Maintaining a large matching and a small vertex cover

Maintaining a Large Matching and a Small Vertex Cover

Krzysztof Onak*
MIT
konak@mit.edu

Ronitt Rubinfeld†
MIT and Tel Aviv University
ronitt@csail.mit.edu

ABSTRACT

We consider the problem of maintaining a large matching and a small vertex cover in a dynamically changing graph. Each update to the graph is either an edge deletion or an edge insertion. We give the first randomized data structure that simultaneously achieves a constant approximation factor and handles a sequence of K updates in $K \cdot \text{polylog}(n)$ time, where n is the number of vertices in the graph. Previous data structures require a polynomial amount of computation per update.

Categories and Subject Descriptors

E.1 [**Data Structures**]: Graphs and Networks; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems; G.2.2 [**Graph Theory**]: Graph Algorithms

General Terms

Algorithms

Keywords

dynamic algorithms, data structures, maximum matching, vertex cover

1. INTRODUCTION

Suppose one is given the task of solving a combinatorial problem, such as vertex cover or maximum matching, for a very large and constantly changing graph. In this setting, it is natural to ask, does one need to recompute the solution from scratch after every update?

*Supported by NSF grants 0732334 and 0728645.

†Supported by NSF grants 0732334 and 0728645, Marie Curie Reintegration grant PIRG03-GA-2008-231077, and the Israel Science Foundation grant nos. 1147/09 and 1675/09.

STOC'10, June 5–8, 2010, Cambridge, Massachusetts, USA.

Such questions have been asked before for various combinatorial quantities—examples include minimum spanning tree, shortest path length, min-cut, and many others (some examples include [3, 2, 6, 15, 8, 14]). Classic works for these problems have shown update times that are sublinear in the input size. For the problem of maximum matching, Sankowski [13] shows that it can be maintained with $O(n^{1.495})$ computation per update (n is the number of vertices in the graph), which for dense graphs is sublinear in the number of edges.

For very large graphs, it may be crucial to maintain the maximum matching with much faster, even polylogarithmic, update time. Note that this may be hard for maximum matching, since obtaining $o(\sqrt{n})$ update time, even in the case when only insertions are allowed, would improve on the 30-year-old algorithm of running time $O(m\sqrt{n})$ due to Micali and Vazirani [11], where m is the number of edges in the graph. Therefore, some kind of approximation may be unavoidable. Following similar considerations, Ivković and Lloyd [7] give a factor-2 approximation to both vertex cover and maximum matching, by maintaining a *maximal* matching (which is well known to give the desired approximation for maximum matching and minimum vertex cover). Their update time is nevertheless still polynomial in n. More precisely, it is $O((n+m)^{0.7072})$, which is $o(n)$ for sparse graphs.

In this paper, we concentrate on the setting in which slightly weaker, but still $O(1)$, approximation factors are acceptable, and in which it is crucial that update times be extremely fast, in particular, polylogarithmic in the graph size.

1.1 Problem Statement and Our Results

Recall that in the *maximum matching* problem, one wants to find the largest subset of vertex disjoint edges. In the *vertex cover* problem, one wants to find the smallest set of vertices such that each edge of the graph is incident to at least one vertex in the set.

Our goal here is to design a data structure that handles edge removals and edge insertions. The data structure provides access to a list of edges that constitute a large matching or a list of vertices that constitute a small vertex cover. We assume that we always start with an empty graph, and n is known in advance.

The main result of the paper is the following:

> *There is a randomized data structure for maximum matching and vertex cover that*
>
> *(a) achieves a constant approximation factor,*

(b) runs in

$$O\left(\min\{K, n^2\} \cdot \log n \cdot \log \frac{1}{\delta} + K \cdot \log^2 n\right)$$

time for a fixed sequence of K updates with probability $1 - \delta$.

Furthermore, the first step in our presentation is a *deterministic* data structure for vertex cover. The data structure keeps a vertex cover that gives an $O(\log n)$ approximation to the minimum vertex cover. The amortized update time of the data structure is $O(\log^2 n)$. Though the approximation factor achieved by this algorithm is relatively weak, the algorithm may be of independent interest because of its relative simplicity and efficient update time.

1.2 Overview of Our Techniques

We present our main result in two stages.

A Deterministic $O(\log n)$-Approximation Data Structure: We construct a data structure that makes use of a carefully designed partition of vertices into a logarithmic number of subsets. The partition is inspired by a simple distributed algorithm of Parnas and Ron [12]. In [12], the first subset in the partition corresponds to removing vertices of degree approximately n. The second subset corresponds to removing vertices of degree close to $n/4$ from the modified graph. In general, the i-th subset is a set of vertices that are approximately $n/4^{i-1}$ in the graph with all previous subsets of vertices removed. Finally, after a logarithmic number of steps, the remaining graph has no edges. This implies that the union of all subsets removed so far constitutes a vertex cover. For each of the removed subsets, it is easy to show that the subset size is bounded by $O(\mathrm{VC}(G))$, where $\mathrm{VC}(G)$ is the size of the minimum vertex cover. Hence the total vertex cover is bounded by $O(\mathrm{VC}(G) \cdot \log n)$.

The main idea behind our data structure is to modify the partition of Parnas and Ron in order to allow efficient maintenance of this partition, under edge insertions and deletions. While this is not possible in the partition of Parnas and Ron, it is possible in our relaxed version of it. As edges are inserted and removed, we want to move vertices between subsets. In order to determine whether to move a vertex, we associate a potential function with every vertex, and we allow a vertex to jump from one set to another only if it has collected enough potential. To do this, we set two thresholds $\tau_1 < \tau_2$ for each subset. A vertex can move into the subset from a subset corresponding to a lower degree if its number of neighbors in a specific graph is at least τ_2. Then the vertex can move back to a subset corresponding to a lower degree only if the number of edges decreases to τ_1 in the same graph. A slight technical difficulty is presented by the fact that moving vertices may increase the potential of other vertices. We overcome this obstacle by carefully selecting constants in the potential function so that the potential of the vertex that moves is spent on increasing the potential of its neighbors.

A Randomized $O(1)$-Approximation Data Structure: In this case, we redesign the partition, building upon the previous one. In the process of defining the partition, whenever we remove a large subset W of vertices of degree approximately $n/4^i$, we also show the existence of a matching M which is smaller than W by at most a constant factor. To build the next set of the partition, we not only remove W but also all vertices matched in M. In this way we achieve a matching and a vertex cover of sizes that are within a constant factor of each other. Therefore, both give a constant factor approximation to their respective optimal solutions.

Efficient maintenance of the new partition is more involved, as we are sometimes forced to recompute a matching. This can happen, for instance, when many edges in the matching are deleted from the graph. Unfortunately, the creation of a new matching is expensive, since we have modified the set of the vertices matched in M that are deleted together with W. If the edges in the matching are deleted too quickly, we have to create a new matching often, in which case we do not know how to maintain small update time. Fortunately, by picking a large *random* matching, we can ensure that it is unlikely that many edges from the matching get deleted in a short span of time. Thus, by the time the matching gets deleted, we are likely to have collected enough potential to pay for the creation of a new matching.

1.3 Other Related Work

A sequence of papers [5, 10, 16, 4] considers computing a large matching or a large weight matching (in the weighted case) in the semi-streaming model. The stream is a sequence of edges, and the goal of an algorithm is to compute a large matching in a small number of passes over the stream, using $O(n \cdot \log^{O(1)} n)$ space, and preferably at most polylog(n) update time. Results in this model correspond to results for dynamically changing graphs in which only edge insertions occur, except that the matching is only output once, at the end of the processing. To the best of our knowledge, it is not known how to achieve a better approximation factor than 2 in one pass for the maximum matching problem.

Lotker, Patt-Shamir, and Rosén [9] show how to maintain a large matching in a distributed network.

2. PRELIMINARIES

We assume that all the necessary simple set operations (insert, remove, find, ...) on ordered sets of size t can be implemented in $O(\log t)$ time. A number of tree based dictionaries (AVL trees, red-black trees, etc.) have this property (see for instance the textbook of Cormen *et al.* [1]). We also assume that the first s items in a set can be accessed in $O(s)$ time, which can usually easily be achieved by augmenting a given data structure with additional links.

Throughout the paper, α is a fixed integer greater than 1. We write k_α to denote $\lfloor \log_\alpha n \rfloor + 2$. Moreover, $\mathrm{VC}(G)$ is the minimum vertex cover size in G, and $\mathrm{MM}(G)$ is the maximum matching size in G.

2.1 Basic Facts

FACT 1. *Let G be a graph, and let M be a matching in G. Then,* $\mathrm{VC}(G) \geq |M|$.

LEMMA 2. *Let G be a graph of maximum degree d. Let V' be a subset of vertices such that every vertex in V' has degree between d/γ and d. The following holds:*

- *There is a matching M of size at least $|V'|/(4\gamma)$ with each edge incident to a vertex in V'.*
- $|V'| \leq 4\gamma \cdot \mathrm{VC}(G)$.

PROOF. There are at least $X \stackrel{\text{def}}{=} \frac{d|V'|}{2\gamma}$ edges incident to vertices in V'. Each such edge is adjacent to at most $Y \stackrel{\text{def}}{=} 2(d-1)$ other such edges. This implies that G has a matching of size at least $X/(Y+1) \geq |V'|/(4\gamma)$. By Fact 1, $|V'| \leq 4\gamma \cdot \mathrm{VC}(G)$. □

2.2 Simple $O(d)$-Update-Time Data Structure

We now describe a straightforward data structure for maintaining a maximal matching in a graph of maximum degree bounded by d. To the best of our knowledge, the data structure was first described by Ivković and Lloyd [7].

For every vertex, the data structure maintains information indicating whether it is matched. Whenever an edge is inserted, the data structure checks if its endpoints are matched or not. If none of them are, the edge is added to the matching. Whenever an edge e in the maximal matching is removed, the data structure checks whether the remaining matching may be extended by adding edges incident to the endpoints of e. To do this, the data structure goes over $O(d)$ edges that were adjacent to e, and greedily tries to extend the matching with each of them.

It is easy to show that the matching held by the data structure is maximal. It can be used for obtaining a 2-approximation of both the minimum vertex cover (use the endpoints of edges in the matching) and the maximum matching (use the maximal matching itself). The `Insert` operation takes $O(1)$ time, and the `Delete` operation requires $O(d)$ time.

3. WARMUP: DETERMINISTIC $O(\log n)$-APPROXIMATION FOR VERTEX COVER

3.1 A Sequential Algorithm

Consider first the sequential Algorithm 1. The algorithm is a modification of a simple distributed algorithm for vertex cover that was used by Parnas and Ron [12].

Algorithm 1: A sequential $O(\log n)$-approximation algorithm for vertex cover

Input: graph G, integer $\alpha > 1$

$k_\alpha := \lfloor \log_\alpha n \rfloor + 2$
$G_{k_\alpha} := G$
for $i := k_\alpha$ **downto** 1 **do**
  $V_i := \{\text{vertices of degree} \geq \alpha^{i-1} \text{ in } G_i\}$
    $\cup \{\text{arbitrary subset of vertices of degree}$
      $\text{in } [\alpha^{i-2}, \alpha^{i-1}) \text{ in } G_i\}$
  $G_{i-1} := G_i$ with vertices in V_i removed
return $\bigcup_{i=1}^{k_\alpha} V_i$

LEMMA 3. *Let α be an integer greater than 1. The size of each set V_i in Algorithm 1 is bounded by $4\alpha^2 \,\mathrm{VC}(G)$. Algorithm 1 computes a vertex cover of size $\leq 4\alpha^2 \cdot k_\alpha \cdot \mathrm{VC}(G)$.*

PROOF. The algorithm repeatedly removes vertices and their adjacent edges from the original graph, and adds the removed vertices to the cover. To see that the algorithm computes a vertex cover, note that the final graph G_0 has no edges, which means that all edges of G have been covered by the output of the algorithm.

Let i be any integer between 1 and k_α. The maximum degree of G_i is bounded by α^i. By Lemma 2, $|V_i| \leq 4\alpha^2 \cdot \mathrm{VC}(G_i) \leq 4\alpha^2 \cdot \mathrm{VC}(G)$. This implies that the size of the cover returned by the algorithm is at most $k_\alpha \cdot 4\alpha^2 \,\mathrm{VC}(G)$. □

3.2 The Data Structure

We design a data structure that keeps a partition of vertices into a logarithmic number of sets V_i, $0 \leq i \leq k_\alpha$. The partition is one that could potentially be created in an execution of Algorithm 1. We refer to sets V_i as *buckets*. The sets V_i with $i > 0$ are sets of vertices removed in consecutive executions of the loop of Algorithm 1, and V_0 is the set of vertices that are not removed from the graph by the algorithm. For $i > 0$, each V_i consists of vertices that at the time of removal, have degree between α^{i-2} and α^i. The union of V_i over $i > 0$ is the current vertex cover.

For every vertex v, we maintain the following variables:

`index`$[v]$**:** the index of the set V_i that contains v.

`neighbors`$[v, j]$ **for** $j \geq$ `index`$[v]$**:** the set of neighbors of v that that belong to V_j.

`below`$[v, j]$ **for** $j \geq$ `index`$[v]$**:** the total number of all neighbors of v in sets V_0 through V_j.

`lower-neighbors`$[v]$**:** the set of neighbors of v that belong to V_i for $i <$ `index`$[v]$.

We call the collection of vertices involving v the *structures of* v.

Initially, the graph is empty, so all sets of neighbors are empty, and `index`$[v] = 0$ and `below`$[v, j] = 0$ for all v and j.

We maintain the following invariants for each vertex v after each update to the graph:

- To ensure that v's bucket number `index`$[v]$ is not too high, i.e., that it has enough edges to nodes in lower buckets, we ask that if `index`$[v] > 0$, then `below`$[v,$ `index`$[v]] > \alpha^{\texttt{index}[v]-2}$.
- On the other hand, to ensure that v's bucket number is not too low, i.e., that it should not have been placed in a higher bucket, we ask that for each $i \in \{$`index`$[v] + 1, \ldots, k_\alpha\}$, `below`$[v, i] < \alpha^{i-1}$.

Note that if this is the case, then the sets V_i, $1 \leq i \leq k_\alpha$ defined as $V_i = \{v \in V :$ `index`$[v] = i\}$ could potentially be created by the non-deterministic Algorithm 1.

As a result of edge removals and insertions, the invariants may no longer hold. We first design a procedure `Restabilize` that given a set of vertices for which the invariant may not hold (we call such vertices *dirty*), attempts to fix the partition given by `index`$[\cdot]$. As long as there is a dirty vertex v, the procedure does the following.

- If there is an $i >$ `index`$[v]$, such that `below`$[v, i] \geq \alpha^{i-1}$, the procedure sets `index`$[v]$ to the highest such i. (This could happen if many edges adjacent to v have been added to the graph, or if many edges have been deleted from v's neighbors that were previously in higher buckets than the i-th, causing them to be demoted to lower buckets.)

  Let t and $t_\star$ be the new and old value of `index`$[v]$, respectively. The move of v from $V_{t_\star}$ to V_t may invalidate the invariant for neighbors of v in buckets V_t to

$V_{t_\star-1}$. Therefore, the procedure marks all of them as dirty.

Next, the procedure updates `neighbors`$[u]$, `below`$[u,\cdot]$, and `lower-neighbors`$[u]$ for all neighbors u of v in buckets V_0 through V_t. Then the procedure updates the structures for v as well. Note that updating all the structures takes at most $O(\texttt{below}[v,t]\cdot\log n)$ time, because this requires at most a constant number of set operations per each of the neighbors in consideration, and for each vertex u, the array `below`$[u,\cdot]$ can be updated in $O(k_\alpha)=O(\log n)$ time.

Finally, the procedure marks v as no longer dirty.

- Otherwise, if $\texttt{index}[v]>0$, and $\texttt{below}[v,\texttt{index}[v]]\le\alpha^{\texttt{index}[v]-2}$, the procedure decreases `index`$[v]$ by one. (This could happen if many edges adjacent to v have been deleted, or if many edges are added to v's neighbors that were previously in lower buckets than v, causing them to jump to higher buckets.)

  Let t be the new value of `index`$[v]$. The move of v can affect the invariant for neighbors of v in sets V_0 through V_{t-1}, so the procedure marks all of them as dirty.

  The procedure also updates `neighbors`$[u]$, `below`$[u,\cdot]$, and `lower-neighbors`$[u]$ for all neighbors u of v in buckets V_0 to V_{t+1}. Next it does the same for the structures of v. In total, this takes $O(\texttt{below}[v,t+1]\cdot\log n)$ time, since at most a constant number of set operations per each of the neighbors in consideration is necessary, and for each of them `below`$[\cdot,\cdot]$ can be updated in $O(k_\alpha)=O(\log n)$ time.

  In this case, the procedure does not change the status of v. It still remains dirty, since the procedure may have to decrease `index`$[v]$ further[1].

- If none of the previous cases occurred, v is already in the right bucket, and there is no need to move it. The procedure marks the vertex as no longer dirty.

It is not immediately clear that the above procedure `Restabilize` always stops. We show that this is the case in Section 3.3.

It is easy to implement operations `Insert` and `Delete` that are responsible for inserting and removing an edge by using `Restabilize`. It suffices to modify first the corresponding `below`$[u,\cdot]$, `neighbors`$[u,\cdot]$, and `lower-neighbors`$[u]$ for each of the edge's endpoints u (this can be done in $O(\log n)$ time), mark the endpoints as dirty, and run `Restabilize` to fix the partition of vertices if necessary.

3.3 Complexity Analysis

THEOREM 4. *The amortized complexity of the operations* `Insert` *and* `Delete` *in the deterministic data structure is* $O(\log^2 n)$ *for* $\alpha=4$.

PROOF. The use the following potential function. The potential of a vertex v equals

$$\Phi(v)\stackrel{\text{def}}{=}\Phi_1(v)+\Phi_2(v),$$

where

$$\Phi_1(v)\stackrel{\text{def}}{=}0,$$

if $\texttt{index}[v]=0$,

$$\Phi_1(v)\stackrel{\text{def}}{=}8\cdot\min\Big\{\max\left\{\alpha^{\texttt{index}[v]-1}-\texttt{below}[v,\texttt{index}[v]],0\right\},\ \alpha^{\texttt{index}[v]-1}-\alpha^{\texttt{index}[v]-2}\Big\}$$

for $\texttt{index}[v]>0$, and

$$\Phi_2(v)\stackrel{\text{def}}{=}12\cdot\sum_{i=\texttt{index}[v]+1}^{k_\alpha}\max\left\{\texttt{below}[v,i]-\alpha^{i-2},0\right\}.$$

$\Phi_1(v)$ corresponds to losing neighbors. When v loses sufficiently many of them, there is enough potential to pay for decreasing `index`$[v]$. $\Phi_2(v)$ is related to the number of neighbors u with $\texttt{index}[u]>\texttt{index}[v]$. We only increase `index`$[v]$ if there are sufficiently many of them, and then $\Phi_2(v)$ provides enough potential to conduct the operation. For the initial empty graph, all $\Phi(v)=0$.

Each unit of the potential corresponds to $O(\log n)$ computation. Inserting or removing an edge can only change the potential of the endpoints of the edge, and the change is bounded by $O(k_\alpha)$, because each of $O(k_\alpha)$ terms can only change by a constant. We show that fixing the invariant of the data structure, i.e., executing the procedure `Restabilize`, is almost entirely paid for by potentials of vertices. More precisely, we show that the amortized complexity of `Restabilize` is the number of vertices that are initially marked as dirty times $O(\log n)$. Assuming this, the amortized cost of both `Insert` and `Delete` is $O(k_\alpha\cdot\log n)=O(\log^2 n)$.

Consider the case when `Restabilize` increases `index`$[v]$ for a vertex v. If this happens, `index`$[v]$ becomes t such that $\texttt{below}[v,t]\ge\alpha^{t-1}$. We can use up to $12\cdot\big(\texttt{below}[v,t]-\alpha^{t-2}\big)$ units of the potential of v. This comes from the decrease in $\Phi(v)$, and more precisely in $\Phi_2(v)$. Once `index`$[v]$ is set to t, $\Phi_1(v)$ becomes 0, and $\Phi_2(v)$ only equals

$$\sum_{i=t+1}^{k_\alpha}\max\left\{\texttt{below}[v,i]-\alpha^{i-2},0\right\}.$$

`Restabilize` updates structures for all neighbors u of v such that $\texttt{index}[u]\le t$. It also marks some of them as dirty, and the potential has to pay also for checking later whether the invariant holds for them. This costs at most $O(\log n)\cdot\texttt{below}[v,t]$, that is, $\texttt{below}[v,t]$ units of potential. Additionally, modifying `index`$[v]$ can result in decreasing $\texttt{below}[u,\texttt{index}[u]]$ for some of the same neighbors u of v. The potential, therefore, has to pay another $8\cdot\texttt{below}[v,t]$ units to compensate for the change in $\Phi_1(u)$ for those neigh-

[1]One could immediately decrease `index`$[v]$ to the right value, but it is easier to analyze the complexity of this version of the procedure.

bors. The total expense can be bounded by

$$\begin{aligned}
&\texttt{below}[v,t] + 8\cdot\texttt{below}[v,t] \\
&\quad= 9\cdot\frac{\texttt{below}[v,t]}{\texttt{below}[v,t]-\alpha^{t-2}}\cdot\left(\texttt{below}[v,t]-\alpha^{t-2}\right)\\
&\quad\le 9\cdot\frac{\alpha^{t-1}}{\alpha^{t-1}-\alpha^{t-2}}\cdot\left(\texttt{below}[v,t]-\alpha^{t-2}\right)\\
&\quad= \frac{9\alpha}{\alpha-1}\cdot\left(\texttt{below}[v,t]-\alpha^{t-2}\right)\\
&\quad= 12\cdot\left(\texttt{below}[v,t]-\alpha^{t-2}\right),
\end{aligned}$$

which is not more than the available budget.

Consider now the other case when `Restabilize` decreases $\texttt{index}[v]$ by one for a vertex v. Let t be the new $\texttt{index}[v]$, i.e., the old $\texttt{index}[v]$ minus one. Note that the old $\Phi_1(v)$ equals $8\cdot(\alpha^t-\alpha^{t-1})$. `Restabilize` updates structures for at most α^{t-1} neighbors u of v, and it also marks some of them as dirty. This costs at most α^{t-1} units of potential together with checking later whether the invariant holds for them. Moving v may also increase $\texttt{below}[u,t]$ by one for all of them, and this may increase their potential Φ_2. The total increase is at most $12\cdot\alpha^{t-1}$ units of potential. Furthermore, the new potential $\Phi_1(v)$ can still be positive, but it can be bounded by $8\cdot(\alpha^{t-1}-\alpha^{t-2})$. Finally, we pay 1 for reverifying if the invariant holds for v after the modification of $\texttt{index}[v]$. The total expense is bounded by

$$\begin{aligned}
&\alpha^{t-1}+12\cdot\alpha^{t-1}+8\cdot(\alpha^{t-1}-\alpha^{t-2})+1\\
&\quad\le\left(\frac{13}{\alpha-1}+\frac{8}{\alpha}+\frac{1}{\alpha^t-\alpha^{t-1}}\right)\cdot(\alpha^t-\alpha^{t-1})\\
&\quad\le\left(\frac{13}{3}+2+\frac{4}{3}\right)\cdot(\alpha^t-\alpha^{t-1})\le 8\cdot(\alpha^t-\alpha^{t-1}),
\end{aligned}$$

which is the old $\Phi_1(v)$. □

4. RANDOMIZED $O(1)$-APPROXIMATION

We now describe a new data structure that maintains an $O(1)$ approximation for maximum matching and vertex cover. The data structure handles a sequence of ℓ updates in $\ell\cdot\text{polylog}(n)$ time. In Section 4.1, we describe a new method of creating a partition of the vertices along with the properties that the partition satisifies. In Section 4.2, we describe how to generate a matching and vertex cover from the partition and bound the approximation factor. In Section 4.4, we give the implementation details which allow one to maintain the partition along with the required properties through the successive updates. In Section 4.5, we describe the amortized analysis of the update time.

In the following description, we use sufficiently large positive constants C_1, C_2, and C_{T}. We require that $C_2 \ll C_1$ and $C_2 \ll C_{\mathrm{T}}$. As before, we assume that the α we use equals 4. Recall also that $k_\alpha = \lfloor\log_\alpha n + 2\rfloor$, that is, $k_\alpha = O(\log n)$.

4.1 The new partition and its properties

As in the partition of Algorithm 1, we partition the vertices into a logarithmic number of sets. We remove a set of vertices at each of logarithmically many phases based on the degrees of the vertices being considered. However, whereas in Algorithm 1, only high degree vertices were removed at each phase, here we may remove additional vertices. In the following, we mainly focus on describing differences from the partition constructed by Algorithm 1.

Let $\Delta \stackrel{\text{def}}{=} 2+\lceil\log_\alpha 2k_\alpha\rceil$. For all $i\in\{1,\ldots,k_\alpha\}$, recall that G_i is the graph remaining in the i-th loop of Algorithm 1, and V_i is the set of vertices of G_i. We define E_i as the edges incident to vertices in V_i in graph G_i. The new partition differs as follows:

1. We stop partitioning the graph when $i\le\Delta$ (i.e., step 3 of Algorithm 1 is changed to "for $i := k_\alpha$ to $\Delta+1$"). The graph G_Δ has degree bounded by $\alpha^\Delta \le 2\alpha^3 k_\alpha = O(\log n)$, and we use the simple data structure of Section 2.2 to maintain a maximal matching $M_\star$ in that graph. Each update costs $O(\log n)$.

2. For each $i>\Delta$, we select a set V_i in the same way as in Algorithm 1. Each $i>\Delta$ is either *heavy* or *light*. In general, when $|V_i|$ is sufficiently large, then i is heavy, and when $|V_i|$ is sufficiently small, then i is light, but there is a range of $|V_i|$ for which either alternative may be the case. In Section 4.4, we describe how the choice is made so that we can bound the amortized complexity of the data structure.

   For each $i>\Delta$, one of the following two is the case in the partition:

   heavy i: Apart from V_i, the partition also identifies a set of vertices V_i', and a matching $M_i\subseteq E_i$ such that $|M_i|\ge|V_i\cup V_i'|/C_1$, and each edge in M_i connects vertices in $V_i\cup V_i'$. G_{i-1} is created by removing not only vertices in V_i, but also those in V_i'. $V_i\cup V_i'$ is a part of the current approximate vertex cover.

   light i: A given i can only be light if $|E_i|\le C_{\mathrm{T}}\cdot\alpha^i$. In this case, we create G_{i-1} by removing only vertices in V_i. V_i is a part of the current approximate vertex cover.

4.2 Approximating the Maximum Matching and the Minimum Vertex Cover

We now describe how the above partition is used to maintain a constant-factor approximation to both the maximum matching and the minimum vertex cover.

The current matching and the current vertex cover kept by the data structure are the following:

- At the end of every `Insert` and `Delete` operation, we compute a matching M_{light} that matches at least one vertex in V_i for each light i with non-empty V_i. We pick one vertex from each such V_i. There are at most k_α such vertices, and each of them has degree greater than $\alpha^\Delta/\alpha^2\ge 2k_\alpha$. This means that considering them in any order and going over the list of their neighbors, we eventually find a neighbor that has not yet been matched. This way, we get a matching of size at least half the number of light i. The whole procedure takes at most $O(\log^2 n)$ time, because we consider at most $O(\log^2 n)$ vertices, and for each of them we check whether it is already matched in M_{light}.

  We write M_{heavy} to denote the union of all M_i for i heavy. Note that $M_{\text{heavy}}\cup M_\star$ is a matching as well. Combining $M_{\text{heavy}}\cup M_\star$ with M_{light} gives a graph of degree at most 2, and we use the simple data structure of Section 2.2 to maintain a matching $\widetilde{M}$ of size at least $\frac{1}{2}\max\{|M_{\text{light}}|, |M_{\text{heavy}}|+|M_\star|\}$. $\widetilde{M}$ is the current matching.

- The current vertex cover $\widetilde{V}$ is the union of all V_i for $i > \Delta$, V'_i for heavy $i > \Delta$, and the vertices matched in $M_\star$.

Since $|\widetilde{M}| \leq \mathrm{MM}(G) \leq \mathrm{VC}(G) \leq |\widetilde{V}|$ (see Fact 1), it suffices to show that $|\widetilde{M}| \geq |\widetilde{V}|/C$, for some constant C to prove that $\widetilde{M}$ and $\widetilde{V}$ are constant factor approximations to maximum matching and vertex cover, respectively. Note that

$$\begin{aligned}
\left|\bigcup_{\text{light } i} V_i\right| &\leq 2 \cdot \alpha^2 \cdot C_{\mathrm{T}} \cdot |M_{\text{light}}|, \\
\left|\bigcup_{\text{heavy } i} (V_i \cup V'_i)\right| &\leq C_1 \cdot |M_{\text{heavy}}|, \\
|\{\text{endpoints of edges in } M_\star\}| &\leq 2 \cdot |M_\star|.
\end{aligned}$$

Therefore,

$$\begin{aligned}
\left|\widetilde{V}\right| &\leq 2\alpha^2 C_{\mathrm{T}} \cdot |M_{\text{light}}| + C_1 \cdot |M_{\text{heavy}}| + 2 \cdot |M_\star| \\
&\leq 2\alpha^2 C_{\mathrm{T}} \cdot |M_{\text{light}}| + 2C_1 \cdot (|M_{\text{heavy}}| + |M_\star|) \\
&\leq 2\alpha^2 C_1 C_{\mathrm{T}} \cdot |M_{\text{light}}| + 2\alpha^2 C_1 C_{\mathrm{T}} \cdot (|M_{\text{heavy}}| + |M_\star|) \\
&\leq 4\alpha^2 C_1 C_{\mathrm{T}} \cdot \max\{|M_{\text{light}}|, |M_{\text{heavy}}| + |M_\star|\} \\
&= 8\alpha^2 C_1 C_{\mathrm{T}} \cdot \frac{1}{2} \max\{|M_{\text{light}}|, |M_{\text{heavy}}| + |M_\star|\} \\
&\leq 8\alpha^2 C_1 C_{\mathrm{T}} \cdot |\widetilde{M}|.
\end{aligned}$$

4.3 Selecting a Large Matching M_i of V_i

We now describe how to select a large matching M_i for i heavy. The lemma states properties of our procedure, which we will use in the next section.

LEMMA 5. *Let G_i be a graph of maximum degree α^i. Let n be the number of vertices in G_i. Let V_i be a set of vertices in G_i, each with degree in $[\alpha^{i-2}, \alpha^i]$. Let E_i be the set of edges in G_i incident to at least one vertex in V_i, and let $|E_i| \geq C_{\mathrm{T}} \cdot \alpha^i$.*

Let $\mathcal{S}$ be the distribution on subsets of E_i created by independently selecting every edge in E_i with probability $p \stackrel{def}{=} 1/(C_2\alpha^i)$.

There is an algorithm $\mathcal{A}$ that with the following properties:

- *$\mathcal{A}$ selects a subset E' of edges from a distribution $\mathcal{S}'$ on subsets of E_i such that the statistical distance between $\mathcal{S}'$ and $\mathcal{S}$ is at most $1/1000$.*
- *The size of E' is at most $\frac{21}{20} \cdot p|E_i|$.*
- *With probability $998/1000$, $\mathcal{A}$ outputs a matching M that is a subset of E' and $|M| \geq \frac{9}{10} \cdot p|E_i|$.*

The running time of $\mathcal{A}$ is $O(|E_i| \log n)$.

PROOF. The set E' is selected as follows. The algorithm goes over all edges in E_i, and selects each edge independently with probability p. If at some point the number of selected edges reaches $\frac{21}{20} \cdot p|E_i|$, the procedure stops selecting new edges. If C_{T} and C_2 are large enough, then this does not happen with probability greater than $1/1000$ (via the Chernoff bound), so the statistical distance between $\mathcal{S}$ and $\mathcal{S}'$ is at most $1/1000$.

Suppose for now that E' is selected according to $\mathcal{S}$, not $\mathcal{S}'$. If C_2 is large enough, then the probability that a given edge in E' intersects with another edge in E' is small. Let M be the set of all those edges in E' that do not intersect with other edges in E'. For sufficiently large C_2 and C_{T} such that $C_2 \ll C_{\mathrm{T}}$, the size of M is close to its expectation via the Chernoff bound, and in particular $|M| \geq \frac{9}{10} p|E_i|$ with probability $999/1000$. Since the statistical distance between $\mathcal{S}$ and $\mathcal{S}'$ is at most $1/1000$, $|M| \geq \frac{9}{10} p|E_i|$ with probability at least $998/1000$, even if E' is selected from $\mathcal{S}'$. □

4.4 Maintaining the Partition

We now describe how the partition is maintained. As before the data structure keep a value `index`$[v]$ for each vertex v. This time `index`$[v]$ can only belong to the set $\{\Delta, \Delta+1, \Delta+2, \ldots, k_\alpha - 1, k_\alpha\}$. The value Δ corresponds to v remaining in the graph G_Δ, for which a separate copy of the simple data structure is kept. Additionally, for each vertex v with `index`$[v] > \Delta$, there is an additional Boolean variable `promoted`$[v]$. If v belongs to $V_{\texttt{index}[v]}$ in the partition, then `promoted`$[v]$ = **false**. Otherwise, if v belongs to $V'_{\texttt{index}[v]}$, then `promoted`$[v]$ = **true**. Initially, each $j \in \{\Delta+1, \Delta+2, \ldots, k_\alpha - 1, k_\alpha\}$ is set to *light*.

We wish that all vertices v obey the invariant that there be no $j >$ `index`$[v]$ such that the number of neighbors u of v with `index`$[u] \leq j$ is at least α^{j-1}. Furthermore, we require that for v with `promoted`$[v]$ = **false** and `index`$[v] > \Delta$, the number of neighbors u with `index`$[u] \leq$ `index`$[v]$ be greater than $\alpha^{\texttt{index}[v]-2}$. Note that the difference from the previous data structures is that some vertices, namely those with `promoted`$[v]$ = **true**, do not obey the second invariant.

As before, some vertices will be marked as dirty in the course of the execution of the algorithm. When an edge is removed or added, we mark its endpoints as dirty and update its structures. Then our algorithm considers consecutive j starting with k_α and goes down to $\Delta+1$. For a given $i > \Delta$:

1. The algorithm checks if there are dirty vertices v with `index`$[v] < i$ such that the number of neighbors u with `index`$[u] \leq i$ is at least α^{i-1}. Those vertices have `index`$[v]$ set to i, and the algorithms updates structures `below`$[v, \cdot]$, `lower-neighbors`$[v]$, and `neighbors`$[v, \cdot]$ for them and for their neighbors u with `index`$[u] \leq i$ accordingly, as we did for the deterministic data structures. Furthermore, if there are vertices v with `index`$[v] = i$ and `promoted`$[v]$ = **true** with the same property of the number of neighbors, the algorithm sets `promoted`$[v]$ = **false** for them. Finally, if there are vertices v with `index`$[v] = i$, `promoted`$[v]$ = **false**, and the number of neighbors u with `index`$[u] \leq i$ is at most α^{i-2}, the algorithm sets `index`$[v] = i - 1$ and updates all the structures accordingly, also marking specific neighbors as dirty whenever necessary.

2. **If i is heavy:**

   The algorithm checks if the old matching is large enough, and if not, the algorithm deletes it, and computes a new matching. More specifically, if it is no longer the case that $|M_i| \geq |V_i \cup V'_i|/C_1$, the algorithm goes over all v with `index`$[v] = i$ and `promoted`$[v]$ = **true**. For each such v, we set `promoted`$[v]$ to **false**. Moreover, for those v with the number of neighbors u with `index`$[u] \leq i$ at most α^{i-2}, we set `index`$[v] = i-1$ and mark them as dirty.

If now $|E_i|$ is at most $C_{\mathrm{T}} \cdot \alpha^i$, we make i light. Otherwise, we use the procedure of Lemma 5 to select a new matching M_i. We set V_i' to the set of vertices matched by M_i that do not belong to V_i. As long as $|M_i| < \frac{9}{10}|E_i|/(C_2\alpha^i)$, we keep repeating the procedure of Lemma 5 until we succeed. For well chosen constants, $\frac{9}{10}|E_i|/(C_2\alpha^i) \geq |V_i \cup V_i'|/C_1$, since $C_1 \gg C_2$. Finally, for endpoints v of edges in the new M_i that have $\texttt{index}[v] < i$, we set $\texttt{index}[v] = i$ and $\texttt{promoted}[v] = \textbf{true}$, and update all the structures for them and their neighbors accordingly, marking some of them as dirty.

3. **If i is light:**

   If $|E_i| > C_{\mathrm{T}} \cdot \alpha^i$, we make i heavy and construct M_i in the same way as for heavy i.

   Otherwise, if $|E_i| \leq C_{\mathrm{T}} \cdot \alpha^i$, and we do nothing.

4.5 Complexity Analysis

We use almost the same potential functions for vertices as before. The only difference is that we multiply all potentials by a constant factor so that when an edge is inserted into some set E_i or removed from it, we can pass one unit of potential to a special fund. We will use this fund to pay for the cost of recomputing matchings M_i. We will show that with large probability the fund deficit is small.

Whenever the data structure artificially moves a vertex v, creating or destroying some V_i', one has to cover the cost associated with changing $\texttt{index}[v]$. In the deterministic data structure the cost of moving vertices around was covered by the collected potential. Here, we have to find another source of funding.

Suppose that we want to create a matching M_i for a heavy i. We run the algorithm of Lemma 5, charging its running time to the fund. If M_i is sufficiently large, we spend $O(|V_i|\alpha^i)$ units of potential from the fund on moving vertices in V' from other buckets (where the constant hidden in the big O notation is very small). Later, moving vertices in V' back to their buckets will cost approximately the same, so we can assume that we charge this cost to the fund in advance. If M_i that has been generated is too small and the data structure has to rerun the process generating M_i, we say that we *lose*.

The matching M_i is relatively large compared to $|V_i \cup V_i'|$ right after we create it. The matching requires recomputation if it becomes relatively small compared to $|V_i \cup V_i'|$. For this to happen, at least one of the following two must be the case:

- A constant fraction of edges in M_i have been removed from E_i.
- The number of vertices in V_i must have grown by a constant factor.

Consider first the latter case. Since each new vertex in V_i contributes to the special fund at least α^{i-1} units of potential, it is easy to set constants so that we can afford to pay for moving vertices in V_i' (if their number is sufficiently small, which is the case if C_2 is large) and for the initial execution of the algorithm of Lemma 5, which requires only $\Theta(|V_i|\alpha^i)$ units of potential. We can also set the constants in the data structure such that in fact, we are left with a surplus of potential. We make sure that we collect a lot of potential. We say that we *win* in this case.

The former case requires probabilistic analysis, which we now describe. Recall that a well chosen M_i is at least a 4/5-fraction of a subset E' of edges E_i. Therefore, to delete at least half the edges of the initial M_i, one has to delete at least a 2/5-fraction of E'. We claim that with probability at least 3/4, one has to delete at least a 1/100-fraction of E_i in order to delete a 2/5-fraction of E'. We now sketch a proof of this claim.

> Suppose to the contrary that one can delete at least a 2/5-fraction of E' by deleting at most a 1/100-fraction of E_i with some probability greater than 1/4. Then, given how E' is selected in Lemma 5, one can delete at least $|E_i|/(5C_2\alpha^i)$ edges of E'' with probability at least $1/4 - 1/100$ (for well chosen constants) by selecting a subset of E_i of size $|E_i|/100$, where E'' is created by independently selecting each edge of E_i with probability $1/(C_2\alpha^i)$. Expressing the last sentence in a slightly different way, the sum of $|E_i|/100$ independent variables X_j is at least $|E_i|/(5C_2\alpha^i)$ with probability at least 24/100, where each X_j is 1 with probability $1/(C_2\alpha^i)$, and 0, otherwise. Using the Chernoff bound, one can show that this is not the case for well chosen constants.

The claim implies that with probability at least 3/4, we collect a lot of potential before a large fraction of M_i gets deleted, and also in this case, we say that we *win*. Otherwise, when the edges are deleted very quickly, we say that we *lose*.

Summarizing, we win with probability at least 2/3, in which case we collect a lot of potential (a large constant times the invested potential), which goes to the fund. We lose with probability at most 1/3, and in this case, the operation is paid by the fund. Consider a logarithmic number of ranges $[2^j, 2^{j+1})$ corresponding to different sizes of $|E_i|$ that may appear in the data structure. For a given range, we can assume that whenever we play, we lose at most $C \cdot 2^j$, for some constant C, and we win at least $10 \cdot C \cdot 2^j$. We initially provide the fund with enough potential to pay for the first $t_\star \stackrel{\text{def}}{=} C' \cdot \log \frac{\log n}{\delta}$ times we play the game, for every j of interest, where C' is a sufficiently large constant. Then, the probability that we ever spend more than we gain for a given j is bounded by $\frac{\delta}{2 \log n}$. To prove this, it suffices to give an upper bound p_t on the probability that we spend more than win in t games. Using the Chernoff bound, one can show p_t's such that $p_{t+1} \leq p_t \cdot c$, for $t \geq t_\star$, where c is a constant in $(0, 1)$. Using the Chernoff bound again, one can show that if C' is sufficiently large, $p_{t_\star} \leq (1 - C)\frac{\delta}{2 \log n}$, and $\sum_{t=t_\star}^{\infty} p_t \leq \frac{\delta}{2 \log n}$. So by the union bound, the probability that we ever spend more than we gain for any j is bounded by δ.

Recall that K is the total number of graph operations, which gives a bound on the maximum size of E_i that can appear. The above analysis implies that for every j such that $2^j \leq n^2$ and $K \geq 2^j$, we can subsidize the fund with $O(2^j \cdot \log \frac{\log n}{\delta})$ units of potential to make sure that with probability $1 - \delta$, the fund's balance is always non-negative. In total, the aid for the fund is bounded by $O(\min\{K, n^2\} \cdot \log \frac{\log n}{\delta})$. Therefore, the total potential $\mathcal{P}$ spent by the al-

gorithm can be bounded by

$$\begin{aligned}\mathcal{P} &= O\left(\min\{K, n^2\} \cdot \log \frac{\log n}{\delta} + K \cdot \log n\right) \\ &= O\left(\min\{K, n^2\} \cdot \log \frac{1}{\delta} + K \cdot \log n\right)\end{aligned}$$

with probability $1-\delta$. Recall that each unit of potential corresponds to $O(\log n)$ computation. Other operations, which include computing M_{light} and combining the three matchings, do not take more than $O(\log^2 n)$ time per update to the graph. Summarizing, we prove the following claim.

COROLLARY 6. *For any sequence of K updates, the randomized data structure runs in*

$$O\left(\min\{K, n^2\} \cdot \log n \cdot \log \frac{1}{\delta} + K \cdot \log^2 n\right)$$

time with probability $1-\delta$, where $\delta \in (0,1)$.

5. OPEN PROBLEMS

The two main questions left open by our paper are:

- Our approximation factors are large constants. How small can they be made with polylogarithmic update time? Can they be made 2? Can the approximation constant be made smaller than 2 for maximum matching?
- Is there a *deterministic* data structure that achieves a constant approximation factor with polylogarithmic update time?

6. REFERENCES

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition*. The MIT Press and McGraw-Hill Book Company, 2001.

[2] D. Eppstein, Z. Galil, and G. F. Italiano. *Dynamic graph algorithms*. CRC Press, 1997.

[3] D. Eppstein, Z. Galil, G. F. Italiano, and A. Nissenzweig. Sparsification—a technique for speeding up dynamic graph algorithms. *J. ACM*, 44(5):669–696, 1997.

[4] L. Epstein, A. Levin, J. Mestre, and D. Segev. Improved approximation guarantees for weighted matching in the semi-streaming model. In *STACS*, 2010.

[5] J. Feigenbaum, S. Kannan, A. McGregor, S. Suri, and J. Zhang. On graph problems in a semi-streaming model. *Theor. Comput. Sci.*, 348(2-3):207–216, 2005.

[6] M. R. Henzinger and V. King. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *J. ACM*, 46(4):502–516, 1999.

[7] Z. Ivković and E. L. Lloyd. Fully dynamic maintenance of vertex cover. In *WG*, pages 99–111, 1993.

[8] P. N. Klein and S. Subramanian. A fully dynamic approximation scheme for shortest paths in planar graphs. *Algorithmica*, 22(3):235–249, 1998.

[9] Z. Lotker, B. Patt-Shamir, and A. Rosén. Distributed approximate matching. In *PODC*, pages 167–174, 2007.

[10] A. McGregor. Finding graph matchings in data streams. In *APPROX-RANDOM*, pages 170–181, 2005.

[11] S. Micali and V. V. Vazirani. An $O(\sqrt{|V|} \cdot |E|)$ algorithm for finding maximum matching in general graphs. In *FOCS*, pages 17–27, 1980.

[12] M. Parnas and D. Ron. Approximating the minimum vertex cover in sublinear time and a connection to distributed algorithms. *Theor. Comput. Sci.*, 381(1-3):183–196, 2007.

[13] P. Sankowski. Faster dynamic matchings and vertex connectivity. In *SODA*, pages 118–126, 2007.

[14] M. Thorup. Fully-dynamic min-cut. In *STOC*, pages 224–230, 2001.

[15] M. Thorup. Worst-case update times for fully-dynamic all-pairs shortest paths. In *STOC*, pages 112–119, 2005.

[16] M. Zelke. Weighted matching in the semi-streaming model. In *STACS*, pages 669–680, 2008.