CS 267 Lecture 12 Dynamic graph matching
Scribe: Jack Chen **Date:** February 19, 2014

1 Graph matching

A matching is a set of edges that share no endpoints. Known algorithms for maximum matching include:

- Hopcroft, Karp '73: maximum matching for bipartite graphs in $O(m\sqrt{n})$ time.
- Micali, Vazirani '80: same time for general graphs.
- Madry '13: $\tilde{O}(m^{10/7})$ time for general graphs.

Usually, we compute maximum matchings by find augmenting paths.

Definition 1.1. *An* ***augmenting path*** *is a path that starts and ends at a free vertex (a vertex that has no edge in the matching) and alternates between non-matching edges and matching edges.*

If we augment along an augmenting path by removing each matching edge from the matching and adding each non-matching edge to the matching, then we get a new matching that has one more edge.

If you take any matching M and a maximum matching M^*, then $E(M) \cup E(M^*)$ is a collection of augmenting paths and alternating cycles (cycles that alternate between matching and non-matching edges). And $|M^*| - |M|$ is the number of augmenting paths (because each augmenting path has exactly one more edge in M^* than in M).

Definition 1.2. *The* ***length*** *of an augmenting path is the number of non-matching edges in the path.*

If G does not contain augmenting paths of length $< k$, then the number of augmenting paths in $E(M) \cup E(M^*)$ is at most $|M^*|/k$. So $|M| = |M^*| - \#$ augmenting paths $\geq |M^*|(1 - 1/k)$

These algorithms (Hopcraft-Karp, Micali-Vazirani) take the shortest augmenting paths, augment along them, and then guarantee that the length of the shortest augmenting path increases. So if you stop after k rounds, you get a $1/(1-1/k)$-approximate matching[1]. This means that these algorithms can also be viewed as approximation algorithms: they run in $O(m/\varepsilon)$ time for a $(1+\varepsilon)$-approximation. So if you want a constant-factor approximation, you can get it in essentially linear time.

Definition 1.3. *A* ***maximal matching*** *is a matching M that has no $(x, y) \in E$ such that x and y are free (no edge in M touches either of them).*

In other words, M is maximal iff for all $(x, y) \in E$, at least one of the endpoints is not free. This definition is equivalent to the statement that there are no augmenting paths of length 1. So any maximal matching is a 2-approximation maximum matching.

2 Dynamic maximum matching

Now, we'll consider how to maintain information about a maximum matching while making updates to the graph (inserting or deleting edges). We wish to be able to answer queries for $|M|$ and for the vertex mate(v) any given $v \in V$ is matched with.

The known algorithms are inefficient. Sankowski gives updates in $O(n^{1.495})$ time. To see how good this is, let's compare to the trivial update. Say we want to delete an edge (u, v). This can create at most one

[1] For a maximization problem, a c-approximation with $c > 1$ means that the solution is at least a $1/c$-fraction of the optimum solution.

augmenting path. So find an augmenting path going from u to v, which takes $O(m)$ time which is $O(n^2)$ time. Sankowski shows that you can improve this slightly to $O(n^{1.495})$ time, using matrix multiplication.

Since the asymptotically fastest known algorithms for matrix multiplication are impractical, we might ask whether we can get fast dynamic maximum matching algorithms that aren't based on matrix multiplication. It turns out that we essentially can't: A.V.'14 showed that any dynamic algorithm for maximum matching with update time $O(n^{2-\varepsilon})$ implies an $O(n^{3-\varepsilon/3})$ boolean matrix multiplication algorithm.

Since the known algorithms for exact dynamic maximum matching are bad, let's consider approximation algorithms, since we know that in the static case, one can get very efficient approximate matching algorithms. We'd like to ask whether you can get $O(n^{1-\varepsilon})$ update time if you allow approximation.

Baswana et al. showed that a maximal matching can be maintained with $O(\log n)$ update time. But there's a catch: the algorithm is randomized, and it only guarantees to be fast if your queries don't know the random bits that your algorithm is using. If the queries can see the random bits of the algorithm, they can make it very bad – no guarantees on the runtime. (This is fast under an oblivious adversary – the adversary can see the code, but not the random seed.)

The state of deterministic algorithms is much worse, however. Neiman and Solomon '13 gave an algorithm with $O(\sqrt{m})$ update time for maximal matching (and hence for a 2-approximation), which we'll show in this lecture, also achieving a 3/2-approximation (with a slight modification). Gupta, Peng '13 gave an algorithm with $O(\sqrt{m}/\varepsilon^2)$ update time for $(1+\varepsilon)$-approx.

The basic idea of Neiman, Solomon '13 is this: when we insert an edge, if both endpoints are free, just add the edge to the matching. If neither is free, do nothing other than add the new edge to the edge set. If one is free and the other is not, maybe look in the neighborhoods of the vertices to change the matching. Why? A simple way to look for a vertex to match with is to look through neighbors. Since we want that not to take very long, we'd like to maintain an invariant that any free vertex has small degree. So if we're doing an insertion between a free vertex and a matched vertex, don't do anything if the free vertex still has small degree. If it has large degree, then do something to match it. If you delete an edge that is in the matching, both endpoints become free, and if they are high degree, you need to find some vertex to swap them with.

Now, here's the formal description of the algorithm.

Let round i be when update i is called, and let m_i be the number of edges in round i.

We maintain a maximal matching M, and maintain three invariants:

1. Any free vertex has degree $\leq \sqrt{2(m_i+n)}$
2. Any free vertex created in round i has degree $\leq \sqrt{2m_i}$
3. M is maximal.

To insert an edge, we run Algorithm **??**.

Algorithm 1: insert(u, v)

```
Add $(u,v)$ to $E$;
$m_i \leftarrow m_{i-1} + 1$;
If $u$ and $v$ are matched, do nothing;
If $u$ and $v$ are free, add $(u,v)$ to $M$;
If $u$ is free and $v$ is matched (the other case is symmetric), surrogate($u$);
```

If u is free and v is matched, then invariant (1) or (2) may no longer hold. We fix it with a procedure surrogate(u), shown in Algorithm **??**. The idea is to look through the neighbors of u: it has low degree so we can afford to do this. We know that none of these neighbors are free, since M is a maximal matching. So each neighbor w of u is matched with a w'. If w' has small degree, then we'll match u with w and unmatch w with w'.

Lemma 2.1. *Let $u \in V$ such that $deg(u) > \sqrt{2m}$ and all neighbors of u are matched. Then there exists w' such that $w \in N(u), (w,w') \in M$, and $deg(w') \leq \sqrt{2m}$.*

Algorithm 2: surrogate(u)

```
foreach $w \in N(u)$ do
    $w' = mate(w)$;
    if $deg(w') \leq \sqrt{2m_i}$ then
        $M \leftarrow M \setminus \{(w,w')\} \cup \{(u,w)\}$;
        if $w'$ has free neighbor $x$ then
            $M \leftarrow M \cup \{(w',x)\}$;
```

Proof. $N(u) = \{w_1, \dots, w_d\}$. Let $w_i' = \text{mate}(w_i)$. If for all i, $deg(w_i) > \sqrt{2m}$, then $|E| > d\sqrt{2m}/2 \geq 2m/2 = m$, which is a contradiction. □

Algorithm 3: delete(u, v)

```
Remove $u, v$ from $E$, $m_i = m_{i-1} - 1$;
if $(u,v) \in M$ then
    foreach $w \in \{u,v\}$ do
        if $w$ has free neighbor $x$ then
            Add $(w,x)$ to $M$;
        else if $deg(w) > \sqrt{2m}$ then
            surrogate($w$);
    $x \leftarrow maxDegreeFreeVertex()$;
    if $deg(x) > \sqrt{2(m_i+n)}$ then
        surrogate($x$);
```

Deletion is similar, as shown in Algorithm **??**. Without the part at the end to fix the maximum degree free vertex, this satisfies invariants 3 and 2, but invariant 1 may not be satisfied because m decreased. To fix invariant 1, we'll find the max degree free vertex x, and if $deg(x) > \sqrt{2(m_i+n)}$, then call surrogate(x). We have to show why this will work, and how to efficiently find the max degree free vertex.

To implement this, we'll maintain F_{max}, a max heap storing free vertices by degree ($O(\log n)$ inserts, deletes, find max).

We also need to maintain a data structure $F(v)$ for each vertex v that can answer whether v has free neighbors and to return a free neighbor of v. Whenever a vertex x is added to the matching, all of the data structures $F(y)$ of its neighbors y need to be notified, and they need to be updated. However, when x first becomes matched, it has at most $O(\sqrt{m})$ neighbors, so that provided that $F(y)$ can be updated efficiently, the overhead of maintaining these is negligible. If a vertex x is no longer matched, its neighbors are searched anyway by the algorithm to look for a new mate. If no new mate is found, the status of x needs to be changed in all the data structures $F(y)$ of its neighbors, but this can again be amortized to the mate search.

***

For each node x, keep an array of length n. Entry i is 1 if vertex i is free and a neighbor, 0 otherwise. Partition into $\sqrt{n}$ blocks of size $\sqrt{n}$. For each block b, also maintain $c(b) = \#$ of 1s in block b. This can be updated efficiently: if we add a free neighbor y, set the entry for y to 1, and increment $c(b)$ for the block b that contains that. And we can check whether there is a free neighbor and if so find one in $O(\sqrt{n})$ time. To check whether x has a free neighbor, check whether some $c(b) > 0$. Then look through the $\sqrt{n}$ entries in b to find one.

Claim 1. *If we fix the highest degree free node after each deletion, then invariant (1) will be satisfied.*

Proof. Assume for contradiction that node u has $deg(u) > \sqrt{2(m_t+n)}$ in round t. Look at the rounds before t. There must exist a round k such that $deg(u) \leq \sqrt{2m_k}$ and for all $l \in [k+1, t]$, $deg(u) > \sqrt{2m_l}$.

If $deg(u)$ changed in $[k+1,t]$ then its degree would have been ok, since the algorithm ensures this for the nodes that are touched. So $deg(u)$ is unchanged. Then $\sqrt{2m_l} < \sqrt{2m_k} \implies m_l < m_k$ for all $l \in [k+1,t]$. So $\sqrt{2(m_t+n)} < \sqrt{2m_k} \implies m_k - m_t > n \implies n$ deletions happened. By invariant (2), any free node created in $(k+1,t)$ has degree $\leq \sqrt{2m_l} < deg(u)$. The number of free nodes in rounds before round $k+1$ is less than n, so u was max degree free node in some deletion. $\square$

In order to modify this to get the (3/2)-approximation, we need to check for length 2 augmenting paths. This can also be done in $O(\sqrt{n})$.