Matching in Dynamic Graphs (2011; Baswana, Gupta, Sen)

Surender Baswana, IIT Kanpur, www.cse.iitk.ac.in/∼sbaswana
Manoj Gupta, IIT Delhi, www.cse.iitd.ernet.in/∼gmanoj
Sandeep Sen, IIT Delhi, www.cse.iitd.ernet.in/∼ssen

SYNONYMS and INDEX TERMS: Matching, maximum matching, maximal matching, dynamic graph, randomized algorithms.

1 PROBLEM DEFINITION

Let $G = (V, E)$ be an undirected graph on $n = |V|$ vertices and $m = |E|$ edges. A matching in G is a set of edges $\mathcal{M} \subseteq E$ such that no two edges in $\mathcal{M}$ share any vertex. Matching has been one of the most well-studied problems in algorithmic graph theory for decades [4]. A matching $\mathcal{M}$ is called *maximum matching* if the number of edges in $\mathcal{M}$ is maximum. The fastest known algorithm for maximum matching, due to Micali and Vazirani [5], runs in $O(m\sqrt{n})$. A matching is said to be *maximal* if it is not strictly contained in any other matching. It is well known that a maximal matching achieves a factor 2 approximation of the maximum matching.

2 Key Result

We address the problem of maintaining maximal matching in a fully dynamic environment - allowing updates in form of both insertion and deletion of edges. Ivković and Llyod [3] designed the first fully dynamic algorithm for maximal matching with $O((n + m)^{0.7072})$ update time. In this article, we present a fully dynamic algorithm for maximal matching that achieves $O(\log n)$ expected amortized time per update.

3 Ideas underlying the algorithm

We begin with some terminologies and notations that will facilitate our description and also provide some intuition behind our approach. Let $\mathcal{M}$ denote a matching in the given graph at any instant - an edge $(u, v) \in \mathcal{M}$ is called a *matched* edge where u is referred to as a *mate* of v and vice versa. An edge in $E\backslash\mathcal{M}$ is an *unmatched* edge. A vertex x is *matched* if there exists an edge $(x, y) \in \mathcal{M}$; otherwise it is *free* or *unmatched*.

In order to maintain a maximal matching, it suffices to ensure that there is no edge (u, v) in the graph such that both u and v are free with respect to the matching $\mathcal{M}$. Therefore, a natural technique for maintaining a maximal matching is to keep track of each vertex if it is matched or free. When an edge (u, v) is inserted, we add (u, v) to the matching if u and v are free. For the case when an unmatched edge (u, v) is deleted, no action is required. Otherwise, for both u and v we search their neighborhoods for any free vertex and update the matching accordingly. It follows that each update takes $O(1)$ computation time except when it involves deletion of a matched edge; in this case the computation time is of the order of the sum of the degrees of the two endpoints of the deleted edge. So this trivial algorithm is quite efficient for *small* degree vertices, but could be expensive for *large* degree vertices. An alternate approach could be to match a free

vertex u with a randomly chosen neighbor, say v. Following the standard adversarial model, it can be observed that an expected $\deg(u)/2$ edges incident to u will be deleted before deleting the matched edge (u, v). So the expected amortized cost per edge deletion for u is roughly $O\left(\frac{\deg(u)+\deg(v)}{\deg(u)/2}\right)$. If $\deg(v) < \deg(u)$, this cost is $O(1)$. But if $\deg(v) \gg \deg(u)$, then it can be as bad as the trivial algorithm. To circumvent this problem, we introduce an important notion, called *ownership* of edges. Intuitively, we assign an edge to that endpoint which has *higher* degree.

The idea of choosing a random mate and the trivial algorithm described above can be combined together to design a simple algorithm for maximal matching. This algorithm maintains a partition of the vertices into two levels. Level 0 consists of vertices which own *fewer* edges and we handle the updates there using the trivial algorithm. Level 1 consists of vertices (and their mates) which own *larger* number of edges and we use the idea of random mate to handle their updates. This 2-LEVEL algorithm achieves $O(\sqrt{n})$ expected amortized time per update. A careful analysis of the 2-LEVEL algorithm suggests that a *finer* partition of vertices could help in achieving a faster update time. This leads to our $\log_2 n$-LEVEL algorithm that achieves expected amortized $O(\log n)$ time per update.

Our algorithm uses randomization very crucially in order to handle the updates efficiently. The matching maintained (based on the random bits) by the algorithm at any stage is not known to the adversary for it to choose the updates adaptively. This oblivious adversarial model is no different from randomized data-structures like universal hashing.

4 The 2-LEVEL Algorithm

The algorithm maintains a partition of the set of vertices into two levels. Each edge present in the graph will be owned by one or both of its endpoints as follows. If both the endpoints of an edge are at level 0, then it is owned by both of them. Otherwise it will be owned by exactly that endpoint which lies at a higher level. If both the endpoints are at level 1, the tie will be broken suitably by the algorithm. Let $\mathcal{O}_u$ denote the set of edges owned by a vertex u at any instant of the algorithm. With a slight abuse of the notation, we will also use $\mathcal{O}_u$ to denote $\{v|(u, v) \in \mathcal{O}_u\}$. As the algorithm proceeds, the vertices will make transition from one level to another and the ownership of edges will also change accordingly.

The algorithm maintains the following three invariants after each update.

1. Every vertex at level 1 is matched. Every free vertex at level 0 has all its neighbors matched.
2. Every vertex at level 0 owns less than $\sqrt{n}$ edges at any stage.
3. Both endpoints of every matched edge are at the same level.

It follows from the first invariant that the matching $\mathcal{M}$ is maximal at each stage. The second and third invariant help in incorporating the two ideas of our algorithm efficiently.

Handling insertion of an edge

Let (u, v) be the edge being inserted. If either u or v are at level 1, there is no violation of any invariant. However, if both u and v are at level 0, then we we proceed as follows. Both u and v become the owner of the edge (u, v). If u and v are free, then we add (u, v) to $\mathcal{M}$. Notice that the insertion of (u, v) also leads to increase of $|\mathcal{O}_u|$ and $|\mathcal{O}_v|$ by one, and so may lead to violation of Invariant 2. We process the vertex that owns more edges; let u be that vertex. If $|\mathcal{O}_u| = \sqrt{n}$, then Invariant 2 has got violated. In order to restore it, u moves to level 1 and gets matched to some vertex, say y, selected uniformly at random from $\mathcal{O}_u$. Vertex y also moves to level 1 to satisfy Invariant 3. If w and x were respectively the earlier mates of u and y at level 0, then the matching of u with y has rendered w and x free. Both w and x search for free neighbors at level 0 and update the matching accordingly. It is easy to observe that in all these cases, it takes $O(\sqrt{n})$ time to handle an edge insertion.

Handling deletion of an edge

Let (u, v) be an edge that is deleted. If $(u, v) \notin \mathcal{M}$, all the invariants are still valid. Let us consider the more important case of $(u, v) \in \mathcal{M}$ - the deletion of (u, v) has caused u and v to become free. Therefore, the first invariant might have got violated for u and v. If edge (u, v) was at level 0, then both u and v search for a free neighbor and update the matching accordingly. This takes $O(\sqrt{n})$ time. If edge (u, v) was at level 1, then u (similarly v) is processed as follows.

First, u disowns all its edges whose other endpoint is at level 1. If $|\mathcal{O}_u|$ is still greater than or equal to $\sqrt{n}$, then u stays at level 1 and selects a random mate from $\mathcal{O}_u$. However, if $|\mathcal{O}_u|$ has fallen below $\sqrt{n}$, then u moves to level 0 and gets matched to a free neighbor (if any). For each neighbor of u at level 0, the transition of u from level 1 to 0 is, effectively, like insertion of a new edge. This transition leads to an increase in the number of owned edges by each neighbor of u at level 0. As a result the second invariant for each such neighbor at level 0 may get violated if the number of edges it owns now becomes $\sqrt{n}$. To take care of these scenarios, we proceed as follows. We scan each neighbors of u at level 0 and for each neighbor w, with $|\mathcal{O}_w| = \sqrt{n}$, a mate is selected randomly from $\mathcal{O}_w$ and w is moved to level 1 along with its mate. This concludes the deletion procedure of edge (u, v).

4.1 Analysis of the algorithm

It may be noted that, unlike insertion, the deletion of an edge could potentially lead to moving of many vertices from level 0 to 1 and this may involve significant computation. However, we will show that the expected amortized computation per update will is $O(\sqrt{n})$.

We analyze the algorithm using the concept of *epochs*.

Definition 4.1 *At any time t, let (u, v) be any edge in $\mathcal{M}$. Then the* **epoch** *of (u, v) at time t is the maximal time interval containing t during which $(u, v) \in \mathcal{M}$.*

The entire life span of an edge (u, v) can be viewed as a sequence of epochs when it is matched separated by periods when it is unmatched. Any edge update that does not change the matching is processed in O(1) time. An edge update that changes the matching results in the start of new epoch(s) or the termination of some existing epoch(s). And it is only during the creation or termination of an epoch that significant computation is involved. For the purpose of analyzing the update time, (when matching is affected), we assign the computation performed to the corresponding epochs created or terminated. It is easy to see that the computation associated with an epoch at level 0 is $O(\sqrt{n})$. The computation associated with an epoch at level 1 is of the order of sum of the degrees of the endpoints of the corresponding matched edge which may be $\Omega(n)$. When a vertex moves from level 0 to 1, although it owns $\sqrt{n}$ edges, this may grow later to $O(n)$. So the computation associated with an epoch at level 1 can be quite high. We will show that the expected number of such epochs that get terminated during any arbitrary sequence of edge updates will be relatively small. The following lemma plays a key role.

Lemma 4.1 *The deletion of an edge (u, v) at level 1 terminates an epoch with probability $\leq 1/\sqrt{n}$.*

Proof: The deletion of edge (u, v) will lead to termination of an epoch only if $(u, v) \in \mathcal{M}$. If edge (u, v) was owned by u at the time of its deletion, note that u owned at least $\sqrt{n}$ edges at the moment of start of its epoch. Since u selected its matched edge uniformly at random from these edges, the (conditional) probability is $\frac{1}{\sqrt{n}}$. The same argument applies if v was the owner, so (u, v) is a matched edge at the time of deletion of (u, v) with probability at most $1/\sqrt{n}$. □

Consider any sequence of m edge updates. We analyze the computation associated with all the epochs that get terminated during these m updates. It follows from Lemma 4.1 and the linearity of expectation that the expected number of epochs terminated at level 1 will be $m/\sqrt{n}$. As discussed above, computation

associated with each epoch at level 1 is $O(n)$. So the expected computation associated with the termination of all epochs at level 1 is $O(m\sqrt{n})$. The number of epochs destroyed at level 0 is trivially bounded by $O(m)$. Each epoch at level 0 has $O(\sqrt{n})$ computation associated with it, so the total computation associated with these epochs is also $O(m\sqrt{n})$. We conclude the following.

Theorem 4.1 *Starting with a graph on n vertices and no edges, we can maintain maximal matching for any sequence of m updates in expected $O(m\sqrt{n})$ time.*

4.2 The $\log_2 n$-LEVEL Algorithm

The key idea for improving the update time lies in the second invariant of our 2-LEVEL algorithm. Let $\alpha(n)$ be the threshold for the maximum number of edges that a vertex at level 0 can own. Consider an epoch at level 1 associated with some edge, say (u, v). The computation associated with this epoch is of the order of the number of edges u and v own which can be $\Theta(n)$ in the worst case. However, the expected duration of the epoch is of the order of the minimum number of edges u can own at the time of its creation, i.e., $\Theta(\alpha(n))$. Therefore, the expected amortized computation per edge deletion at level 1 is $O(n/\alpha(n))$. Balancing this with the $\alpha(n)$ update time at level 0, yields $\alpha(n) = \sqrt{n}$.

In order to improve the running time of our algorithm, we need to decrease the ratio between the maximum and the minimum number of edges a vertex can own during an epoch at any level. It is this ratio that determines the expected amortized time per edge deletion. This observation leads us to a finer partitioning of the ownership classes. When a vertex creates an epoch at level i, it owns at least 2^i edges, and during the epoch it is allowed to own at most $2^{i+1} - 1$ edges. As soon as it owns 2^{i+1} edges, it migrates to a higher level. Notice that the ratio of maximum to minimum edges owned by a vertex during an epoch gets reduced from $\sqrt{n}$ to a constant leading to about $\log_2 n$ levels. Though the $\log_2 n$-LEVEL algorithm can be seen as a natural generalization of our 2-LEVEL algorithm, there are many intricacies that makes the algorithm and its analysis quite involved. For example, a single edge update may lead to a sequence of falls and rise of many vertices across the levels of the data structure. Moreover, there may be several vertices trying to fall or rise at any time while processing an update. Taking a top down approach in processing these vertices simplifies the description of the algorithm. The analysis of the algorithm becomes easier when we analyze each level separately. This analysis at any level is quite similar to the analysis of LEVEL-1 in our 2-LEVEL algorithm. We recommend the interested reader to refer to the journal version of this paper in order to fully comprehend the algorithm and its analysis. The final result result achieved by our $\log_2 n$-LEVEL algorithm is stated below.

Theorem 4.2 *Starting with a graph on n vertices and no edges, we can maintain maximal matching for any sequence of m updates in expected $O(m \log n)$ time.*

Using standard probability tools, it can be shown that the bound on the update time as stated in Theorem 4.2 holds with high probability, as well as with limited independence.

5 OPEN PROBLEMS

There have been new results on maintaining approximate weighted matching [2] and $(1 + \epsilon)$-approximate matching [1, 6] for $\epsilon < 1$. The interested reader should study these results. For any $\epsilon < 1$, whether it is possible to maintain $(1+\epsilon)$-approximate matching in poly-logarithmic update time is still an open problem.

6 EXPERIMENTAL RESULTS

None is reported.

7 DATA SETS

None is reported.

8 URL to CODE

None is reported.

9 CROSS REFERENCES

None is reported. Entry editors please feel free to add some.

10 RECOMMENDED READING

[1] A. Anand, S. Baswana, M. Gupta, and S. Sen. Maintaining approximate maximum weighted matching in fully dynamic graphs. In *FSTTCS*, pages 257–266, 2012.

[2] M. Gupta and R. Peng. Fully dynamic (1+ e)-approximate matchings. In *FOCS*, pages 548–557, 2013.

[3] Z. Ivkovic and E. L. Lloyd. Fully dynamic maintenance of vertex cover. In *WG '93: Proceedings of the 19th International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 99–111, London, UK, 1994. Springer-Verlag.

[4] L. Lovasz and M. Plummer. *Matching Theory*. AMS Chelsea Publishing, North-Holland, AmsterdamNew York, 1986.

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

[6] O. Neiman and S. Solomon. Simple deterministic algorithms for fully dynamic maximal matching. In *STOC*, pages 745–754, 2013.