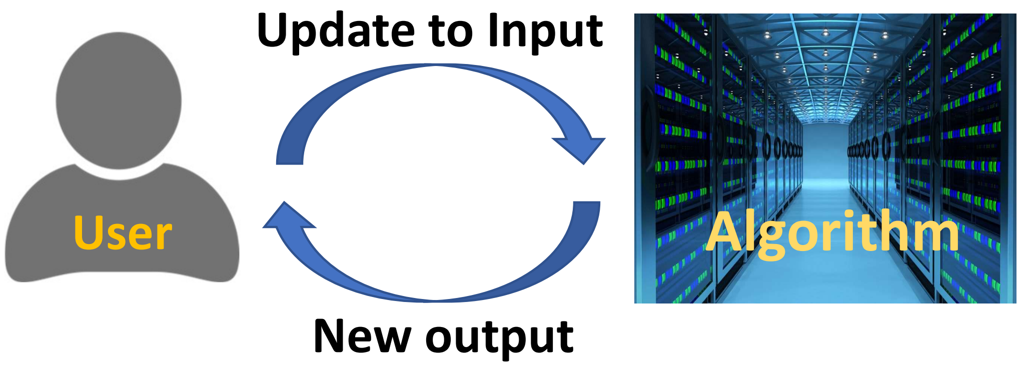
Design Templates of Dynamic Graph Algorithms

Thatchaphol Saranurak
University of Michigan

December 8, 2025

Dynamic Algorithms: Algorithms that Interact



Dynamic graph algorithms

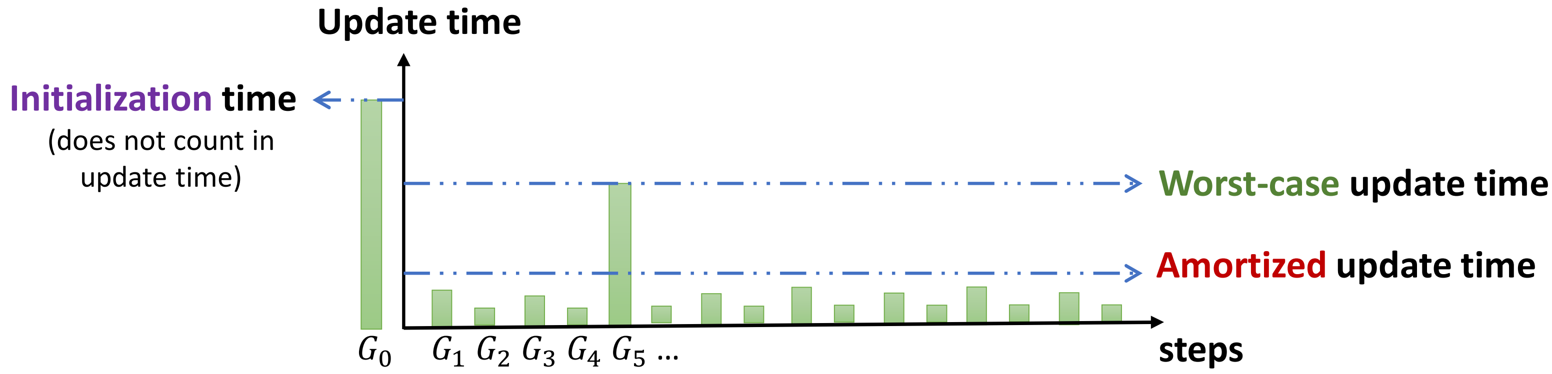
Setting:

1. Given an input graph G_0, preprocess it.
2. Then, for each time **step**,
   - Given an update or a query (generated online/on the fly),
   - update the data structure and/or answer the query.

- Update are often **insertions** and **deletions** of an edge (a vertex sometimes)
- Example of tasks
  - **Answer queries**: Is G connected? what is (s, t)-distance?
  - **Maintain objects**: minimum spanning tree, maximal matching, etc.

Terminology

- G_i = the graph after i steps
- **Update sequence**: the sequence of updates/queries
- **Update time:** time needed at each step.
  - T **worst-case** update time: ***every*** step requires $\leq T$ time.
  - T **amortized** update time: after k steps (for large enough k), the ***total time*** is $\leq kT$
- **Preprocessing/initialization time**: time to process G_0



Fully dynamic vs. Partially Dynamic

- **Fully dynamic** algorithms handle both insertions and deletions
- **Partially Dynamic**
  - **incremental** algorithms handle only insertions
  - **decremental** algorithms handle only deletions

In this talk, you will learn…

3 templates for designing dynamic graph algorithms

1. Rebuild in the background
2. Batching
3. Vertex sparsifiers

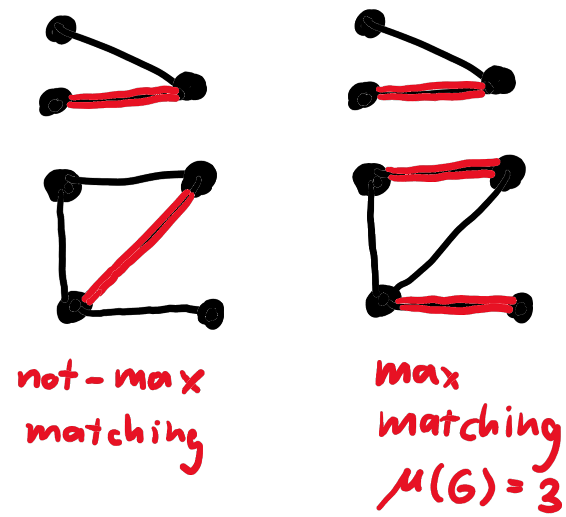
All templates are general.
They work for every problem.
Only Template 3 is specific to graphs.

For each template, I will give a complete proof
of a concrete algorithm.

Template 0 (warm-up): Rebuild

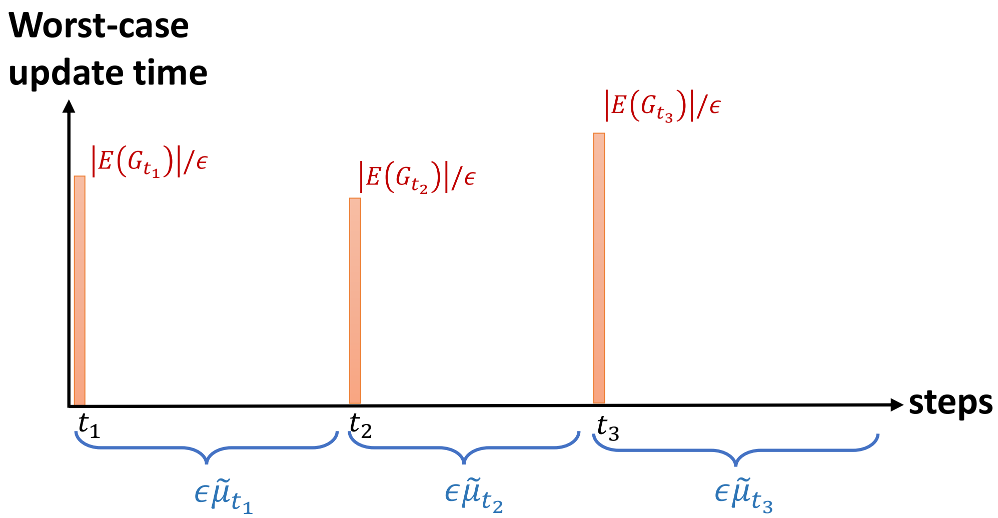
Fully dynamic $(1+\epsilon)$-approx. matching

- Def:
  - **Matching** is a set of vertex-disjoint edges
  - Given graph G, $\boldsymbol{\mu(G)}$ **= size of maximum matching**



- Problem:
  - Init: graph G with n vertices
  - Then: online sequence of edge insertions/deletions
  - **Goal: maintain $(1+\epsilon)$-approx. of $\mu(G)$**
- Algo:
  - **Trivial**: $O(|E(G)|/\epsilon) = O(n^2/\epsilon)$ update time. (recompute from scratch after edge update)
  - **Today**: $O(n/\epsilon^2)$ update time.

Algorithm

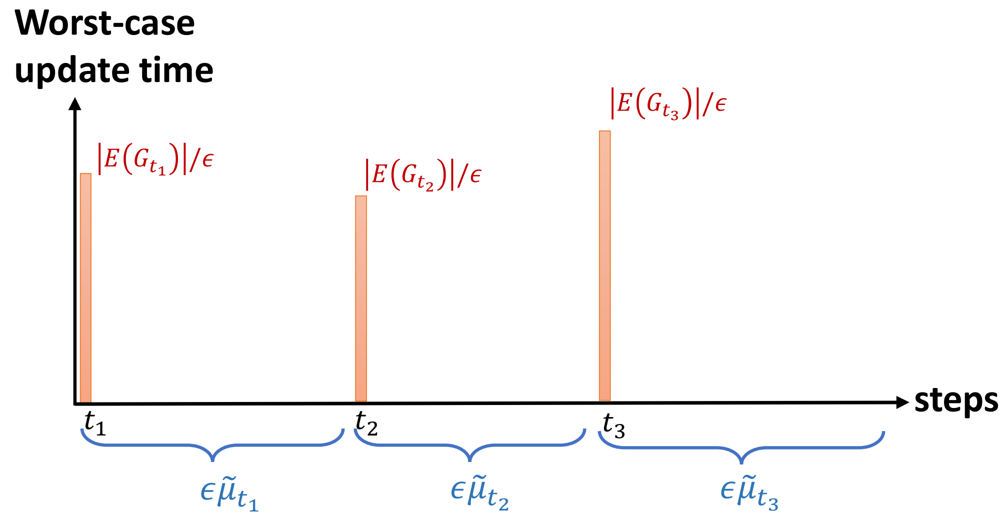


- Repeat:
  - **(Rebuild step):** $\tilde{\mu} \leftarrow (1+\epsilon)$-approx. of $\mu(G)$ computing from scratch
  - For the next $\epsilon\tilde{\mu}$ steps, just return $\tilde{\mu}$.

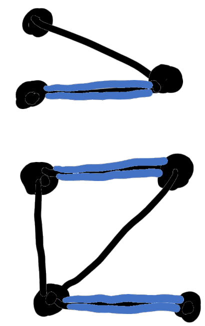
- **Correct**:
  - Each edge update may change the size of $\mu(G)$ by at most 1.
  - So, $\mu(G) = (1 \pm \epsilon)\tilde{\mu} \pm \epsilon\tilde{\mu}$ at all time.
  - That is, $\tilde{\mu}$ is always $(1 + O(\epsilon))$-approx. of $\mu(G)$

Analysis

- **Update time**:
  - $O\left(\frac{|E(G_t)|/\epsilon}{\epsilon\mu(G_t)}\right)$ amortized update time.
  - $= O(n/\epsilon^2)$. Why?

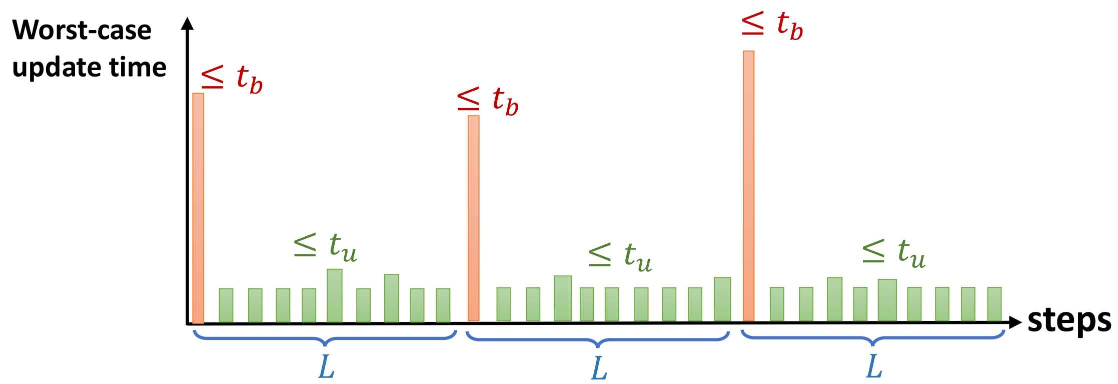


- **Claim**: $|E(G)| \leq \mu(G) \cdot 2n$
  - Let M^* be a maximum matching $|M^*| = \mu(G)$
  - **Observe**: every edge is incident to M^* (otherwise M^* is not max).
  - Deleting an edge $e = (u, v)$ in M^*
    removes at most $\deg(u) + \deg(v) \leq 2n$ edges in G
  - Repeat $\mu(G)$ times, no edge left.



Template 0: Rebuild

- Given algo $\mathcal{A}$
  - can handle L updates
  - t_b = rebuild time
  - t_u = worst-case update time
- Obtain algo $\mathcal{A}'$
  - can handle infinite updates
  - $O(\frac{t_b}{L} + t_u)$ **amortized** update time



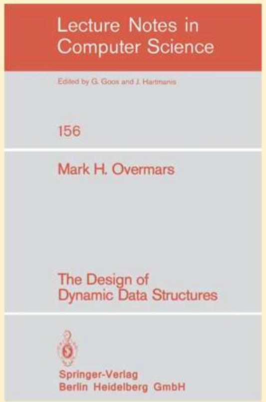
- In our case, fully dynamic $(1+\epsilon)$-matching with $O(n/\epsilon^2)$ **amortized** update time
  - $L = \Theta(\epsilon\mu(G))$, $t_b = O(\frac{E(G)}{\epsilon})$, $t_u = O(1)$

- **Next**: worst-case update time instead

Template 1:
Rebuild in the Background

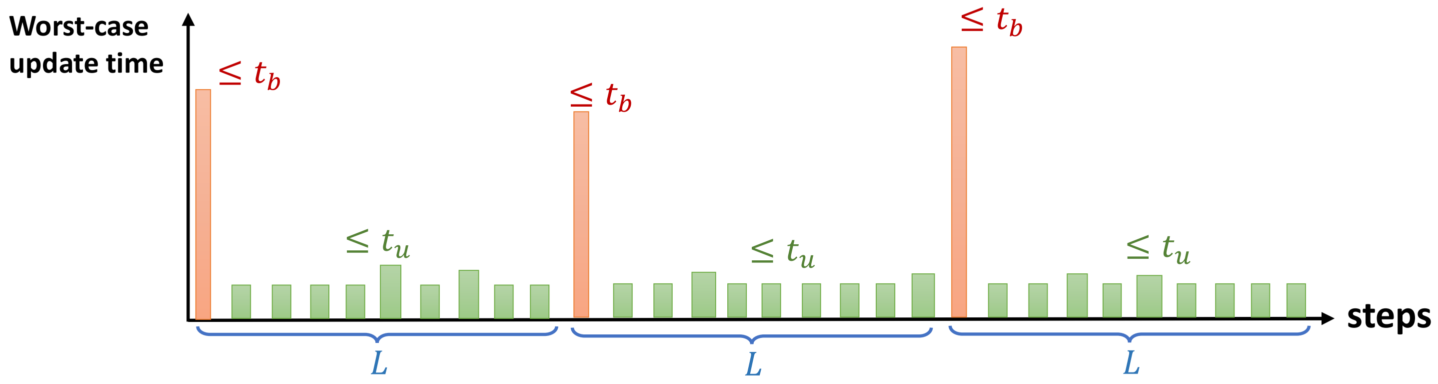
Overmars'83 **"Global Rebuilding"**

Used in many many papers.



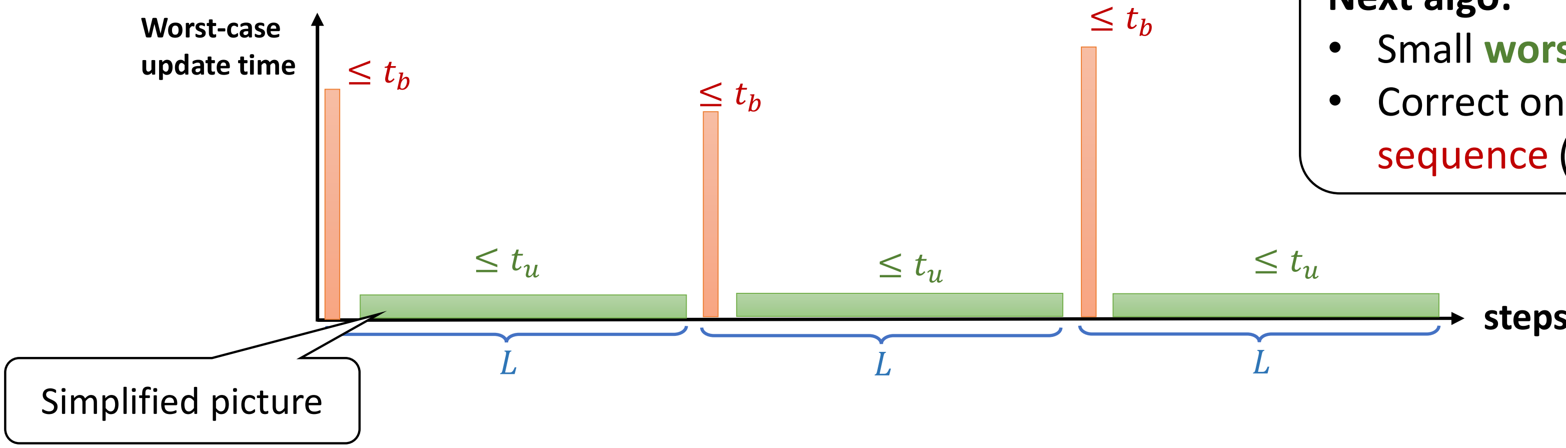
Recall

- Given algo $\mathcal{A}$
  - can handle L updates
  - t_b = rebuild time
  - t_u = worst-case update time
- algo $\mathcal{A}'$ with $O(\frac{t_b}{L} + t_u)$ amortized update time



Recall

- Given algo $\mathcal{A}$
  - can handle L updates
  - t_b = rebuild time
  - t_u = worst-case update time
- algo $\mathcal{A}'$ with $O(\frac{t_b}{L} + t_u)$ amortized update time



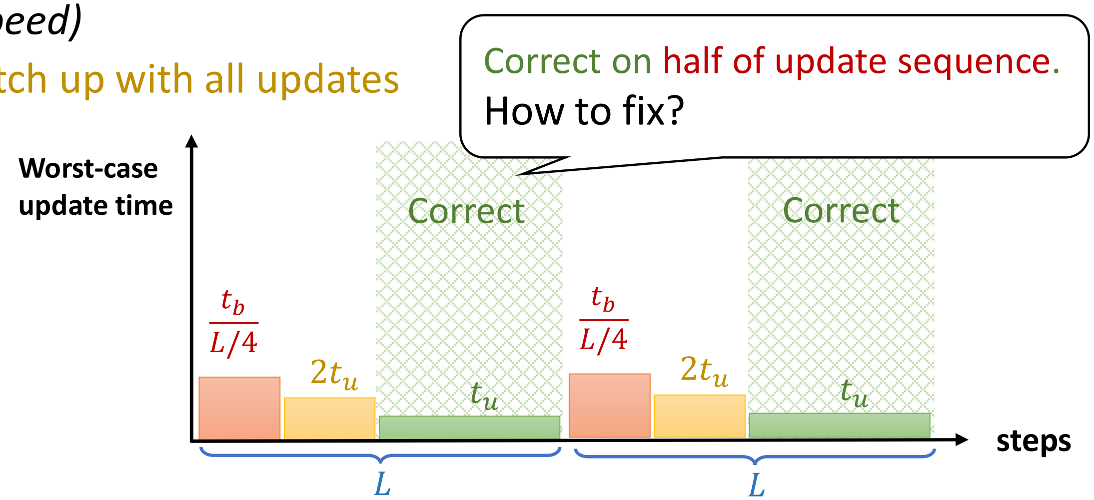
Next algo:
- Small **worst-case update time**.
- Correct on half of update sequence (this is easy to fix)

Algorithm $\mathcal{A}''$: $O(\frac{t_b}{L} + t_u)$ worst-case update time

Divide each phase of L steps from $[t_0, t_0 + L]$ into 3 periods

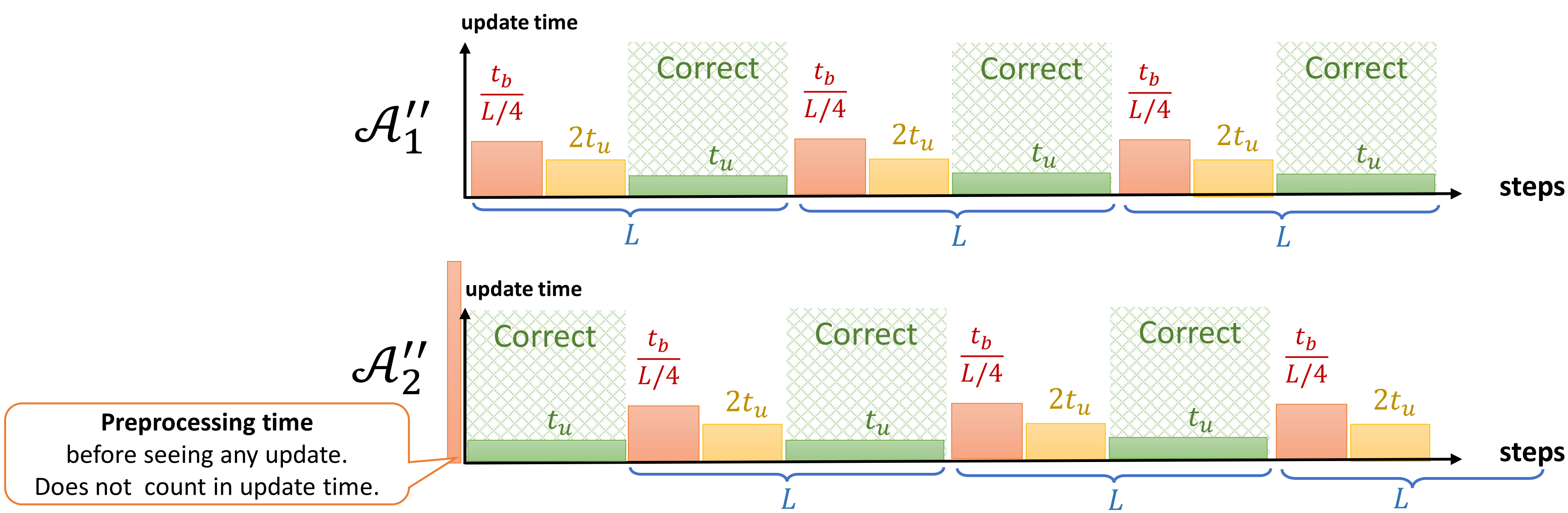
1. **(Rebuild)** first $L/4$ steps:
   - Rebuild data structure for G_{t_0} but distribute the work evenly on the period.
   - Ignore updates.
2. **(Catch-up)** next $L/4$ steps:
   - Each step, feed two updates. *(Double speed)*
   - **Observe:** At the end, data structures catch up with all updates
3. **(Active)** last $L/2$ steps:
   - Feed update normally.
   - Get correct answers in this period

Worst-case update time: $O(t_b/L + t_u)$



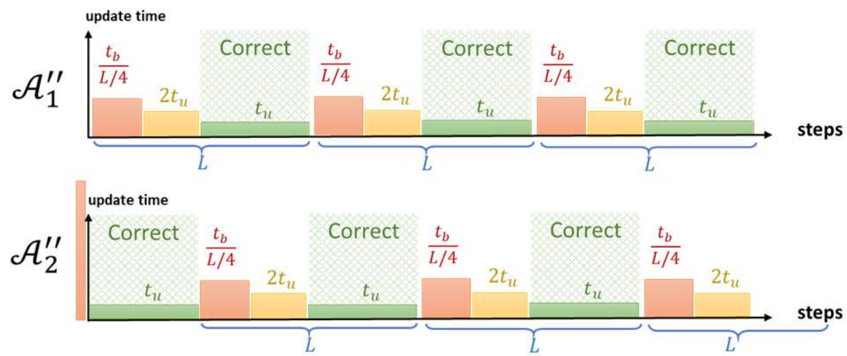
Correct answers at all steps

- Make two instances $\mathcal{A}_1''$ and $\mathcal{A}_2''$. (Increase update time by factor 2.)
  - Schedule their periods so that…
  - At every step, one of them is correct.



Conclude: Rebuild in the background

- Given algo $\mathcal{A}$
  - can handle L updates
  - t_b = rebuild time
  - t_u = worst-case update time
- Obtain algo $\mathcal{A}''$
  - $O(\frac{t_b}{L} + t_u)$ worst-case update time



- **Conclude**:
  - fully dynamic $(1+\epsilon)$-matching with $O(n/\epsilon^2)$ worst-case update time.
  - Exercise: $O(\deg_{\max}(G)/\epsilon^2)$ worst-case update time Hint: $|E(G)| \leq \mu(G) \cdot 2\deg_{\max}(G)$

Template 2:
Batching

Used in

- Dynamic MST [HK'01]
- Dynamic APSP with worst-case update time [Thorup'05] [ACK'17] [PW'20]
- Expander pruning with worst-case update time [NSW'17] [BBGNSSS'22] [JS'22]
- Dynamic DFS [BCCK'16]
- Let me know more

Decremental connectivity

- Problem:
  - Init: graph G with n vertices and m edges
  - Then: online sequence of edge deletions only
  - **Maintain: is G connected?**

- Algo:
  - Trivial: $O(m)$ worst-case update time. (BFS, for example)
  - Today: $\tilde{O}(m^{2/3})$ worst-case update time.

One-batch decremental connectivity

- Setting for one-batch algorithms (Often easier to design)
  - Init: graph G
  - Update: a **single batch** D of d edge deletions.
  - Answer: is $G' = G \setminus D$ connected?

- **Will show later**: one-batch algo $\mathcal{A}_{\text{batch}}$ with
  - Init time: $\tilde{O}(m)$
  - Update time: $\tilde{O}(d^2)$

- **Next**: how to get $O(m^{2/3})$ worst-case update time using $\mathcal{A}_{\text{batch}}$

Reduction to one-batch algorithms

- **Simple idea**: to handle update *sequence* with *one-batch* algos $\mathcal{A}_{\text{batch}}$,
  - Given the i^{th} update u_i,
    feed the **batch of first *i* updates** $(u_1, \dots, u_i)$ into $\mathcal{A}_{\text{batch}}$ (using $\tilde{O}(i^2)$ time, for us)

Recall:

Given algo $\mathcal{A}$

can handle L updates

t_b = rebuild time

t_u = worst-case update time

Obtain algo $\mathcal{A}''$

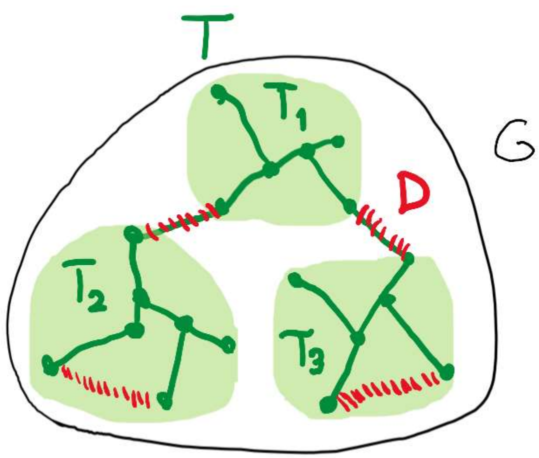
$O(\frac{t_b}{L} + t_u)$ worst-case update time

- Obtain algo $\mathcal{A}$ that
  - can handle L updates
  - rebuild time: $t_b = \tilde{O}(m)$
  - update time: $t_u = \tilde{O}(L^2)$
- Get algo $\mathcal{A}'$ with $\tilde{O}(m/L + L^2) = \tilde{O}(m^{2/3})$ worst-case update time
  - by choosing $L = m^{1/3}$

Remain to show $\mathcal{A}_{\text{batch}}$…

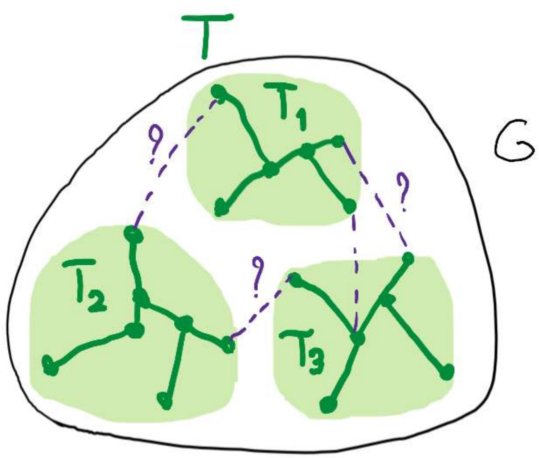
Ideas of one-batch algo $\mathcal{A}_{\mathrm{batch}}$

- **Init:** $T \leftarrow$ spanning tree of G
- **Update:** given a set D of d edge deletions,
  1. $T_1, \dots, T_{d'} \leftarrow$ connected components of $T \setminus D$. ($d' \leq d + 1$)
  2. Sufficient: find a non-tree-edge between T_i and T_j, if exists, for $i, j \in [d']$



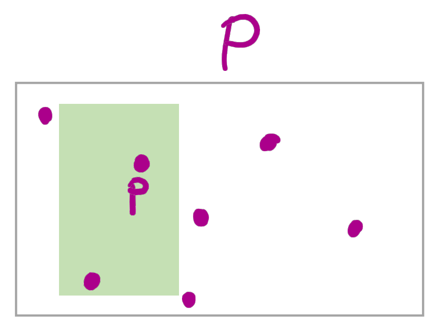
Ideas of one-batch algo $\mathcal{A}_{\text{batch}}$

- **Init:** T ← spanning tree of G
- **Update:** given a set D of d edge deletions,
  1. $T_1, \dots, T_{d'}$ ← connected components of $T \setminus D$. $(d' \le d + 1)$
  2. Sufficient: find a non-tree-edge between T_i and T_j, if exists, for $i, j \in [d']$



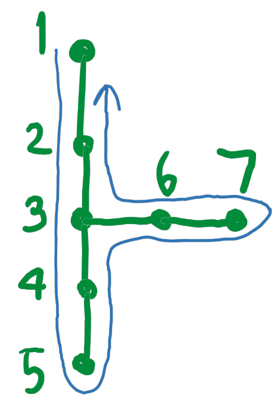
Next:
- show how to find these edges
- Need two tools:
  - Euler-tour
  - 2d-range query

Basic tools for one-batch algo $\mathcal{A}_{\mathbf{batch}}$

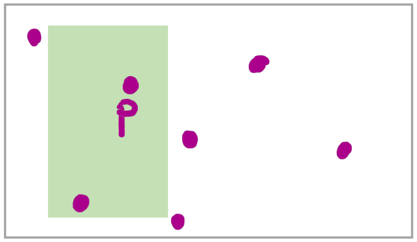


- <u>2d-range query:</u> P is a set of points in 2d.
  1. **Update:** Insert/delete a point in P in $\tilde{O}(1)$ time
  2. **Query:** Given rectangle $I_x \times I_y$, return a point $p \in P \cap (I_x \times I_y)$ in $\tilde{O}(1)$ time

- <u>Euler-tour:</u>
  - Represent any tree T as a path.
  - See example:
    - Euler tour of T is $\mathrm{Eu}(T) = (1,2,3,4,5,4,3,6,7,6,3,2)$
  - Each edge in T corresponds to 2 edges in $\mathrm{Eu}(T)$

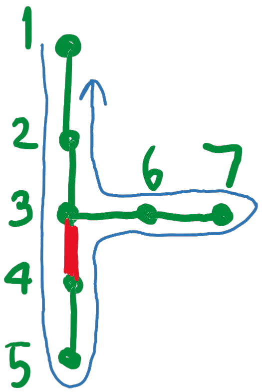


Basic tools for one-batch algo $\mathcal{A}_{\mathbf{batch}}$

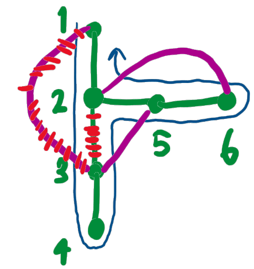


- 2d-range query: P is a set of points in 2d.
  1. **Update:** Insert/delete a point in P in $\tilde{O}(1)$ time
  2. **Query:** Given rectangle $I_x \times I_y$, return a point $p \in P \cap (I_x \times I_y)$ in $\tilde{O}(1)$ time

- Euler-tour:
  - Represent any tree T as a path.
  - See example:
    - Euler tour of T is $\mathrm{Eu}(T) = (1,2,3,4,5,4,3,6,7,6,3,2)$
  - Each edge in T corresponds to 2 edges in $\mathrm{Eu}(T)$

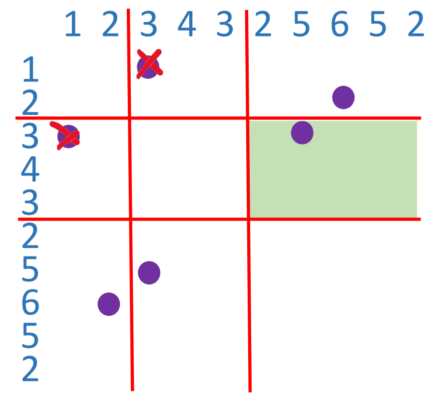


One-batch algo $\mathcal{A}_{\text{batch}}$



- **Init:**

$\tilde{O}(m)$ time
1. $T \leftarrow$ spanning tree of G
2. $\text{Eu}(T) \leftarrow$ Euler-tour of T
3. For each non-tree edge (u, v), add $(\text{minrank}_{\text{Eu}(T)}(u), \text{minrank}_{\text{Eu}(T)}(v))$ to set P



- **Update:** given a set D of d edge deletions,

$\tilde{O}(d)$ time
1. Delete points in P corresponding to deleted non-tree edges
2. $I_1, \ldots, I_{d''} \leftarrow$ intervals of $\text{Eu}(T)$ after deleting tree edges. $d'' = O(d)$

$\tilde{O}(d^2)$ time
3. Query $I_i \times I_j$ to for all $i, j \in [d'']$
→ Find a non-tree-edge between T_i and T_j for all $i, j \in [d']$

Conclude: dynamic algo from <u>one-batch</u> algo

- **Get**: one-batch algo $\mathcal{A}_{\text{batch}}$ with
  - Init time: $\tilde{O}(m)$
  - Update time: $\tilde{O}(d^2)$

- **By previous reduction**: from $\mathcal{A}_{\text{batch}}$, we get
  - Decremental connectivity algorithm with $O(m^{2/3})$ worst-case update time

Conclude: dynamic algo from <u>one-batch</u> algo

- **If we have**: one-batch algo $\mathcal{A}_{\text{batch}}$ with
  - Init time: $\tilde{O}(m)$
  - Update time: $\tilde{O}(d)$
- **By previous reduction**: from $\mathcal{A}_{\text{batch}}$, we would get
  - Decremental connectivity algorithm with $O(m^{1/2})$ update time

Exercise

Remark:

- Most **2-batch** algos generalize to the **k-batch** setting. (not for 1-batch algos)
- For example, Today's 1-batch algorithm does not work in the 2-batch setting. Why?

- Setting for <u>2-batch algorithms:</u>
  - Init: graph G
  - First update: a batch D_1 of d_1 edge deletions.
  - Second update: a batch D_2 of d_2 edge deletions.
  - Answer: is $G' = G \setminus (D_1 \cup D_2)$ connected?

Exercise:

Given 2-batch algo $\mathcal{A}_{\text{batch}}$ with

- Init time: $\tilde{O}(m)$
- First update time: $\tilde{O}(d_1)$
- Second update time: $\tilde{O}(d_2)$

1. Show a dynamic algorithm with $\tilde{O}(m^{1/3})$ worst-case update time.
2. Generalize to the k-batch setting, get $\tilde{O}(m^{1/(k+1)})$ worst-case update time

Before Template 3: Introduction to Vertex Sparsifiers

Extensively studied in both approximation and FPT (kernelization) communities.

Vertex Sparsifiers (aka Mimicking Networks)

Setting:

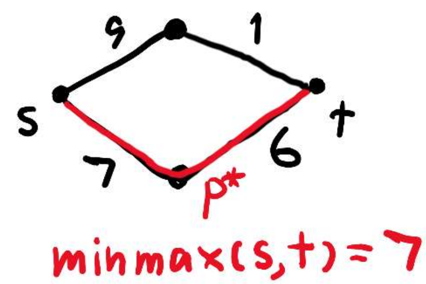
- Input: graph $G = (V, E)$, terminal set $S \subseteq V$
- Output: graph H s.t. $S \subseteq V(H)$ and
  - $|E(H)| \approx |S|$
  - H preserves information in G related to S. Write "$H \approx G$ w.r.t. S"
- H is called a **sparsifier of G w.r.t. S**.

- Example: **Vertex sparsifier for shortest paths** [CGHP**S**'20] based on [TZ'05]
  - For any k, there exists H s.t.
  - $|E(H)| = O(|S| n^{1/k})$
  - For each $u, v \in S$, $\text{dist}_H(u, v) \approx \text{dist}_G(u, v)$ up to $O(k)$ factor.

- Today: **Vertex sparsifier for minmax paths**

Minmax paths and Minimum Spanning Trees (MST)

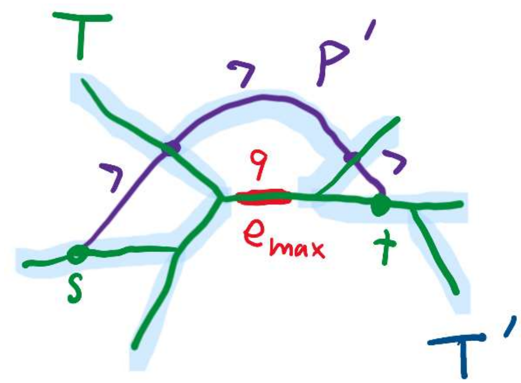
- (s,t)-minmax path P^* has minimum $\max_{e \in P^*} w(e)$ ($w(e)$ is weight of edge e)
- $\text{minmax}(s,t) \coloneqq \max_{e \in P^*} w(e)$



- Key point: **MST preserves all minmax paths**
  - T: MST of G.
  - $P_T(s,t)$: unique (s,t)-path in T

Remember this.

- <u>Claim</u>**:** $P_T(s,t)$ is (s,t)-minmax path in G for all s,t
- **Proof:** Otherwise, there is another (s,t)-path P' s.t.
  - $e_{\max} \leftarrow$ heaviest edge in $P_T(s,t)$
  - $\max_{e \in P^*} w(e) < w(e_{\max})$
- So, there is $e' \in P'$ s.t.
  - $T' = T \cup e' \setminus e_{\max}$ is a spanning tree
  - but $w(T') < w(T)$. **Contradiction.**



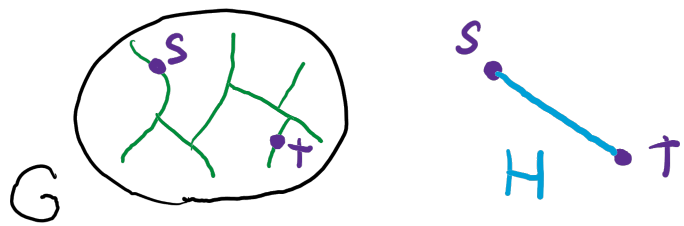
Vertex sparsifier for minmax paths

Input: graph $G = (V, E)$, terminal set $S \subseteq V$

Output: graph H

- $|E(H)| = O(|S|)$
- For all $u, v \in S$, $\mathrm{minmax}_H(u, v) = \mathrm{minmax}_G(u, v)$. (**Write $\boldsymbol{H} \equiv_{\mathbf{minmax}} \boldsymbol{G}$ w.r.t $\boldsymbol{S}$**)

Warming-up Algo:



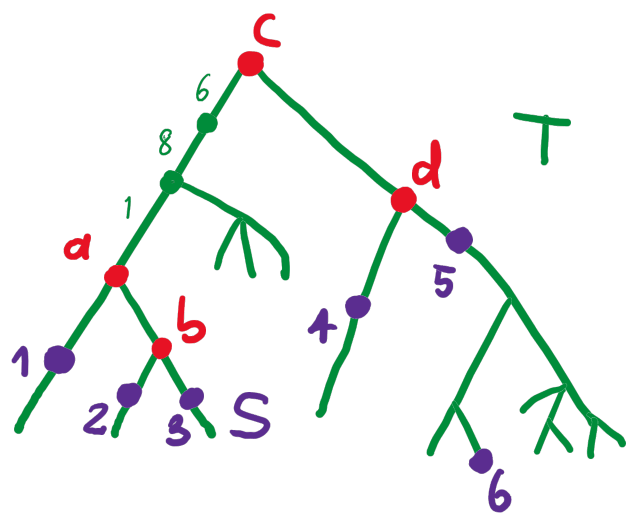
- What if $S = \{u, v\}$?
- Easy: $H \leftarrow \{(u, v)\}$ where $w_H(u, v) \leftarrow \mathrm{minmax}_G(u, v)$

Vertex sparsifier for minmax paths

Input: graph $G = (V, E)$, terminal set $S \subseteq V$

Output: H where $S \subseteq V(H)$

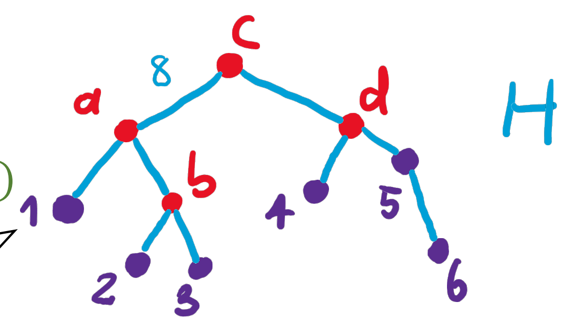
- $|E(H)| = O(|S|)$
- For all $u, v \in S$, $\mathrm{minmax}_H(u, v) = \mathrm{minmax}_G(u, v)$



Algo:

1. $T \leftarrow$ MST of G.
2. $S' \leftarrow \mathrm{LCA\_Closure}_T(S)$.
   - For all $u, v \in S'$, $LCA(u, v) \in S'$.
   - Note that $|S'| \leq 2|S|$.
3. For each "adjacent" $u, v \in S'$,
   Add (u, v) into H where $w_H(u, v) \leftarrow$ max weight in $P_T(u, v)$

Correctness: Exercise
Hint: (MST preserves all minmax paths + the warm-up case)
Time: $\tilde{O}(m)$ **Size:** $O(|S|)$

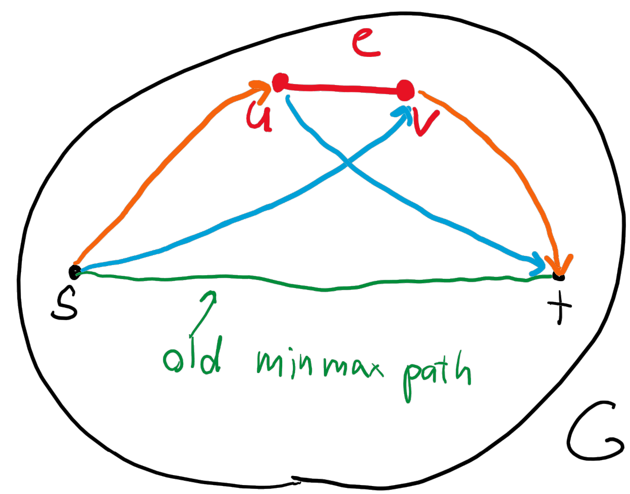


Composability

Def: The similarity relation "≈" is **composable** if

1. $H \approx G$ w.r.t. S
2. $e = (u, v)$ where $u, v \in S$,

Then, $H + e \approx G + e$ w.r.t. S



Lemma: The relation $\equiv_{\mathbf{minmax}}$ is composable.

Proof: Suppose $H \equiv_{\text{minmax}} G$ w.r.t. S. For any (s, t) where $s, t \in S$,

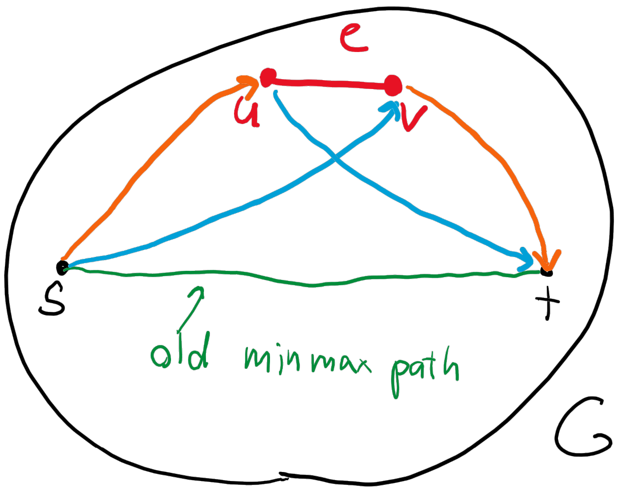
$$\text{minmax}_{G+e}(s,t) = \min\begin{cases} \text{minmax}_G(s,t) \\ \max\{\text{minmax}_G(s,u), w(u,v), \text{minmax}_G(v,t)\} \\ \max\{\text{minmax}_G(s,v), w(v,u), \text{minmax}_G(u,t)\} \end{cases}$$

Composability

Def: The similarity relation "≈" is **composable** if

1. $H \approx G$ w.r.t. S
2. $e = (u, v)$ where $u, v \in S$,

Then, $H + e \approx G + e$ w.r.t. S



Lemma: The relation $\equiv_{\mathbf{minmax}}$ is composable.

Proof: Suppose $H \equiv_{\text{minmax}} G$ w.r.t. S. For any (s, t) where $s, t \in S$,

$$\text{minmax}_{G+e}(s,t) = \min\begin{cases} \text{minmax}_G(s,t) \\ \max\{\text{minmax}_G(s,u), w(u,v), \text{minmax}_G(v,t)\} \\ \max\{\text{minmax}_G(s,v), w(v,u), \text{minmax}_G(u,t)\} \end{cases} = \min\begin{cases} \text{minmax}_H(s,t) \\ \max\{\text{minmax}_H(s,u), w(u,v), \text{minmax}_H(v,t)\} \\ \max\{\text{minmax}_H(s,v), w(v,u), \text{minmax}_H(u,t)\} \end{cases} = \text{minmax}_{H+e}(s,t)$$

Template 3:
Vertex Sparsifiers

Used in

- The template is explicit in [GHP'17, CGHP**S**'20]
- Offline dynamic algo for MST [Epp'94] and O(1)-connectivity [PSS'19, CDLKPPSV'20]. Non-offline version [N**S**W'17, JS'20]
- Dynamic effective resistance [GHP'18, DGGP'19]
- Modern max flows algorithms

Plan

1. Show reduction
   - Given a fast algorithm for vertex sparsifier for minmax paths,
   - Obtain **offline** dynamic algorithm for minmax paths

   The template work for any problem.
2. Discuss how to get **non-offline** dynamic algorithms
3. Open problems (a lot of growth, promising)

Anyway, what are **offline** algorithms?

Offline fully dynamic minmax paths

- Assume for notational convenience:
  - At step i, we are given **both** edge insert/delete (u_i, v_i) **AND** query (s_i, t_i)

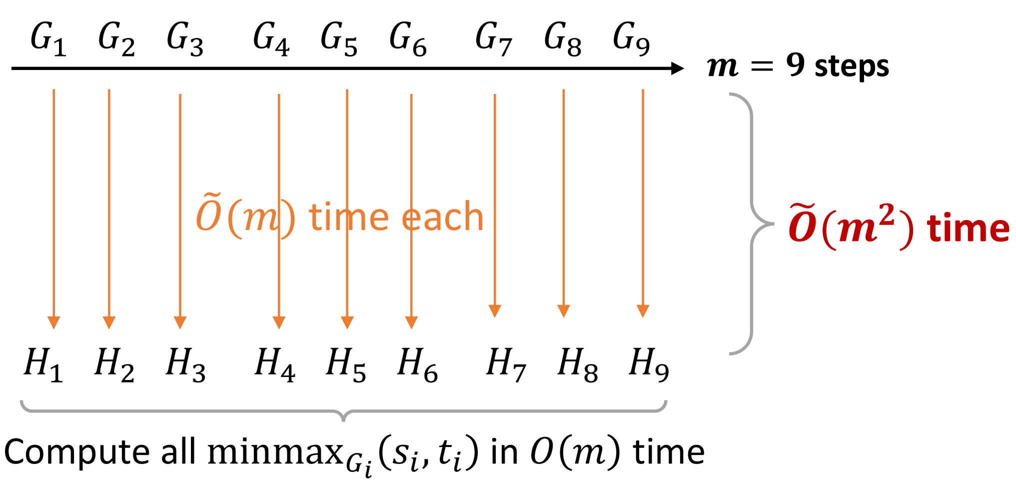
- Input: **whole** sequence of m updates/queries (Think $|E(G_i)| \leq m$ for all steps i)
  - **"Offline":** **get whole** sequence, not revealed to us one by one like before
- Output: for all i, compute $\mathrm{minmax}_{G_i}(s_i, t_i)$

- Trivial: $O(m^2)$ time.
- Today: $\tilde{O}(m^{1.5})$ time… then $\tilde{O}(m)$ time (Think $\tilde{O}(1)$ per update/query)

Offline algo: high-level idea

- **Our goal**:
  - For all i, compute $H_i \equiv G_i$ w.r.t. $\{s_i, t_i\}$. Note $|E(H_i)| = O(1)$.
- **Then**:
  - For all i, compute $\mathrm{minmax}_{G_i}(s_i, t_i) = \mathrm{minmax}_{H_i}(s_i, t_i)$ in $O(m)$ *total* time.

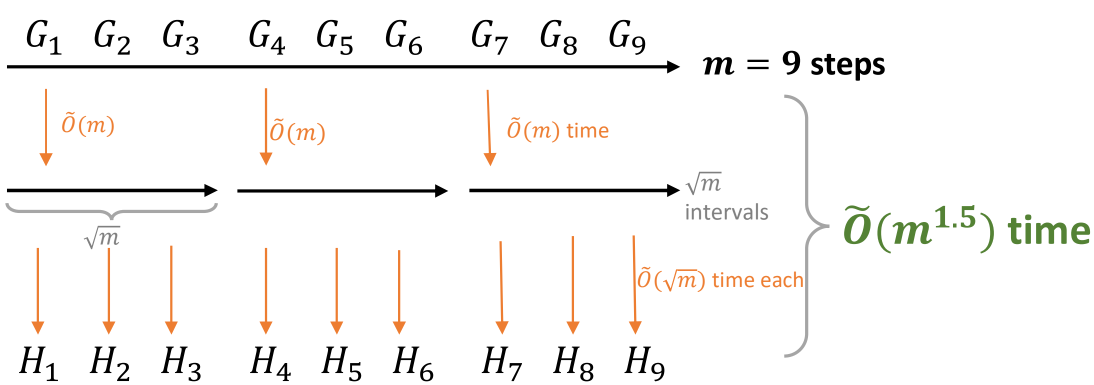
Trivial way to compute H_i



Offline algo: high-level idea

- **Our goal**:
  - For all i, compute $H_i \equiv G_i$ w.r.t. $\{s_i, t_i\}$. Note $|E(H_i)| = O(1)$.
- **Then**:
  - For all i, compute $\mathrm{minmax}_{G_i}(s_i, t_i) = \mathrm{minmax}_{H_i}(s_i, t_i)$ in $O(m)$ *total* time.

Less trivial way to compute H_i

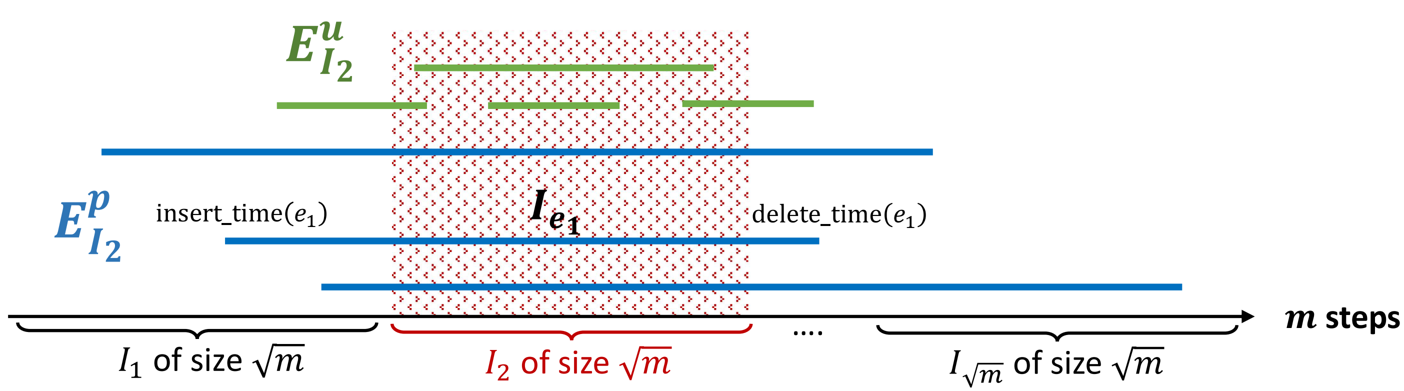


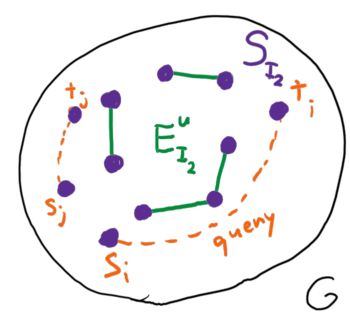
Let's see the details…

Set up notations

Divide update sequence into intervals $\mathcal{I} = \{I_1, \ldots, I_{\sqrt{m}}\}$ of length $\sqrt{m}$. Fix $I \in \mathcal{I}$.

- Interval of edge e: $I_e = [\text{insert\_time}(e), \text{delete\_time}(e)]$
- Permanent edges in I: $E_I^p = \{e \mid I \subseteq I_e\}$ $|E_I^p| \approx m$
- Updated edges in I: $E_I^u = \{e \mid I \cap I_e \neq \emptyset, I\}$ $|E_I^u| \approx \sqrt{m}$
- Terminals for I: $S_I = V(E_I^u) \cup V(Q_I)$ where $Q_I = \cup_{i \in I} \{s_i, t_i\}$. $|S_I| \approx \sqrt{m}$
    - Think: S_I are endpoints of updates/queries during interval I.



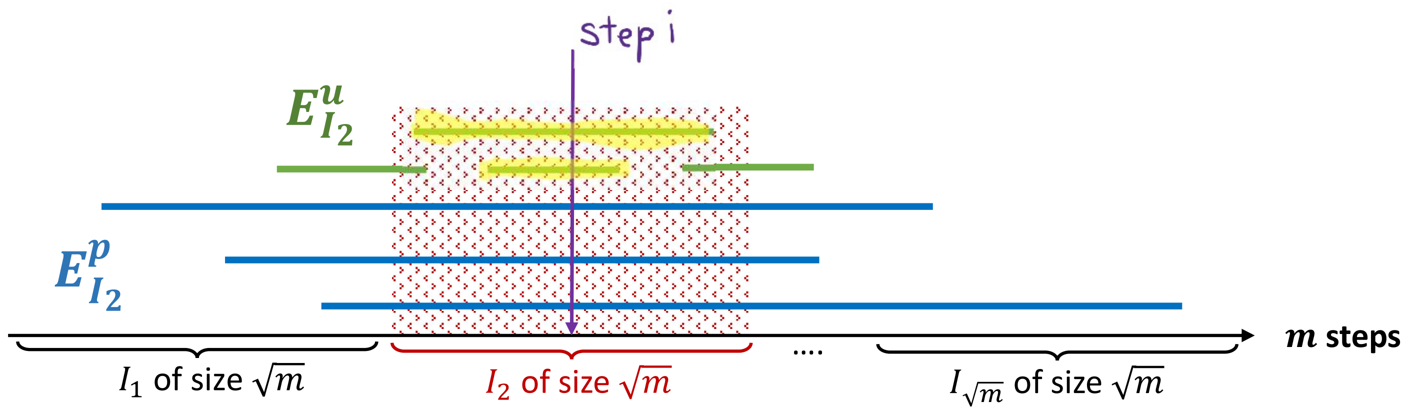


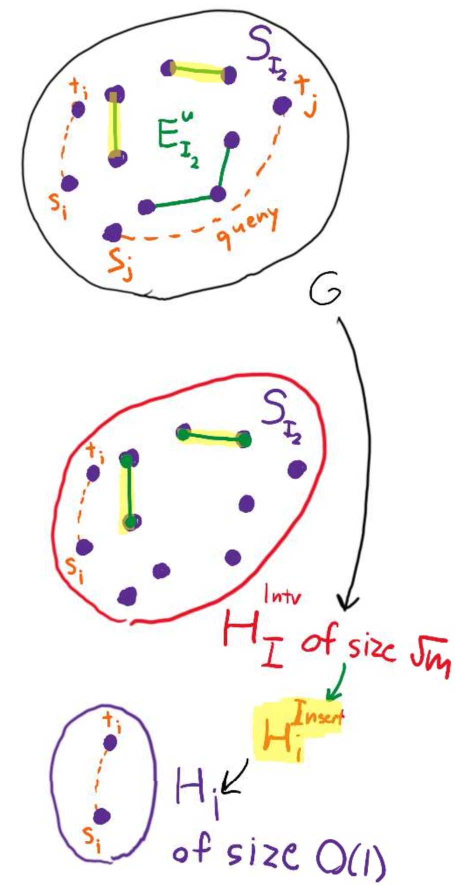
$m^{1.5}$-time offline algo

Permanent edges in I:	$E_I^p = \{e \mid I \subseteq I_e\}$
Updated edges in I:	$E_I^u = \{e \mid I \cap I_e \neq \emptyset, I\}$
Terminals for I:	$S_I = V(E_I^u) \cup V(Q_I)$

Algo: For all $I \in \mathcal{J}$,

1. Build $H_I^{\text{Interval}} \equiv E_I^p$ w.r.t. S_I
2. For each $i \in I$
   - $H_i^{\text{Insert}} \leftarrow H_I^{\text{Interval}} \cup (G_i \setminus E_I^p)$
   - Build $H_i \equiv H_i^{\text{Insert}}$ w.r.t. $\{s_i, t_i\}$





$m^{1.5}$-time offline algo

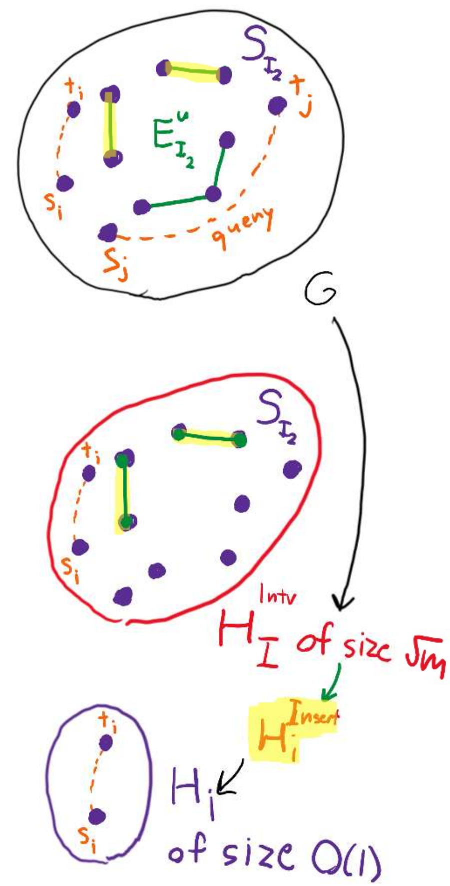
Permanent edges in I: $E_I^p = \{e \mid I \subseteq I_e\}$
Updated edges in I: $E_I^u = \{e \mid I \cap I_e \neq \emptyset, I\}$
Terminals for I: $S_I = V(E_I^u) \cup V(Q_I)$

Algo: For all $I \in \mathcal{I}$,

1. Build $H_I^{\text{Interval}} \equiv E_I^p$ w.r.t. S_I
2. For each $i \in I$
   - $H_i^{\text{Insert}} \leftarrow H_I^{\text{Interval}} \cup (G_i \setminus E_I^p)$
   - Build $H_i \equiv H_i^{\text{Insert}}$ w.r.t. $\{s_i, t_i\}$

Correctness: Want to show $H_i \equiv G_i$ w.r.t. $\{s_i, t_i\}$ for each i

- $H_i^{\text{Insert}} \equiv G_i$ w.r.t. S_I
  - $H_I^{\text{Interval}} \equiv E_I^p$ w.r.t. S_I from Step 1
  - $H_I^{\text{Interval}} \cup (G_i \setminus E_I^p) \equiv E_I^p \cup (G_i \setminus E_I^p)$ w.r.t. S_I by **composability**



$m^{1.5}$-time offline algo

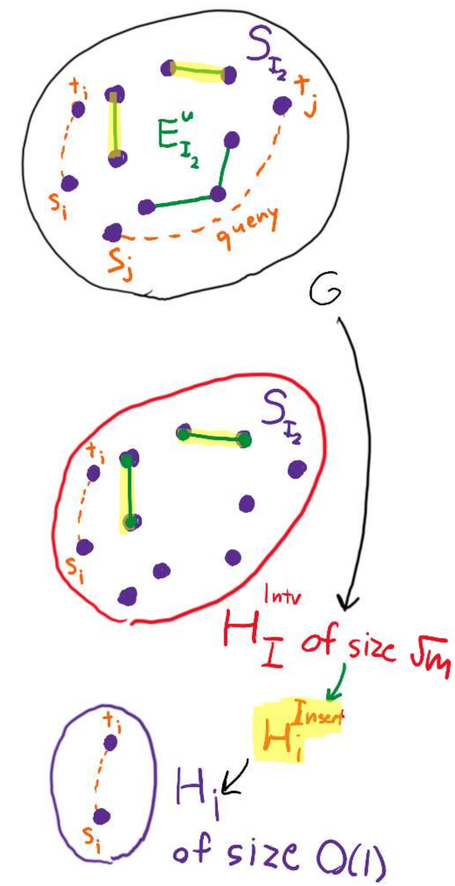
Permanent edges in I: $E_I^p = \{e \mid I \subseteq I_e\}$
Updated edges in I: $E_I^u = \{e \mid I \cap I_e \neq \emptyset, I\}$
Terminals for I: $S_I = V(E_I^u) \cup V(Q_I)$

Algo: For all $I \in \mathcal{I}$,

1. Build $H_I^{\text{Interval}} \equiv E_I^p$ w.r.t. S_I
2. For each $i \in I$
   - $H_i^{\text{Insert}} \leftarrow H_I^{\text{Interval}} \cup \left(G_i \setminus E_I^p\right)$
   - Build $H_i \equiv H_i^{\text{Insert}}$ w.r.t. $\{s_i, t_i\}$

Correctness:

- $H_i^{\text{Insert}} \equiv G_i$ w.r.t. S_I
- $H_i^{\text{Insert}} \equiv G_i$ w.r.t. $\{s_i, t_i\}$ as $S_I \supseteq \{s_i, t_i\}$.
- $H_i \equiv H_i^{\text{Insert}}$ w.r.t. $\{s_i, t_i\}$ by Step 2.2
- $H_i \equiv G_i$ w.r.t. $\{s_i, t_i\}$ **(DONE)**



$m^{1.5}$-time offline algo

Permanent edges in I: $E_I^p = \{e \mid I \subseteq I_e\}$
Updated edges in I: $E_I^u = \{e \mid I \cap I_e \neq \emptyset, I\}$
Terminals for I: $S_I = V(E_I^u) \cup V(Q_I)$

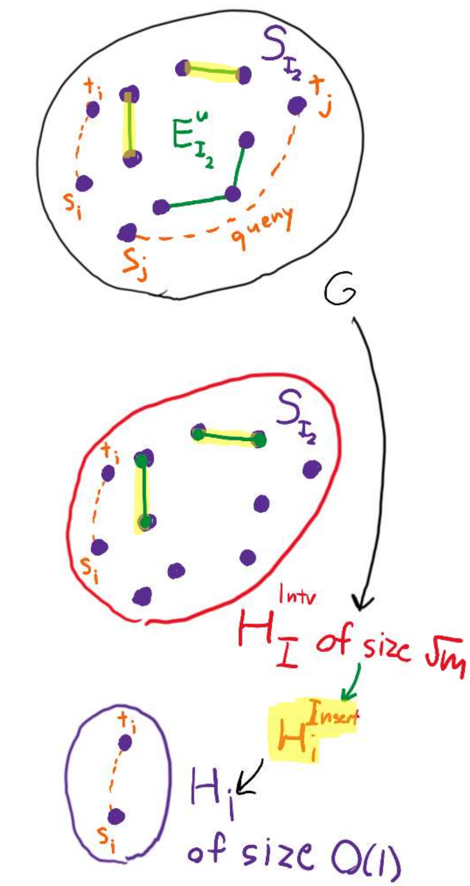
Algo: For all $I \in \mathcal{I}$, — $\sqrt{m}$ loops

1. Build $H_I^{\text{Interval}} \equiv E_I^p$ w.r.t. S_I — $\tilde{O}(m)$ time
2. For each $i \in I$ — m total loops
   - $H_i^{\text{Insert}} \leftarrow H_I^{\text{Interval}} \cup (G_i \setminus E_I^p)$ — $\tilde{O}(\sqrt{m})$ time
   - Build $H_i \equiv H_i^{\text{Insert}}$ w.r.t. $\{s_i, t_i\}$ — $\tilde{O}(\sqrt{m})$ time

Total: $\tilde{O}(m^{1.5})$

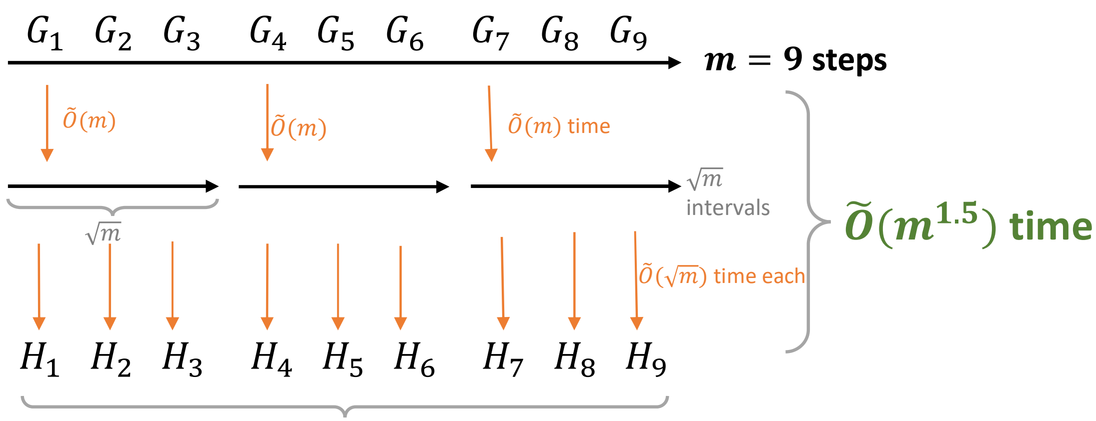
Correctness:

- $H_i^{\text{Insert}} \equiv G_i$ w.r.t. S_I
- $H_i^{\text{Insert}} \equiv G_i$ w.r.t. $\{s_i, t_i\}$ as $S_I \supseteq \{s_i, t_i\}$.
- $H_i \equiv H_i^{\text{Insert}}$ w.r.t. $\{s_i, t_i\}$ by Step 2.2
- $H_i \equiv G_i$ w.r.t. $\{s_i, t_i\}$ **(DONE)**



Summary: $\tilde{O}(m^{1.5})$–time offline algo

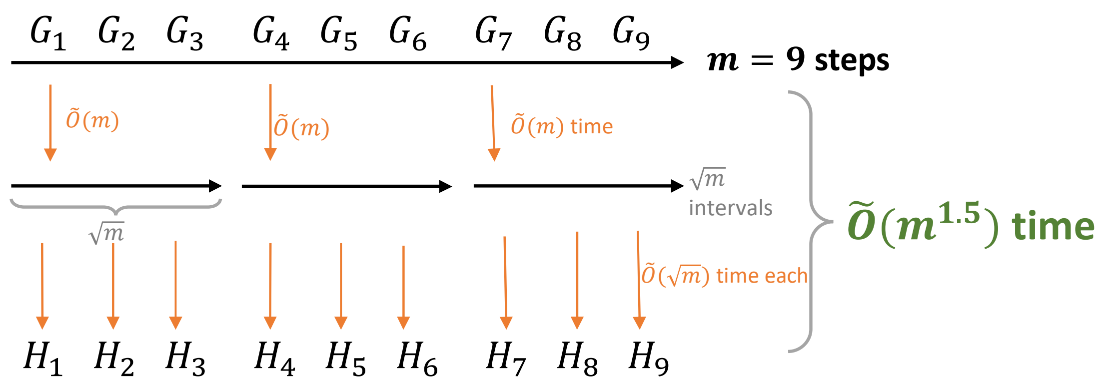
- For all i, obtain $H_i \equiv G_i$ w.r.t. $\{s_i, t_i\}$ in $\tilde{O}(m^{1.5})$ time



Then, compute all $\text{minmax}_{G_i}(s_i, t_i)$ in $O(m)$ time

$\tilde{O}(m)$–time offline algo

- 2 levels of sparsifiers → $\tilde{O}(m^{1.5})$-time algorithm



- k levels of sparsifiers → $\tilde{O}(m^{1+1/k})$-time algorithm
  - **Note:** If sparsifiers have α-approx., get α^k-approx. offline dynamic algorithms.
- For us, $\alpha = 1$ (exact). Setting $k \leftarrow \log n$, get $\tilde{O}(m)$ time

Conclude: dynamic algo from vertex sparsifier

- We saw a black-box transformation:

Fast vertex-sparsifier algorithms → Fast **offline fully dynamic** algorithms

- Can get **non-offline** dynamic algorithm too (very similar, omitted).
  - If, additionally, vertex-sparsifier algorithm can…

Many sparsifier algorithms naturally support **add-terminals**

If handle **add-terminal** operation → Fast **incremental** algorithms

If handle **add-terminal & delete** operations → Fast **fully dynamic** algorithms *omit polylog(n) in size

Open problems in Dynamic vertex sparsifiers

Promising and rich area. **Every red highlight below** shows that some aspect might be improved.

Problems	Setting	Approx	Size	Time
Minmax paths	Fully dyn [Epp'94] [NSW'17]	1	$\|S\|$	$n^{o(1)}$
Shortest paths	Incremental [TZ'05] [CGHPS'20]	k	$\|S\|n^{1/k}$	$\tilde{O}(n^{1/k})$
	Fully dyn [CGHPS'20]	$\log n$	$\beta n + \|S\|$	$\tilde{O}(1/\beta)$
c-connectivity	Static [Liu'23]	1	$\|S\|c^3$	Poly
	Fully dyn [CDLKPPSV'20] [JS'20]	1	$\|S\|c^{O(c)}$	$n^{o(1)}c^{O(c)}$
Max flow (multicommodity)	Static	$\log \|S\|$	$\|S\|$	Poly
	Incremental [RST'14] [CGHPS'20]	$\log^4 n$	$\|S\|$	$\tilde{O}(1)$
	Fully dyn unweighted [GRST'21]	$n^{o(1)}$	$\|S\|$	$n^{o(1)}$
	Fully dyn [CGHPS'20]	$\text{polylog}\, n$	$\beta n + \|S\|$	$\tilde{O}(1/\beta)$
Effective resistance	Static [KS'16] [DKPRS'16]	$1+\epsilon$	$\|S\|$	$\tilde{O}(m)$
	Fully dyn [DGGP'19] [BGJLLPS'21]	$1+\epsilon$	$\beta m + \|S\|$	$\tilde{O}(1/\beta^2)$
Low stretch trees	Fully dyn [CKLPPS'22]	$\log n$	m/k	$kn^{o(1)}$

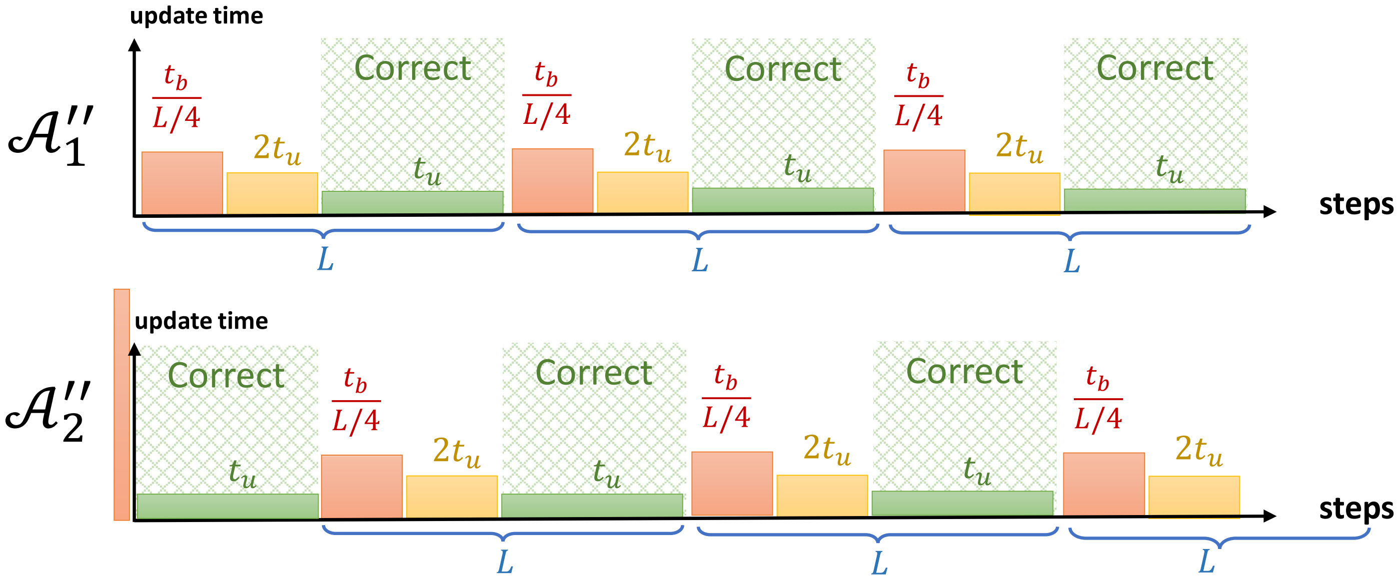
Conclusion

You have learned

- 3 templates for designing dynamic graph algorithms
    1. Rebuild in the background
    2. Batching
    3. Vertex sparsifiers

- Along the way, many terminology in the area
    - worst-case vs. amortized update time
    - fully dynamic vs. partially dynamic (incremental/decremental)
    - one-batch algorithms (a.k.a. sensitivity oracles, emergency algorithms)
    - offline dynamic algorithms

Template 1: Rebuild in the background



Template 2: Batching

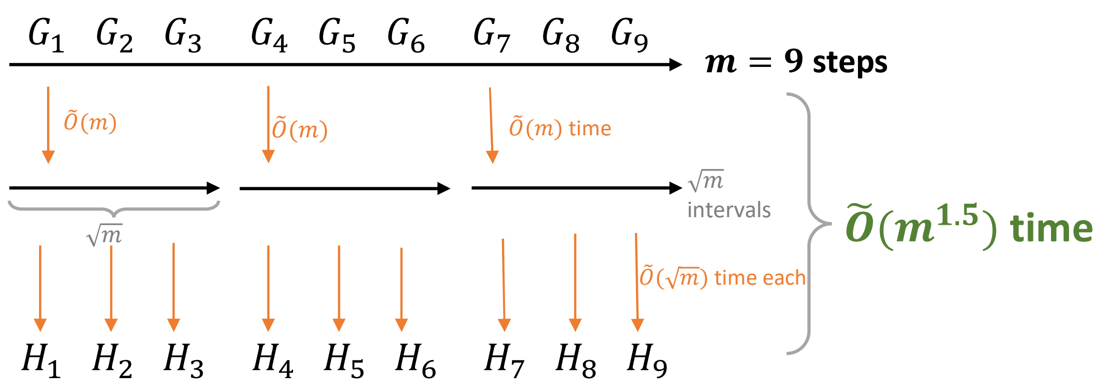
Often easy to design

<u>One-batch</u> algorithms + rebuild in the background

It works for k-batch algorithms too [NSW'17] [BBGNSSS'22] [JS'22]

<u>Dynamic</u> algorithms

Template 3: Vertex sparsifiers



Learn more templates

1. **(Recent and promising):** Optimization methods for dynamic algos.
   - Static solutions robust against update
     (congestion balancing [BG**S**'20, '21], entropy-regularized solutions [JJST'22])
   - Dynamic Multiplicative Weight Update [Gupta'14] [BK**S**'22] [BBL**S**'23]
   - Dynamic Interior Point Methods [BLS'22]
2. Given incremental algo → get offline fully dynamic algo [PR'22]
3. Given decremental algo → get fully dynamic algo
   - Problem-specific: Connectivity, MST, APSP
4. Any "decomposable" problems → get fully dynamic algo [Overmars' book]
   - E.g. dynamic range searching, quad-tree, other geometric objects

Other Generic Techniques in Dynamic Graphs

- Expander decomposition
  - Used for dynamic connectivity, shortest paths in both undirected and directed graphs
  - My tutorial: Part 1 and 2
  - My course on using expanders for fast algorithm (updated version in 1-2 months!)

- Edge-degree constrained subgraph (EDCS):
  - key objects for dynamic matching
  - Aaron's tutorial

- Randomized Greedy:
  - General approach for both dynamic maximal matching and maximal independent set
  - Soheil's tutorial

Thank you!