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

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

1. Hello there everybody, my name is Felipe Noronha and today I'm gonna gonna be doing a quick presentation about the paper professor Cristina and I did at the IME of USP
2. with the topic being: going from partial to full retroactivity in detail
3. This work addresses a practical limitation in Demaine, Iacono & Langerman's 2007 transformation
4. And also iterates over Junior & Seabra's solution from 2022
5. Our contribution shows how to go from partial to full retroactivity with same time complexity without requiring persistent data structures
6. As our main focus of study we had the minimum spanning fores problem, and that's what we are going to start with today

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

1. Start with basic concept of spanning tree - fundamental in graph theory
2. ———- SKIP SLIDE ———-
3. Explain key properties: connected (path between any two vertices), acyclic (no cycles), contains exactly n-1 edges for n vertices
4. ———- SKIP SLIDE ———-
5. Show visual example with graph G (blue edges) and spanning tree T (red wavy edges)
6. In the example: 8 vertices, so spanning tree has exactly 7 edges
7. This builds up the concepts step by step for the incremental MSF problem
8. Emphasize that spanning trees are not unique - there can be many valid spanning trees

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

1. Start with basic concept of spanning tree - fundamental in graph theory
2. ——— SKIP SLIDE ———
3. Explain key properties: connected (path between any two vertices), acyclic (no cycles), contains exactly n-1 edges for n vertices
4. ——— SKIP SLIDE ———
5. Show visual example with graph G (blue edges) and spanning tree T (red wavy edges)
6. In the example: 8 vertices, so spanning tree has exactly 7 edges
7. This builds up the concepts step by step for the incremental MSF problem
8. Emphasize that spanning trees are not unique - there can be many valid spanning trees

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

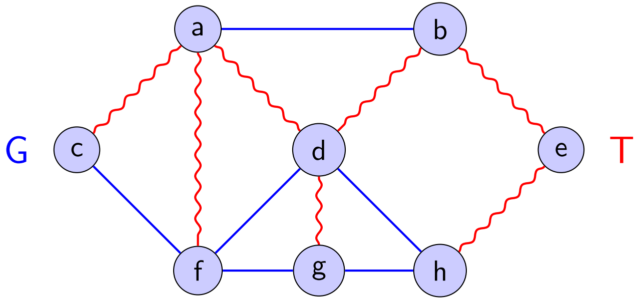
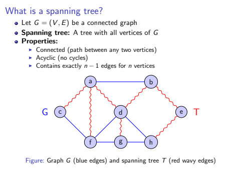


Figure: Graph G (blue edges) and spanning tree T (red wavy edges)



1. Start with basic concept of spanning tree - fundamental in graph theory
2. ———- SKIP SLIDE ———-
3. Explain key properties: connected (path between any two vertices), acyclic (no cycles), contains exactly n-1 edges for n vertices
4. ———- SKIP SLIDE ———-
5. Show visual example with graph G (blue edges) and spanning tree T (red wavy edges)
6. In the example: 8 vertices, so spanning tree has exactly 7 edges
7. This builds up the concepts step by step for the incremental MSF problem
8. Emphasize that spanning trees are not unique - there can be many valid spanning trees

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

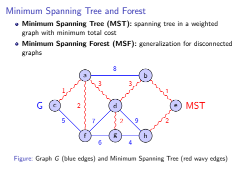
1. Define MST as spanning tree in a graph with weighted edges with minimum total cost - optimization problem
2. ———- SKIP SLIDE ———-
3. Generalize to MSF for disconnected graphs - collection of MSTs for each component
4. ———- SKIP SLIDE ———-
5. Show visual example with weighted edges: blue edges show graph G, red wavy edges show MST
6. Demonstrate that red edges form MST with cost 14 $(1+2+3+2+3+1+2=14)$
7. Explain that any other spanning tree would have higher cost - this is the optimal solution
8. This prepares for the incremental MSF problem where we maintain optimality dynamically
9. Key insight: we need to maintain optimality as edges are added one by one

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

1. Define MST as spanning tree in a graph with weighted edges with minimum total cost - optimization problem
2. ———- SKIP SLIDE ———-
3. Generalize to MSF for disconnected graphs - collection of MSTs for each component
4. ———- SKIP SLIDE ———-
5. Show visual example with weighted edges: blue edges show graph G, red wavy edges show MST
6. Demonstrate that red edges form MST with cost 14 $(1+2+3+2+3+1+2 = 14)$
7. Explain that any other spanning tree would have higher cost - this is the optimal solution
8. This prepares for the incremental MSF problem where we maintain optimality dynamically
9. Key insight: we need to maintain optimality as edges are added one by one



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

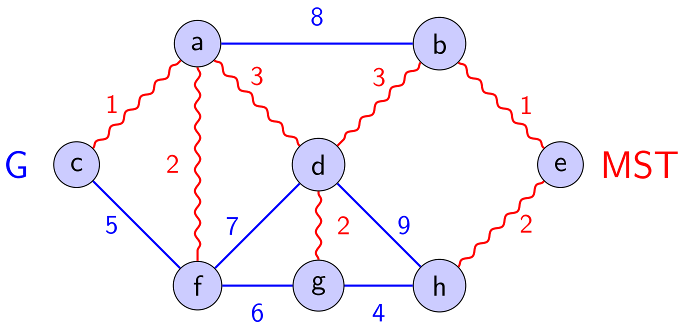


Figure: Graph G (blue edges) and Minimum Spanning Tree (red wavy edges)

1. Define MST as spanning tree in a graph with weighted edges with minimum total cost - optimization problem
2. ———- SKIP SLIDE ———-
3. Generalize to MSF for disconnected graphs - collection of MSTs for each component
4. ———- SKIP SLIDE ———-
5. Show visual example with weighted edges: blue edges show graph G, red wavy edges show MST
6. Demonstrate that red edges form MST with cost 14 $(1+2+3+2+3+1+2 = 14)$
7. Explain that any other spanning tree would have higher cost - this is the optimal solution
8. This prepares for the incremental MSF problem where we maintain optimality dynamically
9. Key insight: we need to maintain optimality as 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

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

1. Define incremental MSF problem clearly: maintain MSF as graph grows
2. Emphasize that graph starts empty and grows - this is crucial for our approach
3. ———- SKIP SLIDE ———-
4. Show the two key operations: add_edge(u,v,w) and get_msf()
5. ———- SKIP SLIDE ———-
6. Mention Frederickson's breakthrough solution from 1983 using link-cut trees
7. Note the cost is $O(logn)$ amortized per edge addition using link-cut trees
8. This is the foundation for retroactive version - we'll extend this to handle time
9. Key insight: we need to maintain MSF not just for current state, but for any time t

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

1. Define incremental MSF problem clearly: maintain MSF as graph grows
2. Emphasize that graph starts empty and grows - this is crucial for our approach
3. ——— SKIP SLIDE ———
4. Show the two key operations: add_edge(u,v,w) and get_msf()
5. ——— SKIP SLIDE ———
6. Mention Frederickson's breakthrough solution from 1983 using link-cut trees
7. Note the cost is $O(logn)$ amortized per edge addition using link-cut trees
8. This is the foundation for retroactive version - we'll extend this to handle time
9. Key insight: we need to maintain MSF not just for current state, but for any time t

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

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

1. Define incremental MSF problem clearly: maintain MSF as graph grows
2. Emphasize that graph starts empty and grows - this is crucial for our approach
3. ——— SKIP SLIDE ———
4. Show the two key operations: add_edge(u,v,w) and get_msf()
5. ——— SKIP SLIDE ———
6. Mention Frederickson's breakthrough solution from 1983 using link-cut trees
7. Note the cost is $O(logn)$ amortized per edge addition using link-cut trees
8. This is the foundation for retroactive version - we'll extend this to handle time
9. Key insight: we need to maintain MSF not just for current state, but for any time t

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

1. Explain Frederickson's key insight: use link-cut trees to maintain MSF dynamically
2. ———- SKIP SLIDE ———-
3. List the specific link-cut tree operations: find_max, link, cut - all $O(logn)$ amortized
4. ———- SKIP SLIDE ———-
5. 1. Check connectivity using link-cut trees $find_rootoperations$
6. 2. If not connected: add edge directly $linkoperation$
7. 3. If connected: find max cost edge on u-v path $find_maxoperation$
8. 4. If new edge cheaper: replace max edge $cut + linkoperations$
9. ———- SKIP SLIDE ———-
10. Emphasize the logarithmic time complexity: $O(logn)$ per edge addition
11. Key insight: link-cut trees support efficient rollback, which we'll need for retroactivity

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

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

1. Explain Frederickson's key insight: use link-cut trees to maintain MSF dynamically
2. ——— SKIP SLIDE ———
3. List the specific link-cut tree operations: find_max, link, cut - all $O(logn)$ amortized
4. ——— SKIP SLIDE ———
5. 1. Check connectivity using link-cut trees $find_rootoperations$
6. 2. If not connected: add edge directly $linkoperation$
7. 3. If connected: find max cost edge on u-v path $find_maxoperation$
8. 4. If new edge cheaper: replace max edge $cut + linkoperations$
9. ——— SKIP SLIDE ———
10. Emphasize the logarithmic time complexity: $O(logn)$ per edge addition
11. Key insight: link-cut trees support efficient rollback, which we'll need for retroactivity

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
- **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 $w <$ max cost: replace max edge with new edge

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
- 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 $w <$ max cost: replace max edge with new edge

1. Explain Frederickson's key insight: use link-cut trees to maintain MSF dynamically
2. ——— SKIP SLIDE ———
3. List the specific link-cut tree operations: find_max, link, cut - all $O(logn)$ amortized
4. ——— SKIP SLIDE ———
5. 1. Check connectivity using link-cut trees $find_rootoperations$
6. 2. If not connected: add edge directly $linkoperation$
7. 3. If connected: find max cost edge on u-v path $find_maxoperation$
8. 4. If new edge cheaper: replace max edge $cut + linkoperations$
9. ——— SKIP SLIDE ———
10. Emphasize the logarithmic time complexity: $O(logn)$ per edge addition
11. Key insight: link-cut trees support efficient rollback, which we'll need for retroactivity

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
- **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 $w <$ max cost: replace max edge with new edge
- **Total cost:** Amortized $\mathcal{O}(\log n)$ per edge addition

Frederickson's link-cut tree solution
Key insight: Use link-cut trees to maintain MSF dynamically
Link-cut tree operations:
find_max(u, v): O(log n) amortized
link(u, v, w): O(log n) amortized
cut(u, v): O(log n) amortized
Algorithm for adding edge (u, v, w):
Check if u and v are in same component
If not: add edge to forest
If yes: find max cost edge on u-v path
If w < max cost: replace max edge with new edge
Total cost: Amortized O(log n) per edge addition

1. Explain Frederickson's key insight: use link-cut trees to maintain MSF dynamically
2. ——— SKIP SLIDE ———
3. List the specific link-cut tree operations: find_max, link, cut - all $O(logn)$ amortized
4. ——— SKIP SLIDE ———
5. 1. Check connectivity using link-cut trees $find_rootoperations$
6. 2. If not connected: add edge directly $linkoperation$
7. 3. If connected: find max cost edge on u-v path $find_maxoperation$
8. 4. If new edge cheaper: replace max edge $cut + linkoperations$
9. ——— SKIP SLIDE ———
10. Emphasize the logarithmic time complexity: $O(logn)$ per edge addition
11. Key insight: link-cut trees support efficient rollback, which we'll need for retroactivity

Incremental MSF example - Step 1

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

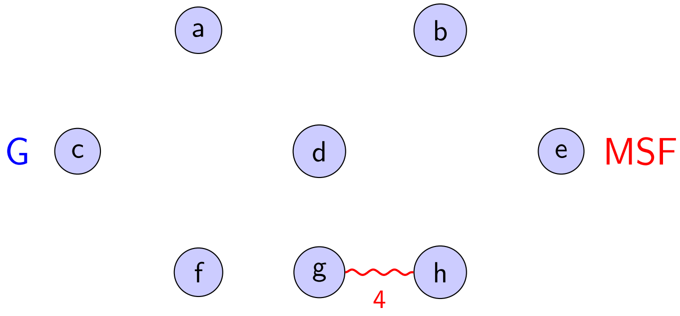
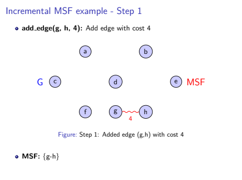


Figure: Step 1: Added edge (g,h) with cost 4

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



1. Show first edge being added: (g,h) with cost 4
2. Explain it's automatically added to MSF since no cycle exists yet
3. Current MSF: g-h with total cost 4
4. This demonstrates the incremental nature: we start with empty graph
5. Each step shows how MSF evolves as edges are added
6. Link-cut tree operations: link(g,h) - O(log n) time

Incremental MSF example - Step 2

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

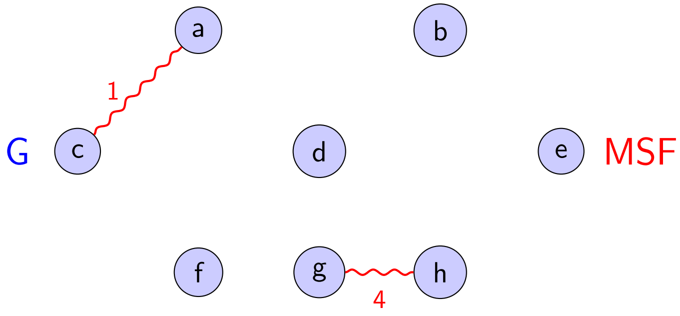
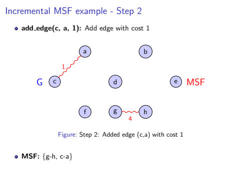


Figure: Step 2: Added edge (c,a) with cost 1

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



1. Show second edge being added: (c,a) with cost 1
2. Still no cycle, so added to MSF directly
3. Current MSF: g-h, c-a with total cost 5
4. Link-cut tree operations: link(c,a) - O(log n) time
5. We now have two separate components: g,h and c,a
6. This shows how MSF grows incrementally without cycles

Incremental MSF example - Step 3

- **add_edge(f, g, 6):** Add edge with cost 6

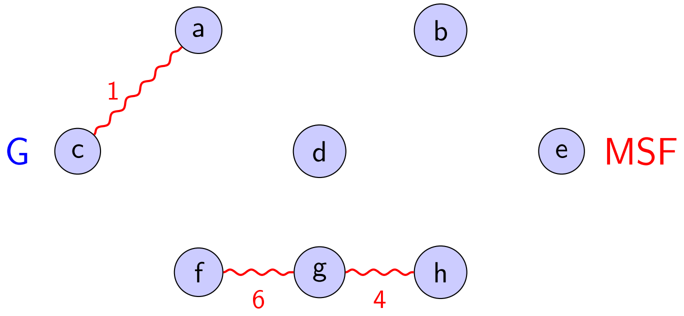
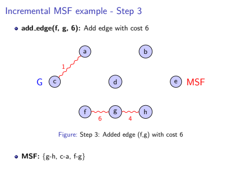


Figure: Step 3: Added edge (f,g) with cost 6

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



1. Show third edge being added: (f,g) with cost 6
2. Still no cycle, so added to MSF directly
3. Current MSF: g-h, c-a, f-g with total cost 11
4. Link-cut tree operations: link(f,g) - O(log n) time
5. Now we have components: g,h,f and c,a
6. This continues the incremental growth pattern

Incremental MSF example - Step 4

- **add_edge(a, f, 2):** Add edge with cost 2

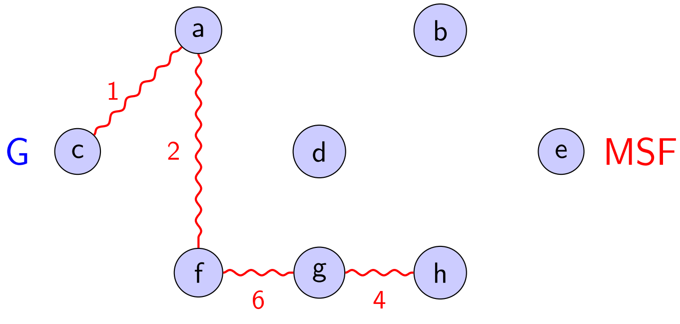
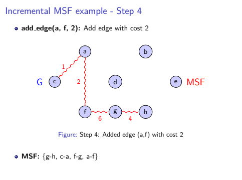


Figure: Step 4: Added edge (a,f) with cost 2

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



1. Show fourth edge being added: (a,f) with cost 2
2. Still no cycle, so added to MSF directly
3. Current MSF: g-h, c-a, f-g, a-f with total cost 13
4. Link-cut tree operations: link(a,f) - $O(\log n)$ time
5. Now we have component: g,h,f,a,c - all vertices connected!
6. This shows how components merge as edges are added

Incremental MSF example - Step 5

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

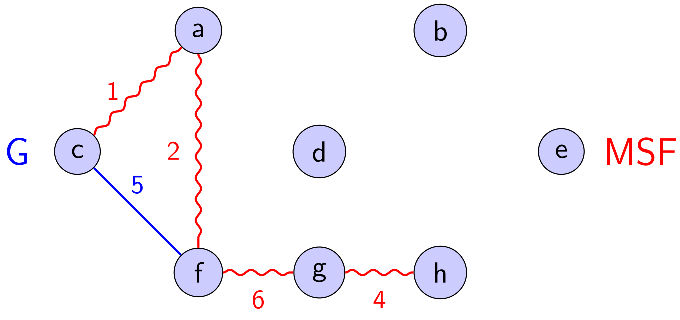
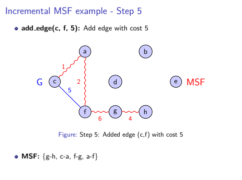


Figure: Step 5: Added edge (c,f) with cost 5

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



1. Show fifth edge being added: c, f with cost 5
2. This creates a cycle! c-a-f-g-h-c forms a cycle
3. Link-cut tree operations: find_max(c,f) finds edge a, f with cost 2
4. Since new edge cost 5, which is grater than 2, it's not added to the MSF
5. Current MSF: g-h, c-a, f-g, a-f with total cost 13
6. This demonstrates the cycle-breaking optimization in Frederickson's algorithm
7. Key insight: we maintain optimality by replacing expensive edges with cheaper ones

Incremental MSF example - Step 6

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

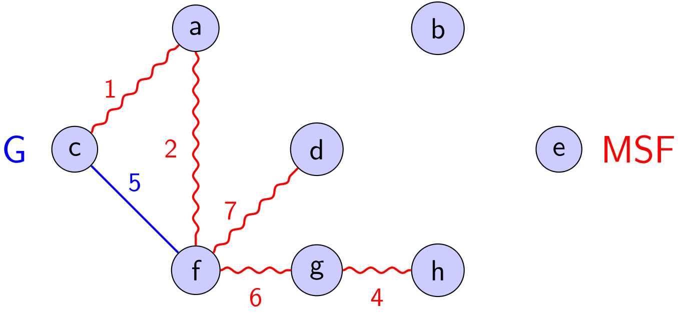
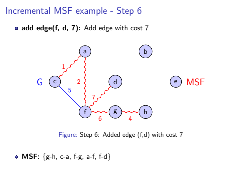


Figure: Step 6: Added edge (f,d) with cost 7

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



1. Show sixth edge being added: f, d with cost 7
2. This creates a cycle! f-d-g-h-f forms a cycle
3. Link-cut tree operations: find_max(f,d) no cycle returned
4. Edge is added to the MSF
5. Current MSF: g-h, c-a, f-g, a-f, f-d with total cost 20

Incremental MSF example - Step 7

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

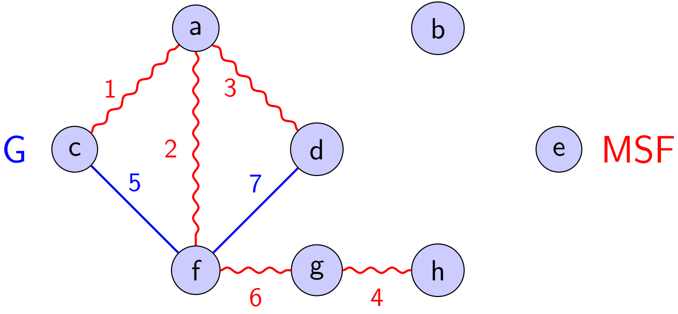
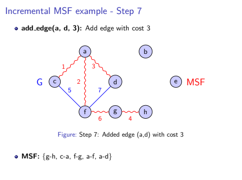


Figure: Step 7: Added edge (a,d) with cost 3

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



1. Show seventh edge being added: a, d with cost 3
2. This creates a cycle! a-d-f-a forms a cycle
3. Link-cut tree operations: find_max(a,d finds edge f, d with cost 7
4. Since new edge cost 3 ¡ max cost 7, we replace (f,d) with (a,d)
5. Current MSF: g-h, c-a, f-g, a-f, a-d with total cost 16 *improved*!
6. This shows continued optimization as better edges are found
7. Key insight: the algorithm continuously improves the MSF as new edges arrive

Incremental MSF example - Step 8

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

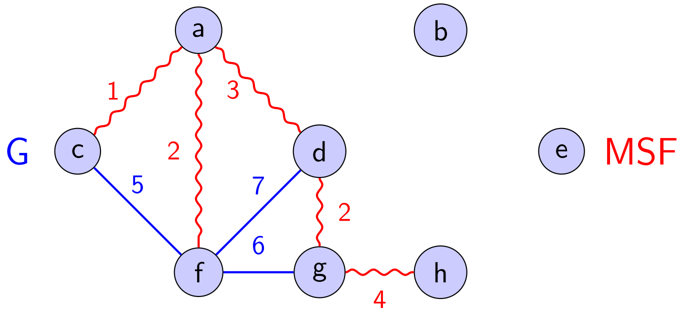
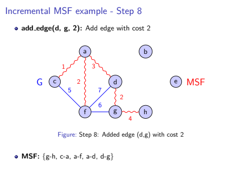


Figure: Step 8: Added edge (d,g) with cost 2

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



1. Show eighth edge being added: d, g with cost 2
2. This creates a cycle! d-g-f-a-d forms a cycle
3. Link-cut tree operations: find_max(d,g) finds edge (f,g) with cost 6
4. Since new edge cost 2 ¡ max cost 6, we replace (f,g) with (d,g)
5. Current MSF: {g-h, c-a, a-f, a-d, d-g} with total cost 12 *improved*!
6. This shows the final optimization step
7. Key insight: the algorithm finds the optimal MSF through incremental improvements
8. Total cost reduced from 14 to 8 through smart edge replacements

Incremental MSF example - Final Result

- **Continue adding edges...**

Incremental MSF example - Final Result
- Continue adding edges...

1. If we continue adding edges and applying this algorithm
2. ——— SKIP SLIDE ———
3. Show final complete MSF with optimal cost = 12
4. Summarize the incremental process: started empty, added edges one by one
5. Transition to Frederickson's solution: $O(logn)$ amortized per edge addition
6. Key insight: link-cut trees enable efficient cycle detection and edge replacement
7. This sets up the retroactive version: what if we want to query MSF at any time t?
8. The challenge: maintain MSF not just for current state, but for any historical time
9. This motivates the need for retroactive data structures

Incremental MSF example - Final Result

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

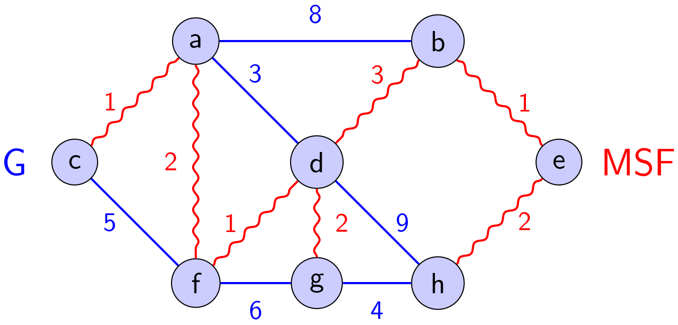
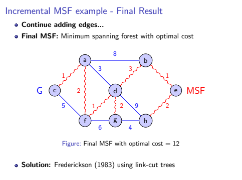


Figure: Final MSF with optimal cost = 12

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



1. If we continue adding edges and applying this algorithm
2. ——— SKIP SLIDE ———
3. Show final complete MSF with optimal cost = 12
4. Summarize the incremental process: started empty, added edges one by one
5. Transition to Frederickson's solution: $O(logn)$ amortized per edge addition
6. Key insight: link-cut trees enable efficient cycle detection and edge replacement
7. This sets up the retroactive version: what if we want to query MSF at any time t?
8. The challenge: maintain MSF not just for current state, but for any historical time
9. This motivates the need for retroactive data structures

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

1. Data structures usually support queries, and generally, the order in which the updates are done will affect the result of the queries
2. Because of this, we don't always have a proper way of correcting mistakes or adding forgotten operations
3. ——— SKIP SLIDE ———
4. That where retroactive comes to play, to allow us to manipulate the sequence of updates, while also allowing for queries at any moment in time
5. ——— SKIP SLIDE ———
6. Show the three key operations: insert, remove, query at any time
7. Emphasize that time stamps must be distinct - this is important for correctness
8. This sets up the distinction between partial and full retroactivity
9. Key insight: we need to maintain state at every possible time, not just current

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

1. Data structures usually support queries, and generally, the order in which the updates are done will affect the result of the queries
2. Because of this, we don't always have a proper way of correcting mistakes or adding forgotten operations
3. ——— SKIP SLIDE ———
4. That where retroactive comes to play, to allow us to manipulate the sequence of updates, while also allowing for queries at any moment in time
5. ——— SKIP SLIDE ———
6. Show the three key operations: insert, remove, query at any time
7. Emphasize that time stamps must be distinct - this is important for correctness
8. This sets up the distinction between partial and full retroactivity
9. Key insight: we need to maintain state at every possible time, not just current

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)

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)

1. Data structures usually support queries, and generally, the order in which the updates are done will affect the result of the queries
2. Because of this, we don't always have a proper way of correcting mistakes or adding forgotten operations
3. ——— SKIP SLIDE ———
4. That where retroactive comes to play, to allow us to manipulate the sequence of updates, while also allowing for queries at any moment in time
5. ——— SKIP SLIDE ———
6. Show the three key operations: insert, remove, query at any time
7. Emphasize that time stamps must be distinct - this is important for correctness
8. This sets up the distinction between partial and full retroactivity
9. Key insight: we need to maintain state at every possible time, not just current

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

1. We also have different flavors of retroactivity, namely partial and semi-retroactivity
2. With full retroactivity, we have the operations we just showed
3. ——– SKIP SLIDE ——–
4. While partial retroactivitty only allows queries on current state - this is the limitation
5. ——– SKIP SLIDE ——–
6. Define semi-retroactive: queries at any time, insertions, but no removals
7. Generally, partial retroactive structures are simpler to implement, and with that, an interesting challange arrives

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

1. We also have different flavors of retroactivity, namely partial and semi-retroactivity
2. With full retroactivity, we have the operations we just showed
3. ——— SKIP SLIDE ———
4. While partial retroactivitty only allows queries on current state - this is the limitation
5. ——— SKIP SLIDE ———
6. Define semi-retroactive: queries at any time, insertions, but no removals
7. Generally, partial retroactive structures are simpler to implement, and with that, an interesting challange arrives

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

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

1. We also have different flavors of retroactivity, namely partial and semi-retroactivity
2. With full retroactivity, we have the operations we just showed
3. ——— SKIP SLIDE ———
4. While partial retroactivitty only allows queries on current state - this is the limitation
5. ——— SKIP SLIDE ———
6. Define semi-retroactive: queries at any time, insertions, but no removals
7. Generally, partial retroactive structures are simpler to implement, and with that, an interesting challange arrives

The challenge

Challenge

How to transform partial $\rightarrow$ full retroactivity?

The challenge
Challenge
How to transform partial $\rightarrow$ full retroactivity?

1. State the main challenge clearly: partial to full retroactivity
2. ——— SKIP SLIDE ———
3. Explain what we need to achieve: queries at any time t
4. ——— SKIP SLIDE ———
5. Introduce the solution approach: square-root decomposition
6. Mention the key insight about checkpoints
7. Reference the Demaine et al. work from 2007
8. This motivates the detailed solution in the next slide
9. Key insight: we need to maintain multiple versions of the data structure
10. The challenge: how to do this efficiently without persistent data structures?

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 → full retroactivity?
Problem: Need to support queries at any time t
Solution approach: Square-root decomposition

1. State the main challenge clearly: partial to full retroactivity
2. ——— SKIP SLIDE ———
3. Explain what we need to achieve: queries at any time t
4. ——— SKIP SLIDE ———
5. Introduce the solution approach: square-root decomposition
6. Mention the key insight about checkpoints
7. Reference the Demaine et al. work from 2007
8. This motivates the detailed solution in the next slide
9. Key insight: we need to maintain multiple versions of the data structure
10. The challenge: how to do this efficiently without persistent data structures?

The challenge

Challenge

How to transform partial → 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)

The challenge
Challenge
How to transform partial → 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)

1. State the main challenge clearly: partial to full retroactivity
2. ——— SKIP SLIDE ———
3. Explain what we need to achieve: queries at any time t
4. ——— SKIP SLIDE ———
5. Introduce the solution approach: square-root decomposition
6. Mention the key insight about checkpoints
7. Reference the Demaine et al. work from 2007
8. This motivates the detailed solution in the next slide
9. Key insight: we need to maintain multiple versions of the data structure
10. The challenge: how to do this efficiently without persistent data structures?

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

1. State Theorem 05 from Demaine, Iacono and Langerman 2007
2. Emphasize the persistent data structure requirement, this is the key limitation
3. ———- SKIP SLIDE ———-
4. Explain square-root decomposition concept: break timeline into $\sqrt{m}$ blocks
5. ———- SKIP SLIDE ———-
6. Show how queries work: find checkpoint, apply updates, rollback
7. Time complexity: $O(\sqrt{m})$ slowdown per operation
8. Space complexity: $O(m)$ using persistent data structures
9. Set up the problem: what if we don't have persistent version?
10. Key insight: persistent data structures are complex to implement
11. Our contribution: same performance without persistence requirement

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

1. State Theorem 05 from Demaine, Iacono and Langerman 2007
2. Emphasize the persistent data structure requirement, this is the key limitation
3. ———- SKIP SLIDE ———-
4. Explain square-root decomposition concept: break timeline into $\sqrt{m}$ blocks
5. ———- SKIP SLIDE ———-
6. Show how queries work: find checkpoint, apply updates, rollback
7. Time complexity: $O(\sqrt{m})$ slowdown per operation
8. Space complexity: $O(m)$ using persistent data structures
9. Set up the problem: what if we don't have persistent version?
10. Key insight: persistent data structures are complex to implement
11. Our contribution: same performance without persistence requirement

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

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:
Find closest checkpoint before t
Apply updates from checkpoint to t
Answer query, then rollback

1. State Theorem 05 from Demaine, Iacono and Langerman 2007
2. Emphasize the persistent data structure requirement, this is the key limitation
3. ———- SKIP SLIDE ———-
4. Explain square-root decomposition concept: break timeline into $\sqrt{m}$ blocks
5. ———- SKIP SLIDE ———-
6. Show how queries work: find checkpoint, apply updates, rollback
7. Time complexity: $O(\sqrt{m})$ slowdown per operation
8. Space complexity: $O(m)$ using persistent data structures
9. Set up the problem: what if we don't have persistent version?
10. Key insight: persistent data structures are complex to implement
11. Our contribution: same performance without persistence requirement

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: $\Theta(m\sqrt{m})$

1. Explain the space issue with naive approach: $\Theta(m\sqrt{m})$ space
2. ———- SKIP SLIDE ———-
3. Show how Demaine et al. solve it with persistent data structures: Om space
4. A persistent data structure is a data structure that always preserves the previous version of itself when it is modified, so you can query at any time t but only update the present
5. ———- SKIP SLIDE ———-
6. State the practical problem: persistent versions are complex to implement
7. ———- SKIP SLIDE ———-
8. Present our key contribution: same performance without persistence
9. Emphasize the space trade-off we make: $\Theta(m\sqrt{m})$ vs Om
10. This motivates our improved rebuilding approach that use idepedent copies

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: $\mathcal{O}(m)$

1. Explain the space issue with naive approach: $\Theta(m\sqrt{m})$ space
2. ——— SKIP SLIDE ———
3. Show how Demaine et al. solve it with persistent data structures: Om space
4. A persistent data structure is a data structure that always preserves the previous version of itself when it is modified, so you can query at any time t but only update the present
5. ——— SKIP SLIDE ———
6. State the practical problem: persistent versions are complex to implement
7. ——— SKIP SLIDE ———
8. Present our key contribution: same performance without persistence
9. Emphasize the space trade-off we make: $\Theta(m\sqrt{m})$ vs Om
10. This motivates our improved rebuilding approach that use idepedent copies

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?

1. Explain the space issue with naive approach: $\Theta(m\sqrt{m})$ space
2. ——— SKIP SLIDE ———
3. Show how Demaine et al. solve it with persistent data structures: Om space
4. A persistent data structure is a data structure that always preserves the previous version of itself when it is modified, so you can query at any time t but only update the present
5. ——— SKIP SLIDE ———
6. State the practical problem: persistent versions are complex to implement
7. ——— SKIP SLIDE ———
8. Present our key contribution: same performance without persistence
9. Emphasize the space trade-off we make: $\Theta(m\sqrt{m})$ vs Om
10. This motivates our improved rebuilding approach that use idepedent copies

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: $\Theta(m\sqrt{m})$

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: $\Theta(m\sqrt{m})$

1. Explain the space issue with naive approach: $\Theta(m\sqrt{m})$ space
2. ———- SKIP SLIDE ———-
3. Show how Demaine et al. solve it with persistent data structures: Om space
4. A persistent data structure is a data structure that always preserves the previous version of itself when it is modified, so you can query at any time t but only update the present
5. ———- SKIP SLIDE ———-
6. State the practical problem: persistent versions are complex to implement
7. ———- SKIP SLIDE ———-
8. Present our key contribution: same performance without persistence
9. Emphasize the space trade-off we make: $\Theta(m\sqrt{m})$ vs Om
10. This motivates our improved rebuilding approach that use idepedent copies

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

1. Start with Junior and Seabra's work as our starting point
2. Explain their semi-retroactive MSF problem: add edge at time t, query at time t
3. Show their operations: add_edgeu, v, w, t and get_msf$t$
4. ———- SKIP SLIDE ———-
5. Describe their square-root decomposition approach: $\sqrt{m}$ checkpoints
6. Show how they use checkpoints: $t_i = i\sqrt{m} fori = 1, ..., sqrtm$
7. ———- SKIP SLIDE ———-
8. Data structures: D_i contains edges before time t_i
9. Time complexity: $O(\sqrt{m} logn)$ per operation
10. This sets up their limitations in the next slide
11. Key insight: they assume fixed m and time range - serious restrictions

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 $i = 1, \ldots, \sqrt{m}$

1. Start with Junior and Seabra's work as our starting point
2. Explain their semi-retroactive MSF problem: add edge at time t, query at time t
3. Show their operations: add_edgeu, v, w, t and get_msf$t$
4. ——— SKIP SLIDE ———
5. Describe their square-root decomposition approach: $\sqrt{m}$ checkpoints
6. Show how they use checkpoints: $t_i = i\sqrt{m} fori = 1, ..., sqrtm$
7. ——— SKIP SLIDE ———
8. Data structures: D_i contains edges before time t_i
9. Time complexity: $O(\sqrt{m} logn)$ per operation
10. This sets up their limitations in the next slide
11. Key insight: they assume fixed m and time range - serious restrictions

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

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

1. Start with Junior and Seabra's work as our starting point
2. Explain their semi-retroactive MSF problem: add edge at time t, query at time t
3. Show their operations: add_edgeu, v, w, t and get_msf$t$
4. ——— SKIP SLIDE ———
5. Describe their square-root decomposition approach: $\sqrt{m}$ checkpoints
6. Show how they use checkpoints: $t_i = i\sqrt{m} fori = 1, ..., sqrtm$
7. ——— SKIP SLIDE ———
8. Data structures: D_i contains edges before time t_i
9. Time complexity: $O(\sqrt{m} logn)$ per operation
10. This sets up their limitations in the next slide
11. Key insight: they assume fixed m and time range - serious restrictions

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

1. Clearly list their three main limitations
2. Emphasize that fixed m and time range are serious restrictions
3. ——— SKIP SLIDE ———
4. State our goal: remove these limitations while maintaining efficiency
5. ——— SKIP SLIDE ———
6. Present our key insight: implement rebuilding process
7. Explain the challenge: how to rebuild without persistent structures
8. This motivates our solution in the next slide
9. Key insight: we need to handle arbitrary growth without knowing m beforehand
10. Our approach: rebuild when m becomes a perfect square

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

1. Clearly list their three main limitations
2. Emphasize that fixed m and time range are serious restrictions
3. ——— SKIP SLIDE ———
4. State our goal: remove these limitations while maintaining efficiency
5. ——— SKIP SLIDE ———
6. Present our key insight: implement rebuilding process
7. Explain the challenge: how to rebuild without persistent structures
8. This motivates our solution in the next slide
9. Key insight: we need to handle arbitrary growth without knowing m beforehand
10. Our approach: rebuild when m becomes a perfect square

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

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

1. Clearly list their three main limitations
2. Emphasize that fixed m and time range are serious restrictions
3. ———- SKIP SLIDE ———-
4. State our goal: remove these limitations while maintaining efficiency
5. ———- SKIP SLIDE ———-
6. Present our key insight: implement rebuilding process
7. Explain the challenge: how to rebuild without persistent structures
8. This motivates our solution in the next slide
9. Key insight: we need to handle arbitrary growth without knowing m beforehand
10. Our approach: rebuild when m becomes a perfect square

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)

1. Explain our key insight: reuse existing data structures
2. Show rebuilding moments: when m is a perfect square ($m = k^2$)
3. ——— SKIP SLIDE ———
4. Walk through the three-step strategy:
5. 1. Create new empty structures D'_0, D'_1
6. 2. Reuse $D_i to D'_{i+2}$ for $i = 0, ..., k-1$
7. 3. Apply missing updates to each D'_i
8. ——— SKIP SLIDE ———
9. Present the key lemma: every update in D_i is within first $(i+2)(k+1)$ updates
10. ——— SKIP SLIDE ———
11. Analyze time complexity: $O(mlogn)$ total, $O(\sqrt{m}logn)$ amortized
12. This sets up the detailed algorithm in the next slide
13. Key insight: we can reuse most of the work from previous structures
14. The offset (i+2) is crucial for correctness

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

1. Explain our key insight: reuse existing data structures
2. Show rebuilding moments: when m is a perfect square ($m = k^2$)
3. ——— SKIP SLIDE ———
4. Walk through the three-step strategy:
5. 1. Create new empty structures D'_0, D'_1
6. 2. Reuse $D_i to D'_{i+2}$ for $i = 0, ..., k-1$
7. 3. Apply missing updates to each D'_i
8. ——— SKIP SLIDE ———
9. Present the key lemma: every update in D_i is within first $(i+2)(k+1)$ updates
10. ——— SKIP SLIDE ———
11. Analyze time complexity: $O(mlogn)$ total, $O(\sqrt{m}logn)$ amortized
12. This sets up the detailed algorithm in the next slide
13. Key insight: we can reuse most of the work from previous structures
14. The offset (i+2) is crucial for correctness

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 $i = 0, \ldots, k-1$
  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:
Create new empty structures D'_0, D'_1
Reuse $D_i \rightarrow D'_{i+2}$ for $i = 0, \ldots, k-1$
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.

1. Explain our key insight: reuse existing data structures
2. Show rebuilding moments: when m is a perfect square ($m = k^2$)
3. ——— SKIP SLIDE ———
4. Walk through the three-step strategy:
5. 1. Create new empty structures D'_0, D'_1
6. 2. Reuse $D_i to D'_{i+2}$ for $i = 0, ..., k-1$
7. 3. Apply missing updates to each D'_i
8. ——— SKIP SLIDE ———
9. Present the key lemma: every update in D_i is within first $(i+2)(k+1)$ updates
10. ——— SKIP SLIDE ———
11. Analyze time complexity: $O(mlogn)$ total, $O(\sqrt{m}logn)$ amortized
12. This sets up the detailed algorithm in the next slide
13. Key insight: we can reuse most of the work from previous structures
14. The offset (i+2) is crucial for correctness

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

Our solution - Rebuilding strategy
Key idea: Reuse existing data structures during rebuilding
Rebuilding moments: When $m = k^2$ (perfect square)
Strategy:
Create new empty structures D_0', D_1'
Reuse $D_i \to D_{i+2}'$ for $i = 0, \ldots, k-1$
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

1. Explain our key insight: reuse existing data structures
2. Show rebuilding moments: when m is a perfect square ($m = k^2$)
3. ——— SKIP SLIDE ———
4. Walk through the three-step strategy:
5. 1. Create new empty structures D_0', D_1'
6. 2. Reuse $D_i to D_{i+2}'$ for $i = 0, ..., k-1$
7. 3. Apply missing updates to each D_i'
8. ——— SKIP SLIDE ———
9. Present the key lemma: every update in D_i is within first $(i+2)(k+1)$ updates
10. ——— SKIP SLIDE ———
11. Analyze time complexity: $O(mlogn)$ total, $O(\sqrt{m}logn)$ amortized
12. This sets up the detailed algorithm in the next slide
13. Key insight: we can reuse most of the work from previous structures
14. The offset (i+2) is crucial for correctness

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 edge
   - $t'_i \leftarrow p.\text{time}$
   - $\textsc{addEdges}(S, t_{i-2}, t'_i, D'_i)$
5. Return $k+1, D', t'$

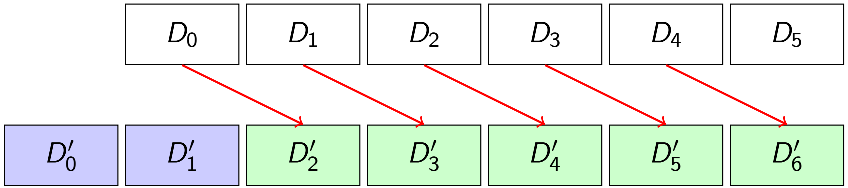
Rebuilding algorithm
1. $D'_0 \leftarrow \textsc{newIncrementalMSF}()$
2. $D'_1 \leftarrow \textsc{newIncrementalMSF}()$
3. For $i = 2$ to $k+1$: $D'_i \leftarrow D_{i-2}$ $\triangleright$ reuse existing
4. For $i = 1$ to $k+1$:
   - $p \leftarrow \textsc{kth}(S, i(k+1))$ $\triangleright$ $i(k+1)$th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

1. Show the step-by-step rebuilding algorithm
2. Explain how we create new empty structures D'_0, D'_1
3. Show how we reuse existing structures with offset: D_i becomes D'_{i+2}
4. Walk through the process of applying missing updates
5. ——— SKIP SLIDE ———
6. Explain the key insight: D_i becomes D'_{i+2} with offset
7. Analyze time complexity: $O(mlogn)$ total, $O(\sqrt{m}logn)$ amortized
8. Space complexity: $\Theta(m\sqrt{m})$ - this is our trade-off
9. This leads to our results in the next slide
10. Key insight: the algorithm is surprisingly simple despite its power
11. The visual shows the reuse pattern clearly

Rebuilding algorithm

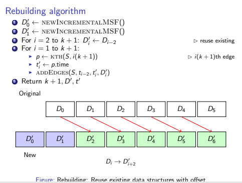
1. $D'_0 \leftarrow \textsc{newIncrementalMSF}()$
2. $D'_1 \leftarrow \textsc{newIncrementalMSF}()$
3. For $i = 2$ to $k + 1$: $D'_i \leftarrow D_{i-2}$ $\quad \triangleright$ reuse existing
4. For $i = 1$ to $k + 1$:
   - $p \leftarrow \textsc{kth}(S, i(k+1))$ $\quad \triangleright$ $i(k+1)$th edge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

Original



New

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



1. Show the step-by-step rebuilding algorithm
2. Explain how we create new empty structures D'_0, D'_1
3. Show how we reuse existing structures with offset: D_i becomes D'_{i+2}
4. Walk through the process of applying missing updates
5. ——— SKIP SLIDE ———
6. Explain the key insight: D_i becomes D'_{i+2} with offset
7. Analyze time complexity: $O(m \log n)$ total, $O(\sqrt{m} \log n)$ amortized
8. Space complexity: $\Theta(m\sqrt{m})$ - this is our trade-off
9. This leads to our results in the next slide
10. Key insight: the algorithm is surprisingly simple despite its power
11. The visual shows the reuse pattern clearly

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 $\mathcal{O}(m)$

1. Summarize our main theoretical contribution
2. Emphasize that we don't need persistent data structures - this is the key advantage
3. Show we achieve the same time complexity as Demaine et al.: O$\sqrt{m}$ per operation
4. Present our MSF implementation results: O$\sqrt{m} log n$ per operation
5. Highlight that we removed the fixed m and time range restrictions
6. This demonstrates the practical value of our approach
7. Key insight: we provide a simpler alternative to persistent data structures
8. Space trade-off: Theta$m\sqrt{m}$vs Om - but much simpler implementation
9. Our approach is more practical for many applications

Results

Our contribution

- **General transformation:** Partial $\to$ 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**

Results
Our contribution
- General transformation: Partial → 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

1. Summarize our main theoretical contribution
2. Emphasize that we don't need persistent data structures - this is the key advantage
3. Show we achieve the same time complexity as Demaine et al.: O$\sqrt{m}$ per operation
4. Present our MSF implementation results: O$\sqrt{m} logn$ per operation
5. Highlight that we removed the fixed m and time range restrictions
6. This demonstrates the practical value of our approach
7. Key insight: we provide a simpler alternative to persistent data structures
8. Space trade-off: Theta$m\sqrt{m}$vs Om - but much simpler implementation
9. Our approach is more practical for many applications

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

1. Emphasize the general applicability of our approach
2. Explain how to extend for full retroactivity with removals
3. Show the adapted rebuilding trigger condition
4. Explain how to handle both insertions and removals
5. List the requirements: partially retroactive, rollback capability
6. This shows how our approach can be extended for full functionality
7. Key insight: our method works for any partially retroactive data structure
8. The rebuilding frequency changes but the core idea remains the same
9. This demonstrates the generality of our approach

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

1. Emphasize the general applicability of our approach
2. Explain how to extend for full retroactivity with removals
3. Show the adapted rebuilding trigger condition
4. Explain how to handle both insertions and removals
5. List the requirements: partially retroactive, rollback capability
6. This shows how our approach can be extended for full functionality
7. Key insight: our method works for any partially retroactive data structure
8. The rebuilding frequency changes but the core idea remains the same
9. This demonstrates the generality of our approach

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

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

1. Emphasize the general applicability of our approach
2. Explain how to extend for full retroactivity with removals
3. Show the adapted rebuilding trigger condition
4. Explain how to handle both insertions and removals
5. List the requirements: partially retroactive, rollback capability
6. This shows how our approach can be extended for full functionality
7. Key insight: our method works for any partially retroactive data structure
8. The rebuilding frequency changes but the core idea remains the same
9. This demonstrates the generality of our approach

Thank you!

Questions?

Thank you!

Questions?

1. Invite questions from the audience
2. Be prepared to answer questions about:
3. * The rebuilding algorithm details
4. * Space vs time trade-offs
5. * Implementation challenges
6. * Comparison with persistent data structures
7. * Applications beyond MSF
8. Key points to emphasize if asked:
9. * Our approach is simpler to implement
10. * Same time complexity as Demaine et al.
11. * No persistent data structure requirement
12. * General applicability to any partially retroactive structure
13. Thank the audience for their attention