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 everyone. My name is Felipe Noronha, and today I'll be presenting our work with Professor Cristina Fernandes from IME-USP.
2. Our paper details a method for transforming partially retroactive data structures into fully retroactive ones.
3. This work is motivated by a practical limitation in the well-known 2007 transformation by Demaine, Iacono, and Langerman...
4. ...and it also builds upon a 2022 solution by Junior and Seabra.
5. Our key contribution is a method to achieve this transformation with the same time complexity, but *without* the need for complex persistent data structures.
6. To illustrate our approach, we'll use the minimum spanning forest problem as our main example. So, let's start by defining what that is.

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. Lets start of by defining what is a spanning tree on a graph G with a set of vertices and edges
2. A spanning tree will be a tree will all the vertices of G
3. ——— - SKIP SLIDE ———-
4. It will have 3 main properties: connected (path between any two vertices), acyclic (no cycles), contains exactly n-1 edges for n vertices
5. ——— - SKIP SLIDE ———-
6. Show visual example with graph G (blue edges) and spanning tree T (red wavy edges)
7. In the example: 8 vertices, so spanning tree has exactly 7 edges
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. Lets start of by defining what is a spanning tree on a graph G with a set of vertices and edges
2. A spanning tree will be a tree will all the vertices of G
3. ——— - SKIP SLIDE ——— -
4. It will have 3 main properties: connected (path between any two vertices), acyclic (no cycles), contains exactly n-1 edges for n vertices
5. ——— - SKIP SLIDE ——— -
6. Show visual example with graph G (blue edges) and spanning tree T (red wavy edges)
7. In the example: 8 vertices, so spanning tree has exactly 7 edges
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

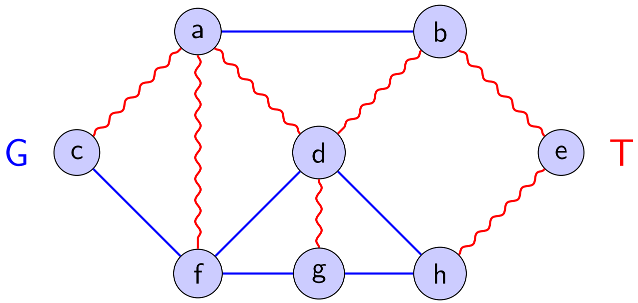
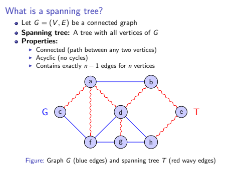


Figure: Graph G (blue edges) and spanning tree T (red wavy edges)



1. Lets start of by defining what is a spanning tree on a graph G with a set of vertices and edges
2. A spanning tree will be a tree will all the vertices of G
3. ———- SKIP SLIDE ———-
4. It will have 3 main properties: connected (path between any two vertices), acyclic (no cycles), contains exactly n-1 edges for n vertices
5. ———- SKIP SLIDE ———-
6. Show visual example with graph G (blue edges) and spanning tree T (red wavy edges)
7. In the example: 8 vertices, so spanning tree has exactly 7 edges
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. Now, let's add weights or costs to the edges. In a weighted graph, a Minimum Spanning Tree, or MST, is a spanning tree that has the minimum possible total cost. It's an optimization problem.
2. ——— - SKIP SLIDE ——— -
3. This concept generalizes to disconnected graphs as well. We call this a Minimum Spanning Forest, or MSF, which is simply the collection of MSTs for each connected component.
4. ——— - SKIP SLIDE ——— -
5. In the visual example, you can see the same graph as before, but now with costs on the edges. The red wavy edges again show the tree, but this time, they've been chosen to be the MST.
6. If we sum the costs of the red edges, we get a total of 14. Any other spanning tree you could build for this graph would have a total cost greater than or equal to 14.
7. This idea of maintaining an optimal-cost forest is central to our problem. Specifically, how to maintain this optimality as the graph changes.

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. Now, let's add weights or costs to the edges. In a weighted graph, a Minimum Spanning Tree, or MST, is a spanning tree that has the minimum possible total cost. It's an optimization problem.
2. ———- SKIP SLIDE ———-
3. This concept generalizes to disconnected graphs as well. We call this a Minimum Spanning Forest, or MSF, which is simply the collection of MSTs for each connected component.
4. ———- SKIP SLIDE ———-
5. In the visual example, you can see the same graph as before, but now with costs on the edges. The red wavy edges again show the tree, but this time, they've been chosen to be the MST.
6. If we sum the costs of the red edges, we get a total of 14. Any other spanning tree you could build for this graph would have a total cost greater than or equal to 14.
7. This idea of maintaining an optimal-cost forest is central to our problem. Specifically, how to maintain this optimality as the graph changes.

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

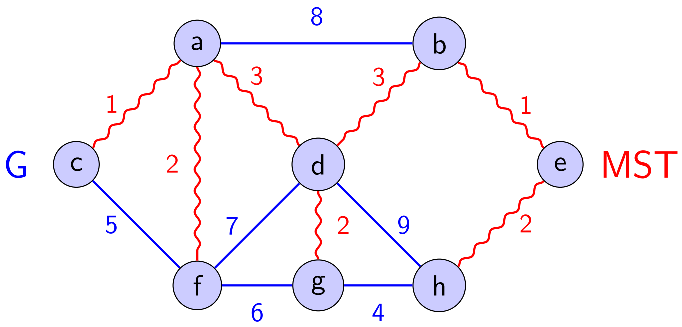
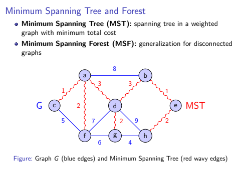


Figure: Graph G (blue edges) and Minimum Spanning Tree (red wavy edges)



1. Now, let's add weights or costs to the edges. In a weighted graph, a Minimum Spanning Tree, or MST, is a spanning tree that has the minimum possible total cost. It's an optimization problem.
2. ———- SKIP SLIDE ———-
3. This concept generalizes to disconnected graphs as well. We call this a Minimum Spanning Forest, or MSF, which is simply the collection of MSTs for each connected component.
4. ———- SKIP SLIDE ———-
5. In the visual example, you can see the same graph as before, but now with costs on the edges. The red wavy edges again show the tree, but this time, they've been chosen to be the MST.
6. If we sum the costs of the red edges, we get a total of 14. Any other spanning tree you could build for this graph would have a total cost greater than or equal to 14.
7. This idea of maintaining an optimal-cost forest is central to our problem. Specifically, how to maintain this optimality as the graph changes.

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. So, this brings us to the Incremental MSF problem. The goal is to maintain a Minimum Spanning Forest in a graph that grows over time.
2. Crucially, the graph starts empty, and edges are only added one by one.
3. ———- SKIP SLIDE ———-
4. This problem is defined by two operations: *add_edge*, which inserts a new weighted edge, and *get_msf*, which returns the current minimum spanning forest.
5. ———- SKIP SLIDE ———-
6. The solution to this was given by Frederickson in 1983. He used a dynamic data structure called link-cut trees which achieves a $O(\log n)$ amortized time per edge addition.
7. This incremental solution is the foundation for the *retroactive* version, which is what we're interested in.

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. So, this brings us to the Incremental MSF problem. The goal is to maintain a Minimum Spanning Forest in a graph that grows over time.
2. Crucially, the graph starts empty, and edges are only added one by one.
3. ——– SKIP SLIDE ——–
4. This problem is defined by two operations: *add_edge*, which inserts a new weighted edge, and *get_msf*, which returns the current minimum spanning forest.
5. ——– SKIP SLIDE ——–
6. The solution to this was given by Frederickson in 1983. He used a dynamic data structure called link-cut trees which achieves a $O(\log n)$ amortized time per edge addition.
7. This incremental solution is the foundation for the *retroactive* version, which is what we're interested in.

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. So, this brings us to the Incremental MSF problem. The goal is to maintain a Minimum Spanning Forest in a graph that grows over time.
2. Crucially, the graph starts empty, and edges are only added one by one.
3. ——— SKIP SLIDE ———
4. This problem is defined by two operations: *add_edge*, which inserts a new weighted edge, and *get_msf*, which returns the current minimum spanning forest.
5. ——— SKIP SLIDE ———
6. The solution to this was given by Frederickson in 1983. He used a dynamic data structure called link-cut trees which achieves a $O(\log n)$ amortized time per edge addition.
7. This incremental solution is the foundation for the *retroactive* version, which is what we're interested in.

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. So, what was Frederickson's key insight? He showed that link-cut trees can efficiently maintain this dynamic forest.
2. ——— SKIP SLIDE ———
3. Link-cut trees provide the exact operations we need, all in $O(\log n)$ amortized time: *find_max* to find the most expensive edge on a path, *link* to add a weighted edge, and *cut* to remove one.
4. ——— SKIP SLIDE ———
5. With this, we can construct a straightforward algorithm that supports adding a new edge (u, v, w):
6. First, we check if u and v are already connected. If they're not, the new edge can't create a cycle, so we just add it to the forest using *link*.
7. If they *are* connected, adding this new edge creates a cycle. We find the most expensive edge on the path in that cycle using *find_max*.
8. If our new edge's cost w is cheaper than that maximum cost, we swap them: we *cut* the old, expensive edge and *link* our new, cheaper edge .
9. ——— SKIP SLIDE ———
10. With these steps using LCT operations, the time per edge addition is $O(\log n)$ 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

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. So, what was Frederickson's key insight? He showed that link-cut trees can efficiently maintain this dynamic forest.
2. ———- SKIP SLIDE ———-
3. Link-cut trees provide the exact operations we need, all in $O(\log n)$ amortized time: *find_max* to find the most expensive edge on a path, *link* to add a weighted edge, and *cut* to remove one.
4. ———- SKIP SLIDE ———-
5. With this, we can construct a straightforward algorithm that supports adding a new edge (u, v, w):
6. First, we check if u and v are already connected. If they're not, the new edge can't create a cycle, so we just add it to the forest using *link*.
7. If they *are* connected, adding this new edge creates a cycle. We find the most expensive edge on the path in that cycle using *find_max*.
8. If our new edge's cost w is cheaper than that maximum cost, we swap them: we *cut* the old, expensive edge and *link* our new, cheaper edge .
9. ———- SKIP SLIDE ———-
10. With these steps using LCT operations, the time per edge addition is $O(\log n)$ 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
- **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): 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

1. So, what was Frederickson's key insight? He showed that link-cut trees can efficiently maintain this dynamic forest.
2. ——— SKIP SLIDE ———
3. Link-cut trees provide the exact operations we need, all in $O(\log n)$ amortized time: *find_max* to find the most expensive edge on a path, *link* to add a weighted edge, and *cut* to remove one.
4. ——— SKIP SLIDE ———
5. With this, we can construct a straightforward algorithm that supports adding a new edge (u, v, w):
6. First, we check if u and v are already connected. If they're not, the new edge can't create a cycle, so we just add it to the forest using *link*.
7. If they *are* connected, adding this new edge creates a cycle. We find the most expensive edge on the path in that cycle using *find_max*.
8. If our new edge's cost w is cheaper than that maximum cost, we swap them: we *cut* the old, expensive edge and *link* our new, cheaper edge .
9. ——— SKIP SLIDE ———
10. With these steps using LCT operations, the time per edge addition is $O(\log n)$ 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
- **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): $\mathcal{O}(\log n)$ 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

1. So, what was Frederickson's key insight? He showed that link-cut trees can efficiently maintain this dynamic forest.
2. ———- SKIP SLIDE ———-
3. Link-cut trees provide the exact operations we need, all in $O(\log n)$ amortized time: *find_max* to find the most expensive edge on a path, *link* to add a weighted edge, and *cut* to remove one.
4. ———- SKIP SLIDE ———-
5. With this, we can construct a straightforward algorithm that supports adding a new edge (u, v, w):
6. First, we check if u and v are already connected. If they're not, the new edge can't create a cycle, so we just add it to the forest using *link*.
7. If they *are* connected, adding this new edge creates a cycle. We find the most expensive edge on the path in that cycle using *find_max*.
8. If our new edge's cost w is cheaper than that maximum cost, we swap them: we *cut* the old, expensive edge and *link* our new, cheaper edge .
9. ———- SKIP SLIDE ———-
10. With these steps using LCT operations, the time per edge addition is $O(\log n)$ amortized.

Incremental MSF example - Step 1

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

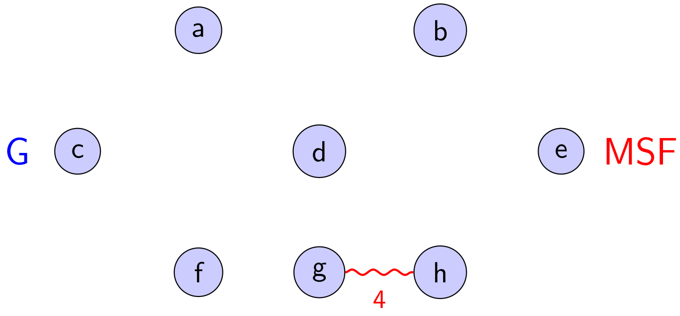
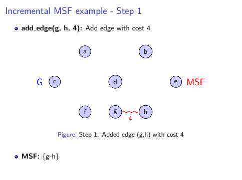


Figure: Step 1: Added edge (g,h) with cost 4

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



1. Let's walk through a quick example. We start with an empty graph.
2. First, we add edge (g, h) with cost 4.
3. Are 'g' and 'h' connected? No. So, by step 2 of the algorithm, we simply add the edge to our MSF.
4. The MSF is now just {g-h}.

Incremental MSF example - Step 2

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

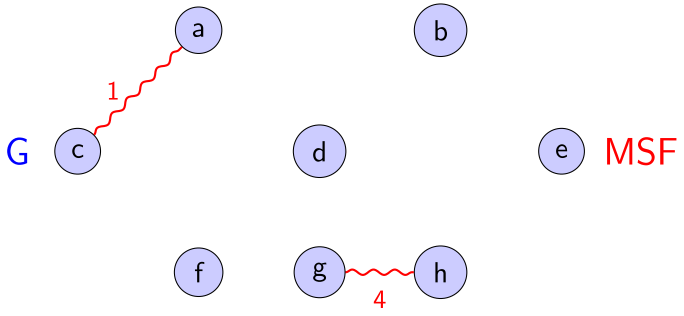
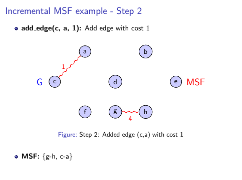


Figure: Step 2: Added edge (c,a) with cost 1

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



1. Next, we add (c, a) with cost 1.
2. Again, are 'c' and 'a' connected? No. They are in a different component from 'g' and 'h'.
3. So, we add it directly. The MSF now has two components: {g-h} and {c-a}.

Incremental MSF example - Step 3

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

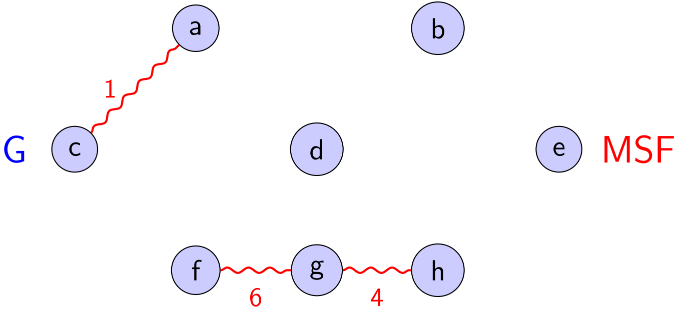
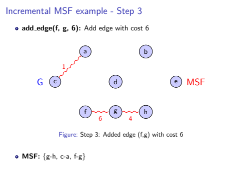


Figure: Step 3: Added edge (f,g) with cost 6

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



1. Now, add (f, g) with cost 6.
2. Are 'f' and 'g' connected? No. 'f' is isolated, and 'g' is in the {g-h} component.
3. We link them. The MSF now contains {c-a} and {f-g-h}.

Incremental MSF example - Step 4

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

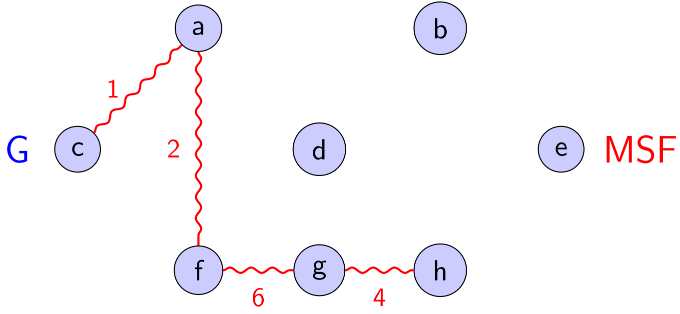
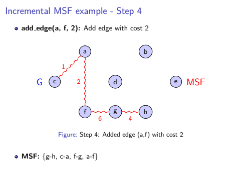


Figure: Step 4: Added edge (a,f) with cost 2

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



1. Add (a, f) with cost 2.
2. Are 'a' and 'f' connected? No. 'a' is in the {c-a} component and 'f' is in the {f-g-h} component.
3. We link these two components. Our forest now becomes a single tree, and all vertices shown so far are connected.

Incremental MSF example - Step 5

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

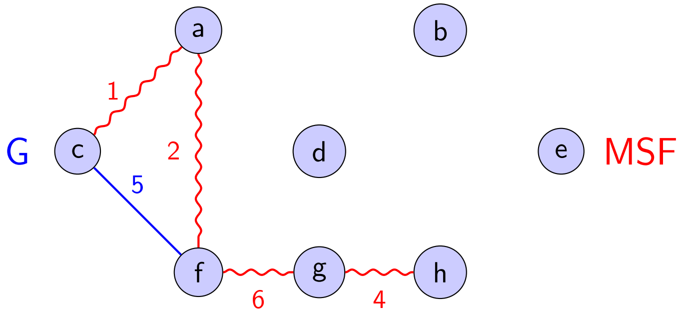
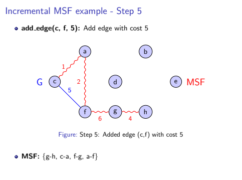


Figure: Step 5: Added edge (c,f) with cost 5

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



1. Okay, now we add (c, f) with cost 5. This is our first interesting case.
2. Are 'c' and 'f' connected? Yes, they are. Adding this edge will create a cycle: c-a-f-c.
3. So, we go to step 3. We find the max-cost edge on the path c-a-f. The edges are (c,a) with cost 1 and (a,f) with cost 2. The max cost is 2.
4. Our new edge costs 5. Since 5 is *not* less than the max cost of 2, we *do not* add this edge. It's discarded.
5. The MSF remains unchanged.

Incremental MSF example - Step 6

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

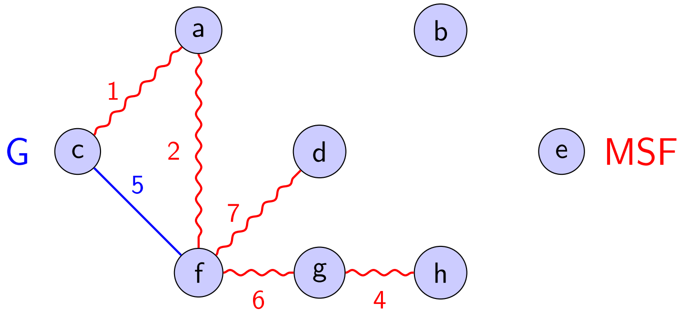
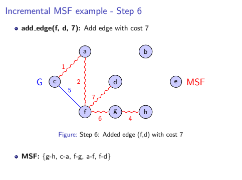


Figure: Step 6: Added edge (f,d) with cost 7

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



1. Next, add (f, d) with cost 7.
2. Are 'f' and 'd' connected? No. 'f' is in the main tree, but 'd' is a new, isolated vertex.
3. Therefore, we simply add the edge. The MSF is updated.

Incremental MSF example - Step 7

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

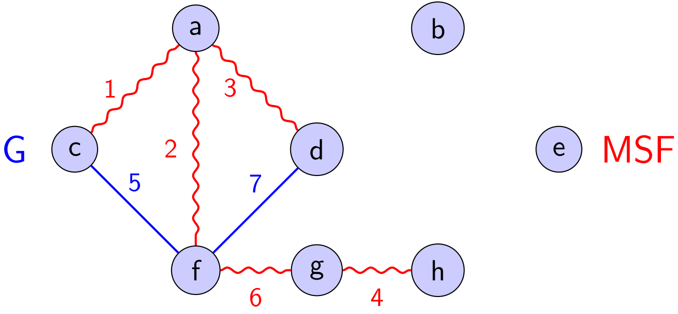
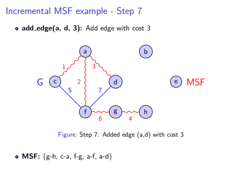


Figure: Step 7: Added edge (a,d) with cost 3

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



1. Now, add (a, d) with cost 3.
2. Are 'a' and 'd' connected? Yes. This creates the cycle a-f-d-a.
3. We find the max-cost edge on the path a-f-d. The edges are (a,f) with cost 2 and (f,d) with cost 7. The max cost is 7.
4. Our new edge costs 3. Since 3 *is* less than 7, we swap them.
5. We ‘cut‘ the expensive edge (f,d) and ‘link‘ our new, cheaper edge (a,d).
6. The MSF is now {g-h, c-a, f-g, a-f, a-d} and its total cost has improved.

Incremental MSF example - Step 8

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

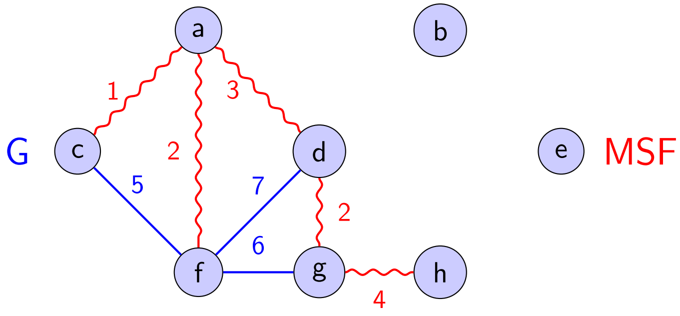
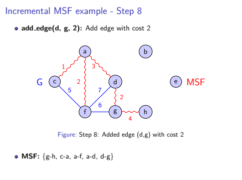


Figure: Step 8: Added edge (d,g) with cost 2

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



1. Finally, let's add (d, g) with cost 2.
2. Are 'd' and 'g' connected? Yes. This creates the cycle d-a-f-g-d.
3. We find the max-cost edge on the path d-a-f-g. The edges are (d,a) cost 3, (a,f) cost 2, and (f,g) cost 6. The max cost is 6, from edge (f,g).
4. Our new edge costs 2. Since 2 *is* less than 6, we swap them.
5. We ‘cut‘ edge (f,g) and ‘link‘ our new edge (d,g).
6. The MSF is updated again, and the total cost is now 12.

Incremental MSF example - Final Result

- **Continue adding edges...**

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

1. If we continue this process, adding all the remaining edges from our original graph...
2. ———- SKIP SLIDE ———-
3. ...we would eventually arrive at the final, optimal Minimum Spanning Tree. The one shown here, for example, has a total cost of 12.
4. So, to summarize, Frederickson's solution gives us an efficient $O(\log n)$ amortized time per *incremental* update.
5. It perfectly handles cycle detection and edge replacement to maintain optimality.
6. But this only answers queries about the *present*. What if we want to ask: "What did the MSF look like 10 updates ago?"
7. This is the core question of retroactivity. How do we efficiently query the past?

Incremental MSF example - Final Result

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

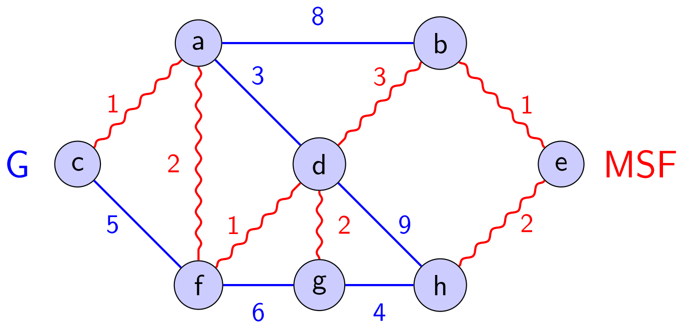
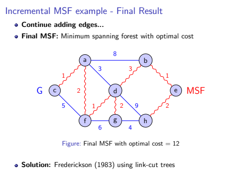


Figure: Final MSF with optimal cost = 12

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



1. If we continue this process, adding all the remaining edges from our original graph...
2. ———- SKIP SLIDE ———-
3. ...we would eventually arrive at the final, optimal Minimum Spanning Tree. The one shown here, for example, has a total cost of 12.
4. So, to summarize, Frederickson's solution gives us an efficient $O(\log n)$ amortized time per *incremental* update.
5. It perfectly handles cycle detection and edge replacement to maintain optimality.
6. But this only answers queries about the *present*. What if we want to ask: "What did the MSF look like 10 updates ago?"
7. This is the core question of retroactivity. How do we efficiently query the past?

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. In a normal data structure, the order of updates is important. The state of the structure, and thus the answers to queries, depends on this sequence.
2. This means we usually don't have a good way to go back and correct mistakes or insert operations we forgot.
3. ——— SKIP SLIDE ———
4. That's where retroactivity comes in. A retroactive data structure allows us to manipulate this sequence of updates.
5. ——— SKIP SLIDE ———
6. Specifically, it adds operations to: Insert a new update at some time t *in the past*...
7. ...Remove an update that *already happened* at time t....
8. ...and, most importantly, Query the state of the structure at *any* time t, not just the present.
9. The key challenge is how to do this efficiently, maintaining the state for every possible time.

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. In a normal data structure, the order of updates is important. The state of the structure, and thus the answers to queries, depends on this sequence.
2. This means we usually don't have a good way to go back and correct mistakes or insert operations we forgot.
3. ———- SKIP SLIDE ———-
4. That's where retroactivity comes in. A retroactive data structure allows us to manipulate this sequence of updates.
5. ———- SKIP SLIDE ———-
6. Specifically, it adds operations to: Insert a new update at some time t *in the past*...
7. ...Remove an update that *already happened* at time t....
8. ...and, most importantly, Query the state of the structure at *any* time t, not just the present.
9. The key challenge is how to do this efficiently, maintaining the state for every possible time.

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. In a normal data structure, the order of updates is important. The state of the structure, and thus the answers to queries, depends on this sequence.
2. This means we usually don't have a good way to go back and correct mistakes or insert operations we forgot.
3. ———- SKIP SLIDE ———-
4. That's where retroactivity comes in. A retroactive data structure allows us to manipulate this sequence of updates.
5. ———- SKIP SLIDE ———-
6. Specifically, it adds operations to: Insert a new update at some time t *in the past*...
7. ...Remove an update that *already happened* at time t....
8. ...and, most importantly, Query the state of the structure at *any* time t, not just the present.
9. The key challenge is how to do this efficiently, maintaining the state for every possible time.

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. There are a few different "flavors" of retroactivity. The most powerful is Fully Retroactive, which supports all the operations we just saw: insert, remove, and query, all at any time t.
2. ——— SKIP SLIDE ———
3. Partially Retroactive is more limited. You can still insert or remove updates anywhere in the timeline, but you can only query the state of the structure at the *current* time, "now". This is a key limitation.
4. ——— SKIP SLIDE ———
5. And finally, there's Semi-Retroactive, which is a bit of a mix. You can query at any time t and insert updates at any time, but you are *not allowed* to remove updates.
6. Generally, partially retroactive structures are much simpler to design. And this leads to an interesting challenge...

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. There are a few different "flavors" of retroactivity. The most powerful is Fully Retroactive, which supports all the operations we just saw: insert, remove, and query, all at any time t.
2. ——— SKIP SLIDE ———
3. Partially Retroactive is more limited. You can still insert or remove updates anywhere in the timeline, but you can only query the state of the structure at the *current* time, "now". This is a key limitation.
4. ——— SKIP SLIDE ———
5. And finally, there's Semi-Retroactive, which is a bit of a mix. You can query at any time t and insert updates at any time, but you are *not allowed* to remove updates.
6. Generally, partially retroactive structures are much simpler to design. And this leads to an interesting challenge...

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. There are a few different "flavors" of retroactivity. The most powerful is Fully Retroactive, which supports all the operations we just saw: insert, remove, and query, all at any time t.
2. ——— SKIP SLIDE ———
3. Partially Retroactive is more limited. You can still insert or remove updates anywhere in the timeline, but you can only query the state of the structure at the *current* time, "now". This is a key limitation.
4. ——— SKIP SLIDE ———
5. And finally, there's Semi-Retroactive, which is a bit of a mix. You can query at any time t and insert updates at any time, but you are *not allowed* to remove updates.
6. Generally, partially retroactive structures are much simpler to design. And this leads to an interesting challenge...

The challenge

Challenge

How to transform partial $\rightarrow$ full retroactivity?

The challenge
Challenge
How to transform partial → full retroactivity?

1. ...which is this: How can we transform a simple partially retroactive structure into a fully retroactive one?
2. ———- SKIP SLIDE ———-
3. The problem is clear: We *have* a structure that lets us query "now", but we *need* a structure that lets us query any time t in the past.
4. ———- SKIP SLIDE ———-
5. A general solution for this was proposed by Demaine, Iacono, and Langerman in 2007.
6. Their approach uses a classic technique called square-root decomposition.
7. The key insight is to store checkpoints, or snapshots, of the data structure's state at various points in time. Let's see how that works.

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

1. ...which is this: How can we transform a simple partially retroactive structure into a fully retroactive one?
2. ——– SKIP SLIDE ——–
3. The problem is clear: We *have* a structure that lets us query "now", but we *need* a structure that lets us query any time t in the past.
4. ——– SKIP SLIDE ——–
5. A general solution for this was proposed by Demaine, Iacono, and Langerman in 2007.
6. Their approach uses a classic technique called square-root decomposition.
7. The key insight is to store checkpoints, or snapshots, of the data structure's state at various points in time. Let's see how that works.

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)

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. ...which is this: How can we transform a simple partially retroactive structure into a fully retroactive one?
2. ——— SKIP SLIDE ———
3. The problem is clear: We *have* a structure that lets us query "now", but we *need* a structure that lets us query any time t in the past.
4. ——— SKIP SLIDE ———
5. A general solution for this was proposed by Demaine, Iacono, and Langerman in 2007.
6. Their approach uses a classic technique called square-root decomposition.
7. The key insight is to store checkpoints, or snapshots, of the data structure's state at various points in time. Let's see how that works.

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. Their paper presented this theorem: any partially retroactive data structure can be made fully retroactive.
2. The cost is an $O(\sqrt{m})$ slowdown per operation and $O(m)$ space, where m is the number of updates.
3. But there's a catch: this transformation *requires* a persistent version of the data structure. This is the key limitation we want to address.
4. ——— SKIP SLIDE ———
5. So, how does it work? The idea is to break the m updates into $\sqrt{m}$ blocks, each of size $\sqrt{m}$.
6. At the beginning of each block, we store a "checkpoint" of the data structure's state.
7. ——— SKIP SLIDE ———
8. Now, to query at some time t:
9. First, we find the closest checkpoint *before* t. We load this saved state.
10. Then, we "roll forward" by applying all the updates between that checkpoint and time t. There are at most $\sqrt{m}$ of them.
11. We answer the query, and then we "roll back" the changes to restore the checkpoint, which is where persistence comes in handy.

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:
- $O(\sqrt{m})$ slowdown per operation
- $O(m)$ space usage
- Requirement: Need persistent version of the data structure
- Key idea: Square-root decomposition
- Keep $\sqrt{m}$ checkpoints with data structure states

1. Their paper presented this theorem: any partially retroactive data structure can be made fully retroactive.
2. The cost is an $O(\sqrt{m})$ slowdown per operation and $O(m)$ space, where m is the number of updates.
3. But there's a catch: this transformation *requires* a persistent version of the data structure. This is the key limitation we want to address.
4. ———- SKIP SLIDE ———-
5. So, how does it work? The idea is to break the m updates into $\sqrt{m}$ blocks, each of size $\sqrt{m}$.
6. At the beginning of each block, we store a "checkpoint" of the data structure's state.
7. ———- SKIP SLIDE ———-
8. Now, to query at some time t:
9. First, we find the closest checkpoint *before* t. We load this saved state.
10. Then, we "roll forward" by applying all the updates between that checkpoint and time t. There are at most $\sqrt{m}$ of them.
11. We answer the query, and then we "roll back" the changes to restore the checkpoint, which is where persistence comes in handy.

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. Their paper presented this theorem: any partially retroactive data structure can be made fully retroactive.
2. The cost is an $O(\sqrt{m})$ slowdown per operation and $O(m)$ space, where m is the number of updates.
3. But there's a catch: this transformation *requires* a persistent version of the data structure. This is the key limitation we want to address.
4. ———- SKIP SLIDE ———-
5. So, how does it work? The idea is to break the m updates into $\sqrt{m}$ blocks, each of size $\sqrt{m}$.
6. At the beginning of each block, we store a "checkpoint" of the data structure's state.
7. ———- SKIP SLIDE ———-
8. Now, to query at some time t:
9. First, we find the closest checkpoint *before* t. We load this saved state.
10. Then, we "roll forward" by applying all the updates between that checkpoint and time t. There are at most $\sqrt{m}$ of them.
11. We answer the query, and then we "roll back" the changes to restore the checkpoint, which is where persistence comes in handy.

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. ...and that leads to the space problem. If we're naive and just store $\sqrt{m}$ *independent copies* of the data structure, one for each checkpoint...
2. ...and each copy can have up to m updates, our space usage explodes to $\Theta(m\sqrt{m})$.
3. ——— SKIP SLIDE ———
4. This is why Demaine et al. use persistent data structures. A persistent structure cleverly shares memory between versions, so all $\sqrt{m}$ checkpoints can be stored efficiently in just $O(m)$ total space.
5. ——— SKIP SLIDE ———
6. But this raises a practical problem: What if we don't have a persistent version of our data structure? Or what if it's just too complex to implement?
7. ——— SKIP SLIDE ———
8. This is our contribution. We propose a simple rebuilding strategy that *doesn't* require persistence.
9. We achieve the *same $O(\sqrt{m})$ time* per operation.
10. The trade-off is that we go back to using $\Theta(m\sqrt{m})$ space, but we argue this is a practical trade-off for a much, much simpler implementation.

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. ...and that leads to the space problem. If we're naive and just store $\sqrt{m}$ *independent copies* of the data structure, one for each checkpoint...
2. ...and each copy can have up to m updates, our space usage explodes to $\Theta(m\sqrt{m})$.
3. ———- SKIP SLIDE ———-
4. This is why Demaine et al. use persistent data structures. A persistent structure cleverly shares memory between versions, so all $\sqrt{m}$ checkpoints can be stored efficiently in just $O(m)$ total space.
5. ———- SKIP SLIDE ———-
6. But this raises a practical problem: What if we don't have a persistent version of our data structure? Or what if it's just too complex to implement?
7. ———- SKIP SLIDE ———-
8. This is our contribution. We propose a simple rebuilding strategy that *doesn't* require persistence.
9. We achieve the *same $O(\sqrt{m})$ time* per operation.
10. The trade-off is that we go back to using $\Theta(m\sqrt{m})$ space, but we argue this is a practical trade-off for a much, much simpler implementation.

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. ...and that leads to the space problem. If we're naive and just store $\sqrt{m}$ *independent copies* of the data structure, one for each checkpoint...
2. ...and each copy can have up to m updates, our space usage explodes to $\Theta(m\sqrt{m})$.
3. ———- SKIP SLIDE ———-
4. This is why Demaine et al. use persistent data structures. A persistent structure cleverly shares memory between versions, so all $\sqrt{m}$ checkpoints can be stored efficiently in just $O(m)$ total space.
5. ———- SKIP SLIDE ———-
6. But this raises a practical problem: What if we don't have a persistent version of our data structure? Or what if it's just too complex to implement?
7. ———- SKIP SLIDE ———-
8. This is our contribution. We propose a simple rebuilding strategy that *doesn't* require persistence.
9. We achieve the *same $O(\sqrt{m})$ time* per operation.
10. The trade-off is that we go back to using $\Theta(m\sqrt{m})$ space, but we argue this is a practical trade-off for a much, much simpler implementation.

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. ...and that leads to the space problem. If we're naive and just store $\sqrt{m}$ *independent copies* of the data structure, one for each checkpoint...
2. ...and each copy can have up to m updates, our space usage explodes to $\Theta(m\sqrt{m})$.
3. ———- SKIP SLIDE ———-
4. This is why Demaine et al. use persistent data structures. A persistent structure cleverly shares memory between versions, so all $\sqrt{m}$ checkpoints can be stored efficiently in just $O(m)$ total space.
5. ———- SKIP SLIDE ———-
6. But this raises a practical problem: What if we don't have a persistent version of our data structure? Or what if it's just too complex to implement?
7. ———- SKIP SLIDE ———-
8. This is our contribution. We propose a simple rebuilding strategy that *doesn't* require persistence.
9. We achieve the *same $O(\sqrt{m})$ time* per operation.
10. The trade-off is that we go back to using $\Theta(m\sqrt{m})$ space, but we argue this is a practical trade-off for a much, much simpler implementation.

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. Our starting point was the 2022 work by Junior and Seabra on a semi-retroactive MSF.
2. Remember, "semi-retroactive" means they can add edges at any time t in the past, and query the MSF at any time t, but they cannot *remove* edges.
3. ——— SKIP SLIDE ———
4. They also use a square-root decomposition. They maintain $\sqrt{m}$ checkpoints, t_i, spaced $\sqrt{m}$ updates apart.
5. ——— SKIP SLIDE ———
6. They use a set of data structures, D_i, where each D_i stores the incremental MSF containing all edges added *before* its checkpoint time t_i.
7. This approach gives them a final time complexity of $O(\sqrt{m}\log n)$ per operation.
8. However, their solution has some significant practical limitations...

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. Our starting point was the 2022 work by Junior and Seabra on a semi-retroactive MSF.
2. Remember, "semi-retroactive" means they can add edges at any time t in the past, and query the MSF at any time t, but they cannot *remove* edges.
3. ——— SKIP SLIDE ———
4. They also use a square-root decomposition. They maintain $\sqrt{m}$ checkpoints, t_i, spaced $\sqrt{m}$ updates apart.
5. ——— SKIP SLIDE ———
6. They use a set of data structures, D_i, where each D_i stores the incremental MSF containing all edges added *before* its checkpoint time t_i.
7. This approach gives them a final time complexity of $O(\sqrt{m}\log n)$ per operation.
8. However, their solution has some significant practical limitations...

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. Our starting point was the 2022 work by Junior and Seabra on a semi-retroactive MSF.
2. Remember, "semi-retroactive" means they can add edges at any time t in the past, and query the MSF at any time t, but they cannot *remove* edges.
3. ——— SKIP SLIDE ———
4. They also use a square-root decomposition. They maintain $\sqrt{m}$ checkpoints, t_i, spaced $\sqrt{m}$ updates apart.
5. ——— SKIP SLIDE ———
6. They use a set of data structures, D_i, where each D_i stores the incremental MSF containing all edges added *before* its checkpoint time t_i.
7. This approach gives them a final time complexity of $O(\sqrt{m} \log n)$ per operation.
8. However, their solution has some significant practical limitations...

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. ...namely, their approach has three main problems:
2. It assumes a *fixed m*, meaning you have to know the total number of operations in advance.
3. It assumes a *fixed time range*, with timestamps from 1 to m. This is not flexible.
4. And most importantly, it has *no rebuilding process*, so it can't handle a growing number of operations.
5. ——— SKIP SLIDE ———
6. Our goal was to remove these limitations, especially the need to know m in advance, while keeping the same efficiency.
7. ——— SKIP SLIDE ———
8. Our key insight is to introduce a rebuilding process.
9. The challenge is: How do we rebuild all $\sqrt{m}$ checkpoints without persistence, and without it being too slow?
10. Our solution is to cleverly *reuse* the data structures we already have.

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. ...namely, their approach has three main problems:
2. It assumes a *fixed m*, meaning you have to know the total number of operations in advance.
3. It assumes a *fixed time range*, with timestamps from 1 to m. This is not flexible.
4. And most importantly, it has *no rebuilding process*, so it can't handle a growing number of operations.
5. ——— SKIP SLIDE ———
6. Our goal was to remove these limitations, especially the need to know m in advance, while keeping the same efficiency.
7. ——— SKIP SLIDE ———
8. Our key insight is to introduce a rebuilding process.
9. The challenge is: How do we rebuild all $\sqrt{m}$ checkpoints without persistence, and without it being too slow?
10. Our solution is to cleverly *reuse* the data structures we already have.

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. ...namely, their approach has three main problems:
2. It assumes a *fixed m*, meaning you have to know the total number of operations in advance.
3. It assumes a *fixed time range*, with timestamps from 1 to m. This is not flexible.
4. And most importantly, it has *no rebuilding process*, so it can't handle a growing number of operations.
5. ———- SKIP SLIDE ———-
6. Our goal was to remove these limitations, especially the need to know m in advance, while keeping the same efficiency.
7. ———- SKIP SLIDE ———-
8. Our key insight is to introduce a rebuilding process.
9. The challenge is: How do we rebuild all $\sqrt{m}$ checkpoints without persistence, and without it being too slow?
10. Our solution is to cleverly *reuse* the data structures we already have.

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. Here's our strategy. The key idea is to reuse the existing structures.
2. We trigger a rebuild whenever the total number of operations, m, becomes a perfect square, say k^2.
3. ——– SKIP SLIDE ——–
4. When we rebuild, we're going from k checkpoints to $k+1$ new ones. Our strategy is:
5. 1. We create two new, *empty* structures, D'_0 and D'_1.
6. 2. Then, we *reuse* our old structures: the old D_0 becomes the new D'_2, the old D_1 becomes the new D'_3, and so on. We shift them over by two spots.
7. 3. Finally, we just apply the "missing" updates to each of these reused structures to get them up to date for their new checkpoint times.
8. ——– SKIP SLIDE ——–
9. The reason this is efficient is based on a key lemma we prove: The updates needed for the new D'_{i+2} are just a continuation of the updates from the old D_i. We don't have to restart from scratch.
10. ——– SKIP SLIDE ——–
11. This rebuilding process takes $O(m \log n)$ time in total.

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. Here's our strategy. The key idea is to reuse the existing structures.
2. We trigger a rebuild whenever the total number of operations, m, becomes a perfect square, say k^2.
3. ——— SKIP SLIDE ———
4. When we rebuild, we're going from k checkpoints to $k+1$ new ones. Our strategy is:
5. 1. We create two new, *empty* structures, D'_0 and D'_1.
6. 2. Then, we *reuse* our old structures: the old D_0 becomes the new D'_2, the old D_1 becomes the new D'_3, and so on. We shift them over by two spots.
7. 3. Finally, we just apply the "missing" updates to each of these reused structures to get them up to date for their new checkpoint times.
8. ——— SKIP SLIDE ———
9. The reason this is efficient is based on a key lemma we prove: The updates needed for the new D'_{i+2} are just a continuation of the updates from the old D_i. We don't have to restart from scratch.
10. ——— SKIP SLIDE ———
11. This rebuilding process takes $O(m \log n)$ time in total.

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:
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.

1. Here's our strategy. The key idea is to reuse the existing structures.
2. We trigger a rebuild whenever the total number of operations, m, becomes a perfect square, say k^2.
3. ———- SKIP SLIDE ———-
4. When we rebuild, we're going from k checkpoints to $k+1$ new ones. Our strategy is:
5. 1. We create two new, *empty* structures, D'_0 and D'_1.
6. 2. Then, we *reuse* our old structures: the old D_0 becomes the new D'_2, the old D_1 becomes the new D'_3, and so on. We shift them over by two spots.
7. 3. Finally, we just apply the "missing" updates to each of these reused structures to get them up to date for their new checkpoint times.
8. ———- SKIP SLIDE ———-
9. The reason this is efficient is based on a key lemma we prove: The updates needed for the new D'_{i+2} are just a continuation of the updates from the old D_i. We don't have to restart from scratch.
10. ———- SKIP SLIDE ———-
11. This rebuilding process takes $O(m \log n)$ time in total.

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 $i = 0, \dots, k-1$
  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

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 $i = 0, \dots, k-1$
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. Here's our strategy. The key idea is to reuse the existing structures.
2. We trigger a rebuild whenever the total number of operations, m, becomes a perfect square, say k^2.
3. ——— SKIP SLIDE ———
4. When we rebuild, we're going from k checkpoints to $k+1$ new ones. Our strategy is:
5. 1. We create two new, *empty* structures, D'_0 and D'_1.
6. 2. Then, we *reuse* our old structures: the old D_0 becomes the new D'_2, the old D_1 becomes the new D'_3, and so on. We shift them over by two spots.
7. 3. Finally, we just apply the "missing" updates to each of these reused structures to get them up to date for their new checkpoint times.
8. ——— SKIP SLIDE ———
9. The reason this is efficient is based on a key lemma we prove: The updates needed for the new D'_{i+2} are just a continuation of the updates from the old D_i. We don't have to restart from scratch.
10. ——— SKIP SLIDE ———
11. This rebuilding process takes $O(m \log n)$ time in total.

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 $k+1, D', t'$

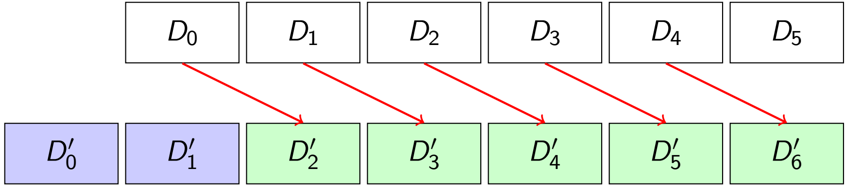
Rebuilding algorithm
1. $D'_0 \leftarrow \textsc{newIncrementalMSF}()$
2. $D'_1 \leftarrow \textsc{newIncrementalMSF}()$
3. For $i = 2$ to $k+1$: $D'_i \leftarrow D_{i-2}$ $\triangleright$ reuse existing
4. For $i = 1$ to $k+1$:
   - $p \leftarrow \textsc{kth}(S, i(k+1))$ $\triangleright$ $i(k+1)$th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

1. This slide shows the algorithm in more detail.
2. Lines 1 and 2 create the two new empty structures, D'_0 and D'_1.
3. Line 3 is the reuse: we loop from $i = 2$ up to $k+1$, and simply assign the old D_{i-2} to be the new D'_i. This is just a pointer swap; it's instant.
4. Line 4 is where the work happens. We loop through our new structures and apply the missing updates to each one, from its old checkpoint time t_{i-2} to its new checkpoint time t'_i.
5. ——— SKIP SLIDE ———
6. The diagram at the bottom visualizes this reuse. The new D'_0 and D'_1 are built from scratch, but all the others, D'_2 through D'_{k+1}, are just the old D_0 through D_{k-1}, shifted over and updated.
7. Again, this gives us the $O(\sqrt{m}\log n)$ amortized time...
8. ...but it requires $\Theta(m\sqrt{m})$ space, because we are storing these $\sqrt{m}$ independent copies.

Rebuilding algorithm

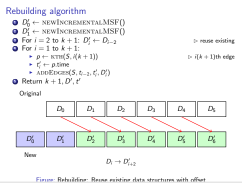
1. $D'_0 \leftarrow$ NEWINCREMENTALMSF()
2. $D'_1 \leftarrow$ NEWINCREMENTALMSF()
3. For $i = 2$ to $k+1$: $D'_i \leftarrow D_{i-2}$ ▷ reuse existing
4. For $i = 1$ to $k+1$:
   - $p \leftarrow$ KTH$(S, i(k+1))$ ▷ $i(k+1)$th edge
   - $t'_i \leftarrow p$.time
   - ADDEDGES(S, t_{i-2}, t'_i, D'_i)
5.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

Original



New

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



1. This slide shows the algorithm in more detail.
2. Lines 1 and 2 create the two new empty structures, D'_0 and D'_1.
3. Line 3 is the reuse: we loop from $i = 2$ up to $k+1$, and simply assign the old D_{i-2} to be the new D'_i. This is just a pointer swap; it's instant.
4. Line 4 is where the work happens. We loop through our new structures and apply the missing updates to each one, from its old checkpoint time t_{i-2} to its new checkpoint time t'_i.
5. ——— SKIP SLIDE ———
6. The diagram at the bottom visualizes this reuse. The new D'_0 and D'_1 are built from scratch, but all the others, D'_2 through D'_{k+1}, are just the old D_0 through D_{k-1}, shifted over and updated.
7. Again, this gives us the $O(\sqrt{m} \log n)$ amortized time...
8. ...but it requires $\Theta(m\sqrt{m})$ space, because we are storing these $\sqrt{m}$ independent copies.

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 → 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. So, to summarize our contributions:
2. We've developed a general transformation to take a partially retroactive data structure and make it fully retroactive.
3. Crucially, our method *does not require persistent data structures*.
4. We match the $O(\sqrt{m})$ slowdown per operation from the Demaine et al. paper...
5. ...at the cost of $\Theta(m\sqrt{m})$ space, which we argue is a very practical trade-off for simplicity.
6. ———- SKIP SLIDE ———-
7. Applying this to our test case, we get a semi-retroactive MSF implementation.
8. It supports adding edges and querying the MSF at any time t in $O(\sqrt{m}\log n)$ amortized time.
9. And, we have successfully removed the limitations from the previous work: our structure works *without* a fixed m or a fixed time range.

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**

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

1. So, to summarize our contributions:
2. We've developed a general transformation to take a partially retroactive data structure and make it fully retroactive.
3. Crucially, our method *does not require persistent data structures*.
4. We match the $O(\sqrt{m})$ slowdown per operation from the Demaine et al. paper...
5. ...at the cost of $\Theta(m\sqrt{m})$ space, which we argue is a very practical trade-off for simplicity.
6. ——— SKIP SLIDE ———
7. Applying this to our test case, we get a semi-retroactive MSF implementation.
8. It supports adding edges and querying the MSF at any time t in $O(\sqrt{m}\log n)$ amortized time.
9. And, we have successfully removed the limitations from the previous work: our structure works *without* a fixed m or a fixed time range.

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. Finally, while we focused on the semi-retroactive MSF, our transformation is general.
2. ——— SKIP SLIDE ———
3. It can be extended to support *full* retroactivity, meaning it can handle *removals* of operations as well.
4. To do this, we just adapt the rebuilding trigger. Instead of rebuilding only when m grows, we rebuild whenever the *number of blocks* (the floor of $\sqrt{m}$) changes, whether from insertions or removals.
5. This just means rebuilding happens a bit more frequently, but the amortized cost remains the same.
6. ——— SKIP SLIDE ———
7. The only requirements for our transformation to work are:
8. You must start with a partially retroactive data structure...
9. ...and it must have rollback capability, which the incremental MSF structure does.
10. If you have those, you can use our method to make it fully retroactive *without* persistence.

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. Finally, while we focused on the semi-retroactive MSF, our transformation is general.
2. ——— SKIP SLIDE ———
3. It can be extended to support *full* retroactivity, meaning it can handle *removals* of operations as well.
4. To do this, we just adapt the rebuilding trigger. Instead of rebuilding only when m grows, we rebuild whenever the *number of blocks* (the floor of $\sqrt{m}$) changes, whether from insertions or removals.
5. This just means rebuilding happens a bit more frequently, but the amortized cost remains the same.
6. ——— SKIP SLIDE ———
7. The only requirements for our transformation to work are:
8. You must start with a partially retroactive data structure...
9. ...and it must have rollback capability, which the incremental MSF structure does.
10. If you have those, you can use our method to make it fully retroactive *without* persistence.

Extending for full retroactivity

- **General applicability:** Works for any partially retroactive data structure
- **Supporting removals:** To achieve full retroactivity
  - Adapt rebuilding trigger: when $||\sqrt{m'}| - \lfloor\sqrt{m}\rfloor| \leq 1$
  - 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 $||\sqrt{m'}| - \lfloor\sqrt{m}\rfloor| \leq 1$
  - 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. Finally, while we focused on the semi-retroactive MSF, our transformation is general.
2. ——— SKIP SLIDE ———
3. It can be extended to support *full* retroactivity, meaning it can handle *removals* of operations as well.
4. To do this, we just adapt the rebuilding trigger. Instead of rebuilding only when m grows, we rebuild whenever the *number of blocks* (the floor of $\sqrt{m}$) changes, whether from insertions or removals.
5. This just means rebuilding happens a bit more frequently, but the amortized cost remains the same.
6. ——— SKIP SLIDE ———
7. The only requirements for our transformation to work are:
8. You must start with a partially retroactive data structure...
9. ...and it must have rollback capability, which the incremental MSF structure does.
10. If you have those, you can use our method to make it fully retroactive *without* persistence.

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