Lookahead can help in maximal matching

Kitti Gelle, Szabolcs Iván

Abstract: In this paper we study a problems in the context of fully dynamic graph algorithms that is, when we have to handle updates (insertions and removals of edges), and answer queries regarding the current graph, preferably in a better time bound than running a classical algorithm from scratch each time a query arrives.

We show that a maximal matching can be maintained in an (undirected) graph with a deterministic amortized update cost of $O(\log m)$ (where m is the all-time maximum number of the edges), provided that a lookahead of length m is available, i.e. we can "peek" the next m update operations in advance.

Introduction and notation

In the past two decades, there has been a growing interest in developing a framework of algorithm design for *dynamic graphs*, that is, graphs which are subject to *updates* – in our case, additions and removals of an edge at a time. The aim of a so-called fully dynamic algorithm (here "fully" means that both addition and removal are permitted) is to maintain the result of the algorithm after each and every update of the graph, in a time bound significantly better than recomputing it from scratch each time.

Also in [2], a systematic investigation of dynamic graph problems in the presence of a so-called *lookahead* was initiated: although the stream of update operations can be arbitrarily large and possibly builds up during the computation time, in actual real-time systems it indeed possible to have some form of *lookahead* available. That is, the algorithm is provided with some prefix of the update sequence of some length (for example, in [2] an assembly planning problem is studied in which the algorithm can access the prefix of the sequence of future operations to be handled of length $\Theta(\sqrt{m/n}\log n)$), where m and n are the number of edges and nodes, respectively. Similarly to the results of [2] (where the authors devised dynamic algorithms using lookahead for the problems of strongly connectedness and transitive closure), we will execute the tasks in batches: by looking ahead at $t = O(m)$ future update operations, we treat them as a single batch, preprocess our current graph based on the information we get from the complete batch, then we run all the updates, one at a time, on the appropriately preprocessed graph. This way, we achieve an amortized update cost of $O(\log m)$ for maintaining a maximal matching.

In this paper, a graph G is viewed as a set (or list) of edges, with $\|G\|$ standing for its cardinality. This way notions like $G \cup H$ for two graphs G and H (sharing the common set $V(G) = V(H)$ of vertices) are well-defined.

Maximal matching with lookahead

In this section we present an algorithm that maintains a maximal matching in a dynamic graph G with constant query and $O(\log m)$ update time (note that $O(\log m)$ is also $O(\log n)$ as $m = O(n^2)$), provided that a *lookahead* of length m is available in the sequence of (update and query) operations. This is an improvement over the currently best-known deterministic algorithm [3] that that has an update cost of $O(\sqrt{m})$, without lookahead, and achieves the same amortized update cost as the best-known randomized algorithm [1].

In this problem, a *matching* of a(n undirected) graph G is a subset $M \subseteq G$ of edges having pairwise disjoint sets of endpoints. A matching M is *maximal* if there is no matching $M' \supsetneq M$ of G. Given a matching M, for each vertex v of G let $\textsc{mate}(v)$ denote the unique vertex u such that $(u, v) \in M$ if such a vertex exists, otherwise $\textsc{mate}(v) = \textsc{null}$.

In the fully dynamic version of the maximal matching problem, the update operations are edge additions $+(u, v)$, edge deletions $-(u, v)$ and the queries have the form $\textsc{mate}(u)$.

The following is clear:

Proposition 1. *Suppose G is a graph in which M is a maximal matching. Then a maximal matching in the graph $G + (u, v)$ is $M \cup \{(u, v)\}$, if $\textsc{mate}(u) = \textsc{mate}(v) = \textsc{null}$, and M otherwise.*

This proposition gives the base algorithm GREEDY for computing a maximal matching in a graph:

```
Let $M$ be an empty list of edges;
for( $(u,v) \in G$ )
  if( MATE($u$) == NULL and MATE($v$) == NULL)
    MATE($u$) := $v$; MATE($v$) := $u$;
    insert $(u,v)$ to $M$;
return $M$;
```

Note that if one initializes the MATE array in the above code so that it contains some non-NULL entries, then the result of the algorithm represents a maximal matching within the subgraph of G spanned by the vertices having NULL MATEs initially. Also, with M represented by a linked list, the above algorithm runs in $O(m)$ total time using no lookahead. Hence, by calling this algorithm on each update operation (after inserting or removing the edge in question), we get a dynamic graph algorithm with no lookahead (so it uses a lookahead of at most m operations), a constant query cost (as it stores the MATE array explicitly) and an $O(m)$ update cost. Using this algorithm A_1, we build up a sequence A_k of algorithms, each having a smaller update cost than the previous ones. (In a practical implementation there would be a single algorithm A taking k as a parameter with the graph G and the update sequence, but for proving the time complexity it is more convenient to denote the algorithms in question by A_1, A_2, and so on.)

In our algorithm descriptions the input is the current graph G (which is $\emptyset$ in the first time we start running the program) and a sequence $(q_1, \ldots, q_t)$ of operations. Of course as the sequence can be arbitrarily long, we do not require an explicit representation, only the access of the first m elements (that is, we have a lookahead of length m).

Lemma 3. *Assume A_k is a fully dynamic algorithm for maintaining a maximal matching with an $f(k) \cdot m^{1/k}$ amortized update cost, constant query cost using a lookahead of length m.*

Then there is a universal constant c such that there exists a fully dynamic algorithm A_{k+1} that also maintains a maximal matching with $(f(k) + c(1 + \log m))m^{1/k+1}$ amortized update cost, and a constant query cost using a lookahead of length m.

Now, as A_1 is an algorithm satisfying the conditions of this lemma with $k = 1$ and $f(k) = c_0$ for some constant c_0, it implies that for each $k > 1$ that there is a fully dynamic algorithm that maintains a maximal matching with an amortized update cost of $(c_0 + kc(1 + \log m))m^{1/k} = O(k \log m \cdot m^{1/k})$. Setting $k = \log m$ we get that $A_{\log m}$ has an amortized update cost of $O(\log^2 m \cdot m^{1/\log m}) = O\left(\log^2 m \cdot \left(2^{\log m}\right)^{1/\log m}\right) = O(\log^2 m \cdot 2) = O(\log^2 m)$.

Hence we get:

Theorem 1. *There exists a fully dynamic algorithm for maintaining a maximal matching with an $O(\log^2 m)$ amortized update cost and constant query cost, using a lookahead of length m.*

Now we prove Lemma 3 by defining the algorithm A_{k+1} below.

- The algorithm A_{k+1} works in *phases* and returns a graph G (as an edge set) and a matching M (as an edge list).
- The algorithm accesses the *global* MATE array in which the current maximal matching of the whole graph is stored. (A_{k+1} might get only a subgraph of the whole actual graph as input.)
- In one phase, A_{k+1} either handles a block $\vec{q} = (q_1, \ldots, q_t)$ of $t = m^{\frac{k}{k+1}}$ operations or a single operation.
- Let G and M respectively denote the current graph and matching we have in the beginning of a phase.
- If $\|G\|$ is smaller than our favorite constant 42, then the phase handles only the next operation by explicitly modifying G, afterwards recomputing a maximal matching from scratch, in $O(42)$ (constant) time. That is,
  1 We iterate through all the edges $(u, v) \in M$, and set MATE$[u]$ and MATE$[v]$ to NULL (in effect, we remove the "local part" M of the global matching);
  2 We apply the next update operation on G;
  3 We set $M :=$ GREEDY$(G,$ MATE$)$.

Otherwise the phase handles t operations as follows:

1 Using lookahead (observe that $t < m$) we collect all the edges involved in $\vec{q}$ (either by a $+(u, v)$ or a $-(u, v)$ update operation) into a graph G'.

2 We construct the graph $G'' = G - G'$.

3 We iterate through all the edges $(u, v) \in M$, and set $\text{MATE}[u] := \text{NULL}$, $\text{MATE}[v] := \text{NULL}$.

4 We run $M := \text{GREEDY}(G'', \text{MATE})$.

5 We call $A_k(G \cap G', (q_1, \ldots, q_t))$. Let G^* and M^* be the graph and matching returned by A_k.

6 We set $G := G'' \cup G^*$ and $M := M \cup M^*$.

We will now show its correctness. That is, we claim that each A_k maintains a maximal matching among those vertices having a NULL MATE when the algorithm is called. This is true for the greedy algorithm A_1. Now assuming A_k satisfies our claim, let us check A_{k+1}. When the graph is small, then the algorithm throws away its locally stored matching M, resetting the MATE array to its original value in the process (in fact, this is the only reason why we store the local matching at each recursion level: the global matching state can be queried by accessing the MATE array alone). Then we handle the update and run GREEDY, which is known to compute a maximal matching on the subgraph of G spanned by the vertices having a NULL mate. So this case is clear.

For the second case, if a block of t operations is handled, then we split the graph into two, namely a difference graph G' and an intersection graph G''. By construction, when handling the block, the edges belonging to G' do not get touched. Hence, at any time point, a maximal matching of G can be computed by starting from a maximal matching of G' and then extending the matching by a maximal matching in the subgraph of G'' not covered by the matching of G'. Thus, if we compute a maximal matching M' in the subgraph of G' spanned by the vertices having a NULL MATE, updating the MATE array accordingly (that is, calling GREEDY on G'), and maintaining a maximal matching M'' over the vertices of G'' having a NULL MATE after that point (which is done by A_k, by the induction hypothesis), we get that at any time $M' \cup M''$ is a maximal matching of G. Hence, the algorithm is correct.

Now we analyse the time complexity of A_{k+1}. When a phase handles t operations, then Step 1 can be executed in $O(t \log t) = O(m \log m)$ time (if we use a self-balancing tree representation for storing our graphs, say an AVL tree). Then in Step 2, we construct the difference of the two sets of size $O(m)$ in $O(m \log m)$ time. Step 3 requires an additional time of $O(m)$,since the matching is of size $O(m)$ and it is stored as a list of edges. For Step 4, as $|G''| \leq \|G\| = m$, also an $O(m)$ time is required, and for Step 5, computing the intersection $G \cap G'$ requires a time of $O(m \log m)$, and A_k, being run on a dynamic graph having at most $t = m^{\frac{k}{k+1}}$ edges during its whole lifecycle of t operations needs $t \cdot f(k) \cdot t^{1/k} = m^{\frac{k}{k+1}} \cdot f(k) \cdot m^{\frac{1}{k+1}} = f(k) \cdot m$ computation steps. Gluing together the graphs and the matchings in Step 6 needs a time of $O(m \log m) + O(m)$. Hence the total cost of Steps 1-6 handling a whole phase is $O(m) + O(m \log m) + O(m) + f(k) \cdot m + O(m \log m) + O(m) = (f(k) + c(1 + \log m))m$ for some universal constant c, and since a phase consists of $m^{\frac{k}{k+1}}$ operations, the amortized cost of a single operation becomes $(f(k) + c(1 + \log m))m^{\frac{1}{k+1}}$ and Lemma 3 is proved.

The careful reader may observe that a major part of the time bound comes from the set operations. If an initialization cost of $O(n^2 \log n)$ is affordable (i.e. if there are $\Omega(n^2 \log n)$ operations in total), then we can do better:

- Each algorithm A_k has an adjacency matrix as well, initialized to an all-zero matrix in the very beginning (this initialization takes the aforementioned $O(n^2 \log n)$ setup cost).
- In Step 1, edges of G' are stored into this matrix (taking still $O(m)$ time).
- Now the graphs $G - G'$ and $G \cap G'$, as lists of edges, can be constructed in $O(m)$ time (since lookup in G' now takes constant time instead of the previous $O(\log m)$).
- Since G'' is represented as an edge list, GREEDY still takes $O(m)$ time.
- After performing Step 5, we have to set the auxiliary matrix to an all-zero matrix by looking ahead once again the very same sequence and setting each accessed edge to 0. This takes $O(m)$ time.
- Also, taking the unions of the graphs and matchings upon returning can be destructive to the original lists, thus it can be done in constant time.

Hence in this case the total cost spent for a phase becomes $O((f(k) + c)m)$ for some universal constant c, yielding an amortized update cost of $O((k + 1) \cdot m^{1/k+1})$ for A_k, which boils down to an amortized $O(\log m)$ update cost by choosing $k = \log m$ and we showed:

Theorem 2. *There exists a fully dynamic algorithm for maintaining a maximal matching with an $O(n^2 \log n)$ initialization cost, $O(\log m)$ amortized update cost and constant query cost using a lookahead of length m.*

Acknowledgements

This work was supported by the ÚNKP-17-3 New National Excellence Program of the Ministry of Human Capacities.

References

[1] Surender Baswana, Manoj Gupta, and Sandeep Sen. Fully dynamic maximal matching in $O(\log n)$ update time. *SIAM Journal on Computing*, 44(1):88–113, 2015.

[2] S. Khanna, R. Motwani, and R. H. Wilson. On certificates and lookahead in dynamic graph problems. *Algorithmica*, 21(4):377–394, Aug 1998.

[3] Ofer Neiman and Shay Solomon. Simple deterministic algorithms for fully dynamic maximal matching. *ACM Trans. Algorithms*, 12(1):7:1–7:15, November 2015.