

# How to go from partial to full retroactivity in detail

Cristina Gomes Fernandes, Felipe Castro 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

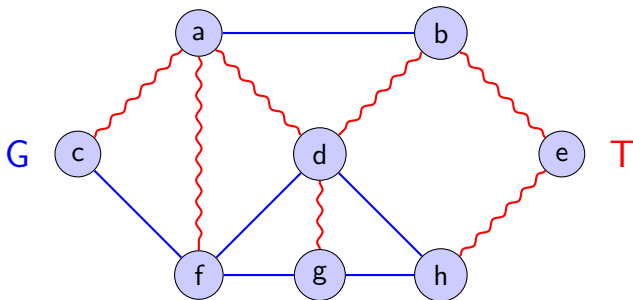


Figure: Graph  $G$  (blue edges) and spanning tree  $T$  (red wavy edges)

# Minimum Spanning Tree and Forest

- **Minimum Spanning Tree (MST):** spanning tree with minimum total cost

# Minimum Spanning Tree and Forest

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

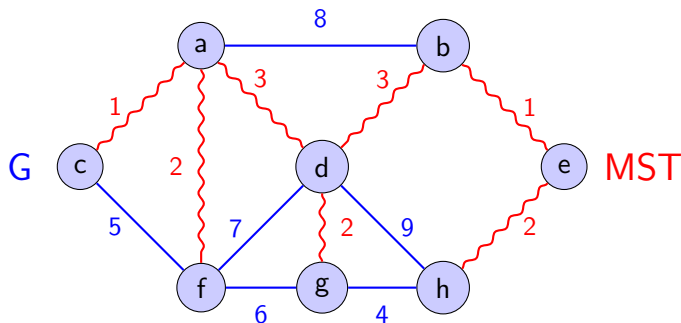


Figure: Graph  $G$  (blue edges) and Minimum Spanning Tree (red wavy edges)

# 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
- **Algorithm for adding edge  $(u, v, w)$ :**
  - 1 Check if  $u$  and  $v$  are in same component
  - 2 If not: add edge to forest
  - 3 If yes: find max cost edge on  $u$ - $v$  path
  - 4 If  $w < \text{max cost}$ : replace max edge with new edge

# Frederickson's link-cut tree solution

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

# Frederickson's link-cut tree solution

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

# Incremental MSF example - Step 1

- **add\_edge(g, h, 4):** Add edge with cost 4

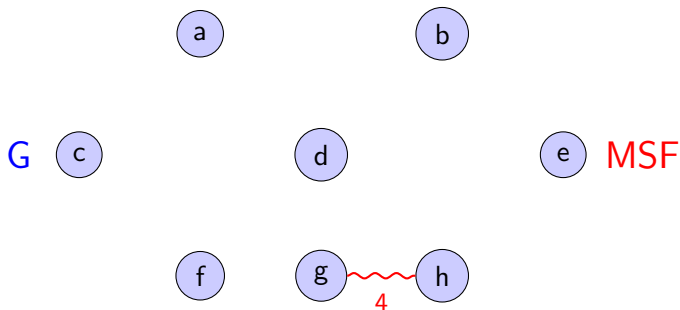


Figure: Step 1: Added edge  $(g, h)$  with cost 4

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

## Incremental MSF example - Step 2

- **add\_edge(c, a, 1):** Add edge with cost 1

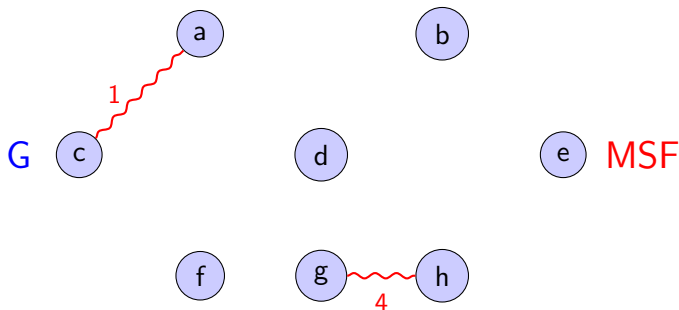


Figure: Step 2: Added edge (c,a) with cost 1

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

## Incremental MSF example - Step 3

- **add\_edge(f, g, 6):** Add edge with cost 6

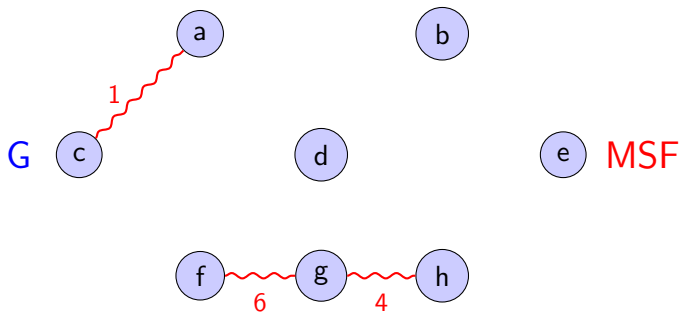


Figure: Step 3: Added edge (f,g) with cost 6

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



## Incremental MSF example - Step 4

- **add\_edge(a, f, 2):** Add edge with cost 2

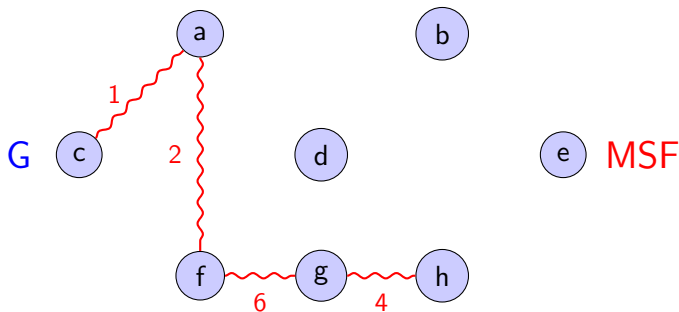


Figure: Step 4: Added edge (a,f) with cost 2

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

## Incremental MSF example - Step 5

- **add\_edge(c, f, 5):** Add edge with cost 5

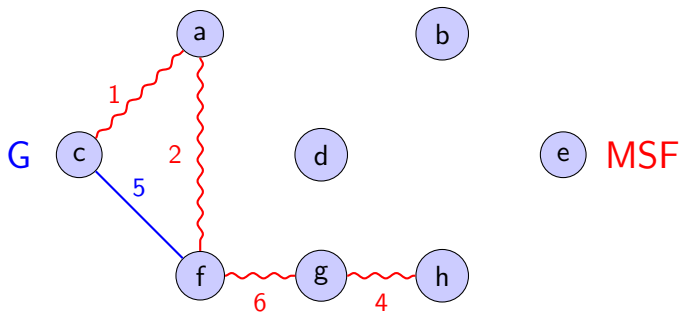


Figure: Step 5: Added edge  $(c, f)$  with cost 5

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

## Incremental MSF example - Step 6

- **add\_edge(f, d, 7):** Add edge with cost 7

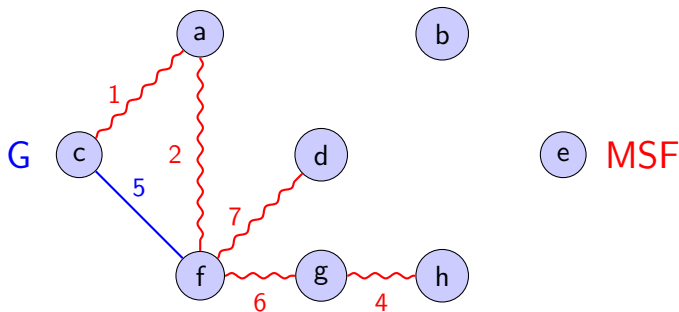


Figure: Step 6: Added edge (f,d) with cost 7

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

## Incremental MSF example - Step 7

- **add\_edge(a, d, 3):** Add edge with cost 3

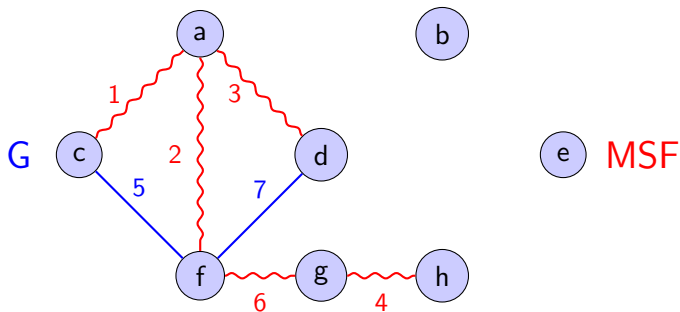


Figure: Step 7: Added edge (a,d) with cost 3

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

## Incremental MSF example - Step 8

- **add\_edge(d, g, 2):** Add edge with cost 2

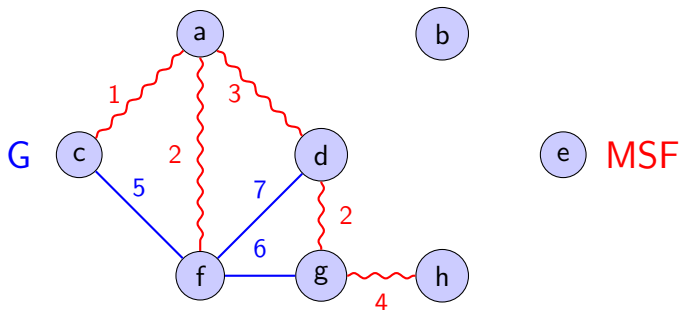


Figure: Step 8: Added edge (d,g) with cost 2

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

# Incremental MSF example - Final Result

- **Continue adding edges...**

## Incremental MSF example - Final Result

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

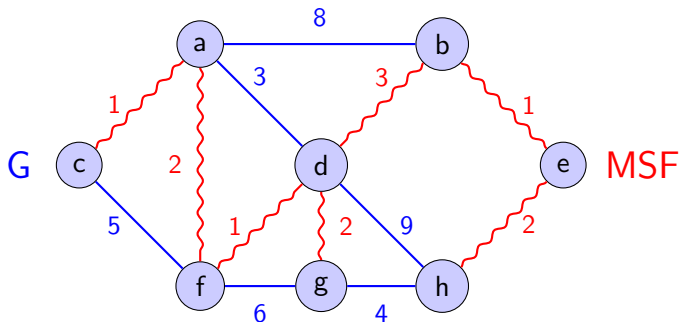


Figure: Final MSF with optimal cost = 12

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

# What is retroactivity?

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



# What is retroactivity?

- **Problem:** Data structures usually support updates and queries
- 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
- 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

## Partially Retroactive

- Queries only on **current** state
- Insert/remove updates at any time
- Example: Dynamic MSF  $\rightarrow$  Partially retroactive MSF

# Partial vs Full retroactivity

## Partially Retroactive

- Queries only on **current** state
- Insert/remove updates at any time
- Example: Dynamic MSF  $\rightarrow$  Partially retroactive MSF

## Fully Retroactive

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

# Partial vs Full retroactivity

## Partially Retroactive

- Queries only on **current** state
- Insert/remove updates at any time
- Example: Dynamic MSF  $\rightarrow$  Partially retroactive MSF

## Fully Retroactive

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

## 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

# 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
- 
- **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

- **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  $i = 1, \dots, \sqrt{m}$

# 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  $i = 1, \dots, \sqrt{m}$
- **Data structures:**  $D_i$  contains edges before time  $t_i$
- **Time:**  $\mathcal{O}(\sqrt{m} \log n)$  per operation

# Limitations

## Problems with their approach

- **Fixed  $m$ :** Must know sequence length beforehand
- **Fixed time range:** Operations must have timestamps 1 to  $m$
- **No rebuilding:** Cannot handle arbitrary growth

# Limitations

## Problems with their approach

- **Fixed  $m$ :** Must know sequence length beforehand
- **Fixed time range:** Operations must have timestamps 1 to  $m$
- **No rebuilding:** Cannot handle arbitrary growth

## Our goal

Remove these limitations while maintaining efficiency

# Limitations

## Problems with their approach

- **Fixed  $m$ :** Must know sequence length beforehand
- **Fixed time range:** Operations must have timestamps 1 to  $m$
- **No rebuilding:** Cannot handle arbitrary growth

## Our goal

Remove these limitations while maintaining efficiency

- **Key insight:** Implement rebuilding process
- **Challenge:** How to rebuild without persistent data structures?
- **Solution:** Reuse existing data structures during rebuilding

## 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  $D_i \rightarrow D'_{i+2}$  for  $i = 0, \dots, k-1$
  - 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  $D_i \rightarrow D'_{i+2}$  for  $i = 0, \dots, k-1$
  - 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  $D_i \rightarrow D'_{i+2}$  for  $i = 0, \dots, k-1$
  - 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

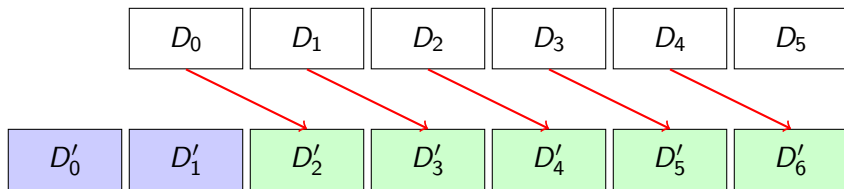
# Rebuilding algorithm

- ①  $D'_0 \leftarrow \text{NEWINCREMENTALMSF}()$
- ②  $D'_1 \leftarrow \text{NEWINCREMENTALMSF}()$
- ③ For  $i = 2$  to  $k + 1$ :  $D'_i \leftarrow D_{i-2}$  ▷ reuse existing
- ④ For  $i = 1$  to  $k + 1$ :
  - ▶  $p \leftarrow \text{KTH}(S, i(k + 1))$  ▷  $i(k + 1)$ th edge
  - ▶  $t'_i \leftarrow p.\text{time}$
  - ▶  $\text{ADDEDGES}(S, t_{i-2}, t'_i, D'_i)$
- ⑤ Return  $k + 1, D', t'$

# Rebuilding algorithm

- ①  $D'_0 \leftarrow \text{NEWINCREMENTALMSF}()$
- ②  $D'_1 \leftarrow \text{NEWINCREMENTALMSF}()$
- ③ For  $i = 2$  to  $k + 1$ :  $D'_i \leftarrow D_{i-2}$  ▷ reuse existing
- ④ For  $i = 1$  to  $k + 1$ :
  - ▶  $p \leftarrow \text{KTH}(S, i(k + 1))$  ▷  $i(k + 1)$ th edge
  - ▶  $t'_i \leftarrow p.\text{time}$
  - ▶  $\text{ADDEDGES}(S, t_{i-2}, t'_i, D'_i)$
- ⑤ Return  $k + 1, D', t'$

Original



New

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

# 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:** `add_edge( $u, v, w, t$ )`, `get_msf( $t$ )`
- **Time:**  $\mathcal{O}(\sqrt{m} \log n)$  per operation
- **Space:**  $\Theta(m\sqrt{m})$
- **No fixed  $m$  or time range restrictions**

# Extending for full retroactivity

- **General applicability:** Works for any partially retroactive data structure

# Extending for full retroactivity

- **General applicability:** Works for any partially retroactive data structure
- **Supporting removals:** To achieve full retroactivity
  - ▶ Adapt rebuilding trigger: when  $|\lfloor \sqrt{m'} \rfloor - \lfloor \sqrt{m} \rfloor| \leq 1$
  - ▶ Handle both insertions and removals in update sequence
  - ▶ Rebuilding frequency: every  $2\lfloor \sqrt{m} \rfloor - 1$  operations

# Extending for full retroactivity

- **General applicability:** Works for any partially retroactive data structure
- **Supporting removals:** To achieve full retroactivity
  - ▶ Adapt rebuilding trigger: when  $|\lfloor \sqrt{m'} \rfloor - \lfloor \sqrt{m} \rfloor| \leq 1$
  - ▶ Handle both insertions and removals in update sequence
  - ▶ Rebuilding frequency: every  $2\lfloor \sqrt{m} \rfloor - 1$  operations
- **Requirements:**
  - ▶ Partially retroactive data structure
  - ▶ Rollback capability
  - ▶ No persistent version needed



# Thank you!

# Questions?