

[CSN-531] Lecture 2

Dynamic Graph Algorithm

Scribe: Nikhil Katoch

1 Introduction

A graph is an abstract data type representing non-linear relationships between data. Graphs contain vertices (nodes), which can be connected to each other through edges (arcs). Assume a graph $G(V, E)$ having $|V| = n$ vertices and $|E| = m$ edges. In real-life scenarios, we mostly deal with dynamic graphs (that always change with time). Prominent examples include google maps, internet routing, and social Networks.

To deal with these kinds of problems, *trivially* one can use the usual graph algorithms (static) to recompute everything from scratch, which are inefficient. So if we are trying to solve problems of that massive scale, we need better algorithms to help us reduce the computation cost on dynamic graphs.

In general Dynamic graph algorithms try to reduce the re-computation part of the algorithms, i.e., for any problem, finding the solution by reiterating the entire process. Instead, we will try to store the partial solution of the problem and, in future runs, use that partial knowledge to compute faster solutions compared to the static approaches.

2 Dynamic Graph Algorithms

Dynamic Graphs are the graphs that keep changing with time or, in the technical sense, Graphs that change by an *online* sequence of updates. Here, *online* refers to the fact that we do not know the updates in advance. **Dynamic Graph Algorithms** is the area that studies the design and analysis of algorithms for dynamic graphs. *Trivially*, we can use the best static algorithm to recompute from scratch after every update. The aim of a dynamic graph algorithm is thus to *reuse previous computation* for computing the updated solution much faster than the static algorithm.

Example: Let's take a basic example of Graph connectivity, i.e., given an undirected graph, the aim is to answer the *query* whether the graph is *connected* or not. The trivial static solution is running BFS/DFS over the graph, and then based on it, we can decide if the graph is connected or not, requiring $O(m+n)$. Now, let us consider the problem in the dynamic setting, where one can always add or delete an edge from the graph. The problem is to report whether the resultant graph is connected or not. Suppose we perform k such operations and always run BFS/DFS over it to check the connectivity, requiring $O(k(m+n))$ time. On the other hand, we can maintain a spanning tree of the graph which ensures efficient updates unless the spanning tree is affected. In case the graph is connected or the spanning tree spans all vertices, any edge insertion will not affect the *connectedness* of the graph. Similarly, if the graph is not connected and has a spanning tree for each component, edge insertion can connect the required components using the spanning tree. Or ignore the insertion if it connects vertices from the same component. Both these operations take $O(1)$ time per update. The edge deletion will not affect the *connectedness* if the deleted edge does not belong to the spanning tree. Otherwise, we need to find a replacement edge to reconnect the subtrees formed by the removed edge. Thus, an edge deletion rarely requires $O(m)$ time, as only $n - 1$ of m edges are in the spanning tree. This rare event can also be exploited by randomization to give efficient *expected* update time.

3 Motivation

In this course, we will deal with *dynamic graphs* as discussed above. Reasons for studying the same include.

- Most real-life problems are based on *dynamic graphs*. Example: Google maps (with congestion on roads changing weights or blockage deleting edges), internet routing (congestion at links or edges, fault in routers or deleted vertices), social networks (with changing connections or insertion/deletion of edges).
- Recomputing solutions on such real-world graphs is *impractical* because of the sheer size of data.
- Studying such algorithms is also significant theoretically because
 - Serves as a black box in other algorithms. Eg. Disjoint Set Union for MST algorithms, which is essentially a graph maintaining components (sets) under insertion of edges (union of sets).
 - Related to the complexity of parallel algorithms. The problems which are harder to have a parallel solution, are also harder to have a dynamic solution.
 - Related to the complexity of Hard problem in P. A new area of complexity theory called as *Fine Grained Complexity* deals with the study of the hardness of problems in *P*, which uses dynamic graph algorithms to establish hardness results on other problems in *P*.

4 Dynamic Graph Model

Dynamic Graphs deal with an online sequence of *graph updates* which are of the following types:

- Insertion of Edges
- Insertion of Vertex
- Deletion of Edges
- Deletion of Vertex

Here the **goal** is to report the solution or maintain the data structure (say DFS, BFS, Number of components, connectedness, shortest path tree etc.) after every update. Consider Figure 1, note that for a static algorithm the input size is $O(m + n)$ hence any non-trivial problem requires $\Omega(m + n)$ time to at least process the entire input. However, the input size for a dynamic algorithm after an update is the size of the update which is merely $O(1)$, thus any dynamic graph algorithm can have an update time $\Omega(1)$ which is potentially much faster than the static algorithm. Say for edge insertion/deletion, the update is merely the indices of the endpoints of the edge requiring $O(1)$ size.

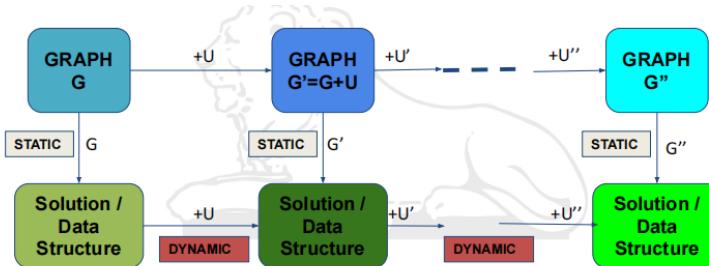


Figure 1: Difference in size of input for static and dynamic algorithms

Dynamic Graphs models are basically classified as follows

1. *Partially Dynamic algorithms* support either insertions or deletions in the graph but not both. These are further classified into incremental algorithms support only insertions and decremental algorithms support only deletions.
2. *Fully Dynamic algorithms* support both insertions and deletions in the graph.
3. Others models include *Fault Tolerance*, *Dynamic Subgraph*, *Oracles* etc.

Comparison between Partially Dynamic and Fully Dynamic

It seems obvious that partially dynamic problems are comparatively easier than fully dynamic variants. But here, we are more interested in formal proof for this comparison. Let's try to prove this using contradiction.

We assume there exists a problem whose partially dynamic variant takes longer than the fully dynamic graph variant. Say for the fully dynamic variant, we have an algorithm X that takes $O(T)$ to do updates (incremental and incremental) on the graph. Similarly, assume in the partially dynamic variant, we have some series of insertions on the graph, each of which takes $\omega(T)$. But if we look carefully, it contradicts our assumption since fully dynamic takes $O(T)$ time, and we know a fully dynamic algorithm can handle both insertion and deletion operations. So if we consider a specific scenario where only the insertion operation we need to perform on the graph, we can do that in $O(T)$ time. And we can use the same algorithm to solve the partial dynamic graph problem. So our assumption that the partially dynamic graph algorithm takes more time compared to the fully dynamic graph algorithm is false, thus

$$\text{Partial Dynamic Graph algorithms} \leq \text{Fully Dynamic Graph Algorithms}$$

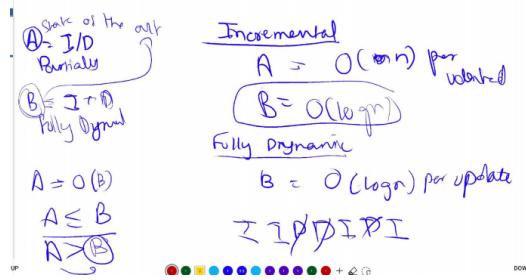


Figure 2: Proof of Partial vs. Fully Dynamic

Lemma 1. *The fully dynamic version of a problem is not easier than the partially dynamic version.*

Proof. Consider an algorithm A that solves the problem in the fully dynamic model. The same algorithm can also solve the problem in the partially dynamic model with equal efficiency because the fully dynamic model allows all types of updates. However, an algorithm A' that solves the partially dynamic problem cannot be used in the fully dynamic model as it may not allow both types of updates. \square

Comparison between Incremental and Decremental

The incremental updates may seem easier than decremental algorithms as described in Section 2 for the problem of Connectedness. The insertions were shown to be handled in $O(1)$ time, while deletions may require $O(m)$ time in the worst case. However, it may not always be the case. Consider the example of maintaining topological order for a DAG. The order is an ordering of vertices of the graph such that any edge (x, y) is directed from a vertex of lower order x to a vertex of higher order y . Clearly, the insertion of an edge can violate the topological order that may be needed to be computed again. However, the deletion of an edge may never hamper the topological order so no update is required. Hence for the problem of maintaining topological order decremental updates are easier to handle as compared to incremental updates.

5 Time complexity

The time complexity of a dynamic graph algorithm is basically measured in the following:

- **Preprocessing Time:** Initial time we take to process the graph. A *polynomial* bound is sufficient.
- **Update Time:** Time is taken after each update to reconstruct the structure.
 - **Max:** $O(\text{Static})$ [as max time as it takes to compute result from scratch]
 - **Min:** $O(1)$ [do not process until query required]
- **Query Time:** Time is taken to answer the query on the updated structure.
 - **Max:** $O(\text{Static})$ [as max time as it takes to compute result from scratch]
 - **Min:** $O(1)$ [process each update completely computing all possible queries]

Generally, there is a tradeoff between Update Time and Query Time by either *lazily* deferring all computations to the query, or *eagerly* computing the complete solution for all queries. Hence, the goal is typically to optimize $O(\max(\text{query}, \text{update}))$ time.

Another significant tradeoff is between Worst Case and Amortized bounds. Amortized bounds are usually easier to get, and the ideal case is to have efficient worst-case bound as well.

PS: Worst-case bound $\Theta(T)$ implies amortized bound $O(T)$ as well.

6 Limitations and Bounds

For a given problem P, let the best static algorithm takes $T_s(n)$, and the best dynamic algorithm takes $T_d(n)$.

Upper Bound: $\max(\text{update}, \text{query}) = T_d(n) = O(T_s(n))$

This is because, there is no use in developing the dynamic algorithm when its complexity is more than the static algorithm, which can be used instead after every update.

Lower Bound: $\max(\text{update}, \text{query}) = T_d(n) = \Omega(T_s(n)/(\text{number of operations}))$

The above bound is only for an *incremental* algorithm requiring $T_d(n)$ time per edge update. Starting from an empty graph on inserting m edges the total time is $O(mT_d(n))$, giving the solution of the problem for the whole graph. This is a valid *static* algorithm as well, hence the best static algorithm $T_s(n) = O(mT_d(n))$ or $T_d(n) = \Omega(T_s(n)/m)$. Note that for vertex insertions updates are n , in general, we get the bound according to the number of updates. The above argument also gives the following result.

Lemma 2. *The incremental version of an NP-Hard problem also required exponential time.*

Proof. To solve an NP-Hard problem using incremental, we require to give a static algorithm using its dynamic operations. We initially have an empty graph, so preprocessing time is $O(1)$.

$$\begin{aligned} \text{Preprocessing} + (m \times \text{Update}) &= \Omega(2^n) && (\text{Static Algorithm is NP-Hard}) \\ O(1) + (m \times T_d(n)) &= \Omega(2^n) && (\text{Preprocessing is } O(1)) \\ T_d(n) &= \Omega\left(\frac{2^n}{m}\right) \end{aligned}$$

So here we have the lower bound for the incremental problem which is also exponential. \square

Note: The same argument cannot be directly used for decremental because of a larger preprocessing time. In the case of a decremental solution, we know that we have been given a graph, and to compute the NP-Hard problem's solution, we first need to at least take preprocessing time over it, which will be $\Omega(2^n)$. So our equation will be:

$$\begin{aligned} \text{Preprocessing} + (m \times \text{Update}) &= \Omega(2^n) && (\text{Static Algorithm is NP-Hard}) \\ \Omega(2^n) + m \times T_d(n) &= \Omega(2^n) \\ T_d(n) &= \Omega(1/m) \end{aligned}$$

Which helps us nothing to generalize the lower bound for decremental problems. That's why the lower bound on incremental does not apply to decremental.

Further Note: For decremental algorithm for some problems we can prove the same. Start from a complete graph whose solution for the NP hard problem may be known (eg. vertex cover). We can end up deleting edges to get the required graph.

This works for unweighted graph problems as Vertex cover, but for weighted graphs as Travelling Salesman it will not work as solution for even complete graph may not be known.