CSL-531: Dynamic Graph Algorithms

Mmukul Khedekar

January 15, 2026

Notes for the course *Dynamic Graph Algorithms* instructed by Dr. Shahbaz Khan at IIT Roorkee, Spring, 2026.

Contents

1 Jan 13, 2026

This course will be graded through the following components:

- **MTE:** 30%
- **CWS:** 30%
- **ETE:** 40%

Both the MTE and the ETE will be written examinations. The CWS component will be based on announced quizzes, tutorial session and a course project. The goal of the course is:

1. to be able to **Design** dynamic graph algorithms using standard design techniques and **prove** its correctness.
2. to be able to **Analyze** dynamic graph algorithms using standard analysis techniques and **prove** its tightness.
3. to be able to **Prove** the hardness of a given dynamic graph problem by **reducing** it to a standard hard problem.

1.1 Introduction to Dynamic Graph Algorithms

> **Definition 1.1.** Dynamic Graphs are graphs that *change* with time. A *change* in a graph refers to **graph updates**, that are *online* sequences of insertion or deletion of edges or vertices.

Therefore we can model a dynamic graph as a sequence of updates on a static graph. Formally, we can view this as

$$G_0 \xrightarrow{\Delta_1} G_1 \xrightarrow{\Delta_2} G_2 \xrightarrow{\Delta_3} \dots$$

where each Δ_i is an update to the graph. Suppose we are interested in computing some quantity $\mathcal{X}$ of the graph G for which there exists a known static graph algorithm $\mathbf{X}$. Upon each update Δ_i to G, we can trivially compute $\mathcal{X}$ for the new graph G' by updating G and running $\mathbf{X}$ on it. The goal of a **dynamic graph algorithm** is to reuse previous computations to update the quantity $\mathcal{X}$ *much faster* than a static graph algorithm.

Most graphs in real world are dynamic. The sizes of parameters for these graphs are significantly large and thus, recomputing solutions on such graphs from scratch is impractical. This motivates us to look for dynamic variants of static graph algorithms. However, we are only interested in dynamic graph algorithms that perform *better* than the best static graph algorithms that exist for the problem.

1.2 Classification of DGAs

Dynamic Graph Algorithms (DGAs) can be classified into

1. **Incremental** (only insertions)
2. **Decremental** (only deletions)
3. **Fully Dynamic** (both insertions and deletions)

It is worth mentioning that in case of weighted graphs, we might want to consider incrementing and decrementing the weights as graph updates too. However, any increment or decrement to the weight of an edge could be modelled as a deletion followed by an insertion. Therefore, we limit the definition of graph updates to

Definition 1.2. A dynamic graph $G(V, E)$ supports the following **graph updates**

- Insertion of edges
- Deletion of edges
- Insertion of vertices
- Deletion of vertices

To analyse the performance of algorithms, we need to equip ourselves with some theory on time complexity analysis.

1.3 Time Complexity Analysis

Some key notations on time complexities.

Definition 1.3. For functions f and g, we have

1. $f(n) = \mathcal{O}(g(n))$, if there exists positive real numbers M and n_0 such that,
$$|f(n)| \leq Mg(n), \qquad \forall n \geq n_0$$
2. $f(n) = o(g(n))$, if for every positive real number M, there exists n_0 such that,
$$|f(n)| \leq Mg(n), \qquad \forall n \geq n_0$$
3. $f(n) = \Omega(g(n))$, if there exists positive real numbers M and n_0 such that,
$$|f(n)| \geq Mg(n), \qquad \forall n \geq n_0$$
4. $f(n) = \omega(g(n))$, if for every positive real number M, there exists n_0 such that,
$$|f(n)| \geq Mg(n), \qquad \forall n \geq n_0$$
5. $f(n) = \Theta(g(n))$, if there exists positive real numbers M_1, M_2 and n_0 such that,
$$M_1 g(n) \leq |f(n)| \leq M_2 g(n), \qquad \forall n \geq n_0$$

A common way to establish a tight bound while analysing an algorithm is to find a worst-case example, which helps us establish a lower bound. In dynamic graphs, we commonly deal with the following.

1. **Preprocessing Time**: Initial time to preprocess the graph.
2. **Update Time**: Time taken after each update to reconstruct the data structure.
3. **Query Time**: Time taken to answer query on the updated structure.

1.4 Update-Query Tradeoff

There is often a trade-off between how fast updates are processed with how fast queries can be answered. Suppose a static algorithm has time complexity $\mathcal{O}(T)$. The extremes of the update-query tradeoff occur when we adopt the following approaches.

- **Eager Approach**
  - $\mathcal{O}(T)$ update time.
  - $\mathcal{O}(1)$ query time.
- **Lazy Approach**
  - $\mathcal{O}(1)$ update time.
  - $\mathcal{O}(T)$ query time.

The above approaches are feasible when queries are more frequent, and when updates are more frequent, respectively. Usually when we design an algorithm, we aim to find a sweet spot between these extremes depending on the constraints and requirements of the problem. At times, even when the worst-case bound is proven to be tight, improving the amortized bound is still considered an improvement.

Example 1.4

DSU (**Disjoin Set Union**) is a popular example of a dynamic graph algorithm. It supports two operations, **Join** and **Find**, both of which achieve an amortized time complexity of $\mathcal{O}(\alpha(n))$. Here, $\alpha(n)$ is the **inverse ackermann function**. Suppose we consider an incremental dynamic graph model that has m edge updates. One might think that the total time complexity of this algorithm would be $\mathcal{O}(m\alpha(n))$. However, we can only have $n-1$ edges in the final graph. Therefore, the total time complexity must be $\mathcal{O}(n\alpha(n))$ and the amortized time complexity achieved per update is

$$\mathcal{O}\left(\frac{n\alpha(n)}{m}\right)$$

1.5 Time Complexity Bounds

So far, we have given an informal discussion of the ideal performance of a dynamic algorithm. Now we consider various scenarios and analyse them in detail.

Consider a problem $\mathcal{P}$ having a best static algorithm $T_s(n)$. Now consider a dynamic algorithm $T_d(n)$ for the problem $\mathcal{P}$.

1.5.1 Upper Bound on $T_d(n)$

Since we cannot have $T_d(n)$ perform worse than $T_s(n)$, therefore

$$T_d(n) = \mathcal{O}(T_s(n))$$

1.5.2 Lower Bound on $T_d(n)$

1. If $T_d(n)$ is an incremental algorithm, then consider a graph with size $\mathcal{O}(n)$ constructed with m incremental updates. Applying $T_d(n)$ at every incremental update and summing up, we get

$$mT_d(n) = \Omega(T_s(n)) \implies T_d(n) = \Omega\left(\frac{T_s(n)}{m}\right)$$

2. If $T_d(n)$ is a decremental algorithm

3. If $T_d(n)$ is fully dynamic

Question 1.5. Think about the lower bound on $T_d(n)$ in the case of a decremental, and fully dynamic algorithm.

Question 1.6. Does there exist a dynamic algorithm for an **NP-hard** problem that lies in **P**?

2 Jan 14, 2026

In this lecture, we elaborate on our discussion of the lower bounds of $T_d(n)$.

Here is the premise of our discussion. Given a problem $\mathcal{P}$ and the best known static algorithm to solve $\mathcal{P}$ called $T_s(n)$, we wish to analyse the time complexity of a dynamic algorithm $T_d(n)$ for solving $\mathcal{P}$.

Lemma 2.1 (Upper Bound on $T_d(n)$)

For any problem $\mathcal{P}$, it always holds that

$$T_d(n) = \mathcal{O}(T_s(n))$$

Proof. If the time complexity of the dynamic algorithm $T_d(n)$ exceeds that of the static algorithm $T_s(n)$, then there is no use in developing the dynamic algorithm. In such a case, one could simply use $T_s(n)$ to recompute the desired quantity after every update. □

2.1 $T_d(n)$ is Incremental

Suppose $T_d(n)$ is an incremental dynamic algorithm for $\mathcal{P}$, meaning that $T_d(n)$ only supports incremental updates. We consider the following two types of incremental updates.

2.1.1 Edge Updates

Let $G(n, m)$ be a graph on n vertices and m edges. We would like to analyse the bounds on $T_d(n)$ when considering **edge insertions** to G.

1. *Upper Bound.* By lemma 2.1, we immediately obtain an upper bound on $T_d(n)$ given by
$$\boxed{T_d(n) = \mathcal{O}(T_s(n))}$$

2. *Lower Bound.* Consider a graph with n vertices and no edges. Constructing this graph takes $\mathcal{O}(1)$ time, and we assume that computing $\mathcal{P}$ on this graph also takes $\mathcal{O}(1)$ time. To obtain the target graph G, we must perform m edge insertions. Applying the dynamic algorithm $T_d(n)$ after each edge update yields a valid *static* algorithm to compute $\mathcal{P}$. Since $T_s(n)$ is the best known algorithm for static computation of $\mathcal{P}$ on G, therefore we must have
$$\mathcal{O}(1) + mT_d(n) = \Omega(T_s(n))$$
which implies
$$\boxed{T_d(n) = \Omega\left(\frac{T_s(n)}{m}\right)}$$

2.1.2 Vertex Updates

Let $G(n, m)$ be a graph on n vertices and m edges. Now we would like to analyse the bounds on $T_d(n)$ when considering **vertex insertions** to G.

1. *Upper Bound.* By lemma 2.1, we have

$$\boxed{T_d(n) = \mathcal{O}(T_s(n))}$$

2. *Lower Bound.* If each update on G inserts a singleton vertex *without* any edges, then assuming that no nontrivial recomputation is required for $\mathcal{P}$, we obtain the lower bound

$$\boxed{T_d(n) = \Omega(1)}$$

   If instead, each update on G adds a singleton vertex *with* edges, then computing the lower bound for $T_d(n)$ in a similar fashion to that for edge insertions, we obtain

$$\mathcal{O}(1) + nT_d(n) = \Omega(T_s(n))$$

   which implies

$$\boxed{T_d(n) = \Omega\left(\frac{T_s(n)}{n}\right)}$$

2.2 $T_d(n)$ is Decremental

We now assume that $T_d(n)$ is a decremental algorithm for $\mathcal{P}$. We first want to solve the problem on a trivial case and then apply a sequence of graph updates to obtain the given graph.

For incremental algorithms, we started with an empty graph. Similarly, for decremental algorithms, we would require a graph from which we could perform a sequence of decremental graph updates and obtain the given graph. A **complete graph** is a suitable choice since, every graph on n vertices is a subgraph of the complete graph on n vertices.

Suppose that we can solve $\mathcal{P}$ on a complete graph trivially, in $\mathcal{O}(1)$ time. If the update operation consists of **edge deletions**, then constructing the complete graph requires $\mathcal{O}(n^2)$ time, and we would require $\mathcal{O}(n^2)$ edge deletions to transform it to G. This implies that

$$\mathcal{O}(n^2) + \mathcal{O}(n^2)T_d(n) = \Omega(T_s(n))$$

From here, we can derive lower bounds on $T_d(n)$ depending upon $T_s(n)$. For example,

$$T_s(n) = \mathcal{O}(n^2) \implies T_d(n) = \Omega(1)$$

and,

$$T_s(n) = \omega(n^2) \implies T_d(n) = \Omega\left(\frac{T_s(n)}{n^2}\right)$$

However, solving $\mathcal{P}$ on a complete graph is not always trivial. For example, the **Traveling Salesman Problem** is exponential on a complete graph. Hence, a general treatment of the time complexity of decremental algorithms is difficult, and we instead analyse specific problems to obtain more meaningful bounds.

2.2.1 Decremental MST (Minimum Spanning Tree)

Let $T_d(n)$ be a decremental algorithm for computing the **Minimum Spanning Tree** of a graph G on n vertices, and let $T_s(n)$ be the best known static algorithm to compute the MST.

> **Definition 2.2.** The **Minimum Spanning Tree** of a connected, weighted, undirected graph G is a subset of edges that connects all the vertices of G and has the minimum sum of weights.

Suppose we consider **edge deletions** for our algorithm. We could obtain a lower bound on $T_d(n)$ by the following method. Introduce a dummy vertex v' and connect it to every vertex $v \in G$ with edges of extremely small weights. Let the resulting graph be G'. To construct G' from G, we require $\mathcal{O}(n)$ time. The minimum spanning tree of G' is trivial because it contains all the edges connecting v'. Deleting these n edges recovers G, and we obtain

$$\mathcal{O}(n) + nT_d(n) = \Omega(T_s(n))$$

and hence

$$\boxed{T_d(n) = \Omega\left(\frac{T_s(n)}{n}\right)}$$

If instead we considered **vertex deletions** for our algorithm, then

$$\mathcal{O}(n) + T_d(n) = \Omega(T_s(n))$$

which implies

$$T_d(n) = \Omega(T_s(n))$$

Using lemma 2.1, we obtain

$$\boxed{T_d(n) = \Theta(T_s(n))}$$

This tells us that the we have a tight bound and there is no asymptotic benefit in designing a decremental algorithm for computing the MST that supports only vertex deletions.

2.2.2 Decremental BFS Tree

Let $T_d(n)$ be a decremental algorithm to compute the **BFS Tree** on a graph G with n vertices, and $T_s(n)$ be the best known static algorithm to compute the BFS Tree.

Definition 2.3. The **BFS Tree** of an unweighted graph G is the spanning tree produced by running the BFS algorithm on G.

Let's say we allow **edge deletions**. Observe that we can trivially compute the BFS Tree on a complete graph in $\mathcal{O}(n)$ time. Hence, we could add dummy edges to G to transform it to a complete graph on n vertices in $\mathcal{O}(n^2)$ time. Moreover, we require $\mathcal{O}\left(n^2\right)$ edge deletions to transform the complete graph back to G. Therefore

$$\mathcal{O}(n^2) + \mathcal{O}(n^2)T_d(n) = \Omega(T_s(n))$$

and hence

$$\boxed{T_d(n) = \Omega\left(\frac{T_s(n)}{n^2}\right)}$$

2.2.3 Decremental Traveling Salesman Problem

Let $T_d(n)$ be a decremental algorithm to compute the **Traveling Salesman Problem** on a graph G with n vertices, and $T_s(n)$ be the best known algorithm to compute this problem.

Definition 2.4. Given a complete, weighted, undirected graph G and a source vertex s, the **Traveling Salesman Problem** asks for a minimum weight cycle starting and ending on s that visits all the vertices in graph G, exactly once.

Suppose we allow **vertex deletions** for our algorithm. Add n dummy vertices to the graph G and for each dummy vertex v, connect all the vertices in G to v with extremely small weights. Let the resulting graph be G'. In this graph, computing the minimum weight cycle that visits all the vertices is trivial. It is just a cycle that alternates between each vertex of G and the dummy vertices. Constructing G' takes $\mathcal{O}(n^2)$ time, and computing the minimum weight cycle requires $\mathcal{O}(n)$ time. Further, we require n vertex deletions to transform G' to G. Hence

$$\mathcal{O}(n^2) + nT_d(n) = \Omega(T_s(n))$$

and therefore

$$\boxed{T_d(n) = \Omega\left(\frac{T_s(n)}{n}\right)}$$

If instead we allow **edge deletions** for our algorithm, then we would have to relax the constraint on the Traveling Salesman Problem for G to be a complete graph, so that every edge deletion results in valid subproblem. We construct graph G' again, but in this case we require $\mathcal{O}(n^2)$ edge deletions to transform G' to G. Therefore

$$\mathcal{O}(n^2) + \mathcal{O}(n^2)T_d(n) = \Omega(T_s(n))\,,$$

and hence

$$\boxed{T_d(n) = \Omega\left(\frac{T_s(n)}{n^2}\right)}.$$

2.3 $T_d(n)$ is Fully Dynamic

For fully dynamic algorithms, it is difficult to make general statements. However, since incremental and decremental algorithms are special cases of fully dynamic algorithms, any lower bound for these special cases also yields a (possibly weaker) lower bound for fully dynamic algorithms.

Remark 2.5 — There cannot exist a polynomial time dynamic algorithm for an **NP-hard** problem, because if such an algorithm existed then we could apply it in an incremental fashion to obtain a polynomial time static algorithm to solve the problem, contradicting NP-hardness.