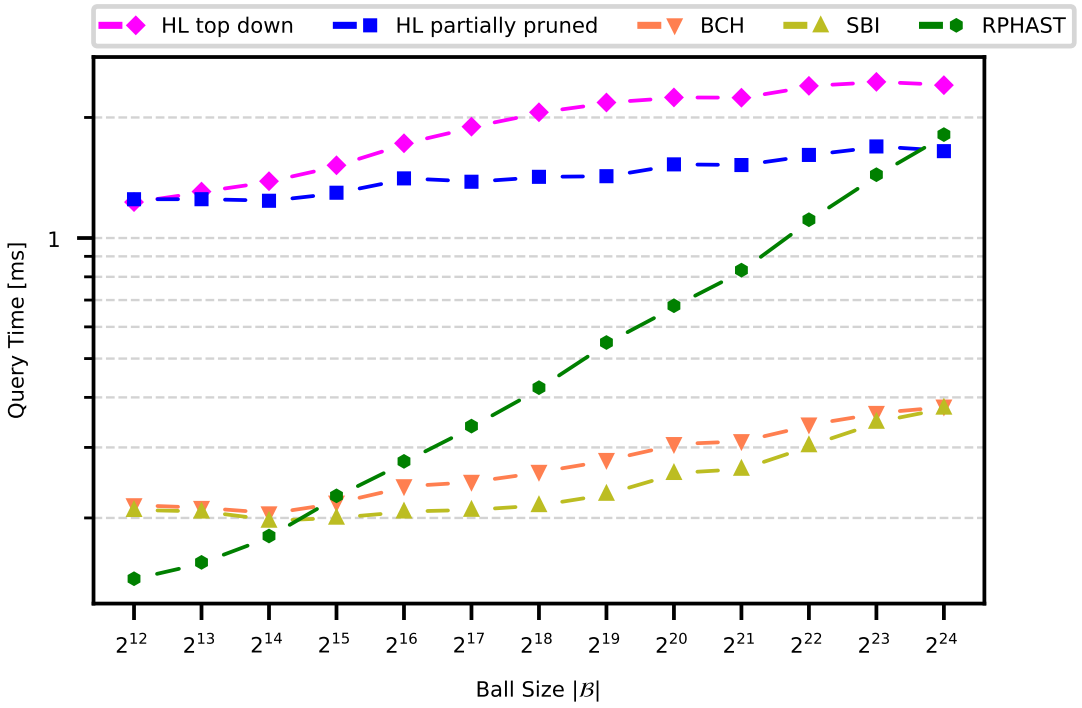


Figure B.5.: Query times of one-to-many algorithms with $|T| = 2^{12}$ targets picked from a ball of varying size $|\mathcal{B}|$; Source picked separately from V.

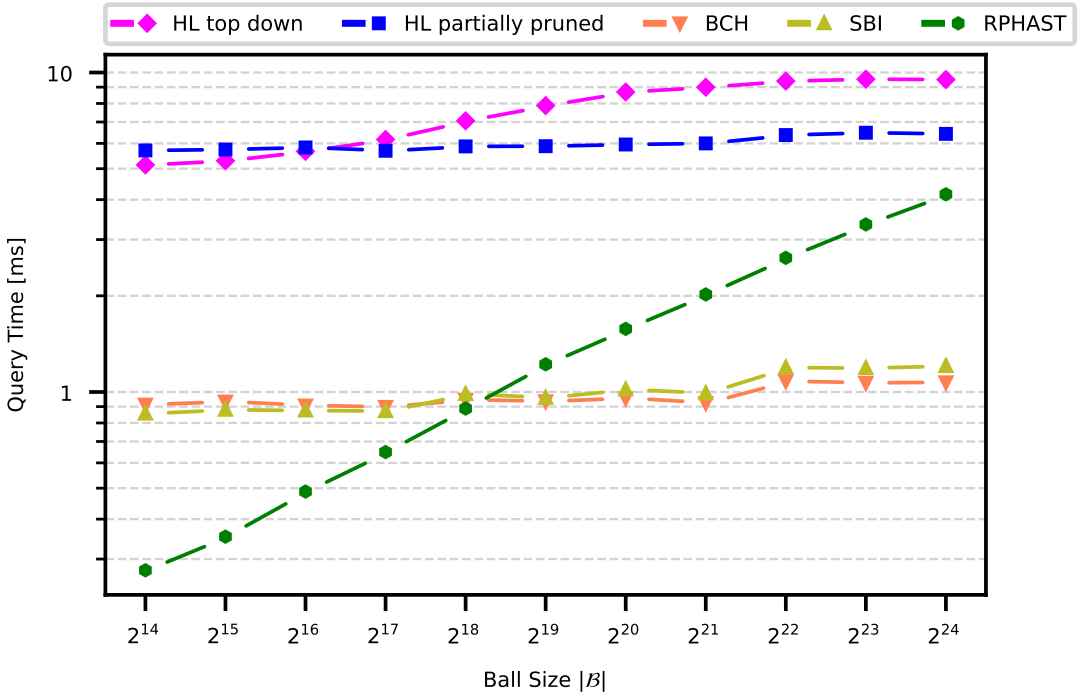


Figure B.6.: Query times of one-to-many algorithms with $|T| = 2^{14}$ targets picked from a ball of varying size $|\mathcal{B}|$; Source picked separately from V.

C. One-to-Many Selection+Query Times - Different Balls

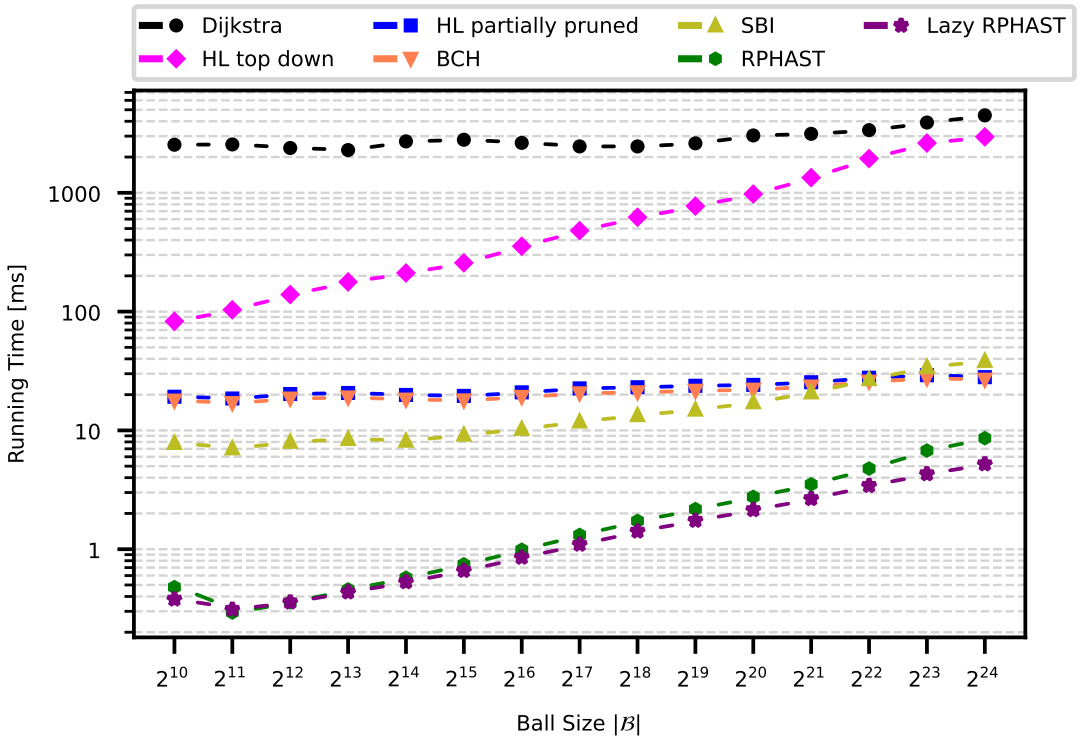


Figure C.7.: Running times of one-to-many algorithms with $|T| = 2^{10}$ targets picked from a ball of varying size $|\mathcal{B}|$; Source picked separately from V.

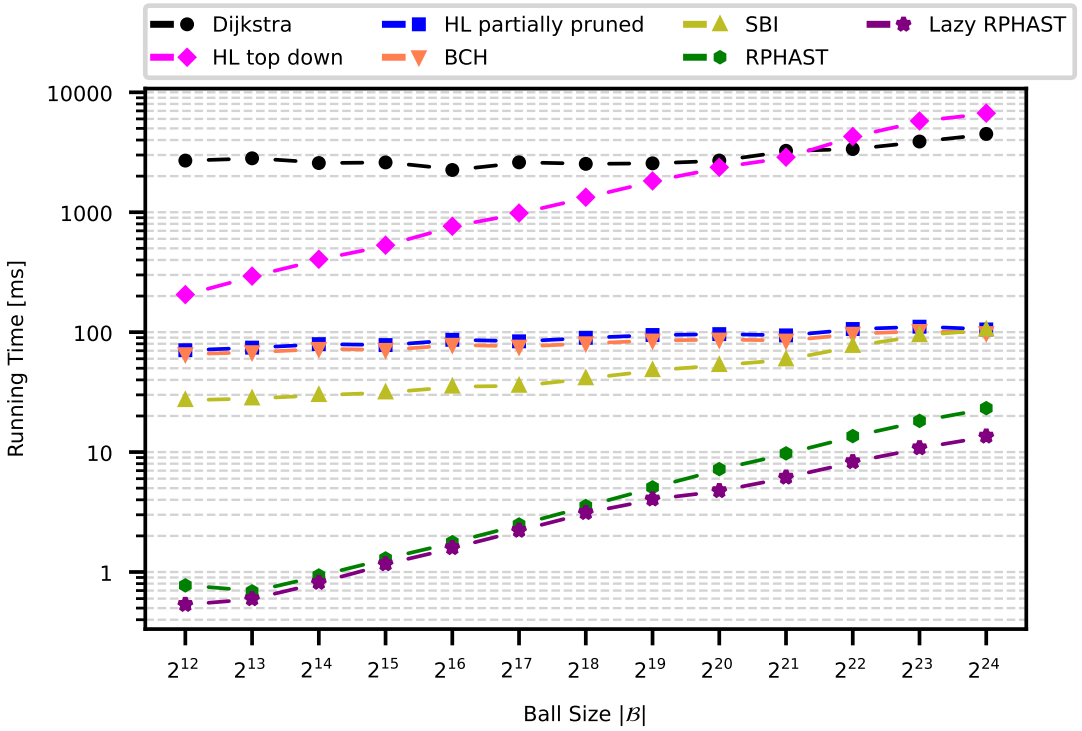


Figure C.8.: Running times of one-to-many algorithms with $|T| = 2^{12}$ targets picked from a ball of varying size $|\mathcal{B}|$; Source picked separately from V.

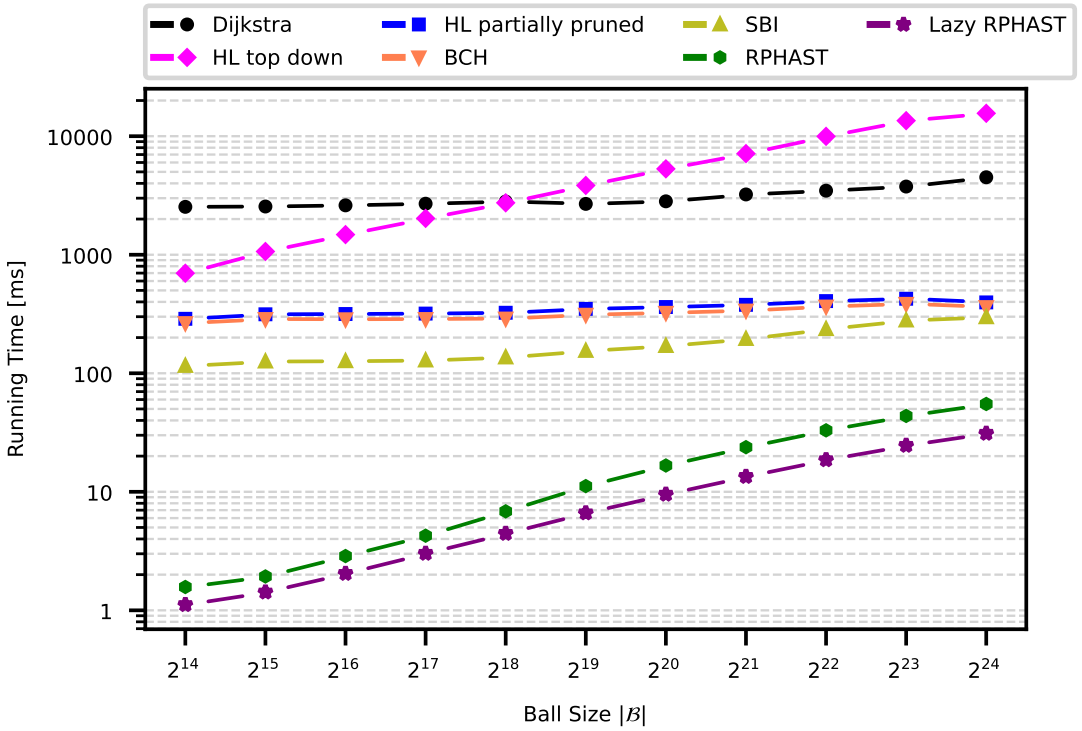


Figure C.9.: Running times of one-to-many algorithms with $|T| = 2^{14}$ targets picked from a ball of varying size $|\mathcal{B}|$; Source picked separately from V.

D. One-to-Many Selection Times - Same Ball

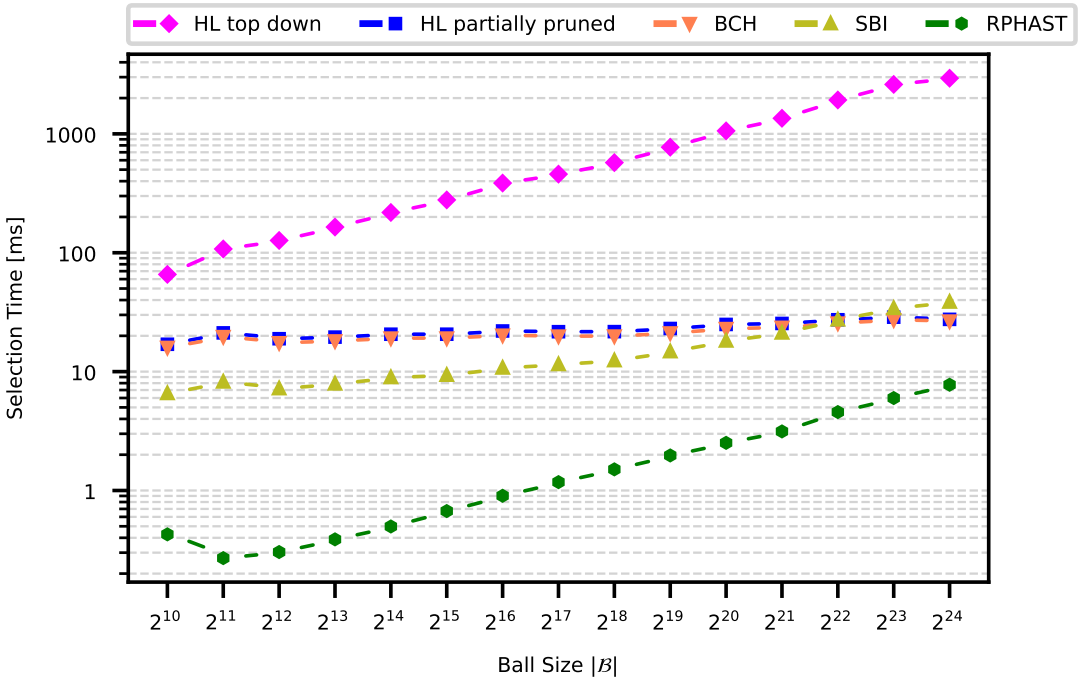


Figure D.10.: Selection times of one-to-many algorithms with $|T| = 2^{10}$ targets picked from a ball of varying size $|\mathcal{B}|$; Source picked from the same ball.

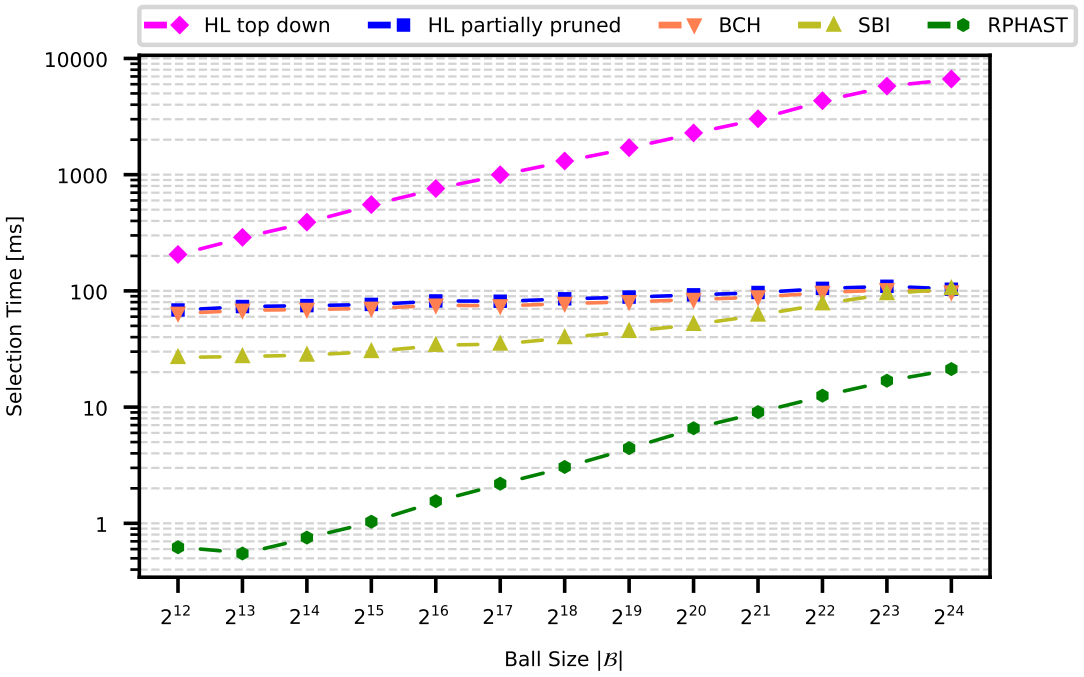


Figure D.11.: Selection times of one-to-many algorithms with $|T| = 2^{12}$ targets picked from a ball of varying size $|\mathcal{B}|$; Source picked from the same ball.

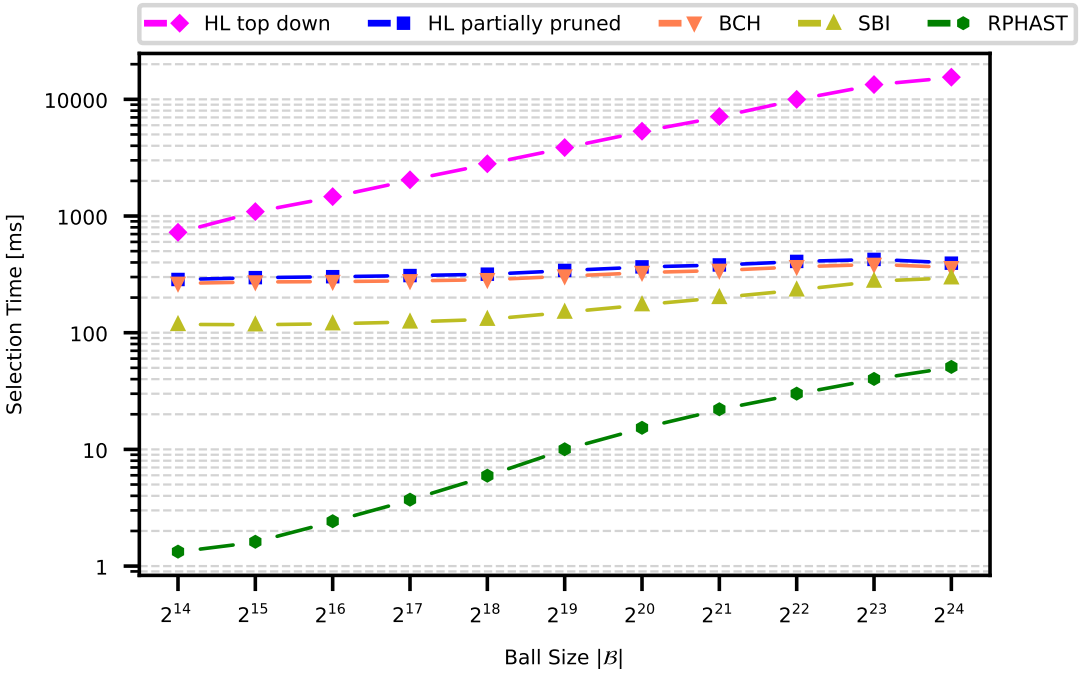


Figure D.12.: Selection times of one-to-many algorithms with $|T| = 2^{14}$ targets picked from a ball of varying size $|\mathcal{B}|$; Source picked from the same ball.

E. One-to-Many Selection Times - Same Ball

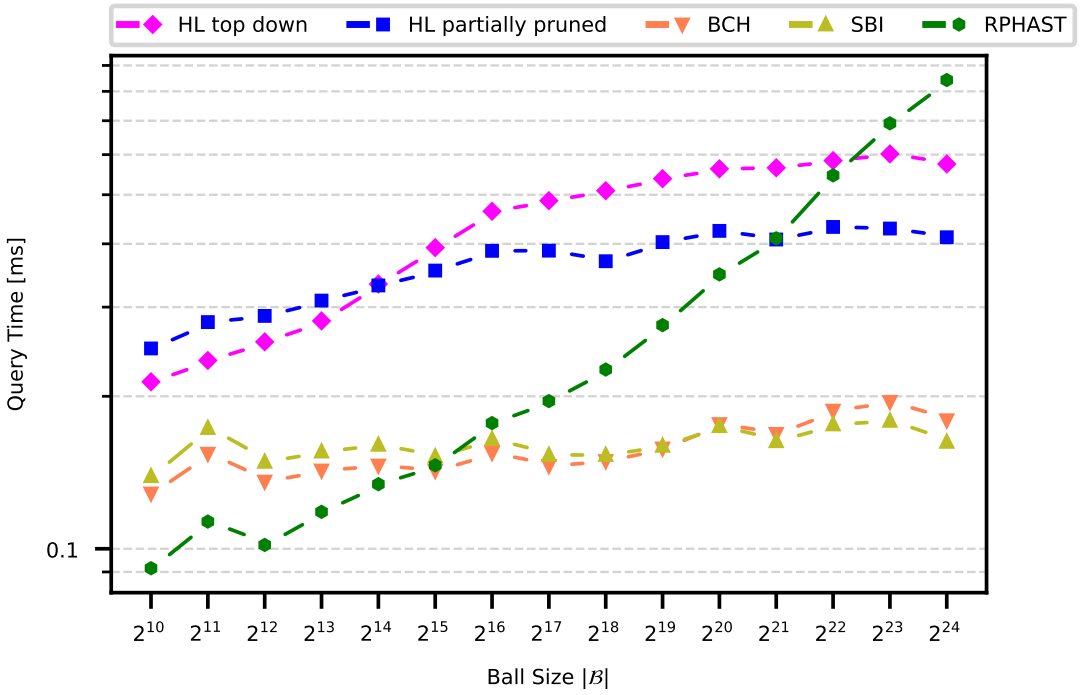


Figure E.13.: Query times of one-to-many algorithms with $|T| = 2^{10}$ targets picked from the a ball of varying size $|\mathcal{B}|$; Source picked from the same ball.

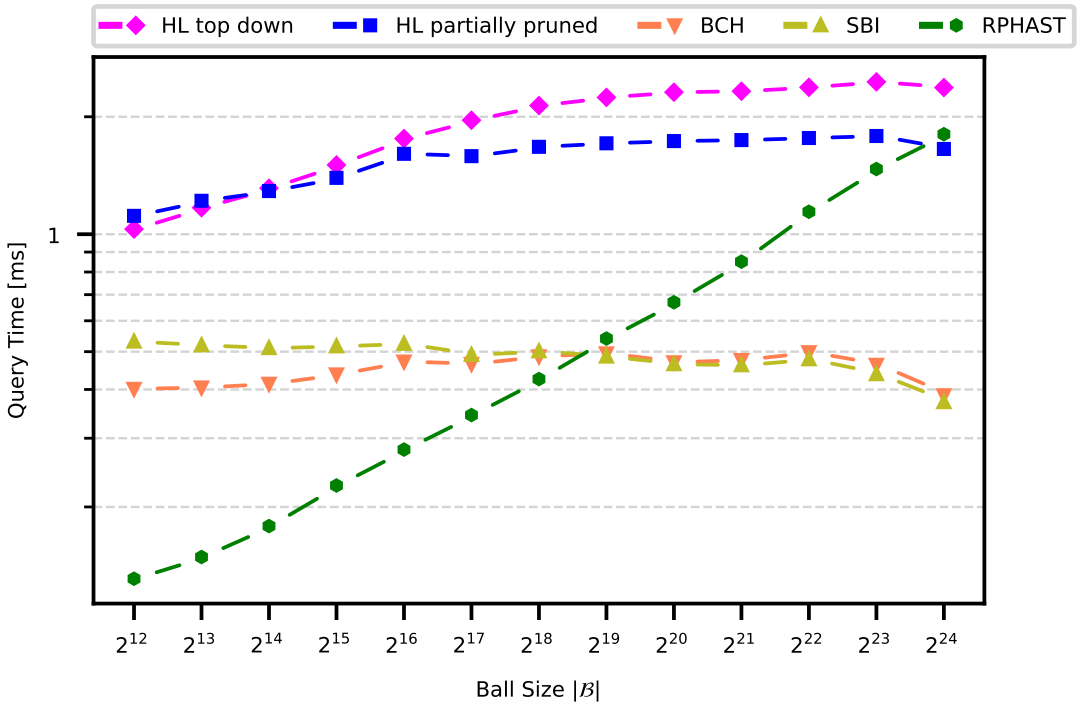


Figure E.14.: Query times of one-to-many algorithms with $|T| = 2^{12}$ targets picked from the a ball of varying size $|\mathcal{B}|$; Source picked from the same ball.

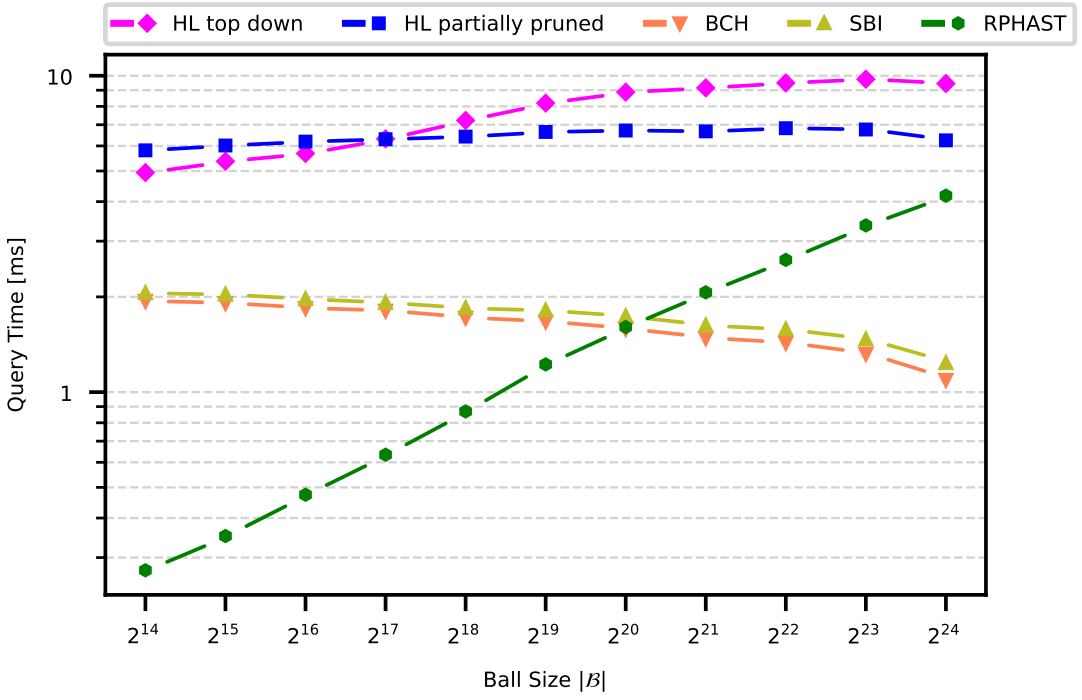


Figure E.15.: Query times of one-to-many algorithms with $|T| = 2^{14}$ targets picked from the a ball of varying size $|\mathcal{B}|$; Source picked from the same ball.

F. One-to-Many Selection+Query Times - Same Ball

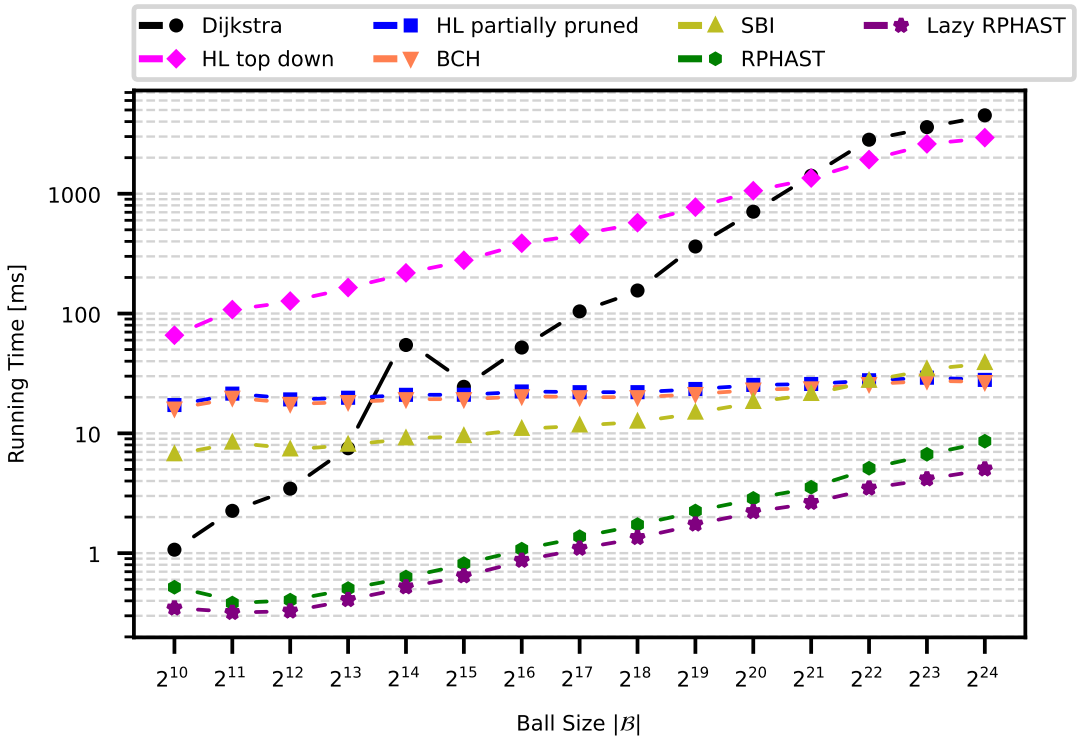


Figure F.16.: Running times of one-to-many algorithms with $|T| = 2^{10}$ targets picked from a ball of varying size $|\mathcal{B}|$; Source picked from the same ball.

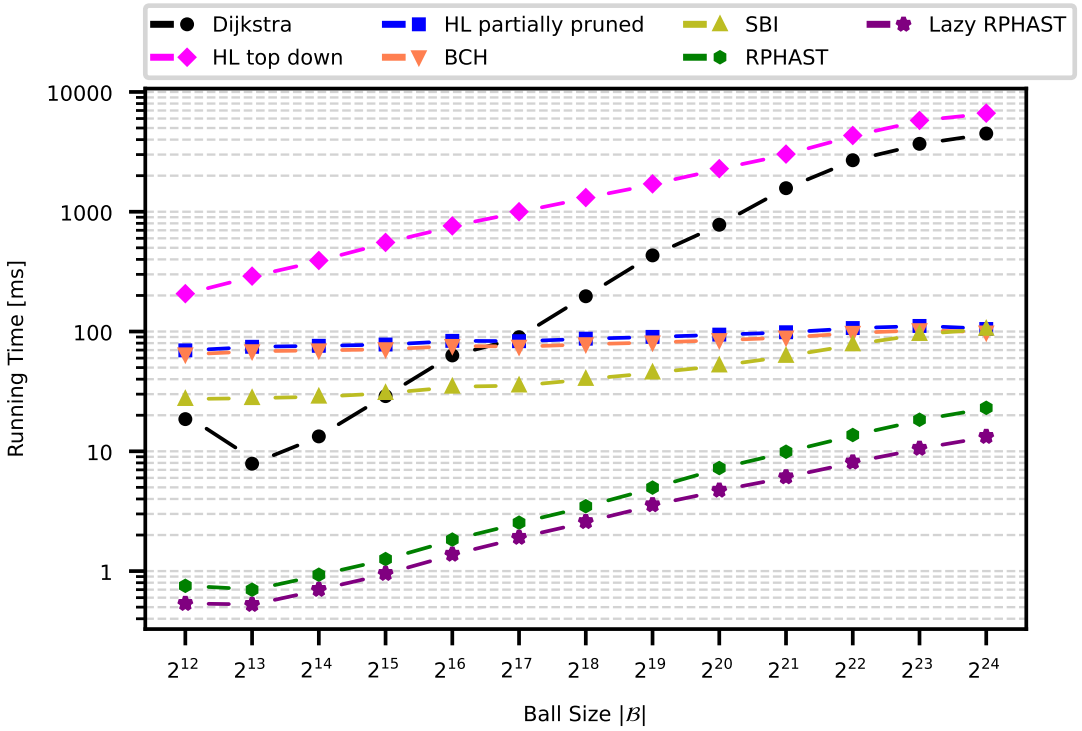


Figure F.17.: Running times of one-to-many algorithms with $|T| = 2^{12}$ targets picked from a ball of varying size $|\mathcal{B}|$; Source picked from the same ball.

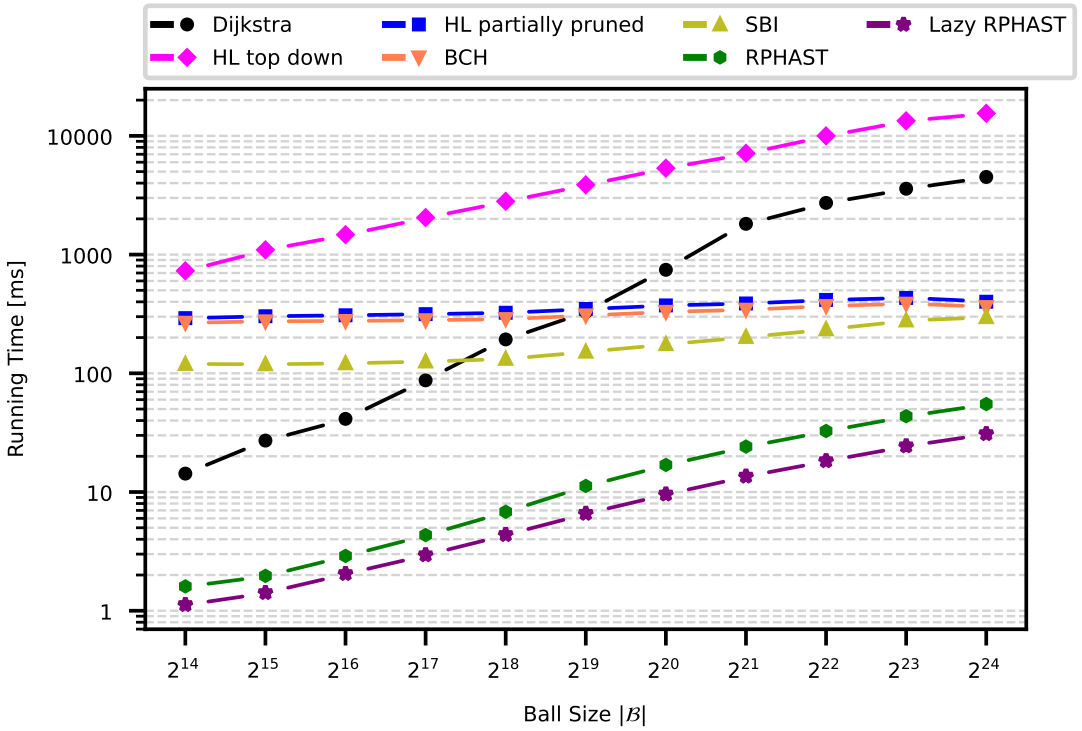


Figure F.18.: Running times of one-to-many algorithms with $|T| = 2^{14}$ targets picked from a ball of varying size $|\mathcal{B}|$; Source picked from the same ball.

G. Many-to-Many RPHAST Variants - Different Balls

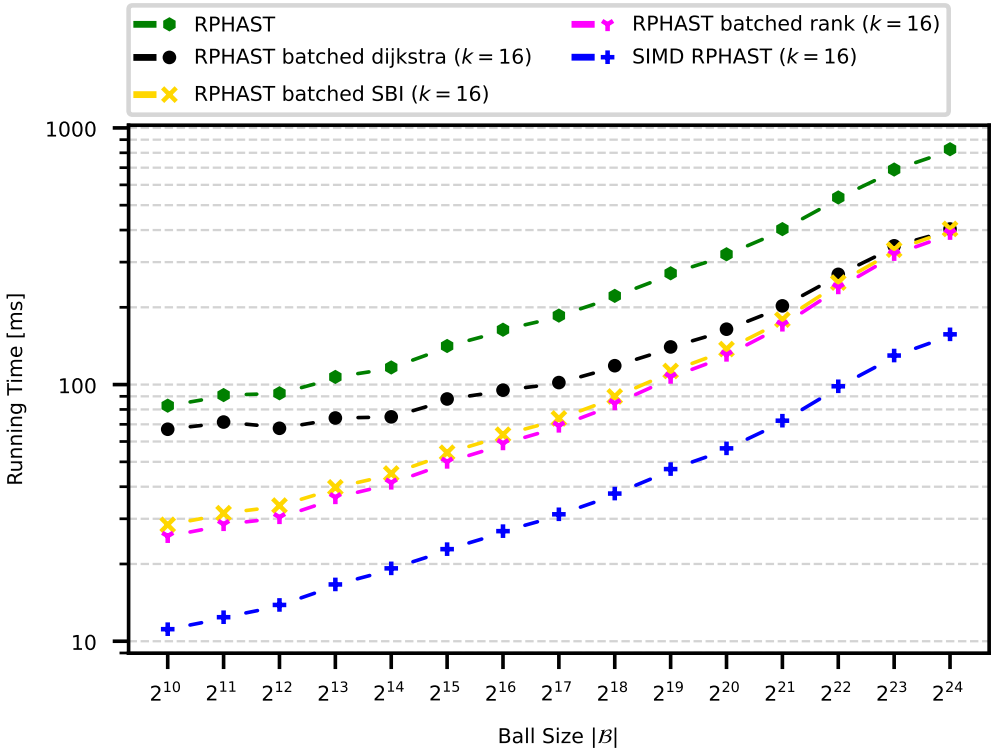


Figure G.19.: Running times of RPHAST algorithms with $|S| = |T| = 2^{10}$ sources and targets picked from different balls of varying size $|\mathcal{B}|$.

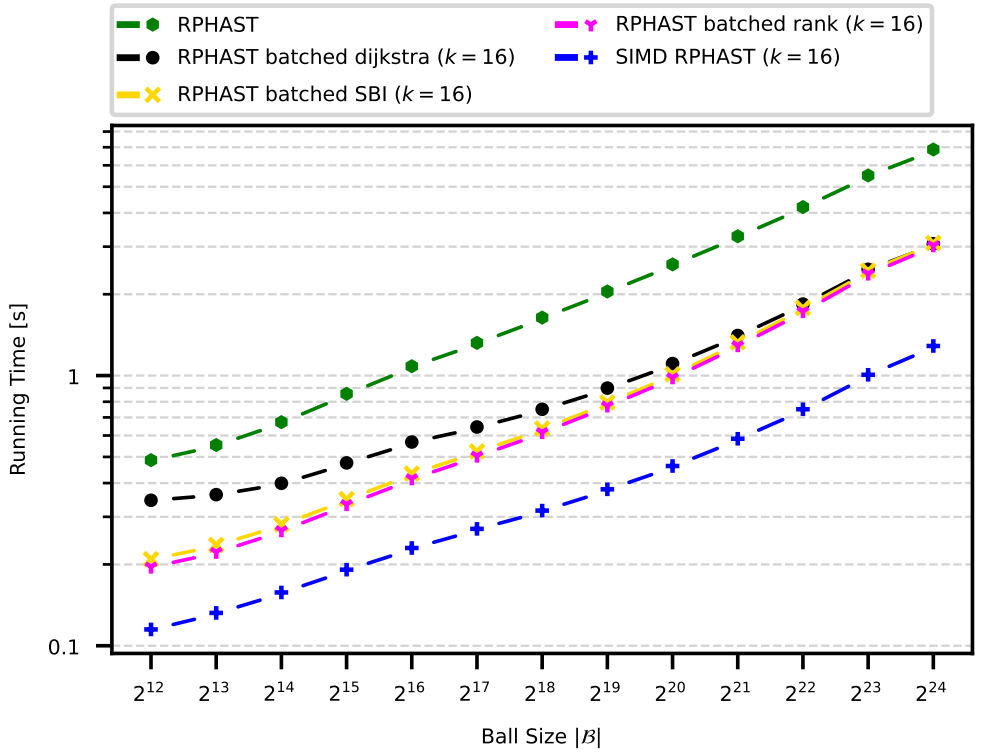


Figure G.20.: Running times of RPHAST algorithms with $|S| = |T| = 2^{12}$ sources and targets picked from different balls of varying size $|\mathcal{B}|$.

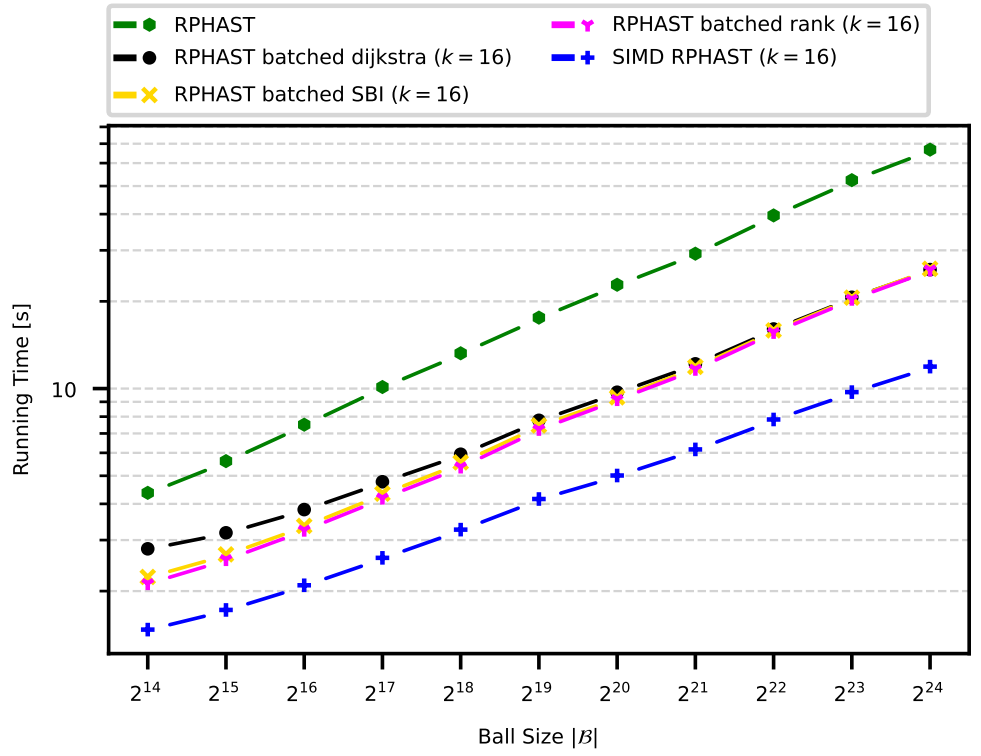


Figure G.21.: Running times of RPHAST algorithms with $|S| = |T| = 2^{14}$ sources and targets picked from different balls of varying size $|\mathcal{B}|$.

H. Many-to-Many HL Variants - Different Balls

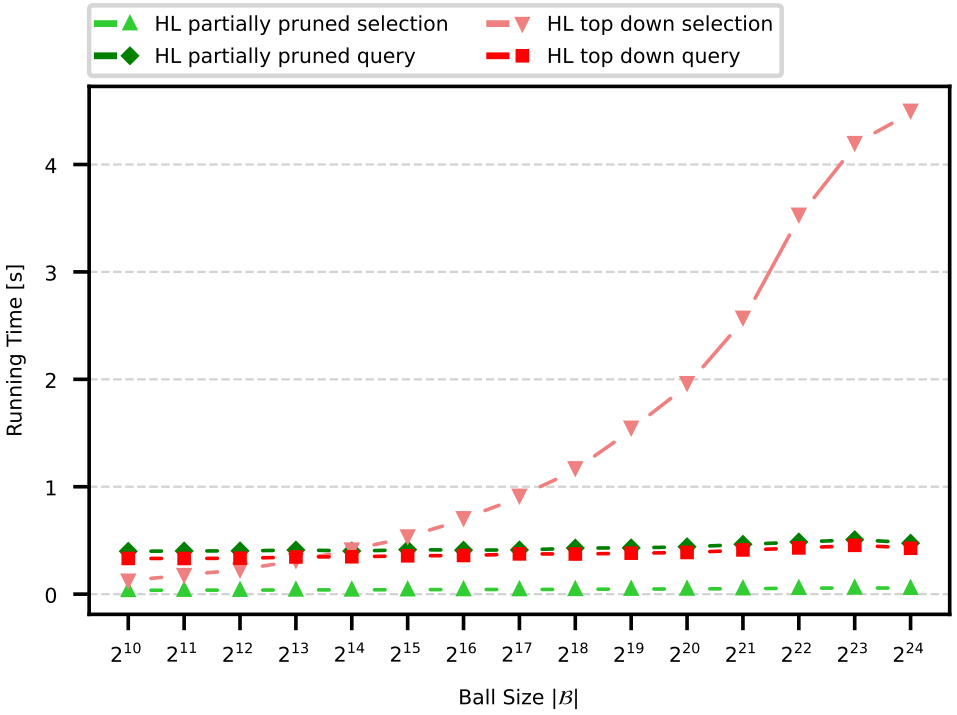


Figure H.22.: Running times of HL algorithms with $|S| = |T| = 2^{10}$ sources and targets picked from different balls of varying size $|\mathcal{B}|$.

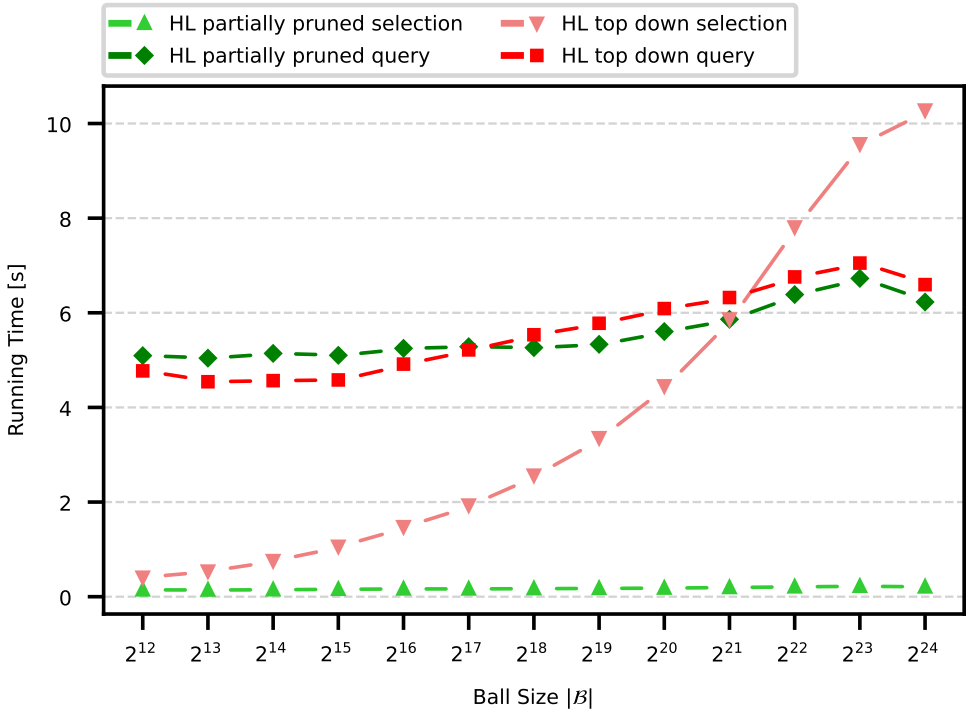


Figure H.23.: Running times of HL algorithms with $|S| = |T| = 2^{12}$ sources and targets picked from different balls of varying size $|\mathcal{B}|$.

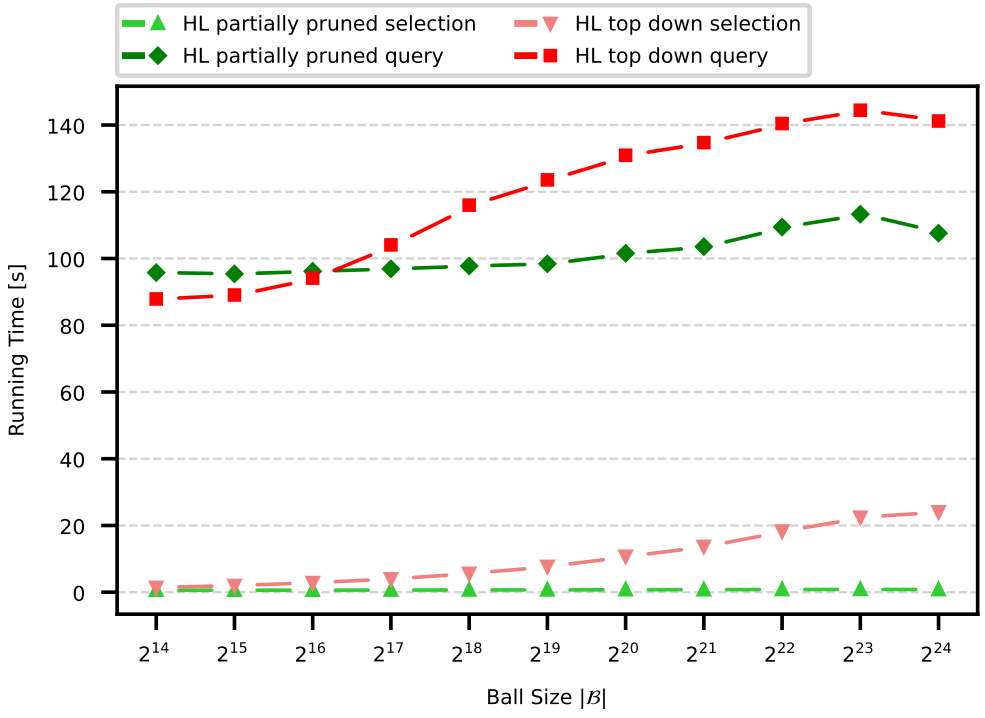


Figure H.24.: Running times of HL algorithms with $|S| = |T| = 2^{14}$ sources and targets picked from different balls of varying size $|\mathcal{B}|$.

I. Many-to-Many Bucket Variants - Different Balls

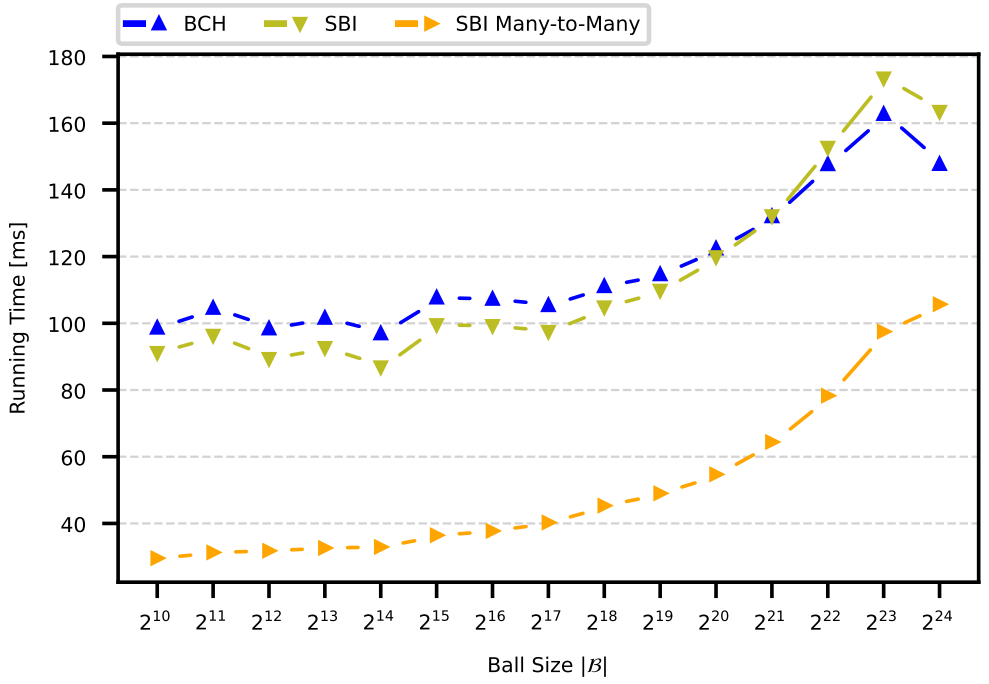


Figure I.25.: Running times of bucket-based algorithms with $|S| = |T| = 2^{10}$ sources and targets picked from different balls of varying size $|\mathcal{B}|$.

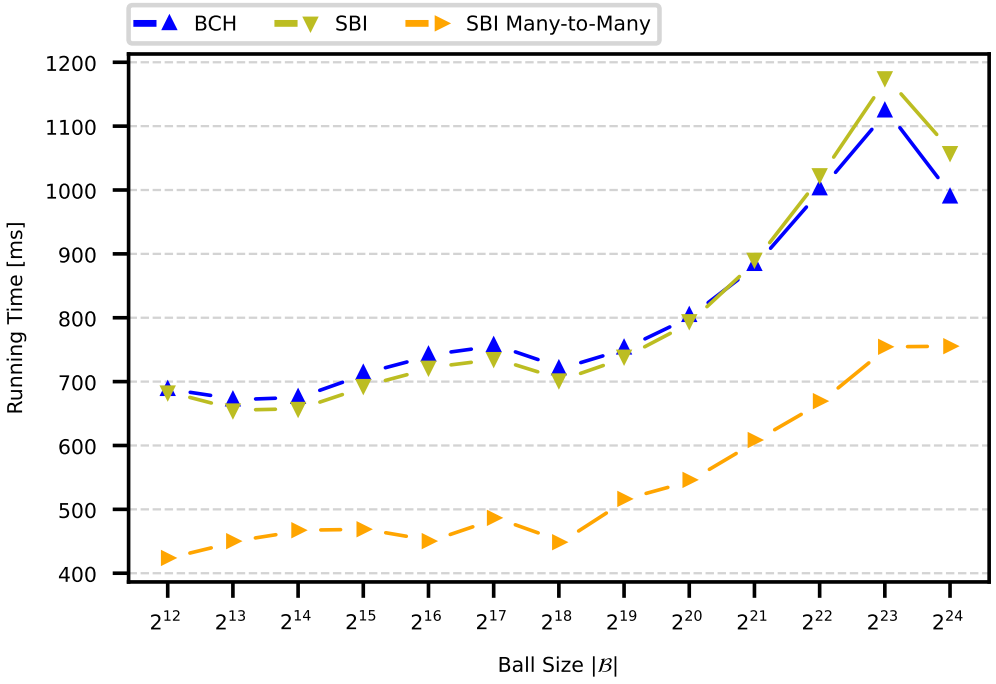


Figure I.26.: Running times of bucket-based algorithms with $|S| = |T| = 2^{12}$ sources and targets picked from different balls of varying size $|\mathcal{B}|$.

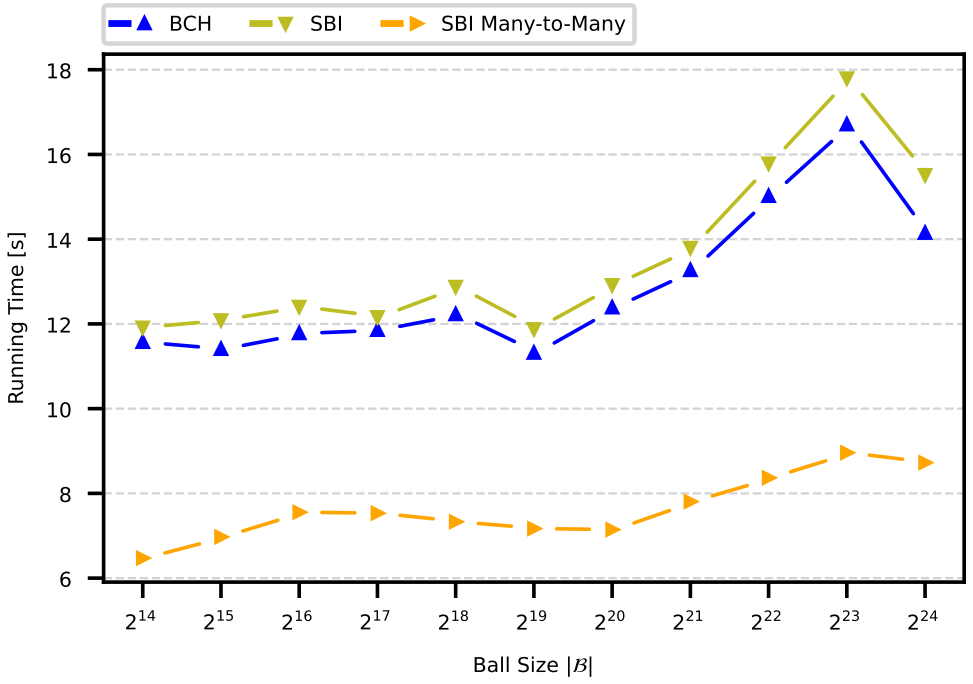


Figure I.27.: Running times of bucket-based algorithms with $|S| = |T| = 2^{14}$ sources and targets picked from different balls of varying size $|\mathcal{B}|$.

J. Many-to-Many Asymmetric - Different Balls

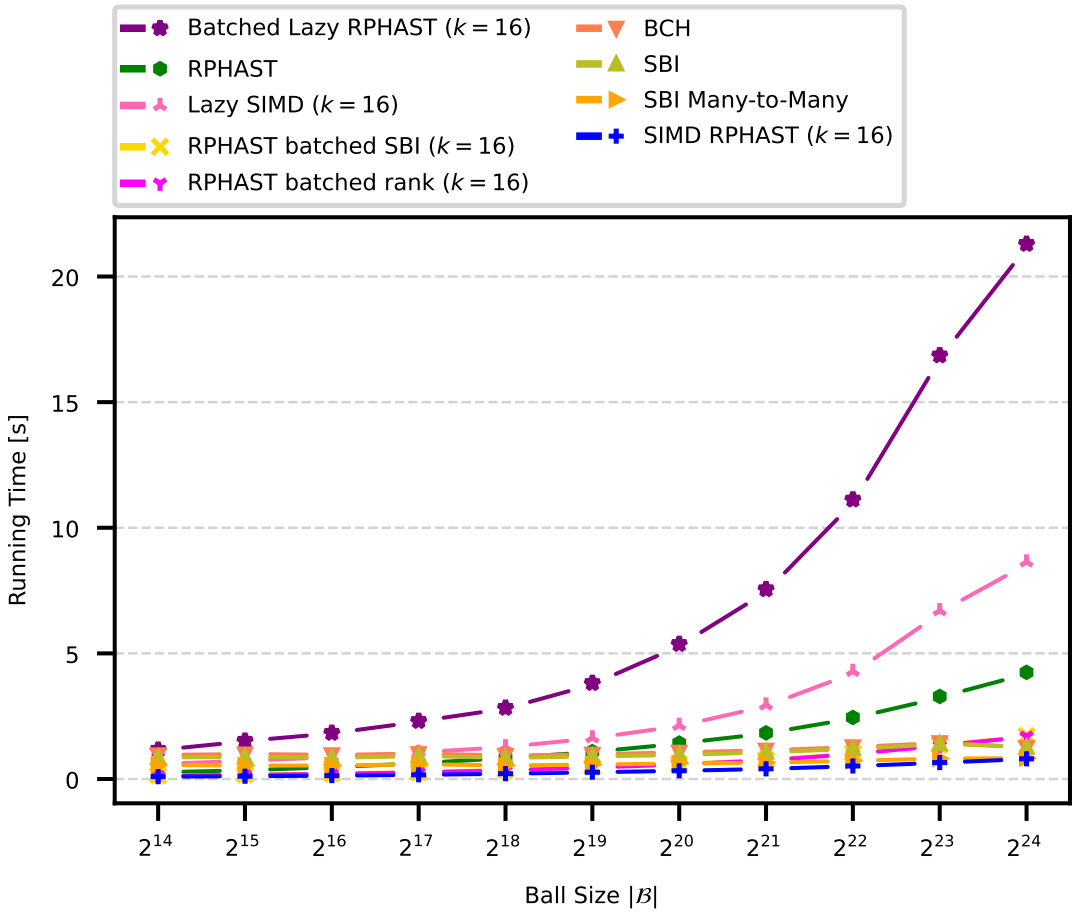


Figure J.28.: Running times of many-to-many algorithms with $|S| = 2^{10}$ sources and $|T| = 2^{14}$ targets picked from different balls of varying size $|\mathcal{B}|$.

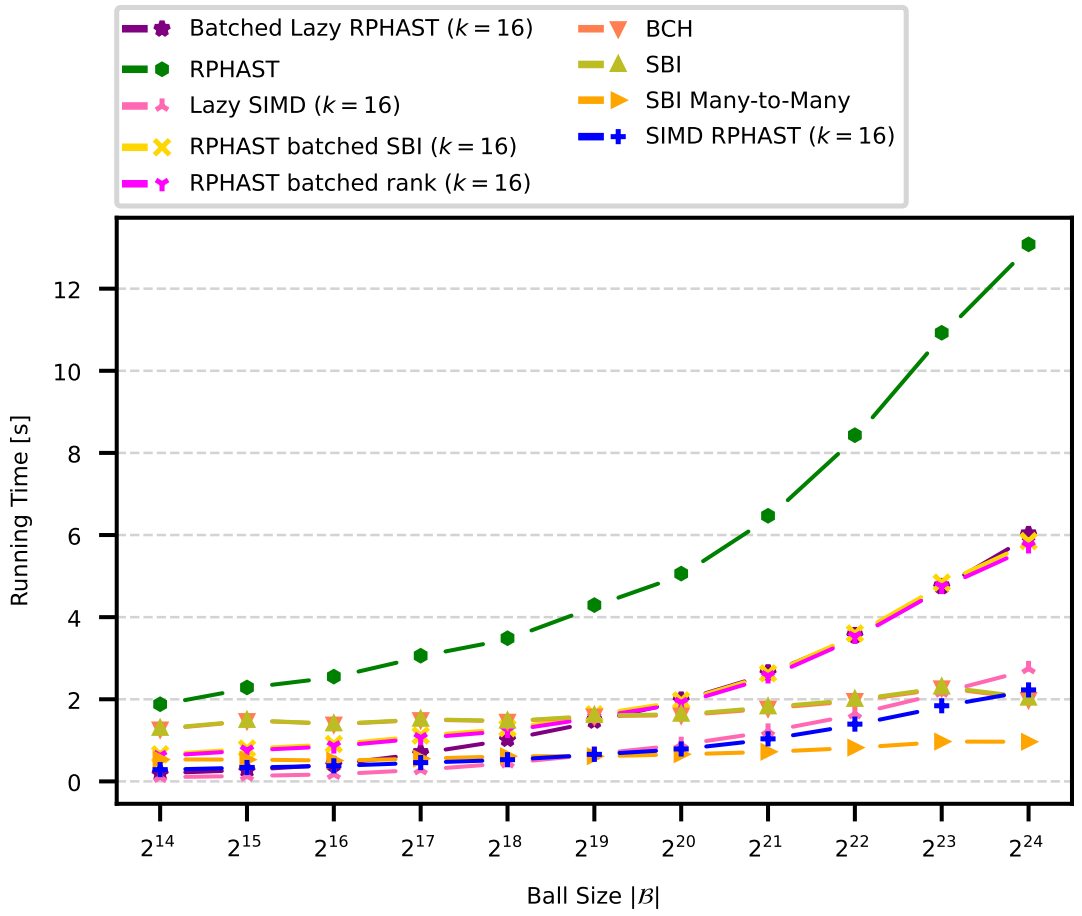


Figure J.29.: Running times of many-to-many algorithms with $|S| = 2^{14}$ sources and $|T| = 2^{10}$ targets picked from different balls of varying size $|\mathcal{B}|$.

K. Many-to-Many RPHAST batched SBI - Batch Size - Different Balls

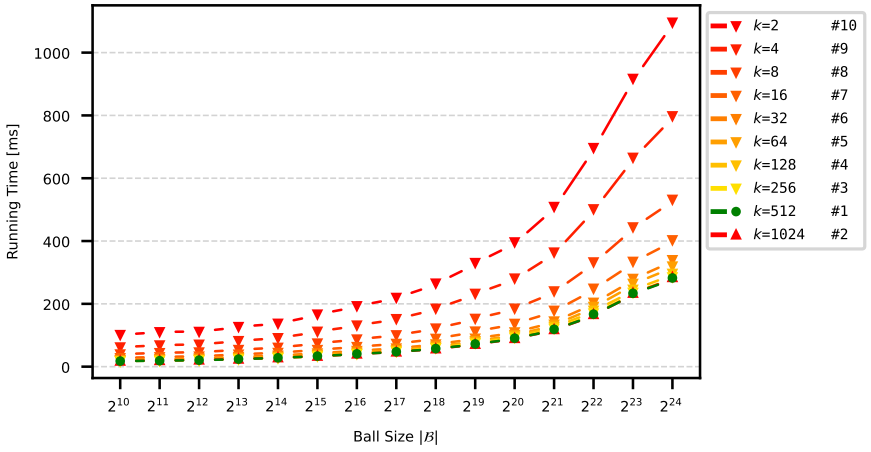


Figure K.30.: Running times of RPHAST batched SBI with different batch sizes $k \in \{2^x \mid x \in (1, \ldots, 10)\}$; $|S| = |T| = 2^{10}$ sources and targets picked from different balls of varying size $|\mathcal{B}|$. We rank all batch sizes k based on their average performance across all ball sizes $|\mathcal{B}|$. *#1* is associated with the batch size k for which all distances are calculated the most quickly.

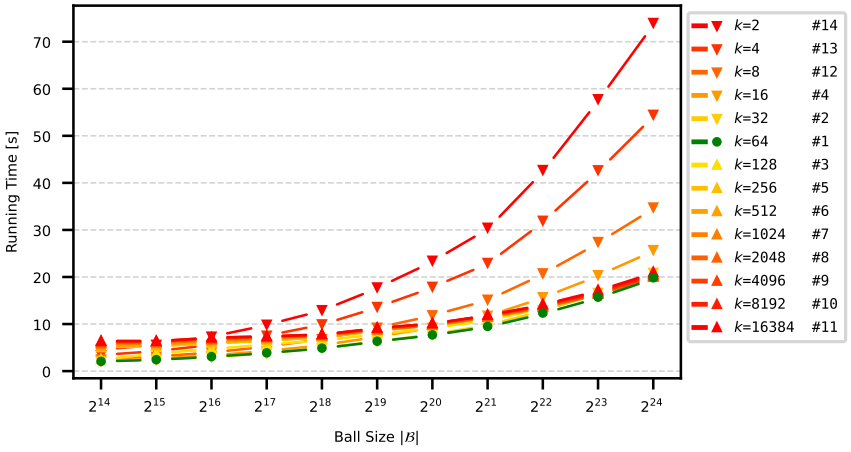


Figure K.31.: Running times of RPHAST batched SBI with different batch sizes $k \in \{2^x \mid x \in (1, \ldots, 14)\}$; $|S| = |T| = 2^{14}$ sources and targets picked from different balls of varying size $|\mathcal{B}|$. We rank all batch sizes k based on their average performance across all ball sizes $|\mathcal{B}|$. *#1* is associated with the batch size k for which all distances are calculated the most quickly.

L. Many-to-Many RPHAST batched rank - Batch Size - Different Balls

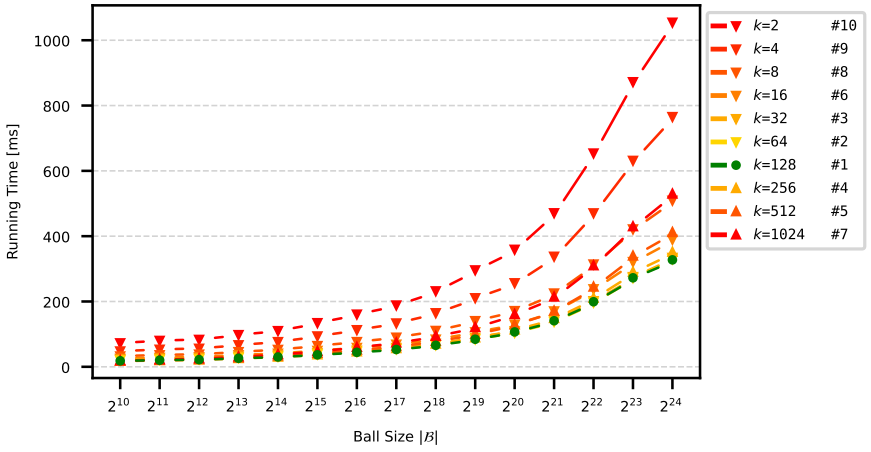


Figure L.32.: Running times of RPHAST batched rank with different batch sizes $k \in \{2^x \mid x \in (1, \ldots, 10)\}$; $|S| = |T| = 2^{10}$ sources and targets picked from different balls of varying size $|\mathcal{B}|$. We rank all batch sizes k based on their average performance across all ball sizes $|\mathcal{B}|$. *#1* is associated with the batch size k for which all distances are calculated the most quickly.

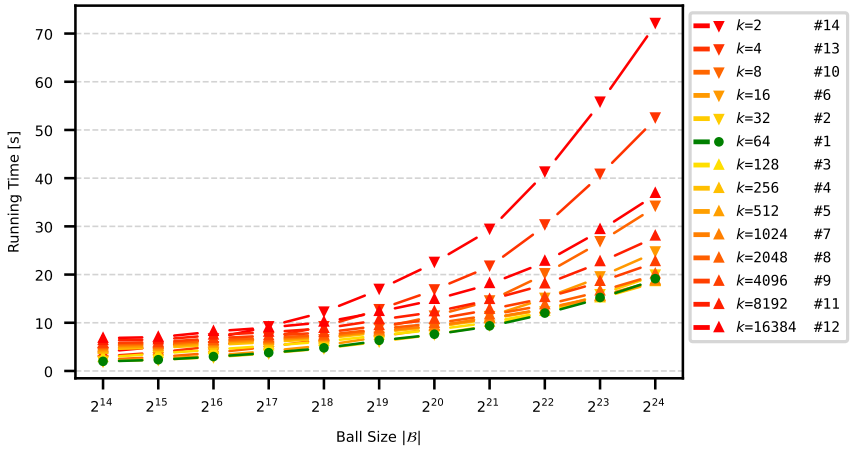


Figure L.33.: Running times of RPHAST batched rank with different batch sizes $k \in \{2^x \mid x \in (1, \ldots, 14)\}$; $|S| = |T| = 2^{14}$ sources and targets picked from different balls of varying size $|\mathcal{B}|$. We rank all batch sizes k based on their average performance across all ball sizes $|\mathcal{B}|$. *#1* is associated with the batch size k for which all distances are calculated the most quickly.

M. Many-to-Many Batched Lazy RPHAST - Batch Size - Different Balls

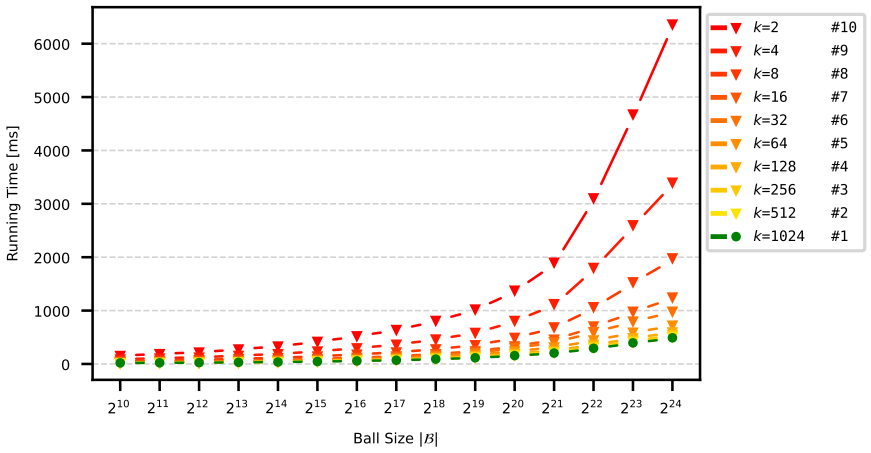


Figure M.34.: Running times of Batched Lazy RPHAST with different batch sizes $k \in \{2^x \mid x \in (1, \ldots, 10)\}$; $|S| = |T| = 2^{10}$ sources and targets picked from different balls of varying size $|\mathcal{B}|$. We rank all batch sizes k based on their average performance across all ball sizes $|\mathcal{B}|$. *#1* is associated with the batch size k for which all distances are calculated the most quickly.

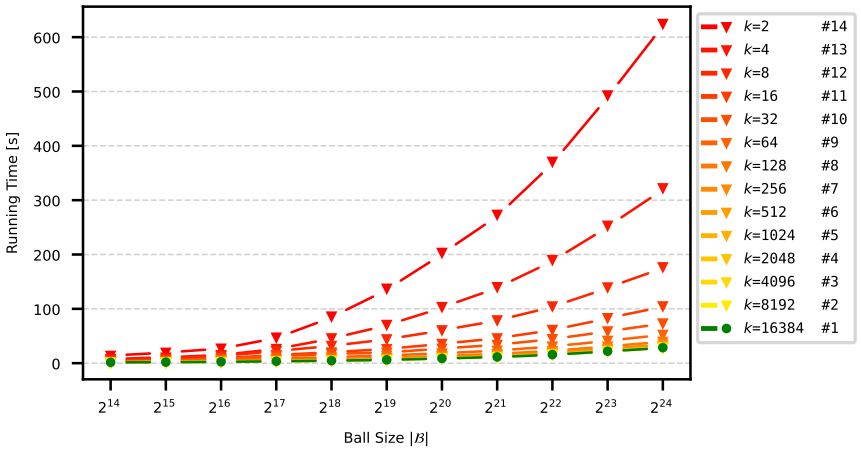


Figure M.35.: Running times of Batched Lazy RPHAST with different batch sizes $k \in \{2^x \mid x \in (1, \ldots, 14)\}$; $|S| = |T| = 2^{14}$ sources and targets picked from different balls of varying size $|\mathcal{B}|$. We rank all batch sizes k based on their average performance across all ball sizes $|\mathcal{B}|$. *#1* is associated with the batch size k for which all distances are calculated the most quickly.