On Fully Dynamic All Pairs Shortest Path in Sparse Digraphs

Lennart Küssner

LUND
UNIVERSITY

Department of Automatic Control

MSc Thesis
TFRT-6275
ISSN 0280-5316

Department of Automatic Control
Lund University
Box 118
SE-221 00 LUND
Sweden

© 2025 Lennart Küssner. All rights reserved.
Printed in Sweden by Tryckeriet i E-huset
Lund 2025

Acknowledgements

A great appreciation goes out to Maximilian Probst Gutenberg and Tianyi Zhang for the **amazing** help you have given as supervisors. Thank you for always pushing forward, for proof reading and criticizing and for always having another idea that we could try out.

I would also like to express my gratitude toward Giacomo Como and Emma Tegling for going out of control to supervise and examine, making it at all possible for me to write a thesis on this topic.

A big thank you also to Josefine, Botan-Erik and Linn for putting up with me, and for kicking me when I was being defeatist.

In a grander scale of things, this would not have been possible without my dear parents who have always shown fierce support, even when I became the black sheep of the family by not going into medicine.

I also cannot thank Andreas Gunnarsson enough for helping my mathematical interests unfold.

Abstract

The *All Pairs Shortest Path* (APSP) problem is ubiquitous in computer science. The problem consists of finding for any vertex pair, the weighted shortest paths in a network with real-weighted edges. The problem has been studied thoroughly, both for dense graphs, where the number of edges can be quadratic in the number of vertices, and for sparse graphs, where the number of edges is considerably lower than so.

Many real world networks are subject to small changes over time. For instance, road work or traffic accidents can considerably change the shortest paths through a road network. As most real world networks are also sparse, this motivates the study of the sparse fully dynamic APSP problem, where a data stucture is maintained for a sparse graph subject to vertex insertions and deletions, from which the shortest path between any pair of vertices should be quickly queryable.

This thesis comes with two main results:
Firstly, the key methods that led to the improvement in the dense case have been studied to determine whether they can be applied in the sparse case. Conclusively it has been shown that the techniques studied rely on subroutines that iterate through all vertex pairs, with no real possibility to change this. Therefore, they cannot be transferred to the sparse case to give an algorithm with update time subquadratic in the number of vertices. Given this, no improvement over the state of the art in the sparse case is possible using the new dense methods.

Secondly, a novel algorithm with a trade-off between the update and query time of $\tilde{O}(mn^2/t^{3/2})$, $\tilde{O}(t)$ for $t \in [n^{2/3}, n^{8/9}]$ is presented, making it possible to reach an update time of $\tilde{O}(mn^{2/3})$, improving over the state of the art.

Contents

1. Introduction **9**

2. Preliminaries **13**

2.1 Terminology and Notation 13
2.2 The Problem Statement 16
2.3 Basic Principles of APSP 16
2.4 Basic Principles of Dynamic APSP 19
2.5 The Sparse Domain 27

3. Previous Work **29**

3.1 Karczmarz and Sankowski 2023 29
3.2 Mao 2024 31

4. Unsuccessful Application of Dense Techniques to the Sparse Setting **37**

4.1 ACK-Repair vs GWN-Repair 38
4.2 Source of Issues: Via-Points 38
4.3 Layered Congested Sets 39
4.4 Repair of Hop Dominant Paths 40
4.5 Further Analysis of `RestrictedDijkstra` 43

5. Novel Algorithm with Improved Update-Query Time Trade-Off **47**

5.1 Introduction 47
5.2 Technical Overview 47
5.3 Low Hop Paths 48
5.4 Mid Hop Paths 53
5.5 High Hop Paths 55
5.6 Vertex Insertions 55
5.7 Negative Edge Weights 56
5.8 Space Complexity 56
5.9 Total Time Complexity And Parameter Balancing 56
5.10 Getting Past Graph Size Upper Bounds 57

6. Ranges of Optimality in Practical Use Cases **59**

7. Discussion and Conclusions **61**

Bibliography **63**

1

Introduction

The all pairs shortest path (APSP) problem is a fundamental problem taught in undergraduate algorithmic courses, of intuitive theoretical importance as well as real world uses. In a graph $G = (V, E, W)$ the task is to find all weighted shortest distances and paths. Classical algorithms include Dijkstra's algorithm [Dijkstra, 1959] for single source shortest paths with time complexity $\tilde{O}(|E|)$[1], the Bellman-Ford algorithm [Kociumaka and Polak, 2023] with time complexity $O(|V||E|)$ and Johnson's algorithm [Johnson, 1977] with time complexity $\tilde{O}(|V||E|)$.

Many real world networks are not static in nature, but change over time. For example, roads are built and closed off, undersea internet cables are damaged by anchors. After such a small change in the network one naturally wants to avoid having to recompute the shortest paths from scratch, which in some contexts could have infeasibly high runtime. In the dynamic setting of this algorithmic problem, a data structure based on the graph is used to help decrease the time necessary to update distances after the graph has been altered. Computer scientists have studied the dynamic APSP problem for almost 50 years, starting with an algorithm allowing for vertex insertions by Johnson [1977], with the first fully dynamic algorithm produced by King [1999].

For a weighted directed graph subject to vertex deletions/insertions, the aim is to find an algorithm and a data structure that allows for queries of arbitrary distances and shortest paths. The performance of the algorithm is evaluated by the update time, which is the time necessary to find the new distance matrix and shortest paths after a vertex insertion/deletion, and the space necessary for the data structure, both expressed as functions of the size of the vertex set n.

In 2004, Demetrescu and Italiano revolutionized the field with an algorithm handling general vertex updates with a close to optimal amortized time complexity of

[1] $\tilde{O}(\cdot)$ hides polylogarithmic factors while the $O(\cdot)$ used later hides only constant factors. For a more in-depth explanation see section 2.1.

$\tilde{O}(n^2)$ and space complexity of $O(n^3)$. In the worst case this algorithm can however take $O(n^3)$ time for an update. There has however been progress on worst case update times since 2004. Thorup [2005] devised the first algorithm with sub-cubic worst case update time, a deterministic algorithm with time complexity $\tilde{O}(n^{2+3/4})$. Abraham et al. [2017] improved this to a $\tilde{O}(n^{2+2/3})$ update time using randomization, but still with $O(n^3)$ space.

Gutenberg and Wulff-Nilsen [2019] improve deterministic time complexity to $\tilde{O}(n^{2+5/7})$, and perhaps more importantly the space complexity to $\tilde{O}(n^2)$, both for the deterministic algorithm and for a Las Vegas randomized algorithm matching the speed of [Abraham et al., 2017]. Chechik and Zhang [2023] improved the deterministic time complexity further to $\tilde{O}(n^{2+41/61})$. At this point, the state of the art is [Mao, 2024], which takes the randomized time complexity down to $\tilde{O}(n^{2+1/2})$. Mao also argues that further polynomial improvement may be severely difficult.

However, many real-world dynamic graphs are sparse, meaning that the number of edges, $m = |E|$, is much closer to n than n^2. Using Johnson's algorithm gives trivial update speed $\tilde{O}(nm)$ with query speed $O(1)$, while querying with Dijkstra's algorithm gives no update time at all with query time $\tilde{O}(m)$. Therefore any relevant algorithm would have update and query time less than $\tilde{O}(nm)$ and $\tilde{O}(m)$ respectively. For any sparse graph (with $m < n^{3/2}$) the static Johnson's algorithm is more efficient than even the algorithm by Mao [2024]. In the sparse case even just updating the distance matrix takes too long time, as a time complexity of $O(n^2)$ is not necessarily a polynomial improvement over $\tilde{O}(nm)$. Therefore, in the sparse setting there is no distance matrix that is being maintained. There is instead a query procedure that extracts the shortest path from the data structure. The time complexity of this query procedure, called the query time, next to the update time becomes important in evaluating a sparse fully dynamic APSP algorithm.

Karczmarz and Sankowski [2023] developed the first algorithm for the sparse dynamic context, with a update time complexity of $mn^{4/5}$ and a query time complexity of $n^{4/5}$. Karczmarz and Sankowski [2023] also manage to construct an algorithm that is flexible in the sense that one can trade an increased query time $\tilde{O}(t)$ against an decreased update time $\tilde{O}(mn/t^{1/4})$. The space complexity can be bounded with $\tilde{O}(n^2)$, using the technique developed by Gutenberg and Wulff-Nilsen [2019]. To this point in time, [Karczmarz and Sankowski, 2023] is the only result improving over triviality in the sparse case. This thesis will study the fully dynamic directed APSP problem in a sparse setting, and attempt to improve upon the result by Karczmarz and Sankowski [2023].

This thesis presents a novel algorithm refining the methods of [Karczmarz and Sankowski, 2023], with the following result:

Let $G = (V, E, W)$, be a real-weighted directed graph with $|V|$ upper-bounded by n and $|E|$ upper-bounded by m. There exists a randomized data structure maintaining G under fully dynamic vertex updates and answering arbitrary pairwise distance queries with $\tilde{O}(mn^2/t^{3/2} + mn^{2/3})$ worst-case update time and $\tilde{O}(t + n^{2/3})$ query time with $t \leq n^{8/9}$ and using $\tilde{O}(n^2)$ space. The queries are supported only when G has no negative cycles. After answering a distance query, some corresponding shortest path π can be reported in $O(|\pi|)$ time. Queries are answered correctly with high probability, and time complexities are also true with high probability.

This poses an improvement over the trade-off presented in [Karczmarz and Sankowski, 2023], managing to attain polynomially lower update time for query times in the range $(n^{4/5}, n^{8/9}]$, as illustrated in figure 1.1.

This thesis also presents convincing arguments for that the techniques used in [Mao, 2024] are not suitable for the sparse version of the dynamic APSP problem. This, as they all require $O(n^2)$ subroutines to be run in every update. In very sparse graphs, where $m = \tilde{O}(n)$ this is not an improvement over even the static recomputation of APSP using Dijkstra's algorithm.

The remainder of the thesis is organized as follows. Chapter 2 introduces theory and notation. Chapter 3 gives an overview of [Karczmarz and Sankowski, 2023] and [Mao, 2024]. Chapter 4 explains why the techniques used in [Mao, 2024] are not compatible with sparsity. Chapter 5 goes into the technical details of the novel algorithm with improved update-query time tradeoff. Chapter 6 compares asymptotic time complexity for several dynamic APSP algorithms under varying graph densities and query loads. Chapter 7 concludes the thesis with a discussion of the results and further research directions.

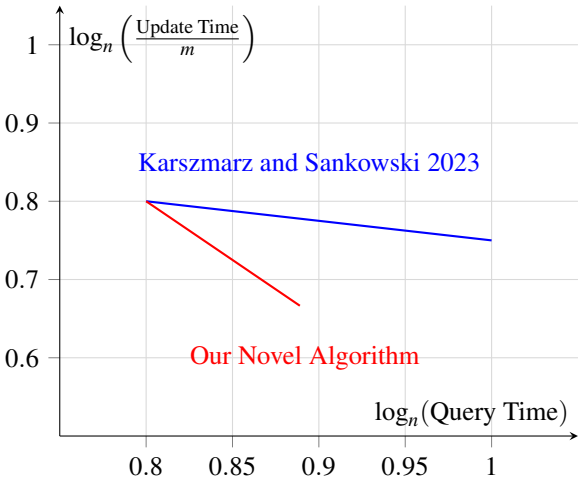

Figure 1.1 Comparison of trade-offs between update time (on the vertical axis) and query time on the horizontal axis. The graph shows the asymptotic behavior plotting for update times the exponent y in $\tilde{O}(mn^y)$ and for query times the exponent x in $\tilde{O}(n^x)$, ignoring polylog factors. The upper blue line features the trade-off possible in [Karczmarz and Sankowski, 2023], while the superior trade-off found in this thesis is shown in the lower red line.

2

Preliminaries

2.1 Terminology and Notation

Below follows an explanation of the graph theoretical and algorithmic terminology needed to understand this thesis.

Basic Graph Theory

A weighted directed graph $G = (V, E, W)$ is defined by its vertex set V, its edge set $E \subset V \times V$, and its edge weights which are a function $W : E \to \mathbb{R}$. In a directed graph edges are ordered pairs, and $(u, v) \in E$ does not necessarily mean $(v, u) \in E$. Furthermore the graphs studied will be without self-loops, so $(u, u) \notin E$.

First we study an unweighted graph. The example graph in figure 2.1 has vertex set $V = \{a, b, c, d\}$, and edge set
$E = \{(a, b), (b, d), (a, c), (c, b), (c, d), (d, c)\}$

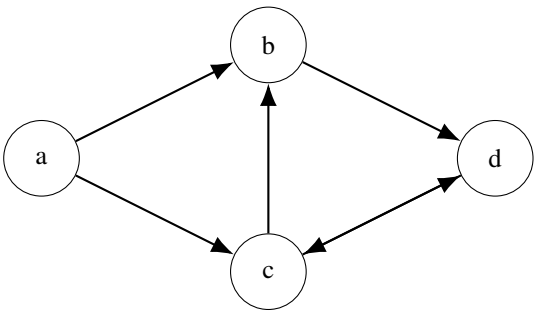

Figure 2.1 Unweighted example graph.

The degree of a vertex is the number of edges intersecting that vertex. In directed graphs we distinguish between in-degree and out-degree. b for example has in-degree two and out-degree one in the example graph. As every edge increases the

in/out degree of some vertex by one, $\sum_{v \in V} \text{in-degree}(v) = \sum_{v \in V} \text{out-degree}(v) = |E|$.

A **walk** is a sequence of vertices $u_1, u_2 ... u_{k-1}, u_k$ s.t $\forall i \in [k-1], (u_i, u_{i+1}) \in E$. In the example graph a, b, d, c, b would be a walk, while a, b, c would not be a walk as $(b, c) \notin E$.

If all vertices of a walk are unique it is called a **path**. In the example graph a, b, d would be a path. a, b, d, c, b is a walk, but not a path as b is recurrent. We can concatenate two paths if the endpoint of the first is the same as the starting point of the second, and denote by $\pi_1 \circ \pi_2$ the concatenation of the paths p_1 and p_2.

A **cycle** is a path $u_1, u_2, \ldots u_i$ so that $(u_i, u_1) \in E$. b, d, c is a cycle in the example graph.

For $V' \subset V$ the **induced subgraph** to $G = (V, E)$ is defined as $G[V'] = (V', \{(u, v) \in E \text{ so that } u, v \in V'\})$. Though a bit sloppy, the notation $G \setminus D$ will be used synonymously to $G[V \setminus D]$ in this thesis.

The graphs studied in this thesis will further be weighted, in that edges have associated weights $w(e)$ for $e \in E$. Weights can generally be any real number. The **length** of a path in a weighted graph is the sum of the edge-weights of the edges contained in the path. In the example graph of figure 2.2 the path a, c, b has length $2 + (-5) = -3$.

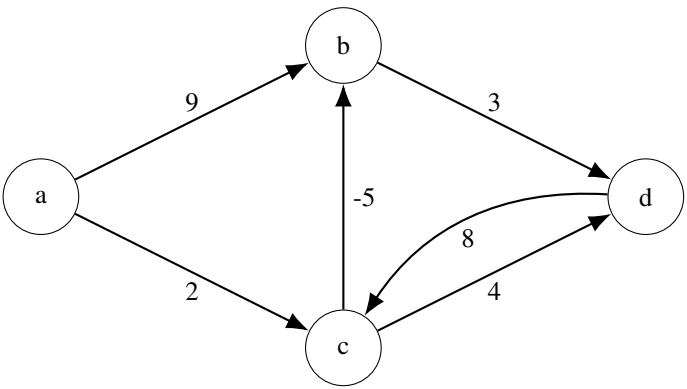

Figure 2.2 Weighted example graph.

While this thesis concerns itself with shortest weighted path lengths, from an algorithmic point of view the number of edges on a path are also important. We call this the **hop** of a path. The hop of a path is trivially one less than the number of vertices

on the path. The path a,b,d has hop 2, and contains 3 vertices. When studying a path π, $|\pi|$ will refer to the hop, not the number of vertices.

For a graph G, we define $\overleftarrow{G}$ as a graph on the same vertex set, but having edges in the opposite directions. This means $(u,v) \in E(G) \Leftrightarrow (v,u) \in E(\overleftarrow{G})$. This reverse graph is used to repurpose algorithms for finding paths starting at a vertex v to instead find paths ending at v.

Time Complexity and Big-O-Notation

An algorithm is a list of instructions that solve some problem. Running the algorithm requires computations to be made. The smallest such computations (addition, subtraction, adding pointers, etc) are called basic operations. When an algorithm terminates after fewer basic operations than $Cf(n,m)$, where C is constant and n,m are the size of the problem instance, we say that it has time complexity $O(f(n,m))$. When instead the bound is $C\log^x(n)\log^y(m)f(n,m)$ where x,y and C are constant, we say that the algorithm has time complexity $\tilde{O}(f(n,m))$.

When studying the time complexity of algorithms whose input are graphs, the size of the graph is of utmost importance because termination will usually require more operations in a larger graph. Both the number of vertices $n = |V|$ and the number of edges $m = |E|$ can be important for the complexity of the algorithm. One trivial observation is that as $E \subset V \times V$, $m \leq n^2$. This is an asymptotically tight bound as for a directed graph without self loops m can maximally be $n^2 - n$.

In the dynamic setting it becomes complicated to talk about n and m, as the sizes of vertex set and edge set can both grow and shrink. The simplest solution is to say that n and m upper-bound the sizes of vertex set and edge set respectively. This way, as time complexities grow with graph size, n and m can always be used to bound the time complexity. If one wants to use the vertex set and edge set sizes after each update as the parameter used in the time complexity, one has to handle dramatic growth or shrinkage in the graph.

Randomized Algorithms

A widely used tool in algorithm design is randomness. There are two main classes of randomized algorithms:

- **Monte Carlo Algorithms**: Which provide false results with a very low probability.
- **Las Vegas Algorithms**: Which are guaranteed to provide correct results, but might break their time complexity bounds with a low probability.

A standard concept when working with randomized algorithms is the terminology "With High Probability" (abbreviated **whp**), which means that some event happens

with a probability of at least $1 - n^{-\alpha}$, where n is the problem input size and α is a positive constant. With problem instances of increasing size this leads to almost guaranteed success.

2.2 The Problem Statement

For a weighted directed graph G subject to fully dynamic vertex updates, find an algorithm that allows for queries of (s,t) distances and minimum length (s,t) paths for any $s,t \in V$.

In the fully dynamic setting both vertex insertions and deletions are allowed. A vertex insertion means adding an additional vertex v to V, and adding to the edge set an arbitrary subset of $\{v\} \times (V \setminus \{v\}) \cup (V \setminus \{v\}) \times \{v\}$. These edges can have arbitrary real weights. A vertex deletion means removing a vertex from V and all intersecting edges from E.

The problem situation starts with an adversary providing a graph $G = (V,E,W)$, and parameters n,m, where n is the upper bound for $|V|$, and m is the upper bound for $|E|$. After this the algorithm may perform some preprocessing of the graph. After this, an arbitrary number of rounds unfold. In every round the adversary does a vertex update, upon which the algorithm may run an update procedure. After termination of the update procedure, the adversary may ask an arbitrary number of queries of the form (s,t). After each query the algorithm may run a query procedure, finding the shortest (s,t)-distance, and a shortest (s,t)-path.

Performance is measured in 3 ways:

- Update time complexity - The time it takes for the data structure to be queriable after insertion/deletion of a vertex.
- Query time complexity - The time it takes given a query (s,t) to find the length of the shortest (s,t)- path. Outputting the shortest path $\pi_{s,t}$ after that is expected to take time $O(|\pi_{s,t}|)$.
- The space complexity of the necessary data structure.

All three performance measures are to be given as functions of n and m. Oftentimes time- and space complexity is most practically given in big-O-notation.

2.3 Basic Principles of APSP

Before exploring the more complex domain of **dynamic** APSP it is beneficial for the reader to be comfortable with the standard algorithms for **static** APSP.

Dijkstra's Algorithm

Dijkstra's algorithm finds the shortest distances from a single source vertex to all other vertices of a graph with non-negative edge weights [Dijkstra, 1959]. It can easily be altered to also output paths. It works by setting the distance to every other vertex in the graph to ∞, except for the source that can of course be reached at a distance 0. After this, as long as there are unvisited vertices left in the graph, the vertex v with the lowest distance to the source is visited. Upon visit, its neighbours' distances are updated if they are reachable faster by v. See the pseudo code in Algorithm 1.

Algorithm 1: Dijkstra's Algorithm

Input: Graph $G = (V, E, W)$, source node s
Output: Distance vector d of shortest path distances from s

```
1 Initialize d[v] ← ∞ for all v ∈ V, except d[s] ← 0
2 Initialize priority queue Q with all nodes in V
3 while Q is not empty do
4 |   Extract node u from Q with minimum d[u]
5 |   foreach neighbor v of u do
6 |   |   if d[u] + w(u,v) < d[v] then
7 |   |   |   d[v] ← d[u] + w(u,v)
8 return d
```

Correctness. Dijkstra's algorithm is only guaranteed to be correct if all weights of the graph are positive.

LEMMA 2.3.1 Dijkstra's algorithm outputs the shortest (s,t) distance for a chosen $s \in V$ and all other $t \in V$. [Dijkstra, 1959] □

Time Complexity. Because the algorithm looks at outgoing neighbours of every $v \in V$ once, it enters line 6 exactly m times (given that the graph is connected, otherwise fewer times). The extractions from the priority queue can using a Fibonacci heap be made to take $O(log(n))$ time [Fredman and Tarjan, 2001]. This leads to a total time complexity of $O(m + nlog(n)) = \tilde{O}(m)$.

The Bellman-Ford Algorithm

As edge weights may in some cases need to be negative, there is a need for an algorithm that can handle negative edge weights. The classical such SSSP algorithm is the Bellman-Ford Algorithm [Kociumaka and Polak, 2023].

Algorithm 2: The Bellman-Ford Algorithm

Input: Graph $G = (V, E)$, source vertex s, edge weights $w(u, v)$
Output: Shortest path distances $d[v]$ from s to all $v \in V$

```
1  Initialize d[v] ← ∞ for all v ∈ V
2  d[s] ← 0
3  for i ← 1 to |V| − 1 do
4      foreach edge (u, v) ∈ E do
5          if d[u] + w(u, v) < d[v] then
6              d[v] ← d[u] + w(u, v)
7  foreach edge (u, v) ∈ E do
8      if d[u] + w(u, v) < d[v] then
9          Report negative weight cycle and terminate
10 return d[v] for all v ∈ V
```

Correctness. *Definition:* π is a **h-hop restricted shortest path** between s and t if there is no path strictly shorter than π that has hop $\leq h$.

LEMMA 2.3.2 After running the Bellman-Ford algorithm the SSSP is found. More specifically after k rounds in the loop on line 3, the Bellman-Ford algorithm has found the k-hop restricted shortest paths from s.

Proof. Correctness can be proven inductively.

Base Case: At the beginning of round 1 all hop 0 shortest paths are known as there cannot be any.

Inductive Step: Assume that at the start of round i, all $i-1$-hop restricted shortest paths are found. Study an arbitrary i-hop restricted shortest (s,t) path $\pi_{s,v} \circ (v,t)$ in G. Clearly, $\pi_{s,v}$ is an $i-1$-hop restricted shortest path, and was thus found before round i. When entering row 4 for the edge (v,t), the path $\pi_{s,v} \circ (v,t)$ is found.

By induction, after round i all i-hop restricted shortest paths are known. As any path must have hop $\leq n-1$, all shortest paths have been found after $n-1$ rounds.□

Time Complexity. Every edge is looked at n times, leading to a time complexity of $O(mn)$.

Negative Weights

On a literal level, the shortest (s,t)-path is always defined, since there is a finite number of paths that can simply be iterated over to find the shortest one. In a graph with cycles, however there are possibly an infinite number of walks between any

two vertices. In a graph with no negative weight cycle a shortest walk is however always a path. When there is a negative weight cycle, a walk which takes one more lap in the cycle will be shorter. In this way, arbitrarily low walk lengths can be created, leading to undefined distances in the graph. Therefore we only consider the APSP problem well-defined if there are no negative weight cycles in the graph.

The Johnson transformation is a technique that can transform a graph without negative weight cycles into a graph without negative weight edges, but identical shortest paths. It requires running the Bellman-Ford algorithm once, and thus has time complexity $O(mn)$ [Johnson, 1977]. This transformation can be dynamically maintained with update time complexity $\tilde{O}(m)$ [Karczmarz, 2021].

Uniqueness of Shortest Paths

Reasoning about shortest paths in some cases becomes much easier if one can assume that shortest paths are unique.

A tool used by among others [Mao, 2024] is small random perturbations of edge weights. By adding tiny (randomized) numbers to the edge weights one can get a high probability of no two shortest paths being of equal length. As this is easy to do and has no disadvantages asymptotically, for the rest of the thesis we will assume that shortest paths are unique.

SSSP to APSP

Outside of possible special cases with extremely sparse or dense graphs, the fastest known asymptotic time complexity for static APSP in directed graphs with negative weights is reached using Johnson's algorithm. This algorithm has two steps:

- First: By running the Bellman-Ford algorithm from one source the graph can be transformed into a different graph with non-negative edge-weights but identical shortest paths.
- Second: The shortest paths are found by running Dijkstra's algorithm from every source.

This gives time complexity $O(mn + n(n\log(n) + m)) = O(mn + n^2\log(n))$.

2.4 Basic Principles of Dynamic APSP

The dynamic APSP problem is highly relevant in practice, as many real-world networks are inherently dynamic. The challenge lies in the fact that even a single vertex insertion or deletion can potentially alter the shortest paths between literally all pairs of vertices. As a result, dynamic APSP algorithms tend to be more complex than their static counterparts, although they often rely on static techniques as

foundational subroutines. A key focus is identifying the right set of paths to build a robust data structure, while ensuring that it can be efficiently updated and repaired as the dynamic graph evolves. Insertions and deletions furthermore require disparate measures, and will for clarity be discussed separately.

Incremental Problem

[Johnson, 1977] showed that the incremental problem (where only insertions are allowed) is fundamentally easy. The $(s,t)-$paths in G are a superset of the paths in $G \setminus \{v\}$. Therefore, if $\pi_{s,t}$ is the shortest (s,t)-path in G, then if $\pi_{s,t}$ does not intersect v, it must also have been the shortest path in $G \setminus \{v\}$. Consequently, the shortest (s,t)-path in G must be either a shortest path in $G \setminus \{v\}$ or a path including the vertex v.

Decremental Problem

Because vertex insertions can be handled so easily, the worst case fully dynamic algorithms have all focused on handling deletions. [Thorup, 2005; Abraham et al., 2017; Gutenberg and Wulff-Nilsen, 2019; Karczmarz and Sankowski, 2023; Mao, 2024] have all used the idea that one can separately handle those shortest paths that have low hop and those that have high hop. Somewhat counterintuitively the higher hop paths are much easier to find (as explained in Section 3.1). Additionally, shorter paths are at lower risk of having their component vertices deleted. The high hop paths are therefore simply recomputed every round, while careful attention is placed on repairability and repair of low hop shortest paths.

We can find all shortest paths including v, by performing Dijkstra's algorithm from v, once in G and once in $\overleftarrow{G}$. The shortest path through v will be the concatenation of the shortest $(s,v)-$path and the shortest $(v,t)-$path. The shortest path trees are found in $\tilde{O}(m)$. Taking the minimum between the original shortest path and the via-v-shortest path must be done for every pair (s,t). This means an additional cost n^2 per inserted vertex.

[Thorup, 2005; Abraham et al., 2017; Gutenberg and Wulff-Nilsen, 2019; Karczmarz and Sankowski, 2023; Mao, 2024] all use algorithms divided in preprocessing and batch deletion, where one preprocessing is used over a phase of several updates. At the beginning of each phase the preprocessing creates a data structure which calculates low hop shortest paths. This structure is the same throughout the phase, but successively greater batches of vertices are deleted from the data structure (as more and more vertices are deleted and inserted). To deamortize (spread out) the preprocessing time over updates, the next preprocessing starts while the previous one is still being used. Every preprocessing is built during Δ updates, and used for Δ updates, and hence has to handle at most 2Δ insertions and deletions. This is formalized in Lemma 2.4.1 below:

LEMMA 2.4.1 ([HENZINGER AND KING, 2001; THORUP, 2005]) Suppose there exists a data structure for a graph G that can handle the deletion of up to 2Δ vertices at once, for any subset $D \subset V$. This data structure computes, for every pair $(s,t) \in V^2$, a shortest path $\pi_{s,t}$ in the graph $G \setminus D$, and can return the first k edges of $\pi_{s,t}$ in $O(k)$ time.

If the preprocessing time is t_{pre} and the worst-case time for processing a batch deletion is t_{del}, then there exists a fully dynamic APSP algorithm with worst-case update time

$$O\left(\frac{t_{\text{pre}}}{\Delta} + t_{\text{del}} + \Delta n^2\right). \qquad \square$$

With this basic construction, in a sparse graph we instead get an update time of $O(t_{pre}/\Delta + t_{del} + \Delta m)$ and a query time of $O(\Delta)$, as we cannot afford to find the via-point that minimizes the (s,t)-distance for every pair (s,t) during update.

Finding High Hop Paths Using Via-Points

There is a simple Monte Carlo randomized algorithm for finding all shortest paths with hop at least h in time $\tilde{O}(nm/h)$, first used in [Ullman and Yannakakis, 1991] (in a different context). The algorithm, seen in Algorithm 3, models a folklore algorithm for finding hitting sets.

H is a random subset of V such that every vertex in V is placed in H independently with probability $10\log(n)/h$. For every vertex of H, Dijkstra's algorithm is run to find in- and out-going shortest paths. To find the shortest (s,t)-path passing through H one simply needs to iterate through H and choose the concatenation $(s \to H) \circ (H \to t)$ of minimal length.

Algorithm 3: Monte Carlo High Hop Shortest Path Search

Input: Graph $G = (V,E)$, query (s,t), hop bound h
Output: Approximate shortest path distance $Dist_{s,t}$

$H \leftarrow \emptyset$
foreach $v \in V$ **do**
  With probability $p = \frac{10\log n}{h}$: $H \leftarrow H \cup \{v\}$ ▷ Construct vertex subset H
foreach $v \in H$ **do**
  $d_v \leftarrow \text{Dijkstra}(G, v)$ ▷ Outgoing distances
  $\overleftarrow{d_v} \leftarrow \text{Dijkstra}(\overleftarrow{G}, v)$ ▷ Ingoing distances
Upon Query (s,t):
$Dist_{s,t} \leftarrow \min_{v \in H} \left(\overleftarrow{d_v}[s] + d_v[t] \right)$ ▷ Pick distance minimizing via-point
return $Dist_{s,t}$

A shortest path π of hop k has $k+1$ vertices on it. For every vertex of π the probability of lying in H is $p = 10\log(n)/h$ independently. Therefore, the probability of π not intersecting H can be expressed as:

$$Pr(\pi \cap H = \emptyset) = (1-p)^{k+1} = (1 - \frac{10\log(n)}{h})^{k+1} \leq$$

$$\leq \exp(-10\frac{k}{h}\log n) \leq \exp(-10\log n) = n^{-10}$$

Where we used the standard inequality $\exp(x) \geq 1 + x$. If we assume shortest paths to be unique, there are at most n^2 shortest paths of hop greater than h. With a union bound over these we get an upper bound $n^{-10} \cdot n^2 = n^{-8}$ on the probability of missing any shortest path of hop greater than h. We know **whp** that this algorithm finds all shortest paths of hop greater than h. Note that the 10 in $10\log(n)/h$ is not a special number, any number strictly greater than 2 would have sufficed to get correctness **whp**.

As the size of H can be modeled as a sum of n independent Bernoulli distributed random variables, a Chernoff bound gives [Hagerup and Rüb, 1990]:

$$Pr(|H| > 20n\log(n)/h) \leq \exp(\frac{-10n\log(n)}{3h}) = n^{-10n/3h}$$

Because $n \geq h$ this bound means **whp** the size of H is $\tilde{O}(n/h)$. This bounds the time complexity of the algorithm (which is $\tilde{O}(|H|m)$) with $\tilde{O}(nm/h)$ **whp**.

Hop Restricted Shortest Paths

The h-hop restricted shortest path between s and t is defined as being the shortest (s,t)-path of hop no greater than h. h-hop restricted shortest paths trivially have

the essential property that if the shortest path has hop less than h, it is the h-hop restricted shortest path. Another crucial property of h-hop restricted shortest paths is that if the shortest path in $G \setminus D$ has hop less than h, then either the h-hop restricted shortest path in G is shortest **or** it intersects D. It is positive to know that it was broken, as that frees up resources from the congestion scheme to repair the path.

Hop-restricted shortest paths are found in optimal time using the Bellman-Ford algorithm [Kociumaka and Polak, 2023] (Algorithm 2). The time complexity for finding single-source all h-hop-restricted shortest paths is $O(mh)$, simply because the loop on line 3 must only be run h times to find the $h-$hop-restricted shortest paths. The proof of correctness and time complexity is almost identical as for the original algorithm.

Repair of Low-Hop Paths

When using the framework introduced in [Henzinger and King, 2001; Thorup, 2005], one must be able to handle the deletion of a set D from the vertex set V. View the graph G that was preprocessed, and the graph after deletions $G \setminus D$.

Assuming that the data structure saved the shortest (s,t) path in G, and let us call this path π. If $\pi \cap D = \emptyset$, then π must also be the shortest path in $G \setminus D$. Therefore, in this case we must not do anything. If however $\pi \cap D \neq \emptyset$ there is a problem. Any path sharing a vertex with D is not a path in the new graph $G \setminus D$. Finding the new shortest (s,t)-path must be done, but is possible to do in a variety of different ways.

There are two main (known) types of repair suitable for worst case time guarantees, the first of which we will call ACK-repair, first used in [Abraham et al., 2017], and the second of which we will call GWN-repair, first used in [Gutenberg and Wulff-Nilsen, 2019].

ACK-Repair. [Abraham et al., 2017; Karczmarz and Sankowski, 2023] both use the type of repair that we will call ACK-repair, which is compatible with sparseness as the repair of an (s,t)-path has a time complexity proportional to the degree of the endpoint t. The algorithm is first described in text, and then in pseudo code in Algorithm 4.

A fundamental observation is that a hop-restricted shortest path in G (if it is preserved) is a hop-restricted shortest path in $G \setminus D$. If however the hop-restricted shortest (s,t)-path in G intersects D, the shortest (s,t)-path in $G \setminus D$ is made out of two segments: A prefix $s \to v$ that is the shortest $(s,w)-$path in G, and a suffix $v \to t$ made of edges (u,w) s.t the shortest $(s,w)-$path in G intersects D [Abraham et al., 2017].

Repairing all broken (s,t)-paths for a single source s thus reduces to computing the shortest paths from s in a subgraph H_s constructed of:

- The source s.
- Q_s, the set of vertices for which the shortest (s,t)-path in G intersects D.
- All vertices u for which $(u,w) \in E$ for a $w \in Q_s$
- All intact h-hop restricted shortest paths from s to vertices u in H_s.

See an illustration of such a subgraph in figure 2.3.

LEMMA 2.4.2 The paths found are not necessarily h-hop restricted, as Dijkstra's algorithm gives no such guarantees, but as the h-hop restricted shortest path is in the graph on which we run Dijkstra's algorithm, the found paths are at least as short as the h-hop restricted shortest.

See [Abraham et al., 2017; Karczmarz and Sankowski, 2023] for a proof. □

The repair cost (as we use Dijkstra for the shortest paths in H_s) is $\tilde{O}(|E(H_s)|)$. As every $(u,w) \in E(H_s)$ can only introduce one path $\pi_{s,v}$, we can bound the total cost for repair with $\tilde{O}(\sum_{\pi_{s,w} \cap D \neq \emptyset} deg_{in}(w))$. As this cost is highly related to the degrees of vertices in G, ACK-repair is qualitatively compatible with sparsity.

To quickly be able to find Q_s we use a data structure Π, which allows us to query for all paths containing a given vertex in time linear in the number of such paths.

Algorithm 4: ACK-REPAIR$(s, D, G = (V,E), \Pi)$

Input: Source node s, deleted nodes D, graph G, path set Π
Output: Repaired paths or updated graph

```
1 Q_s ← {t : π_{s,t} ∈ Π(v) for some v ∈ D};
2 H_s ← ({s} ∪ Q_s, ∅)                                  ▷ Sparse graph H_s
3 foreach t ∈ Q_s do
4     foreach (v,t) ∈ E do
5         Add (v,t) to H_s;
6         if v ∉ Q_s then
7             Add Π(s,v) to H as edge (s,v) with weight l(Π(s,v));
8 Run Dijkstra(s, H_s);
```

GWN-Repair. [Gutenberg and Wulff-Nilsen, 2019] and [Mao, 2024] instead use the fact that random sets of vertices of size $\tilde{O}(n/h)$ **whp** hit paths of length h. Knowing all h-hop restricted shortest paths, we can quite easily repair all $3/2h-$hop restricted paths.

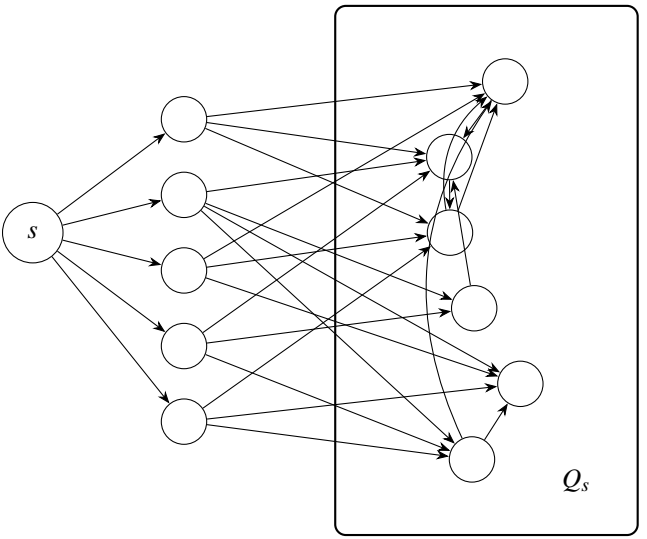

Figure 2.3 Illustration of possible ACK-repair subgraph. The only vertices to appear are the source s, the vertices of Q_s whose path from s was broken, and vertices with an outgoing edge to Q_s.

Consider the $3h/2-$ hop restricted path between two arbitrary vertices s and t. Randomly choose $\tilde{O}(n/h)$ vertices, these make up the set H. For all paths where the $3/2h$-hop restricted path in G intersects D we do:

$$\pi_{s,t,3/2h} = \min_{v \in H \cup \{t\}} \{l(\pi_{s,v,h} \circ \pi_{v,t,h})\}$$

If π has hop at least h, its middle third has length $O(h)$. Therefore **whp** it intersects H. Let v be in the intersection of the inner third of π and H. Then concatenating the h-hop restricted shortest (s,v)-paths and (v,t)-paths gives a path at least as short as π. If the $3h/2-$ restricted shortest path is not of hop greater than h we simply choose the $h-$hop restricted path as $3h/2-$restricted shortest.

Per broken $h-$hop restricted shortest path this costs $|H| = \tilde{O}(n/h)$, as we iterate through H. As n/h is independent to the sparsity of G, this method is worse suited for sparse graphs than ACK-repair.

Cost of Deletion

The cost of a single broken low hop path being known, we are able to describe the cost of deleting a specific vertex. When deleting a vertex, all paths including that

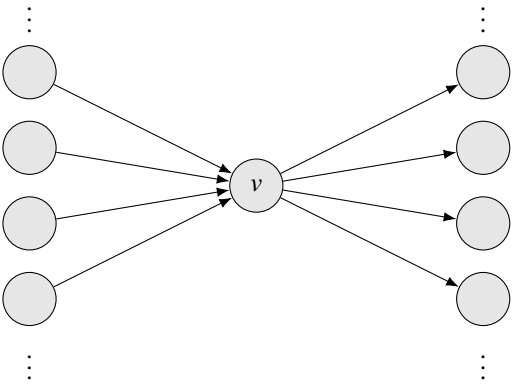

Figure 2.4 v lies on a large amount of shortest paths.

vertex are no longer paths of the graph. This means with ACK-repair the cost of removing a vertex v can be described as:

$$\text{cost}(v) = \sum_{\pi_{s,t} \in \Pi, s.t\ v \in \pi_{s,t}} \deg_{in}(t)$$

Where Π is the data structure storing paths. For GWN-repair there must be a data structure Π_j per h_j, s.t $h_0 = 1$ and $h_{j+1} \approx 3h_j/2$ up to some $h < n$. The cost for deleting v becomes:

$$\text{cost}(v) = \sum_{j=1 \ldots \log_{3/2} h} \sum_{\pi_{s,t} \in \Pi_j, \text{ so that } v \in \pi_{s,t}} n/h_j$$

Potentially there might be a vertex that sits on many/all shortest paths, see figure 2.4 below. This would make deleting it **very** expensive, if Π were simply to contain all low hop shortest paths. This issue has been circumvented in two main different ways, both of which based on removing vertices from the data structure before they become too expensive to delete.

Stones in Piles. [Abraham et al., 2017] handles exceptionally expensive vertices by processing vertices one by one, and ignoring previously processed vertices. By (every other round) processing the most expensive vertex in the graph, extremal values are bounded.

This method is based on the stones-in-piles problem, where in every round of a game N stones are distributed between k piles. If we always remove the pile of stones with the most stones, the maximal rock pile has at any stage at most $N \log(k)$ stones in it [Levcopoulos and Overmars, 1988]

This means that if processing one vertex distributes τ units of congestion in the graph, and at least every second vertex removed is maximally congested, the maximal level of congestion in the graph becomes bounded by $2\log(n)\tau$.

The amount of congestion distributed when finding all h-hop restricted paths from a single source s in ACK-repair becomes $h \cdot \sum_{v \in V \setminus \{s\}} \approx mh$, while the same for GWN-repair is $h \cdot n \cdot n/h = n^2$. 1

Congestion Threshold. The *stones in piles* method only works when viewing the processed vertices as via-points. When instead processing vertices as sources, another method can be used. [Gutenberg and Wulff-Nilsen, 2019] introduced the usage of a congested set of vertices C, to store vertices who intersect too many shortest paths in G (and whose deletion would require a lot of repair). C is chosen as the set of vertices for which the time complexity of updating broken paths exceeds τ, a given congestion threshold. After a vertex reaches the threshold during preprocessing, it is ignored when finding new paths. If the total amount of congestion distributed during preprocessing can be bounded by T, the number of congested vertices can be bounded with T/τ. Finding all paths going through these can be done in time $O((m+n^2)T/\tau)$, in the same way as we would handle insertions. At the same time, deletions of Δ vertices now costs at most $\Delta\tau$.

2.5 The Sparse Domain

Sparse Graphs

A sparse graph is generally understood to be a graph with relatively few edges, though the term is not precisely defined. When designing algorithms tailored for sparse graphs, the goal is not to restrict correctness to only sparse instances. Rather, the aim is for the algorithm to perform significantly more efficiently on sparse graphs than algorithms designed for dense ones, leveraging the fact that fewer edges often make problems easier to solve. This efficiency allows sparse-graph algorithms to scale to graphs with much larger vertex sets, provided the edge density remains low.

Inherent Difficulties for Sparse Graph Algorithms

The design of algorithms for the sparse domain differs from the dense domain primarily though that subroutines taking $\geq O(n^2)$ effectively become prohibited. In dense graphs, generally $m = O(n^2)$. This means that when designing an algorithm for the dense case, there is no real difference between a subroutine with time complexity $O(n^2)$ and one with $O(m)$. If a perfect sparsification of Mao's paper were to be possible and any factor n^2 could be turned into m the resulting update time would be $\tilde{O}(mn^{1/2})$. Because of the more limited toolbox, this perfect sparsification is also improbable from the beginning. This is further elaborated upon in Chapter 4.

Query Time

In [Karczmarz and Sankowski, 2023] there is, outside of the update time, also a query time, which comes from iterating through via-point sets. A natural question is whether it is possible to achieve lower than $O(nm)$ update time with a $\tilde{O}(1)$ or even $O(1)$ query time. Interestingly, a preliminary proof by Tianyi Zhang and Danupon Nanongkai shows that it is impossible to solve the sparse dynamic APSP problem with $\tilde{O}(1)$ query time and $\tilde{O}(n^2)$ update time, even in very sparse graphs.

Since the conjecture that the preliminary proof relies on (cubic hardness of the 3 set disjointness problem) is widely believed, it is implausible that it is possible to create a reasonable algorithm for the sparse regime without a query time exceeding $\tilde{O}(1)$.

3

Previous Work

This chapter presents two algorithms of special interest to the thesis. First to be presented is [Karczmarz and Sankowski, 2023] which is the only sparse APSP algorithm published to this point in time. As this algorithm shows the best theoretical guarantees of performance for sparse graphs it is both a source of inspiration in latter chapters and the benchmark which defines improvement over the state of the art. For both purposes, a solid understanding of the algorithm is necessary.

Second to be presented is [Mao, 2024] which is the second major source of inspiration. Mao's paper, which was only published shortly after [Karczmarz and Sankowski, 2023], vastly improved the worst case time complexity from $\tilde{O}(n^{2+2/3})$ to $\tilde{O}(n^{2+1/2})$. This is the result of introducing new concepts to the area of dynamic APSP. As these concepts were not connected to the subject area at the time of the research behind [Karczmarz and Sankowski, 2023] it was clearly motivated to ascertain the usefulness of these concepts in the sparse domain.

3.1 Karczmarz and Sankowski 2023

Presented below is the basic version of the algorithm by Karczmarz and Sankowski [2023]. Even though the authors manage to alter the algorithm into a deterministic one, we choose to present this version as it is both simpler and more closely related to the results of this thesis. Some details are also omitted for ease of exposition.

Algorithm Description

The preprocessing, batch deletion and query procedures are described separately.

Preprocessing. What is saved in [Karczmarz and Sankowski, 2023] are roughly the hop-restricted shortest paths in G. Vertices of V are processed in an arbitrary order. For each vertex the h-hop restricted shortest paths to all other vertices in G/C are found and stored (where C is the set of congested vertices).

Congestion is applied upon finding a hop-restricted shortest path. After finding an (s,t)-path $\pi_{s,t}$ $deg(t)$ units of congestion are given to all internal vertices of that path. If, after processing a vertex v, some vertices in the graph have congestion greater than the congestion threshold $\tau > mh$, then they are removed from the graph and put in the set of congested vertices C.

The preprocessing is dominated by finding hop-restricted shortest paths, which gives per vertex time complexity $O(mh)$, and overall time complexity $t_{pre} = O(mnh)$. The total congestion that can be distributed is at most $\sum_{s\in V}\sum_{t\in V} deg(t)hop(\pi_{s,t}) \leq h\sum_{s\in V}\sum_{t\in V} deg(t) = mnh$.

Update / Batch Deletion. The vertices are processed as starting points in an arbitrary order. For each starting point we find the end points whose hop-restricted shortest path intersects the set of deleted vertices D. The paths are repaired in the ACK-fashion in $\tilde{O}(\sum_{s\in V}\sum_{t\in V,\pi_{s,t}\cap D\neq\emptyset}\deg(t)) = \tilde{O}(\tau|D|)$, because of the congestion scheme. This is because no vertex in $(G\setminus C)\cap D$ has congestion greater than $\tau + mh = O(\tau)$.

To also find the high hop paths, we randomly sample a subset H of size $\tilde{O}(n/h)$ and find the shortest paths in and out from H. We do the same with C and I (where I is the set of new insertions since preprocessing). This means that $t_{del} = \tilde{O}(\tau|D| + (|H| + |C| + |I|)m)$.

Query. When querying we use the minimum distance of the hop-restricted shortest path that was repaired, and the shortest path through $H \cup C \cup I$. Iterating through all these sets takes time $O(|H| + |C| + |I|)$.

Parameter Tuning

We let Δ rounds go between preprocessings. That means $|D|, |I| \leq \Delta$. Since the total amount of congestion is bounded by $O(mnh)$ we get that $|C| = O(mnh/\tau)$

This gives a total update time complexity of:

$$O(mnh/\Delta + \Delta\tau + m^2nh/\tau + \Delta m)$$

And the parameters h, τ, Δ to be chosen. Setting $h = n^{1/5}$, $\tau = mn^{2/5}$ and $\Delta = n^{2/5}$ gives a total update complexity of $O(mn^{4/5})$ and a query complexity of $n^{4/5}$.

Update Time - Query Time Tradeoff

[Karczmarz and Sankowski, 2023] includes a way to alter the algorithm so that an even lower update time can be achieved, at the cost of a higher query time.

Instead of computing the ingoing and outgoing shortest paths to/from C, [Karczmarz and Sankowski, 2023] use the stones and piles method from [Abraham

et al., 2017] and process in- and outgoing $h-$hop restricted shortest paths to/from C. It is only necessary to find the h-hop restricted shortest paths through C, as the guarantee to hit high hop paths from H is general to the whole graph, and will also find paths passing through C.

This structure can be repaired in the same way as the original data structure, by ACK-repair, which is faster than running Dijkstra's algorithm every time. This allows for a larger congested set C, which makes for lower congestion leading to faster update times, but also to a slower query time as iterating through C naturally takes longer.

The tradeoff presented is: Query time $\tilde{O}(t)$ with $t \in [n^{4/5}, n]$ and update time of $\tilde{O}(mn/t^{1/4})$. This means the lowest possible update time achievable is $\tilde{O}(mn^{3/4})$.

Negative Weights

[Karczmarz and Sankowski, 2023] uses the dynamic algorithm for potential function originally presented in [Karczmarz, 2021],that makes any non-negative weight algorithm translatable to a non-negative cycle algorithm. This is of great value as it makes it possible to use fast algorithms like Dijkstra's, even though the original problem had negative weights.

3.2 Mao 2024

The algorithm presented in [Mao, 2024] is much more complex than the one in [Karczmarz and Sankowski, 2023]. It presents the novel tool of hop-dominant paths, and has a GWN-repair scheme, both of which add complexity. This Section is a concise explanation, focusing on the tools that are explored in chapter 4. Some aspects of the algorithm will be left out entirely, for ease of exposition.

Hop Dominant Paths

Hop dominant paths are a sub-class of hop-restricted shortest paths that were introduced into the study of APSP by [Mao, 2024]. Mao distinguishes between two different types:

- h-hop dominant paths: Paths that have hop not exceeding h, so that there is no (s,t)-path of hop less than $2h$ that is shorter.
- Strongly hop dominant paths: Paths π so that there is no shorter path with hop at most $4|\pi|$ (connecting the same vertex pair).

Trivially any shortest path is also strongly hop dominant. A property that [Mao, 2024] uses is that any strongly hop dominant path of hop h' must be h-hop dominant

for $h \in [h', 2h']$, which follows quite naturally from the two definitions. Of course, not all h-hop dominant paths are strongly hop dominant.

Hop dominant paths are not as straight-forward as hop restricted shortest paths. If the shortest path after deletions has hop less than h, this sadly says nothing about the destruction of strongly hop dominant paths, as it would have for hop restricted shortest paths. In fact there might not have existed any hop-dominant paths of hop anywhere close to h before deletions. It is however possible to couple the non-existence of a hop-dominant path to the presence of hop-restricted paths of different hop. Not being a strongly hop dominant path when having hop h' means either not being h' hop restricted shortest or not being $4h'$-hop restricted shortest. More clearly this is expressed in the following Lemma:

LEMMA 3.2.1 (Lemma 6.8 in [Mao, 2024]) If π of hop $\leq h$ is the shortest (s,t)-path after deletions, but it was not a strongly hop dominant path before deletions, then either the $h-$hop or the $4h$-hop restricted shortest (s,t)- path was broken by deletions.

Proof. Assume that the $4h-$hop-restricted path π' before deletions was not broken by deletions, it is therefore still present in the graph. In that case, since π of hop h is the shortest (s,t)-path after deletions, it must be identical to π'. This however means that π is of hop $\leq h$, and was the $4h-$restricted shortest path before deletions, making it a hop-dominant path before deletions. This contradicts the assumption that it was not. □

Finding Hop Dominant Paths

[Mao, 2024] introduces an algorithm that finds all h-hop dominant paths. This algorithm is called `RestrictedDijkstra`, and as the name suggests works just like Dijkstra's algorithm, except for that it stops when a certain depth has been reached in the graph, and at equal weighted distances it favors paths of lower hop. The pseudo code for the procedure is in Algorithm 5.

Algorithm 5: `RestrictedDijkstra(s, G, h)`

Input: Graph G, hop limit h, source s
Output: Superset of h-hop dominant paths from s

foreach $v \in V \setminus \{s\}$ **do**
  $\pi[v] \leftarrow \perp$ ▷ Initialize all paths to undefined
$\pi[s] \leftarrow (s)$
$V_0 \leftarrow V$ ▷ Set of unvisited vertices
while $V_0 \cap \{v \mid \pi[v] \neq \perp\} \neq \emptyset$ **do**
  $u \leftarrow \arg\min\{w(\pi[v]) \mid v \in V_0\}$
  $V_0 \leftarrow V_0 \setminus \{u\}$
  if $|\pi[u]| < h$ **then**
    foreach $v \in V_0$ **do**
      if $w(\pi[v]) > w(\pi[u] \circ (u,v))$ *or* $((w(\pi[v]) = w(\pi[u] \circ (u,v))$ *and* $|\pi[v]| > |\pi[u]| + 1))$ **then**
        $\pi[v] \leftarrow \pi[u] \circ (u,v)$
return $\{\pi[v] \mid v \in V\}$

Algorithm Description

The improvement from this algorithm builds on the fact that hop-dominant paths can be quickly found ($\tilde{O}(m)$ instead of $O(mh)$). That said, hop dominant paths do not hold the same guarantees as do hop-restricted shortest paths. [Mao, 2024] however manages to couple the cost necessary for repairing hop-dominant paths to the cost necessary for repairing hop restricted shortest paths.

Preprocessing. In parallel two processes are being run. A congestion scheme in a hypothetical data structure based on hop-restricted shortest paths and the preprocessing of the graph with hop dominant paths.

The preprocessing uses a layered congested set approach. It works in L rounds, counting down from L. In every layer there are two sets of vertices that are of special interest: The set of centers C_i and the set of congested vertices O_i. It starts out with a set of centers $C_L = V$. The preprocessing works in rounds where the vertices that are congested in round i are centers in round $i-1$. We use an exponentially growing congestion threshold $\tau_i = \tau/2^i$, which makes sure that the size of the congested set shrinks exponentially.

The first thing we do is sample a congested set O_i. This is done by randomly sampling some of the vertices with probability p_j and finding hop-restricted shortest paths through the centers. Vertices on a h_j-hop-restricted shortest path are congested with $\frac{1}{p_j} n/h_j$. The expected congestion still becomes the same as if we

would have processed all vertices. Congested vertices are put into O_i. By randomly sampling sources, it becomes possible to achieve low preprocessing time while still constructing a hop-restricted shortest path congestion scheme that closely approximates one built using all possible sources.

After this we process all vertices but instead of the Bellman-Ford-algorithm use the Restricted Dijkstra algorithm through the set of centers C_i in G/O_i. By running `RestrictedDijkstra` for every hop limit $h_j = \lfloor (3/2)^j \rfloor$, for $j = 1, 2, ..., i_h$, s.t $h_{i_h} = h$ all strongly hop dominant paths of hop less than h are found, and put into a structure Π_i.

After this we additionally find all concatenations between 2 paths of the same hop-level and same congestion layer, and replace the original path with such a concatenation if the concatenation is shorter.

In the next round $C_{i-1} = O_i$. By choosing the minimum length path of each hop-class we find all hop-dominant paths not going through O_0 (the smallest congested set). This is shown with a telescoping argument: Assuming there is even a h_j-hop dominant path. Level i finds the hop dominant path if it passes through C_i but not through C_{i-1}. Therefore the shortest path from any layer is the hop dominant path if it passes through $C_L = V$ but not through $C_{-1} = O_0$.

The whole process is described concisely in the pseudo code of algorithms 6 and 7:

Algorithm 6: SSAHDP (Single source all hop dominant paths)

Input: Graph G, hop h, Source s
Output: Set P
for $j \leftarrow 1$ *to* $\lceil \log_{3/2} h \rceil$ **do**
    $h_j \leftarrow \lfloor (3/2)^j \rfloor$
    $P \leftarrow P \cup$ `RestrictedDijkstra`(s, G, h_j)

Algorithm 7: [Mao, 2024] Preprocessing Algorithm

Input: i, C_i

Output: Updated Π_i and C_{i-1}

$O_i \leftarrow \text{Sample}(i, C_i)$ ▷ Congested set found through random sampling

$\Pi_i[*,*,*] \leftarrow \perp$ ▷ Initialize all paths to undefined **foreach** $c \in C_i$ **do**

  $P_{\text{from}} \leftarrow \text{SSAHDP}(G \setminus O_i, h, c)$ ▷ Outgoing hop dominant paths

  $P_{\text{to}} \leftarrow \text{SSAHDP}(\overleftarrow{G \setminus O_i}, h, c)$ ▷ Ingoing hop dominant paths **foreach** $p_{from} \in P_{from}$, $p_{to} \in P_{to}$ **do**

    $p \leftarrow p_{\text{to}} \circ p_{\text{from}}$ ▷ Concatenate[a] paths

    $j \leftarrow h^{-1}(|p|)$

    if $w(p) < w(\Pi_i[p[0], p[|p|], j])$ **then**

      $\Pi_i[p[0], p[|p|], j] \leftarrow p$

$C_{i-1} \leftarrow O_i$

[a] We define for any p, $p \circ \perp = \perp \circ p = \perp$.

To analyze time complexity let us study an arbitrary layer j. We need to process $|C_j|$ vertices, at a cost of $|C_j|n^2$. This preprocessing is repeated every Δ_j rounds. In between preprocessings at most Δ_j vertices are removed, leading to a repair cost of $\Delta_j \tau_j$. $\Delta_j = O(2^j)$, $\tau_j = \tau/2^i$, $|C_j| = \tilde{O}(n^3/\tau_j) = \tilde{O}(n^3 2^j/\tau)$. This means preprocessing costs $\tilde{O}(n^5/\tau)$ per round and updates cost $O(\tau)$ per round, which can be balanced out to $\tilde{O}(n^{2+1/2})$ with the choice of $\tau = n^{2+1/2}$.

Update. Paths of hop greater than h are found anyways using the standard method detailed in Section 2.4. Therefore the requirement that the update procedure needs to fulfill is that when the shortest (s,t)- path has hop smaller than h, then the algorithm must find it.

Since the algorithm is based on GWN-repair all repairs are concatenations between two paths. [Mao, 2024] however, distinguishes between two fundamentally different path repairs.

- Type I reconstructions: When a repaired path is a concatenation between two paths that were saved in the data structure before deletions.
- Type II reconstructions: When at least one of the concatenated paths was itself repaired.

Type I. All type I reconstructions are done before all Type II reconstructions. All paths saved that intersect D are exchanged with their shortest GWN-repair concatenation. After this all type I reconstructions are done. This will not be discussed further, as the discussion in chapter 4 focuses on type II reconstructions.

Type II. Before doing type II reconstructions it is vital to be able to query high-hop path lengths. This is because one can on by Lemma 3.2.1 only relate costs to hop-restricted path congestion is one repairs a globally optimal path. Since the preprocessing only regards itself with low hop paths, one must check high hop paths to make sure this condition is met. This is done using the standard technique with a hitting set, and the results are stored in a distance matrix $A_{s,t}$.

Type II reconstructions are done as follows: The (s,t)-pairs that have had a type I reconstruction are all put into a set Q. Pairs from this set are then taken one by one to be extended, always extracting the pair of minimal distance.

Upon extraction from Q, if the (s,t)- path is shorter than the distance stored in $A_{s,t}$, we know that we have found the optimal path. If this is the case one tries concatenating the (s,t)-path with $\tilde{O}(n/h_j)$ different $(v,s)-$paths and $(t,v)-$paths. Those $(v,t)-$ and $(s,v)-$ pairs for which this new path is shorter than the path that was saved there get the new concatenated path and are put into Q. Because we only extend paths that are globally optimal we can use Lemma 3.2.1 to know that the saved hop-restricted shortest path was broken for h_j or h_{j+4}. This makes it so that the cost of extending $O(n/h_j)$ is at most a constant factor larger than the cost of repairing in the hypothetical data structure.

The guarantee of correctness is built from low hop to high hop, roughly as follows: We study a shortest path $\pi_{s,t}$ of hop $\in [h_{j-1}+1, h_j]$, that is of type II. Assuming that all shortest paths of hop $\leq h_{j-1}$ were recovered, also a subpath of π of suitable length has been recovered. As this path was extended, π must have been found.

Query. As the algorithm is optimized for the dense case, there will be a distance matrix that can be queried in $O(1)$ time. This holds for any pair $(s,t) \in V \times V$ the smallest distance among (s,t)-paths in the following categories:

- The shortest path in Π of any hop class h_j and layer i
- The reconstructed path if there is one
- The shortest path going through the random hitting set H
- The shortest path going through the smallest congested set O_0
- The shortest path going through some vertex inserted since the last preprocessing.

4

Unsuccessful Application of Dense Techniques to the Sparse Setting

Since graphs are allowed to be dense in the algorithm by [Mao, 2024], when analyzing time complexity the bound $m = O(n^2)$ is used. This means that when designing an algorithm for the dense case, there is no real difference between a subroutine with time complexity $O(n^2)$ and one with $O(m)$. If a perfect sparsification of Mao's paper were to be possible and any factor n^2 could be turned into m, the resulting update time would be $\tilde{O}(mn^{1/2})$. This result would be an impressive improvement upon [Karczmarz and Sankowski, 2023]. Since, however, the toolbox is more limited when constructing a sparse-graph algorithm with sub n^2 query time, what one can hope for is somewhere between $\tilde{O}(mn^{1/2})$ and $\tilde{O}(mn^{4/5})$ as modeled by [Karczmarz and Sankowski, 2023], using the tools from [Mao, 2024].

This chapter systematically goes through different tools and subroutines used in [Mao, 2024], and explores whether these can plainly, or with some level of modification be used in the sparse context for an update time complexity better than the trivial $\tilde{O}(nm)$ or $O(n^2)$.

The first two sections explore the fundamental reasons dense techniques cannot be applied directly to sparse graphs without incurring some loss in efficiency. This is followed by a discussion that demonstrates that layered congested sets cannot be effectively sparsified. The chapter concludes with a detailed explanation of why the `RestrictedDijkstra` procedure and hop-dominant paths are inherently incompatible with the sparse setting.

4.1 ACK-Repair vs GWN-Repair

The cost of repairing the same h-hop restricted path $\pi_{s,t}$ is in ACK-repair $\deg_{in}(t)$ and GWN-repair n/h. To allow for worst-case guarantees these costs are bound in congestion schemes, which leads to $h \cdot deg(t)$ units of congestion in an algorithm with ACK-repair in contrast to $h \cdot n/h = n$ units of congestion in an algorithm with GWN-repair.

In a dense graph $deg(t)$ may generally be $O(n)$. This means that using ACK-repair will actually cost a factor h more in dense graphs than GWN-repair. Thus, ignoring whether it is even possible, implementing the algorithm by [Mao, 2024] with ACK-repair would already lead to a factor h increase in batch deletion complexity. In the dense case this would at best yield an update time of $\tilde{O}(n^{2+2/3})$, making the best update and query time possible for a sparse algorithm mimicking [Mao, 2024] $\tilde{O}(mn^{2/3}), \tilde{O}(n^{2/3})$.

With this said, working with GWN-repair in the sparse domain is not the solution, as in that case it becomes impossible to link the batch deletion complexity to the number of edges in G. Total graph congestion reaches $\tilde{O}(n^3)$ any time all pairs h-hop restricted shortest paths are found, no matter the value of h, meaning that even removing an averagely congested vertex requires $O(n^2)$ time for path repair.

4.2 Source of Issues: Via-Points

As detailed in chapter 2.4, taking a subset $H \in V$ of the vertices in the graph and calculating the shortest paths that go through that vertex set is a standard technique used in [Abraham et al., 2017], [Gutenberg and Wulff-Nilsen, 2019], [Mao, 2024],[Karczmarz and Sankowski, 2023].

If one simply stores the shortest in- and outgoing path to/from the via-point set, the algorithm only takes time $\tilde{O}(m|H|)$. The time needed to turn in- and outgoing shortest paths into a distance matrix, however, amounts to $O(n^2|H|)$. This is because this entails taking the element-wise minimum of $|H|$ $n \times n$ matrices.

Since $O(n^2|H|)$ or $O(n^3/h)$ update time is unacceptable in the sparse case, a sparse update algorithm should avoid the computation of a distance matrix though a via-point set. The algorithm in [Karczmarz and Sankowski, 2023] contains via-point distances, but keeps the information as shortest path trees in G and $\overleftarrow{G}$. During query H must be iterated through, which gives additional query time $O(|H|)$.

The dense case algorithms, however, also use via-points for other purposes than just being able to query $> h$ hop paths in the query phase. During the same up-

date/preprocessing a distance through a via-point set might have to be calculated and queried for many ($O(n^2)$) (s,t)-pairs.

As this quite instantly leads to a time complexity $> n^2$ these subroutines from [Mao, 2024] and others cannot be used for a sparse algorithm, at least without significant alterations. The next sections all highlight how it is difficult to get the performance gains that come with the techniques of [Mao, 2024] without doing via-point computations incompatible with a sparse algorithm faster than $O(n^2)$.

4.3 Layered Congested Sets

First used in [Gutenberg and Wulff-Nilsen, 2019], and refined by [Mao, 2024], the method of having layers of congested sets is helpful for the dense algorithms as they allow for the top layer to have a small congested set C_i, the paths through which must be found using Dijkstra's algorithm in every round. All layers have the same (optimal) average preprocessing cost per round, but by having the layered structure it is possible to combine this optimal preprocessing cost with having a very small top layer congested set.

Layered congested sets are not something that can be used to gain anything in a sparse setting. To argue for this we will first make define what we mean with layered congested sets:

Definition: By *Layered congested sets* is meant a construction with sets $C_0 ... C_i$, such that $C_0 = V$ and $|C_j| < |C_{j-1}|$ for all j, or at least the upper bounds for set sizes have this relation. Furthermore, (s,t)-distances must be queryable in the structure with query time $O(|C_i|)$, and strictly less time than $|C_{i-1}|$. For each layer j there exists τ_j so that no vertex is more than τ_j congested in ACK-repair sense in this layer. For every layer there exists Δ_j so that the layer is recomputed every Δ_j updates. τ_j is increasing with j, and Δ_j is decreasing with j.

View layer $i-1$. Since we are not allowed to iterate through C_{i-1}, the h-hop restricted shortest paths through C_{i-1} but not C_i must be calculated in the preprocessing phase.

The preprocessing of layer $i-1$ can be done in 2 different ways:

If done by finding in- and outgoing paths to C_{i-1}, this takes time $O(|C_{i-1}|(mh+n^2)$. This means that to get a deamortized update time lower than n^2, we require $\Delta_{i-1} > |C_{i-1}|$. This means as $\Delta_{i-1}\tau_{i-1} < nm$, $\tau_{i-1} < nm/|C_i|$. This however means that $|C_i| \approx nmh|C_{i-1}|/nm = h|C_{i-1}|$. Hence, C_i might become even greater than C_{i-1}, which clearly strays from the definition of the layered congested set.

There is however a way to find the shortest paths through a set, without concatenation. This is done by finding from every source the hop restricted shortest paths through the set C_{i-1}. This preprocessing takes $O(mnh)$ time. This means that we get update time complexity $\tilde{O}(nmh/\Delta_{i-1} + \tau_{i-1}\Delta_{i-1} + (mh)mnh/\tau_{i-1} + n/h)$, and as we have to iterate through C_i, which is of size $O(mnh/\tau_{i-1})$, we get query time $\tilde{O}(mnh/\tau_{i-1} + n/h)$. Removing the index $i-1$ gives us terms larger than the ones in the time complexity of [Karczmarz and Sankowski, 2023], meaning that the layered congested set method cannot improve over this algorithm. Effectively, with this approach all layers below the top layer collapse down to a single layer.

Consequently, the layered congested set approach is not suitable for sparse graphs, as long as it adheres to the (fairly general) definition stated above in this section.

4.4 Repair of Hop Dominant Paths

The key technique that makes [Mao, 2024] an improvement over earlier work is the use of hop-dominant paths instead of hop-restricted shortest paths. Mao manages to combine the faster preprocessing of hop-dominant paths with the repair time complexity of hop-restricted shortest paths. If this were possible in a sparse context, that would instantly lead to an improved time complexity.

As no algorithm can get subquadratic time with GWN-repair, hop dominant paths would have to be repaired with ACK-repair, shy of developing a fundamentally different repair technique. This is not implausible, though, as the statement underpinning ACK-repair: *Any h-hop restricted shortest (s,t)-path in $G \setminus D$ is the concatenation of a h-hop restricted (s,v)- path in D and a (v,t)-path consisting of edges (u,w) such that the h-hop restricted (s,w)-path in G intersected D.* can be slightly modified in the case of hop-dominant paths to: *Any h-hop dominant (s,t)-path in $G \setminus D$ is the concatenation of a h-hop dominant (s,v)-path in D and a (v,t)-path consisting of edges (u,w) such that the h-hop dominant (s,w)-path in G intersected D or did not exist in G.* This might or might not be strong enough to make some version of ACK-repair work, however there is a more compelling reason why hop dominant paths could not work in a sparse context.

Problems with Lemma 3.2.1

In [Mao, 2024] Lemma 3.2.1 is what attributes cost from the repairs of hop dominant paths to a congestion scheme built with hop-restricted shortest paths. [Mao, 2024] uses a slightly altered version of Lemma 3.2.1:

LEMMA 4.4.1 (LEMMA 6.8 FROM [MAO, 2024]) If the globally optimal path after deletions is of hop less than h, and it is repaired during the update procedure at a hop h_j, then either the h_j-hop-restricted or the $(3/2)^4 h_j$-hop-restricted shortest path in G intersects D. □

Given the repair of a path in the update procedure of [Mao, 2024] one knows that it is the globally optimal path (and therefore strongly hop dominant), but only if the globally shortest path is not of hop higher than H. This condition can be checked by comparing the path length of the repaired path with a random hitting set via-path construction as explained in chapter 3.1. This is where problems arise in a sparse context, as querying these via-point distances either gives a query time of $\tilde{O}(n/h)$ or an update time of $\tilde{O}(n^3/h)$. Since an update time of $\tilde{O}(n^3/h)$ cannot possibly be acceptable in a sparse context, we entertain the though of a query time of $\tilde{O}(n/h)$.

Let Q be the set of (s,t)-pairs for which the h-hop path is repaired in a batch deletion algorithm. Assume there is a regular graph ($\deg_{in}(v) = m/n$), where we use ACK-repair with an update time of $\tilde{O}(mn^{3/4})$. This gives us a rough bound for the size of Q. Repair Costs $= \sum_{(s,t)\in Q} \deg_{in}(t) = |Q|\frac{m}{n} \Rightarrow |Q| \approx n^{1+3/4}$.

With a hitting set of size $n^{3/4}$ for long paths, which creates a query time of $n^{3/4}$, this would let the time in the updates for just checking whether we are extending an optimal path reach a prohibitive $O(n^{2.5})$. Taking these queries into account there would need to be a bound on the size of Q of $|Q| \leq n$, meaning that every vertex is allocated only a single repair, which is hardly realistic.

Bypassing Lemma 3.2.1

Since Lemma 3.2.1 showed little promise for sparse graphs, we entertained the idea of designing a different way to couple a congestion scheme with hop restricted shortest paths with repairs on hop dominant paths.

To gain a fast preprocessing of a congestion scheme with hop restricted shortest paths it is necessary to subsample sources (as done in [Mao, 2024]). To be able to relate the time needed to repair hop dominant paths to the congestion in the hop restricted shortest paths, there needs to be some knowledge of when hop restricted shortest paths are being broken. Subsampling sources, however, makes it so that one does not know when a hop-restricted shortest path has been broken.

If one managed a superset of the hop-restricted shortest paths in the graph, then every time a hop restricted path is broken we know. A fairly straight-forward way of doing this in a way that allows for quick preprocessing is using via-points in a

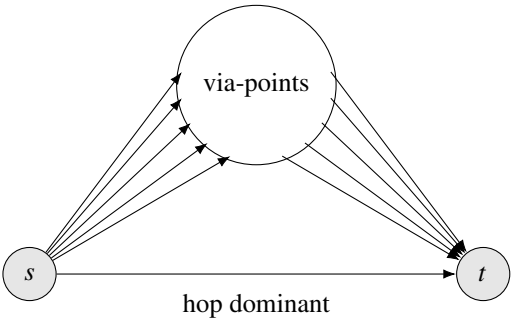

Figure 4.1 Alternative congestion scheme. If any of the hop restricted shortest paths to/from the via-point set breaks, then resources are freed to repair the hop dominant path.

fashion similar to [Abraham et al., 2017], but to never evaluate which via-point is optimal.

The construction is as follows: For every $i \in [\log(h)]$ we sample a set of size $\tilde{O}(n/2^i)$ and calculate in- and outgoing 2^{i+1} hop restricted paths to/from the sampled set. We keep track of which in/outgoing paths intersect D to get a superset of which hop-restricted shortest paths intersect D. This construction would bypass Lemma 3.2.1 altogether, by coupling the extension costs of an (s,t) pair to the potential destruction of a hop restricted shortest path directly, instead of coupling it indirectly by the new optimality of the (s,t)-path. See an illustration in figure 4.1.

When using this construction, the decisive problem is that upon deletion from an outgoing path (c,t) from a via-point c, we have to extend all (v,t), $v \in V$. This is because potentially any (s,t) path over v is the hop restricted shortest path before comparing with other pivots. This adds an extra factor n to the congestion distributed.

This means that if we let the extension of pair (s,t) cost $\deg(t)$, as it would with ACK-repair: Per pivot vertex we would need:

$$n2^{i+1} \sum_{t \in V} (\deg(t))$$

As we need $\tilde{O}(n/2^i)$ pivots to find all shortest paths in the hop range $[2^i, 2^{i+1}]$ **whp**, this leads us to a total congestion in the graph of:

$$\tilde{O}(n^2 m)$$

Since no congestion scheme could ever work with a threshold smaller than the average:

$$\sum_{v \in V} \frac{\text{Congestion}(v)}{n}$$

this means that at least:

$$\tau \geq \tilde{O}(mn)$$

This means, however, that any deleted vertex can incur cost equal to the static recomputation of APSP. The extra factor n is the culprit here, but can not be removed as long as the algorithm does not establish for each vertex pair and hop limit, which is the best via-point to pass.

4.5 Further Analysis of `RestrictedDijkstra`

[Mao, 2024] is a faster algorithm than [Gutenberg and Wulff-Nilsen, 2019], solely because it has a faster preprocessing. The fast preprocessing stems from the `RestrictedDijkstra` subroutine being a factor h faster than h-level Bellman-Ford.

Preceding sections show that there are considerable difficulties with embedding hop dominant paths in a congestion scheme based on hop restricted shortest paths. This does not, however, rule out that hop dominant paths could in some way be used in the sparse context. The following questions are still open: Can one design an algorithm with the `RestrictedDijkstra` procedure as its main preprocessing subroutine that:

- Does not rely on hop-restricted shortest paths for its time complexity guarantees?
- Can be used in the sparse context for an improvement over $\tilde{O}(mn^{4/5})$ update and $\tilde{O}(n^{4/5})$ query time?

To have any chance of doing any of the above, one must find combinatorial properties of the paths found with `RestrictedDijkstra` which have not been used in [Mao, 2024]. The following sections hold findings on the paths output from `RestrictedDijkstra`.

Lacking Specificity

One detail is that while strongly-hop dominant paths are a subset of the hop-restricted shortest paths, `RestrictedDijkstra` by no means only finds hop-restricted shortest paths. Taking for example the construction in figure 4.2, if we

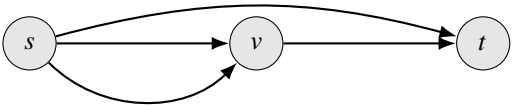

Figure 4.2 The arrows are not edges but instead represent paths of different hop and length. Let the straight path be the h-hop restricted shortest (s,t)-path. If however the lower (s,v)-path is shorter, but with a hop close to h, then `RestrictedDijkstra` will instead find the upper (s,t)-path though it is not hop-restricted shortest.

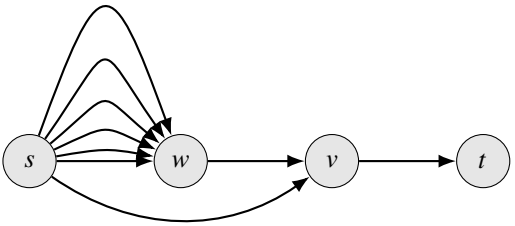

Figure 4.3 Let the curved (s,w)-paths be increasingly shorter but higher hop paths of hop $h_j - 1$. Let the (w,v)-segment be of hop 2. Let (s,v) and (v,t) be edges, and let the weight of (s,v) be enormous. It is clear that at every hop level `RestrictedDijkstra` finds the (s,t)-path s,v,t.

let a shorter path of higher hop intersect the h-hop restricted path then a path which is not the h-hop restricted path may be found, instead of the h-hop restricted shortest one.

Non-Existence

This behaviour, of not even finding hop restricted shortest paths, can even be continued throughout **all** levels of a hop-level scheme. as shown by figure 4.3. This means that if an algorithm simply uses `RestrictedDijkstra` to preprocess low hop paths it cannot only repair/optimize the stored (s,t) path if they have been broken. Imagine the highest hop path in the example graph is broken. Then random hitting sets will not be guaranteed to find the second highest hop path because it is not of hop large enough. The algorithm will therefore give the wrong result with some nonnegligible probability.

This is something that we can see handled in [Mao, 2024], as even vertex pairs with intact paths can be extended to in the repair procedure.

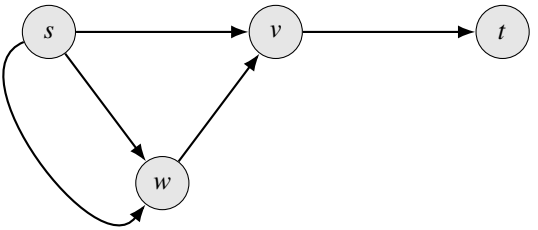

Figure 4.4 The bent (s,w)-segment has hop $h-1$ and is shorter than less bent (s,w)-segment. Less bent (s,w,v) path is of hop $h-1$. Straight (s,v,t) path is h hop restricted since passing through w gives hop $> h$. The straight (s,t)-path will be found with `RestrictedDijkstra` even though $s-w-v$ is shorter than the straight path (s,v) prefix.

Non-Equivalence Between Shorter Prefix and Not Finding Path

A fact used in [Mao, 2024] is that if the (s,t)-path p is not found, we know that there is $v \in p$, such that there exists a h-hop restricted p' from s to v that is shorter than the $s-v$-prefix of p. Curiously, the converse is not true:

Even if there is a h-hop restricted path shorter than some prefix of p, p might still be found, as visible in figure 4.4

ACK-Repair Applied to `RestrictedDijkstra` Paths

With the goal of a fast sparse algorithm, ACK-repair is much more plausible to work than GWN-repair, since GWN-repair cannot use vertex degrees (and therefore sparsity) in congestion schemes. Several issues with repairing `RestrictedDijkstra` paths with ACK-repair are however apparent. ACK-repair is based on being able to partition the vertex set into vertices whose hop restricted shortest path in G does not intersect D and those whose path has been broken.

We study a source $s \in V$ as in figure 4.3. Assume between s and t the stored path was not broken. As one can not know whether there is a shorter path than that path, it becomes unclear whether to add t as a non-reachable vertex.

Upon deletion of the shortest path in figure 4.5 (that has hop greater than h), it becomes apparent that we must find the h-hop restricted shortest path from s to t. However, not only is there no way of knowing that the path should be repaired as we never stored the shortest path as an (s,t)-path, but to construct the new optimal path one must go through vertex v that was never processed in the preprocessing, and thus, if only using `RestrictedDijkstra` it is impossible to assign congestion to v in a way that relates the cost of repairing paths via them to the deletion of

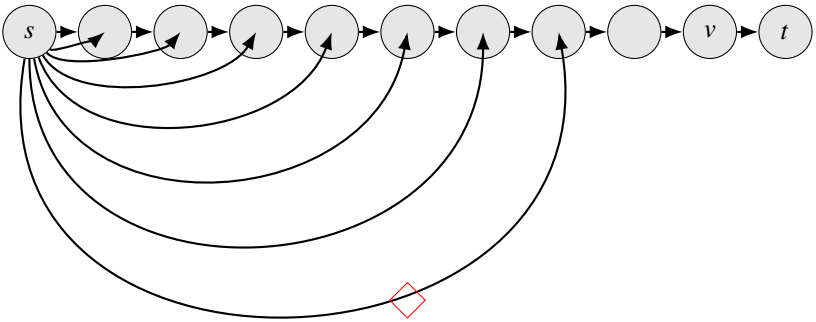

Figure 4.5 Let each of the curved paths be shorter than the previous, and of hop $h_j - 1$ for increasing j. `RestrictedDijkstra` never finds a h-hop restricted path to t even though there is a path of hop 10. Upon a deletion on the shortest (highest hop) bent path (marked by the red square), the algorithm must find a h-hop restricted path.

certain vertices.

5

Novel Algorithm with Improved Update-Query Time Trade-Off

5.1 Introduction

This chapter describes a novel algorithm that manages to surpass [Karczmarz and Sankowski, 2023] in terms of query time - update time trade-off. The result is formulated in Theorem 5.1 below:

THEOREM 5.1 Let $G = (V, E, W)$ be a real-weighted directed graph, where $|V| \leq n, |E| \leq m$ and let t be an arbitrary number so that $t \leq n^{8/9}$. Assume further that G has no negative length cycles at any point during updates. There exists a randomized data structure maintaining G under fully dynamic vertex updates and answering arbitrary pairwise distance queries with $\tilde{O}(mn^{2/3} + mn^2/t^{3/2})$ worst-case update time and $\tilde{O}(t + n^{2/3})$ query time, using $\tilde{O}(n^2)$ space. After answering a distance query, some corresponding shortest path π can be reported in $O(|\pi|)$ time. Both correctness and time complexities are true **whp**. □

5.2 Technical Overview

To create an update-query time trade-off [Karczmarz and Sankowski, 2023] manage to remove the bottleneck of having to do $O(|C|)$ Dijkstra computations in every update. There is however a main bottleneck left: Paths longer than h must be found using a hitting set. This requires a hitting set of size $\tilde{O}(n/h)$, and therefore $\tilde{O}(mn/h)$ update time from Dijkstra computations. As $\tau > mh$, the three terms $\tau\Delta + mnh/\Delta + mn/h$ already guarantee an update time $> mn^{3/4}$.

On the other hand, [Abraham et al., 2017] is not at all designed as a sparse algorithm. It has a major sparsification bottleneck when it comes to handling low

hop paths, as it requires us to find the shortest path going through a set of size $O(n)$. Thereby acquiring at least a query complexity of $O(n^2)$ which is hardly an improvement over the trivial Dijkstra-based approach in a sparse graph.

The novel approach taken by this partially original algorithm is that shortest paths of hop between some h_1 and some h_2 will be handled separately. This means shortest paths are grouped into 3 categories:

- **Low hop paths** are defined as having hop $\leq h_1$. These are handled using the algorithm in [Karczmarz and Sankowski, 2023] used for trade-off, detailed in Section 3.1.
- **Mid hop paths** are defined as having hop between h_1 and h_2. These are handled using subprocedures from [Abraham et al., 2017].
- **High hop paths** are defined as paths of hop greater than h_2. These are handled using the standard random hitting set trick detailed in Section 2.4.

The algorithm can match the time complexity of [Karczmarz and Sankowski, 2023], but has a better trade-off between update and query time. This effectively makes it possible to reach an update time of $\tilde{O}(mn^{2/3})$ unattainable in [Karczmarz and Sankowski, 2023].

5.3 Low Hop Paths

Algorithm Description

The algorithm for keeping track of low hop paths follows the approach used by [Karczmarz and Sankowski, 2023] to create an algorithm with an update-query time trade-off.

The algorithm takes as its input the graph G and a hop limit h_1. It outputs the set of congested vertices C, as well as a data structure Π, storing:

- For every vertex v all those paths found intersecting v, queriable as $\Pi(v)$
- For every vertex pair (s,t) a h_1-hop restricted C-improving[1] (s,t)-path in G.

It also outputs a data structure $\overleftarrow{\Pi}$ storing:

- For every vertex v all those paths found intersecting v, queriable as $\overleftarrow{\Pi}(v)$
- For every vertex pair $(c,t), c \in C$ a h_1-hop restricted C-improving (c,t)-path in $\overleftarrow{G}$.

[1] That is, no longer (but possibly shorter) than the h_1-restricted shortest path in $G \setminus C$

The preprocessing algorithm has two stages. In the first stage all vertices are processed, and we find a congested set C'. In the second stage paths intersecting congested vertices are handled. The first stage starts out with an empty congested set C. While there are unprocessed vertices in V, vertices from V are processed one by one in an arbitrary order.

To process a vertex v, outgoing h-hop restricted shortest paths are computed using the Bellman-Ford algorithm, and the found paths are put into a temporary data structure Paths(v). For each path found, all vertices v on the path receive congestion equal the in-degree of the path endpoint, and a pointer from $\Pi(v)$ to the path.

After the processing of a vertex, if any vertices have exceeded the congestion threshold τ, they are put in the congested set C'.

After the processing of V is done, the congested set needs to be processed. This is stage 2 of the algorithm. First, congestion is reset and a new congestion scheme is instantiated. We thereafter alternate between visiting a maximally congested vertex in the prior congested set C' and the most congested vertex in $V \setminus (C \cup C')$, and call this vertex c. Upon visiting c we find in- and outgoing h_1-hop restricted shortest paths in $G \setminus C$ to/from c, congesting these and adding them to Π and $\overleftarrow{\Pi}$. Thereafter c is added to C.

The batch deletion algorithm is very simple. The outgoing paths from every vertex in V, and additionally the ingoing paths for vertices in C must be ACK-repaired, as described in Chapter 2.4.

Algorithm 8: `Visit`$(v, d, \Pi, \overleftarrow{\Pi}, G, \text{Cong})$

```
1  OutPaths(c) ← Bellman-Ford(c, G, d)
2  InPaths(c) ← Bellman-Ford(c, ←G, d)
3  foreach π_{c,t} ∈ OutPaths(c) do
4  |   Π(c,t) ← π_{c,t}
5  |   foreach w on π_{c,t} do
6  |   |   Add π_{c,t} to Π(w)
7  |   |   Cong(w) ← Cong(w) + deg_in(t)
8  |   end
9  end
10 foreach π_{c,t} ∈ InPaths(c) do
11 |   ←Π(c,t) ← π_{c,t}
12 |   foreach w on π_{c,t} do
13 |   |   Add π_{c,t} to ←Π(w)
14 |   |   Cong(w) ← Cong(w) + deg_in(t)
15 |   end
16 end
```

Algorithm 9: Preprocessing **low hop paths**

$C' \leftarrow \emptyset$ ▷ Stage 1
$\forall v, \text{Cong}(v) \leftarrow 0$ ▷ Congestion is initialized.
while *Unprocessed vertex left in* V **do**
  $\text{Paths}(v) \leftarrow$ Bellman-Ford$(v, G \setminus C', h_1)$
  foreach $\pi_{v,t} \in$ *Paths*(v) **do**
    $\Pi(v,t) \leftarrow \pi_{v,t}$;
    foreach *w on* $\pi_{v,t}$ **do**
      Add $\pi_{v,t}$ to $\Pi(w)$;
      $\text{Cong}(w) \leftarrow \text{Cong}(w) + \deg_{in}(t)$;
    end
  end
  if *Some vertices in* $V \setminus C'$ *have congestion* $> \tau$ **then**
    Add these vertices to C'.
  end
end
$\forall v, \text{Cong}(v) \leftarrow 0$ ▷ Congestion is reset.
$C \leftarrow \emptyset$
while *C' not empty* **do**
  Every other round let c be the most congested vertex from C' and every other time the most congested vertex in $V/(C \cup C')$.
  `Visit`$(c, h_1, \Pi, \overleftarrow{\Pi}, G \setminus C, \text{Cong})$
  $C \leftarrow C \cup c$
end

Algorithm 10: Batch Deletion **low hop paths**

foreach $v \in V$ **do**
  Do ACK-repair(v, D, G, Π)
end
foreach $c \in C$ **do**
  Do ACK-repair(c, D, $\overleftarrow{G}$, $\overleftarrow{\Pi}$)
end

Correctness

The proof of correctness is divided into 2 parts: Proving correctness for paths intersecting C and paths not intersecting C.

Definition: Let C^i be the set of the first i vertices added to C. Let c_i be the i'th vertex added to C.

LEMMA 5.3.1 A shortest (s,t)-path π of hop less than h, going through some vertex $c_j \in C$ is going to be found using the algorithm.

Proof. Take the c_i on π of lowest index. Since π is shortest, the subpaths $s \to c_i$ and $c_i \to t$ are also shortest paths and especially h-hop-restricted shortest paths. They are furthermore shortest paths in $G \setminus C^{i-1}$. This means that by the correctness of the Bellman-Ford algorithm and ACK-repair, after batch deletion is completed these paths will be in Π. □

LEMMA 5.3.2 If the shortest (s,t)-path in $G \setminus D$ is h-hop-restricted and does not intersect C, then it is in Π.

Proof. $\Pi(s,t)$ must from definition be C-improving, as it is found with h-hop restricted Bellman-Ford in $G \setminus C^i$ for some i. Therefore, if it is not broken by deletions, and the shortest path after deletions is π, then $\Pi(s,t) = \pi$.

If however $\Pi(s,t)$ is broken, π must be found, as ACK-repair is guaranteed to find the shortest path in $G \setminus C_i$ if it has hop $\leq h$. As $\pi \subset G \setminus C_i$ and $|\pi| \leq h$ the sufficient conditions are met. □

Time Complexity

Preprocessing. The regular processing of one vertex requires running Bellman-Ford, as well as entering line 6 potentially nh times. Calls to `Visit` applied to vertices from C requires these same things to be done twice.

The time required for running Bellman-Ford clearly dominates, and takes $O(mnh_1)$ time, as bellman ford is run at most 3 times from every vertex. Over Δ rounds this can be deamortized to $O(mnh/\Delta_1)$.

Batch Deletion. The time complexity from the batch deletion comes from ACK-repair calls, and must be bounded by the total congestion of deleted vertices.

LEMMA 5.3.3 No vertex is more than 2τ congested during stage 1 in the preprocessing algorithm.

Proof. As any vertex can only amass at most m units of congestion per processed vertex, it can only have as much as $\tau + m < 2\tau$ units of congestion before being put in C', after which it receives no congestion. □

LEMMA 5.3.4 No vertex is more than $4\log(n)mh_1$ congested during stage 2 in the algorithm.

Proof. At all times in the algorithm, after at most two calls to `Visit` the most congested vertex is put into C. As two calls to `Visit` can distribute only $4mh_1$ units of congestion, by the stone and pile theorem we have the desired upper bound. □

As we can bound the time complexity from repairing paths from vertices not in C with the first congestion scheme, valid until row 14 in the pseudocode, and likewise, the time complexity from repairing in- and outgoing paths to vertices in C by the second congestion scheme we know that the batch deletion time complexity is $O((\tau + mh_1 \log(n))\Delta_1 = O(\tau\Delta_1)$, where we can remove the term $mh_1 \log(n)$ as it will be chosen much smaller than τ.

Query. The query time comes from having to iterate through C to find the shortest h_1-hop restricted path that passes through C. To bound the size of C we use that $|C| = 2|C'|$, and bound the size of C'.

As at most mnh_1 congestion is distributed in stage 1 of the algorithm, at most mnh/τ vertices can have congestion greater than τ. This means the query time for low hop paths is bounded by $O(nmh_1/\tau)$.

5.4 Mid Hop Paths

Algorithm Description

In this part of the algorithm we build a data structure in multiple layers, almost identical to the one described in [Abraham et al., 2017]. Every layer has the responsibility for a range of hops $[2^i, 2^{i+1}]$, where $i \in [\lfloor \log(h_1) \rfloor, \lfloor \log(h_2) \rfloor]$. For each such i we define $d_i = 2^i$. After this we take a random subset $S_i \subset V$, where every vertex from V is included independently with probability $10\log(n)/d_i$ to get a set intersecting all shortest paths of hop at least d_i **whp**.

Every layer has a set C_i, which starts out empty. The preprocessing algorithm alternates between processing the most congested vertex from S_i and processing the most congested vertex of $V \setminus (C_i \cup S_i)$. These vertices are processed as follows in the method `Visit`: Ingoing and outgoing $2d_i$-hop restricted paths are found to/from v in $G \setminus C_i$, and put into the data structures Π_i and $\overleftarrow{\Pi}_i$. v is put into C_i, as to not be used any further. For a path $\pi_{s,t}$ with an endpoint t, $deg(t)$ units of congestion are added to every $w \in \pi$. For all $w \in \pi$, $\Pi_i(w)$ also gets a pointer to π.

The batch deletion algorithm is fairly simple. Every set C_i must be iterated through, and the in- and outgoing paths must be repaired using ACK-repair.

Algorithm 11: Preprocessing **mid hop paths**

for $i = \lfloor \log(h_1) \rfloor$ **to** $\lfloor \log(h_2) \rfloor$ **do**
  $\forall v, \text{Cong}(v) \leftarrow 0$ ▷ Congestion is initialized.
  $d_i \leftarrow 2^i$
  $C_i \leftarrow \emptyset$
  Randomly sample S_i from V with probability $\frac{10\log(n)}{d_i}$
  while S_i *not empty* **do**
    $v \leftarrow$ most congested vertex in S
    $S \leftarrow S \setminus \{v\}$
    $\texttt{Visit}(v, 2d_i, \Pi_i, \overleftarrow{\Pi}_i, G \setminus C_i, \text{Cong}_i)$
    $C_i \leftarrow C_i \cup \{v\}$
    $c \leftarrow$ most congested vertex in $V \setminus (C_i \cup S)$
    $\texttt{Visit}(c, 2d_i, \Pi_i, \overleftarrow{\Pi}_i, G \setminus C_i, \text{Cong}_i)$
    $C_i \leftarrow C_i \cup \{c\}$
  end
end

Algorithm 12: Batch Deletion **mid hop paths**

for $i = \lfloor \log(h_1) \rfloor$ **to** $\lfloor \log(h_2) \rfloor$ **do**
  foreach $c \in C_i$ **do**
    ACK-repair(c, D, G, Π_i)
    ACK-repair(c, D, $\overleftarrow{G}$, $\overleftarrow{\Pi}_i$)
  end
end

Correctness

LEMMA 5.4.1 Let π be a shortest (s,t)-path in the graph $G \setminus D$, of hop $k \in [2^i, 2^{i+1}]$, where $h_1 \leq k \leq h_2$. For some $c \in C_i$, $c \in \pi$ **and** $\overleftarrow{\Pi}_i(c,s) \circ \Pi_i(c,t) = \pi$.

Proof. Because the vertices of S_i is chosen independently uniformly at random, using a sufficiently high probability, **whp** at least one of the vertices on the path is in $S_i \subset C_i$.

Let c be the vertex in $\pi \cap C_i$ that was first added into C_i. Let C' be the set of vertices present in C_i before c. Then π must be a shortest path in $G \setminus (D \cup C')$, hence in $G \setminus (D \cup C')$, the (s, c) subpath and (c, t) subpath are both shortest paths of hop $\leq 2^{i+1}$.After ACK-repair we know that the paths stored in Π are at least as short as 2^{i+1}-hop restricted shortest paths in $G \setminus (D \cup C')$, see Lemma 2.4.2. Therefore $\overleftarrow{\Pi}_i(c,s) \circ \Pi_i(c,t) = \pi$. □

Time Complexity

Preprocessing. As for the preprocessing of low hop paths, the time complexity once again is dominated by Bellman-Ford calls, since these are the most expensive procedure in the algorithm, as well as being performed most often, exactly $2|C_i|$ times.

$|C_i| \leq 2|S_i| = \tilde{O}(n/d_i)$ **whp**. For each level i, 2^{i+1} hop Bellman-Ford is performed from $\tilde{O}(n/2^i)$ source vertices, leading to a time complexity of $\tilde{O}(nm)$ for each i. Since we look at fewer than $\log(n)$ different i, this leads to (deamortized over Δ_2 rounds) time complexity $\tilde{O}(nm/\Delta_2)$.

Batch Deletion. The time required for ACK-repair can be bounded by $O(\sum_{d \in D} \text{Congestion}(d))$. As every layer has its own congestion framework we are interested in the layer with the highest congestion values. The highest congestion values, and therefore, the highest batch deletion time complexity will appear in the layer with $i = \lfloor \log(h_2) \rfloor$.

The congestion added to the graph per round in the for loop on line 4 is $O(mh_2)$, this means that vertices will be congested at most $\tilde{O}(mh_2)$. This means that ACK-repair will take time at most $\tilde{O}(mh_2\Delta_2)$ at that layer. Since the congestion increases exponentially with i, the total time complexity can be modelled as a geometric sum, which is bounded by a constant times its largest term. This gives time complexity for all layers $\tilde{O}(mh_2\Delta_2)$.

Query. We can bound the query time complexity as we know that it is proportional to the sum of the sizes of the via-point sets C_i. The hitting sets decrease in size exponentially with i, and therefore in the same way as for the update time the sum will be a geometric series, this time with largest term $|C_{\lfloor \log(h_1) \rfloor}|$. This gives a time complexity of $\tilde{O}(n/h_1)$ **whp**.

5.5 High Hop Paths

As shortest paths of hop $\leq h_2$ are found by the two other parts of the algorithm, we can use a hitting set of size $\tilde{O}(n/h_2)$. Here we simply use the standard hitting set trick detailed in Section 2.4. This leads to update time complexity:$\tilde{O}(nm/h_2)$, and query time complexity:$\tilde{O}(n/h_2)$ **whp**.

5.6 Vertex Insertions

Vertex insertions are handled in the usual fashion. The vertices inserted since the last recomputation (of the oldest data structure) are stored in a set I. after each update, in-and outgoing shortest paths are found to/from this set using Dijkstra's

algorithm. This gives update takes time bounded by $\tilde{O}(max(\Delta_1, \Delta_2)m)$, and query time bounded by $O(\max(\Delta_1, \Delta_2))$.

5.7 Negative Edge Weights

The algorithm can be made to handle negative edge weights using a technique presented in (Theorem 23 of) [Karczmarz, 2021] to dynamically maintain whether there is a negative cycle, and a price function in $\tilde{O}(m)$ time. This means that the rest of the algorithm can work as through there are no negative weight edges. The additional update time complexity of $\tilde{O}(m)$ is wholly inconsequential.

5.8 Space Complexity

To reach the time complexity stated in Theorem 5.1 we can use a trick from [Gutenberg and Wulff-Nilsen, 2019].

The high hop paths require no storage, as we can resample H in each update. The mid hop data structure requires us for each level i to store $\tilde{O}(n^2/2^i)$ paths of hop $O(2^i)$, leading trivially to a $\tilde{O}(n^2)$ space bound for all levels. The short path data structure consists of $O(n^2)$ paths of hop at most h_1. Without a trick to compress this information this would lead to a space complexity of $\tilde{O}(n^2 h_1)$.

[Gutenberg and Wulff-Nilsen, 2019] gives us a sub-routine $\texttt{BELLMANFORDSPACEEFFICIENT}(s, G, h)$ with the same asymptotic time complexity as h-round Bellman-Ford, that finds paths of length $O(h)$, no longer than the h-hop restricted shortest paths, and has an output that can be stored in $\tilde{O}(n)$ space. As the output paths have hop $O(h_1)$, we know that the total congestion only increases by a constant factor. Therefore Lemma 5.2.3 is not affected in an asymptotic sense.

Furthermore, length and hop of paths are queryable in $O(1)$, finding the paths intersecting a specific vertex (needed for constructing $\Pi(v)$) is linear in the number of intersecting paths, and reporting edges on a path π takes time $O(|\pi|)$. This means that the asymptotic time complexity does not change from using Bellman-Ford's algorithm, while the space needed for the $O(n)$ single source hop restricted shortest path outputs can be bounded with $\tilde{O}(n^2)$.

5.9 Total Time Complexity And Parameter Balancing

Taking the update and query time complexity from all 3 parts of the algorithm the total time complexity becomes:

$$\text{Update: } \tilde{O}(nmh_1/\Delta_1 + \tau\Delta_1 + mn/\Delta_2 + mh_2\Delta_2 + nm/h_2 + \max(\Delta_1, \Delta_2)m)$$

$$\text{Query: } \tilde{O}(nmh_1/\tau + n/h_1 + n/h_2 + \max(\Delta_1, \Delta_2) = \tilde{O}(nmh_1/\tau + n/h_1 + \Delta_1 + \Delta_2)$$

Noticeable is that the choice of h_2 and Δ_2, as long as $h_2 > h_1$ and $\Delta_2 < \Delta_1$ does not at all impact the query time. This means that h_2 and Δ_2 can be chosen only to optimize update time. Ignoring polylog-factors, the ideal choice is $h_2 = \Delta_2 = n^{1/3}$, which makes the terms to which these variables contribute $\tilde{O}(mn^{2/3})$.

h_1, Δ_1 and τ, however, decide the trade-off between update- and query time. Wanting a query time $\tilde{O}(t)$, we may choose $h_1 = n/t, \Delta = t^{1/2}, \tau = mn^2/t^2$, which lead to an update time of $\tilde{O}(mn^2/t^{3/2} + mn^{2/3})$, proving Theorem 5.1.

Using $h_1 = n^{1/5}, \Delta_1 = n^{2/5}, \tau = mn^{2/5}$ this gives update time complexity $\tilde{O}(mn^{4/5})$ and query time complexity $\tilde{O}(n^{4/5})$.

The trade-off between query and update time is better than in [Karczmarz and Sankowski, 2023] as $h_1 = n^{1/9}, \Delta_1 = n^{4/9}, \tau = mn^{2/9}$ gives update time complexity $\tilde{O}(mn^{2/3})$ and query time complexity $\tilde{O}(n^{8/9})$ which is unattainable using [Karczmarz and Sankowski, 2023]. It is however impossible to attain lower update time than $mn^{2/3}$, as this is decided by the terms only depending on h_2 and Δ_2, and optimal for these.

An aspect that was not commented upon by [Karczmarz and Sankowski, 2023], is that at the cost of longer update time one can gain a shorter query time, something that was already possible using their algorithm as it was presented. This aspect is not improved in our novel algorithm, but allows for query times as low as $\tilde{O}(n^{2/3})$ before the update time reaches the trivial $\tilde{O}(mn)$.

5.10 Getting Past Graph Size Upper Bounds

Letting the adversary decide the upper bounds n and m instead of allowing arbitrary series of vertex updates makes the problem a bit tidier. One can however make minor changes in the algorithm to let time complexities be dependent on the changing quantities $|V|$ and $|E|$. Effectively, the update and query times are dependent on the size of V and E at the time where the active preprocessings are initialized. This means that the situation which needs guarding against is that V or E shrink dramatically compared to the vertex and edge sets used in the active preprocessing.

Since $\Delta_1, \Delta_2 \ll n$, for $|V|$ this is not a problem. A sparse graph can however quickly lose a substantial amount of edges. This can be handled as following: Before preprocessing a data structure that is recomputed after Δ rounds, one removes the Δ highest degree vertices (greedily). These are handled as if they were inserted after

preprocessing, which is no problem asymptotically as it only doubles the maximal number of inserted vertices in a phase. That way, the edge set of the dynamic graph is always greater than the edge set used during preprocessing.

6

Ranges of Optimality in Practical Use Cases

Clearly, there are many different algorithms that solve fully dynamic APSP. As there are many algorithms that may be used, it is of interest for practitioners to know which work optimally, dependent both on properties of the graph and characteristics of potential use cases. We will evaluate based on sparsity and query intensity which algorithm works best asymptotically.

Assuming a dynamic algorithm is used upon a graph with n vertices, $m \in [n, n^2]$ edges, and $q \in [1, n^2]$ queries per update. We will study our novel algorithm against [Mao, 2024] as it is the fastest fully dynamic algorithm in the worst case. These more sophisticated algorithms are also compared with the trivial approaches. Namely, against running Dijkstra's algorithm for each query which might be optimal in cases with sparse graphs and few queries, and also against running Dijkstra's algorithm from every source upon update, which might be better in situations with many queries.

The time complexity for [Mao, 2024] is easy to analyze. As distance queries take $O(1)$ time, the algorithm needs $\tilde{O}(n^{5/2})$ per update, irrespective of q.

Naive Dijkstra calls are also relatively easy to analyze. They require no update time and a query time of $\tilde{O}(m)$, meaning that the time required per update is $\tilde{O}(mq)$. The only pitfall here is that Dijkstra calls from the same source can be used again for other end points. This model therefore assumes that no source is used twice in the queries.

Simply running Dijkstra's algorithm for every source as an update algorithm has update time $\tilde{O}(nm)$ and query time $O(1)$.

Analyzing our novel algorithm means finding the right $t \leq n^{8/9}$ to minimize

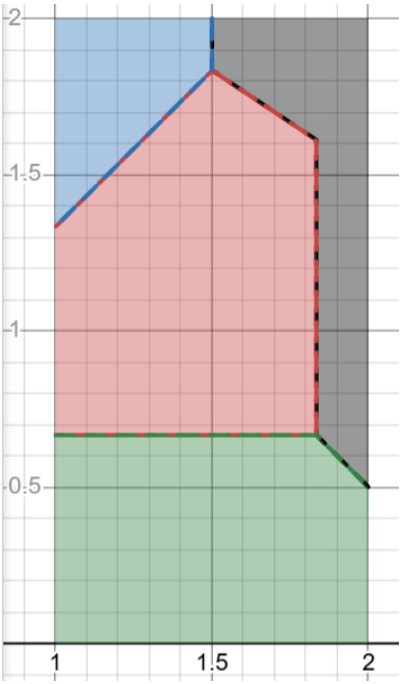

Figure 6.1 On vertical axis is $\log_n(q)$ and on horizontal axis is $\log_n(m)$. Querying through Dijkstra is optimal in the green area, updating with Dijkstra is optimal in blue area, Mao's method is optimal in grey area, and our novel algorithm is optimal in red area.

$mn^{2/3} + mn^2/t^{3/2} + tq$. This is $t = m^{2/5}n^{4/5}q^{-2/5}$, leading to a time per update $\tilde{O}(m^{2/5}n^{4/5}q^{3/5} + mn^{2/3} + qn^{2/3})$.

The range of asymptotic optimality for repeated Dijkstra calls is simply $q < n^{2/3} \wedge qm < n^{5/2}$. The range of optimality for [Mao, 2024] is $m > n^{11/6} \wedge qm > n^{5/2}$. The novel algorithm is optimal in the interval $(q > n^{2/3} \wedge m < n^{11/6}) \vee (m > n^{3/2} \wedge m^4q^6 > n^{17})$. The rest of the area $(m,q) \in [1,n^2] \times [1,n^2]$ our novel algorithm performs asymptotically best.

As the inequalities in the previous paragraph are somewhat difficult to get a feeling for, figure 6.1 shows the regions of optimality in a logarithmic sense. As one would expect, the novel algorithm performs best when queries are many (but not too many) and the graph is sparse.

7

Discussion and Conclusions

There are 3 reasons why we failed to turn [Mao, 2024] into a sparse algorithm with update time $mn^{1/2}$ query time $n^{1/2}$:

- We lose once in preprocessing because we cannot handle via-points necessary to have layered congested sets, leading to larger sets to query through.
- We lose again in preprocessing because of via-points for the high hop paths needed for the efficient preprocessing algorithm.
- We lose during repair because the GWN-repair method takes n/h time instead of degree(endpoint) time, which is strictly better in dense graphs.

The two first we inherently cannot do anything about. The last one we do not want to do anything about. After all, our algorithm still works faster in sparse graphs than Mao's does. As an additional point, even without all the trouble with lemma 3.2.1, ACK-repair cannot in a meaningful way be applied to the paths output by RestrictedDijkstra.

The novel algorithm described in chapter 5 manages to improve upon [Karczmarz and Sankowski, 2023] because it is able to truly free the update time from the $\tilde{O}(nm/h)$, that come when one only handles h-hop restricted shortest paths and high hop paths. Although our novel algorithm gives a significantly better Update-Query time trade-off than [Karczmarz and Sankowski, 2023], it is not clear whether there could be a better trade-off possible.

The novel algorithm could not lower the update time past $mn^{2/3}$, no matter the query time. This could be connected to an impossibility of getting an update time below $n^{2+2/3}$ without leaving hop restricted shortest paths. Just finding single source $h-$hop restricted shortest paths already gives $O(nmh/\Delta + nm/h)$ update time, which to get $mn^{1/2}$ forces $\Delta = O(n)$, which by allowing a total reshaping of

the graph makes this very unlikely to work.

The way past update time $mn^{4/5}$, and query time $n^{4/5}$ entails actually finding new techniques. The path to a new and better algorithm could fathomably go though either a faster preprocessing method or through a new sparsity-compatible repair method. As Bellman-Ford is proven optimal for finding hop-restricted shortest paths, a faster preprocessing would reasonably require some other object than hop restricted shortest paths to be used. As shown in this thesis, hop dominant paths are probably not the solution, but there might be other combinatorial properties to graphs that can be exploited.

One must also remember that it is possible that no algorithm can solve dynamic sparse APSP faster with simultaneously query time lower than $\tilde{O}(n^{4/5})$ and update time lower than $\tilde{O}(mn^{4/5})$. Therefore, further conditional lower bounds could also be a possible direction of further research.

Bibliography

Abraham, I., S. Chechik, and S. Krinninger (2017). "Fully dynamic all-pairs shortest paths with worst-case update-time revisited". en. In: *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 440–452. DOI: 10.1137/1.9781611974782.28. URL: https://epubs.siam.org/doi/10.1137/1.9781611974782.28 (visited on 2025-01-29).

Chechik, S. and T. Zhang (2023). "Faster Deterministic Worst-Case Fully Dynamic All-Pairs Shortest Paths via Decremental Hop-Restricted Shortest Paths". In: *Proceedings of the 2023 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. Proceedings. Society for Industrial and Applied Mathematics, pp. 87–99. DOI: 10.1137/1.9781611977554.ch4. URL: https://epubs.siam.org/doi/10.1137/1.9781611977554.ch4 (visited on 2025-02-06).

Demetrescu, C. and G. F. Italiano (2004). "A new approach to dynamic all pairs shortest paths". *J. ACM* **51**:6, pp. 968–992. ISSN: 0004-5411. DOI: 10.1145/1039488.1039492. URL: https://dl.acm.org/doi/10.1145/1039488.1039492 (visited on 2025-05-13).

Dijkstra, E. W. (1959). "A note on two problems in connexion with graphs". *Numerische Mathematik* **1**:1, pp. 269–271. ISSN: 0945-3245. DOI: 10.1007/BF01386390. URL: https://doi.org/10.1007/BF01386390.

Fredman, L. and R. E. Tarjan (2001). "Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms". en. *Journal ofthe Association for Computing Machinery* **34**:3. DOI: 10.1007/BF00299635. URL: https://link.springer.com/article/10.1007/BF00299635 (visited on 2025-05-12).

Gutenberg, M. P. and C. Wulff-Nilsen (2019). "Fully-Dynamic All-Pairs Shortest Paths: Improved Worst-Case Time and Space Bounds". In: *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms (SODA)*. Proceedings. Society for Industrial and Applied Mathematics, pp. 2562–2574. DOI: 10.1137/1.9781611975994.156. URL: https://epubs.siam.org/doi/10.1137/1.9781611975994.156 (visited on 2025-05-16).

Hagerup, T. and C. Rüb (1990). "A guided tour of chernoff bounds". *Information Processing Letters* **33**:6, pp. 305–308. ISSN: 0020-0190. DOI: 10.1016/0020-0190(90)90214-I. URL: https://www.sciencedirect.com/science/article/pii/002001909090214I (visited on 2025-05-27).

Henzinger, M. R. and V. King (2001). "Maintaining Minimum Spanning Forests in Dynamic Graphs". en. *SIAM Journal on Computing* **31**:2, pp. 364–374. ISSN: 0097-5397, 1095-7111. DOI: 10.1137/S0097539797327209. URL: http://epubs.siam.org/doi/10.1137/S0097539797327209 (visited on 2025-05-12).

Johnson, D. B. (1977). "Efficient Algorithms for Shortest Paths in Sparse Networks". *J. ACM* **24**:1, pp. 1–13. ISSN: 0004-5411. DOI: 10.1145/321992.321993. URL: https://dl.acm.org/doi/10.1145/321992.321993 (visited on 2025-05-13).

Karczmarz, A. (2021). "Fully Dynamic Algorithms for Minimum Weight Cycle and Related Problems". In: Bansal, N. et al. (Eds.). *48th International Colloquium on Automata, Languages, and Programming (ICALP 2021)*. Vol. 198. Leibniz International Proceedings in Informatics (LIPIcs). ISSN: 1868-8969. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 83:1–83:20. ISBN: 978-3-95977-195-5. DOI: 10.4230/LIPIcs.ICALP.2021.83. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ICALP.2021.83 (visited on 2025-05-16).

Karczmarz, A. and P. Sankowski (2023). "Fully Dynamic Shortest Paths in Sparse Digraphs". *LIPIcs, Volume 261, ICALP 2023* **261**. arXiv:2408.14406 [cs], 84:1–84:20. ISSN: 1868-8969. DOI: 10.4230/LIPICS.ICALP.2023.84. URL: http://arxiv.org/abs/2408.14406 (visited on 2025-02-06).

King, V. (1999). "Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs". In: *40th Annual Symposium on Foundations of Computer Science (Cat. No.99CB37039)*. ISSN: 0272-5428, pp. 81–89. DOI: 10.1109/SFFCS.1999.814580. URL: https://ieeexplore.ieee.org/document/814580 (visited on 2025-05-13).

Kociumaka, T. and A. Polak (2023). "Bellman-Ford Is Optimal for Shortest Hop-Bounded Paths". In: Gørtz, I. L. et al. (Eds.). *31st Annual European Symposium on Algorithms (ESA 2023)*. Vol. 274. Leibniz International Proceedings in Informatics (LIPIcs). ISSN: 1868-8969. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 72:1–72:10. ISBN: 978-3-95977-295-2. DOI: 10.4230/LIPIcs.ESA.2023.72. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ESA.2023.72 (visited on 2025-05-16).

Levcopoulos, C. and M. H. Overmars (1988). "A balanced search tree with O(1) worst-case update time". en. *Acta Informatica* **26**:3, pp. 269–277. ISSN: 1432-

0525. DOI: 10.1007/BF00299635. URL: https://doi.org/10.1007/BF00299635 (visited on 2025-05-12).

Mao, X. (2024). "Fully Dynamic All-Pairs Shortest Paths: Likely Optimal Worst-Case Update Time". In: *Proceedings of the 56th Annual ACM Symposium on Theory of Computing*. STOC 2024. Association for Computing Machinery, New York, NY, USA, pp. 1141–1152. ISBN: 979-8-4007-0383-6. DOI: 10.1145/3618260.3649695. URL: https://dl.acm.org/doi/10.1145/3618260.3649695 (visited on 2025-05-16).

Thorup, M. (2005). "Worst-case update times for fully-dynamic all-pairs shortest paths". en. In: *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*. ACM, Baltimore MD USA, pp. 112–119. ISBN: 978-1-58113-960-0. DOI: 10.1145/1060590.1060607. URL: https://dl.acm.org/doi/10.1145/1060590.1060607 (visited on 2025-05-12).

Ullman, J. D. and M. Yannakakis (1991). "HIGH-PROBABILITY PARALLEL TRANSITIVE- CLOSURE ALGORITHMS". en. *SIAM Journal on Computing* **20**:1, pp. 100–125. DOI: 10.1145/97444.97686. URL: https://doi.org/10.1145/97444.97686.

Lund University
Department of Automatic Control
Box 118
SE-221 00 Lund Sweden

Document name
MASTER'S THESIS

Date of issue
June 2025

Document Number
TFRT-6275

Author(s)
Lennart Küssner

Supervisor
Giacomo Como, Dept. of Automatic Control, Lund University, Sweden
Emma Tegling, Dept. of Automatic Control, Lund University, Sweden (examiner)

Title and subtitle
On Fully Dynamic All Pairs Shortest Path in Sparse Digraphs

Abstract

The All Pairs Shortest Path (APSP) problem is ubiquitous in computer science. The problem consists of finding for any vertex pair, the weighted shortest paths in a network with real-weighted edges. The problem has been studied thoroughly, both for dense graphs, where the number of edges can be quadratic in the number of vertices, and for sparse graphs, where the number of edges is considerably lower than so.
Many real world networks are subject to small changes over time. For instance, road work or traffic accidents can considerably change the shortest paths through a road network. As most real world networks are also sparse, this motivates the study of the sparse fully dynamic APSP problem, where a data stucture is maintained for a sparse graph subject to vertex insertions and deletions, from which the shortest path between any pair of vertices should be quickly queryable.
This thesis comes with two main results: Firstly, the key methods that led to the improvement in the dense case have been studied to determine whether they can be applied in the sparse case. Conclusively it has been shown that the techniques studied rely on subroutines that iterate through all vertex pairs, with no real possibility to change this. Therefore, they cannot be transferred to the sparse case to give an algorithm with update time subquadratic in the number of vertices. Given this, no improvement over the state of the art in the sparse case is possible using the new dense methods. Secondly, a novel algorithm with a trade-off between the update and query time of ˜O (mn2/t3/2), ˜O(t) for t ∈ [n2/3,n8/9] is presented, making it possible to reach an update time of ˜O(mn2/3), improving over the state of the art.

Keywords

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title
0280-5316

ISBN

Language
English

Number of pages
1-65

Recipient's notes

Security classification

http://www.control.lth.se/publications/