Finding Graph Matchings in Data Streams

Andrew McGregor*

Department of Computer and Information Science, University of Pennsylvania
Philadelphia, PA 19104, USA
andrewm@cis.upenn.edu

Abstract. We present algorithms for finding large graph matchings in the streaming model. In this model, applicable when dealing with massive graphs, edges are streamed-in in some arbitrary order rather than residing in randomly accessible memory. For $\epsilon > 0$, we achieve a $\frac{1}{1+\epsilon}$ approximation for maximum cardinality matching and a $\frac{1}{2+\epsilon}$ approximation to maximum weighted matching. Both algorithms use a constant number of passes and $\tilde{O}(|V|)$ space.

1 Introduction

Given a graph $G = (V, E)$, the *Maximum Cardinality Matching* (MCM) problem is to find the largest set of edges such that no two adjacent edges are selected. More generally, for an edge-weighted graph, the *Maximum Weighted Matching* (MWM) problem is to find the set of edges whose total weight is maximized subject to the condition that no two adjacent edges are selected. Both problems are well studied and exact polynomial solutions are known [1–4]. The fastest of these algorithms solves MWM with running time $O(nm + n^2 \log n)$ where $n = |V|$ and $m = |E|$.

However, for massive graphs in real world applications, the above algorithms can still be prohibitively computationally expensive. Examples include the virtual screening of protein databases. (See [5] for other examples.) Consequently there has been much interest in faster algorithms, typically of $O(m)$ complexity, that find good approximate solutions to the above problems. For MCM, a linear time approximation scheme was given by Kalantari and Shokoufandeh [6]. The first linear time approximation algorithm for MWM was introduced by Preis [7]. This algorithms achieved a $1/2$ approximation ratio. This was later improved upon by the $(2/3 - \epsilon)$ linear time[1] approximation algorithm given by Drake and Hougardy [5].

In addition to concerns about time complexity, when computing with massive graphs it is no longer reasonable to assume that we can store the entire input graph in *random access memory*. In this case the above algorithms are not applicable as they require random access to the input. With this in mind, we

* This work was supported by NSF ITR 0205456.

[1] Note that here, and throughout this paper, we assume that ϵ is an arbitrarily small *constant*.

C. Chekuri et al. (Eds.): APPROX and RANDOM 2005, LNCS 3624, pp. 170–181, 2005.

consider the *graph stream model* discussed in [8–10]. This is a particular formulation of the *data stream model* introduced in [11–14]. In this model the edges of the graph stream-in in some arbitrary order. That is, for a graph $G = (V, E)$ with vertex set $V = \{v_1, v_2, \ldots, v_n\}$ and edge set $E = \{e_1, e_2, \ldots, e_m\}$, a *graph stream* is the sequence $e_{i_1}, e_{i_2}, \ldots, e_{i_m}$, where $e_{i_j} \in E$ and $i_1, i_2, \ldots, i_m$ is an arbitrary permutation of $\{1, 2, \ldots, m\}$.

The main computational restriction of the model is that we have limited space and therefore we can not store the entire graph (this would require $\tilde{O}(m)$ space.) In this paper we restrict our attention to algorithms that use $\tilde{O}(n)$ [2] space. This restriction was identified as a "sweet-spot" for graph streaming in a summary article by Muthukrishnan [8] and subsequently shown to be necessary for the verification of even the most primitive of graph properties such as connectivity [10]. We may however have multiple passes of the graph stream (as was assumed in [10, 11]). To motivate this assumption one can consider external memory systems in which seek times are typically the bottleneck when accessing data. In this paper, we will assume that we can only have constant passes. Lastly it is important to note that while most streaming algorithms use polylog(m) space on a stream of length m, this is not always the case. Examples include the streaming-clustering algorithm of Guha *et al.* [15] that uses m^ϵ space and the "streaming"[3] algorithm of Drineas and Kannan [16] that uses space $\sqrt{m}$.

Our Results: In this paper we present algorithms that achieve the following approximation ratios:

1. For $\epsilon > 0$, a $\frac{1}{1+\epsilon}$ approximation to maximum cardinality matching.
2. For $\epsilon > 0$, a $\frac{1}{2+\epsilon}$ approximation to maximum weighted matching.

MCM and MWM have previously been studied under similar assumptions by Feigenbaum *et al.* [9]. The best previously attained results were a $\frac{1}{6}$ approximation to MWM and for $\epsilon > 0$ and a $\frac{2}{3+\epsilon}$ approximation to MCM on the assumption that the graph is a bipartite graph. Also in the course of this paper we tweak the $\frac{1}{6}$ approximation to MWM to give a $\frac{1}{3+2\sqrt{2}}$ approximation to MWM that uses only one pass of the graph stream.

2 Unweighted Matchings

2.1 Preliminaries

In this section we describe a streaming algorithm that, for $\epsilon > 0$, computes a $1/(1 + \epsilon)$ approximation to the MCM of the streamed graph. The algorithm will use a constant number of passes. We start by giving some basic definitions common to many matching algorithms.

[2] Sometimes known as the *semi-streaming* space restriction.

[3] Instead of "streaming," the authors of [16] use the term "pass-efficient" algorithms.

Definition 1 (Basic Matching Theory Definitions). *Given a matching M in a graph $G = (V, E)$, we call a vertex* free *if it does not appear as the end point of any edge in M. A length $2i+1$* augmenting path *is a path $u_1 u_2 \dots u_{2i+2}$ where u_1 and u_{2i+2} are free vertices and $(u_j, u_{j+1}) \in M$ for even j and $(u_j, u_{j+1}) \in E \setminus M$ for odd j.*

Note that if M is a matching and P is an augmenting path then $M \triangle P$ (the symmetric difference of M and P) is a matching of size strictly greater than M. Our algorithm will start by finding a maximal matching and then, by finding short augmenting paths, increase the size of the matching by making local changes. Note that finding a maximal matching is easily achieved in one pass – we select an edge iff we have not already selected an adjacent edge. Finding maximal matchings in this way will be at the core of our algorithm and we will make repeated use of the fact that the maximum matching has cardinality at most twice that of any maximal matching.

The following lemma establishes that, when there are few short augmenting paths, the size of the matching found can be lower-bound in terms of the size of the maximum cardinality matching Opt.

Lemma 1. *Let M be a maximal matching and* Opt *be a matching of maximum cardinality. Consider the connected components in* Opt$\triangle M$*. Ignoring connected components with the same number of edges from M as from* Opt*, let $\alpha_i M$ be the number of connected components with i edges from M. Then*

$$\max_{1 \le i \le k} \alpha_i \le \frac{1}{2k^2(k+1)} \Rightarrow M \ge \frac{\text{Opt}}{1 + 1/k}$$

Proof. In each connected component with i edges from M there is either i or $i+1$ edges from Opt. Therefore, Opt $\le \sum_{1 \le i \le k} \alpha_i \frac{i+1}{i} |M| + \frac{k+2}{k+1}(1 - \sum_{1 \le i \le k} \alpha_i)|M|$. By assumption

$$\sum_{1 \le i \le k} \alpha_i \frac{i+1}{i} + \frac{k+2}{k+1}(1 - \sum_{1 \le i \le k} \alpha_i) \le \frac{1}{k(k+1)} + \frac{k+2}{k+1} = (1 + 1/k)$$

The result follows.

So, if there are $\alpha_i M$ components in Opt$\triangle M$ with $i+1$ edges from Opt and i edges from M, then there are at least $\alpha_i M$ length $2i+1$ augmenting paths for M. Finding an augmenting path allows us to increase the size of M. Hence, if $\max_{1 \le i \le k} \alpha_i$ is small we already have a good approximation to Opt whereas, if $\max_{1 \le i \le k} \alpha_i$ is large then there exists $1 \le i \le k$ such that there are many length $2i+1$ augmenting paths.

2.2 Description of the Algorithm

Now we have defined the basic notion of augmenting paths, we are in a position to give an overview of our algorithm. We have just reasoned that, if our matching

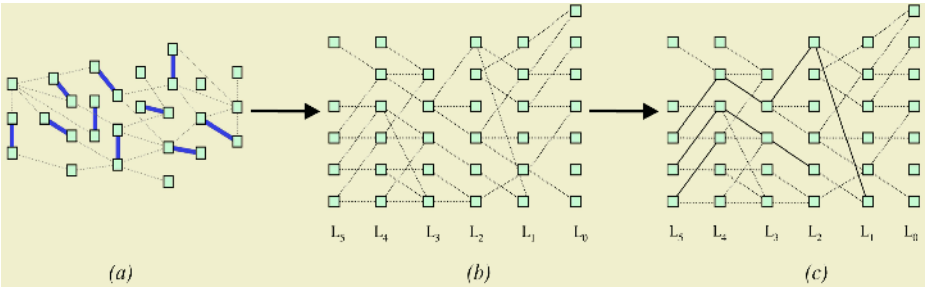

Fig. 1. A schematic of the procedure for finding length 9 augmenting paths. Explained in the text.

is not already large, then there exists augmenting paths of some length no greater than $2k+1$. Our algorithm looks for augmenting paths of each of the k different lengths separately. Consider searching for augmenting paths of length $2i+1$. See Fig. 1 for a schematic of this process when $i = 4$. In this figure, (a) depicts the graph G with heavy solid lines denoting edges in the current matching. To find length $2i+1$ we randomly "project" the graph (and current matching) into a set of graphs, $\mathcal{L}_i$, which we now define.

Definition 2. *Consider a graph whose n nodes are partitioned into $i+2$ layers $L_{i+1}, \ldots L_0$ and whose edgeset is a subset of $\cup_{1 \le j \le i+1}\{(u,v) : u \in L_j, v \in L_{j-1}\}$. We call the family of such graphs $\mathcal{L}_i$. We call a path $u_l, u_{l-1}, \ldots u_0$ such that $u_j \in L_j$ an l-*path.

The random projection is performed as follows. The image of a free node in $G = (V, E)$ is a node in either L_{i+1} or L_0. The image of a matched edge $e = (v, v')$ is a node, either $u_{(v,v')}$ or $u_{(v',v)}$, in one of $L_i, \ldots L_1$ chosen at random. The edges in the projected graph G' are those in G that are "consistent" with the mapping of the free nodes and the matched edges, ie. there is an edge between a node $u_{(v_1,v_2)} \in L_j$ and $u_{(v_3,v_4)} \in L_{j-1}$ if there is an edge $(v_2, v_3) \in E$. Now note that an $i+1$-path in G' corresponds to a $2i+1$ augmenting path in G. Unfortunately the converse is not true, there may be $2i+1$ augmenting paths in G that do not correspond to $i+1$-paths in G' because we only consider consistent edges. However, we will show later that a constant fraction of augmenting paths exist (with high probability) as $i+1$-paths in G'. In Figure 1, (b) depicts G', the layered graph formed by randomly projecting G into $\mathcal{L}_i$.

We now concern ourselves with finding a nearly maximal set of node disjoint $i+1$-paths in a graph G'. See algorithm *Find-Layer-Paths* in Figure 2. The algorithm finds node disjoint $i+1$-paths by doing something akin to a depth first search. Finding a maximal set of node disjoint $i+1$-paths can easily be achieved in the RAM model by actually doing a DFS, deleting nodes of found $i+1$-paths and deleting edges when backtracking. Unfortunately this would necessitative too many passes in the streaming model as each backtrack potentially requires another pass of the data. Our algorithm in essence blends a DFS and BFS in

such a way that we can substantially reduce the number of backtracks required. This will come at the price of possibly stopping prematurely, ie. when there may still exist some $i+1$-paths that we have not located.

The algorithm first finds a maximal matching between L_{i+1} and L_i. Let S' be the subset of nodes L_i involved in this first matching. It then finds a maximal matching between S' and L_{i-1}. We continue in this fashion, finding a matching between $S'' = \{u \in L_{i-1} : u \text{ matched to some } u' \in L_i\}$ and L_{i-2}. One can think of the algorithm as growing node disjoint paths from left to right. (Fig. 1 (c) tries to capture this idea. Here, the solid lines represent matchings between the layers.) If the size of the maximal matching between some subset S of a level L_j and L_{j-1} falls below a threshold we declare all vertices in S to be dead-ends and conceptually remove them from the graph (in the sense that we never again use these nodes while try to find $i+1$-paths.) At this point we start back-tracking. It is the use of this threshold that ensures a limit on the amount of back-tracking performed by the algorithm. However, because of the threshold, it is possible that a vertex may be falsely declared to be a dead-end, ie. there may still be a node disjoint path that uses this vertex. With this in mind we want the threshold to be low such that this does not happen often and we can hope to find all but a few of a maximal set of node disjoint $i+1$-paths. When we grow some node disjoint paths all the way to L_0, we remove these paths and recurse on the remaining graph. For each node v, the algorithm maintains a tag indicating if it is a "Dead End" or, if we have found a $i+1$ path involving v, the next node in the path.

It is worth reiterating that in each pass of the stream we simply find a maximal matching between some set of nodes. The above algorithm simply determines within which set of nodes we find a maximal matching.

Our algorithm is presented in detail in Fig. 2. Here we use the notation $s \in_R S$ to denote choosing an element s uniformly at random from a set S. Also, for a matching M, $\Gamma_M(u) = v$ if $(u,v) \in M$ and $\emptyset$ otherwise.

2.3 Correctness and Running Time Analysis

We first argue that the use of thresholds in *Find-Layer-Paths* ensures that we find all but a small number of a maximal set of $i+1$-paths.

Lemma 2 (Running Time and Correctness of *Find-Layer-Paths*). *Given $G' \in \mathcal{L}_i$, Find-Layer-Paths algorithm finds at least $(\gamma - \delta)|M|$ of the $i+1$-paths where $\gamma|M|$ is the size of some maximal set of $i+1$-paths. Furthermore the algorithm takes a constant number of passes.*

Proof. First note that *Find-Layer-Paths*$(\cdot,\cdot,\cdot,l)$ is called with argument $\delta^{2^{i+1-l}}$. During the running of *Find-Layer-Paths*$(\cdot,\cdot,\cdot,l)$ when we run line 15, the number of $i+1$-paths we rule out is at most $2\delta^{2^{i+1-l}}|L_{l-1}|$ (the factor 2 comes from the fact that a maximal matching is at least half the size of a maximum matching.) Let E_l be the number of times *Find-Layer-Paths*$(\cdot,\cdot,\cdot,l)$ is called: $E_{i+1} = 1$, $E_l \leq E_{l+1}/\delta^{2^{i+1-l}}$ and therefore $E_l \leq \delta^{-\sum_{0 \leq j \leq i-l} 2^j} = \delta^{-2^{i-l+1}+1}$. Hence, we

Algorithm *Find-Matching*(G, ϵ)
($*$ Finds a matching $*$)
Output: A matching
1. Find a maximal matching M
2. $k \leftarrow \lceil \frac{1}{\epsilon} + 1 \rceil$
3. $r \leftarrow 4k^2(8k+10)(k-1)(2k)^k$
4. **for** $j = 1$ to r:
5. **for** $i = 1$ to k:
6. **do** $M_i \leftarrow$ *Find-Aug-Paths*(G, M, i)
7. $M \leftarrow \text{argmax}_{M_i}|M_i|$
8. **return** M

Algorithm *Find-Aug-Paths*(G, M, i)
($*$ Finds length $2i+1$ augmenting paths for a matching M in G $*$)
1. $G' \leftarrow$*Create-Layer-Graph*(G, M, i)
2. $\mathcal{P} =$*Find-Layer-Paths*$(G', L_{i+1}, \frac{1}{r(2k+2)}, i+1)$
3. **return** $M \triangle \mathcal{P}$

Algorithm *Create-Layer-Graph*(G, M, i)
($*$ Randomly constructs $G' \in \mathcal{L}_i$ from a graph G and matching M $*$)
1. **if** v is a free vertex **then** $l(v) \in_R \{0, i+1\}$
2. **if** $e=(u,v) \in M$ **then** $j \in_R [i]$ and $l(e) \leftarrow j$, $l(u) \leftarrow ja$ and $l(v) \leftarrow jb$ or vice versa.
3. $E_i \leftarrow \{(u,v) \in E : l(u) = i+1, l(v) = ia\}$, $E_0 \leftarrow \{(u,v) \in E : l(u) = 1b, l(v) = 0\}$
4. **for** $j = 0$ to $i+1$
5. **do** $L_j \leftarrow l^{-1}(j)$
6. **for** $j = 1$ to $i-1$
7. **do** $E_j \leftarrow \{(u,v) \in E : l(u) = (j+1)b, l(v) = ja\}$
8. **return** $G' = (L_{i+1} \cup L_i \cup \ldots \cup L_0, E_i \cup E_{i-1} \cup \ldots \cup E_0)$

Algorithm *Find-Layer-Paths*(G', S, δ, j)
($*$ Finds many j-paths from $S \subset L_j$ $*$)
1. Find maximal matching M' between S and untagged vertices in L_{j-1}
2. $S' \leftarrow \{v \in L_{j-1} : \exists u, (u,v) \in M'\}$
3. **if** $j = 1$
4. **then if** $u \in \Gamma_{M'}(L_{j-1})$ **then** $t(u) \leftarrow \Gamma_{M'}(u)$, $t(\Gamma_{M'}(u)) \leftarrow \Gamma_{M'}(u)$
5. **if** $u \in S \setminus \Gamma_{M'}(L_{j-1})$ **then** $t(u) \leftarrow$ "Dead End "
6. **return**
7. **repeat**
8. *Find-Layer-Paths*$(G', S', \delta^2, j-1)$
9. **for** $v \in S'$ such that $t(v) \neq$"Dead End"
10. **do** $t(\Gamma_{M'}(v)) \leftarrow v$
11. Find maximal matching M' between untagged vertices in S and L_{j-1}.
12. $S' \leftarrow \{v \in L_{j-1} : \exists u, (u,v) \in M'\}$
13. **until** $|S'| \leq \delta |L_{j-1}|$
14. **for** $v \in S$ untagged
15. **do** $t(b) \leftarrow$ "Dead End".
16. **return**

Fig. 2. An Algorithm for Finding Large Cardinality Matchings. (See text for an informal description.)

remove at most $2E_l\delta^{2^{i+1-l}}|L_l| \leq 2\delta|L_l|$. Note that when nodes are labeled as dead-ends in a call to *Find-Layer-Paths*$(\cdot,\cdot,\cdot,1)$, they really are dead-ends and declaring them such rules out no remaining i+1-paths. Hence the total number of paths not found is at most $2\delta\sum_{1\leq j\leq i}|L_j| \leq 2\delta|M|$. The number of invocations of the recursive algorithm is

$$\sum_{1\leq l\leq i+1} E_l \leq \sum_{1\leq l\leq i+1} \delta^{-2^{i+1-l}+1} \leq \delta^{-2^{i+1}}$$

i.e. $O(1)$ and each invocation requires one pass of the data stream to find a maximal matching.

When looking for length $2i+1$ augmenting paths for a matching M in graph G, we randomly create a layered graph $G' \in \mathcal{L}_{i+1}$ using *Create-Layer-Graph* such that $i+1$-paths in G' correspond to length $2i+1$ augmenting paths. We now need to argue that a) many of the $2i+1$ augmenting paths in G exist in G' as $i+1$-paths and b) that finding a maximal, rather that a maximum, set of $i+1$-paths in G' is sufficient for our purposes.

Theorem 1. *If G has $\alpha_i M$ length $2i+1$ augmenting paths, then the number of length $i+1$-paths found in G' is at least*

$$(b_i\beta_i - \delta)|M| \ ,$$

where $b_i = \frac{1}{2i+2}$ and β_i is a random variables distributed as $\mathbf{Bin}(\alpha_i|M|, \frac{1}{2(2i)^i})$.

Proof. Consider a length $2i+1$ augmenting path $P = u_0u_1\ldots u_{2i+1}$ in G. The probability that P appears as an $i+1$-path in G' is at least,

$$2\mathbb{P}\left(l(u_0)=0\right)\mathbb{P}\left(l(u_{2i+1})=i+1\right)\prod_{j\in[i]}\mathbb{P}\left(l(u_{2j})=ja \text{ and } l(u_{2j-1})=jb\right)=\frac{1}{2(2i)^i}.$$

Given that the probability of each augmenting path existing as a $i+1$-path in G' is independent, the number of length $i+1$-paths in G' is distributed as $\mathbf{Bin}(\alpha_i|M|, \frac{1}{2(2i)^i})$. The size of a maximal set of node disjoint $i+1$-paths is at least a $\frac{1}{2i+2}$ fraction of the maximum size node-disjoint set i+1-paths. Combining this with Lemma 2 gives the result.

Finally, we argue that we only need to try to augment our initial matching a constant number of times.

Theorem 2 (Correctness). *With probability $1-f$ by running $O(\log\frac{1}{f})$ copies of the algorithm Find-Matching in parallel we find a $1-\epsilon$ approximation to the matching of maximum cardinality.*

Proof. We show that the probability that a given run of *Find-Matching* does not find a $(1+\epsilon)$ approximation is bounded above by e^{-1}.

Define a *phase* of the algorithm to be one iteration of the loop started at line 4 of *Find-Matching*. At the start of phase p of the algorithm, let M_p be the

current matching. In the course of phase p of the algorithm we augment M_p by at least $|M_p|(\max_{1\le i\le k}(b_i\beta_{i,p})-\delta)$ edges where $\beta_{i,p}|M_p| \sim \mathbf{Bin}(\alpha_{i,p}|M_p|, \frac{1}{2(2i)^i})$. Let A_p be the value of $|M_p|\max_i(b_i\beta_{i,p})$ in the pth phase of the algorithm. Assume that for each of the r phases of the algorithm $\max \alpha_{i,p} \ge \alpha^* := \frac{1}{2k^2(k-1)}$. (By Lemma 1, if this is ever not the case, we already have a sufficiently sized matching.) Therefore, A_p dominates $b_k\mathbf{Bin}(\alpha^*|M_1|, \frac{1}{2(2k)^k})$. Let $(X_p)_{1\le p\le r}$ be independent random variables, each distributed as $b_k\mathbf{Bin}(\alpha^*|M_1|, \frac{1}{2(2k)^k})$. Therefore,

$$\begin{aligned}
&\mathbb{P}\left(|M_1|\prod_{1\le p\le r}(1+\max\{0, \max_{1\le i\le k}(b_i\beta_{i,p})-\delta\}) \ge 2|M_1|\right)\\
&\ge \mathbb{P}\left(\sum_{1\le p\le r}\max_{1\le i\le k} b_i\beta_{i,p} \ge 2+r\delta\right)\\
&\ge \mathbb{P}\left(\sum_{1\le p\le r} X_p \ge |M_1|\frac{2+r\delta}{b_k}\right)\\
&= \mathbb{P}\left(Z \ge |M_1|(4k+5)\right) \ ,
\end{aligned}$$

for $\delta = b_k/r$ where $Z = \mathbf{Bin}(\alpha^*|M_1|r, \frac{1}{2(2k)^k})$. Finally, by an application of the Chernoff bound,

$$\mathbb{P}\left(Z \ge |M_1|(4k+5)\right) = 1-\mathbb{P}\left(Z < \mathbb{E}\left(Z\right)/2\right) > 1-e^{-2(8k+10)|M_1|} \ge 1-e^{-1} \ ,$$

for $r = 2(2k)^k(8k+10)/\alpha^*$. Of course, since M_1 is already at least half the size of the maximal matching. This implies that with high probability, at some point during the r phases our assumption that $\max \alpha_{i,p} \ge \alpha^*$, became invalid and at this point we had a sufficiently large matching.

3 Weighted Matching

We now turn our attention to finding maximum weighted matchings. Here each edge $e \in E$ of our graph G has a weight $w(e)$ (wlog. $w(e) > 0$). For a set of edges S let $w(S) = \sum_{e\in S} w(e)$. We seek to maximize $w(S)$ subject to the constraint that S contains no two adjacent edges.

Consider the algorithms given in Fig. 3. The algorithm *Find-Weighted-Matching* can be viewed as a parameterization of the one pass algorithm given in [9] in which γ was implicitly equal to 1. The algorithm greedily collects edges as they stream past. The algorithm maintains a matching M at all points. On seeing an edge e, if $w(e) > (1+\gamma)w(\{e'|e' \in M, e' \text{ and } e \text{ share an end point}\})$ then the algorithm removes any edges in M sharing an end point with e and adds e to M. The algorithm *Find-Weighted-Matching-Multipass* generalizes this to a multi-pass algorithm that in effect, repeats the one pass algorithm until the improvement yielded falls below some threshold. We start by recapping some

notation introduced in [9]. While unfortunately macabre, this notation is nevertheless helpful for developing intuition.

Definition 3. *In a given pass of the graph stream, we say that an edge e is* born *if $e \in M$ at some point during the execution of the algorithm. We say that an edge is* killed *if it was born but subsequently removed from M by a newer heavier edge. This new edge* murdered *the killed edge. We say an edge is a* survivor *if it is born and never killed. For each survivor e, let the* Trail of the Dead *be the set of edges $T(e) = C_1 \cup C_2 \cup \ldots$, where $C_0 = \{e\}$, $C_1 = \{$the edges murdered by $e\}$, and $C_i = \cup_{e' \in C_{i-1}} \{$the edges murdered by $e'\}$.*

Lemma 3. *For a given pass let the set of survivors be S. The weight of the matching found at the end of that pass is therefore $w(S)$.*

1. $w(T(S)) \leq w(S)/\gamma$
2. $\textsc{Opt} \leq (1+\gamma)\left(w(T(S)) + 2w(S)\right)$

Proof. 1. For each murdering edge e, $w(e)$ is at least $(1+\gamma)$ the cost of murdered edges, and an edge has at most one murderer. Hence, for all i, $w(C_i) \geq (1+\gamma)w(C_{i+1})$ and therefore $(1+\gamma)w(T(e)) = \sum_{i\geq 1}(1+\gamma)w(C_i) \leq \sum_{i \geq 0} w(C_i) = w(T(e)) + w(e)$. The first point follows.

2. We can charge the costs of edges in $\textsc{Opt}$ to the $S \cup T(S)$ such that each edge $e \in T(S)$ is charged at most $(1+\gamma)w(e)$ and each edge $e \in S$ is charged at most $2(1+\gamma)w(e)$. See [9] for details.

Hence in the one pass algorithm we get an $\frac{1}{\frac{1}{\gamma}+3+2\gamma}$ approximation ratio since

$$\textsc{Opt} \leq (1+\gamma)(w(T(S)) + 2w(S)) \leq (3 + \frac{1}{\gamma} + 2\gamma)w(S)$$

The maximum of this function is achieved for $\gamma = \frac{1}{\sqrt{2}}$ giving approximation ratio $\frac{1}{3+2\sqrt{2}}$. This represents only a slight improvement over the 1/6 ratio attained previously. However, a much more significant improvement is realized in the multi-pass algorithm *Find-Weighted-Matching-Multipass*.

Theorem 3. *The algorithm Find-Weighted-Matching-Multipass finds a $\frac{1}{2(1+\epsilon)}$ approximation to the maximum weighted matching. Furthermore, the number of passes required is at most,*

$$\frac{\log(3/2+\sqrt{2})}{\log \frac{(2\epsilon/3)^3}{(1+2\epsilon/3)^2-(2\epsilon/3)^3}} + 1 \ .$$

Proof. First we prove that the number of passes is as claimed. We increase the weight of our solution by a factor $1+\kappa$ each time we do a pass and we start with a $1/(3+2\sqrt{2})$ approximation. Hence, if we take, $\log_{1+\kappa}(3/2+\sqrt{2})$ passes we have already found a maximum weighted matching. Substituting in $\kappa = \frac{\gamma^3}{(1+\gamma)^2-\gamma^3}$ establishes the bound on the number of passes.

Algorithm *Find-Weighted-Matching*(G, γ)
(∗ Finds Large Weighted Matchings in One Pass ∗)
1. $M \leftarrow \emptyset$
2. **for** each edge $e \in G$
3. **do if** $w(e) > (1+\gamma)w(\{e' | e' \in M, e'$ and e share an end point$\})$
4. **then** $M \leftarrow M \cup \{e\} \setminus \{e' | e' \in M, e'$ and e share an end point$\}$
5. **return** M

Algorithm *Find-Weighted-Matching-Multipass*(G, ϵ)
(∗ Finds Large Weighted Matchings ∗)
1. $\gamma \leftarrow \frac{2\epsilon}{3}$
2. $\kappa \leftarrow \frac{\gamma^3}{(1+\gamma)^2 - \gamma^3}$
3. Find a $\frac{1}{3+2\sqrt{2}}$ weighted matching, M
4. **repeat**
5. $S \leftarrow w(M)$
6. **for** each edge $e \in G$
7. **do if** $w(e) > (1+\gamma)w(\{e' | e' \in M, e'$ and e share an end point$\})$
8. **then** $M \leftarrow M \cup \{e\} \setminus \{e' | e' \in M, e'$ and e share an end point$\}$
9. **until** $\frac{w(M)}{S} \leq 1 + \kappa$
10. **return** M

Fig. 3. Algorithms for Finding Large Weighted Matchings.

Let M_i be the matching constructed after the i-th pass. Let $B_i = M_i \cap M_{i-1}$. Now, $(1+\gamma)(w(M_{i-1}) - w(B_i)) \leq w(M_i) - w(B_i)$ and so,

$$\frac{w(M_i)}{w(M_{i-1})} = \frac{w(M_i)}{w(M_{i-1}) - w(B_i) + w(B_i)} \geq \frac{(1+\gamma)w(M_i)}{w(M_i) + \gamma w(B_i)} \ .$$

If $\frac{w(M_i)}{w(M_{i-1})} < (1+\kappa)$, then we deduce that $w(B_i) \geq \frac{\gamma - \kappa}{\gamma + \gamma\kappa} w(M_i)$. Appealing to Lemma 3, this means that, for all i,

$$OPT \leq (1/\gamma + 3 + 2\gamma)(w(M_i) - w(B_i)) + 2(1+\gamma)w(B_i) \ ,$$

since edges in B_i have empty trails of the dead. So if $w(B_i) \geq \frac{\gamma - \kappa}{\gamma + \gamma\kappa} w(M_i)$ we get that,

$$\begin{aligned} OPT &\leq (1/\gamma + 3 + 2\gamma)(w(M_i) - w(B_i)) + 2(1+\gamma)w(B_i) \\ &\leq (1/\gamma + 3 + 2\gamma - (1/\gamma + 1)\frac{\gamma - \kappa}{\gamma + \gamma\kappa})w(M_i) \\ &\leq (2 + 3\gamma)w(M_i) \ . \end{aligned}$$

Since $\gamma = \frac{2\epsilon}{3}$ the claimed approximation ratio follows.

4 Conclusions and Open Questions

New constant pass streaming algorithms, using $\tilde{O}(n)$ space, have been presented for the MCM and MWM problems. The MCM algorithms uses a novel randomized technique that allows us to find augmenting paths and thereby find a matching of size $\textsc{Opt}/(1+\epsilon)$. The MWM algorithm builds upon previous work to find a matching whose weight is at least $\textsc{Opt}/(2+\epsilon)$.

It is worth asking if there exists a streaming algorithm that achieves a $1/(1+\epsilon)$ approximation for the MWM problem. It is possible, although non-trivial, to extend some of the ideas for the MCM problem to deal with the case when edges are weighted. The main problem lies in the fact that, in the weighted case, there are "augmenting cycles" in addition to (weight-)augmenting paths (naturally defined). Unfortunately, finding cycles in the streaming model seems to be inherently difficult. In particular, this author fears that lower bounds concerning finding the girth of a graph [10] and finding common neighborhoods [17] may suggest a negative result.

Finally, the careful reader may have noticed that the number of passes required by the MCM algorithm has a rather strong dependence on ϵ in the sense that, as ϵ becomes small, the number of passes necessary, grows very quickly. This paper was rather cavalier with the dependence on ϵ as we were primarily concerned with ensuring that the number of passes was independent of n. However, for the sake of practical applications, a weaker dependence would be desirable.

References

1. Edmonds, J.: Maximum matching and a polyhedron with 0,1-vertices. J. Res. Nat. Bur. Standards **69** (1965) 125–130
2. Gabow, H.N.: Data structures for weighted matching and nearest common ancestors with linking. In: Proc. ACM-SIAM Symposium on Discrete Algorithms. (1990) 434–443
3. Hopcroft, J.E., Karp, R.M.: An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. SIAM J. Comput. **2** (1973) 225–231
4. Micali, S., Vazirani, V.: An $O(\sqrt{V}E)$ algorithm for finding maximum matching in general graphs. In: Proc. 21st Annual IEEE Symposium on Foundations of Computer Science. (1980)
5. Drake, D.E., Hougardy, S.: Improved linear time approximation algorithms for weighted matchings. In Arora, S., Jansen, K., Rolim, J.D.P., Sahai, A., eds.: RANDOM-APPROX. Volume 2764 of Lecture Notes in Computer Science., Springer (2003) 14–23
6. Kalantari, B., Shokoufandeh, A.: Approximation schemes for maximum cardinality matching. Technical Report LCSR–TR–248, Laboratory for Computer Science Research, Department of Computer Science. Rutgers University (1995)
7. Preis, R.: Linear time 1/2-approximation algorithm for maximum weighted matching in general graphs. In Meinel, C., Tison, S., eds.: STACS. Volume 1563 of Lecture Notes in Computer Science., Springer (1999) 259–269
8. Muthukrishnan, S.: Data streams: Algorithms and applications. (2003) Available at http://athos.rutgers.edu/∼muthu/stream-1-1.ps.

9. Feigenbaum, J., Kannan, S., McGregor, A., Suri, S., Zhang, J.: On graph problems in a semi-streaming model. In: Proc. 31st International Colloquium on Automata, Languages and Programming. (2004) 531–543
10. Feigenbaum, J., Kannan, S., McGregor, A., Suri, S., Zhang, J.: Graph distances in the streaming model: The value of space. Proc. 16th ACM-SIAM Symposium on Discrete Algorithms (2005)
11. Munro, J., Paterson, M.: Selection and sorting with limited storage. Theoretical Computer Science **12** (1980) 315–323
12. Henzinger, M.R., Raghavan, P., Rajagopalan, S.: Computing on data streams. Technical Report 1998-001, DEC Systems Research Center (1998)
13. Alon, N., Matias, Y., Szegedy, M.: The space complexity of approximating the frequency moments. Journal of Computer and System Sciences **58** (1999) 137–147
14. Feigenbaum, J., Kannan, S., Strauss, M., Viswanathan, M.: An approximate L^1 difference algorithm for massive data streams. SIAM Journal on Computing **32** (2002) 131–151
15. Guha, S., Mishra, N., Motwani, R., O'Callaghan, L.: Clustering data streams. In: Proc. 41th IEEE Symposium on Foundations of Computer Science. (2000) 359–366
16. Drineas, P., Kannan, R.: Pass efficient algorithms for approximating large matrices. In: Proc. 14th ACM-SIAM Symposium on Discrete Algorithms. (2003) 223–232
17. Buchsbaum, A.L., Giancarlo, R., Westbrook, J.: On finding common neighborhoods in massive graphs. Theor. Comput. Sci. **1-3** (2003) 707–718