Comparison of Offline and Online Algorithms for Fully-Dynamic Graph Connectivity

Rumen Hristov
Massachusetts Institute of Technology
Cambridge, MA
E-mail: rhristov@mit.edu

December 11, 2013

Abstract

In the fully dynamic graph connectivity problem we are modifying a graph by inserting and deleting edges and answering queries if two vertices are connected by a path. For the offline case we are given all of the queries and modifications beforehand, while for the online we should respond without any knowledge for the future operations.

In this paper we are going to compare implementation of the online algorithm by Holm, de Lichtenberg, and Thorup[1] which has running time of $O(\log^2 n)$ per operation with an implementations of a modified version of Eppstein’s $O(\log n)$ offline algorithm for dynamic minimum spanning tree[2] adapted to solve dynamic connectivity.

1 History

In the fully dynamic graph connectivity problem, we are considering a graph $G$ with $n$ vertices and initially no edges. The goal is to support the following operations in it:

- **insert**(u, v) - insert an edge between $u$ and $v$
- **delete**(u, v) - delete the edge between $u$ and $v$
- **connected**(u, v) - determine if there is a path from vertex $u$ to vertex $v$ in $G$

We will call the inserts and deletes operations - updates, while the connected operation as query. We can assume that we are going to have at most one edge between two vertices and will not want to delete a non-existent edge. Unless otherwise specified the general problem concerns the online case.

The first non-trivial algorithm was given by Frederickson[3] and had an update time of $O(\sqrt{m})$, where $m$ is the number of edges in the graph and query time of $O(1)$. The
update time was later improved by Eppstein, Galil, Italiano and Nissenzweig\cite{4} to $O(\sqrt{n})$. Henzinger and King\cite{5} gave a randomized, amortized algorithm with update time of $O(\log^3 n)$ and query time of $O(\log n/\log \log n)$. In 1996 Henzinger and Thorup\cite{6} lowered the update time to $O(\log^2 n)$.

Holm, de Lichtenberg and Thorup\cite{1} gave a deterministic algorithm with update time of $O(\log^2 n)$ and query time of $O(\log n/\log \log n)$. Later Thorup\cite{6} used a randomized data structure to get a slightly better running times, which were improved by Wulff-Nilsen\cite{7}. The latter introduced a modification to Thorup’s algorithm which led to update time of $O(\log^2 n/\log \log n)$ and query time of $O(\log n/\log \log n)$. Thorup’s randomized data structure\cite{6} is the first to use $O(m)$ space, while all the previous polylogarithmic algorithms use $O(m + n \log n)$.

Patrascu and Demaine\cite{8} showed that in order to solve the dynamic connectivity problem, at least one of the operations should have a running time of $\Omega(\log n)$, so the current algorithms are pretty close to this lower bound.

The offline version of the problem is much easier than the online and hasn’t received so much attention. As we will show later, dynamic connectivity is reducible to dynamic minimum spanning tree. In the dynamic MST problem, we are given a weighted graph and we can modify it by changing the cost of the edges and the queries are asking what is the cost of the minimum spanning tree. Eppstein has provided a $O(\log n)$ per operation offline algorithm for dynamic MST\cite{2}, while Holm, de Lichtenberg and Thorup gave a $O(\log^4 n)$ online algorithm\cite{1}.

## 2 Previous work

### 2.1 HDT algorithm for dynamic connectivity

All of the polylogarithmic algorithms presented translate each update and query to the same operation over one or several trees. To do these operations efficiently in a forest\footnote{We will use the abbreviation HDT to refer to Holm, de Lichtenberg and Thorup’s algorithm}, one can use link-cut trees\cite{9} or Euler Tour trees\cite{10}. Both data structures achieve a running time of $O(\log n)$ per operation for both updates and queries. Both of them are based on keeping the vertices in a binary search tree and that’s where the $\log(n)$ factor comes from.

The algorithm associates a level with every edge currently is in the graph. For each $i$, $G_i$ denotes the subgraph composed of edges with level at least $i$. $F_i$ denotes the maximum spanning tree in $G_i$ with respect to the levels of the edges. Then $F_0$ corresponds to a spanning forest.

We call every edge that is part in one of the spanning trees a tree edge, while we call the others a regular edge. On a very high level we can implement the operations in the following way.

---

\footnote{These data structures don’t support forming a cycle in the graph. They are intended to use when you have trees and you add or remove edges from them.}
connected(u, v) - Because $F_0$ is a spanning forest of the whole graph, we can use our tree data structure to answer the query.

insert(u, v) - We set the level of the edge to 0 and insert it in $G$. If $u$ and $v$ were not connected before, it becomes a tree edge and we connect them in $F_0$. If they were connected then it is a regular edge.

delete(u, v) - If we are deleting an regular edge, we can just remove it from the graph, because it wasn’t important for the structure of our trees. Otherwise we can suppose that we are deleting edge $e$ which has level $l$. If we delete it, it is going to divide $F_l$ in two parts - $T_u$ and $T_v$. 

Now there may be some other edge to replace edge $e$ and connect back $T_u$ and $T_v$. We can take the smaller of the two components and check all of the regular edges that are going out from it and have level $l$. If an edge connects $T_u$ and $T_v$ we can make it a tree edge and connect the components. If the edge doesn’t connect them, we increase it’s level by one. If we don’t find any edge that has level $l$ and connects the components then we try edge edges with level $l-1$ and so on. If we reach level 0 then we are sure that the components can’t be connected.

When we need to remove any edge from $F_l$ we also need to remove it from all $F_i$, where $i < l$. If we find edge $e'$ with which we replace $e$ in $F_l$, we know that we are going to replace $e$ with $e'$ in all $F_i$ where $i < l$.

This is a very high level description of the algorithm and is missing some important details, but one can get a good insight how it works. We can easily do insert and connected in $O(\log n)$ time using our tree data structure.

The main idea behind the $O(\log^2 n)$ amortized running time of delete is that when we discover that an edge doesn’t connect back the two components, we increase it’s level. This way we pay for looking at it. We can maintain the tree in such a way to guarantee that the maximum level is $O(\log n)$. Combined with the $O(\log n)$ factor to check connectivity, this brings an amortized running time of $O(\log^2 n)$ per delete.

### 2.2 Eppstein’s algorithm for dynamic MST

Eppstein’s algorithm is based on several steps that contract the graph. Given graph $G$ with $n$ vertices, $m$ edges and a list of updates $k$, we can contract the graph to have $k + 1$ vertices and $2k - 1$ edges. After that we can still perform the same updates but on much smaller graph, under the assumption that $k$ is smaller than $n$.

Let’s denote the set of edges from the $k$ updates $S$. We assign a weight of $-\infty$ to every edge $\in S$ and find the minimum spanning tree. There are exactly $n - k - 1$ edges from $G \not\in S$ which were part of the MST. Let’s denote the set of these edges by $T$. No matter what the actual weights of the edges in $S$ are, edges in $T$ will be in the MST after every of the $k$ updates.
We create a new graph $G'$. Every connected component created by the edges of $T$ will be vertex in $G'$. An edge from $u$ to $v$ in $G$ will be an edge from the connected component containing $u$ in $G'$ to the one containing $v$.

We reduced $G$ to graph with $k + 1$ vertices and we can reduce the number of edges to $2k - 1$. Let $G''$ denotes the graph $G$ without the edges of $S$. Let $R$ be the set of edges from the minimum spanning tree of $G''$. To process the updates you need to consider only the edges $S \cup R$, leaving us with $2k - 1$ up to edges.

A detailed proof of the above steps can be found in [2]. If we have the list of all the edges sorted, the running time of the contractions steps is just the one of finding MST. Karger, Klein and Tarjan[11] have given a linear time randomized algorithm for minimum spanning tree.

Starting with $k = m$ updates we can contract the graph to $k$ nodes and $2k - 1$ edges. We split the updates in two consecutive groups of size $k/2$ and recursively solve them. We have $O(\log k)$ levels and on every level we have $O(k)$ vertices and edges. When we start with $k = m$ updates, this leads to a running time of $O(\log n)$ per operation.

2.3 From dynamic MST to dynamic graph connectivity

**Lemma 1.** We can easily reduce the fully dynamic graph connectivity problem to solving dynamic minimum spanning tree.

**Proof.** In the dynamic connectivity problem we don’t have edge weights. We transform every edge insertion into change of edge’s weight to 0 and every edge deletion to changing its weight to 1.

For each connectivity query $(u, v)$, if the edge $(u, v)$ exist, than obviously $u$ and $v$ are connected in this moment of time. If the edge doesn’t exist then we can transform it to series of two updates. First we set it’s weight to 0 and then we set it back to 1.

If the value of the MST changed after lowering the weight of the edge, we have connected two components. This implies that $u$ and $v$ were not connected before the query. Contrary if the value of the MST changed, then they were connected by a path.

3 Implementation details

HDT is not the algorithm with the lowest asymptotic running time for dynamic graph connectivity. Using Wulff-Nilsen’s algorithm one can improve the query asymptotic running time by a factor of $O(\log \log n)$. Although in theory this should lead to a better running time, the practical behavior of that algorithm is unknown and we decided to implement the simpler algorithm.

A previous implementation of HDT’s algorithm by Iyer, Karger, Rahul and Thorup[12] has used an Euler Tour tree using treaps. We decided to implement HDT using link-cut trees with splay trees. One disadvantage of our approach is the added amortization. The
height of a splay tree can be $O(n)$. In our case this is not a huge issue, because the HDT algorithm is already amortized.

An advantage of splay trees is that they maintain the frequently used nodes near the root. When doing an update in HDT, we are going to perform multiple splays on small subset of the nodes. This will lead to a smaller constant factor than using other binary search trees.

4 Input families

The work presented in this paper stands with relation to the previous implementation work of Iyer, Karger, Rahul and Thorup[12]. We have implemented most of their test cases and added additional to analyze the behaviour of the offline algorithm. We were able to use much bigger inputs than them because of the hardware improvements during the years, so the $O(\log n)$ factor, by which the algorithms are different from each other, should be more noticeable.

If we are doing operations only on subset of the edges $S$, the offline algorithm takes advantage of that. It will reduce the graph to one of $O(|S|)$ nodes, but the edge sizes will not change by much. Still the algorithm’s performance will benefit from having a small $|S|$.

For HDT the input matters even more. If we are deleting a regular edge, we are doing $O(1)$ operations. But if it was a tree edge we will do $O(\log^2 n)$ operations amortized. This volatility in the running time suggests that we should analyze the inputs compared mostly to HDT.

To better evaluate the difference, we decided to compare the two implementations with different values of the number of vertices.

4.1 Random graphs

The most obvious input that one can analyze are randomized graphs. There is no point in having a lot more edges than vertices. If this is the case we will most probably have a small number of big components and around $n$ tree edges.

We generated tests with $n$ vertices and $3n$ operations. For big number of vertices, this is enough to make the graph non trivial. We start by building a spanning tree on the graph. Then on each step we randomly chose an operation and an edge on which to perform it.

4.2 Worst case for HDT

We wanted to see how our implementation of HDT behaves. The worst case for it is when the input is just a line of $n = 2^k$ vertices. On the first step we remove the edge in the middle. After that we are left with two lines of size $2^{k-1}$ and we can recursively repeat.

The algorithm every time will check half of the vertices in the line that we just split. But when we delete an edge, we remove it from several trees in the internal implementation. This will cause going through several halves of components to look for replacement edges.
In this case the algorithm will exhibit it’s worst case bound of $O(\log^2 n)$ per operation. A more detailed proof why this is true can be found in [12].

If we generate a test only as a line, our implementation will see that there aren’t any regular edges available in the component and will not search for any. A better way to generate this test is if every vertex represents a clique\(^3\) of size 3. This way we will always have some regular edges in the graph which will be available, but we can’t make use of them.

### 4.3 Online better than offline

We tried to generate test cases where the online algorithm is better than the offline.

If the set of the edges that we are working with is big, then Eppstein’s running time should be the same no matter what the operations are. However for HDT, the inserts and queries are much faster than replaces of tree edges.

It looks like the online algorithm can’t get worse than the offline algorithm. One possibility to get a sublogarithmic process time in the online case is, if we have some vertices that are frequently accessed. Because we implement HDT with link cut and splay trees then the last used values are near the roots. They will need less time to be accessed and the running time can be better than $O(\log n)$. We tried the following inputs to see if HDT does better:

1. The number of queries and inserts is a lot more than the number of deletes.
2. We have a small number of components and pick one vertex from each of them. Then we modify and query edges only between these vertices.
3. We choose a small subset of the vertices $S$. Every operation is a query asking if two vertices from $S$ are connected.

We will again start from a spanning tree and after that we will generate random graphs.

### 5 Results

All of the running times that we are presenting in this section are averaged over 5 runs of the implementation with the same test data.

#### 5.1 Random graphs

For random graphs we didn’t expect any anomalies in the running time of the two algorithms. We correctly guessed that the offline algorithm will be much faster.

\(^3\)Clique is a graph where every two vertices are connected by an edge
The reason that the offline is so much faster is it’s lower asymptotic complexity. Also it has much smaller constant factor, because of the simplicity of it’s operations. We looked at the performance of HDT and it was executing the first 1 million updates extremely fast. That is because we are just building a spanning tree and we don’t do any deletions, which are more complicated and slower.

An interesting observation is how the online algorithm is slower almost exactly by a factor of \( \log(n) \) for all of the values that we have on the plot. This matches exactly our expectations.

### 5.2 Worst case for HDT

In this test we expected to observe the biggest differences between the online and the offline algorithms. The HDT algorithm was supposed to reach its worst case.
As the plot shows, we have 40 times difference between the two implementations, compared to the random graphs where we had 15-20 times difference. HDT reached its worst case performance and it’s slower by a factor of more than \( \log(n) \) from the offline algorithm.

The running time of Eppstein’s algorithm is very consistent, except the case for \( n = 100 \). Our explanation is that in the beginning of the execution there is a small overhead from the memory allocation. Because we are measuring seconds per million operations, this impacts our measurements.
5.3 Online better than offline

Our first test was, when the number of queries and inserts is much bigger then the number of deletes. We generated random graphs where the number of deletes was around 0.1% of the total number of operations.

We expected that the difference between the two implementations will be a factor smaller than 15 - the one that we observed for random graphs.

For 1 million vertices we reached a factor of 5.2 difference, which is significantly better. While Eppstein’s algorithm doesn’t benefit from the input, HDT doesn’t perform its slow updates. Asymptotically both solutions should run in $O(\log n)$ time, but again the simplicity of the implementation of the offline algorithm dominates. The added constant factor of all of the data structures makes HDT over 5 times slower.

The second test that we ran was having a small number of components (5 in our case) and using only a one vertex from each component. Then we were operations were concerning only 5 vertices.
Here we knew that HDT will exploit it’s splay tree and will do the operations on these five vertices extremely fast so we expected to get close to the running time of Eppstein’s algorithm.

The HDT algorithm was just a factor of 2 slower. The main reason why it is not even better is because the offline algorithm performed very fast as well. As we said earlier Eppstein’s algorithm is faster on inputs where we have a small subset of edges that we are modifying. Here it uses the fact that most of the time we are working on a very small set of edges, so this optimizes the performance.

The last test inputs were only connectivity queries working on a limited set of vertices. We expected that the HDT algorithm will outperform the offline algorithm on this data set,
because of the locality of reference of the splay tree and the almost constant responses that it will give. The Eppstein’s algorithm should still run very fast, but it should go through more edges on every level.

The two solutions had a factor of 5 difference, which was more than the previous data set. The only explanation that we have is that all of our queries return that the two vertices are in the same component, because the graph is a spanning tree. Then every time we are doing rotations in the same tree. Despite the concerned vertices being always the 5 closest to the root of the splay tree, we are doing a lot of rotations to splay the one that we need. Added to the maintenance that we need to support the link-cut data structure, it is possible that this is the reason for observed slowdown of 5 times.
6 Conclusions

We briefly presented a very high level explanation of two algorithms for solving the dynamic connectivity problem. Despite the offline algorithm having a much better asymptotic running time, we found test data where the online can perform very similarly to it.

Although using splay trees for balancing was a very big challenge in the implementation of HDT, on some inputs this led to performance improvement.

7 Acknowledgments

We would like to thank Richard Peng for helpful discussion and providing us with a link cut tree code, which stood as the basis of our HDT implementation.

References


