Spring 2019

Dynamic Graph Algorithms

Mohammadreza Moravej

1 Dynamic Graph Algorithms - Definition and Examples

An update on a graph is a modification to the current existing graph, such as an insertion, a deletion, or a change in attributes associated with edges or vertices of the graph. In a dynamic graph problem, we are concerned with efficiently updating the graph so that we can query a set of specified properties of the graph at any given point of time. Dynamic graph algorithms keep track of the graph and receive online updates about the graph's properties and modify the graph data structure accordingly. Dynamic graph algorithms consists of three main parts:

- Prepossessing: this is when the original graph is processed and loaded into some data structure.
- Updating: when new change is applied to the graph and the graph must be updated. The changes that are commonly considered in dynamic graph problems are adding/removing an edge.
- Querying: when some information regarding the graph is queried.

A dynamic algorithm is called fully dynamic if it can handle both edge insertions and deletions. A partially dynamic algorithm can handle either edge insertions or edge deletions, but not both. The algorithm is incremental if it supports insertions only, and decremental if it only supports deletions. Often when doing the complexity analysis of the query and update operations, the amortized run time is considered. This is due to the fact that a dynamic algorithm may sometimes take some time handling the operations but perform efficiently in the long run.
There are many examples of dynamic graph problems.A famous one being the dynamic connectivity problem. This problem is about maintaining a graph data structure under edge deletions and new edge insertions. The algorithm should also handle queries of form (u,v) and determine if u and v are in the same connected component of G. Another example of a dynamic graph problem is planarity testing, where the algorithm is asked if inserting and edge (u,v) into the graph violates planarity.
We will be concerned with the dynamic transitive closure and maximum matching problems and present some of the most effective algorithms for each problem.

2 Dynamic Transitive Closure

2.1 Problem Definition

The dynamic reachability problems can be divided into three categories based on the type of queries supported. In the Dynamic Single-Source reachability problem, there is a source vertex in the graph and the query asks if a vertex u is reachable from the source vertex. In the Dynamic st-Reachability problem, there there are two marked vertices s and t in the graph that the query wished to know if t is reachable from s.
In this paper, we will focus on the Dynamic Transitive Closure which is a generalization of the other two problems; Given a directed acyclic graph G, and dynamic deletion and insertion of edges in G, query if for input (u,v) there is a path from u to v. The static version of the problem asks to output a $|V| \times |V|$ matrix C where $C(u, v) = 0$ iff there is no path from v to u. We will discuss two algorithms that aim to solve the dynamic transitive closure problem. For the sake of simplicity, we ignore the case where cycles exists in the graph when discussing the relative algorithms.

2.2 Previous Work

- The first algorithm was proposed by Ibaraki and Katoh in 1983 for the incremental version of this problem. The run time of the algorithm was O(n^3) over any sequence of insertions and $O(1)$ for querying.[5]
- The first decremental algorithm was also proposed by Ibaraki and Katoh which had a run time of $O(n^2)$ per deletion.[5]
- The bound of the incremental algorithm was improved to $O(n)$ amortized time per insertion and $O(1)$ per query by Italiano.[6]
- Yallin presented a $O(m * \sigma_m ax)$ algorithm for m edge insertions where m* is the number of edges in the final transitive closure and $\sigma_m ax$ is the maximum out-degree of the final graph.[7]
- The best overall algorithm for the decremental version was given by Henzinger and King with query time $O(n/logn)$ and amortized update time $O(nlog^2n)$.[8]
- The first fully dynamic transitive closure algorithm was a monto-carlo algorithm with a query time $O(n/logn)$ and amortized update time $O(ns^{0.575}log^2n)$, where s is the average number of edges in the graph throughout the whole updating sequence.[8]
- King proposed an algorithm with $O(1)$ query time and $O(n^2logn)$ amortized time per update. [9]

In this paper, we will present two algorithms for the fully dyanmic transitive closure problem. Algorithm I has an amortized update time of $O(n^2)$ and a query time of $O(1)$. Algorithm II has an update time of $O(n^{1.575})$ and query time of $O(n^{0.575})$.

2.3 Algorithm I

A similar idea to the Floyd Warshall algorithm (which is used for solving the All Pairs Shortest Path problem) is used here. Off course, unlike this algorithm Floyd Warshall is a static algorithm. The algorithm, proposed by King and Sagert [1], maintains a transitive closure matrix C and updates it every time the graph G is modified. The matrix C is updated as follows:

- insert(x,y): For every pair $(u, v) \in G$, update $C(u, v) \leftarrow C(u, v) + C(u, x).C(y, u)$
- delete(x,y): For every pair $(u, v) \in G$, update $C(u, v) \leftarrow C(u, v) - C(u, x).C(y, u)$
- query(x,y): return true iff $C(u, v) \neq 0$

Notice how the matrix C is defined such that C(u,v) = the number of paths from u to v.

Time Complexity of the Algorithm

- First, note that the algorithm has an (amortized) lower bound of $\Omega(n^2)$, since it stores a transitive closure matrix. Consider the following instance; G = $G' \cup G''$, $G' \cap G'' = \emptyset$ where G' and G" have $\Omega(n)$ vertices. Pick $u \in G'$ and $v \in G$. Removing and inserting (u,v) would require $\Omega(n^2)$ entries to be updated.

- Clearly, the query operation takes constant time since the algorithm just has to look up the (u,v) entry in C.

- There are $O(n^2)$ updates to perform after a deletion/insertion. However, the run time can actually be beyond $O(n^3)$ if the number of paths between nodes is very large. King and Sageri showed that the complete directed acyclic graph with n vertices has 2^{n-2} paths between the vertex with lowest indegree to the vertex with the highest. That is, let G be a complete directed acyclic graph with $n2$ vertices such that for all $i < j$, there is a directed edge (i, j). Then, there are 2^{n-2} paths between the vertex 1 to n. The proof of the claim is via induction:

  B.C: for n=2 there is $1 = 2^{2-2}$ paths between vertex 1 and 2.
  Inductive Step: Consider G with n vertices. A path from v_1 to v_n either goes through v_2 or it does not. Hence, the number of paths from v_1 to v_n equals the number of paths from v_1 to v_n that do not go through vertex v_2 + the number of paths from v_2 to v_n in $G - v_1$. By IH, both of these are 2^{n-3}. Then, $2 \times 2^{n-3} = 2^{n-2}$ and the claim holds.

  The above claim shows that the numbers in entries can be exponential. Thus, representing such values at each entry of C would require $\Omega(n)$ bits. Since multiplication on n bit integers takes $\Omega(n)$ time, the worst case amortized runtime of the algorithm is $\Omega(n^3)$.

- Note: The upper bound on the algorithm can be improved to $O(n^2)$ if we perform a random prime modulus operation instead of multiplication. This would make the algorithm a monto-carlo algorithm. However, if the prime number is between n^c and n^{2c} for some constant c, then the probability of all queries being correct becomes greater than or equal to $1 - O(1/n)$.

2.4 Algorithm II

Demetrescu and Italiano [2] showed the rather counter intuitive result that the amortized update time can be improved if the algorithm does not maintain a transitive closure matrix C. I will only briefly mention the abstract idea of the algorithm, since it involves lots of complicated non-combinatorial explanations that do not relate to the topic of the course and the paper. Let C_i be the matrix C after the i_{th} update (inserting/deleting (x,y)). For every pair of nodes $u, v \in V$, the first algorithm would perform the following operation to update C:

$$C_i(u,v) \leftarrow C_{i-1}(u,v) \pm C_{i-1}(u,x).C_{i-1}(y,v)$$

where addition is done for insertion and subtraction for deletion.
Instead of computing the above for all u and v, we can simplify the above expression by doing a single dot product:

$$C_i(u,v) \leftarrow C_{i-1} \pm C^{col}_{i-1}(u,x).C^{row}_{i-1}(y,v)$$

where $C^{row}_{i-1}(y,v)$ is the y_{th} row vector of C, and $C^{col}_{i-1}(u,x)$ is the x_{th} column vector. Assume we have a total of A update operations. All the A updates can be expressed as a matrix addition and dot product:

$$C_A \leftarrow C_0 + = \begin{bmatrix} C^{col}_o(x_1) C^{col}_1(x_2) .... C^{col}_{A-1}(x_A) \end{bmatrix} \begin{bmatrix} C^{row}_o(y_1) \\ C^{row}_o(y_2) \\ . \\ . \\ C^{row}_{A-1}(y_A) \end{bmatrix}$$

(The entries in the row matrix would be negative if the operation is deletion). We will skip over how matrix multiplication can be implemented efficiently to achieve an amortized run time of $O(\frac{n^2}{A^{3-\sigma}})$, where $\sigma \leq 3$. We will also skip the details of how the columns and rows of each C_i can be computed efficiently. Demetrescu and Italiano show that with fast matrix multiplication it is possible to achieve an amortized bound $O(n^{1.75})$ for updating and an amortized bound of $O(n^{0.575})$

3 Dynamic Maximum Matching

3.1 Problem Definition

The matching in a graph G is the set of edges that have no common vertices. A matching is maximum if it contains the largest possible number of edges. This is not to be confused with maximal matching, which is a matching M such that for all $e \in G$, e$\cup$M is no longer a matching.
A dynamic maximum matching algorithm does the following task: given a graph G, dynamically delete or insert edges in G based on the received input, and query for the size of the maximum matching in G. Other query operations are Text(e) and mate(v). Text(e) asks the algorithm if there is a matching in G which uses the edge e. mate(v) asks for the vertex that has a common edge

to v in M. The current known algorithms for dynamic maximum matching have bad update upper bounds and are practically inefficient. Hence, the majority of work has been on approximation algorithms. Below is a list of influential papers in this area.

Previous Work

- In 1980, Micali and Vazirani created an algorithm that takes O($\sqrt{n}m$) time.

- In 2007, Sankowski devised a randomized algorithm with O(n^1.495) time per update that maintains a maximal matching.

- In 2011, Baswana et al. showed a randomized algorithm that maintains a maximal 2-approximate matching in O(log n) expected amortized update time. [3]

- In 2012, Neiman and Solomon devised an algorithm with deterministic worst case update time O($sqrtm$) for a 3/2-approximation matching. [4]

3.2 Algorithm I

Before presenting the main O(log n) algorithm, we present a conceptually simple by Baswana et al [3] . Observe that 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 (if there happens to be one such edge, we can simply add it to the matching). When the algorithm is asked to insert an edge, it simply inserts it to the matching iff u and v are both free. When asked to deleted an edge (u,v), the algorithm does the following:

  - if u and v are free, just delete the edge
  - otherwise search their neighbours for any free vertex and update the matching.

Each update operation takes O(1), except deletion of a matched edge which is O(degree(u) + degree(v)) $= O(\|E\|)$.
The algorithm is effective for small degree vertices, but it has a bad performance otherwise.

3.3 Algorithm II

Another algorithm presented by Baswana et al. is a modification of Algorithm I which uses randomization to handle the deletion of a matched edge. A randomly chosen neighbor u is selected to be matched with a vertex u. Baswana et al observed that following the standard adversarial model, an expected degree(u)/2 edges incident to u will be deleted before the matched edge (u,v) is deleted. Hence, the expected amortized cost per deletion for u is O($\frac{degree(u)+degree(v)}{deg(u)/2}$). If we have degree(v) < deg(u), then the amortized cost for deletion is O(1), otherwise the bound shown is not very great.

3.4 Algorithm III

This algorithm presented by Baswana et al achieves an expected amortized time of O($\sqrt{n}$) per update operation. The algorithm is a slightly different, but conceptually simpler version of the O(log n) algorithm.
In high level, the algorithm defines a notion of ownership where each edge is owned by the endpoint vertex which has a higher degree. The algorithm also maintains a partition of the set of vertices into two levels. Level 0 is the set of vertices which own fewer edges than a determined threshold. level 2 consists of the other vertices which own a greater number of edges than the threshold. The algorithm takes advantage of algorithm I(presented above) to handle updates on the vertices in level 1. Similarly, it uses algorithm II (presented above) to handle updates on vertices in level 2.
A finer partition on V can make the algorithm achieve the expected amortized time O(log n) per update. The complexity analysis of this algorithm will be skipped since it is quite long and requires multiple definitions and proofs.

3.5 Algorithm IV

General Description
Neiman et al. present a deterministic algorithm for the dynamic maximal matching which achieves a $O(\sqrt{m})$ update time. Since the algorithm maintains a maximal matching, it is a 2-approximation of the optimal maximum matching. The general idea behind the algorithm is as follows:
When an edge in inserted, add the edge to the matching if both endpoints are free. If neither of the endpoints is free, just add the edge to the graph. If one endpoint is free and the other is not, look into the neighborhoods of the endpoint verities to possibly change the matching. Looking through the neighbourhood may not be very efficient if the endpoints have high degrees and/or the adjacent vertices of an endpoint have a lot of edges between each other. To prevent this scenario, the algorithm maintains and invariant that any free vertex has a small degree. That is, high degree vertices are never free. In particular, the algorithm ensures that vertices of degree larger than $\sqrt{2(m+n)}$ are matched at all times.

Upon inserting an edge between a free vertex and a matched vertex:

- if the free vertex has a small degree, do nothing
- if the free vertex has a large degree, then try to search in the neighbourhoods and modify the current matching to match the newly inserted edge.

When deleting an edge that is in the matching, both endpoints become free. If the endpoints have a high degree, then try to modify the matching so that an edge from the endpoints gets included in the matching. Otherwise just delete the edge and do not do anything extra.

Let's now look at a more detailed description of the algorithm. Denote i the round when an update i is called. Let m_i be the total number of edges of G in round i. There are three invariants that the algorithm wishes to maintain:

- M must remain maximal in every stage

– Any free vertex must have degree less than or equal to $\sqrt{2(m_i + n)}$

– Any free vertex in round i has degree less than or equal to $\sqrt{2m_i}$

Here is how **inserting** an edge (u,v) would work:

– If u and v are matched, do nothing

– If u and v are free, add (u,v) to the matching

– If u is free and v is matched, surrogate(u) (and if the opposite is true, surrogate (v))

Before addressing the algorithm above, consider the following lemme:

lemma
Let u be a vertex in G such that degree(u) > $\sqrt{2m}$, and every neighbour of u is matched. Then, there must exist a vertex w' such that w$\in N(u)$, (w,w') $\in M$, and degree(w') $\leq \sqrt{2m}$

proof
Let N(u) = $\{w_1, w_2, ..., w_s\}$. Let w_i' be the vertex that has an edge to w_i in the matching (in other words $w_i' = mate(w_i)$). Assume for the sake of contradiction that for every i, degree(w_i) > $\sqrt{2m}$, then $\|E\| > d\sqrt{2m}/2 \geq 2m/2 = m$, contradiction.

Now going back to the algorithm for insertion above, the third case of the algorithm is tricky because it violates in variants 2 and 3. The algorithm fixes this issue doing the following in surrogate(u) :

– Iterate through the neighbours of u (which is not expensive since u has a small degree)

– If w' = mate(w) has a small degree, then match u with w and unmatch w with w'

In other words, here is what surrogate(u) does:

```
for w in N(u):
    w' = mate(w)
    if degree(w') =< sqrt(2mi):
        M = M\ {(u,w) U (w,w')}
        for t in N(w'):
            if t is free:    #if w' has a free neighbour
                M = M U {(w',t)}
```

Note that the lemma above guarantees the existence of the desired w'.

Now, lets consider the case for **deleting** an edge (u,v):

```
Remove u,v from E
if (u,v) is in M:
    for w in {u,v}:
        if w has a free neighbor t:
            M += {}(w,t)}
        else if degree(w) > sqrt(2m):
            surrogate(w)

    t = maximum_degree_vertex
    if degree(x) > sqrt(2mi + n):
        surrogate(t)
```

Clearly, the above algorithm satisfies invariants 1 and 3. The last three lines of the algorithm is to make sure that invariant 2 is satisfied as well. Without it, invariant 2 may not be satisfies at each round since m decreases. To show why the last 3 lines ensure that invariant 2 is satisfied, Neiman et al claims if the highest degree free vertex is fixed after each deletion, then invariant 2 will hold. The proof of this claim is quite long and will be skipped in this paper. The above high level description outlines the algorithm which has an upper bound of $O(\sqrt{n+m})$. The faster algorithm for the dynamic maximal matching which achieves $(\sqrt{m})$ update time is similar to the algorithm above, though it is far longer and more complicated.

.

References

[1] King and G. Sagert. A fully dynamic algorithm for maintaining the transitive closure. STOC 1999.

[2] C. Demetrescu and G.F. Italiano. Trade-offs for Fully Dynamic Transitive Closure on DAGs: Breaking Through the O(n^2) Barrier. FOCS 2000.

[3] SURENDER BASWANA, MANOJ GUPTA, SANDEEP SEN, Fully Dynamic Maximal Matching in O(log n) update time, 2011.

[4] Ofer Neiman and Shay Solomon. 2015. Simple deterministic algorithms for fully dynamic maximal matching. ACM Trans, 2012.

[5] T. Ibaraki and N. Katoh. On-line computation of transitive closure for graphs. Information Processing Letters, 16:9597, 1983.

[6] G. F. Italiano. Finding paths and deleting edges in directed acyclic graphs. Information Processing Letters, 28:511, 1988.

[7] D. M. Yellin. Speeding up dynamic transitive closure for bounded degree graphs. Acta Informatica, 30:369384, 1993.

[8] M. Henzinger and V. King. Fully dynamic biconnectivity and transitive closure. InProc. 36th IEEE Symposium on Foundations of Computer Science (FOCS95), pages 664672, 1995.

[9] V. King. Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs. In Proc. 40th IEEE Symposium on Foundations of Computer Science (FOCS99), pages 8191, 1999.

Note: The main content of this survey paper comes from the 4 first papers. The rest of the papers are previous work done in the area of dynamic graph algorithms.