# How to go from partial to full retroactivity in detail

Cristina G. Fernandes, Felipe C. de Noronha

IME-USP – Brazil

LAGOS 25 – November 10-14, 2025

# What is a spanning tree?

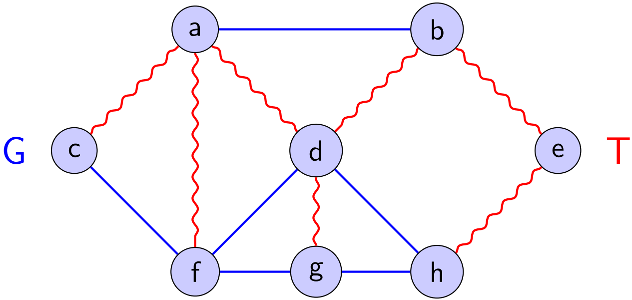
- Let $G = (V, E)$ be a connected graph
- **Spanning tree:** A tree with all vertices of $G$

# What is a spanning tree?

- Let $G = (V, E)$ be a connected graph
- **Spanning tree:** A tree with all vertices of $G$
- **Properties:**
  - Connected (path between any two vertices)
  - Acyclic (no cycles)
  - Contains exactly $n - 1$ edges for $n$ vertices

# What is a spanning tree?

- Let $G = (V, E)$ be a connected graph
- **Spanning tree:** A tree with all vertices of $G$
- **Properties:**
  - Connected (path between any two vertices)
  - Acyclic (no cycles)
  - Contains exactly $n - 1$ edges for $n$ vertices



# Minimum Spanning Tree and Forest

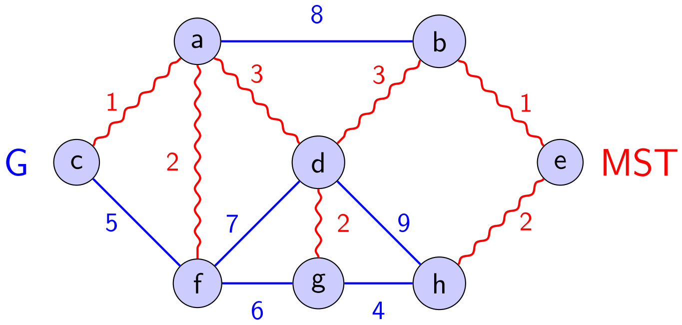
- **Minimum Spanning Tree (MST):** spanning tree in a weighted graph with minimum total cost

# Minimum Spanning Tree and Forest

- **Minimum Spanning Tree (MST):** spanning tree in a weighted graph with minimum total cost
- **Minimum Spanning Forest (MSF):** generalization for disconnected graphs

# Minimum Spanning Tree and Forest

- **Minimum Spanning Tree (MST):** spanning tree in a weighted graph with minimum total cost
- **Minimum Spanning Forest (MSF):** generalization for disconnected graphs



# Incremental MSF problem

- **Problem:** Keep track of an MSF in a graph that grows over time
- Graph starts empty, edges are added one by one

# Incremental MSF problem

- **Problem:** Keep track of an MSF in a graph that grows over time
- Graph starts empty, edges are added one by one
- **Operations:**
  - `add_edge`$(u, v, w)$: add edge with cost $w$ between vertices $u$ and $v$
  - `get_msf`$()$: return a list with the edges of an MSF of $G$

# Incremental MSF problem

- **Problem:** Keep track of an MSF in a graph that grows over time
- Graph starts empty, edges are added one by one
- **Operations:**
  - `add_edge`$(u, v, w)$: add edge with cost $w$ between vertices $u$ and $v$
  - `get_msf`$()$: return a list with the edges of an MSF of $G$
- **Solution:** Frederickson (1983) using link-cut trees

# Frederickson's link-cut tree solution

- **Key insight:** Use link-cut trees to maintain MSF dynamically

# Frederickson's link-cut tree solution

- **Key insight:** Use link-cut trees to maintain MSF dynamically
- **Link-cut tree operations:**
  - `find_max`$(u, v)$: $\mathcal{O}(\log n)$ amortized
  - `link`$(u, v, w)$: $\mathcal{O}(\log n)$ amortized
  - `cut`$(u, v)$: $\mathcal{O}(\log n)$ amortized
  - `is_connected`$(u, v)$: $\mathcal{O}(\log n)$ amortized

# Frederickson's link-cut tree solution

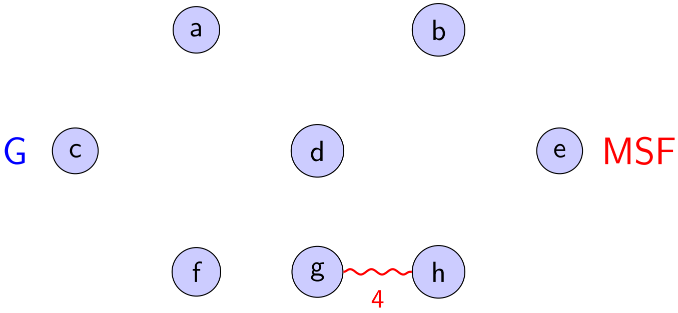
- **Key insight:** Use link-cut trees to maintain MSF dynamically
- **Link-cut tree operations:**
  - `find_max`$(u, v)$: $\mathcal{O}(\log n)$ amortized
  - `link`$(u, v, w)$: $\mathcal{O}(\log n)$ amortized
  - `cut`$(u, v)$: $\mathcal{O}(\log n)$ amortized
  - `is_connected`$(u, v)$: $\mathcal{O}(\log n)$ amortized
- **Algorithm for adding edge $(u, v, w)$:**
  1. Check if $u$ and $v$ are in connected in the same component
  2. If not: add edge $(u, v, w)$ to forest
  3. If yes: find the edge with maximum cost on the $u$-$v$ path
  4. If $w <$ maximum cost: replace maximum cost edge with $(u, v, w)$

# Frederickson's link-cut tree solution

- **Key insight:** Use link-cut trees to maintain MSF dynamically
- **Link-cut tree operations:**
  - `find_max`$(u, v)$: $\mathcal{O}(\log n)$ amortized
  - `link`$(u, v, w)$: $\mathcal{O}(\log n)$ amortized
  - `cut`$(u, v)$: $\mathcal{O}(\log n)$ amortized
  - `is_connected`$(u, v)$: $\mathcal{O}(\log n)$ amortized
- **Algorithm for adding edge $(u, v, w)$:**
  1. Check if $u$ and $v$ are in connected in the same component
  2. If not: add edge $(u, v, w)$ to forest
  3. If yes: find the edge with maximum cost on the $u$-$v$ path
  4. If $w <$ maximum cost: replace maximum cost edge with $(u, v, w)$
- **Total cost:** Amortized $\mathcal{O}(\log n)$ per edge addition

# Incremental MSF example - Step 1

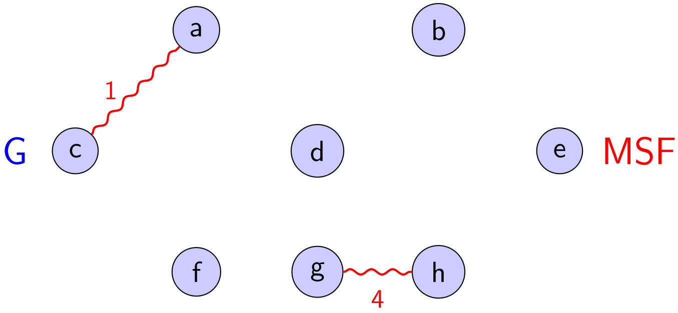
- **add_edge(g, h, 4):** Add edge $(g, h)$ with cost 4




- **MSF:** {g-h}

# Incremental MSF example - Step 2

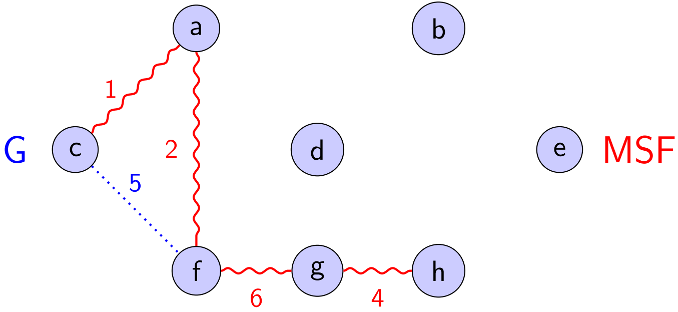
- **add_edge(c, a, 1):** Add edge with cost 1




- **MSF:** {g-h, c-a}

# Incremental MSF example - Step 3 (Cycle Check)

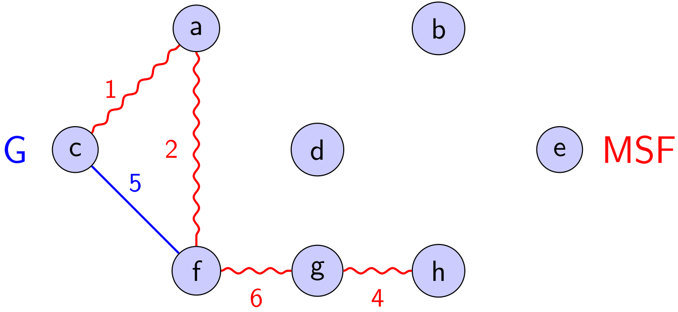
- **add_edge(c, f, 5):** Add edge with cost 5




- **MSF:** {g-h, c-a, f-g, a-f}

# Incremental MSF example - Step 3 (Result)

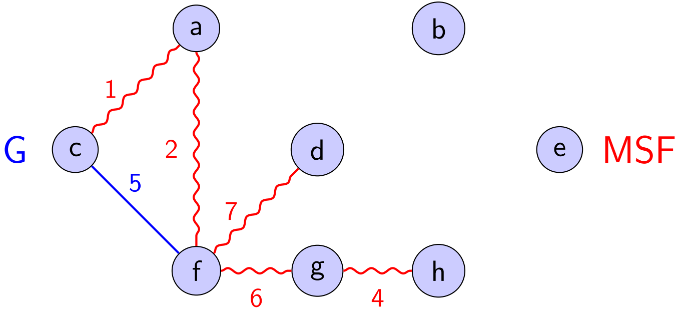
- **add_edge(c, f, 5):** Edge cost is greater than max cost (2), not added




- **MSF:** {g-h, c-a, f-g, a-f}

# Incremental MSF example - Step 4

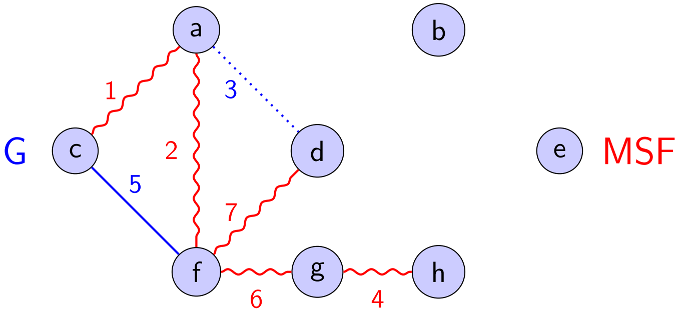
- **add_edge(f, d, 7):** Add edge with cost 7




- **MSF:** {g-h, c-a, f-g, a-f, f-d}

# Incremental MSF example - Step 5 (Cycle Check)

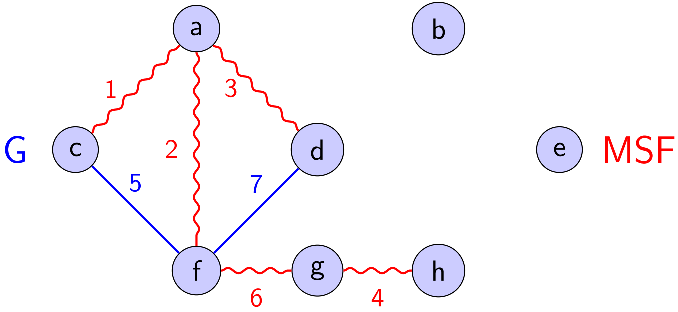
- **add_edge(a, d, 3):** Add edge with cost 3




- **MSF:** {g-h, c-a, f-g, a-f, f-d}

# Incremental MSF example - Step 5 (Result)

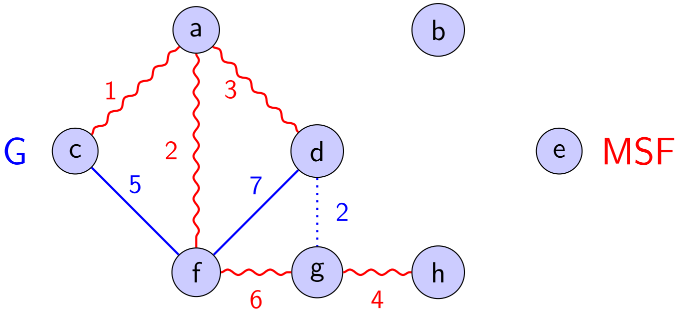
- **add_edge(a, d, 3):** Cost 3 ¡ max cost 7, swap edges




- **MSF:** {g-h, c-a, f-g, a-f, a-d}

# Incremental MSF example - Step 6 (Cycle Check)

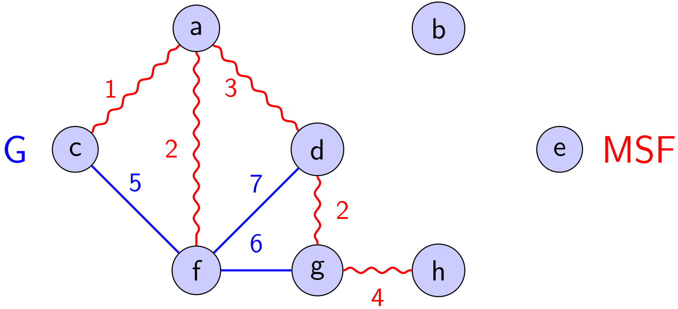
- **add_edge(d, g, 2):** Add edge with cost 2




- **MSF:** {g-h, c-a, f-g, a-f, a-d}

# Incremental MSF example - Step 6 (Result)

- **add_edge(d, g, 2):** Cost 2 ¡ max cost 6, swap edges



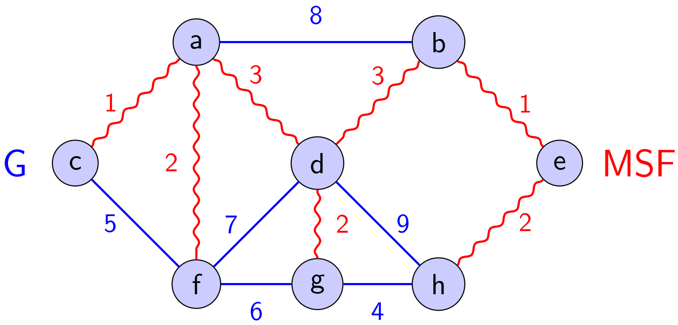

- **MSF:** {g-h, c-a, a-f, a-d, d-g}

# Incremental MSF example - Step 7 (Final Result)

- **Continue adding edges...**

# Incremental MSF example - Step 7 (Final Result)

- **Continue adding edges...**
- **Final MSF:** Minimum spanning forest with optimal cost




- **Solution:** Frederickson (1983) using link-cut trees

# What is retroactivity?

- **Problem:** Data structures usually support updates and queries
- The order of updates affects the state of the data structure

# What is retroactivity?

- **Problem:** Data structures usually support updates and queries
- The order of updates affects the state of the data structure
- **Retroactivity:** Manipulate the sequence of updates

# What is retroactivity?

- **Problem:** Data structures usually support updates and queries
- The order of updates affects the state of the data structure
- **Retroactivity:** Manipulate the sequence of updates
- **Operations:**
  - Insert update at time $t$ (possibly in the past)
  - Remove update at time $t$
  - Query at time $t$ (not just present)

# Partial vs Full retroactivity

## Fully Retroactive

- Queries at **any** time $t$
- Insert/remove updates at any time

# Partial vs Full retroactivity

## Fully Retroactive

- Queries at **any** time $t$
- Insert/remove updates at any time

## Partially Retroactive

- Queries only on **current** state
- Insert/remove updates at any time

# Partial vs Full retroactivity

### Fully Retroactive

- Queries at **any** time $t$
- Insert/remove updates at any time

### Partially Retroactive

- Queries only on **current** state
- Insert/remove updates at any time

### Semi-Retroactive

- Queries at **any** time $t$
- Insert updates at any time
- **No removal** of updates

# The challenge

**Challenge**

How to transform partial $\rightarrow$ full retroactivity?

# The challenge

**Challenge**

How to transform partial $\rightarrow$ full retroactivity?

- **Problem:** Need to support queries at any time $t$
- **Solution approach:** Square-root decomposition

# The challenge

**Challenge**

How to transform partial $\rightarrow$ full retroactivity?

- **Problem:** Need to support queries at any time $t$
- **Solution approach:** Square-root decomposition
- **Key insight:** Keep checkpoints with data structure states
- **Implementation:** Demaine, Iacono & Langerman (2007)

# Demaine, Iacono & Langerman's solution

**Theorem (Theorem 05)**

*Any partially retroactive data structure can be transformed into a fully retroactive one with:*

- $\mathcal{O}(\sqrt{m})$ *slowdown per operation*
- $\mathcal{O}(m)$ *space usage*
- **Requirement:** *Need persistent version of the data structure*

*Where m is the number of updates.*

# Demaine, Iacono & Langerman's solution

## Theorem (Theorem 05)

*Any partially retroactive data structure can be transformed into a fully retroactive one with:*

- $\mathcal{O}(\sqrt{m})$ *slowdown per operation*
- $\mathcal{O}(m)$ *space usage*
- **Requirement:** *Need persistent version of the data structure*

*Where m is the number of updates.*

- **Key idea:** Square-root decomposition
- Keep $\sqrt{m}$ checkpoints with data structure states

# Demaine, Iacono & Langerman's solution

**Theorem (Theorem 05)**

*Any partially retroactive data structure can be transformed into a fully retroactive one with:*

- $\mathcal{O}(\sqrt{m})$ *slowdown per operation*
- $\mathcal{O}(m)$ *space usage*
- **Requirement:** *Need persistent version of the data structure*

*Where m is the number of updates.*

- **Key idea:** Square-root decomposition
- Keep $\sqrt{m}$ checkpoints with data structure states
- **Query at time $t$:**
  1. Find closest checkpoint before $t$
  2. Apply updates from checkpoint to $t$
  3. Answer query, then rollback

# The space problem

- **Naive approach:** Keep $\sqrt{m}$ independent copies
- Space usage: $\Theta(m\sqrt{m})$

# The space problem

- **Naive approach:** Keep $\sqrt{m}$ independent copies
- Space usage: $\Theta(m\sqrt{m})$
- **Demaine et al. solution:** Use persistent data structures
- Space usage: $\mathcal{O}(m)$

# The space problem

- **Naive approach:** Keep $\sqrt{m}$ independent copies
- Space usage: $\Theta(m\sqrt{m})$
- **Demaine et al. solution:** Use persistent data structures
- Space usage: $\mathcal{O}(m)$

**Problem**

What if we don't have or don't want to use persistent data structures?

# The space problem

- **Naive approach:** Keep $\sqrt{m}$ independent copies
- Space usage: $\Theta(m\sqrt{m})$
- **Demaine et al. solution:** Use persistent data structures
- Space usage: $\mathcal{O}(m)$

### Problem

What if we don't have or don't want to use persistent data structures?

### Our contribution

Simple rebuilding strategy without persistent data structures

- Same time complexity: $\mathcal{O}(\sqrt{m})$ per operation
- Space usage: $\Theta(m\sqrt{m})$

# Starting point

- **Junior & Seabra's solution:** Semi-retroactive incremental MSF
- **Operations:**
  - `add_edge`$(u, v, w, t)$: add edge at time $t$
  - `get_msf`$(t)$: get MSF at time $t$

# Starting point

- **Junior & Seabra's solution:** Semi-retroactive incremental MSF
- **Operations:**
  - `add_edge`$(u, v, w, t)$: add edge at time $t$
  - `get_msf`$(t)$: get MSF at time $t$
- **Implementation:** Square-root decomposition
- **Checkpoints:** $t_i = i\sqrt{m}$ for $i = 1, \ldots, \sqrt{m}$

# Starting point

- **Junior & Seabra's solution:** Semi-retroactive incremental MSF
- **Operations:**
  - `add_edge`$(u, v, w, t)$: add edge at time $t$
  - `get_msf`$(t)$: get MSF at time $t$
- **Implementation:** Square-root decomposition
- **Checkpoints:** $t_i = i\sqrt{m}$ for $i = 1, \ldots, \sqrt{m}$
- **Data structures:** $D_i$ contains edges before time $t_i$
- **Time:** $\mathcal{O}(\sqrt{m} \log n)$ per operation

# Limitations → Key Insight

**Problems with the existing static approach**

- **Fixed $m$:** Requires knowing the maximum sequence length (**m**) beforehand. Meaning that it cannot handle arbitrary growth or dynamic operation counts.

# Limitations → Key Insight

## Problems with the existing static approach

- **Fixed** $m$**:** Requires knowing the maximum sequence length (**m**) beforehand. Meaning that it cannot handle arbitrary growth or dynamic operation counts.

## Our dynamic goal and solution

**Goal:** Remove the **Fixed m** dependency while preserving time complexity.

- **Key Insight:** Introduce a dynamic rebuilding process to handle arbitrary growth.
- **Challenge:** Rebuilding $\sqrt{\mathbf{m}}$ checkpoints must be fast, avoiding complex persistent data structures.
- **Solution:** Reuse the existing data structures to efficiently reconstruct new checkpoints.

# Our solution - Rebuilding strategy

- **Key idea:** Reuse existing data structures during rebuilding
- **Rebuilding moments:** When $m = k^2$ (perfect square)

# Our solution - Rebuilding strategy

- **Key idea:** Reuse existing data structures during rebuilding
- **Rebuilding moments:** When $m = k^2$ (perfect square)
- **Strategy:**
  1. Create new empty structures $D'_0, D'_1$
  2. Reuse $D_i \to D'_{i+2}$ for $i = 0, \ldots, k-1$
  3. Apply missing updates to each $D'_i$

# Our solution - Rebuilding strategy

- **Key idea:** Reuse existing data structures during rebuilding
- **Rebuilding moments:** When $m = k^2$ (perfect square)
- **Strategy:**
  1. Create new empty structures $D'_0, D'_1$
  2. Reuse $D_i \to D'_{i+2}$ for $i = 0, \ldots, k-1$
  3. Apply missing updates to each $D'_i$

### Key Lemma

Every update in $D_i$ is within the first $(i+2)(k+1)$ updates in the new sequence.

# Our solution - Rebuilding strategy

- **Key idea:** Reuse existing data structures during rebuilding
- **Rebuilding moments:** When $m = k^2$ (perfect square)
- **Strategy:**
  1. Create new empty structures $D'_0, D'_1$
  2. Reuse $D_i \to D'_{i+2}$ for $i = 0, \ldots, k-1$
  3. Apply missing updates to each $D'_i$

**Key Lemma**

Every update in $D_i$ is within the first $(i+2)(k+1)$ updates in the new sequence.

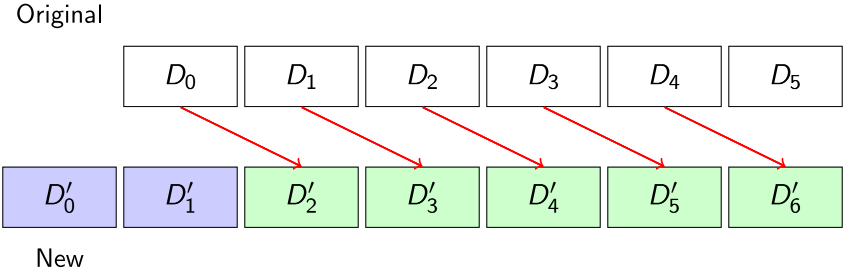
- **Time per rebuilding:** $\mathcal{O}(m \log n)$
- **Amortized cost:** $\mathcal{O}(\sqrt{m} \log n)$ per operation

# Rebuilding algorithm

1. $D'_0 \leftarrow \text{NEWINCREMENTALMSF}()$
2. $D'_1 \leftarrow \text{NEWINCREMENTALMSF}()$
3. For $i = 2$ to $k+1$: $D'_i \leftarrow D_{i-2}$ $\quad \triangleright$ reuse existing
4. For $i = 1$ to $k+1$:
   - $p \leftarrow \text{KTH}(S, i(k+1))$ $\quad \triangleright$ $i(k+1)$th item in the sequence of updates
   - $t'_i \leftarrow p.\text{time}$
   - $\text{ADDEDGES}(S, t_{i-2}, t'_i, D'_i)$
5. Return $k+1, D', t'$

# Rebuilding algorithm

1. $D'_0 \leftarrow \textsc{newIncrementalMSF}()$
2. $D'_1 \leftarrow \textsc{newIncrementalMSF}()$
3. For $i = 2$ to $k+1$: $D'_i \leftarrow D_{i-2}$ $\qquad\triangleright$ reuse existing
4. For $i = 1$ to $k+1$:
   - $p \leftarrow \textsc{kth}(S, i(k+1))$ $\qquad\triangleright$ $i(k+1)$th item in the sequence of updates
   - $t'_i \leftarrow p.\text{time}$
   - $\textsc{addEdges}(S, t_{i-2}, t'_i, D'_i)$
5. Return $k+1, D', t'$




$$D_i \to D'_{i+2}$$

# Results

## Our contribution

- **General transformation:** Partial $\rightarrow$ Full retroactivity
- **No persistent data structures needed**
- **Same time complexity:** $\mathcal{O}(\sqrt{m})$ per operation
- **Space trade-off:** $\Theta(m\sqrt{m})$ vs $\mathcal{O}(m)$

# Results

## Our contribution

- **General transformation:** Partial $\rightarrow$ Full retroactivity
- **No persistent data structures needed**
- **Same time complexity:** $\mathcal{O}(\sqrt{m})$ per operation
- **Space trade-off:** $\Theta(m\sqrt{m})$ vs $\mathcal{O}(m)$

## Semi-retroactive MSF implementation

- **Operations:** $\texttt{add\_edge}(u, v, w, t)$, $\texttt{get\_msf}(t)$
- **Time:** $\mathcal{O}(\sqrt{m}\log n)$ per operation
- **Space:** $\Theta(m\sqrt{m})$
- **No fixed** $m$ **or time range restrictions**

Thank you!

Questions?