Maximum Bipartite Matching in $n^{2+o(1)}$ Time via a Combinatorial Algorithm

Julia Chuzhoy*
Toyota Technological Institute at Chicago
Chicago, USA
cjulia@ttic.edu

Sanjeev Khanna†
University of Pennsylvania
Philadelphia, USA
sanjeev@cis.upnn.edu

ABSTRACT

Maximum bipartite matching (MBM) is a fundamental problem in combinatorial optimization with a long and rich history. A classic result of Hopcroft and Karp (1973) provides an $O(m\sqrt{n})$-time algorithm for the problem, where n and m are the number of vertices and edges in the input graph, respectively. For dense graphs, an approach based on fast matrix multiplication achieves a running time of $O(n^{2.371})$. For several decades, these results represented state-of-the-art algorithms, until, in 2013, Madry introduced a powerful new approach for solving MBM using continuous optimization techniques. This line of research, that builds on continuous techniques based on interior-point methods, led to several spectacular results, culminating in a breakthrough $m^{1+o(1)}$-time algorithm for min-cost flow, that implies an $m^{1+o(1)}$-time algorithm for MBM as well.

These striking advances naturally raise the question of whether combinatorial algorithms can match the performance of the algorithms that are based on continuous techniques for MBM. One reason to explore combinatorial algorithms is that they are often more transparent than their continuous counterparts, and that the tools and techniques developed for such algorithms may be useful in other settings, including, for example, developing faster algorithms for maximum matching in general graphs. A recent work of Chuzhoy and Khanna (2024) made progress on this question by giving a combinatorial $\tilde{O}(m^{1/3}n^{5/3})$-time algorithm for MBM, thus outperforming both the Hopcroft-Karp algorithm and matrix multiplication based approaches, on sufficiently dense graphs. Still, a large gap remains between the running time of their algorithm and the almost linear-time achievable by algorithms based on continuous techniques. In this work, we take another step towards narrowing this gap, and present a randomized $n^{2+o(1)}$-time combinatorial algorithm for MBM. Thus in dense graphs, our algorithm essentially matches the performance of algorithms that are based on continuous methods.

Similar to the classical algorithms for MBM and the approach used in the work of Chuzhoy and Khanna (2024), our algorithm is based on iterative augmentation of a current matching using augmenting paths in the corresponding (directed) residual flow network. Our main contribution is a recursive algorithm that exploits the special structure of the resulting flow problem to recover an $\Omega(1/\log^2 n)$-fraction of the remaining augmentations in $n^{2+o(1)}$ time.

Finally, we obtain a randomized $n^{2+o(1)}$-time algorithm for maximum vertex-capacitated s-t flow in directed graphs when all vertex capacities are identical, using a standard reduction from this problem to MBM.

CCS CONCEPTS

• **Theory of computation** → **Graph algorithms analysis**; **Dynamic graph algorithms**; **Network flows**.

KEYWORDS

Bipartite matching, Vertex-capacitated flows

ACM Reference Format:
Julia Chuzhoy and Sanjeev Khanna. 2024. Maximum Bipartite Matching in $n^{2+o(1)}$ Time via a Combinatorial Algorithm. In *Proceedings of the 56th Annual ACM Symposium on Theory of Computing (STOC '24), June 24–28, 2024, Vancouver, BC, Canada.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3618260.3649725

1 INTRODUCTION

We consider the classical Maximum Bipartite Matching problem, where the goal is to compute a maximum-size matching in the given input bipartite graph G. Maximum Bipartite Matching is one of the most central and extensively studied problems in computer science and related disciplines, with connections to many other fundamental graph optimization problems.

Throughout, we denote the number of vertices and the number of edges in G by n and m, respectively. It is well known that Maximum Bipartite Matching can be reduced to computing a maximum s-t flow in a directed flow network with unit edge capacities. The Ford-Fulkerson algorithm [14] for maximum s-t flow then immediately implies an $O(mn)$-time algorithm for Maximum Bipartite Matching. The algorithm is conceptually simple, and maintains a matching M, starting with $M = \emptyset$. As long as M is not optimal, we can augment it by computing an s-t path in the resulting residual flow network. A celebrated work of Hopcroft and Karp [21] provides a significantly more efficient $O(m\sqrt{n})$-time implementation of this idea by iteratively computing a maximal collection of internally disjoint augmenting s-t paths of shortest possible length in

*Supported in part by NSF grant CCF-2006464 and NSF HDR TRIPODS award 2216899.
†Supported in part by NSF awards CCF-1934876 and CCF-2008305.

STOC '24, June 24–28, 2024, Vancouver, BC, Canada

ACM ISBN 979-8-4007-0383-6/24/06
https://doi.org/10.1145/3618260.3649725

the residual flow network. This result remained the fastest known algorithm for several decades, except for the special case of very dense graphs, where fast matrix multiplication techniques were shown to yield an $O(n^{\omega})$-time algorithm [22, 30]. Starting in 2008, a new paradigm emerged, namely, the use of continuous techniques as a method for obtaining fast algorithms for various flow problems, that ultimately revolutionized the field. As a first illustration of this paradigm, the work of Daitch and Spielman [11], building on the breakthrough result of Spielman and Teng [33] for efficiently solving Laplacian systems, gave an $\widetilde{O}(m^{3/2})$-time algorithm for directed maximum s-t flow. Later, Madry [27] used this paradigm to design an algorithm for directed maximum s-t flow with $\tilde{O}(m^{10/7})$ running time, obtaining the first substantial improvement over the algorithm of Hopcroft and Karp for Maximum Bipartite Matching in sparse graphs. A sequence of remarkable developments [4, 10, 24, 25, 28, 35] recently culminated in a deterministic $m^{1+o(1)}$-time algorithm for directed maximum s-t flow [8, 34], thereby providing an almost linear-time algorithm for Maximum Bipartite Matching. In all these recent algorithms, the directed flow problem is cast as a linear program, which is then solved via interior-point methods (IPM). In every iteration of the IPM, one needs to either solve a Laplacian system, or another efficiently solvable problem on undirected graphs, such as min-ratio cycle in [8]. This approach is further combined with dynamic graph data structures to make it even more efficient.

In view of this recent history, it is natural to ask whether combinatorial techniques can be used to design algorithms for Maximum Bipartite Matching (and also other flow-like problems), whose performance matches that of algorithms that are based on continuous methods. There are several reasons to focus on combinatorial techniques. First, they tend to be more transparent than their continuous counterparts. Second, it is likely that tools and techniques that are developed in order to design a combinatorial algorithm for as fundamental a problem as Maximum Bipartite Matching will prove useful in other applications. Lastly, while continuous techniques led to an $m^{1+o(1)}$-time algorithm for Maximum Bipartite Matching, the landscape of fast algorithms for the Maximum Matching problem in general graphs did not benefit from these developments. In dense graphs, a fast-matrix multiplication based approach gives $O(n^{2.371})$-time algorithm for Maximum Matching in general graphs [22, 30]. More interestingly, in sparse to moderately dense graphs, the best known runtime still stands on $\tilde{O}(m\sqrt{n})$ [15, 18, 29, 36] and utilizes an augmenting-paths based approach, similar to that used in combinatorial algorithms for Maximum Bipartite Matching.

In a very recent work, Chuzhoy and Khanna [9] made progress on narrowing the striking gap between the performance of combinatorial and IPM-based approaches for Maximum Bipartite Matching, by providing a combinatorial deterministic $\tilde{O}(m^{1/3}n^{5/3})$-time algorithm, thus outperforming both the Hopcroft-Karp algorithm, and the matrix multiplication based approaches on sufficiently dense graphs. Still, a large gap remains between the performance of the best combinatorial algorithms and the almost linear-time achievable by algorithms based on continuous techniques. In particular, on dense graphs, the performance gap incurred by the current best combinatorial algorithm is $\Omega(n^{1/3})$. In this work, we take another step towards narrowing this performance gap, and essentially eliminate it in dense graphs. Our main result is summarized below.

Theorem 1.1. *There is a randomized combinatorial algorithm for the* Maximum Bipartite Matching problem, that, given an n-vertex bipartite graph G, outputs a maximum matching M in G with probability at least $1 - 1/\text{poly}(n)$. The running time of the algorithm is $O\left(n^2 \cdot 2^{O(\sqrt{\log n}\cdot\log\log n)}\right)$.

Our algorithm outperforms the Hopcroft-Karp algorithm on graphs with $\omega(n^{1.5})$ edges, and in dense graphs, it essentially matches the performance of algorithms based on continuous techniques. Furthermore, in almost all edge density regimes, this algorithm outperforms the runtime achieved in [9].

Using a standard reduction from vertex-capacitated flow in directed graphs to Maximum Bipartite Matching (see Theorem 16.12 in [32], for instance), we also obtain a combinatorial algorithm with similar running time for maximum vertex-capacitated flow when all vertex capacities are identical.

Corollary 1.2. *There is a randomized combinatorial algorithm for the directed maximum s-t flow problem with uniform vertex capacities, that given an n-vertex directed graph G, outputs a maximum s-t flow with probability at least $1 - 1/\text{poly}(n)$. The running time of the algorithm is $O\left(n^2 \cdot 2^{O(\sqrt{\log n}\cdot\log\log n)}\right)$.*

Similarly to the classical algorithms for Maximum Bipartite Matching, our approach for proving Theorem 1.1 is based on iteratively augmenting a current matching using augmenting paths in the residual flow network. We employ the multiplicative weights update (MWU) framework, that effectively reduces the underlying flow problem to decremental single-source shortest paths (SSSP) in directed graphs, a connection first observed by Madry [26] and also used in [9]. As observed in [9], this reduction results in a *special case* of decremental SSSP that appears significantly easier than general decremental directed SSSP. Our main contribution is a recursive algorithm that exploits the special structure of the resulting flow problem to recover an $\Omega(1/\log^2 n)$-fraction of the remaining augmentations in $n^{2+o(1)}$ time. We abstract this task as a problem called RouteAndCut, where the input is a directed graph G, two disjoint sets A, B of its vertices with $|A| \leq |B|$, and two additional parameters $1 \leq \eta \leq \Delta$. The goal is to either compute a collection $\mathcal{P}$ of at least $\Omega(\Delta/\text{poly}\log n)$ paths that connects distinct vertices of A to distinct vertices of B with vertex-congestion at most η; or to output a cut that approximately certifies infeasibility of the desired routing. Our main result is a randomized algorithm for RouteAndCut, whose running time is bounded by $n^{1+o(1)} \cdot (n - |B|)$. It is worth highlighting that when $|B|$ is sufficiently large, this running time may be much smaller than $|E(G)|$. This performance gain for large sets B serves as a crucial building block for our $n^{2+o(1)}$-time algorithm.

As in the work of [9], the task of efficiently solving RouteAndCut in turn relies on an efficient algorithm for maintaining an expander in a dynamically changing graph, a problem that we refer to as MaintainCluster. One key contribution of our work is the introduction of a *parameterized version* of both these problems that allows us to use a bootstrapping approach in the design of our algorithm, where we exploit efficient algorithms for one problem to obtain efficient algorithms for the other problem and vice versa. Another key technical contribution is an efficient algorithm for a new problem that we introduce, called ConnectToCenters, whose goal is to efficiently maintain short paths from all vertices of a given graph G to a pre-specified collection of "center" vertices, even as G undergoes online updates. This problem may be viewed as a representative abstraction of a common paradigm used in many graph algorithms, in which an expander is embedded into the input graph, and all graph vertices are then routed to the vertices of the expander. As such, our algorithm for this problem may prove useful in other applications. Finally, another insight utilized by our algorithm is an explicit recognition of the fact that each iteration of the MWU algorithm leads to very specific kind of updates in the underlying SSSP instance, namely, well-behaved increases in the lengths of some edges. While these length increases can easily be simulated as edge deletions, a black-box simulation as an instance of decremental SSSP gives away some of the inherent algorithmic advantages offered by these special kind of updates that our algorithm exploits. We give a detailed overview of our algorithm and its comparison to the algorithm of [9] in the next subsection.

We conclude by noting that in addition to the conceptual simplicity of a combinatorial augmenting path based approach to solve Maximum Bipartite Matching, the techniques developed here for speeding up augmentations may also prove useful in obtaining faster algorithms for Maximum Matching in general graphs. We also believe that some of the technical tools that we introduce, such as an efficient algorithm for the ConnectToCenters problem that we describe in more detail below, are of independent interest.

1.1 Our Techniques

Our algorithm builds on and extends the techniques of [9], which, in turn, build on the algorithm of [6] for the directed decremental Single-Source Shortest Path (SSSP) problem. We start with a high-level overview of the algorithm of [9], and then provide the description of our improved algorithm.

It is well known that the Maximum Bipartite Matching problem in a graph G can be equivalently cast as the problem of computing an integral maximum s-t flow in a corresponding directed flow network G' with unit edge capacities. We can view any given matching M in G as defining an s-t flow f in G' of value $|M|$. We let $H = G'_f$ be the corresponding residual flow network, that we refer to as the *residual flow network corresponding to matching* M. We note that the residual flow network H has a special structure: namely, each vertex in H has in-degree 1 or out-degree 1. Therefore, if $\mathcal{P}$ is a collection of paths in H causing edge-congestion at most η, then the paths in $\mathcal{P}$ cause vertex-congestion at most η and vice versa. For all problems that we define below, we assume that their input graph also has this special structure. For convenience, we will focus on edge-congestion, and on edge-based cuts in such graphs. We also note that any directed graph can be converted into a graph with this special structure by replacing every vertex v with a pair v^+, v^- of new vertices, such that all edges that enter v become incident to v^-, and all edges leaving v become incident to v^+, and inserting the edge (v^-, v^+) into the graph.

The residual network H corresponding to a matching M contains $\Delta = \mathrm{OPT} - |M|$ edge-disjoint s-t paths, where OPT is the value of the maximum bipartite matching. Suppose now that we can design an algorithm that computes a collection $\mathcal{P}$ of $\Omega(\Delta/\mathrm{poly}\log n)$ s-t paths in H, that cause $O(\mathrm{poly}\log n)$ edge-congestion. Using standard methods, we can then efficiently recover a collection $\mathcal{P}'$ of $\Omega(\Delta/\mathrm{poly}\log n)$ edge-disjoint s-t paths in H, which can in turn be used in order to augment the current matching M, thereby obtaining a new matching M' of cardinality $|M| + \Omega(\Delta/\mathrm{poly}\log n)$. In other words, $\mathrm{OPT} - |M'| \leq (\mathrm{OPT} - |M|) \cdot (1 - 1/\mathrm{poly}\log n)$, so the gap between the optimal solution value and the size of the matching the algorithm maintains reduces by at least factor $(1 - 1/\mathrm{poly}\log n)$. It is then easy to verify that, after $O(\mathrm{poly}\log n)$ such iterations, the algorithm obtains an optimal matching. This is precisely the high-level approach that was used by [9], and we follow it in this work as well. In order to obtain an algorithm for Maximum Bipartite Matching, it is now sufficient to design a procedure that, given a residual flow network H corresponding to the current matching M, efficiently computes the set $\mathcal{P}$ of s-t paths in H with the above properties.

For technical reasons that will become clear later, we define a slightly more general problem, that we call RouteAndCut. In this problem, the input is a directed graph H, two disjoint sets A, B of its vertices with $|A| \leq |B|$, and two parameters $1 \leq \Delta \leq \min\{|A|, |B|\}$ and $1 \leq \eta \leq \Delta$. The goal is to either compute a collection $\mathcal{Q}$ of $\Omega(\Delta/\mathrm{poly}\log n)$ paths, each of which connects a distinct vertex of A to a distinct vertex of B, such that the paths in $\mathcal{Q}$ cause congestion $\tilde{O}(\eta)$; or to compute a cut (X, Y) in H with $|E_H(X, Y)| \ll \Delta/\eta$, with X containing a large fraction of vertices of A, and Y containing a large fraction of the vertices of B. While [9] do not explicitly define and solve this problem, their algorithm can be adapted to solve a special case of RouteAndCut, where $\eta \leq O(\mathrm{poly}\log n)$. So for brevity, we will say that the algorithm of [9] solves this special case of RouteAndCut. Clearly, an algorithm for the RouteAndCut problem can be used in order to compute a collection $\mathcal{P}$ of paths in the residual flow network H corresponding to the current matching M, with the desired properties that we described above.

The RouteAndCut problem falls into the extensively studied class of graph routing and flow problems. One standard approach for obtaining fast algorithms for such problems, due to [3, 13, 16], is via the Multiplicative Weight Update (MWU) method. It was further observed by Madry [26] that this approach can be viewed as reducing a given flow problem to a variant of decremental SSSP or APSP. In our case, the reduction is to decremental SSSP in directed graphs. While strong lower bounds are known for exact algorithms for decremental SSSP and APSP (see, e.g. [1, 2, 12, 20, 31]), we can exploit the special properties of the SSSP instances that arise from the RouteAndCut problem in order to obtain faster algorithms, an approach that was also used by [9].

We note that [7] provided a $(1 + \epsilon)$-approximation algorithm for directed decremental SSSP with total update time $\widetilde{O}(n^2/\epsilon)$, assuming all edge lengths are poly-bounded. Unfortunately, their algorithm can only withstand an *oblivious adversary* whereas instances of decremental SSSP arising from the MWU framework crucially require algorithms that can withstand an *adaptive adversary*, since the choice of the edge to be deleted in every update may depend on the algorithm's past behavior. A recent work of [6] provided a $(1+\epsilon)$-approximation algorithm for the directed decremental SSSP problem with an adaptive adversary, that achieves total update time $O\left(\frac{n^{8/3+o(1)}}{\epsilon}\right)$ (assuming that all edge lengths are poly-bounded). While this total update time is too high for speeding up algorithms for Maximum Bipartite Matching, their approach was adapted by [9] to handle the specific instances of SSSP that they obtain, leading to faster algorithms for Maximum Bipartite Matching. Specifically, one of the key observations of [9] is that the SSSP instances that arise from applying the MWU method to the Maximum Bipartite Matching problem have the property that all queries are between a fixed pair (s,t) of vertices, and a rather large approximation factor is acceptable. Moreover, by slightly modifying the standard MWU framework, they ensure that it is sufficient that the algorithm for the SSSP problem only responds to shortest-path queries as long as the current graph H contains a collection of least $\Omega(\Delta/\operatorname{poly}\log n)$ disjoint and short s-t paths, where Δ is the target number of augmenting paths that the algorithm aims to produce. We also follow their approach, and reduce the RouteAndCut problem, via a slightly modified MWU method, to a special case of directed decremental SSSP, that we refer to as decremental s-t-SP, that has all of the above properties. We note that s-t-SP is somewhat more general than the special case of the SSSP problem that was considered in [9], since they only provide an algorithm for the special case of RouteAndCut where $\eta \leq O(\operatorname{poly}\log n)$, while we need an algorithm that works for a wider range of values of parameter η. For now we focus on the description of their algorithm, and we assume for simplicity that $\eta = 1$ in this discussion.

The algorithm of [9] for a special case of the decremental s-t-SP problem follows the high-level approach of [6], that consists of two parts. First, they maintain a partition $\mathcal{X}$ of graph $H \setminus \{s,t\}$ into a collection of *expander-like graphs*; we abstract the problem of maintaining each such graph, that we call MaintainCluster problem, below. Intuitively, the MaintainCluster subroutine is given as input a vertex-induced subgraph H' of H, and a distance parameter d, with H' undergoing an online sequence of edge deletions. It needs to efficiently support short-path queries in H': given a pair $x,y \in V(H')$ of vertices, return an x-y path of length at most d in H'. However, it may, at any time, produce a cut (A,B) in H' of sparsity at most $O\left(\frac{\operatorname{poly}\log n}{d}\right)$, after which the vertices on one side of the cut are deleted from H', and the algorithm needs to continue with the resulting graph. The second main ingredient in the algorithm of [9] is the Approximate Topological Order ($\mathcal{ATO}$) framework of [7] (which is in turn based on the works of [19] and [5]), combined with the algorithm of [7] for decremental SSSP on "almost" DAG's. The latter algorithm is applied to the graph $\hat{H}$, that is obtained from H by contracting every almost-expander $X \in \mathcal{X}$ into a single vertex. We now discuss each of these components in turn, starting with the $\mathcal{ATO}$ framework.

The $\mathcal{ATO}$ framework. The Approximate Topological Order ($\mathcal{ATO}$) framework of [7, 19] is a central component in the algorithms of [6, 9], as well as our algorithm. An $\mathcal{ATO}$ data structure in a dynamic graph H must maintain a *partition* $\mathcal{X}$ of the vertices of H into subsets. We refer to the sets $X \in \mathcal{X}$ as *clusters*, and to $\mathcal{X}$ as a *clustering*. The only allowed changes to the clustering $\mathcal{X}$ are *cluster splittings*: given an existing cluster $X \in \mathcal{X}$ and a subset $X' \subseteq X$ of its vertices, delete the vertices of X' from X, and add X' as a new cluster to $\mathcal{X}$. We assume further that the input graph H contains two special vertices s and t, and that clusters $S = \{s\}$ and $T = \{t\}$ always lie in $\mathcal{X}$. In addition to maintaining the clustering $\mathcal{X}$, the $\mathcal{ATO}$ must maintain an *ordering* σ of its clusters. Assume that $\mathcal{X} = \{X_1, \ldots, X_r\}$, and that the clusters are indexed according to the ordering σ. Assume further that a cluster X_i undergoes splitting, with the new cluster $X'_i \subseteq X_i$ inserted into $\mathcal{X}$. Then the ordering σ must evolve in a *consistent* manner, that is, the new ordering must be either $(X_1, \ldots, X_{i-1}, X'_i, X_i \setminus X'_i, X_{i+1}, \ldots, X_r)$, or $(X_1, \ldots, X_{i-1}, X_i \setminus X'_i, X'_i, X_{i+1}, \ldots, X_r)$. Consider now some edge $e = (x,y)$ of H, and assume that $x \in X_i$ and $y \in X_j$. If X_i appears before X_j in the ordering σ, then we say that e is a *left-to-right* edge; if $i = j$, we say that it is a *neutral* edge; and otherwise we say that it is a *right-to-left* edge. If e is a right-to-left edge, then we define its *span*: $\operatorname{span}(e) = \sum_{i'=j}^{i} |X_{i'}|$ (we assume here that the sets in $\mathcal{X}$ are indexed according to the ordering σ). Let $\hat{H}$ be the contracted graph corresponding to H: that is, $\hat{H}$ is obtained from H by contracting each of the clusters $X \in \mathcal{X}$ into a vertex v_X. For simplicity, we will refer to the vertices v_S and v_T as s and t, respectively. Intuitively, if we could maintain the $\mathcal{ATO}$ without introducing any right-to-left edges, then the corresponding contracted graph $\hat{H}$ is a DAG, and the ordering σ associated with the $\mathcal{ATO}$ naturally defines a topological ordering of the vertices of $\hat{H}$. We could then use the algorithm of [7] for decremental SSSP in DAG's, that builds on the work of [5, 19], in order to support approximate shortest path queries in $\hat{H}$ between s and other vertices of $\hat{H}$, with total update time $\widetilde{O}(n^2)$. But in order to be able to support shortest s-t path queries in the original graph H, we need to ensure that the diameters of the subgraphs $H[X]$ corresponding to the clusters $X \in \mathcal{X}$ are sufficiently small, and that we can support approximate shortest-path queries between *arbitrary pairs* of vertices within each such graph efficiently.

Towards this end, it was observed by [7] that the algorithm for decremental SSSP in DAG's can be further extended to "almost DAG's": suppose G is a directed graph, and let ρ be a fixed ordering of its vertices. Assume that $V(G) = \{v_1, \ldots, v_n\}$, where the vertices are indexed according to the ordering ρ. If $e = (v_i, v_j)$ is an edge with $i > j$, then we say that e is a *right-to-left* edge of G with *width* $(i-j)$. It was shown in [7] that the algorithm for decremental SSSP on DAG's can be efficiently extended to such graphs G, provided that the total width of all right-to-left edges is relatively small. Specifically, the running time of their algorithm becomes roughly $\tilde{O}(n^2 + \Gamma \cdot n)$, where Γ is the total width of the right-to-left edges.

Assume now that the algorithm for SSSP on the almost-DAG graph G only needs to respond to approximate shortest-path queries between a specific fixed pair s,t of vertices, and moreover, that it only needs to support such queries as long as G contains $\Omega(\Delta)$ short edge-disjoint s-t paths. It was observed in [9] that, in such a case, the running time of the algorithm of [7] improves to roughly $\tilde{O}(n^2 + \Gamma \cdot n/\Delta)$. This observation was one of the key insights that allowed them to obtain a faster running time for the special case of the s-t-SP problem, and for Maximum Bipartite Matching.

We now provide additional relevant details of the algorithm of [9]. Like in [6], the MaintainCluster problem is exploited in order to maintain an $\mathcal{ATO}$ of the input graph H. Initially, the clustering $\mathcal{X}$ contains three clusters: $S = \{s\}$, $T = \{t\}$, and $U = V(H) \setminus \{s, t\}$. The algorithm for the MaintainCluster problem is then initialized on graph $H[U]$, with an appropriately chosen distance parameter d_U. In general, whenever a new cluster X joins $\mathcal{X}$, the algorithm for the MaintainCluster problem is initialized on $H[X]$. Whenever that algorithm produces a sparse cut (A, B) in X, we select a subset $Z \in \{A, B\}$ of vertices to be deleted from X, update X by deleting these vertices, and add Z as a new cluster to $\mathcal{X}$, after which the algorithm for the MaintainCluster problem is initialized on $H[Z]$. The key idea is that, since the cuts produced by the algorithm for the MaintainCluster problem are sparse, we can ensure that the total span of all right-to-left edges is sufficiently small. If we then consider the contracted graph $\hat{H}$, this, in turn, ensures that the total width of all right-to-left edges in $\hat{H}$ is low. We can now apply the algorithm of [7] for decremental SSSP on almost-DAG's to support approximate shortest s-t path queries in $\hat{H}$, while exploiting the fact that such queries only need to be supported as long as $\hat{H}$ contains a large number of short edge-disjoint s-t paths, in order to speed it up. For every cluster $X \in \mathcal{X}$, we can then use the algorithm for the MaintainCluster problem on $H[X]$, in order to respond to approximate shortest-path queries between pairs of vertices in X. Combining these data structures together, we can support approximate shortest s-t path queries in the original graph H, as long as H contains many short edge-disjoint s-t paths. This high-level approach allows one to obtain algorithms for decremental s-t-SP, and for the RouteAndCut problem, from algorithms for the MaintainCluster problem, that we now discuss in more detail.

The MaintainCluster *problem.* To recap, in the MaintainCluster problem, the input is a graph G that undergoes an online sequence of edge deletions, and a distance parameter d. The goal is to efficiently support short-path queries: given a pair x, y of vertices of G, return a path of length at most d connecting them in G. The algorithm may, however, at any time, produce a cut (A, B) in G of sparsity at most $O\left(\frac{\text{poly}\log n}{d}\right)$, following which, the vertices of one side of the cut are deleted from G. The algorithm is used in order to maintain individual clusters of the $\mathcal{ATO}$. A similar problem was considered by [6], who provide an algorithm with total update time $\widetilde{O}(|E(G)| \cdot d^2)$ for it, where the time to respond to each query is roughly proportional to the number of edges on the path that the algorithm returns. In [9] this problem was considered in a more relaxed setting, where the number of queries that the algorithm must support is bounded by a given parameter Δ, and the goal is to minimize the total running time of the algorithm, that is, the sum of the total update time, and the time required to respond to all queries. The algorithm of [9] for this setting has running time $\widetilde{O}(|E(G)| \cdot d + |V(G)|^2) \cdot \max\left\{1, \frac{\Delta \cdot d^2}{|V(G)|}\right\}$, which, for the specific parameters that they employ, becomes $\widetilde{O}(|E(G)| \cdot d + |V(G)|^2)$. In order to obtain our improved algorithm for Maximum Bipartite Matching, we need to generalize this result so that it works for a wider range of parameters, achieving running time $|V(G)|^{2+o(1)}$.

The algorithm of [9] follows a rather standard approach. First, they use the Cut-Matching game in order to compute a large expander graph $\hat{G}$, and to embed it into G via short paths that cause low congestion. The algorithm for the Cut Player is implemented in a rather straightforward manner, since they can afford a running time that is as high as $\Theta(|V(\hat{G})|^2)$. The algorithm for the Matching Player essentially needs to solve an instance of the RouteAndCut problem. Using the MWU approach as before, it can be reduced to solving an instance of directed decremental SSSP. The algorithm of [9] then uses the standard Even-Shiloach tree data structure in order to solve the latter problem. In addition to the expander $\hat{G}$ and its embedding into G, the algorithm of [9] maintains two additional Even-Shiloach trees in G. Both trees are rooted in the vertices of $\hat{G}$, and have depth roughly d. One of the trees has all its edges directed away from the root, and the other has all of its edges directed towards the root. In order to respond to a query between a pair x, y of vertices of G, the two Even-Shiloach trees are used to compute a short path connecting x to some vertex $x' \in V(\hat{G})$, and a short path connecting some vertex $y' \in V(\hat{G})$ to y. A simple BFS search in the expander $\hat{G}$ then yields a short path connecting x' to y', which can be turned into an x'-y' path in G by exploiting the embedding of $\hat{G}$ into G.

We now describe several sources of inefficiency of the algorithm of [9], and then describe our approach to overcoming them. First, both the algorithms of [9] and [6] for the MaintainCluster problem are only designed for graphs with unit edge-lengths. However, both of these works solve (a variant of) the SSSP problem in graphs with arbitrary edge lengths. To overcome this difficulty, [9] use the same approach as [6]: namely, they choose a threshold τ, and initially delete all edges whose length is greater than τ (called *long edges*) from the input graph H. The lengths of the remaining edges (called *short edges*) are set to 1 for the sake of maintaining the $\mathcal{ATO}$ and solving the MaintainCluster problem on the resulting instances. The long edges however are reinserted into the contracted graph $\hat{H}$, and the actual lengths of the short edges are used in it as well. This approach unfortunately results in a rather large number of right-to-left edges with a large width in $\hat{H}$, as it may potentially include all long edges. In order to overcome this difficulty, we design an algorithm for the MaintainCluster problem that can handle arbitrary edge lengths, which adds an additional dimension of technical challenges.

The second main source of inefficiency in the algorithm of [9] is the use of Even-Shiloach trees in their algorithm for MaintainCluster, both in implementing the Matching Player in the Cut-Matching game, and in order to maintain short paths connecting all vertices of G to the vertices of the expander $\hat{G}$. It is immediate to see that the problem that the Matching Player needs to solve is essentially an instance of the RouteAndCut problem. We also observe that an algorithm for a variant of the RouteAndCut problem can be exploited in order to maintain the paths connecting all vertices of G to the vertices of $V(\hat{G})$. We abstract this as a new problem, that we call ConnectToCenters, and discuss it below. We believe that this problem and our algorithm for solving it are of independent interest. We remark that this reduction from the ConnectToCenters problem to the RouteAndCut problem requires that the algorithm for the RouteAndCut problem works for arbitrary congestion parameter $\eta \leq \Delta$, and this is the reason for our more general definition of the RouteAndCut problem.

To summarize, as already shown in previous work, in order to obtain an efficient algorithm for the RouteAndCut problem, it is enough to obtain an efficient algorithm for the MaintainCluster problem, and we observe that the opposite is also true: an efficient algorithm for the RouteAndCut problem implies an efficient algorithm for the MaintainCluster problem. This, however, creates a chicken-and-egg issue, where in order to solve one of the two problems efficiently, we need to design an efficient algorithm for the other. We overcome this barrier by using a recursive approach.

A recursive approach. We parameterize both the RouteAndCut and the MaintainCluster problem using a parameter $r > 0$. We say that an instance of the MaintainCluster problem on an n-vertex graph G with a distance parameter d is r*-restricted*, if $d \leq 2^{r\cdot\sqrt{\log n}}$. Consider now an instance of the RouteAndCut problem on an n-vertex graph H, with two subsets A, B of its vertices, and parameters Δ and η. It is not hard to see that, if $\mathcal{P}$ is any collection of $\Omega(\Delta/\text{poly}\log n)$ paths connecting vertices of A to vertices of B, that cause vertex-congestion at most η, then a large fraction of the paths in $\mathcal{P}$ have length $\tilde{O}(n\eta/\Delta)$. We say that an instance of the RouteAndCut problem is r*-restricted* if $\frac{n\eta}{\Delta} \leq 2^{r\cdot\sqrt{\log n}}$. We show a straightforward algorithm for the 1-restricted RouteAndCut problem. Then for all $r \geq 1$, we show that an efficient algorithm for the r-restricted RouteAndCut problem implies an efficient algorithm for the r-restricted MaintainCluster problem. We also show that an efficient algorithm for the r-restricted MaintainCluster problem implies an efficient algorithm for the $(r+1)$-restricted RouteAndCut problem. Using induction on r, we then simultaneously obtain efficient algorithms for the RouteAndCut and the MaintainCluster problems for the entire range of values for the parameter r.

ConnectToCenters *problem.* The ConnectToCenters problem is employed as a subroutine in the algorithm for the MaintainCluster problem, but we feel that it is interesting in its own right, as it seems to arise in many different settings. Suppose we are given a dynamic n-vertex graph G; for now we will assume that G undergoes an online sequence of edge deletions, but in fact our algorithm considers other updates, as described later. In addition to graph G, assume that we are given a parameter d, and a subset $T \subseteq V(G)$ of vertices that we call *centers*. The goal is to maintain, for every vertex $v \in V(G)$, a path $P(v)$ of length at most d, connecting v to some vertex of T. As the time progresses, some vertices may be deleted from T, but we are guaranteed that T always contains a large enough fraction of the vertices of G, e.g. $|T| \geq \Omega(|V(G)|/(d\,\text{poly}\log n)$. In order to ensure that the deletion of edges from G does not impact too many paths in $\mathcal{P}^* = \{P(v) \mid v \in V(G)\}$, it is desirable that the paths cause a small edge-congestion (say at most $\tilde{O}(d)$), and for similar reasons it is desirable that every vertex $x \in T$ serves as an endpoint of relatively few such paths (say at most $\tilde{O}(d)$). At any time, the algorithm may compute a cut (A, B) of sparsity at most $O\left(\frac{\text{poly}\log n}{d}\right)$, with $|A \cap T| \ll |A|$, after which the vertices of A are deleted from G and the algorithm continues. We note that whenever the by now standard paradigm of embedding an expander into the input graph G and then maintaining short paths connecting all vertices of G to the vertices of the expander is used, one essentially needs to solve a variant of the ConnectToCenters problem. So far this was typically done by using Even-Shiloach trees, but this data structure becomes inefficient once the depth parameter d is sufficiently large. It is for this reason that we believe that our algorithm for the ConnectToCenters problem is of independent interest.

It is easy to see that the initial collection $\mathcal{P}^* = \{P(v) \mid v \in V(G)\}$ of paths of length $O(d)$ each, connecting every vertex of G to some vertex of T, that cause edge-congestion $\tilde{O}(d)$, can be computed by employing an algorithm for the RouteAndCut problem. As edges are deleted from G, and vertices are deleted from T, some of the paths in $\mathcal{P}^*$ may be destroyed. Whenever a path $P(v) \in \mathcal{P}^*$ is destroyed, we say that vertex v becomes *disconnected*. We then need to *reconnect* all disconnected vertices to T. This, again, can be done by employing an algorithm for the RouteAndCut problem, but doing so directly may be very inefficient. Assume, for example, that we are given an algorithm $\mathcal{A}$ for the RouteAndCut problem, that, on an n-vertex graph G, has running time $O(n^{2+o(1)})$. Every time a subset of vertices of G becomes disconnected, we would need to employ this algorithm in order to reconnect them to T. However, it is possible that only a small number of vertices become disconnected at a time, and spending $\Theta(n^{2+o(1)})$ time to reconnect them each time is prohibitively expensive. A better approach seems to be to consider the set U of vertices that are currently connected, and the set U' of vertices that are currently disconnected. We could then attempt to route the vertices of U' to the vertices of U by constructing a new collection $\mathcal{Q} = \{Q(u) \mid u \in U'\}$ of paths, where each path $Q(u)$ connects u to some vertex of U; and then exploit the existing paths in $\{P(v) \mid v \in U\}$ in order to compute paths connecting every vertex of U' to the vertices of T. However, we cannot afford to spend $\Theta(n^2)$ time in order to compute the set $\mathcal{Q}$ of paths. On the other hand, intuitively, if $|U'| \ll U$, then we may not need to explore the entire graph G in order to compute the set $\mathcal{Q}$ of paths. In order to overcome this difficulty, we require that the algorithm for the RouteAndCut problem has running time roughly $n^{1+o(1)} \cdot (n - |B|)$, instead of $n^{2+o(1)}$. In particular, if the graph G is sufficiently dense and $|B|$ is sufficiently large, then this running time may be much smaller than $|E(G)|$. Our reduction from the $(r+1)$-restricted RouteAndCut problem to the r-restricted MaintainCluster problem follows the high-level approach of [6] and [9]. However, this additional strict requirement on the efficiency of the algorithm for the RouteAndCut problem, in addition to the requirement that the algorithm should work for arbitrary values of the congestion parameter η, make the reduction more challenging and technical.

Assume now that we are given an algorithm $\mathcal{A}$ for RouteAndCut, that, on an instance (G, A, B, Δ, η) has running time $n^{1+o(1)} \cdot (n - |B|)$, where $n = |V(G)|$. As described above, in order to implement our algorithm for the ConnectToCenters problem, whenever we are given a collection U' of vertices that are currently disconnected from T, we can now employ Algorithm $\mathcal{A}$ to construct a collection $\mathcal{Q}$ of paths connecting them to the vertices of U (the set of currently connected vertices), and then compose $\mathcal{Q}$ with the collection $\{P(v) \mid v \in U\}$ of paths to obtain the desired collection $\{P(v) \mid v \in U'\}$ of paths that reconnects the vertices of U' to T. Unfortunately, if we follow this approach, and keep appending paths to each other iteration after iteration, we may obtain paths whose lengths are prohibitively large. Instead, we follow a *layered* approach. For a parameter $\lambda = O(\log n)$, we maintain at all times a partition $(U_0, \ldots, U_\lambda)$ of the vertices of G into *layers*, where $U_0 = T$, and U_λ contains all vertices that are currently disconnected. For all $1 \leq i < \lambda$, we also maintain a collection $\mathcal{Q}_i = \{K(v) \mid v \in U_i\}$ of paths, where each path $K(v)$ connects a vertex $v \in U_i$ to a vertex in $\bigcup_{i'=0}^{i-1} U_{i'}$. The paths in each set $\mathcal{Q}_i$ have length at most $\frac{d}{4\lambda}$, and cause congestion $\tilde{O}(d)$ in G. By composing the paths from different sets $\mathcal{Q}_i$, we can obtain, for every vertex $v \in V(G)$, a path that connects it to some vertex of T. At a high level, for all $1 \leq i < \lambda$, we reconstruct layer U_i and the set $\mathcal{Q}_i$ of paths from scratch every time that roughly $2^{\lambda-i}$ new vertices become disconnected, and we also ensure that $|U_i| \leq 2^{\lambda-i}$ holds. So when we employ the algorithm for the RouteAndCut problem in order to reconnect the vertices of U_i, in the resulting instance of RouteAndCut, $n - |B| \leq 2^{\lambda-i}$ holds, and the running time of the algorithm is roughly $n^{1+o(1)} \cdot 2^{\lambda-i}$. Therefore, as index i becomes smaller, the running time of the RouteAndCut algorithm that is used to reconnect the vertices of U_i increases. However, for smaller values of index i, we need to reconstruct the set U_i of vertices and the set $\mathcal{Q}_i$ of paths less often. This eventually leads to the desired $n^{2+o(1)}$ running time.

Edge-deletion versus edge-length-increase updates. We would like to highlight here what we feel is a somewhat surprising insight from our work, that may appear minor at first sight, but we believe that it may be useful in other problems. Consider the following two settings for dynamic graphs: the first one is the standard decremental setting, where edges are deleted from the input graph G as the time progresses. The second setting is a somewhat more unusual one, where the only updates that are allowed in the input graph G is the doubling of the lengths of its edges; we refer to this type of updates as edge-length-increases. Generally, it is not hard to see that both models are roughly equivalent. Indeed, in order to simulate edge-length-increases in the standard decremental setting, we can simply create a large number of copies of each edge e of various lengths, and then, as the length of e increases, some of these copies are deleted. The reduction in the other direction is also immediate: we can simulate the deletion of an edge e from G by repeatedly doubling its length, until it becomes very high.

The dynamic graphs that arise from using the MWU framework typically only undergo edge-length-increase updates, which are then typically implemented as edge-deletions, in order to reduce the problem to the more standard decremental SSSP, as described above. However, the edge-length-increases that the input graph G undergoes under this implementation of the MWU method are rather well-behaved: specifically, the lengths of the edges are only increased moderately, and all length increases occur on the edges that participate in the paths that the algorithm returns in response to queries. We observe that the resulting SSSP problem appears to be easier if we work with edge-length-increase updates directly, instead of the more traditional approach of transforming them into edge-deletion updates.

In order to illustrate this, consider the following simple scenario: we are given a graph G, and initially the length of every edge in G is 1. Assume also that we have computed another graph X (it may be convenient to think of X as an expander), and embedded X into G via paths of length at most d, that cause edge-congestion at most η. If some edge e is deleted from G, then every edge $\hat{e} \in E(X)$, whose embedding path $Q(\hat{e})$ uses e, must be deleted from X as well. Therefore, the deletion of a single edge from G may lead to the deletion of η edges from X. Assume now that, instead, the edges of G only undergo increases in their lengths, where the length of each edge may be iteratively doubled, but the total increase in the lengths of all edges is moderate. If the length of an edge $e \in E(G)$ is doubled, then for every edge $\hat{e} \in E(X)$ whose embedding path $Q(\hat{e})$ uses e, the length of $Q(\hat{e})$ increases only slightly, and so there is no need to delete $\hat{e}$ from $E(X)$. We can usually wait until the length of the path $Q(\hat{e})$ increases significantly before edge $\hat{e}$ needs to be deleted from X. As mentioned already, in instances arising from the MWU framework, the total increase in the lengths of all edges in G over the course of the entire algorithm is usually not very large, and in particular most edges whose lengths are doubled are short. This allows us to maintain the expander X and its embedding into G over the course of a much longer sequence of updates to G. This is one of the insights that allowed us to obtain a more efficient algorithm for the MaintainCluster problem.

To summarize, our algorithm departs from the algorithm of [9] in the following key aspects. First, we use the MWU method to reduce the RouteAndCut problem to s-t-SP in dynamic graphs that undergo edge-length-increases instead of edge-deletion updates. Second, our algorithm for the RouteAndCut problem has running time that significantly decreases when the set B of vertices contains almost all vertices of G; in some cases the running time may even be lower than $|E(G)|$. We extend the MaintainCluster problem to handle arbitrary edge lengths, but unlike [9] we only allow edge-length-increases, instead of edge-deletion updates. We design an algorithm for the ConnectToCenters problem, that we believe is of independent interest, and that can be viewed as reducing the MaintainCluster problem to RouteAndCut. Lastly, we use a recursive approach, in which algorithms for RouteAndCut rely on algorithms for MaintainCluster and vice versa, by parametrizing both problems with an auxiliary parameter r, and then inductively develop algorithms for both problems for the entire range of r.

Organization. We start with preliminaries in Section 2, and then provide a high-level overview of our algorithm in Section 3, where we also formally define the RouteAndCut problem, state our main result for it, and describe how our algorithm for the RouteAndCut problem implies the main result of this paper. We defer the complete proofs to the full version of the paper.

2 PRELIMINARIES

In this paper we work with both directed and undirected graphs. By default graphs do not contain parallel edges or self-loops.

Let G be a graph with capacities $c(e) \geq 0$ on edges $e \in E(G)$, and let $\mathcal{P}$ be a collection of simple paths in G. We say that the paths in $\mathcal{P}$ cause *edge-congestion* η, or just *congestion* η, if every edge $e \in E(G)$ participates in at most $\eta \cdot c(e)$ paths in $\mathcal{P}$. When edge capacities are not explicitly given, we assume that they are unit. If every edge of G belongs to at most one path in $\mathcal{P}$, then we say that the paths in $\mathcal{P}$ are *edge-disjoint*. Similarly, if we are given a flow value $f(e) \geq 0$ for every edge $e \in E(G)$, we say that flow f causes *congestion* η if, for every edge $e \in E(G)$, $f(e) \leq \eta \cdot c(e)$

holds. If $f(e) \le c(e)$ holds for every edge $e \in E(G)$, we may say that f causes *no congestion*.

3 HIGH-LEVEL OVERVIEW OF THE ALGORITHM

Throughout the paper we will work with special kinds of directed graphs that we refer to as *well-structured graphs*, and define below.

Definition 3.1 (Well-Structured Graphs). *Let $G = (L, R, E)$ be a bipartite directed graph. We call the edges of $E_G(L, R)$* regular *and the edges of $E_G(R, L)$* special. *We say that G is a* well-structured graph, *if every vertex of G is incident to at most one special edge.*

If G is a well-structured graph, then we assume that the partition (L, R) of its vertices is given as part of the description of G.

In the remainder of this section, we define a new problem, called RouteAndCut, which is one of the main building blocks of our algorithm. We then state the main theorem for this section, that provides an efficient algorithm for the RouteAndCut problem, and show that our main result – the proof of Theorem 1.1 follows from it. We also provide a high-level overview of the algorithm for the RouteAndCut problem.

3.1 The RouteAndCut problem

In this subsection we define the RouteAndCut problem, which is one of the main building blocks of our algorithm. Before we do so, we need to define the notion of *routing*.

Definition 3.2 (Routing). *Let $G = (V, E)$ be a directed graph, and let A, B be two disjoint subsets of its vertices. A* routing *from A to B is a collection $\mathcal{Q}$ of paths in G, such that every path in $\mathcal{Q}$ connects a vertex of A to a vertex of B, and the endpoints of the paths in $\mathcal{Q}$ are all disjoint. The* congestion *of the routing is the edge-congestion that the set $\mathcal{Q}$ of paths causes in graph G. Vertices of $A \cup B$ may serve both as endpoints and as inner vertices of the paths in $\mathcal{Q}$ simultaneously.*

We are now ready to define the RouteAndCut problem, and its special case, called r-restricted RouteAndCut. Intuitively, we use the notion of the r-restricted RouteAndCut problem in order to discretize the problem instances: our algorithm for RouteAndCut will consider, by induction, r-restricted instances of the problem, from smaller to larger values of r.

Definition 3.3. *The input to* RouteAndCut *problem is a well-structured n-vertex graph $G = (L, R, E)$, that is given in the adjacency-list representation, parameters $N \ge n$, $\Delta \ge 1$, and $1 \le \eta \le \Delta$, and two disjoint subsets A, B of vertices of G, with $|A|, |B| \ge \Delta$.*

The output of the problem is a routing $\mathcal{Q}$ from A to B, whose congestion is bounded by $4\eta \log N$. Additionally, if $|\mathcal{Q}| < \Delta$, the output must contain a cut (X, Y) in G with $|E_G(X, Y)| \le \frac{64\Delta}{\eta \log^4 n} + \frac{256|\mathcal{Q}|}{\eta}$, such that, if $A' \subseteq A$, $B' \subseteq B$ denote the subsets of vertices that do not serve as endpoints of the paths in $\mathcal{Q}$, then $A' \subseteq X$ and $B' \subseteq Y$ hold. The algorithm for the RouteAndCut *problem is also allowed to return "FAIL" without producing any output, but the probability of the algorithm doing so must be bounded by $1/2$. We say that an instance $(G, A, B, \Delta, \eta, N)$ of the* RouteAndCut *problem is* r-restricted, *for an integer $1 \le r \le \left\lceil \sqrt{\log N} \right\rceil$, iff $\frac{(n-|B|)\cdot\eta}{\Delta} \le 2^{r\cdot\sqrt{\log N}}$ holds.*

To get some intuition about the definition of *r-restricted* instances, suppose we compute a routing from A to B in graph G of cardinality Δ that causes edge-congestion at most η. Assume w.l.o.g. that, if $b \in B$ is an inner vertex on some path of the routing, then it serves as an endpoint of some other path of the routing. Then it is not hard to show that most of the paths in the routing must have length at most $d = O\left(\frac{(n-|B|)\cdot\eta}{\Delta}\right)$. The definition or r-restricted instances requires that this parameter d is roughly bounded by $2^{r\sqrt{\log N}}$. It is easy to see that our starting instance must be r^*-restricted, for $r^* = \left\lceil \sqrt{\log N} \right\rceil$. In order to solve this problem instance, we will need to solve the problem recursively on smaller subgraphs G' of G, but it is important for us to ensure that the resulting instances are r'-restricted, for $r' < r^*$. In order to do so, we will let the parameter N in the resulting subinstances correspond to the number of vertices in the original graph G (we may need to slightly adjust it for technical reasons but the adjustments are minor). By appropriately setting the parameters Δ and η for the resulting subinstances, we can then ensure that they are indeed r'-restricted, for some $r' < r^*$. Overall, the parameter N can be thought of as roughly the number of vertices in the original instance of the RouteAndCut problem that we try to solve, and it is used mostly to define the notion of r-restricted instances. The notion of r-restricted instances allows us to construct algorithms for RouteAndCut inductively, from smaller to larger values of r. The following theorem provides one of our main technical results, namely an efficient algorithm for the RouteAndCut problem.

Theorem 3.4. *There is a randomized algorithm for* RouteAndCut *problem, that, on an input $(G, A, B, \Delta, \eta, N)$, has running time at most $O\left(n \cdot (n - |B|) \cdot 2^{O(\sqrt{\log N}\cdot\log\log N)}\right)$, where $n = |V(G)|$.*

We next show that the proof of Theorem 1.1 follows from the above theorem.

3.2 Completing the Proof of Theorem 1.1

Recall that in the Maximum Bipartite Matching problem, the input is an undirected n-vertex bipartite graph $G = (L, R, E)$, and the goal is to compute a matching of maximum cardinality in G.

We can assume w.l.o.g. that we are given a target integral value $C^* > 0$. If $C^* \le \mathrm{OPT}$, then our algorithm must either produce a matching of cardinality C^*, or terminate with a "FAIL", but we require that the latter only happens with probability at most $1/\mathrm{poly}(n)$ in this case. If $C^* > \mathrm{OPT}$, then our algorithm may return an arbitrary matching, or terminate with a "FAIL". We can then use binary search to compute the optimal solution with high probability. From now on, we assume that we are given a target value C^*, and that G contains a matching of cardinality C^*. Our algorithm must either compute a matching of cardinality C^*, or to return "FAIL", but the probability for returning "FAIL" must be bounded by $1/\mathrm{poly}(n)$. For convenience, we denote C^* by OPT. We can assume that n is greater than a large enough constant, since otherwise the problem can be solved in $O(1)$ time.

It is well known that the Maximum Bipartite Matching problem can be reduced to computing maximum s-t flow in a directed flow network with unit edge capacities. In order to do so, we start with the graph $G = (L, R, E)$, and direct all its edges from the vertices of L towards the vertices of R. We then add a source vertex s, that connects with an edge to every vertex of L, and a destination vertex t, to which every vertex of R is connected. All edge capacities are set to 1. Let G' denote the resulting directed flow network. It is easy to verify that the value of the maximum s-t flow in G' is equal to the cardinality of maximum matching in G. Moreover, given an integral flow f in G', we can compute a matching M of cardinality $\mathrm{val}(f)$ in G, in time $O(n)$: we simply include in M all edges of G that carry 1 flow unit.

Our algorithm consists of $O(\log^3 n)$ phases. Throughout the algorithm, we maintain a matching M in G, starting with $M = \emptyset$, and we denote $\Delta^* = \mathrm{OPT} - |M|$. If M is the matching at the beginning of a phase, and M' is the matching obtained at the end of the phase, then we require that $|M'| \geq |M| + \Omega(\Delta^*/\log^2 n)$. We show a combinatorial algorithm, that, given an initial matching M, either returns "FAIL", or computes such a matching M', in time $O\left(n^2 \cdot 2^{O(\sqrt{\log n}\log\log n)}\right)$, where the probability that the algorithm returns "FAIL" is bounded by $1/\mathrm{poly}(n)$. This will ensure that the number of the phases is bounded by $O(\log^3 n)$, and the total running time of the algorithm is $O\left(n^2 \cdot 2^{O(\sqrt{\log n}\log\log n)}\right)$. From now on we focus on the description of a single phase.

Implementation of a Single Phase. We assume that we are given some matching M in the input graph G, and we denote $\Delta^* = \mathrm{OPT} - |M|$. Our goal is to compute a matching M' of cardinality at least $|M| + \Omega(\Delta^*/\log^2 n)$. As observed already, matching M defines a flow f of value $|M|$ in the directed flow network G' with unit edge capacities. We denote by $H = G'_f$ the corresponding residual flow network, and we say that H is the *residual flow network of G with respect to matching M*. Observe that H is a directed flow network with unit edge capacities, and that the value of the maximum s-t flow in H is at least $\mathrm{OPT} - |M| = \Delta^*$.

Next, we will define an instance of the RouteAndCut problem. Consider first the graph $H' = H \setminus \{s, t\}$. Notice that, for every edge $e = (u, v)$ of G with $u \in L$ and $v \in R$, if $e \in M$, then edge (v, u) is present in H', and otherwise edge (u, v) is present in H'. Therefore, all edges of H' that are directed from vertices of R to vertices of L correspond to the edges of the current matching M. Clearly, every vertex of H' may be incident to at most one edge of $E_{H'}(R, L)$, and so graph H' is a well-structured graph.

We let $A \subseteq L$ be the set of vertices v, such that edge (s, v) is present in graph H, and we let $B \subseteq R$ be the set of vertices u, such that edge (u, t) is present in H. We also define parameters $\eta = 1$, $N = n$, and $\Delta = \Delta^*$. It is easy to verify that $(H', A, B, \Delta, \eta, N)$ is a valid input to the RouteAndCut problem, and we can compute an adjacency-list representation of H' in time $O(|E(G)|)$. Recall that there is an s-t flow f of value Δ^* in graph H, and, from the integrality of maximum flow in integer-capacity networks, we can assume that this flow is integral. This flow naturally defines a routing $\mathcal{Q}^*$ from A to B, that causes congestion $\eta = 1$, with $|\mathcal{Q}^*| = \Delta^*$.

We now apply the algorithm from Theorem 3.4 to the instance $(H', A, B, \Delta, \eta, N)$ of the RouteAndCut problem. As long as the algorithm returns "FAIL", we keep executing it, for up to $\lceil 100 \log n \rceil$ iterations. If the algorithm from Theorem 3.4 returned "FAIL" in all $\lceil 100 \log n \rceil$ consecutive iterations, we terminate our algorithm and return "FAIL". It is easy to verify that this may happen with probability at most $1/n^{100}$. Otherwise, the algorithm from Theorem 3.4 must return a routing $\mathcal{Q}$ from A to B with congestion at most $4 \log n$. It is not hard to see that $|\mathcal{Q}| \geq \frac{\Delta^*}{\log n}$ must hold; a formal proof of this fact can be found in the full version of the paper.

Notice that the paths in $\mathcal{Q}$ naturally define a collection $\mathcal{Q}'$ of at least $\frac{\Delta^*}{\log n}$ s-t paths in graph H (the residual flow network with respect to G and the current matching M), and they cause congestion at most $4 \log n$ (since the endpoints of the paths in $\mathcal{Q}$ are disjoint). Next, we show an algorithm that computes a collection $\mathcal{Q}''$ of $\Omega\left(\frac{\Delta^*}{\log^2 n}\right)$ edge-disjoint s-t paths in graph H. We will then use the paths in $\mathcal{Q}''$ in order to augment the current flow f in graph G', which, in turn, will allow us to compute the new augmented matching M'.

In order to compute the collection $\mathcal{Q}''$ of paths, we construct a directed graph $H'' \subseteq H$, that consists of all vertices and edges that participate in the paths of $\mathcal{Q}'$. The capacity of every edge in H'' remains unit – the same as its capacity in H. The observation below summarizes some useful properties of the graph H''; its proof is deferred to the full version.

Observation 3.5. *$|E(H'')| \leq O(n \log n)$, and there is an s-t flow of value at least $\frac{\Delta^*}{4 \log^2 n}$ in H''.*

Next, we compute an integral maximum s-t flow in H'' that obeys the edge capacities in H'', using the standard Ford-Fulkerson algorithm. Each iteration of the Ford-Fulkerson algorithm takes $O(|E(H'')|) = \widetilde{O}(n)$ time, and, since there can be at most n iterations, in $\widetilde{O}(n^2)$ time we recover a collection $\mathcal{Q}''$ of at least $\Omega\left(\frac{\Delta^*}{\log^2 n}\right)$ edge-disjoint paths connecting s to t in H''. Since $H'' \subseteq H$, we have now obtained the desired collection $\mathcal{Q}''$ of at least $\Omega\left(\frac{\Delta^*}{\log^2 n}\right)$ edge-disjoint s-t paths in H. We note that we could also directly round the initial fractional s-t flow in H'', in expected time $\widetilde{O}(|E(H'')|) = \widetilde{O}(n)$, e.g. by using the algorithm from Theorem 5 in [23], that builds on the results of [17]. But since the bottlenecks in the running time of our algorithm lie elsewhere, we instead use the above simple deterministic algorithm.

We can now augment the current flow in G' via the collection $\mathcal{Q}''$ of augmenting paths, obtaining a new integral flow in graph G' of value $|M| + |\mathcal{Q}''|$, which, in turn, defines a new matching M' with $|M'| \geq |M| + |\mathcal{Q}''| \geq |M| + \Omega\left(\frac{\Delta^*}{\log^2 n}\right)$.

We now bound the running time of a single phase. Recall that we may execute the algorithm from Theorem 3.4 at most $O(\log n)$ times per phase. The running time of a single such execution is $O\left(n^2 \cdot 2^{O(\sqrt{\log N} \cdot \log\log N)}\right) \leq O\left(n^2 \cdot 2^{O(\sqrt{\log n} \cdot \log\log n)}\right)$. Additionally, the time required to compute the graph H'', and to compute the maximum flow in it is bounded by $\widetilde{O}(n^2)$. Overall, the running time of a single phase is $O\left(n^2 \cdot 2^{O(\sqrt{\log n} \cdot \log\log n)}\right)$. Since the number of phases is bounded by $O(\log^3 n)$, the total running time of the algorithm is bounded by $O\left(n^2 \cdot 2^{O(\sqrt{\log n} \cdot \log\log n)}\right)$, and the probability that the algorithm ever returns "FAIL" bounded by $1/\mathrm{poly}(n)$. In the remainder of the paper, we focus on the proof of Theorem 3.4.

3.3 Proof of Theorem 3.4 – High Level Overview

To prove Theorem 3.4, we use two large enough constants $c_1 \gg c_2$, whose values we set later, and we employ the following theorem.

Theorem 3.6. *For all $r \geq 1$, there is a randomized algorithm for the r-restricted* RouteAndCut *problem, that, on an input $(G, A, B, \Delta, \eta, N)$ with $|V(G)| = n$, runs in time at most $c_1 \cdot n \cdot (n - |B|) \cdot 2^{c_2\sqrt{\log N}} \cdot (\log N)^{16c_2(r-1)+8c_2}$.*

Note that, if $(G, A, B, \Delta, \eta, N)$ is an instance of the RouteAndCut problem, and $|V(G)| = n$, then, from the problem definition, $\eta \leq \Delta$ and $\frac{(n-|B|)\cdot\eta}{\Delta} \leq n$ holds. Therefore, any instance of the problem is also an instance of r^*-restricted RouteAndCut problem, for $r^* = \left\lceil\sqrt{\log n}\right\rceil$. By using Theorem 3.6 with parameter $r^* = \left\lceil\sqrt{\log n}\right\rceil$, we obtain an algorithm for the general RouteAndCut problem whose running time is $O\left(n \cdot (n - |B|) \cdot 2^{O(\sqrt{\log N}\cdot\log\log N)}\right)$.

Therefore, in order to prove Theorem 3.4, it is enough to prove Theorem 3.6. At a high level, the proof of Theorem 3.6 proceeds by induction on r, and it relies on a slight modification of the Multiplicative Weight Update (MWU) framework of [13, 16], that essentially reduces the r-restricted RouteAndCut problem to a special case of the directed SSSP problem, that we call r-Restricted s-t-SP. Before we describe this approach, we summarize a useful transformation, that allows us to reduce the number of edges in the input graph, in the following claim, whose proof appears in the full version of the paper.

Claim 3.7. *There is a deterministic algorithm, that, given an instance $(G, A, B, \Delta, \eta, N)$ of the r-restricted* RouteAndCut *problem (where graph G is given as an adjacency list), constructs another instance $(G', A, B, \Delta, \eta, N)$ of the r-restricted* RouteAndCut *problem with $G' \subseteq G$, such that $|E(G')| \leq O(n \cdot (n - |B|))$, and moreover, if $(\mathcal{Q}, (X, Y))$ is a valid solution to instance $(G', A, B, \Delta, \eta, N)$, then it is also a valid solution to instance $(G, A, B, \Delta, \eta, N)$. The running time of the algorithm is $O(n \cdot (n - |B|))$.*

In our algorithms for the RouteAndCut problem, we will use Claim 3.7 in order to ensure that the number of edges in the input graph is bounded by $O(n \cdot (n - |B|))$. We now turn to describe the MWU-based approach for solving r-restricted RouteAndCut.

3.4 Solving r-Restricted RouteAndCut

In this subsection we provide a high-level description of our algorithm for the r-restricted RouteAndCut problem, including the modified MWU framework that we use, and a reduction to a special case of the SSSP problem, that we call r-restricted s-t-SP. We show that an algorithm for the latter problem implies an algorithm for r-restricted RouteAndCut, via the modified MWU framework.

Let $(G, A, B, \Delta, \eta, N)$ be the input to the r-restricted RouteAndCut problem instance, where $G = (L, R, E)$ is a well-structured graph with $|V(G)| = n$, that is given as an adjacency list, A and B are disjoint subsets of $V(G)$, and $N \geq n$, $\Delta \geq 1$, and $1 \leq \eta \leq \Delta$ are parameters, with $|A|, |B| \geq \Delta$. Since the instance is r-restricted, $\frac{(n-|B|)\cdot\eta}{\Delta} \leq 2^{r\cdot\sqrt{\log N}}$ holds. By using the algorithm from Claim 3.7, we convert this instance into another instance $(G', A, B, \Delta, \eta, N)$ of r-restricted RouteAndCut with $|E(G')| \leq O(n \cdot (n - |B|))$, in time $O(n \cdot (n - |B|))$. From now on we focus on solving instance $(G', A, B, \Delta, \eta, N)$, and, for convenience, we denote G' by G.

A Preprocessing Step. We start with a simple preprocessing step. We greedily construct a maximal collection $\mathcal{Q}_0$ of disjoint paths, where every path connects a distinct vertex of A to a distinct vertex of B, and consists of a single regular edge. This can be done in time $O(|E(G)|) \leq O(n \cdot (n - |B|))$. Let $A_1 \subseteq A$ and $B_1 \subseteq B$ be the sets of vertices that do not serve as endpoints of any path in $\mathcal{Q}_0$. This step ensures that there is no regular edge in G connecting a vertex of A_1 to a vertex of B_1, so every path connecting a vertex of A_1 to a vertex of B_1 must contain at least one special edge.

Next, we describe a modified MWU framework that will allow us to compute a solution to the r-restricted RouteAndCut problem instance. The modified MWU framework is designed so as to reduce the r-restricted RouteAndCut problem to a special case of SSSP, that we call r-restricted s-t-SP, which seems more tractable than the general decremental SSSP problem in directed graphs. We will then design an algorithm for the r-restricted s-t-SP problem, that will allow us to obtain the desired algorithm for r-restricted RouteAndCut. The algorithm for r-restricted s-t-SP will in turn rely on an algorithm for the $(r-1)$-restricted RouteAndCut problem. The details are deferred to the full version of the paper.

3.4.1 *The Modified* MWU Framework.

We now describe an algorithm that is based on the modified MWU framework, but ignore the issue of the efficient implementation of the algorithm. We address this issue later, by reducing the problem to the r-restricted s-t-SP problem.

The algorithm uses a parameter $\Lambda = (n - |B_1|) \cdot \frac{\eta \log^5 n}{\Delta}$. While the set B_1 of vertices may change over the course of the algorithm, the value of the parameter Λ is set at the beginning of the algorithm and remains unchanged throughout the algorithm. Our algorithm maintains an assignment $\ell(e) \geq 0$ of *lengths* to the edges of G. At the beginning of the algorithm, we assign an initial length $\ell(e)$ to every edge $e \in E(G)$, as follows. If e is a regular edge, we set $\ell(e) = 0$, and if it is a special edge, we set $\ell(e) = \frac{1}{\Lambda}$. As the algorithm progresses, the lengths of the special edges may grow, but the lengths of the regular edges remain unchanged. Whenever we talk about distances between vertices and lengths of paths, it is always with respect to the current lengths $\ell(e)$ of edges $e \in E(G)$.

The algorithm gradually constructs a routing $\mathcal{Q}$ from A_1 to B_1 in G, starting with $\mathcal{Q} = \emptyset$. It performs at most Δ iterations, and in each iteration, a single path P, connecting some vertex $a \in A_1$ to some vertex $b \in B_1$, is added to $\mathcal{Q}$. We then delete a from A_1 and b from B_1. Additionally, we may double the lengths of *some* special edges on the path P. Specifically, our algorithm maintains, for every special edge e of G, a counter $n(e)$, which, intuitively, counts the number of paths that were added to $\mathcal{Q}$ that contained e, since $\ell(e)$ was last doubled (or since the beginning of the algorithm, if $\ell(e)$ was never doubled yet). At the beginning, we set $n(e) = 0$ for every special edge e. Whenever a path P is added to $\mathcal{Q}$, we increase the counter $n(e)$ of every special edge $e \in E(P)$. If, for any such edge e, the counter $n(e)$ reaches η, then we double the length of edge e, and reset the counter $n(e)$ to be 0.

The Oracle. At the heart of our algorithm is an *oracle* – an algorithm that, in every iteration, either computes a path P of length at most 1 in the current graph G connecting a vertex of A_1 to a vertex of B_1, or produces a new assignment $\ell'(e)$ for edges $e \in E(G)$, that we call a *cut-witness*, and define next. Intuitively, the cut-witness is designed in such a way that it can be easily transformed into the desired cut (X, Y) in G with $|E_G(X, Y)| \le \frac{64\Delta}{\eta \log^4 n} + \frac{256|Q|}{\eta}$, such that, if A' and B' denote the current sets A_1 and B_1 respectively, then $A' \subseteq X$ and $B' \subseteq Y$ hold. Once the oracle produces a cut-witness, the algorithm terminates.

Definition 3.8 (Cut-witness). *A* cut-witness *is an assignment of lengths $\ell'(e) \ge 0$ to every edge $e \in E(G)$, such that, if we denote by A' and B' the current sets A_1 and B_1 of vertices respectively, and by E^* the set of all special edges with both endpoints in B', then:*

- $\sum_{e \in E(G)} \ell'(e) \le \frac{\Delta}{2\eta \log^4 n} + \sum_{e \in E(G) \setminus E^*} \ell(e)$*; and*
- *the distance in graph G, with respect to edge lengths $\ell'(\cdot)$, from A' to B' is at least $\frac{1}{64}$.*

Next, we define the notion of an *acceptable path* in G, and we will require that the oracle, in every iteration, either returns an acceptable path, or returns a cut-witness.

Definition 3.9 (Acceptable Path). *A path P in the current graph G is called* acceptable *if P is a simple path, connecting a vertex of A_1 to a vertex of B_1, the length of P with respect to the current edge lengths $\ell(\cdot)$ is at most* 1*, and no inner vertices of P belong to B_1.*

We are now ready to define the oracle.

Definition 3.10 (The Oracle). *An oracle for the* MWU framework *is an algorithm that, in every iteration, either returns an* acceptable path P, or returns a cut-witness, or returns "FAIL". The probability that the oracle ever returns "FAIL" must be bounded by 1/2.

The MWU*-Based Algorithm.* We describe the modified MWU-based algorithm, denoted by ALG-MWU in Figure 1; the description excludes the implementation of the oracle.

We now show that we can use the algorithm in order to solve the r-restricted RouteAndCut problem. Observe first that, if the total running time of the oracle, over the course of all iterations, is bounded by T, then the total running time of Algorithm ALG-MWU is bounded by $O(|T|) + O(|E(G)|) \le O(|T|) + O(n \cdot (n - |B|))$. Moreover, since the probability that the oracle ever returns "FAIL" is at most $1/2$, the probability that algorithm ALG-MWU terminates with a "FAIL" is bounded by $1/2$.

Let Q be the set of paths obtained when Algorithm ALG-MWU terminates, and let $Q' = Q \cup Q_0$ be the final set of paths that we obtain. Clearly, every path in Q' connects a vertex of A to a vertex of B. The following simple observation shows that the paths in Q' cause congestion at most $4\eta \log n$, and that the endpoints of all paths in Q' are disjoint; the proof is deferred to the full version.

Observation 3.11. *The endpoints of the paths in Q' are disjoint, and the congestion caused by the paths in Q' in G is bounded by $4\eta \log n$.*

Let A' and B' denote the sets A_1 and B_1, respectively, at the end of Algorithm ALG-MWU. Notice that A' is a set of all vertices $a \in A$ that do not serve as endpoints of the paths in Q', and similarly, B' contains all vertices $b \in B$ that do not serve as endpoints of the paths in Q'. The claim below shows an algorithm, that, given a cut-witness $\{\ell'(e)\}_{e \in E(G)}$, computes a cut (X, Y) in G with $|E_G(X, Y)| \le \frac{64\Delta}{\eta \log^4 n} + \frac{256|Q|}{\eta}$, such that $A' \subseteq X$ and $B' \subseteq Y$ hold; we defer its proof to the full version.

```
Alg-MWU
• Initialize the data structures:
  – Set $Q = \emptyset$;
  – For every edge $e \in E(G)$, if $e$ is a regular edge, set $\ell(e) = 0$, otherwise set $\ell(e) = \frac{1}{\Lambda}$.
  – For every special edge $e$, set $n(e) = 0$.
• Perform at most $\Delta - |Q_0|$ iterations, where in each iteration we apply the oracle.
  – If the oracle returned "FAIL", then return "FAIL" and terminate the algorithm;
  – If the oracle returned a cut-witness, return the cut-witness and terminate the algorithm;
  – Otherwise, the oracle must have returned an acceptable path $P$ connecting a vertex $a \in A_1$ to a vertex $b \in B_1$.
    * remove $a$ from $A_1$ and $b$ from $B_1$;
    * add $P$ to $Q$;
    * for each special edge $e \in E(P)$, increase $n(e)$ by 1, and, if $n(e)$ reaches $\eta$, double $\ell(e)$ and set $n(e) = 0$.
```

Figure 1: Alg-MWU

Claim 3.12. *There is a deterministic algorithm, that, given a cut-witness $\{\ell'(e)\}_{e \in E(G)}$ for G, obtained at the end of Algorithm* ALG-MWU*, computes a cut (X, Y) in G with $|E_G(X, Y)| \le \frac{64\Delta}{\eta \log^4 n} + \frac{256|Q'|}{\eta}$, such that $A' \subseteq X$ and $B' \subseteq Y$ hold. The running time of the algorithm is $O(|E(G)|) \le O(n \cdot (n - |B|))$.*

We now discuss an efficient implementation of our algorithm for the RouteAndCut problem, given an efficient implementation of the oracle. Recall that the time required for the preprocessing step, and for computing the final cut (X, Y) by the algorithm from Claim 3.12, is bounded by $O(n \cdot (n - |B|))$. If the running time of the oracle is bounded by T, then the running time of Algorithm ALG-MWU and hence of the algorithm for the RouteAndCut problem, is bounded by $O(T + n \cdot (n - B))$. In order to obtain an efficient implementation of the oracle, we reduce it to a special case of decremented directed SSSP, that we call r-restricted s-t-SP problem. We then provide an algorithm for the latter problem, that, in turn, relies on an algorithm for $(r-1)$-restricted RouteAndCut. Due to lack of space, the details are deferred to the full version of the paper.

REFERENCES

[1] Amir Abboud, Karl Bringmann, and Nick Fischer. 2023. Stronger 3-SUM Lower Bounds for Approximate Distance Oracles via Additive Combinatorics. In *Proceedings of the 55th Annual ACM Symposium on Theory of Computing, STOC 2023, Orlando, FL, USA, June 20-23, 2023*, Barna Saha and Rocco A. Servedio (Eds.). ACM, 391–404. https://doi.org/10.1145/3564246.3585240

[2] Amir Abboud, Karl Bringmann, Seri Khoury, and Or Zamir. 2022. Hardness of Approximation in P via Short Cycle Removal: Cycle Detection, Distance Oracles, and Beyond. *arXiv preprint arXiv:2204.10465* (2022).

[3] Baruch Awerbuch, Yossi Azar, and Serge A. Plotkin. 1993. Throughput-Competitive On-Line Routing. In *34th Annual Symposium on Foundations of Computer Science, Palo Alto, California, USA, 3-5 November 1993*. 32–40.

[4] Kyriakos Axiotis, Aleksander Madry, and Adrian Vladu. 2020. Circulation Control for Faster Minimum Cost Flow in Unit-Capacity Graphs. In *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020, Durham, NC, USA, November 16-19, 2020*. 93–104.

[5] Aaron Bernstein. 2017. Deterministic Partially Dynamic Single Source Shortest Paths in Weighted Graphs. In *LIPIcs-Leibniz International Proceedings in Informatics*, Vol. 80. Schloss Dagstuhl-Leibniz-Center for Computer Science.

[6] Aaron Bernstein, Maximilian Probst Gutenberg, and Thatchaphol Saranurak. 2020. Deterministic decremental reachability, scc, and shortest paths via directed expanders and congestion balancing. In *2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, 1123–1134.

[7] Aaron Bernstein, Maximilian Probst Gutenberg, and Christian Wulff-Nilsen. 2020. Near-optimal decremental sssp in dense weighted digraphs. In *2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, 1112–1122.

[8] Li Chen, Rasmus Kyng, Yang P. Liu, Richard Peng, Maximilian Probst Gutenberg, and Sushant Sachdeva. 2022. Maximum Flow and Minimum-Cost Flow in Almost-Linear Time. In *63rd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2022, Denver, CO, USA, October 31 - November 3, 2022*. 612–623.

[9] Julia Chuzhoy and Sanjeev Khanna. 2024. A Faster Combinatorial Algorithm for Maximum Bipartite Matching. In *Proceedings of the Thirty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*.

[10] Michael B. Cohen, Aleksander Madry, Piotr Sankowski, and Adrian Vladu. 2017. Negative-Weight Shortest Paths and Unit Capacity Minimum Cost Flow in $\tilde{O}$ ($m^{10/7}$ log W) Time (Extended Abstract). In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*. 752–771.

[11] Samuel I. Daitch and Daniel A. Spielman. 2008. Faster approximate lossy generalized flow via interior point algorithms. In *Proceedings of the 40th Annual ACM Symposium on Theory of Computing, Victoria, British Columbia, Canada, May 17-20, 2008*, Cynthia Dwork (Ed.). ACM, 451–460.

[12] Dorit Dor, Shay Halperin, and Uri Zwick. 2000. All-Pairs Almost Shortest Paths. *SIAM J. Comput.* 29, 5 (2000), 1740–1759. https://doi.org/10.1137/S0097539797327908

[13] Lisa Fleischer. 2000. Approximating Fractional Multicommodity Flow Independent of the Number of Commodities. *SIAM J. Discrete Math.* 13, 4 (2000), 505–520. https://doi.org/10.1137/S0895480199355754

[14] L. R. Ford, Jr. and D. R. Fulkerson. 1956. Maximal flow through a network. *Canadian Journal of Mathematics* 8 (1956), 399–404.

[15] Harold N. Gabow. 2017. The Weighted Matching Approach to Maximum Cardinality Matching. *Fundam. Informaticae* 154, 1-4 (2017), 109–130.

[16] Naveen Garg and Jochen Könemann. 1998. Faster and Simpler Algorithms for Multicommodity Flow and Other Fractional Packing Problems. In *39th Annual Symposium on Foundations of Computer Science, FOCS '98, November 8-11, 1998, Palo Alto, California, USA*. 300–309. https://doi.org/10.1109/SFCS.1998.743463

[17] Ashish Goel, Michael Kapralov, and Sanjeev Khanna. 2010. Perfect matchings in o(n log n) time in regular bipartite graphs. In *Proceedings of the 42nd ACM Symposium on Theory of Computing, STOC 2010, Cambridge, Massachusetts, USA, 5-8 June 2010*. 39–46.

[18] Andrew V. Goldberg and Alexander V. Karzanov. 2004. Maximum skew-symmetric flows and matchings. *Math. Program.* 100, 3 (2004), 537–568.

[19] Maximilian Probst Gutenberg and Christian Wulff-Nilsen. 2020. Decremental SSSP in weighted digraphs: Faster and against an adaptive adversary. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 2542–2561.

[20] Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. 2015. Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture. In *Proceedings of the forty-seventh annual ACM symposium on Theory of computing*. 21–30.

[21] John E. Hopcroft and Richard M. Karp. 1973. An $n^{5/2}$ Algorithm for Maximum Matchings in Bipartite Graphs. *SIAM J. Comput.* 2, 4 (1973), 225–231.

[22] Oscar H. Ibarra and Shlomo Moran. 1981. Deterministic and Probabilistic Algorithms for Maximum Bipartite Matching Via Fast Matrix Multiplication. *Inf. Process. Lett.* 13, 1 (1981), 12–15.

[23] Yin Tat Lee, Satish Rao, and Nikhil Srivastava. 2013. A new approach to computing maximum flows using electrical flows. In *Symposium on Theory of Computing Conference, STOC'13, Palo Alto, CA, USA, June 1-4, 2013*. 755–764.

[24] Yin Tat Lee and Aaron Sidford. 2019. Solving Linear Programs with Sqrt(rank) Linear System Solves. *CoRR* abs/1910.08033 (2019).

[25] Yang P. Liu and Aaron Sidford. 2020. Faster energy maximization for faster maximum flow. In *Proccedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing, STOC 2020, Chicago, IL, USA, June 22-26, 2020*. 803–814.

[26] Aleksander Madry. 2010. Faster approximation schemes for fractional multicommodity flow problems via dynamic graph algorithms. In *Proceedings of the forty-second ACM symposium on Theory of computing*. 121–130.

[27] Aleksander Madry. 2013. Navigating Central Path with Electrical Flows: From Flows to Matchings, and Back. In *54th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2013, 26-29 October, 2013, Berkeley, CA, USA*. 253–262.

[28] Aleksander Madry. 2016. Computing Maximum Flow with Augmenting Electrical Flows. In *IEEE 57th Annual Symposium on Foundations of Computer Science, FOCS 2016, 9-11 October 2016, Hyatt Regency, New Brunswick, New Jersey, USA*. 593–602.

[29] Silvio Micali and Vijay V. Vazirani. 1980. An O(sqrt(|v|) |E|) Algorithm for Finding Maximum Matching in General Graphs. In *21st Annual Symposium on Foundations of Computer Science, Syracuse, New York, USA, 13-15 October 1980*. IEEE Computer Society, 17–27.

[30] Marcin Mucha and Piotr Sankowski. 2004. Maximum Matchings via Gaussian Elimination. In *45th Symposium on Foundations of Computer Science (FOCS 2004), 17-19 October 2004, Rome, Italy, Proceedings*. 248–255.

[31] Liam Roditty and Uri Zwick. 2011. On Dynamic Shortest Paths Problems. *Algorithmica* 61, 2 (2011), 389–401. https://doi.org/10.1007/s00453-010-9401-5

[32] A. Schrijver. 2003. *Combinatorial Optimization: Polyhedra and Efficiency*. Springer.

[33] Daniel A. Spielman and Shang-Hua Teng. 2004. Nearly-linear time algorithms for graph partitioning, graph sparsification, and solving linear systems. In *Proceedings of the 36th Annual ACM Symposium on Theory of Computing, Chicago, IL, USA, June 13-16, 2004*. 81–90.

[34] Jan van den Brand, Li Chen, Rasmus Kyng, Yang P. Liu, Richard Peng, Maximilian Probst Gutenberg, Sushant Sachdeva, and Aaron Sidford. 2023. A Deterministic Almost-Linear Time Algorithm for Minimum-Cost Flow. In *64th IEEE Annual Symposium on Foundations of Computer Science (FOCS), (to appear)*.

[35] Jan van den Brand, Yin Tat Lee, Danupon Nanongkai, Richard Peng, Thatchaphol Saranurak, Aaron Sidford, Zhao Song, and Di Wang. 2020. Bipartite Matching in Nearly-linear Time on Moderately Dense Graphs. In *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020, Durham, NC, USA, November 16-19, 2020*. 919–930.

[36] Vijay V. Vazirani. 1994. A Theory of Alternating Paths and Blossoms for Proving Correctness of the O(sqrt{V E}) General Graph Maximum Matching Algorithm. *Comb.* 14, 1 (1994), 71–109. https://doi.org/10.1007/BF01305952

Received 12-NOV-2023; accepted 2024-02-11