Near-Optimal Fully Dynamic Densest Subgraph

Saurabh Sawlani
Georgia Tech
Atlanta, GA, USA
sawlani@gatech.edu

Junxing Wang
CMU
Pittsburgh, PA, USA
junxingw@cs.cmu.edu

ABSTRACT

We give the first fully dynamic algorithm which maintains a $(1-\epsilon)$-approximate densest subgraph in worst-case time $\text{poly}(\log n, \epsilon^{-1})$ per update. Dense subgraph discovery is an important primitive for many real-world applications such as community detection, link spam detection, distance query indexing, and computational biology. We approach the densest subgraph problem by framing its dual as a graph orientation problem, which we solve using an augmenting path-like adjustment technique. Our result improves upon the previous best approximation factor of $(1/4 - \epsilon)$ for fully dynamic densest subgraph [Bhattacharya *et. al.*, STOC '15]. We also extend our techniques to solving the problem on vertex-weighted graphs with similar runtimes.

Additionally, we reduce the $(1-\epsilon)$-approximate densest subgraph problem on directed graphs to $O(\log n/\epsilon)$ instances of $(1-\epsilon)$-approximate densest subgraph on vertex-weighted graphs. This reduction, together with our algorithm for vertex-weighted graphs, gives the first fully-dynamic algorithm for directed densest subgraph in worst-case time $\text{poly}(\log n, \epsilon^{-1})$ per update. Moreover, combined with a near-linear time algorithm for densest subgraph [Bahmani *et. al.*, WAW '14], this gives the first near-linear time algorithm for directed densest subgraph.

CCS CONCEPTS

• **Theory of computation → Dynamic graph algorithms**.

KEYWORDS

dense subgraph discovery, fully dynamic algorithm, linear programming dual, graph orientation, data structures

ACM Reference Format:
Saurabh Sawlani and Junxing Wang. 2020. Near-Optimal Fully Dynamic Densest Subgraph. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing (STOC '20), June 22–26, 2020, Chicago, IL, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3357713.3384327

1 INTRODUCTION

A majority of real-world networks are very large in size, and a significant fraction of them are known to change rather rapidly [58]. This has necessitated the study of efficient dynamic graph algorithms - algorithms which use the existing solution to quickly find an updated solution for the new graph. Due to the size of these graphs, it is imperative that each update be processed in sub-linear time.

Data structures which efficiently maintain solutions to combinatorial optimization problems have shot into prominence over the last few decades [27, 61]. Many fundamental graph problems such as graph connectivity [36, 37, 42], maximal and maximum matchings [11, 12, 14, 15, 33], maximum flows and minimum cuts [32, 39, 64] have been shown to have efficient dynamic algorithms which only require sub-linear runtime per update. On the other hand, lower bounds exist for the update times for a number of these problems [1–4, 35]. [34] contains a comprehensive survey of many graph problems and their state-of-the-art dynamic algorithms.

In this paper, we consider the *densest subgraph problem*. Given an undirected graph $G = \langle V, E \rangle$, the *density* of a subgraph induced by $S \subseteq V$ is defined as $\rho_G(S) = |E(S)|/|S|$, where $E(S)$ is the set of all edges within S. The densest subgraph problem (DSP) asks to find a a set $S \in V$ such that

$$\rho_G^* \stackrel{\text{def}}{=} \rho_G(S) = \max_{U \subseteq V} \rho_G(U).$$

We call ρ_G^* the maximum subgraph density of G.

The densest subgraph problem has great theoretical relevance due to its close connection to fundamental graph problems such as *network flow* and *bipartite matching*[1]. While near-linear time algorithms exist for finding matchings in graphs [24, 28, 51], the same cannot be said for flows on directed graphs [49]. In this sense, DSP acts as an indicative middle ground, since it is both a specific instance of a flow problem [9, 31], as well as a generalization of bipartite b-matchings. Interestingly, DSP does allow near-linear time algorithms [9].

In terms of dynamic algorithms, the state-of-the-art data structure for maintaining $(1+\epsilon)$-approximate maximum matchings takes $O(\sqrt{m}\epsilon^{-2})$ time per update [33]. [13] maintain a constant factor approximation to the b-matching problem in $O(\log^3 n)$ time. For flow-problems, algorithms which maintain a constant factor approximation in sublinear update time have proved to be elusive.

In addition to its theoretical importance, dense subgraph discovery is an important primitive for several real-world applications such as community detection [21, 23, 46, 47, 54], link spam detection [30], story identification [6], distance query indexing [5, 22, 40] and computational biology [38, 56, 57], to name a few. Due to its practical relevance, many related notions of subgraph density, such as *k-cores* [60], *quasi-cliques* [19], *α-β-communities* [52] have been studied in the literature. [48, 63, 65] contain several other applications of dense subgraphs and related problems.

[1] We describe this connection explicitly in Sections 2 and 3.1.

STOC '20, June 22–26, 2020, Chicago, IL, USA

ACM ISBN 978-1-4503-6979-4/20/06…$15.00
https://doi.org/10.1145/3357713.3384327

1.1 Background and Related Work

As defined in [16], we say that an algorithm is a *fully dynamic γ-approximation algorithm* for the densest subgraph problem if it can process the following operations: (i) insert/delete an edge into/from the graph; (ii) query a value which is at least γ times the maximum subgraph density of the graph.

Goldberg [31] gave the first polynomial-time algorithm to solve the densest subgraph problem by reducing it to $O(\log n)$ instances of maximum flow. This was subsequently improved to use only $O(1)$ instances, using parametric max-flow [29]. Charikar [20] gave an exact linear programming formulation of the problem, while at the same time giving a simple greedy algorithm which gives a $1/2$-approximate densest subgraph (first studied in [8]). Despite the approximation factor, this algorithm is popular in practice [22] due to its simplicity, its efficacy on real-world graphs, and due to the fact that it runs in linear time and space.

Obtaining fast algorithms for approximation factors better than $1/2$, however, has proved to be a harder task. One approach towards this is to sparsify the graph in a way that maintains subgraph densities [50, 53] within a factor of $1-\epsilon$, and run the exact algorithm on the sparsifier. However, this algorithm still incurs a term of $n^{1.5}$ in the running time, causing it to be super-linear for sparse graphs. A second approach is via numerical methods to solve positive LPs[2] approximately. Bahmani *et al.* [9] gave a $O(m \log n \cdot \epsilon^{-2})$ algorithm by bounding the width of the dual LP for this problem, and using the multiplicative weights update framework [7, 55] to find an $(1-\epsilon)$-approximate solution. Su and Vu [62] used a similar technique to obtain an efficient distributed $(1-\epsilon)$-approximation algorithm. Alternately, using accelerated methods to solve positive LPs [17] gives a $\widetilde{O}(m\Delta\epsilon^{-1})$ algorithm[3], where Δ is the maximum degree in the input graph.

In terms of dynamic and streaming algorithms for the densest subgraph problem, the first result is by Bahmani *et al.* [10], where they modified Charikar's greedy algorithm to give a $(1/2-\epsilon)$-approximation using $O(\log_{1+\epsilon} n)$ passes over the input. Das Sarma *et al.* [59] adapted this idea to maintain a $(1/2-\epsilon)$ approximate densest subgraph efficiently in the distributed CONGEST model. Using the same techniques as in the static case, Bahmani *et al.* [9] obtained a $(1-\epsilon)$-approximation algorithm that requires $O(\log n\epsilon^{-2})$ passes over the input.

Subsequently, Bhattacharya *et al.* [16] developed a more nuanced data structure to enable a 1-pass streaming algorithm which finds a $(1/2-\epsilon)$ approximation. They also gave the first dynamic algorithm for DSP - a fully dynamic $(1/4-\epsilon)$ approximation algorithm using amortized time $O(\text{poly}(\log n, \epsilon^{-1}))$ per update. Around the same time, Epasto *et al.* [25] gave a fully dynamic $(1/2-\epsilon)$-approximation algorithm for DSP in amortized time $O(\log^2 n\epsilon^{-2})$ per update, with the caveat that edge deletions can only be random.

Kannan and Vinay [41] defined a notion of density on directed graphs, and subsequently gave a $O(\log n)$ approximation algorithm for the problem. Charikar [20] gave a polynomial-time algorithm for directed DSP by reducing the problem to solving $O(n^2)$ LPs. On the other hand, Khuller and Saha [43] used parametrized maximum flow to derive a polynomial-time algorithm. In the same paper, the gave a linear time 2-approximation algorithm for the problem.

[2] A positive linear program is one in which all coefficients, variables and constraints are non-negative. They are alternatively known as Mixed Packing and Covering LPs.
[3] $\widetilde{O}$ hides polylogarithmic factors in n.

1.2 Our Results

We use a "dual" interpretation of the densest subgraph problem to gain insight on the optimality conditions, as in [9, 20]. Specifically, we translate it into a problem of assigning edge loads to incident vertices so as to minimize the maximum load across vertices. Viewed another way, we want to orient edges in a directed graph so as to minimize the maximum in-degree of the graph. This view gives a local condition for near-optimality of the algorithm, which we then leverage to design a data structure to handle updates efficiently. As our primary result, we give the first fully dynamic $(1-\epsilon)$-approximation algorithm for DSP which runs in $O(\text{poly}(\log n, \epsilon^{-1}))$ worst-case time per update:

Theorem 1.1. *Given a graph G with n vertices, there exists a deterministic fully dynamic $(1+\epsilon)$-approximation algorithm for the densest subgraph problem using $O(1)$ worst-case time per query and $O(\log^4 n \cdot \epsilon^{-6})$ worst-case time per edge insertion or deletion.*

Moreover, at any point, the algorithm can output the corresponding approximate densest subgraph in time $O(\beta + \log n)$, where β is the number of vertices in the output.

Charikar [20] gave a reduction from the densest subgraph problem on directed graphs to solving a number of instances of an LP. We visualize this LP as DSP on a vertex-weighted graph. We show that our approach on unweighted graphs extends naturally to those with vertex weights, thereby also giving a fully dynamic $(1-\epsilon)$-approximation algorithm for directed DSP which runs in $O(\text{poly}(\log n, \epsilon^{-1}))$ worst-case time per update:

Theorem 1.2. *Given a directed graph G with n vertices, there exists a deterministic fully dynamic $(1-\epsilon)$-approximation algorithm for the densest subgraph problem on G using $O(\log n/\epsilon)$ worst-case query time and worst-case update times of $O(\log^5 n \cdot \epsilon^{-7})$ per edge insertion or deletion.*

Moreover, at any point, the algorithm can output the corresponding approximate densest subgraph in time $O(\beta + \log n)$, where β is the number of vertices in the output.

1.3 Organization

In Section 2, we define essential notation, and formulate DSP as a linear program. In Section 3, we give our primary result - a fully dynamic $1+\epsilon$ approximation algorithm for DSP with updates in worst-case time $\text{polylog}(n, \epsilon^{-1})$. In Section 4, we extend our results from Section 3 to vertex-weighted graphs. In Section 5, we give a detailed reduction from directed DSP to undirected vertex-weighted DSP.

2 PRELIMINARIES

We represent any undirected graph G as $G = \langle V, E \rangle$, where V is the set of vertices in G, E is the set of edges in G. For any subset of vertices $S \subseteq V$, we denote using $E(S)$ the subset of all edges within S.

We define $\rho_G(S)$ as the density of subgraph induced by S in G, i.e.,

$$\rho_G(S) \stackrel{\text{def}}{=} \frac{|E(S)|}{|S|}.$$

The maximum subgraph density of G, ρ^*_G, is simply the maximum among all subgraph densities, i.e.,

$$\rho^*_G \stackrel{\text{def}}{=} \max_{S \subseteq V} \rho_G(S).$$

2.1 LP Formulation and Dual

The following is a well-known LP formulation of the densest subgraph problem, introduced in [20]. Associate each vertex v with a variable $x_v \in \{0, 1\}$, where $x_v = 1$ signifies v being included in S. Similarly, for each edge, let $y_e \in \{0, 1\}$ denote whether or not it is in $E(S)$. Relaxing the variables to be real numbers, we get the following LP, which we denote by $\textsc{Primal}(G)$, whose optimal is known to be ρ^*_G.

$\textsc{Primal}(G)$

$$\begin{aligned} \text{maximize} \quad & \sum_{e \in E} y_e \\ \text{subject to} \quad & y_e \le x_u, x_v, && \forall e = uv \in E \\ & \sum_{v \in V} x_v \le 1, \\ & y_e \ge 0, x_v \ge 0, && \forall e \in E, \forall v \in V \end{aligned}$$

As in [9, 62], we take greater interest in the dual of the above problem. Let $f_e(u)$ be the dual variable associated with the first $2m$ constraints of the form $y_e \le x_u$ in $\textsc{Primal}(G)$, and let D be associated with the last constraint. We get the following LP, which we denote by $\textsc{Dual}(G)$.

$\textsc{Dual}(G)$

$$\begin{aligned} \text{minimize} \quad & D \\ \text{subject to} \quad & f_e(u) + f_e(v) \ge 1, && \forall e = uv \in E \\ & \sum_{e \ni v} f_e(v) \le D, && \forall v \in V \\ & f_e(u) \ge 0, f_e(v) \ge 0, && \forall e = uv \in E \end{aligned}$$

This LP can be visualized as follows. Each edge $e = uv$ has a load of 1, which it wants to assign to its end points: $f_e(u)$ and $f_e(v)$ such that the total load on each vertex is at most D. The objective is to find the minimum D for which such a load assignment is feasible.

For a fixed D, the above formulation resembles a bipartite graph between edges and vertices. Then, the problem is similar to a bipartite b-matching problem [13], where the demands on one side are at most D, and the other side are at least 1.

From strong duality, we know that the optimal objective values of both linear programs are equal, i.e., exactly ρ^*_G. Let ρ_G be the objective of any feasible solution to $\textsc{Primal}(G)$. Similarly, let $\hat{\rho}_G$ be the objective of any feasible solution to $\textsc{Dual}(G)$. Then, by optimality of ρ^*_G and weak duality,

$$\rho_G \le \rho^*_G \le \hat{\rho}_G. \tag{1}$$

3 FULLY DYNAMIC ALGORITHM

In this section, we describe the main result of the paper: a deterministic fully-dynamic algorithm which maintains a $(1-\epsilon)$-approximation to the densest subgraph problem in $\text{poly}(\log n, \epsilon^{-1})$ worst-case time per update.

3.1 Intuition and Overview

At a high level, our approach is to view the densest subgraph problem via its dual problem, i.e., "assigning" each edge fractionally to its endpoints (as we discuss in Section 2). We view this as a load distribution problem, where each vertex is assigned some load from its incident edges. Then, the objective of the problem is simply to find an assignment such that the maximum vertex load is minimized. It is easy to verify that an optimal load assignment in the dual problem is achieved when no edge is able to reassign its load such that the maximum load among its two endpoints gets reduced. In other words, local optimality implies global optimality.

In fact, this property holds even for approximately optimal solutions. We show in Section 3.2 that any solution $\boldsymbol{f}$ which satisfies an η-additive approximation to local optimality guarantees an approximate global optimal solution with a multiplicative error of at most $1 - O(\sqrt{\eta \log n / \hat{\rho}_G})$, where $\hat{\rho}_G$ denotes the maximum vertex load in $\boldsymbol{f}$. Here, an η-additive approximation implies that for any edge, the maximum among its endpoint loads can only be reduced by at most η by reassigning the edge. So, given an estimate of $\hat{\rho}_G$ and a desired approximation factor ϵ, we can deduce the required slack parameter η, which we will alternatively denote as a function $\eta(\hat{\rho}_G, \epsilon)$.

To do away with fractional edge assignments, in Section 3.3 we scale up the graph by duplicating each edge an appropriate number of times. When η is an integer, one can always achieve an η-additive approximation to local optimality by assigning each edge completely to one of its endpoints. We visualize such a load assignment via a directed graph, by orienting each edge towards the vertex to which it is assigned. Now, the load on every vertex v is simply its in-degree $\boldsymbol{d}_{\text{in}}(v)$. Then, an η-approximate local optimal solution is achieved by orienting each edge such that there is no edge $\overrightarrow{uv}$ with $\boldsymbol{d}_{\text{in}}(v) - \boldsymbol{d}_{\text{in}}(u) > \eta$, because otherwise, we can flip the edge to achieve a better local solution. Let us call this a *locally η-stable oriented graph*.

This leaves the following challenges in extending this idea to a fully dynamic algorithm:

(1) How can we maintain a *locally η-stable oriented graph* under insertion/deletion operations efficiently?
(2) How do we maintain an accurate estimate of η while the graph (and particularly $\hat{\rho}_G$) undergoes changes?

In Sections 3.4 and 3.5, we solve the first issue using a technique similar to that used by Kopelowitz *et al.* [44] for the graph orientation problem. When an edge is inserted or deleted, it causes a vertex to change its in-degree, which might cause an incident edge to break the invariant for local η-stability. If we flip the edge to fix this instability, it might cause further instabilities. To avoid this cascading of unstable edges, we first identify a maximal chain of "tight" edges - edges that are close to breaking the local stability constraint, and flip all edges in such a chain. This way, we only increment the degree of the last vertex in the chain. Since the chain

was maximal, this increment maintains the stability condition. By defining a "tight" edge appropriately, and applying the same argument to the deletion operation, we show that each update incurs at most $O(\hat{\rho}_G/\eta)$ flips. This chain of tight edges closely relates to the concept of augmenting paths in network flows [26] and matchings [24, 51], which seems fitting, considering our intuition that densest subgraph relates closely to these problems.

In Section 3.6, we solve the second issue - by simply running the algorithm for $O(\log n)$ values of η, and using the appropriate version of the algorithm to query the solution.

3.2 Sufficiency of Local Approximation

From Equation 1, we know that the optimal solution to $\textsc{Dual}(G)$ gives the exact maximum subgraph density of G, ρ^*_G. Let us interpret the variables of $\textsc{Dual}(G)$ as follows:

- Every edge $e = uv$ assigns itself fractionally to one of its two endpoints. $f_e(u)$ and $f_e(v)$ denote these fractional loads.
- $\sum_{e \ni v} f_e(v)$ is the total load assigned to v. We denote this using ℓ_v.
- The objective is simply $\max_{v \in V} \ell_v$.

If there is any edge $e = uv$ such that $f_e(u) > 0$ and $\ell_u > \ell_v$. Then e can transfer an infinitesimal amount of load from u to v while not increasing the objective. Hence, there always exists an optimal solution where for any edge $e = uv$, $f_e(u) > 0 \implies \ell_u \le \ell_v$. Using this intuition, we write the approximate version of $\textsc{Dual}(G)$ by providing a slack of η to the above condition. We call this relaxed LP as $\textsc{Dual}(G, \eta)$.

$$\textsc{Dual}(G, \eta)$$

$$\begin{aligned} \ell_v &= \sum_{e \ni v} f_e(v) && \forall u \in V \\ f_e(u) + f_e(v) &= 1, && \forall e = uv \in E \\ f_e(u), f_e(v) &\ge 0, && \forall e = uv \in E \\ \ell_u &\le \ell_v + \eta, && \forall e = uv \in E, f_e(u) > 0 \end{aligned}$$

Theorem 3.1 states that this local condition is, in fact, also sufficient to achieve global near-optimality. Specifically, it shows that $\textsc{Dual}(G, \eta)$ provides a $1/(1-\epsilon)$-approximation to ρ^*_G, where η is a parameter depending on ϵ described later. Kopelowitz *et al.* [44] use an identical argument to show the sufficiency of local optimality for the graph orientation problem.

Theorem 3.1. *Given an undirected graph G with n vertices, let $\hat{\boldsymbol{f}}, \hat{\boldsymbol{\ell}}$ denote any feasible solution to $\textsc{Dual}(G, \eta)$, and let $\hat{\rho}_G \stackrel{\text{def}}{=} \max_{v \in V} \hat{\ell}_v$. Then,*

$$\left(1 - 3\sqrt{\frac{\eta \log n}{\hat{\rho}_G}}\right) \cdot \hat{\rho}_G \le \rho^*_G \le \hat{\rho}_G.$$

Proof. Any feasible solution of $\textsc{Dual}(G, \eta)$ is also a feasible solution of $\textsc{Dual}(G)$, and so we have $\rho^*_G \le \hat{\rho}_G$.

Denote by T_i the set of vertices with load at least $\hat{\rho}_G - \eta i$, i.e., $T_i \stackrel{\text{def}}{=} \left\{v \in V \mid \hat{\ell}_v \ge \hat{\rho}_G - \eta i\right\}$. Let $0 < r < 1$ be some adjustable parameter we will fix later. We define k to be the maximal integer such that for any $1 \le i \le k$, $|T_i| \ge |T_{i-1}|(1+r)$. Note that such a maximal integer k always exists because there are finite number of vertices in G and the size of T_i grows exponentially. By the maximality of k, $|T_{k+1}| < |T_k|(1+r)$. In order to bound the density of this set T_{k+1}, we compute the total load on all vertices in T_k. For any $u \in T_k$, the load on u is given by

$$\hat{\ell}_u = \sum_{uv \in E} \hat{f}_{uv}(u).$$

However, we know that $f_{uv}(u) > 0 \implies \hat{\ell}_v \ge \hat{\ell}_u - \eta$, and hence we only need to count for $v \in T_{k+1}$. Summing over all vertices in T_{k+1}, we get

$$\sum_{u \in T_k} \hat{\ell}_u = \sum_{u \in T_k, v \in T_{k+1}} \hat{f}_{uv}(u) \le \sum_{u \in T_{k+1}, v \in T_{k+1}} \hat{f}_{uv}(u) = |E(T_{k+1})|.$$

Consider the density of set T_{k+1},

$$\rho_G(T_{k+1}) = \frac{|E(T_{k+1})|}{|T_{k+1}|} \ge \frac{\sum_{u \in T_k} \hat{\ell}_u}{|T_{k+1}|} \ge \frac{|T_k| \cdot (\hat{\rho}_G - \eta k)}{|T_{k+1}|},$$

where the last inequality follows from the definition of T_k.

Using the fact that $|T_k|/|T_{k+1}| > 1/(1+r) \ge 1 - r$,

$$\rho_G(T_{k+1}) \ge (1-r)(\hat{\rho}_G - \eta k) \ge \hat{\rho}_G(1-r)\left(1 - \frac{2\eta \log n}{r \cdot \hat{\rho}_G}\right),$$

where the last inequality comes from the fact that $n \ge |T_k| \ge (1+r)^k$, which implies that $k \le \log_{1+r} n \le 2 \log n / r$.

Now, we can set our parameter r to maximize the term on the RHS. By symmetry, the maximum is achieved when both terms in the product are equal and hence we set

$$r = \sqrt{\frac{2\eta \log n}{\hat{\rho}_G}}.$$

This gives

$$\begin{aligned} \rho_G(T_{k+1}) &\ge \hat{\rho}_G \cdot \left(1 - \sqrt{\frac{2\eta \log n}{\hat{\rho}_G}}\right)^2 \\ &\ge \hat{\rho}_G \cdot \left(1 - 2\sqrt{\frac{2\eta \log n}{\hat{\rho}_G}}\right) \\ &\ge \hat{\rho}_G \cdot \left(1 - 3\sqrt{\frac{\eta \log n}{\hat{\rho}_G}}\right). \end{aligned}$$

Lastly, since $\rho_G(T_{k+1})$ can be at most the maximum subgraph density ρ^*_G, the theorem follows. □

The set T_{k+1}, in the above proof, is actually a subgraph of G with density at least $\rho^*_G(1 - 3\sqrt{\eta \log n / \hat{\rho}_G})$. However, we need the exact value of $\hat{\rho}_G$ to find this set. As we will see in later sections, we will only have access to an estimate ρ^{est} of the form: $\rho^{\text{est}} \le \hat{\rho}_G \le 2\rho^{\text{est}}$. So, if we instead set

$$r = \sqrt{\frac{2\eta \log n}{\rho^{\text{est}}}}, \tag{2}$$

we get

$$\begin{aligned} \rho_G(T_{k+1}) &\ge \hat{\rho}_G \cdot \left(1 - \sqrt{\frac{2\eta \log n}{\hat{\rho}_G}}\right)\left(1 - 2\sqrt{\frac{\eta \log n}{\hat{\rho}_G}}\right) \\ &\ge \hat{\rho}_G \cdot \left(1 - 4\sqrt{\frac{\eta \log n}{\hat{\rho}_G}}\right). \end{aligned}$$

Using the fact that $\hat{\rho}_G \geq \rho^*_G, \rho^{\text{est}}$ gives us the following corollary.

Corollary 3.2.

$$\rho_G(T_{k+1}) \geq \rho^*_G \cdot \left(1 - 4\sqrt{\frac{\eta \log n}{\rho^{est}}}\right),$$

where T_{k+1} is as defined in the proof of Theorem 3.1, using the value of r as defined in (2).

We can now set η corresponding to the desired error ϵ and the estimate ρ^{est}.

3.3 Equivalence to the Graph Orientation Problem

To obtain an ϵ approximation, we need to set $\eta = \frac{\epsilon^2 \rho^{\text{est}}}{16 \log n}$. For simpler analysis and to avoid working with fractional loads, we duplicate each edge $\alpha \stackrel{\text{def}}{=} \frac{64 \log n}{\epsilon^2}$ times. By doing this, we ensure that $\rho^{\text{est}} \geq \hat{\rho}_G/2 \geq \rho^*_G/2 \geq \alpha/4$, and thus, $\eta \geq 1$. This means we can do away with fractional assignments of edges and so each edge u, v is now assigned to either u or v. We can now frame the question as follows:

> Given an undirected graph G and an integer η, we want to assign directions to edges in such a way that for any edge $\overrightarrow{uv}$,
>
> $$\boldsymbol{d}_{\text{in}}(v) \leq \boldsymbol{d}_{\text{in}}(u) + \eta.$$

The above graph orientation problem, i.e., dynamically orienting edges of a graph to minimize the maximum in-degree, is well studied [18, 44, 45]. Kopelowitz *et al.* give an efficient dynamic algorithm for the problem, where the update time depends on the arboricity[4] of the graph with worst-case time bounds. Our technique for inserting and deleting edges mimics the algorithm by Kopelowitz *et al.* [44]. However, for our problem, the slack parameter η grows linearly with the maximum vertex load. Hence, we can exploit this additional power to arrive at worst-case times independent of any measure of actual density in the graph. Additionally, to bound the cost of a vertex informing its updated degree to its neighbors, we use a lazy round-robin informing technique, in which not all neighbors are always informed of the latest updates. We expand on these details in the rest of the section.

3.4 Data Structure for Edge Flipping in Directed Graphs

At the lowest level, we want to build a data structure that maintains a directed graph undergoing changes. Ideally, we want each vertex to know its neighbors' labels, so that we can quickly find any edge violating or exactly satisfying the approximation condition. We refer to the latter as a *tight* edge. However, this property is expensive because each vertex could possibly have too many neighbors to inform. Specifically, each vertex could have up to $\hat{\rho}_G$ in-neighbors and as many as $n-1$ out-neighbors.

[4] Arboricity is an alternate measure of density defined as $\alpha_G(V) = |E(V)|/(|V|-1)$, and is within $O(1)$ of our density measure.

We deal with this issue in the following way. Since a vertex can have $\Omega(n)$ out-neighbors, it does not inform its changes to its out-neighbors, but only its in-neighbors. So, any vertex remembers the labels of its out-neighbors. Hence, it is easy to find a tight outgoing edge; however, to find a tight incoming edge, we need to query the labels of all its in-neighbors. Hence, both the update subroutines and finding a tight incoming edge - use as many as $\hat{\rho}_G$ operations.

However, $\hat{\rho}_G$ can also get prohibitively large when the graph sees many insertions, and can reach $\Omega(n)$ (e.g. in a clique). To tackle this, we relax the requirement for tightness of an edge: we say that an edge $\overrightarrow{uv}$ is tight if $\boldsymbol{d}_{\text{in}}(v) \geq \boldsymbol{d}_{\text{in}}(u) + \eta/2$. Now, finding a tight edge becomes less strict - importantly it now suffices to update one's in-neighbors (or query one's in-neighbors) once every $\eta/4$ iterations. So, in each update, a vertex v only informs $4\boldsymbol{d}_{\text{in}}(v)/\eta$ of its neighbors in round-robin fashion. This reduces the number of operations to $O(\alpha)$ per update, as desired.

Lemma 3.3. *There exists a data structure* LazyDirectedLabels(G, η) *which can maintain a directed graph $G(V, E)$, appended with vertex labels $d : V \mapsto \mathbb{Z}^+$ while undergoing the following operations:*

- `add` $\left(\overrightarrow{uv}\right)$*: add an edge into G,*
- `remove` $\left(\overrightarrow{uv}\right)$*: remove an edge from G,*
- `increment`(u)*: increment $d(u)$ by 1,*
- `decrement`(u)*: decrement $d(u)$ by 1,*
- `flip` $\left(\overrightarrow{uv}\right)$*: flip the direction of an edge in G,*
- `tight_in_nbr`(u)*: find an in-neighbor v with $d(v) \leq d(u) - \eta/2$, and*
- `tight_out_nbr`(u)*: find an out-neighbor v with $d(v) \geq d(u) + \eta/2$.*
- `label`(u)*: output $d(u)$.*
- `max_label`$()$*: output $\max_{v \in V} d(v)$.*
- `maximal_label_set`(r)*: Output all elements with labels $\geq$* `max_label`$() - \eta \cdot i$*, where i is the smallest integer such that $|\text{labels} \geq \eta \cdot (i+1)| < (1+r)|\text{labels} \geq \eta \cdot i|$.*

Moreover, the operations `add`, `remove` *and* `flip` *can be processed in $O(\log n)$ time;* `tight_in_nbr`, `increment` *and* `decrement` *can be processed in $O(\alpha)$ time; and* `tight_out_nbr` *and* `max_label` *can be processed in $O(1)$ time.* `maximal_label_set` *can be processed in time in the order of the output size.*

The pseudocode for this data structure is in Algorithm 1.

Proof. The correctness of the data structure follows from the description in Algorithm 1. The operation `add` involves inserting an element into a list and a priority queue - giving a worst-case runtime of $O(\log n)$. The runtimes for `remove` and `flip` follow similarly. The operations `increment` and `decrement` involve 1 update to a balanced BST and $O(\alpha)$ priority-queue updates, giving a worst-case runtime of $O(\alpha \log n)$ per call. `tight_in_nbr` queries $O(\alpha)$ neighbors, resulting in a worst-case runtime of $O(\alpha)$ per call. `tight_out_nbr`, `label` and `max_label` simply check an element pointer, resulting in a $O(1)$ runtime. Lastly, `maximal_label_set` traverses a balanced BST, until it exceeds the desired threshold. The time taken is $O(\beta + \log n)$ where β is the number of elements read, which is also the size of the output. □

We maintain the following global data structure:

- LABELS: Balanced binary search tree with all labels. We store the max element separately.

Each vertex u maintains the following data structures:

- $d(u)$: u's label, initialized to 0.
- INNBRS_u: List of u's in-neighbors, initialized to $\emptyset$.
- OUTNBRS_u: Max-priority queue of u's out-neighbors indexed using d_u, initialized to $\emptyset$.

Operation `add`$(\overrightarrow{uv})$
  Add u to INNBRS_v
  Add v to OUTNBRS_u with key $d_u(v) \leftarrow d(v)$

Operation `remove`$(\overrightarrow{uv})$
  Remove u from INNBRS_v
  Remove v from OUTNBRS_u

Operation `flip`$(\overrightarrow{uv})$
  `remove`$(\overrightarrow{uv})$
  `add`$(\overrightarrow{vu})$

Operation `increment`(u)
  $d(u) \leftarrow d(u) + 1$
  Update $d(u)$ in LABELS
  for $v \in$ *the next* $\frac{4\boldsymbol{d}_{in}(u)}{\eta}$ INNBRS_u **do**
    Update $d_v(u) \leftarrow d(u)$ in OUTNBRS_v

Operation `decrement`(u)
  $d(u) \leftarrow d(u) - 1$
  Update $d(u)$ in LABELS
  for $v \in$ *the next* $\frac{4\boldsymbol{d}_{in}(u)}{\eta}$ INNBRS_u **do**
    Update $d_v(u) \leftarrow d(u)$ in OUTNBRS_v

Operation `tight_in_nbr`(u)
  for $v \in$ *the next* $\frac{4\boldsymbol{d}_{in}(u)}{\eta}$ INNBRS_u **do**
    if $d(v) \le d(u) - \eta/2$ **then return** v
  return null

Operation `tight_out_nbr`(u)
  $t \leftarrow \text{OUTNBRS}[u]$.max
  if $d_u(v) \ge d(u) + \eta/2$ **then return** v
  else return null

Operation `label()`
  return $d(u)$

Operation `max_label()`
  return LABELS.max

Operation `maximal_label_set`(r)
  $m \leftarrow$ `max_label()`
  do
    $A \leftarrow$ elements $\ge m - \eta$ in LABELS
    $B \leftarrow$ elements $\ge m - 2\eta$ in LABELS
    $m \leftarrow m - \eta$
  while $|B|/|A| \ge 1 + r$
  return B

Algorithm 1: LAZYDIRECTEDLABELS(G, η): A data structure to maintain a directed graph with vertex labels. V and η are known.

3.5 Fully Dynamic Algorithm for a Given Density Estimate

Here, we assume that an estimate of ρ_G^* (equivalently, an estimate of $\hat{\rho}_G$) is known. We denote this estimate as ρ^{est}, where $\rho^{\text{est}} \le \hat{\rho}_G \le 2\rho^{\text{est}}$. Using this, we can compute the appropriate $\eta(\rho^{\text{est}}, \epsilon) \stackrel{\text{def}}{=} 2\rho^{\text{est}}/\alpha$. Recall that α was defined as $\alpha \stackrel{\text{def}}{=} 64 \log n \cdot \epsilon^{-2}$. From Section 3.4, we have an efficient data structure to maintain a directed graph, which we will use to maintain a locally η-stable orientation. This in turn gives a fully dynamic algorithm which processes updates efficiently, as we explain below.

We first define a *tight edge* in a locally stable oriented graph:

Definition 3.4. An edge $\overrightarrow{uv}$ is said to be *tight* if $\boldsymbol{d}_{\text{in}}(v) \ge \boldsymbol{d}_{\text{in}}(u) + \eta/2$.

Now, consider inserting an edge $\overrightarrow{xy}$ into a locally η-stable oriented graph. Since y's in-degree increases, it could potentially have an in-neighbor z such that $\boldsymbol{d}_{\text{in}}(z) < \boldsymbol{d}_{\text{in}}(y) - \eta$. Note that for this to happen, $\overrightarrow{zy}$ was necessarily a tight edge. To "fix" this break in stability, we flip the edge yz; however, this causes w's in-degree to increase, which we now possibly need to fix. Before explaining how we circumvent this issue, let us define a *maximal tight chain*.

Definition 3.5. A maximal tight chain *from* a vertex v is a path of tight edges $\overrightarrow{uv_1}, \overrightarrow{v_1v_2}, \ldots, \overrightarrow{v_kw}$, such that w has no tight outgoing edges.

A maximal tight chain *to* a vertex v is a path of tight edges $\overrightarrow{wv_1}, \overrightarrow{v_1v_2}, \ldots, \overrightarrow{v_kv}$, such that w has no tight incoming edges.

Now, instead of fixing the "unstable" edge caused by the increase in y's in-degree right away, we instead find a maximal tight chain *to* y and flip all the edges in the chain. This way, the in-degrees of all vertices in the chain except the start remain the same. Due to the maximality of the chain, the start of the chain has no incoming tight edges, and hence increasing its in-degree by 1 will not break local stability. The same argument holds when we delete $\overrightarrow{xy}$, except we find a maximal tight chain *from* y.

The approximate density is nothing but the highest load in the graph. For querying the actual subgraph itself, we use the observation from Section 3.2, where the required subgraph can be found by: (i) finding sets of vertices with load at most $\eta \cdot i$ less than the maximum (T_i), and (ii) returning the first T_{i+1} such that $|T_{i+1}|/|T_i| < 1+r$, where r is an appropriate function of η.

Lemma 3.6. *There exists a data structure* THRESHOLD(G, η) *which can maintain an undirected graph* $G(V, E)$ *while undergoing the following operations:*

- `insert`(u, v)*: insert an edge into* G,
- `delete`(u, v)*: delete an edge from* G*, and report the vertex with decreased load*
- `query_load`(u)*: output the current load of* u.
- `query_density`$()$*: output a value* ρ_{out} *such that* $(1-\epsilon)\rho_G^* \leq \rho_{\text{out}} \leq \rho_G^*$.
- `query_subgraph`$()$*: output a subgraph with density at least* $(1-\epsilon)\rho_G^*$.

Moreover, the operation `insert` *takes* $O(\alpha^2)$ *time,* `delete` *takes* $O(\alpha \log n)$ *time,* `query` *takes* $O(1)$ *time, and* `query_subgraph` *takes* $O(\beta + \log n)$ *time, where* β *is the size of the output.*

The pseudocode for this data structure is in Algorithm 2.

Let us denote by $d_u(v)$, the apparent label of v as seen by u. This concept is needed because when the label of a vertex changes, it doesn't relay this change to all its in-neighbors immediately. However, we can claim the following:

Lemma 3.7. *The local gap constraint is always maintained, i.e., for any edge* $\overrightarrow{uv}$, $d(v) \leq d(u) + \eta$.

Proof. There are two ways that this invariant could become unsatisfied: via a decrement to u or an increment to v.

Recall that v informs each in-neighbor its label once every $\eta/4$ updates, hence $|d_u(v) - d(v)|$ cannot be larger than $\eta/4$. u only decrements if it cannot find a tight out-neighbor, which means that $d_u(v) < d(u) + \eta/2$. Hence, at any instant that $d(u)$ is decremented, $d(v) \leq d(u) + 3\eta/4$.

On the other hand, $d(v)$ is only incremented if v cannot find a tight in-neighbor. Consider the last time that $d(u)$ is decremented before this instant. At this point, $d(v) \leq d(u) + 3\eta/4$. After this, there can only be less than $\eta/4$ increments of $d(v)$ before it queries $d(u)$ and flips. Hence, $d(v) \leq d(u) + \eta$. □

Using Lemma 3.7 and Corollary 3.2, we get the following corollary, which shows the correctness of Lemma 3.6.

Corollary 3.8. *Let* $\rho_{\text{out}} = (1-\epsilon)\max_{v \in V} d(v)$. *Then,* $(1-\epsilon)\rho_G^* \leq \rho_{\text{out}} \leq \rho_G^*$.

Proof of Lemma 3.6. Corollary 3.8 gives the correctness proof. It remains to show the time bounds. Note that in both `insert` and `delete` operations, the maximum chain of tight edges can only be of length at most $2\hat{\rho}_G/\eta = O(\alpha)$. The `insert` operation calls `add` and `increment` once, `flip` and `tight_in_nbr` $O(\alpha)$ times. From Lemma 3.3, this results in a worst-case runtime of $O(\alpha^2)$ per insertion. The `delete` operation calls `remove` and `decrement` once, `flip` and `tight_out_nbr` $O(\alpha)$ times. From Lemma 3.3, this results in a worst-case runtime of $O(\alpha \cdot \log n)$ per deletion. `query_density` only needs one `max_label` call which is $O(1)$ worst-case. `query_load` also needs one `label` call which is $O(1)$ worst-case. Lastly, `query_subgraph`'s runtime follows from Lemma 3.3. □

3.6 Overall Algorithm

Now, we have a sufficient basis to show our main theorem, which we restate:

Theorem 1.1. *Given a graph* G *with* n *vertices, there exists a deterministic fully dynamic* $(1+\epsilon)$*-approximation algorithm for the densest subgraph problem using* $O(1)$ *worst-case time per query and* $O(\log^4 n \cdot \epsilon^{-6})$ *worst-case time per edge insertion or deletion.*

Moreover, at any point, the algorithm can output the corresponding approximate densest subgraph in time $O(\beta + \log n)$*, where* β *is the number of vertices in the output.*

From Section 3.5, we now have an efficient fully dynamic data structure Threshold$(G, \rho^{\text{est}}, \epsilon)$ to maintain a $1-\epsilon$ approximation to the maximum subgraph density, provided the optimum remains within a constant factor of some estimate ρ^{est}. Particularly, Threshold$(G, \rho^{\text{est}}, \epsilon)$ requires $\hat{\rho}_G/\eta$ to be small to work efficiently. On the other hand, too small an η results in a bad approximation factor.

To ensure that we always work with the right estimate ρ^{est}, we will construct $\log_2 n$ copies of Threshold, one copy for each possible ρ^{est}, or equivalently each possible value of η. In the ith copy of the data structure, we set $\rho_i^{\text{est}} \leftarrow 2^{i-2}\alpha$, and so $\eta_i \leftarrow 2^{i-1}$. Let us call this ith copy of the data structure as $\mathcal{T}_i \leftarrow$ Threshold$(G, \rho_i^{\text{est}}, \epsilon)$. We also define $\eta_0 = 0$ for the sake of the empty graph.

We say that $\mathcal{T}_i$ is *accurate* if $\rho_i^{\text{est}} \leq \hat{\rho}_G$, or equivalently $\eta_i \leq 2\hat{\rho}_G/\alpha$. Note that we will never use a copy that is not accurate to deduce the approximate solution. On the other hand, we say that $\mathcal{T}_i$ is *affordable* if the maximum possible chain length is less than 2α, i.e., $\eta_i > \hat{\rho}_G/\alpha$. On copies that are not affordable, if there are any additions which can cause the maximum load *in that copy* to increase, we hold these off until a later time.

Lastly, note that for any value of $\hat{\rho}_G$, there is exactly one copy which is both accurate and affordable. We call this the *active* copy. The solution is extracted at any point from this copy. Suppose the index of the current active copy is i. Then, after an insertion, this can be either i or $i+1$. We first test this by querying the maximum density in $\mathcal{T}_{i+1}$, and accordingly update the active index. Similarly, after a deletion, this can be i or $i-1$. For insertions which are not affordable, we store the edges in a `pending` list. Consider an insertion (u, v) which is not affordable in $\mathcal{T}_i$. This means that the loads on both u and v are at the limit $(\eta_i\alpha)$. We save (u, v) in the pending list. For $\mathcal{T}_i$ to become affordable, one of u's or v's load must decrease. At this point, we insert (u, v). The pseudocode for the overall algorithm is in Algorithm 3.

Notice, importantly, that insertions are made into $\mathcal{T}_i$ only when it is affordable. However, we always allow deletions because these are either deletions from the pending edges or from the graph currently stored in $\mathcal{T}_i$, which is still affordable.

Proof of Theorem 1.1. To show the correctness of Algorithm 3, we need to prove that at all times, $\hat{\rho}_G/2 \leq \rho^{\text{est}}_{\texttt{active}} < \hat{\rho}_G$. We know that this is true at the start of the algorithm. Assume this property is true at some instant before an update. When an edge is inserted, the first inequality might break. So, we test this after every addition and increment `active` accordingly. The argument follows similarly for deletions. However, we also need to make sure that when some $\mathcal{T}_i$ is queried, there are no edges remaining in $\texttt{pending}_i$, otherwise the queried density could possibly be incorrect. Consider an edge (u, v) inserted into $\texttt{pending}_i$ at some point during the algorithm. For $\mathcal{T}_i$ to be queried, it must be active, which means that at some point, the load of either u or v decreased, causing (u, v) to be inserted. Even when there are multiple such edges adjacent to the same high-load

- Initialize data structure $\mathcal{L} \leftarrow$ LazyDirectedLabels(G, η) with: $G = (V, \emptyset)$, $\alpha \leftarrow 64 \log n \cdot \epsilon^{-2}$, $\eta \leftarrow 2\rho^{\text{est}}/\alpha$

Operation `insert`$((u, v))$
  if $d(u) \geq d(v)$ **then**
    $\mathcal{L}$.add $\left(\overrightarrow{uv}\right)$
    $w \leftarrow v$
  else
    $\mathcal{L}$.add $\left(\overrightarrow{vu}\right)$
    $w \leftarrow u$
  while $\mathcal{L}$.tight_in_nbr$(w) \neq$ null **do**
    $w' \leftarrow \mathcal{L}$.tight_in_nbr$(w)$
    $\mathcal{L}$.flip $\left(\overrightarrow{w'w}\right)$
    $w \leftarrow w'$
  $\mathcal{L}$.increment(w)

Operation `query_subgraph()`
  $r \leftarrow \sqrt{2\eta \log n/\rho^{\text{est}}}$
  return $\mathcal{L}$.maximal_label_set(r);

Operation `delete`$((u, v))$
  if $u \in$ InNbrs$_v$ **then**
    $\mathcal{L}$.remove $\left(\overrightarrow{uv}\right)$
    $w \leftarrow v$
  else
    $\mathcal{L}$.remove $\left(\overrightarrow{vu}\right)$
    $w \leftarrow u$
  while $\mathcal{L}$.tight_out_nbr$(w) \neq$ null **do**
    $w' \leftarrow \mathcal{L}$.tight_out_nbr$(w)$
    $\mathcal{L}$.flip $\left(\overrightarrow{ww'}\right)$
    $w \leftarrow w'$
  $\mathcal{L}$.decrement(w)
  return(w)

Operation `query_density()`
  return $\mathcal{L}$.max_label $\times (1 - \epsilon)$;

Operation `query_load`(u)
  return $\mathcal{L}$.label(u);

Algorithm 2: Threshold$(G, \rho^{\text{est}}, \epsilon)$: Update routines on G when an estimate to its maximum load is known. Additionally V, $n = |V|$, and ϵ are known.

vertex, we are assured to see at least that many decrements at that vertex.

From Lemma 3.6, it follows that a query takes $O(1)$ worst-case time, and finding the subgraph takes $O(\beta + \log n)$ time, where β is the size of the output subgraph. Each insert or delete operation is first duplicated α times. Secondly, the updates are made individually in $\log_2 n$ copies of the data structure.

First, note that any insert or delete operation in `pending` can be processed in $O(\log n)$ time. This is also true for searching using a single end point of an edge owing to the manner in which `pending` is defined.

When an edge is added, it makes two load queries and then possibly inserts in $\mathcal{T}_i$. From Lemma 3.6, this gives a worst-case runtime of $O(\alpha^3 \log n)$ time per insertion.

As for deleting an edge, it sometimes also requires an insertion into $\mathcal{T}_i$. Again, plugging in runtimes from Lemma 3.6 gives a worst-case runtime of $O(\alpha^3 \log n)$ time per deletion. □

4 VERTEX-WEIGHTED DENSEST SUBGRAPH

In this section, we extend the ideas from Section 3 to extend to graphs with vertex weights. As we will see in Section 5, this extension is crucial in arriving at efficient dynamic algorithms for DSP on directed graphs.

Let us first formally define the concept of density in vertex-weighted graphs. Given a graph $G = \langle V, E, w \rangle$, where $w : V \mapsto \mathbb{Q}^{\geq 1}$, the density of a subgraph induced by a vertex subset $S \subseteq V$ is

$$\rho_G(S) \stackrel{\text{def}}{=} \frac{|E(S)|}{\sum_{v \in S} \omega(v)}.$$

For ease of notation we denote $\omega(S) \stackrel{\text{def}}{=} \sum_{v \in S} \omega(v)$. Constructing the approximate dual like in Sections 2 and 3, we get the same conditions except the *load* on a vertex v is now defined as

$$\ell_v = \frac{1}{\omega(v)} \sum_{e \ni v} f_e(v).$$

Let $\omega_{\min}$ and $\omega_{\max}$ denote the smallest and largest vertex weight in G. We multiply all the weights by $1/\omega_{\min}$ and later divide the answer by the same amount. This ensures that all weights are at least 1, and the maximum weight is now given by $W \stackrel{\text{def}}{=} \omega_{\max}/\omega_{\min}$.

We first show that local approximations also suffice for vertex-weighted DSP. We reuse the notation used in Section 3 for the exact and approximate dual LP – Dual(G) and Dual(G, η), but with vertex weights included.

Theorem 4.1. *Given an undirected vertex-weighted graph G with n vertices, with maximum vertex weight W, let $\hat{f}, \hat{\ell}$ denote any feasible solution to* Dual(G, η)*, and let $\hat{\rho}_G \stackrel{\text{def}}{=} \max_{v \in V} \hat{\ell}_v$. Then,*

$$\left(1 - 3\sqrt{\frac{\eta \log(nW)}{\hat{\rho}_G}}\right) \cdot \hat{\rho}_G \leq \rho^*_G \leq \hat{\rho}_G.$$

Proof. The proof follows the proof of Theorem 3.1 almost identically.

Any feasible solution of Dual(G, η) is also a feasible solution of Dual(G), and so we have $\rho^*_G \leq \hat{\rho}_G$.

Denote by T_i the set of vertices with load at least $\hat{\rho}_G - \eta i$, i.e., $T_i \stackrel{\text{def}}{=} \left\{v \in V \mid \hat{\ell}_v \geq \hat{\rho}_G - \eta i\right\}$. Let $0 < \alpha < 1$ be some adjustable parameter we will fix later. We define k to be the maximal integer such that for any $1 \leq i \leq k$, $\omega(T_i) \geq \omega(T_{i-1}) \cdot (1 + \alpha)$. Note that such a maximal integer k always exists because there are finite number of vertices in G and the size of T_i grows exponentially. By the maximality of k, $\omega(T_{k+1}) < \omega(T_k) \cdot (1 + \alpha)$. In order to bound

- **for** $i \leftarrow 1$ *to* $\log_2 n$ **do**
  - $\alpha \leftarrow 64 \log n \cdot \epsilon^{-2}$; $\rho_i^{\text{est}} \leftarrow 2^{i-2}\alpha$
  - Initialize $\mathcal{T}_i \leftarrow \textsc{Threshold}(G, \rho_i^{\text{est}}, \epsilon)$
  - Initialize a sorted list of edges $\texttt{pending}_i \leftarrow \emptyset$ using two balanced BSTs (one sorted using the first vertex of the edge, and another using the second)
  - Set $\texttt{active} \leftarrow 0$

Operation `query()`
- **return** $\mathcal{T}_{\texttt{active}}$.`query()`

Operation `query_subgraph()`
- **return** $\mathcal{T}_{\texttt{active}}$.`query_subgraph()`

Operation `insert`$((u,v))$
- **for** $k \leftarrow 1$ *to* α **do** `// duplicating` (u,v) α `times`
  - **for** $i \leftarrow \log_2 n$ *to* $\texttt{active}+1$ **do** $\mathcal{T}_i$.`insert`$((u,v))$ `// affordable copies`
  - $\rho \leftarrow \mathcal{T}_{\texttt{active}+1}$.`query()`
  - **if** $\rho \geq 2\rho^{est}_{\texttt{active}}$ **then** $\texttt{active} \leftarrow \texttt{active}+1$
  - **else** $\mathcal{T}_{\texttt{active}}$.`insert`$((u,v))$
  - **for** $i \leftarrow \texttt{active}-1$ *to* 1 **do** `// unaffordable copies`
    - $\ell_u \leftarrow \mathcal{T}_i$.`query_load`$(u)$; $\ell_v \leftarrow \mathcal{T}_i$.`query_load`$(v)$
    - **if** *both* $\ell_u, \ell_v \geq 2\rho^{est}$ **then** add (u,v) to $\texttt{pending}_i$
    - **else** $\mathcal{T}_i$.`insert`$((u,v))$ `// edge is still insertable`

Operation `delete`$((u,v))$
- **for** $k \leftarrow 1$ *to* α **do** `// duplicating` (u,v) α `times`
  - **for** $i \leftarrow \log_2 n$ *to* $\texttt{active}+1$ **do** $\mathcal{T}_i$.`delete`$((u,v))$ `// affordable copies`
  - $\rho \leftarrow \mathcal{T}_{\texttt{active}}$.`query()`
  - **if** $\rho < \rho^{est}$ **then**
    - $\texttt{active} \leftarrow \texttt{active}-1$
    - $\mathcal{T}_{\texttt{active}}$.`delete`$((u,v))$
  - **for** $i \leftarrow \texttt{active}-1$ *to* 1 **do** `// unaffordable copies`
    - **if** $(u,v) \in \texttt{pending}_i$ **then** remove one copy of (u,v) from $\texttt{pending}_i$
    - **else**
      - $w \leftarrow \mathcal{T}_i$.`delete`$((u,v))$ `// w's load was decremented`
      - **if** $(w,w') \in \texttt{pending}_i$ *for any* w' **then**
        - $\mathcal{T}_i$.`insert`$((w,w'))$
        - Remove (w,w') from $\texttt{pending}_i$

Algorithm 3: Main update algorithm. V, $n = |V|$, and ϵ are known quantities.

the density of this set T_{k+1}, we compute the total load on all vertices in T_k. For any $u \in T_k$, the load on u is given by

$$\hat{\ell}_u = \frac{1}{\omega(u)} \sum_{uv \in E} \hat{f}_{uv}(u)$$

However, we know that

$$f_{uv}(u) > 0 \implies \hat{\ell}_v \geq \hat{\ell}_u - \eta$$

and hence we only need to count for $v \in T_{k+1}$. Summing over all vertices in T_{k+1}, we get

$$\sum_{u \in T_k} \omega(u)\hat{\ell}_u = \sum_{u \in T_k, v \in T_{k+1}} \hat{f}_{uv}(u) \leq \sum_{u \in T_{k+1}, v \in T_{k+1}} \hat{f}_{uv}(u) = |E(T_{k+1})|.$$

Consider the density of set T_{k+1},

$$\rho(T_{k+1}) = \frac{|E(T_{k+1})|}{\omega(T_{k+1})} \geq \frac{\sum_{u \in T_k} \hat{\ell}_u}{\omega(T_{k+1})} \geq \frac{\omega(T_k) \cdot (\hat{\rho}_G - \eta k)}{\omega(T_{k+1})},$$

where the last inequality follows from the definition of T_k.

Since $\rho(T_{k+1})$ can be at most the maximum subgraph density ρ^*_G, and using the fact that $\omega(T_k)/\omega(T_{k+1}) > 1/(1+\alpha) \geq 1-\alpha$,

$$\rho^*_G \geq (1-\alpha)(\hat{\rho}_G - \eta k) \geq \hat{\rho}_G(1-\alpha)\left(1 - \frac{2\eta \log(nW)}{\alpha \cdot \hat{\rho}_G}\right),$$

where the last inequality comes from the fact that $nW \geq \omega(T_k) \geq (1+\alpha)^k$, which implies that $k \leq \log_{1+\alpha}(nW) \leq 2\log(nW)/\alpha$.

Now, we can set our parameter α to maximize the term on the RHS. By symmetry, the maximum is achieved when both terms in the product are equal and hence we set

$$\alpha = \sqrt{\frac{2\eta \log(nW)}{\hat{\rho}_G}}.$$

This gives

$$\rho_G^* \geq \hat{\rho}_G \cdot \left(1 - \sqrt{\frac{2\eta \log(nW)}{\hat{\rho}_G}}\right)^2$$
$$\geq \hat{\rho}_G \cdot \left(1 - 2\sqrt{\frac{2\eta \log(nW)}{\hat{\rho}_G}}\right)$$
$$\geq \hat{\rho}_G \cdot \left(1 - 3\sqrt{\frac{\eta \log(nW)}{\hat{\rho}_G}}\right). \qquad \square$$

Once again, scaling the graph up by a factor of $\alpha \stackrel{\text{def}}{=} \frac{64\log(nW)}{\epsilon^2}$, we can frame the question as the following graph orientation problem:

> Given an undirected graph G with vertex-weights $w : V \mapsto \mathbb{Q}^+$ and a slack parameter η, we want to assign directions to edges in such a way that for any edge $u \to v$,
> $$\frac{\boldsymbol{d}_{\text{in}}(v)}{\omega(v)} \leq \frac{\boldsymbol{d}_{\text{in}}(u)}{\omega(u)} + \eta.$$

To adapt the data structure from Algorithm 1, we only need to make the following change:

- `increment`(u) and `decrement`(u) no longer increment/decrement by 1 but by $1/\omega(u)$.
- Each entry in the Labels data structure is additionally appended with vertex weights - because instead of computing $|A|$ and $|B|$, we need to compute $\omega(A)$ and $\omega(B)$ in `maximal_label_set`.
- Since we assumed that $\omega(v) \geq 1$ for all $v \in V$, we do not have to adjust the conditions for tight edges.

Once we are provided with an estimate of $\hat{\rho}_G$, we can use the data structure from Algorithm 2 without any changes. Similar to Section 3.6, we now need to guess a value for $\hat{\rho}_G$. Notice that the range of values can now be $O(nW)$. Hence, using $O(\log(nW))$ values, we can apply Algorithm 3 to also solve the vertex-weighted version of DSP.

This gives us the following result.

Theorem 4.2. *Given a vertex-weighted graph G with n vertices, and vertex-weights in the range $\omega_{\min}$ and $\omega_{\max}$, there exists a deterministic fully dynamic $(1-\epsilon)$-approximation algorithm for the densest subgraph problem on G using $O(1)$ worst-case query time and worst-case update times of $O(\log^4(nW)\cdot\epsilon^{-6})$ per edge insertion or deletion.*

Moreover, at any point, the algorithm can output the corresponding approximate densest subgraph in time $O(\beta + \log n)$, where β is the number of vertices in the output.

5 DIRECTED DENSEST SUBGRAPH

The directed version of the densest subgraph problem was introduced by Kannan and Vinay [41]. In a directed graph $G = \langle V, E\rangle$, for a pair of sets $S, T \subseteq V$, we denote using $E(S,T)$ the set of directed edges going from a vertex in S to a vertex in T. The density of a pair of sets $S, T \subseteq V$ is defined as:

$$\rho_G(S,T) \stackrel{\text{def}}{=} \frac{|E(S,T)|}{\sqrt{|S||T|}}.$$

The maximum subgraph density of G is then defined as:

$$\rho_G^* \stackrel{\text{def}}{=} \max_{S,T\subseteq V} \rho_G(S,T).$$

Note that we use the same notation for density for undirected and directed graphs, as the distinction is clear from the graph in the subscript.

Charikar [20] reduced directed DSP to $O(n^2)$ instances of solving an LP, and also observed that only $O(\log n/\epsilon)$ suffice to extract a $(1-\epsilon)$ approximation. Khuller and Saha [43] used the same reduction, but further simplified the algorithm to $O(1)$ instances of a parametrized maximum flow problem.

In this section, we recount this reduction, but by visualizing the problem reduced to as a densest subgraph problem on vertex-weighted graphs, as defined in Section 4.

5.1 Reduction from Directed DSP to Vertex-weighted Undirected DSP

Given a directed graph $G = \langle V, E\rangle$ and a parameter $t > 0$, we construct a vertex-weighted undirected graph

$$G_t = \langle V_t, E_t, \omega_t\rangle$$

where,

- $V_t \stackrel{\text{def}}{=} V_t^{(L)} \cup V_t^{(R)}$, in which $V_t^{(L)}$ and $V_t^{(R)}$ are both clones of the original vertex set V;
- $E_t \stackrel{\text{def}}{=} \left\{(u,v) \mid u \in V_t^{(L)}, v \in V_t^{(R)}, (u,v) \in E\right\}$ projects each original directed edge $(u,v) \in E$ into an undirected edge between $V_t^{(L)}$ and $V_t^{(R)}$, and
- $\omega_t(u) \stackrel{\text{def}}{=} \begin{cases} 1/2t & u \in V_t^{(L)} \\ t/2 & u \in V_t^{(R)} \end{cases}$

To understand the intuition behind this reduction, consider a pair of sets $S, T \subseteq V$. Consider the set $S^{(L)}$ corresponding to S in $V_t^{(L)}$, and the set $T^{(R)}$ corresponding to T in $V_t^{(R)}$. $\rho_G(S,T) = \frac{|E(S,T)|}{\sqrt{|S||T|}}$, whereas $\rho_{G_t}(S^{(L)} \cup T^{(R)}) = \frac{2|E(S,T)|}{(1/t)|S| + t|T|}$. Picking t carefully lets us relate the two notions, leveraging the AM-GM inequality as indicated by the two denominators. Lemmas 5.1 and 5.2 show this relation in detail.

Lemma 5.1. *For any directed graph $G = \langle V, E\rangle$, let G_t be defined as above. Then for any choice of parameter t,*

$$\rho_G^* \geq \rho_{G_t}^*.$$

Proof. Let $S^{(L)} \cup T^{(R)}$ denote the densest (vertex-weighted) subgraph in G_t, where $S^{(L)} \in V_t^{(L)}$ and $T^{(R)} \in V_t^{(R)}$. Let S and T denote the corresponding vertex sets in V. Then we have

$$|E_t(S^{(L)} \cup T^{(R)})| = \rho_{G_t}^* \cdot (|S^{(L)}|/t + t|T^{(R)}|)/2$$
$$\geq \rho_{G_t}^* \sqrt{|S^{(L)}| \cdot |T^{(R)}|},$$

where the inequality follows from the AM-GM property. Using the facts $|E_t(S^{(L)} \cup T^{(R)})| = E(S,T)$, $|S^{(L)}| = |S|$, and $|T^{(R)}| = |T|$, we get that

$$\frac{E(S,T)}{\sqrt{|S| \cdot |T|}} \geq \rho^*_{G_t}.$$

Lastly, since the density of the pair of sets S, T in the directed graph G is at most ρ^*_G, we get that $\rho^*_G \geq \rho^*_{G_t}$. □

So, G_t provides a ready lower bound for computing maximum subgraph density, for any t. The next lemma shows that a careful choice of t can give equality between the two optimums.

Lemma 5.2. *For any directed graph $G = \langle V, E\rangle$ and a pair of subsets S, T that provides the maximum subset density, i.e., $\rho^*_G = \rho_G(S,T)$, we have*

$$\rho^*_G = \rho^*_{G_t},$$

where $t = \sqrt{\frac{|S|}{|T|}}$.

Proof. Now, consider the sets $S^{(L)} \in V_t^{(L)}$ and $T^{(R)} \in V_t^{(R)}$ corresponding to S and T respectively. The density of set $S \cup T$ can be at most $\rho^*_{G_t}$:

$$\rho^*_{G_t} \geq \frac{2|E(S,T)|}{|S|/t + |T| \cdot t}.$$

Substituting t with $|S|/|T|$,

$$\rho^*_{G_t} \geq \frac{2|E(S,T)|}{|S|\sqrt{\frac{|T|}{|S|}} + |T|\sqrt{\frac{|S|}{|T|}}} = \frac{|E(S,T)|}{\sqrt{|S| \cdot |T|}} = \hat{\rho}^*_G.$$

Combining this with the bound from Lemma 5.1 gives that $\rho^*_G = \rho^*_{G_t}$. □

Note, however, that this does not directly give an algorithm for directed densest subgraph, since we do not know the optimum value of $|S|/|T|$. Since both $|S|$ and $|T|$ are integers between 0 and n, there can be at most $O(n^2)$ distinct values of $|S|/|T|$. So, to find the exact solution, we can simply find $\rho^*_{G_t}$ for all possible t values, and report the maximum.

This connection was first observed by Charikar [20], where he reduced the directed densest subgraph problem to solving $O(n^2)$ linear programs. However, our construction helps view these LPs as DSP on vertex-weighted graphs, for which there are far more optimized algorithms than solving generic LPs, in both static and dynamic paradigms. Charikar [20] also observed that a $1+\epsilon$ approximate solution could be obtained by only checking $O(\log n/\epsilon)$ values of t. As one would expect, to obtain an approximate solution for the directed DSP, it is not necessary to obtain an exact solution to the undirected vertex-weighted DSP. As we show in Lemma 5.3, we only require $O(\log n/\epsilon)$ computations of a $1+\epsilon/2$ approximation to the densest subgraph problem.

Lemma 5.3. *For any directed graph $G = \langle V, E\rangle$ and a pair of subsets S, T that provides the maximum subset density, i.e., $\rho_G(S,T) = \rho^*_G$, we have*

$$\rho^*_{G_t} \geq (1-\epsilon)\rho^*_G,$$

where $\sqrt{\frac{|S|}{|T|}} \cdot (1-\epsilon) \leq t \leq \sqrt{\frac{|S|}{|T|}} \cdot \frac{1}{(1-\epsilon)}$.

Proof. Consider the vertices $S^{(L)} \in V_t^{(L)}$ and $T^{(R)} \in V_t^{(R)}$ corresponding to S and T respectively. The density of set $S^{(L)} \cup T^{(R)}$ can be at most $\rho^*_{G_t}$:

$$\rho^*_{G_t} \geq \frac{2|E(S,T)|}{|S|/t + |T| \cdot t}.$$

Substituting the bounds for t,

$$\rho^*_{G_t} \geq \frac{2(1-\epsilon)|E(S,T)|}{|S|\sqrt{\frac{|T|}{|S|}} + |T|\sqrt{\frac{|S|}{|T|}}} = (1-\epsilon)\rho_G(S,T) = (1-\epsilon)\rho^*_G.$$ □

5.2 Implications of the Reduction

The above reduction implies that finding a $(1-\epsilon)$-approximate solution to directed DSP can be reduced to $O(\log n/\epsilon)$ instances of $(1-\epsilon/2)$-approximate vertex-weighted undirected DSP.

Theorem 5.4. *Given a directed graph G, with m edges and n vertices, and a $T(m,n,\epsilon)$ time algorithm for $(1-\epsilon)$-approximate vertex-weighted undirected densest subgraph, then there exists an $(1-\epsilon)$-approximate algorithm for finding the densest subgraph in G in time $T(m,2n,\epsilon/2) \cdot O(\log n/\epsilon)$.*

Proof. For each value of t in

$$\left[\frac{1}{\sqrt{n}}, \frac{1}{(1-\epsilon/2)\sqrt{n}}, \frac{1}{(1-\epsilon/2)^2\sqrt{n}}, \ldots, \sqrt{n}\right],$$

we find an approximate value ρ such that $\rho \geq (1-\epsilon/2)\rho^*_{G_t}$, and output the maximum such value. Using $\epsilon/2$ as the error parameter in Lemma 5.3, we get that $\rho \geq (1-\epsilon)\rho^*_G$.

The number of values of t is $\log_{1/(1-\epsilon/2)} n = O(\log n/\epsilon)$. □

The current fastest algorithms for $(1-\epsilon)$-approximate static densest subgraph [9, 17] rely on approximately solving Dual(G), which is a positive linear program, and subsequently extracting a primal solution. Both these parts of the algorithm extend naturally to vertex-weighted graphs. Substituting these runtimes in for $T(m,n,\epsilon)$, we get the following corollary.

Corollary 5.5. *Let G be a directed graph with m edges and n vertices, and let Δ be the maximum value among all its in-degrees and out-degrees. Then, there exists an algorithm to find a $(1-\epsilon)$-approximate densest subgraph in G in time $\widetilde{O}(m\epsilon^{-2} \cdot \min(\Delta, \epsilon^{-1}))$.*

Here, $\widetilde{O}$ hides polylogarithmic factors in n.

The same reduction also applies to fully dynamic algorithm for directed DSP.

Theorem 5.6. *Suppose there exists a fully dynamic $(1-\epsilon)$-approximation algorithm for undirected vertex-weighted DSP on an n-vertex graph with update time $U(n,\epsilon)$ and query time $Q(n,\epsilon)$. Then, there exists a deterministic fully dynamic $(1-\epsilon)$-approximation algorithm for directed DSP on an n-vertex graph using $U(2n,\epsilon/2) \cdot O(\log n/\epsilon)$ query time and $Q(2n,\epsilon/2) \cdot O(\log n/\epsilon)$ query time.*

Substituting the runtimes from Theorem 4.2 in Section 4, we get our result for dynamic DSP on directed graphs.

Theorem 1.2. *Given a directed graph G with n vertices, there exists a deterministic fully dynamic $(1-\epsilon)$-approximation algorithm for the densest subgraph problem on G using $O(\log n/\epsilon)$ worst-case*

query time and worst-case update times of $O(\log^5 n \cdot \epsilon^{-7})$ per edge insertion or deletion.

Moreover, at any point, the algorithm can output the corresponding approximate densest subgraph in time $O(\beta + \log n)$, where β is the number of vertices in the output.

ACKNOWLEDGEMENTS

We thank Richard Peng and Gary Miller for their feedback and insightful discussions.

REFERENCES

[1] Amir Abboud and Søren Dahlgaard. 2016. Popular Conjectures as a Barrier for Dynamic Planar Graph Algorithms. In *IEEE 57th Annual Symposium on Foundations of Computer Science, FOCS 2016, 9-11 October 2016, Hyatt Regency, New Brunswick, New Jersey, USA*. 477–486. https://doi.org/10.1109/FOCS.2016.58

[2] Amir Abboud, Loukas Georgiadis, Giuseppe F. Italiano, Robert Krauthgamer, Nikos Parotsidis, Ohad Trabelsi, Przemyslaw Uznanski, and Daniel Wolleb-Graf. 2019. Faster Algorithms for All-Pairs Bounded Min-Cuts. In *46th International Colloquium on Automata, Languages, and Programming, ICALP 2019, July 9-12, 2019, Patras, Greece (LIPIcs)*, Christel Baier, Ioannis Chatzigiannakis, Paola Flocchini, and Stefano Leonardi (Eds.), Vol. 132. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 7:1–7:15. https://doi.org/10.4230/LIPIcs.ICALP.2019.7

[3] Amir Abboud, Robert Krauthgamer, and Ohad Trabelsi. 2020. New Algorithms and Lower Bounds for All-Pairs Max-Flow in Undirected Graphs. In *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms, SODA 2020, Salt Lake City, UT, USA, January 5-8, 2020*, Shuchi Chawla (Ed.). SIAM, 48–61. https://doi.org/10.1137/1.9781611975994.4

[4] Amir Abboud and Virginia Vassilevska Williams. 2014. Popular Conjectures Imply Strong Lower Bounds for Dynamic Problems. In *55th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2014, Philadelphia, PA, USA, October 18-21, 2014*. 434–443. https://doi.org/10.1109/FOCS.2014.53

[5] Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. 2013. Fast Exact Shortest-path Distance Queries on Large Networks by Pruned Landmark Labeling. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, New York, USA) *(SIGMOD '13)*. ACM, New York, NY, USA, 349–360. https://doi.org/10.1145/2463676.2465315

[6] Albert Angel, Nick Koudas, Nikos Sarkas, Divesh Srivastava, Michael Svendsen, and Srikanta Tirthapura. 2014. Dense subgraph maintenance under streaming edge weight updates for real-time story identification. *VLDB J.* 23, 2 (2014), 175–199. https://doi.org/10.1007/s00778-013-0340-z

[7] Sanjeev Arora, Elad Hazan, and Satyen Kale. 2012. The Multiplicative Weights Update Method: a Meta-Algorithm and Applications. *Theory of Computing* 8, 1 (2012), 121–164. https://doi.org/10.4086/toc.2012.v008a006

[8] Yuichi Asahiro, Kazuo Iwama, Hisao Tamaki, and Takeshi Tokuyama. 2000. Greedily finding a dense subgraph. *Journal of Algorithms* 34, 2 (2000), 203–221.

[9] Bahman Bahmani, Ashish Goel, and Kamesh Munagala. 2014. Efficient Primal-Dual Graph Algorithms for MapReduce. In *Algorithms and Models for the Web Graph - 11th International Workshop, WAW 2014, Beijing, China, December 17-18, 2014, Proceedings*. 59–78. https://doi.org/10.1007/978-3-319-13123-8_6

[10] Bahman Bahmani, Ravi Kumar, and Sergei Vassilvitskii. 2012. Densest Subgraph in Streaming and MapReduce. *Proc. VLDB Endow.* 5, 5 (Jan. 2012), 454–465. https://doi.org/10.14778/2140436.2140442

[11] Aaron Bernstein and Cliff Stein. 2016. Faster Fully Dynamic Matchings with Small Approximation Ratios. In *Proceedings of the Twenty-seventh Annual ACM-SIAM Symposium on Discrete Algorithms* (Arlington, Virginia) *(SODA '16)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 692–711. http://dl.acm.org/citation.cfm?id=2884435.2884485

[12] Sayan Bhattacharya, Monika Henzinger, and Giuseppe F. Italiano. 2018. Deterministic Fully Dynamic Data Structures for Vertex Cover and Matching. *SIAM J. Comput.* 47, 3 (2018), 859–887. https://doi.org/10.1137/140998925

[13] Sayan Bhattacharya, Monika Henzinger, and Giuseppe F. Italiano. 2018. Dynamic algorithms via the primal-dual method. *Inf. Comput.* 261, Part (2018), 219–239. https://doi.org/10.1016/j.ic.2018.02.005

[14] Sayan Bhattacharya, Monika Henzinger, and Danupon Nanongkai. 2016. New deterministic approximation algorithms for fully dynamic matching. In *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2016, Cambridge, MA, USA, June 18-21, 2016*. 398–411. https://doi.org/10.1145/2897518.2897568

[15] Sayan Bhattacharya, Monika Henzinger, and Danupon Nanongkai. 2017. Fully Dynamic Approximate Maximum Matching and Minimum Vertex Cover in $O(\log^3 n)$ Worst Case Update Time. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*. 470–489. https://doi.org/10.1137/1.9781611974782.30

[16] Sayan Bhattacharya, Monika Henzinger, Danupon Nanongkai, and Charalampos Tsourakakis. 2015. Space- and Time-Efficient Algorithm for Maintaining Dense Subgraphs on One-Pass Dynamic Streams. In *Proceedings of the Forty-seventh Annual ACM Symposium on Theory of Computing* (Portland, Oregon, USA) *(STOC '15)*. ACM, New York, NY, USA, 173–182. https://doi.org/10.1145/2746539.2746592

[17] Digvijay Boob, Saurabh Sawlani, and Di Wang. 2019. Faster width-dependent algorithm for mixed packing and covering LPs. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, 8-14 December 2019, Vancouver, BC, Canada*, Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (Eds.). 15253–15262. http://papers.nips.cc/paper/9663-faster-width-dependent-algorithm-for-mixed-packing-and-covering-lps

[18] Gerth Stølting Brodal and Rolf Fagerberg. 1999. Dynamic Representation of Sparse Graphs. In *Algorithms and Data Structures, 6th International Workshop, WADS '99, Vancouver, British Columbia, Canada, August 11-14, 1999, Proceedings*. 342–351. https://doi.org/10.1007/3-540-48447-7_34

[19] Mauro Brunato, Holger H. Hoos, and Roberto Battiti. 2008. On Effectively Finding Maximal Quasi-cliques in Graphs. In *Learning and Intelligent Optimization*, Vittorio Maniezzo, Roberto Battiti, and Jean-Paul Watson (Eds.). Springer-Verlag, Berlin, Heidelberg, 41–55. https://doi.org/10.1007/978-3-540-92695-5_4

[20] Moses Charikar. 2000. Greedy Approximation Algorithms for Finding Dense Components in a Graph. In *Proceedings of the Third International Workshop on Approximation Algorithms for Combinatorial Optimization (APPROX '00)*. Springer-Verlag, Berlin, Heidelberg, 84–95. http://dl.acm.org/citation.cfm?id=646688.702972

[21] J. Chen and Y. Saad. 2012. Dense Subgraph Extraction with Application to Community Detection. *IEEE Transactions on Knowledge and Data Engineering* 24, 7 (July 2012), 1216–1230. https://doi.org/10.1109/TKDE.2010.271

[22] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. 2003. Reachability and Distance Queries via 2-Hop Labels. *SIAM J. Comput.* 32, 5 (May 2003), 1338–1355. https://doi.org/10.1137/S0097539702403098

[23] Yon Dourisboure, Filippo Geraci, and Marco Pellegrini. 2007. Extraction and Classification of Dense Communities in the Web. In *Proceedings of the 16th International Conference on World Wide Web* (Banff, Alberta, Canada) *(WWW '07)*. ACM, New York, NY, USA, 461–470. https://doi.org/10.1145/1242572.1242635

[24] Ran Duan and Seth Pettie. 2014. Linear-Time Approximation for Maximum Weight Matching. *J. ACM* 61, 1 (2014), 1:1–1:23. https://doi.org/10.1145/2529989

[25] Alessandro Epasto, Silvio Lattanzi, and Mauro Sozio. 2015. Efficient Densest Subgraph Computation in Evolving Graphs. In *Proceedings of the 24th International Conference on World Wide Web* (Florence, Italy) *(WWW '15)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, Switzerland, 300–310. https://doi.org/10.1145/2736277.2741638

[26] D. R. Ford and D. R. Fulkerson. 2010. *Flows in Networks*. Princeton University Press, Princeton, NJ, USA.

[27] Greg N. Frederickson. 1985. Data Structures for On-Line Updating of Minimum Spanning Trees, with Applications. *SIAM J. Comput.* 14, 4 (1985), 781–798. https://doi.org/10.1137/0214055

[28] Harold N. Gabow and Robert Endre Tarjan. 1991. Faster Scaling Algorithms for General Graph-Matching Problems. *J. ACM* 38, 4 (1991), 815–853. https://doi.org/10.1145/115234.115366

[29] G. Gallo, M. D. Grigoriadis, and R. E. Tarjan. 1989. A Fast Parametric Maximum Flow Algorithm and Applications. *SIAM J. Comput.* 18, 1 (Feb. 1989), 30–55. https://doi.org/10.1137/0218003

[30] David Gibson, Ravi Kumar, and Andrew Tomkins. 2005. Discovering Large Dense Subgraphs in Massive Graphs. In *Proceedings of the 31st International Conference on Very Large Data Bases* (Trondheim, Norway) *(VLDB '05)*. VLDB Endowment, 721–732. http://dl.acm.org/citation.cfm?id=1083592.1083676

[31] A. V. Goldberg. 1984. *Finding a Maximum Density Subgraph*. Technical Report UCB/CSD-84-171. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/1984/5956.html

[32] Gramoz Goranci, Monika Henzinger, and Thatchaphol Saranurak. 2018. Fast Incremental Algorithms via Local Sparsifiers. (2018). unpublished manuscript.

[33] Manoj Gupta and Richard Peng. 2013. Fully Dynamic (1+ e)-Approximate Matchings. In *54th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2013, 26-29 October, 2013, Berkeley, CA, USA*. 548–557. https://doi.org/10.1109/FOCS.2013.65

[34] Monika Henzinger. 2018. The State of the Art in Dynamic Graph Algorithms. In *SOFSEM 2018: Theory and Practice of Computer Science*, A Min Tjoa, Ladjel Bellatreche, Stefan Biffl, Jan van Leeuwen, and Jiří Wiedermann (Eds.). Springer International Publishing, Cham, 40–44.

[35] 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* (Portland, Oregon, USA) *(STOC '15)*. ACM, New York, NY, USA, 21–30. https://doi.org/10.1145/2746539.2746609

[36] Monika Rauch Henzinger and Valerie King. 1999. Randomized Fully Dynamic Graph Algorithms with Polylogarithmic Time per Operation. *J. ACM* 46, 4 (1999),

502–516. https://doi.org/10.1145/320211.320215

[37] Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. 2001. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. ACM* 48, 4 (2001), 723–760. https://doi.org/10.1145/502090.502095

[38] Haiyan Hu, Xifeng Yan, Yu Huang, Jiawei Han, and Xianghong Jasmine Zhou. 2005. Mining Coherent Dense Subgraphs Across Massive Biological Networks for Functional Discovery. *Bioinformatics* 21, 1 (Jan. 2005), 213–221. https://doi.org/10.1093/bioinformatics/bti1049

[39] Giuseppe F. Italiano and Piotr Sankowski. 2010. Improved Minimum Cuts and Maximum Flows in Undirected Planar Graphs. *CoRR* abs/1011.2843 (2010). arXiv:1011.2843 http://arxiv.org/abs/1011.2843

[40] Ruoming Jin, Yang Xiang, Ning Ruan, and David Fuhry. 2009. 3HOPP: A High-compression Indexing Scheme for Reachability Query. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data* (Providence, Rhode Island, USA) *(SIGMOD '09)*. ACM, New York, NY, USA, 813–826. https://doi.org/10.1145/1559845.1559930

[41] Ravi Kannan and Vinay V. 1999. Analyzing the structure of large graphs. (1999). unpublished manuscript.

[42] Bruce M. Kapron, Valerie King, and Ben Mountjoy. 2013. Dynamic graph connectivity in polylogarithmic worst case time. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2013, New Orleans, Louisiana, USA, January 6-8, 2013*. 1131–1142. https://doi.org/10.1137/1.9781611973105.81

[43] Samir Khuller and Barna Saha. 2009. On Finding Dense Subgraphs. In *Proceedings of the 36th International Colloquium on Automata, Languages and Programming: Part I* (Rhodes, Greece) *(ICALP '09)*. Springer-Verlag, Berlin, Heidelberg, 597–608. https://doi.org/10.1007/978-3-642-02927-1_50

[44] Tsvi Kopelowitz, Robert Krauthgamer, Ely Porat, and Shay Solomon. 2014. Orienting Fully Dynamic Graphs with Worst-Case Time Bounds. In *Automata, Languages, and Programming - 41st International Colloquium, ICALP 2014, Copenhagen, Denmark, July 8-11, 2014, Proceedings, Part II*. 532–543. https://doi.org/10.1007/978-3-662-43951-7_45

[45] Lukasz Kowalik. 2007. Adjacency queries in dynamic sparse graphs. *Inf. Process. Lett.* 102, 5 (2007), 191–195. https://doi.org/10.1016/j.ipl.2006.12.006

[46] Ravi Kumar, Jasmine Novak, and Andrew Tomkins. 2006. Structure and Evolution of Online Social Networks. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Philadelphia, PA, USA) *(KDD '06)*. ACM, New York, NY, USA, 611–617. https://doi.org/10.1145/1150402.1150476

[47] Ravi Kumar, Prabhakar Raghavan, Sridhar Rajagopalan, Sridhar Rajagopalan, Andrew Tomkins, Andrew Tomkins, and Andrew Tomkins. 1999. Trawling the Web for Emerging Cyber-communities. *Comput. Netw.* 31, 11-16 (May 1999), 1481–1493. https://doi.org/10.1016/S1389-1286(99)00040-7

[48] Victor E. Lee, Ning Ruan, Ruoming Jin, and Charu Aggarwal. 2010. *A Survey of Algorithms for Dense Subgraph Discovery*. Springer US, Boston, MA, 303–336. https://doi.org/10.1007/978-1-4419-6045-0_10

[49] Aleksander Madry. 2011. *From graphs to matrices, and back: new techniques for graph algorithms*. Ph.D. Dissertation. Massachusetts Institute of Technology, Cambridge, MA, USA. http://hdl.handle.net/1721.1/66014

[50] Andrew McGregor, David Tench, Sofya Vorotnikova, and Hoa T. Vu. 2015. Densest Subgraph in Dynamic Graph Streams. In *Mathematical Foundations of Computer Science 2015 - 40th International Symposium, MFCS 2015, Milan, Italy, August 24-28, 2015, Proceedings, Part II*. 472–482. https://doi.org/10.1007/978-3-662-48054-0_39

[51] 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*. 17–27. https://doi.org/10.1109/SFCS.1980.12

[52] Nina Mishra, Robert Schreiber, Isabelle Stanton, and Robert Endre Tarjan. 2008. Finding Strongly Knit Clusters in Social Networks. *Internet Mathematics* 5, 1 (2008), 155–174. https://doi.org/10.1080/15427951.2008.10129299

[53] Michael Mitzenmacher, Jakub Pachocki, Richard Peng, Charalampos Tsourakakis, and Shen Chen Xu. 2015. Scalable Large Near-Clique Detection in Large-Scale Networks via Sampling. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Sydney, NSW, Australia) *(KDD '15)*. ACM, New York, NY, USA, 815–824. https://doi.org/10.1145/2783258.2783385

[54] M. E. J. Newman. 2006. Modularity and community structure in networks. *Proceedings of the National Academy of Sciences* 103, 23 (2006), 8577–8582. https://doi.org/10.1073/pnas.0601602103 arXiv:https://www.pnas.org/content/103/23/8577.full.pdf

[55] Serge A Plotkin, David B Shmoys, and Éva Tardos. 1995. Fast approximation algorithms for fractional packing and covering problems. *Mathematics of Operations Research* 20, 2 (1995), 257–301.

[56] Jun Ren, Jianxin Wang, Min Li, and Lusheng Wang. 2013. Identifying protein complexes based on density and modularity in protein-protein interaction network. *BMC Systems Biology* 7, 4 (23 Oct 2013), S12. https://doi.org/10.1186/1752-0509-7-S4-S12

[57] Barna Saha, Allison Hoch, Samir Khuller, Louiqa Raschid, and Xiao-Ning Zhang. 2010. Dense Subgraphs with Restrictions and Applications to Gene Annotation Graphs. In *Research in Computational Molecular Biology*, Bonnie Berger (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 456–472.

[58] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. 2017. The Ubiquity of Large Graphs and Surprising Challenges of Graph Processing. *Proc. VLDB Endow.* 11, 4 (Dec. 2017), 420–431. https://doi.org/10.1145/3186728.3164139

[59] Atish Das Sarma, Ashwin Lall, Danupon Nanongkai, and Amitabh Trehan. 2012. Dense Subgraphs on Dynamic Networks. In *Distributed Computing - 26th International Symposium, DISC 2012, Salvador, Brazil, October 16-18, 2012. Proceedings*. 151–165. https://doi.org/10.1007/978-3-642-33651-5_11

[60] Stephen B. Seidman. 1983. Network structure and minimum degree. *Social Networks* 5, 3 (1983), 269 – 287. https://doi.org/10.1016/0378-8733(83)90028-X

[61] Daniel D. Sleator and Robert Endre Tarjan. 1983. A Data Structure for Dynamic Trees. *J. Comput. Syst. Sci.* 26, 3 (June 1983), 362–391. https://doi.org/10.1016/0022-0000(83)90006-5

[62] Hsin-Hao Su and Hoa T. Vu. 2019. Distributed Dense Subgraph Detection and Low Outdegree Orientation. *CoRR* abs/1907.12443 (2019).

[63] Lei Tang and Huan Liu. 2010. *Graph Mining Applications to Social Network Analysis*. Springer US, Boston, MA, 487–513. https://doi.org/10.1007/978-1-4419-6045-0_16

[64] Mikkel Thorup. 2007. Fully-Dynamic Min-Cut. *Combinatorica* 27, 1 (2007), 91–127. https://doi.org/10.1007/s00493-007-0045-2

[65] Charalampos E. Tsourakakis. 2014. A Novel Approach to Finding Near-Cliques: The Triangle-Densest Subgraph Problem. *CoRR* abs/1405.1477 (2014). arXiv:1405.1477 http://arxiv.org/abs/1405.1477