Dynamic Connectivity II

Ran Duan

In this lecture

- Worst-case dynamic connectivity in $\tilde{O}(m^{1/2})$
- Improves to $\tilde{O}(n^{1/2})$ by sparsification
- Subgraph connectivity in amortized $\tilde{O}(m^{2/3})$ update time

- $\tilde{O}(\bullet)$ hides poly-logarithmic factors
  - For example, $\tilde{O}(n^2)$ means $O(n^2 \bullet \log^k n)$ for some constant k.

About dynamic algorithms

- Measures of complexity:
  - Memory space to store the required data structures
  - Initial construction time for the data structure
  - Insertion/deletion time: time required to modify the data structure
    - Update time
  - Query time: time needed to answer an query

Overview of Dynamic Connectivity Results

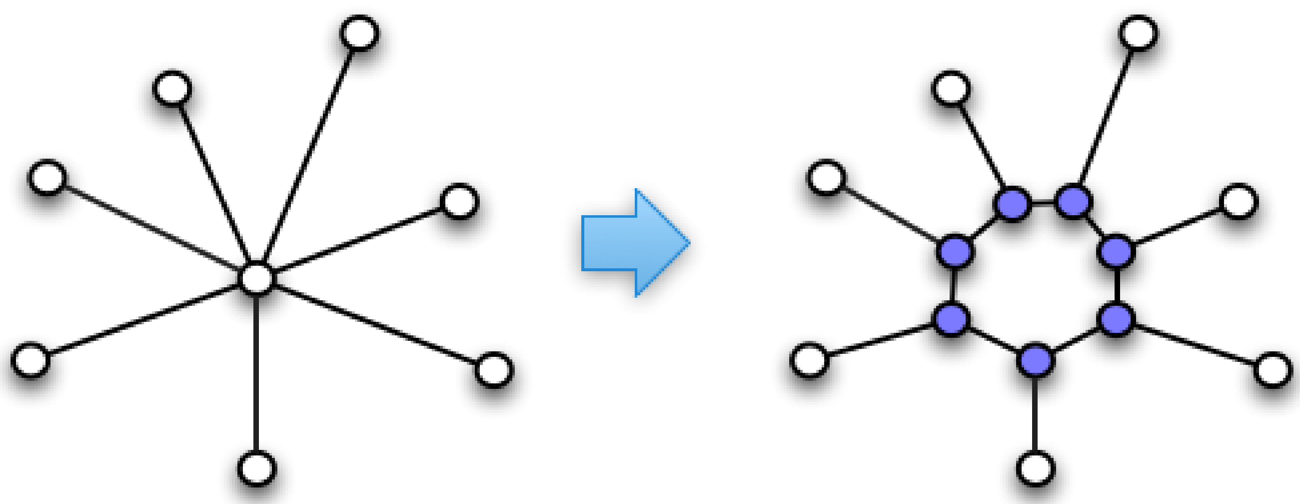
- Edge update—amortized time
  - Holm, Lichtenberg, and Thorup '1998: $O(\log^2 n)$
- Edge update—worst-case
  - Frederickson '1983: $\tilde{O}(m^{1/2})$
  - Eppstein, Galil, Italiano, Nissenzweig '1992: $\tilde{O}(n^{1/2})$
  - Not improved for 20 years, still a large gap to the amortized bound
  - Major challenge in dynamic algorithms

Overview of Frederickson's algorithm

- Make G degree-bounded
- Partition T into components with O(z) vertices
- Maintain the set of edges between every pair of components

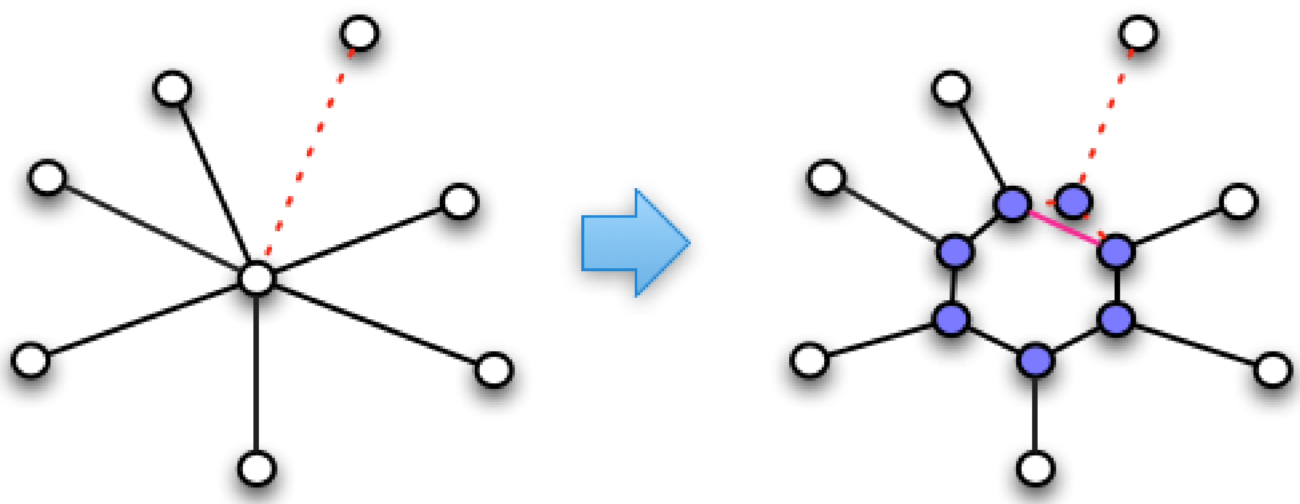
Degree-bounded

- Make the degree of every vertex no greater than 3
- By adding O(m) vertices
- Now |V|,|E| are both O(m)



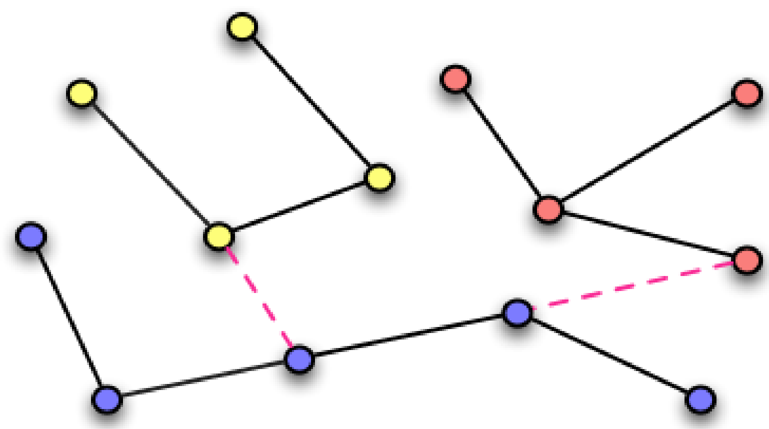
Degree-bounded

- Make the degree of every vertex no greater than 3
- By adding O(m) vertices
- Updating an edge in original graph only affects O(1) vertices and edges.



Partition of the spanning tree

- For a spanning tree T, find an edge set E' whose removal from T leaves subtrees with [z,3z-2] vertices.
  - So the number of such subtrees is O(m/z)



Algorithm for partition

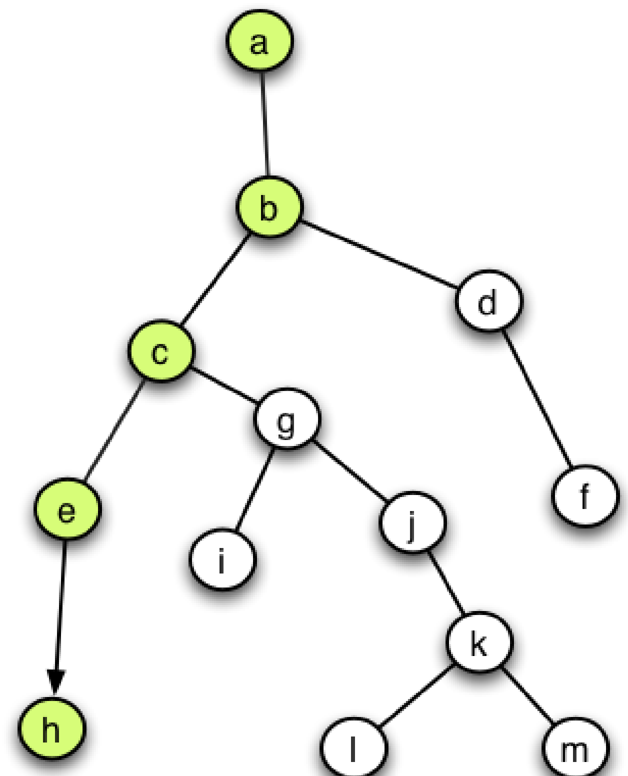
- O(m) time algorithm:
  - Starting from any leaf vertex r, call it the "root"
  - Run the depth-first search:

```
Search(v)
    clust={v}
    for each child w of v do
        clust:=clust∪Search(w)
    endfor
    if |clust|<z then return(clust)
                  else output(clust); return(Ø)
    endif
```

Algorithm for partition

- O(m) time algorithm:
  - Starting from any leaf vertex r, call it the “root”
  - Run the depth-first search:

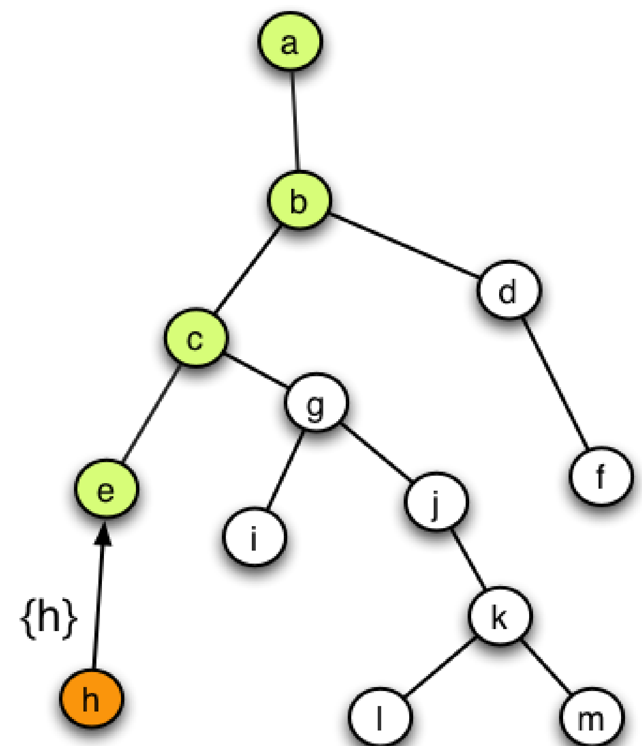
```
Search(v)
    clust={v}
    for each child w of v do
        clust:=clust∪Search(w)
    endfor
    if |clust|<z then return(clust)
                else output(clust); return(Ø)
    endif
```



Algorithm for partition

- O(m) time algorithm:
  - Starting from any leaf vertex r, call it the "root"
  - Run the depth-first search:

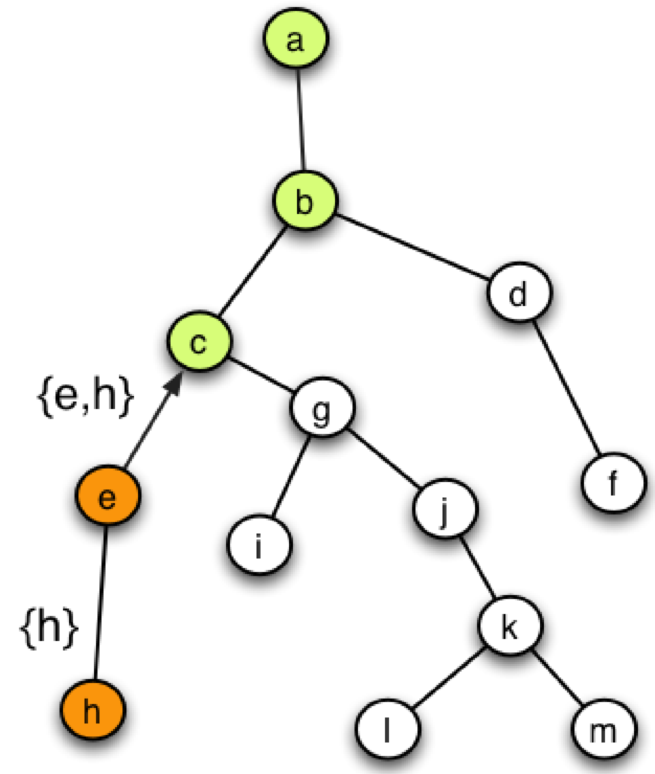
```
Search(v)
    clust={v}
    for each child w of v do
        clust:=clust∪Search(w)
    endfor
    if |clust|<z then return(clust)
                else output(clust); return(Ø)
    endif
```



Algorithm for partition

- O(m) time algorithm:
  - Starting from any leaf vertex r, call it the "root"
  - Run the depth-first search:

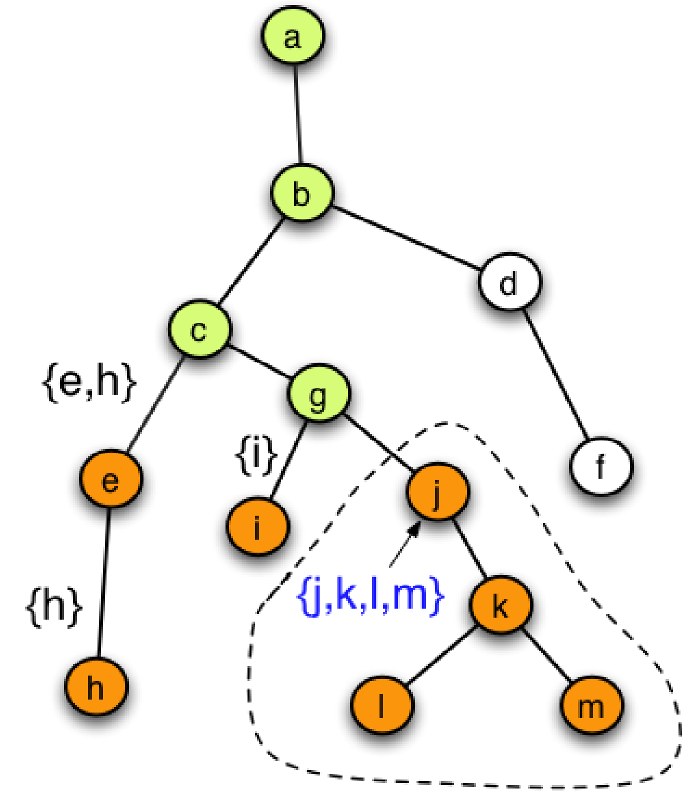
```
Search(v)
    clust={v}
    for each child w of v do
        clust:=clust∪Search(w)
    endfor
    if |clust|<z then return(clust)
                else output(clust); return(Ø)
    endif
```



Algorithm for partition

- O(m) time algorithm:
  - Starting from any leaf vertex r, call it the "root"
  - Run the depth-first search:

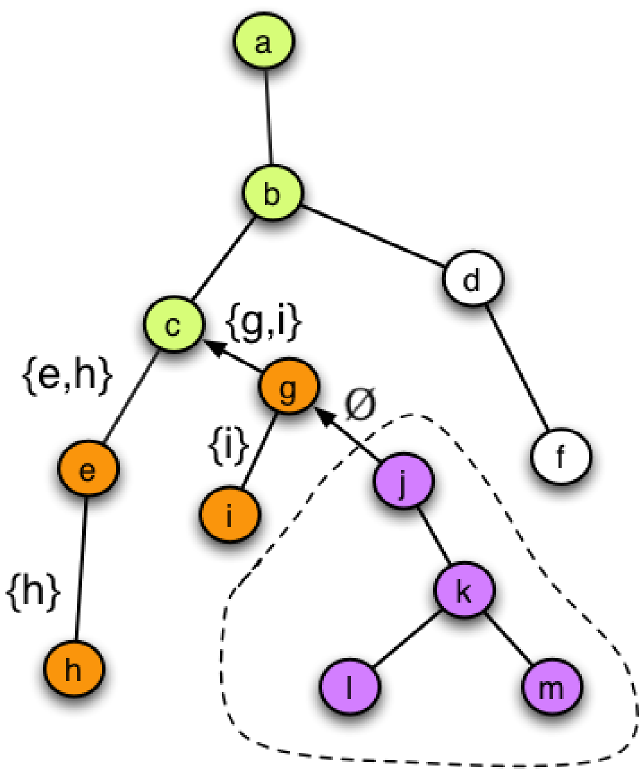
```
Search(v)
      clust={v}
      for each child w of v do
            clust:=clust∪Search(w)
      endfor
      if |clust|<z then return(clust)
                        else output(clust); return(Ø)
      endif
```



Algorithm for partition

- O(m) time algorithm:
  - Starting from any leaf vertex r, call it the “root”
  - Run the depth-first search:

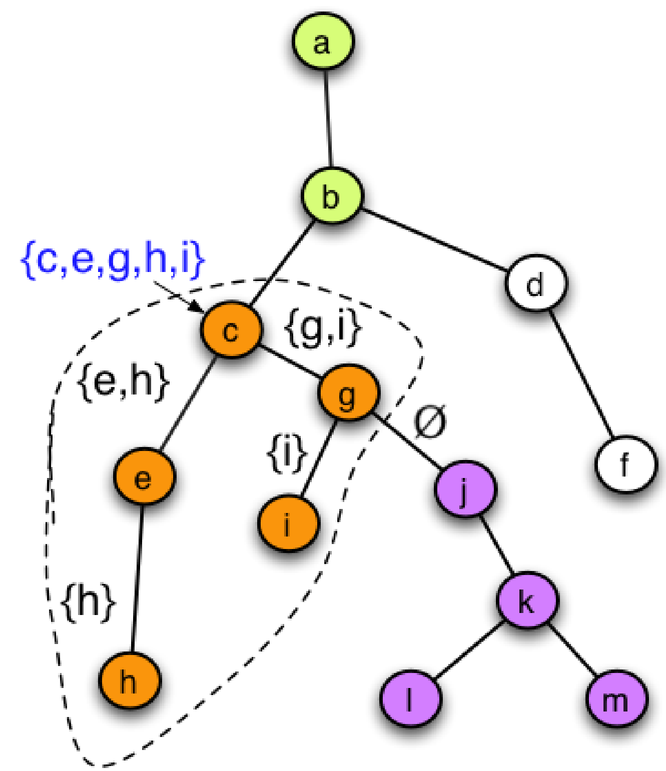
```
Search(v)
      clust={v}
      for each child w of v do
            clust:=clust∪Search(w)
      endfor
      if |clust|<z then return(clust)
                        else output(clust); return(Ø)
      endif
```



Algorithm for partition

- O(m) time algorithm:
  - Starting from any leaf vertex r, call it the "root"
  - Run the depth-first search:

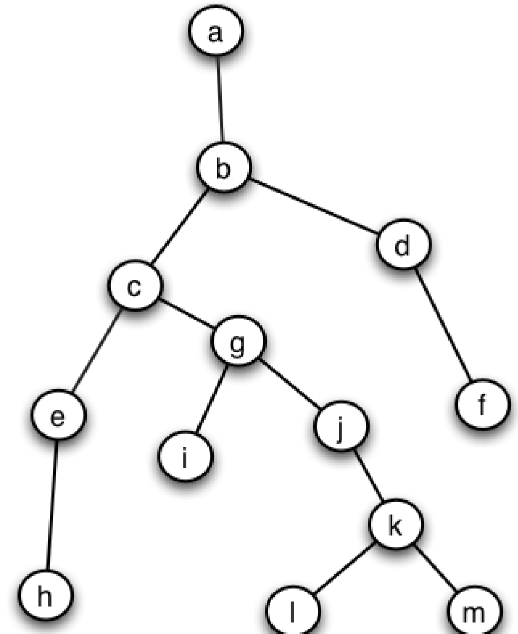
```
Search(v)
    clust={v}
    for each child w of v do
        clust:=clust∪Search(w)
    endfor
    if |clust|<z then return(clust)
                else output(clust); return(Ø)
    endif
```



  - If the procedure Search(r) finally return a non-empty set of size <z, union it with the last output set.

Correctness

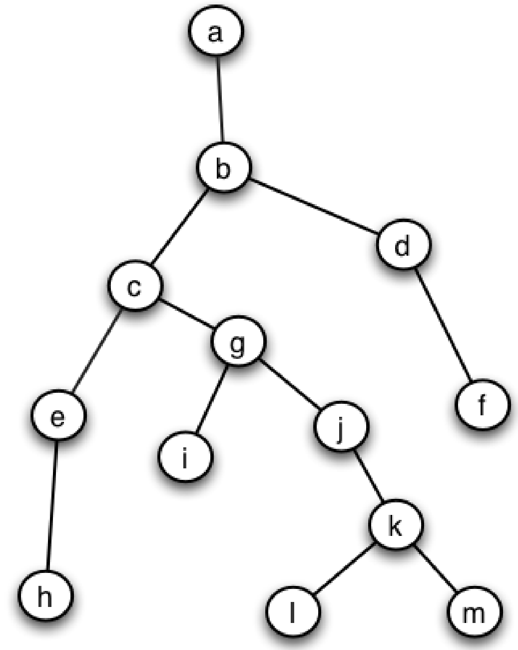
```
Search(v)
    clust={v}
    for each child w of v do
        clust:=clust∪Search(w)
    endfor
    if |clust|<z then return(clust)
                else output(clust); return(Ø)
    endif
```



- Since we start from a leaf, and the graph has degree bound 3, every vertex has at most 2 children
- The function Search(v) will always return a set of size ≤z-1
- So the output set has at most 2(z-1)+1=2z-1 vertices

Correctness

```
Search(v)
    clust={v}
    for each child w of v do
        clust:=clust∪Search(w)
    endfor
    if |clust|<z then return(clust)
            else output(clust); return(Ø)
    endif
```



- Since we start from a leaf, and the graph has degree bound 3, every vertex has at most 2 children
- The function Search will always return a set of size $\leq z-1$
- So the output set has at most $2(z-1)+1=2z-1$ and at least z vertices

- *If the procedure Search(r) finally return a non-empty set of size <z, union it with the last output set.*
- The bound of this set is $(z-1)+(2z-1)=3z-2$

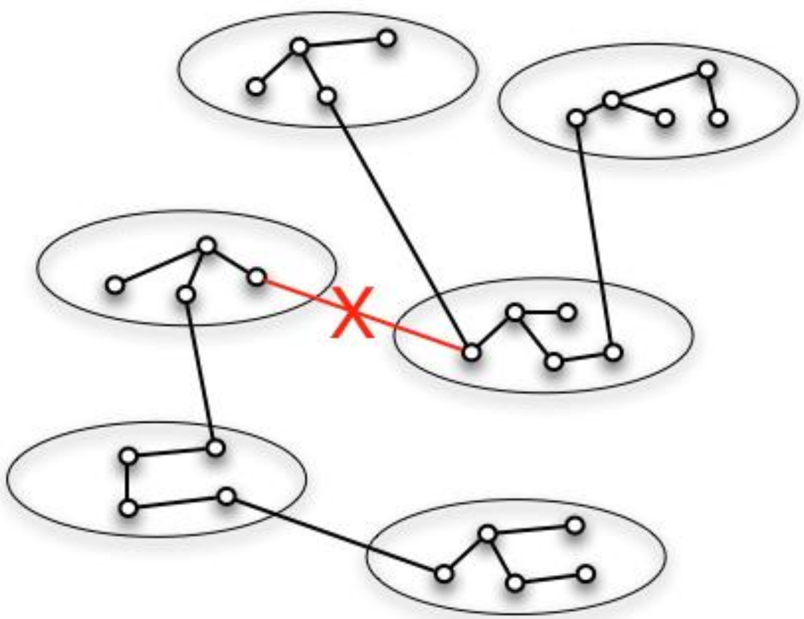
Partition of the spanning tree

- Thus for a spanning tree T, in O(m) time, we can find an edge set E' whose removal from T leaves vertex sets of size [z,3z-2]
  - So the number of such subtrees is O(m/z)
  - *Topological partition of order z*

- Let E_{ij} be the set of non-tree edges connecting sets V_i and V_j
  - The number of such edge sets is $O(m^2/z^2)$
  - $\left|\bigcup_j E_{ij}\right| = O(z)$

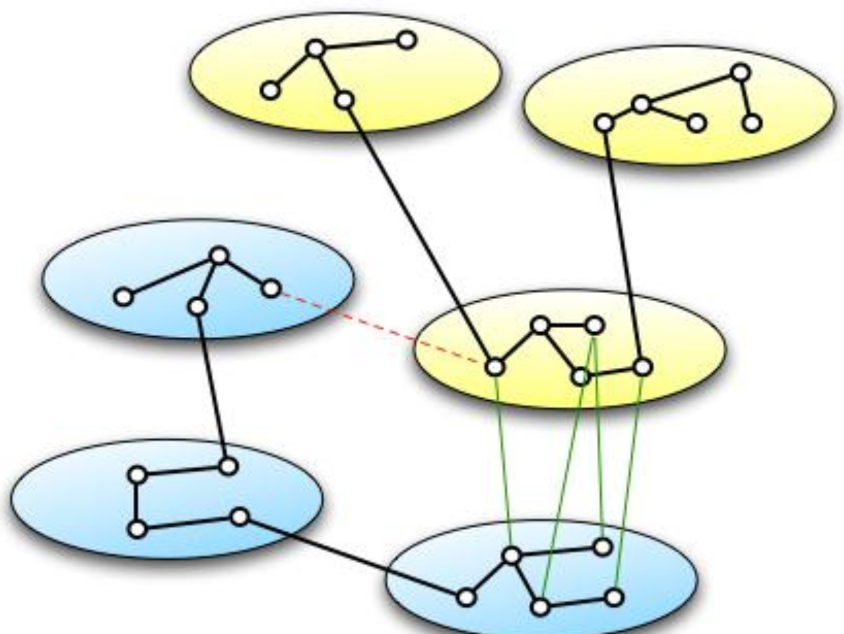
Update

- Insertion is trivial
- When deleting a tree edge e=(x,y)
  - If x,y are not in the same set, then we need to check all the edge set E_{ij} where V_i and V_j are in different subtrees



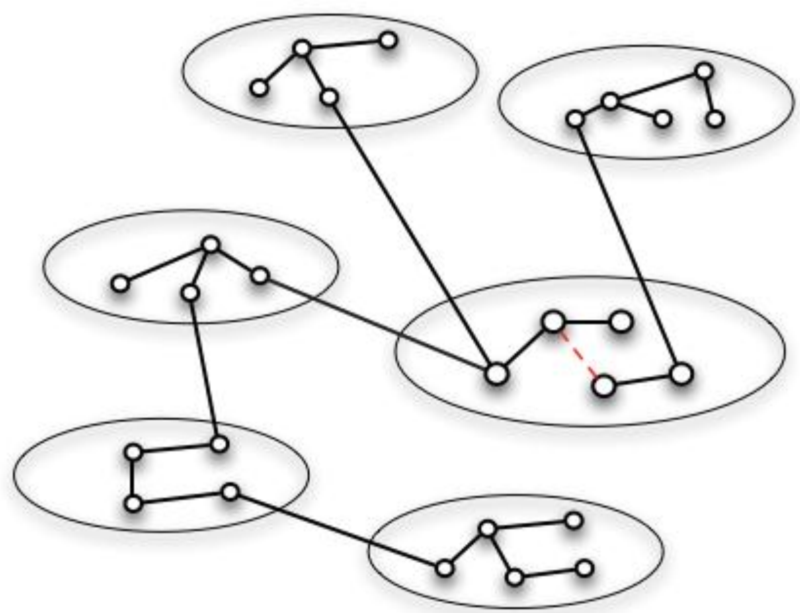
Update

- When deleting a tree edge e=(x,y)
  - If x,y are not in the same set, then we need to check all the edge set E_{ij} where V_i and V_j are in different subtrees
  - The number of such E_{ij} is $O(m^2/z^2)$



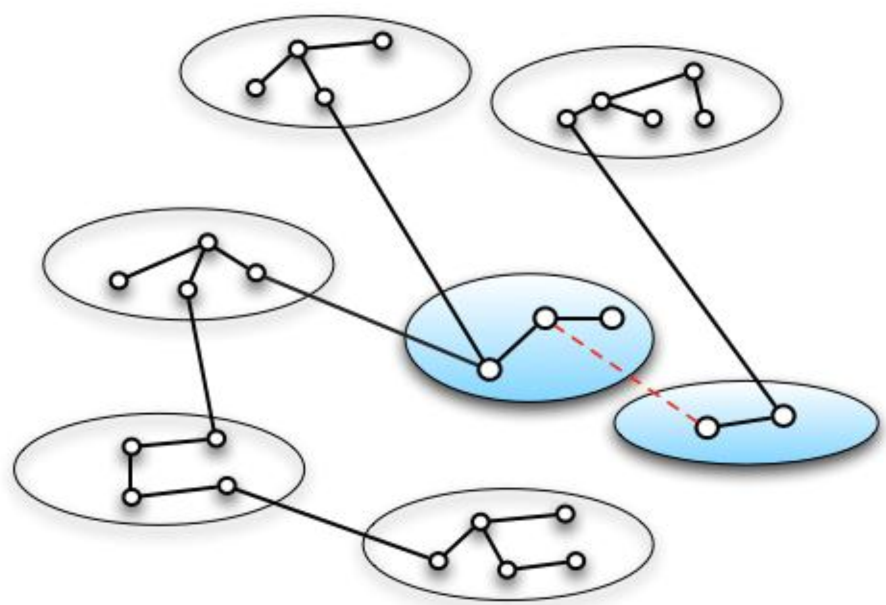
Update

- When deleting a tree edge $e=(x,y)$
  - If x,y are in the same set V_i, then V_i will be divided into two parts
  - Then we need to check all edges in E_{ii} and check E_{ij} for every V_j



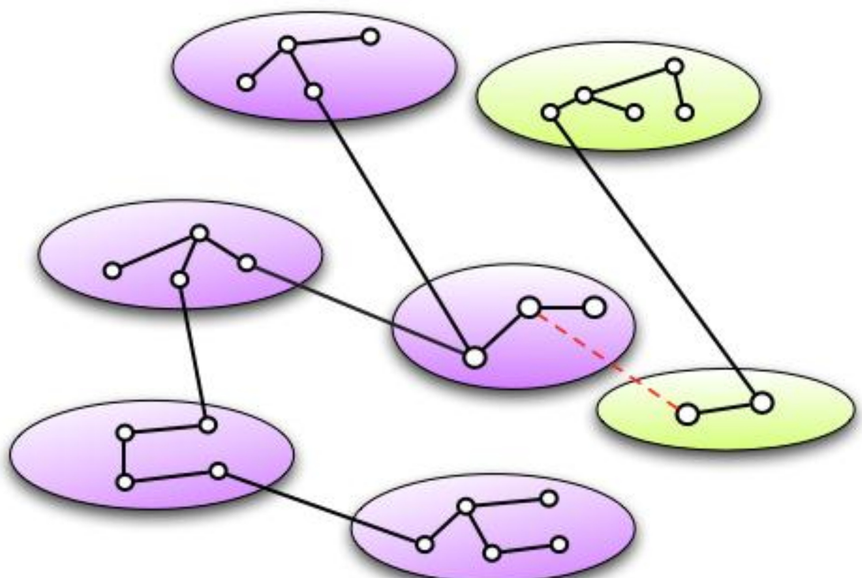
Update

- When deleting a tree edge $e=(x,y)$
  - If x,y are in the same set V_i, then V_i will be divided into two parts
  - Then we need to check all edges in E_{ii} and check E_{ij} for every V_j



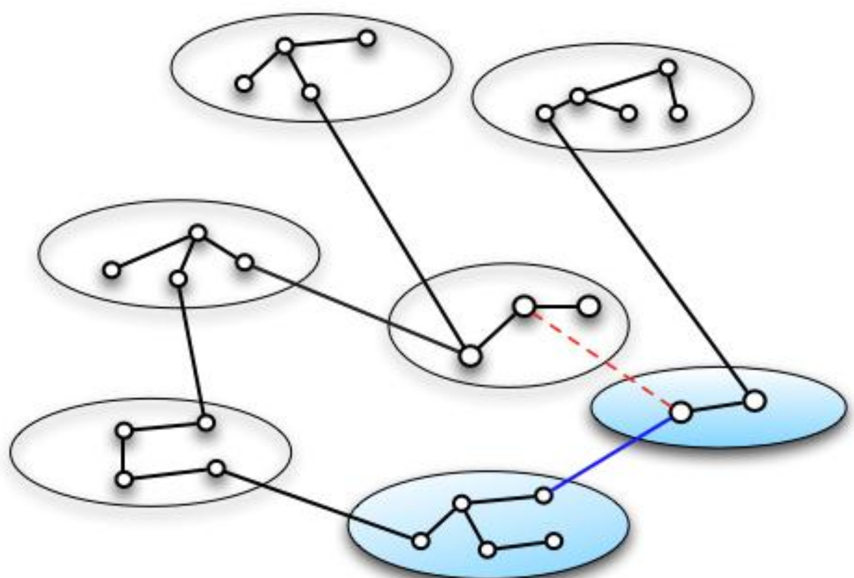
Update

- When deleting a tree edge $e=(x,y)$
  - If x,y are in the same set V_i, then V_i will be divided into two parts
  - Then we need to check all edges in E_{ii} and check E_{ij} for every V_j, and also other edge sets connecting two subtrees
    - Time needed: $O(z+m^2/z^2)$



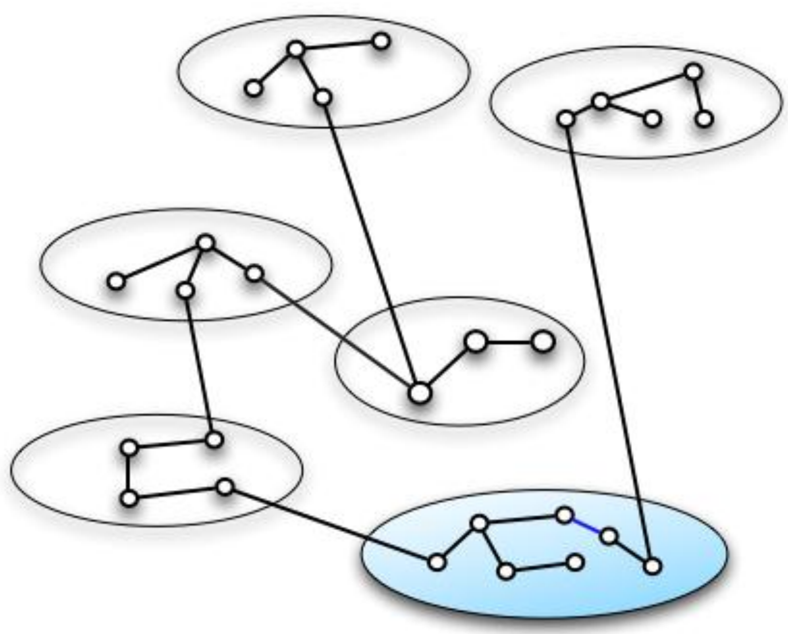
Deleting an edge inside one component

- We can see the two sets V’ and V” split from V_i may have <z vertices.
- Rearrange: Combine each of them with one adjacent vertex set and perform the *topological partition*.



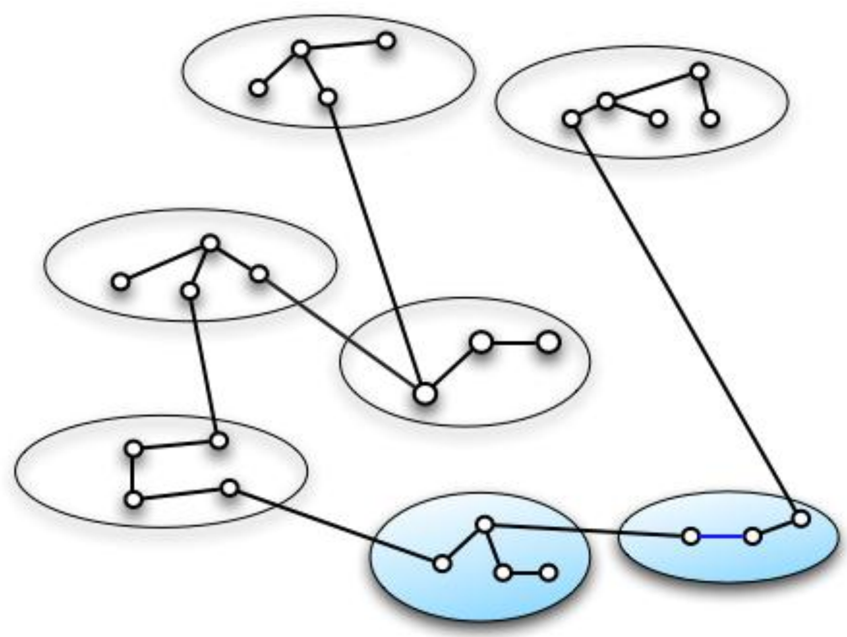
Deleting an edge inside one component

- We can see the two sets V' and V'' split from V_i may have <z vertices.
- Rearrange: Combine each of them with one adjacent vertex set and perform the *topological partition*.
  - The bound for the combined set: ≤4z-3



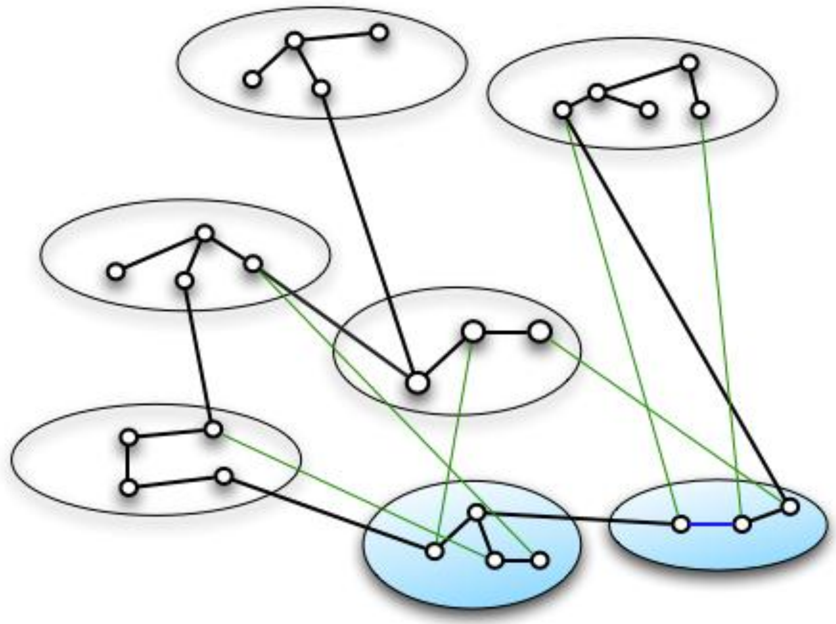
Deleting an edge inside one component

- We can see the two sets V' and V'' split from V_i may have <z vertices.
- Rearrange: Combine each of them with one adjacent vertex set and perform the *topological partition*.
  - The bound for the combined set: ≤4z-3



Deleting an edge inside one component

- We can see the two sets V' and V'' split from V_i may have <z vertices.
- Rearrange: Combine each of them with one adjacent vertex set and perform the *topological partition*.
  - The bound for the combined set: ≤4z-3
  - Also rearrange all the edge sets associated with these vertex sets
  - Since the graph has degree bound 3, the number of such edges is still O(z)
  - Time needed: O(z)



Running time

- $O(z+m^2/z^2)$ to find a replacement edge
- $O(z)$ to rearrange so that each set still has [z,3z-2]vertices

- When balancing these two, we get $z=m^{2/3}$, and the running time is $O(m^{2/3})$.

Improving it to $O(m^{1/2} \log n)$

- In the previous algorithm we need to check every pair of vertex sets between two split trees.
- We can store all the edges from a vertex set to the other sets in the tree in one structure
  - Use the ET-tree structure
  - In Frederickson's old paper (1983), they describe another structure called "topology trees" with similar functions.

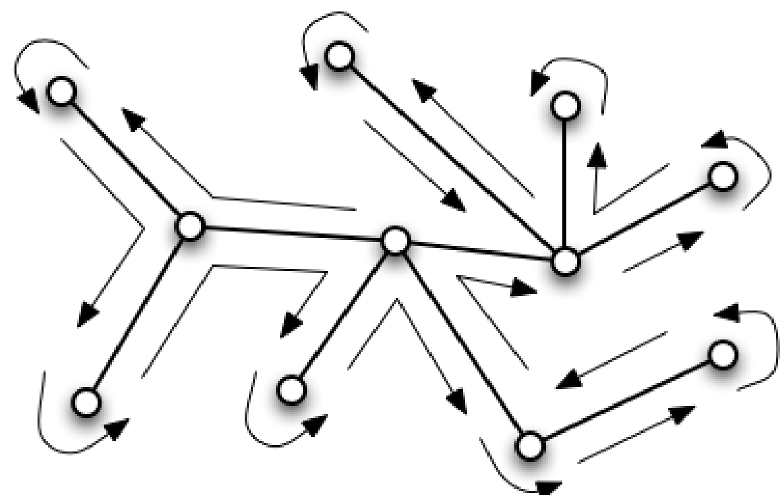
ET-tree

- We need to keep dynamic forest
  - Merge two tree by an edge
  - Split a tree into two subtrees
  - Find the tree containing a given vertex
  - Return the size of a tree
  - Min-key: returns the minimal key in a tree

- These operations can all be done in O(log n) time.

ET-trees

- Euler Tour of T:

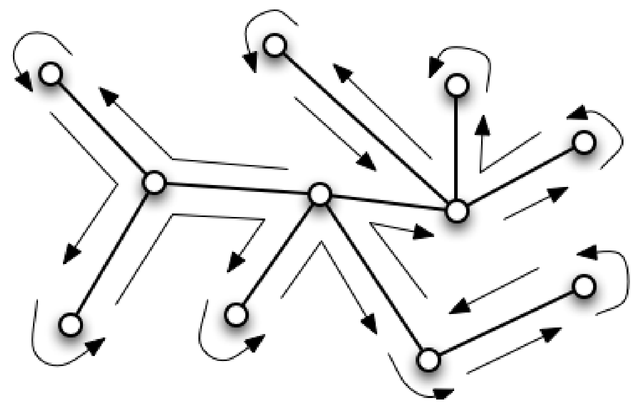


- Every vertex can appear many times in the Euler Tour, but we only keep any one of them for each vertex to form a ET-list:

  $v_1, \ v_2, \ \ldots v_n$

Euler Tour

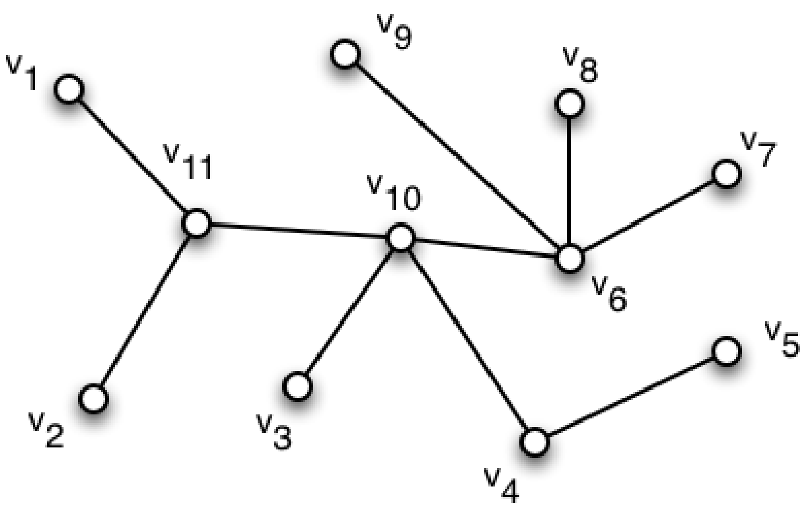
- Euler Tour of T:

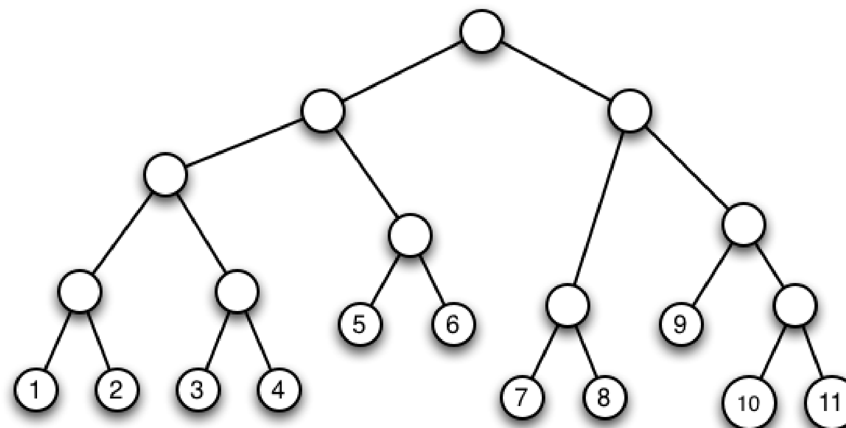


- So we only need O(1) link & cut operations to maintain the ET-lists per tree merging or splitting.
- We need balanced binary trees to keep the ET-lists.

ET-tree structure

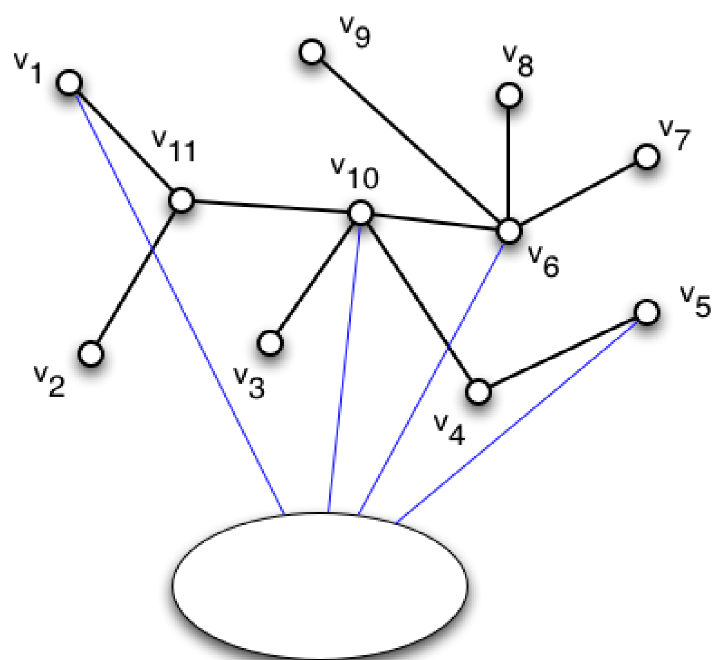
- Euler Tour of a tree
- Binary tree for the ET-list



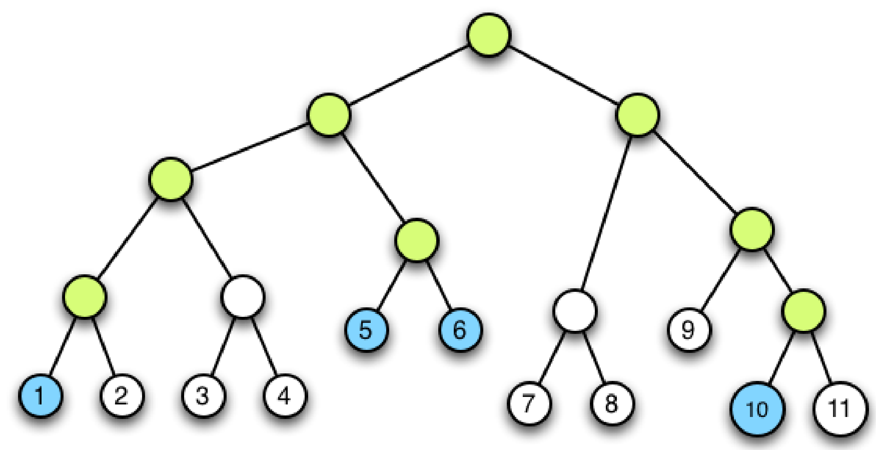


ET-tree structure

- Euler Tour of a tree
  - If some edges are connected to a component

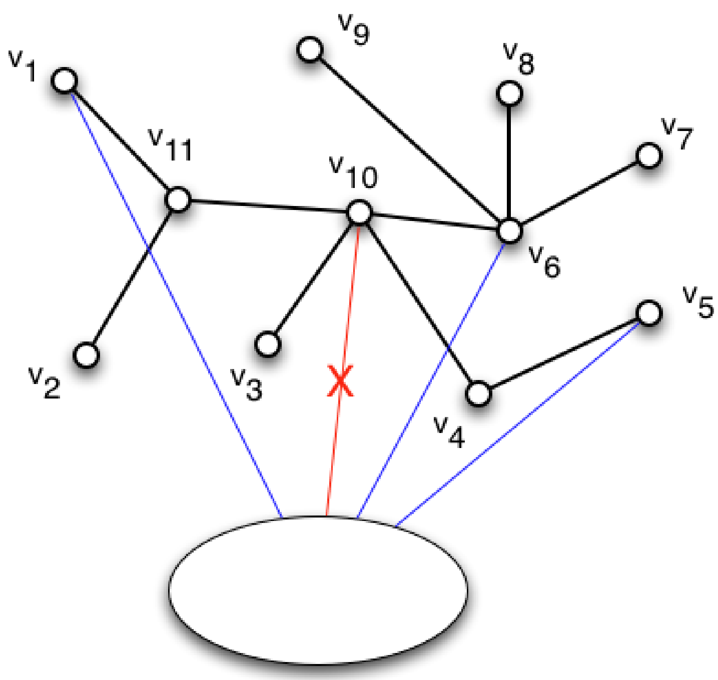


- Binary tree for the ET-list
  - Then we can easily check whether there is an edge from this tree to that component



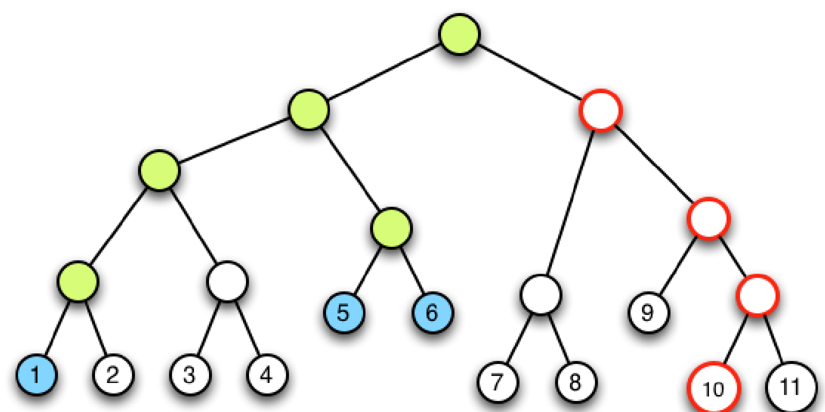
Updating ET-tree structure

- Euler Tour of a tree
  - When we update an edge



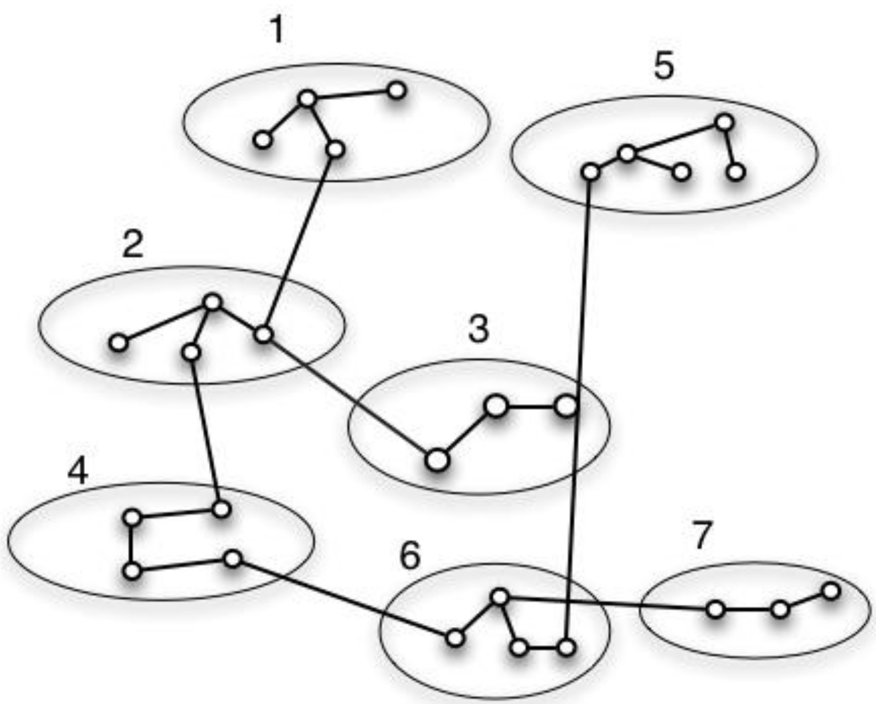
- Binary tree for the ET-list
  - We just need to update

O(log n) nodes in the binary tree

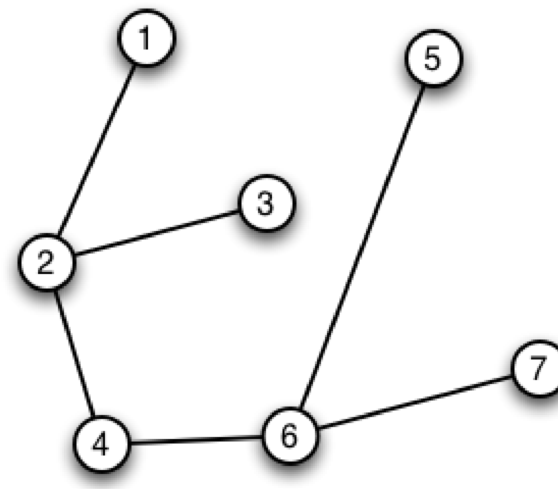


Represent tree of components

- Components on vertex sets

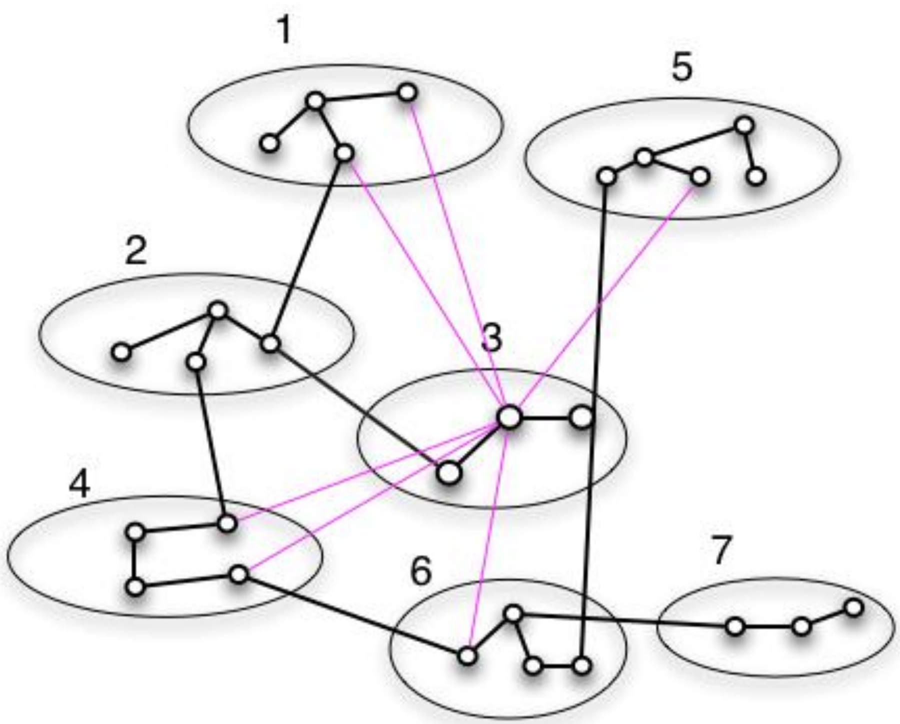


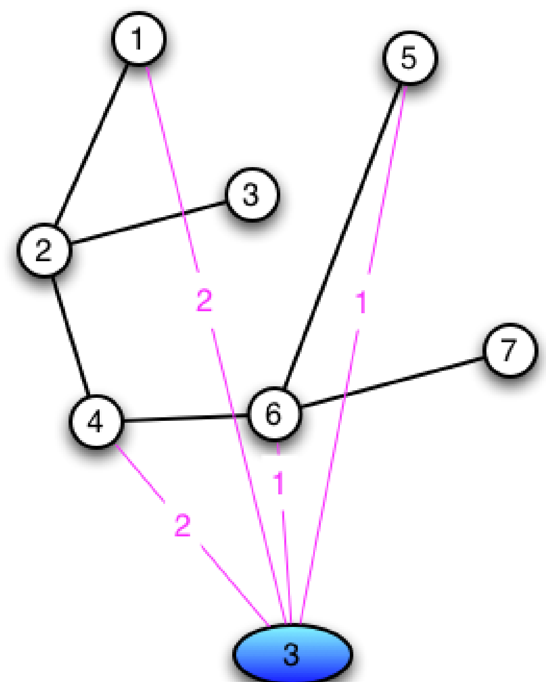
- Tree of components



Store E_{ij} for all V_j in the tree

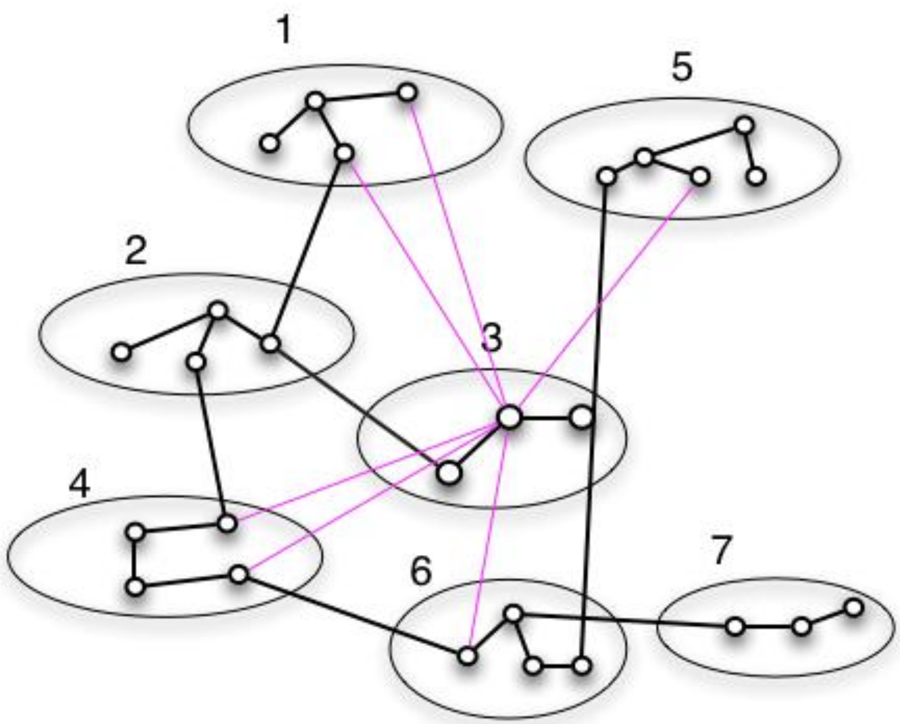
- Components on vertex sets
- Tree of components



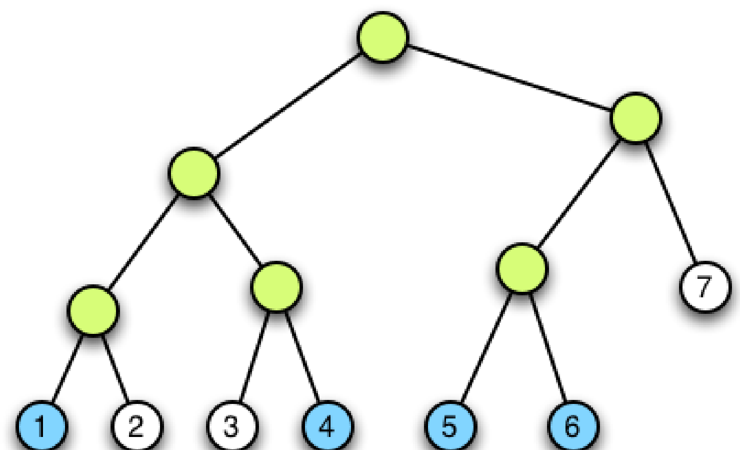


Store E_{ij} for all V_j in the tree

- Components on vertex sets



- Binary tree used to represent edges connecting V_3 to other sets



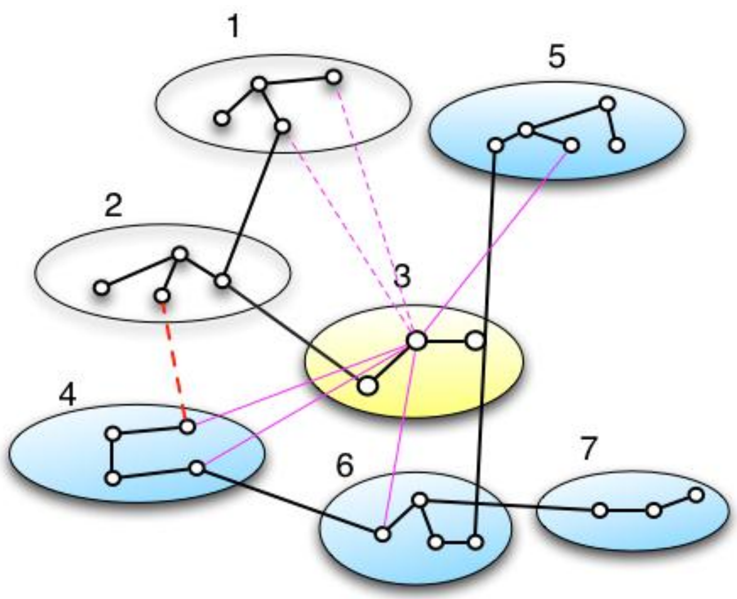
- We call this structure $H(V_i, T)$

Euler Tour

- We only need O(1) link & cut operations to maintain the ET-lists per tree merging or splitting.
- It takes O(log n) time to rebalancing the binary tree after a update,
- So when we merging or splitting trees, the time needed to maintain $H(V_i,T)$ for a set V_i is O(log n).
  - Total time: O(k•log n) for all $V_1,\ldots,V_k$

Update

- When deleting a tree edge e=(x,y)
  - If x,y are not in the same set, then first we split the tree into two parts
    - This will need O(1) links and cuts to ET-lists,
    - So O(m/z•log n) operations to update $H(V_i,T_1)$ and $H(V_i,T_2)$ for all sets V_i
  - Then we need to check all structure $H(V_i,T_1)$ for V_i in T_2 or $H(V_i,T_2)$ for V_i in T_1
  - The number of such H-structures is O(m/z)

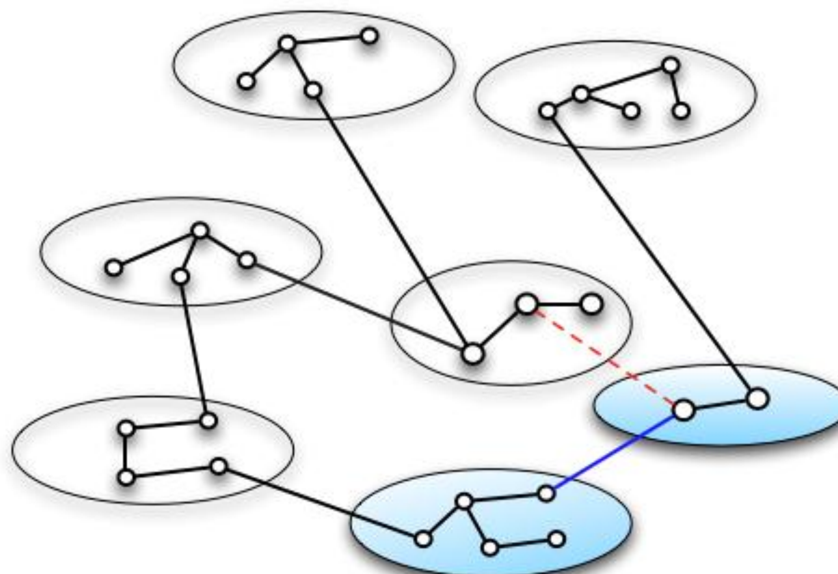


Deleting an edge inside one component

- Then we need to check all edges in E_{ii} and E_{ij} for every V_j, and also other H-structures connecting two subtrees
- We can see the two sets V' and V'' split from V_i may have <z vertices.
- Rearrange: Combine each of them with one adjacent vertex set and perform the *topological partition*.

- When rearranging a non-tree edge from V_i to V_j, we need to update $O(\log n)$ nodes in $H(V_i,T)$, $H(V_j,T)$

- Total time: $O(m/z \cdot \log n + z \cdot \log n)$



Running time

- $O(z+m/z \cdot \log n)$ to find a replacement edge
- $O(z \cdot \log n)$ to rearrange and update H-structures

- When balancing these two, we get $z=m^{1/2}$, and the running time is $\tilde{O}(m^{1/2})$.

Dynamic minimum spanning tree

- This $\tilde{O}(m^{1/2})$ structure can be easily extended to dynamic minimum spanning tree structure:
  - Sort all the edges in every E_{ij}
  - Maintain the min-key in every $H(V_i,T)$
  - …

Improve Frederickson's algorithm to $\tilde{O}(n^{1/2})$

- Sparsification:
  - *"Sparsification – A technique for speeding up dynamic graph algorithms"*
  - By Eppstein, Galil, Italiano, Nissenzweig, **Journal of ACM 1997**

Improve Frederickson's algorithm to $\tilde{O}(n^{1/2})$

- Sparsification:
  - *"Sparsification – A technique for speeding up dynamic graph algorithms"*
  - By Eppstein, Galil, Italiano, Nissenzweig, **Journal of ACM 1997**

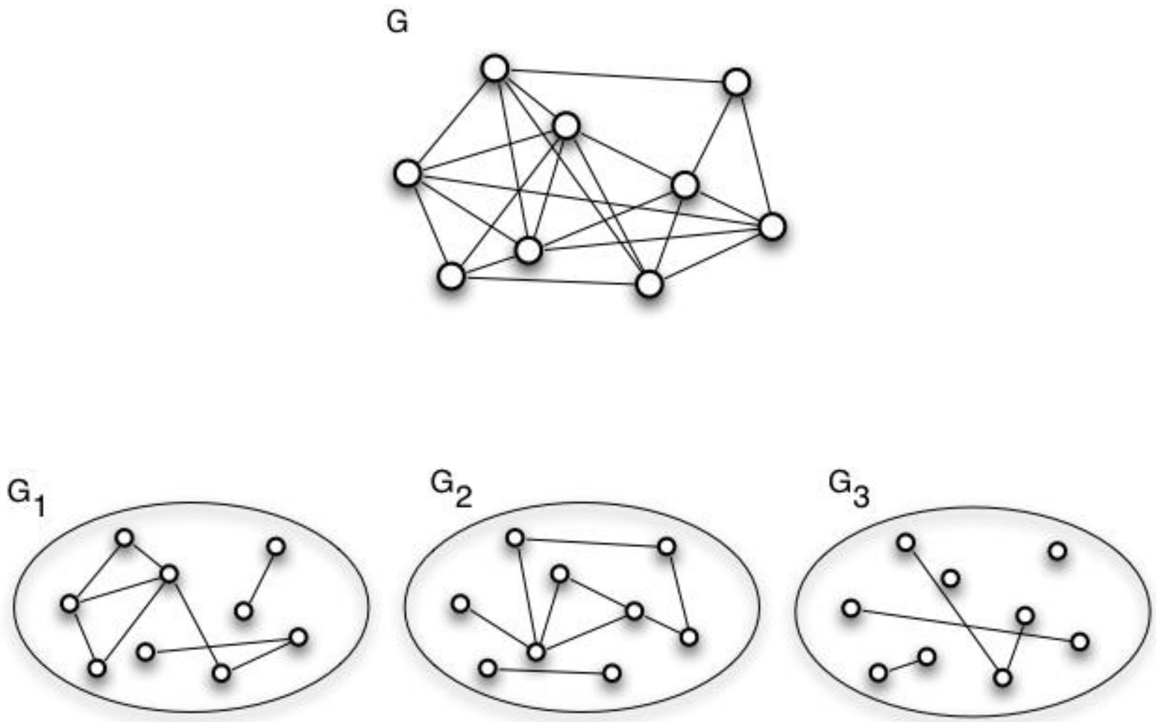
  - They divide the edge set E into subsets $\{E_i\}$
  - Maintain spanning trees T_i on $\{E_i\}$
  - Maintain spanning trees on the union of some T_i

Partition of E

- Partition E into sets of n edges:
  - $\{E_1, E_2, \ldots E_k\}$, where $k \le \lceil m/n \rceil$, and $|E_i| = n$ for $i = 1, \ldots, k-1$
- When inserting an edge e,
  - Insert e to E_k if $|E_k| < n$
  - Otherwise create a new set E_{k+1}
- When deleting an edge from E_i
  - Move one edge of E_k to Ei
  - If E_k becomes empty, remove E_k

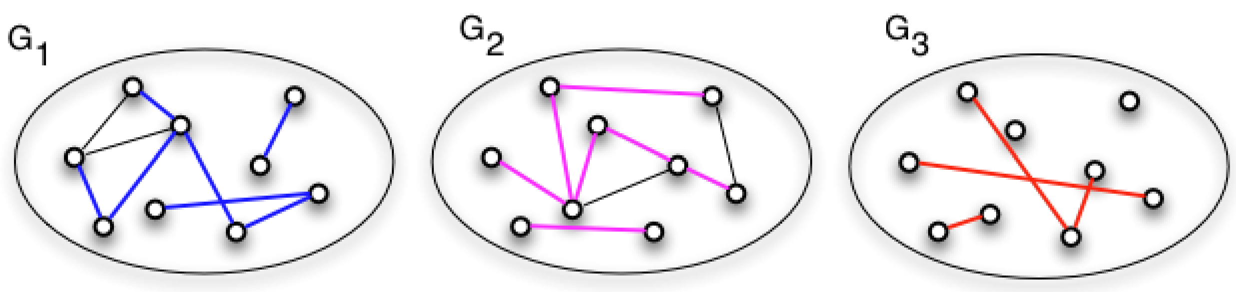
Partition of E

- Assume a dynamic connectivity structure of update time f(n,m)
- Maintain spanning forests F_i in $G_i=(V, E_i)$



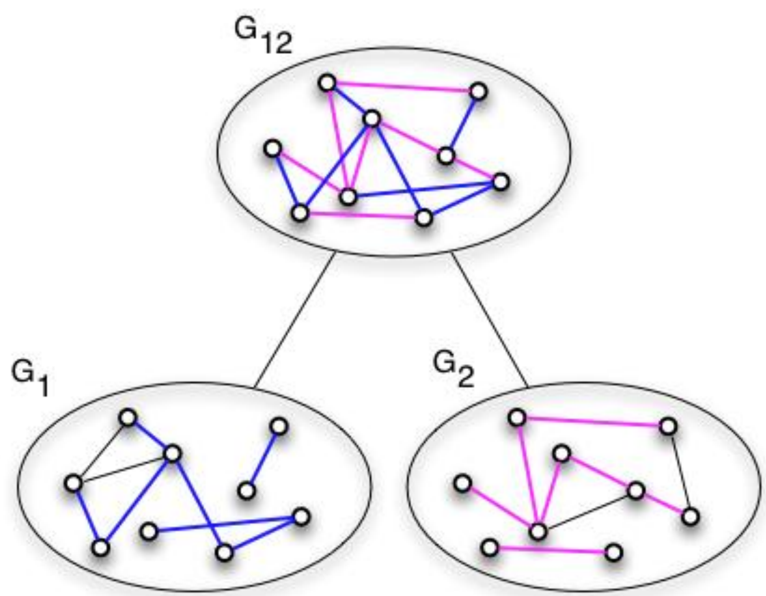
Partition of E

- Assume a dynamic connectivity structure of update time f(n,m)
- Maintain spanning forests F_i in $G_i=(V, E_i)$



Partition of E

- Assume a dynamic connectivity structure of update time f(n,m)
- Maintain spanning forests F_i in $G_i=(V, E_i)$
- In the graph $G_{12}=(V,F_1 \cup F_2)$, maintain a spanning forest
  - G_{12} also has O(n) edges
  - It maintains the connectivity of $(V, E_1 \cup E_2)$

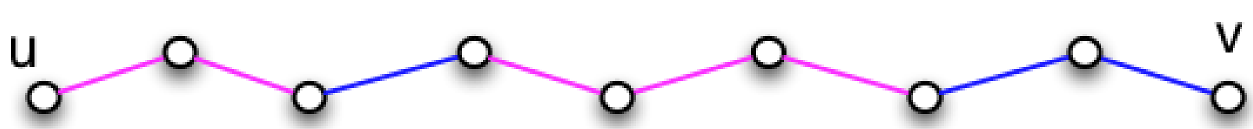


Proof of correctness

- F is a *strong certificate* of connectivity for G:
  - If F_1, F_2 are spanning forests on G_1, G_2, then u,v are connected in $F_1 \cup F_2$ if they are connected in $G_1 \cup G_2$.

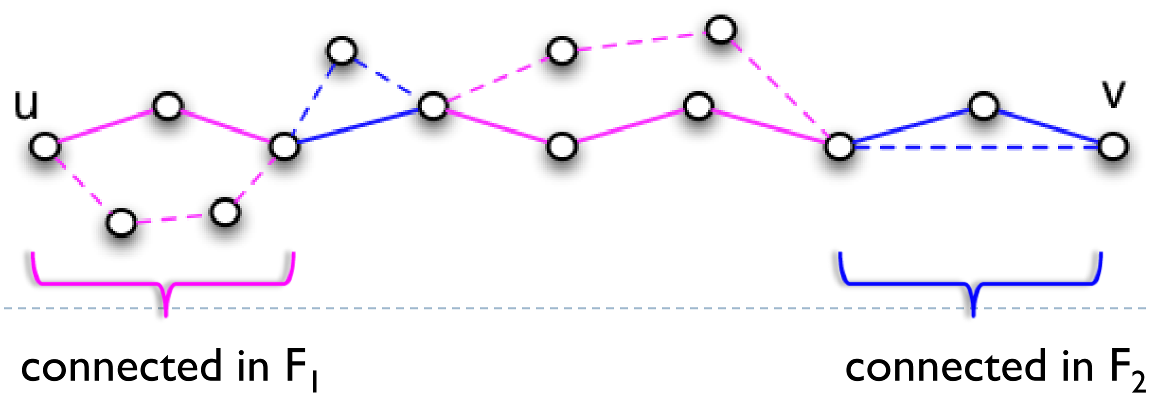
Proof of correctness

- F is a *strong certificate* of connectivity for G:
  - If F_1, F_2 are spanning forests on G_1, G_2, then u,v are connected in $F_1 \cup F_2$ if they are connected in $G_1 \cup G_2$.
  - Proof: Consider the path from u to v in $G_1 \cup G_2$:
    - edges of G_1
    - edges of G_2



Proof of correctness

- F is a *strong certificate* of connectivity for G:
  - If F_1, F_2 are spanning forests on G_1, G_2, then u,v are connected in $F_1 \cup F_2$ if they are connected in $G_1 \cup G_2$.
  - Proof: Consider the path from u to v in $G_1 \cup G_2$:
    - edges of G1, edges of G_2
    - (dash lines: edges in F_1 and F_2)

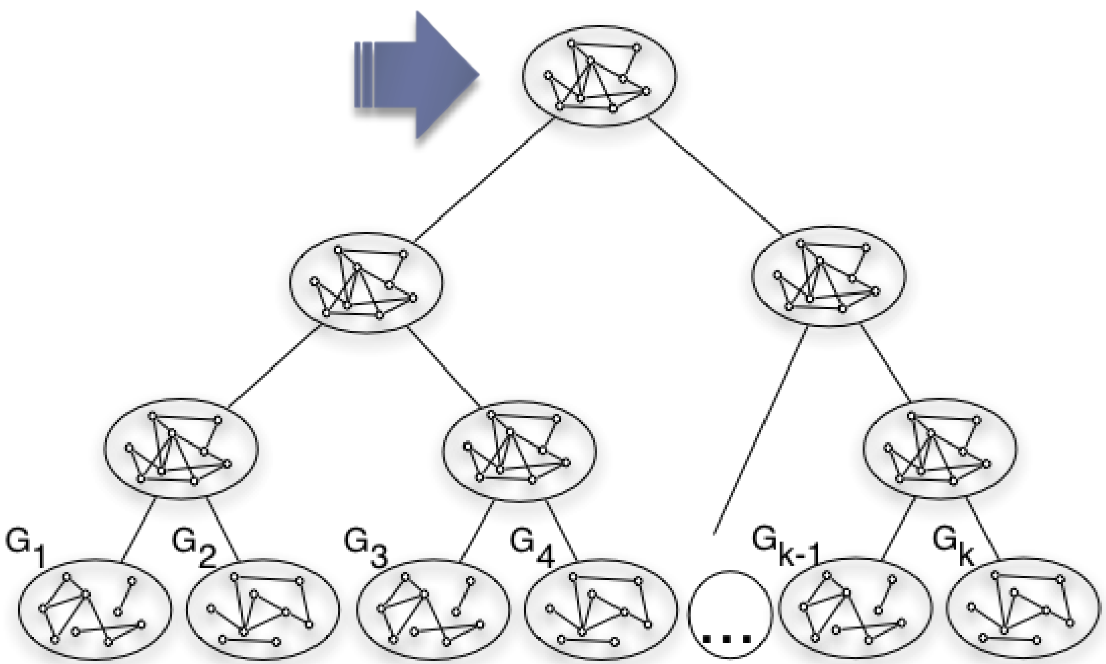


Proof of correctness

- F is a *strong certificate* of connectivity for G:
  - If F_1, F_2 are spanning forests on G_1, G_2, then u,v are connected in $F_1 \cup F_2$ if they are connected in $G_1 \cup G_2$.
- F is stable:
  - We only need to update O(1) edges in F when updating an edge of G

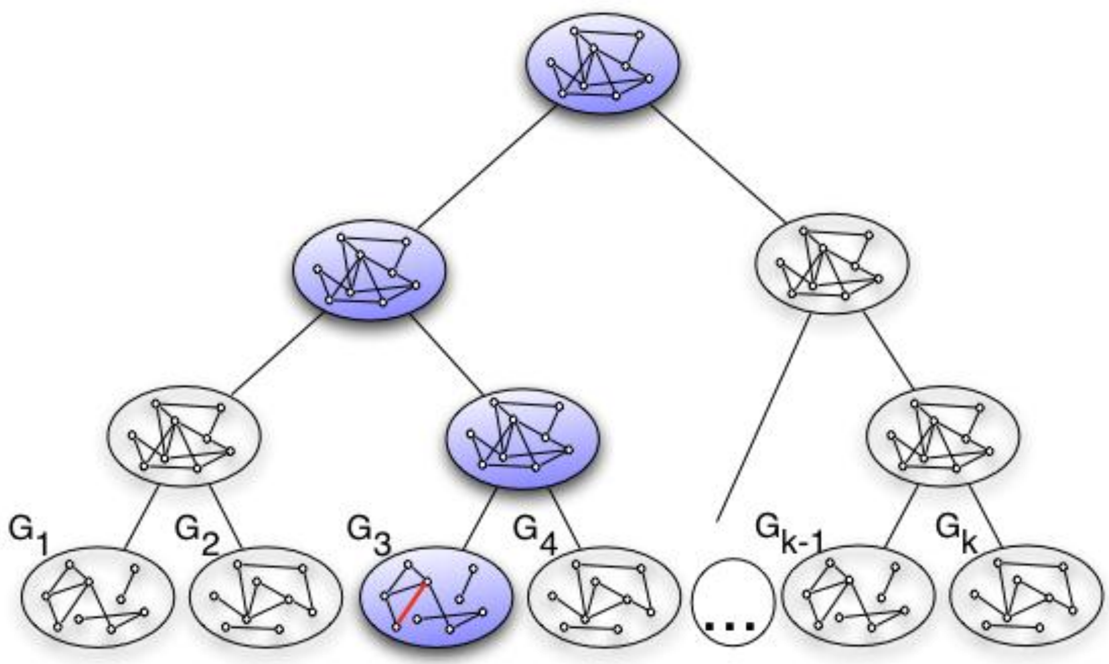
Maintain the structure on a binary tree

- Each node represents a subgraph of G
- The edges of each node is the union of the spanning forests of its children.
  - The number of levels: O(log k)=O(log(m/n))
  - The number of edges in each node: O(n)
  - Finally we can check the connectivity in the root node



Maintain the structure on a binary tree

- When updating an edge in G_i, we only need to update at most 2 edges in G_i's ancestors.
  - Since the number of levels is O(log k)=O(log(m/n)) and the number of edges in each node is O(n)
  - Update time is f(n,O(n))•O(log(m/n)) using a structure with f(n,m) update time.



Maintain the structure on a binary tree

- When updating an edge in G_i, we only need to update at most 2 edges in G_i's ancestors.
  - Since the number of levels is $O(\log k)=O(\log(m/n))$ and the number of edges in each node is $O(n)$
  - Update time is $f(n,O(n))\bullet O(\log(m/n))$ using a structure with $f(n,m)$ update time
  - By the Frederickson's structure of update time $\tilde{O}(m^{1/2})$,
  - We get a dynamic connectivity structure of update time $\tilde{O}(n^{1/2})$.

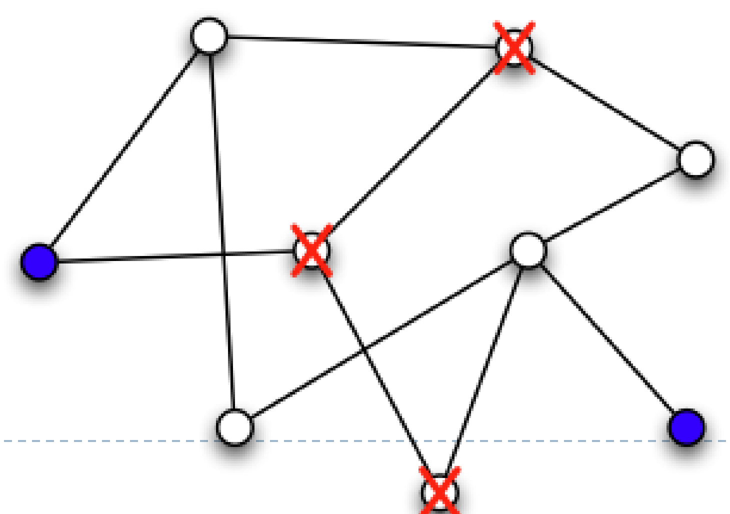
Summary of Dynamic Connectivity Results

- Edge update—amortized time
  - Holm, Lichtenberg, and Thorup: $O(\log^2 n)$

- Edge update—worst-case
  - Frederickson, Eppstein et al: $\tilde{O}(n^{1/2})$

Dynamic Subgraph Model

- There is a fixed underlying graph G, every vertex in G is in one of the two states "on" and "off".
- Construct a dynamic data structure:
  - Update: Switch a vertex "on" or "off".
  - Query: For a pair (u,v), answer connectivity/shortest path between u and v in the subgraph of G induced by the "on" vertices.



Dynamic Connectivity

	Edge Updates	Vertex Updates (Subgraph)
Amortized	$O(\log^2 n)$ [Holm, Lichtenberg & Thorup '1998]	$\tilde{O}(m^{2/3})$, with query time $\tilde{O}(m^{1/3})$ [Chan, Pâtraşcu & Roditty '2008]
Worst-Case	$O(n^{1/2})$ [$O(m^{1/2})$ by Frederickson '1985] [Improved by Eppstein, Galil, Italiano, Nissenzweig '1992]	$\tilde{O}(m^{4/5})$, with query time $\tilde{O}(m^{1/5})$ [Duan 2010]

Dynamic Subgraph Connectivity (Optional)

- Dynamic subgraph connectivity with $\tilde{O}(m^{2/3})$ amortized pdate time and $\tilde{O}(m^{1/3})$ query time.
  - *"Dynamic Connectivity: Connecting to Networks and Geometry"*
  - By Chan, Pâtraşcu & Roditty '2008

- Do not maintain a spanning forest for the whole graph.

Trivial Algorithm

- Since a vertex can associate with at most n-1 edges, so by the edge connectivity structure, we can get a vertex connectivity structure of amortized update time $\tilde{O}(n)$ and query time $O(1)$.

Preprocessing

- Maintain two sets of active vertices: P,Q
  - Initially all active vertices are in P
  - We only delete vertices from P
  - When we turn a vertex on, we add it into Q
- Thus,
  - P just supports deletions (decremental structure)
  - Q supports both insertions and deletions

Preprocessing

- Maintain two sets of active vertices: P,Q
  - Initially all active vertices are in P
  - We only delete vertices from P
  - When we turn a vertex on, we add it into Q
- Thus,
  - P just supports deletions (decremental structure)
  - Q supports both insertions and deletions

- Reinitialize after $q=m^{2/3}$ updates.
- So the size of Q is at most $m^{2/3}$.

High and low components

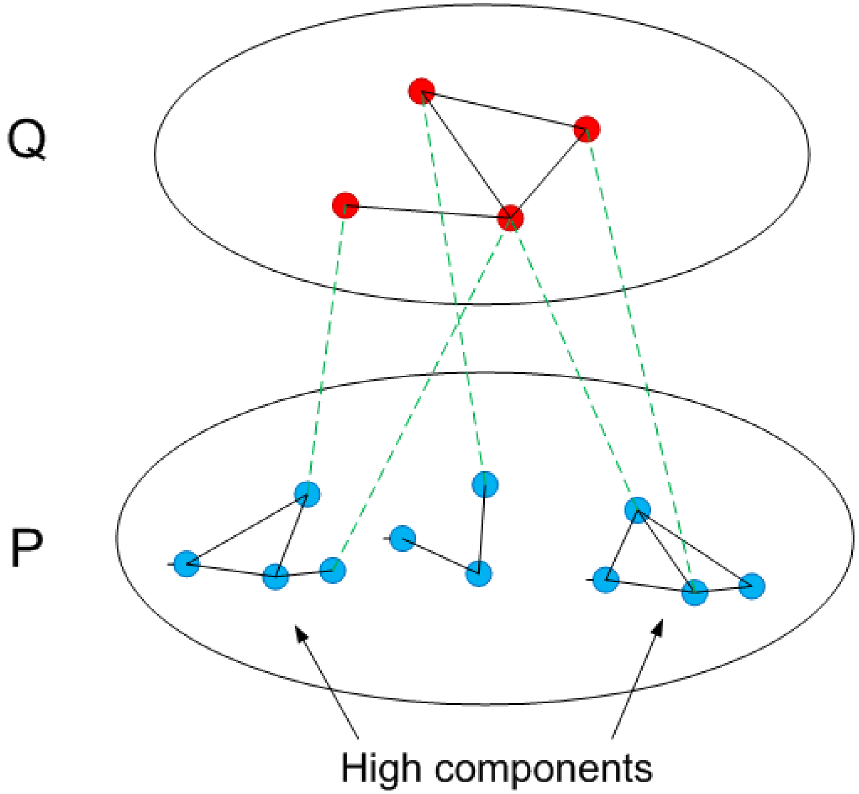
- Maintain the decremental connectivity structure with edge updates in P.
- For a connected component in P, if the sum of its degrees exceeds $m^{1/3}$, it is called a high component, otherwise it is a low component.
- The number of high components is bounded by $O(m^{2/3})$.

Maintain a new graph

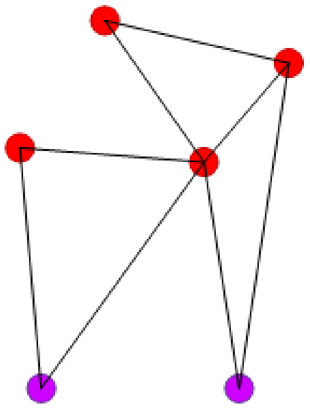
- Maintain a graph G* of vertices $O(m^{2/3})$:
  - Vertices of Q
  - Vertices set H where each vertex represents a high component in P
  - And the original edges in G connecting those vertices and components.

Example

The sets P and Q:



The graph G*



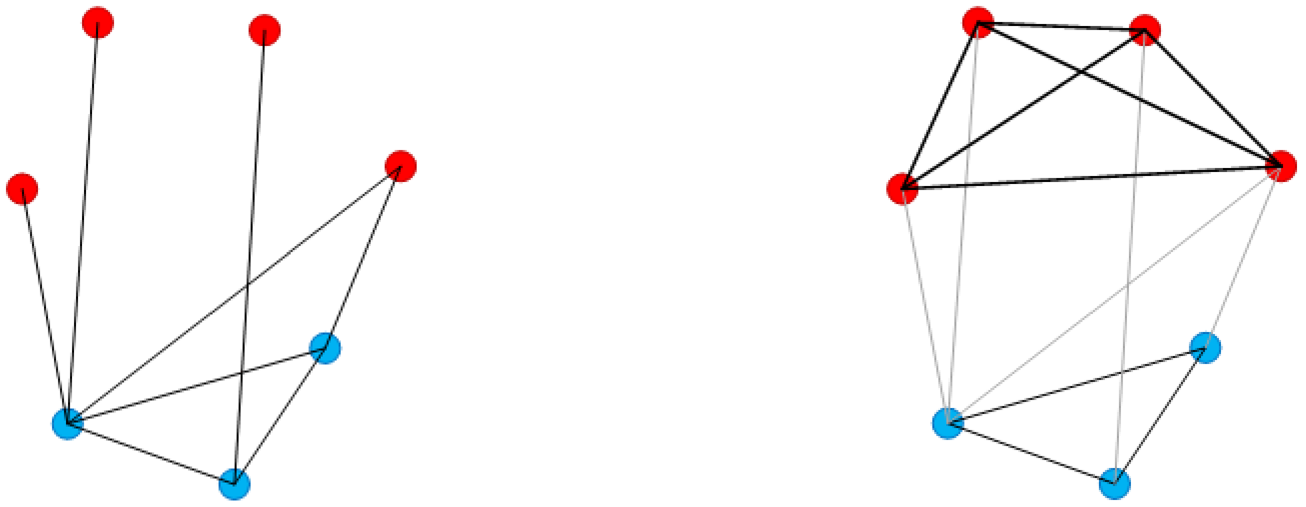
Note that there is no edges connecting these components

- But what if two vertices of Q can be connected by a low component?

The Edge Set Γ

- We construct a edge set Γ on the vertices in G
- If both u and v are adjacent to the same low component in P, then there is an edge (u,v) in Γ.
- There can be multiple edges between u and v.

Example



The size of Γ

- For every edge connecting a low component and a vertex, since the number of edges associate with that low component is at most $m^{1/3}$, so the number of edges in Γ generated by this edge is at most $m^{1/3}$.
- So the total number of edges in Γ is at most $m^{4/3}$.

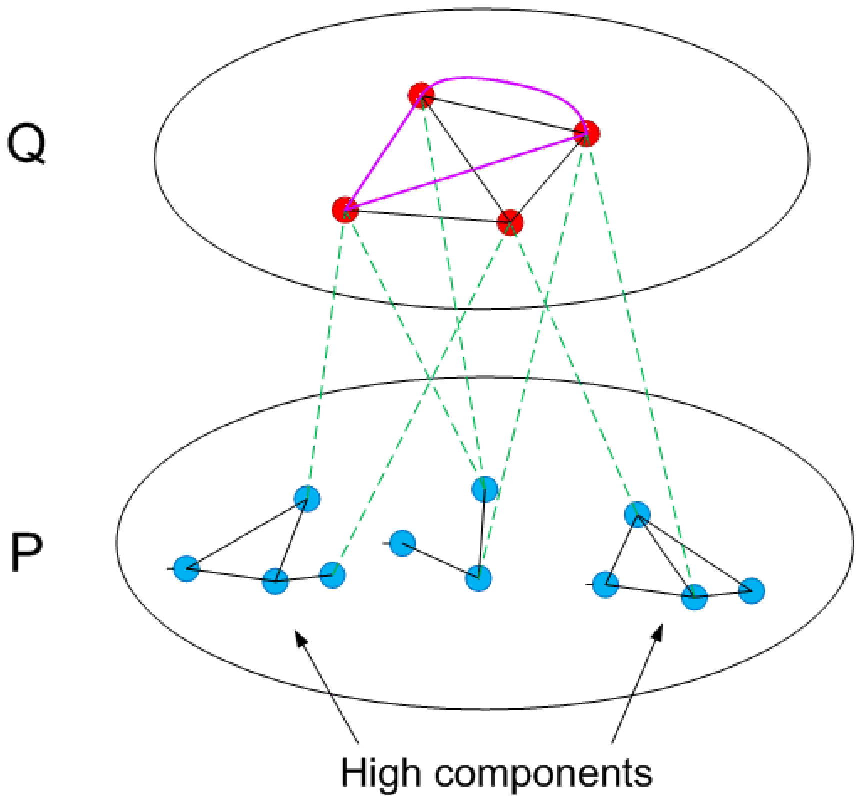
Maintain a new graph

- Maintain a graph G* of vertices $O(m^{2/3})$:
  - Vertices of Q
  - Vertices set H where each vertex represents a high component in P
  - And the original edges in G connecting those vertices and components.
  - Include the edges of Γ into G*

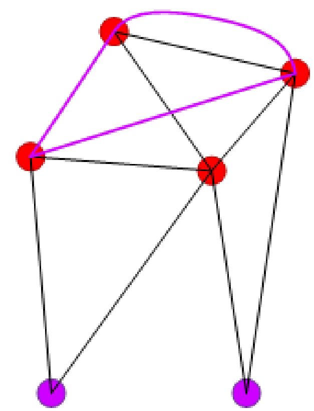
- It is easy to check that for every pair of active vertices in Q, they are connected in G iff they are connected in G*

Example

The sets P and Q:



The graph G*



Maintain G* in an edge connectivity oracle.

Query

- It takes O(log n) time to find a vertex in Q which component it is in.
- For vertices in high components:
  - Find the vertex in G* which represents that component
- For vertices in low components:
  - Search for an active vertex of Q adjacent to the component.
  - If it does not exist, the component is isolated.
  - Query time: $\tilde{O}(m^{1/3})$, since the edges associated with a low component is $O(m^{1/3})$.

Preprocessing Time

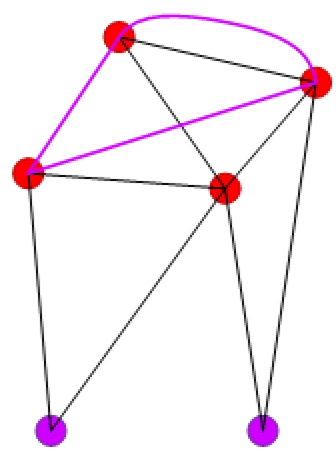
- Initializing G^* and Γ takes $\tilde{O}(m^{4/3})$ time.
- Since we will reinitialize after $m^{2/3}$ updates, so the amortized cost for every update is $\tilde{O}(m^{2/3})$.

Analysis of Updates

- Initially all active vertices are in P, and Q is empty.
- P -- deletion only

Analysis of Updates

- When update (insert/delete) a vertex v from Q
  - Update G*: check for every vertex in G* whether it is adjacent to v, update those edges
  - Time: $\tilde{O}(m^{2/3})$.

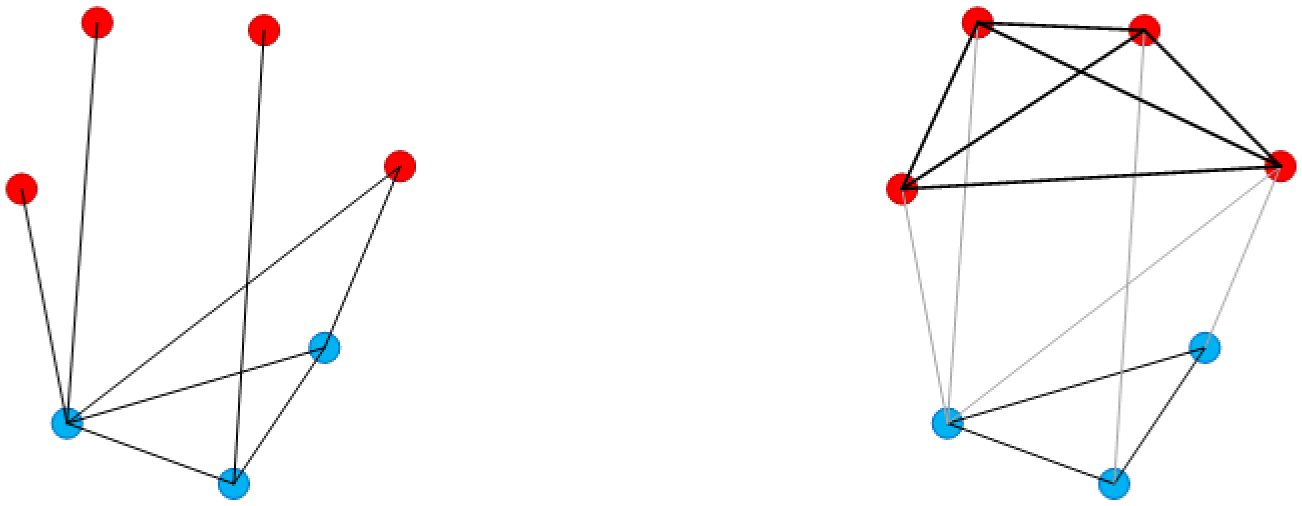


Analysis of Updates

- When deleting a vertex of a low component in P:
  - Recompute the edges in Γ generated by it.
  - Update those edges in G*

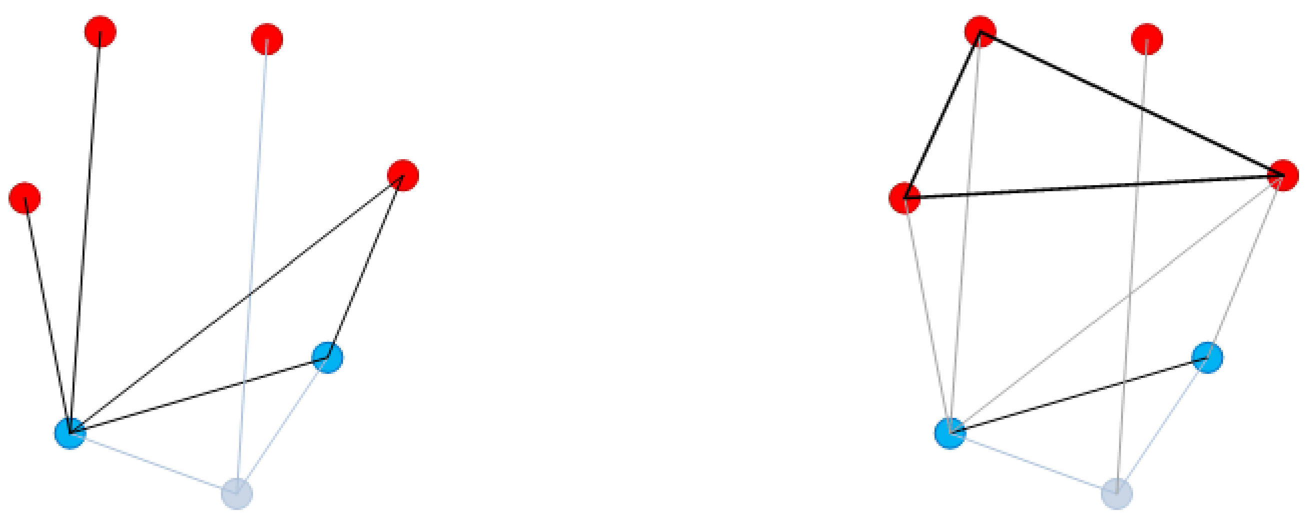
Analysis of Updates

- When deleting a vertex of a low component in P:
  - Recompute the edges in Γ generated by it.
  - Update those edges in G*



Analysis of Updates

- When deleting a vertex of a low component in P:
  - Recompute the edges in Γ generated by it.
  - Update those edges in G*



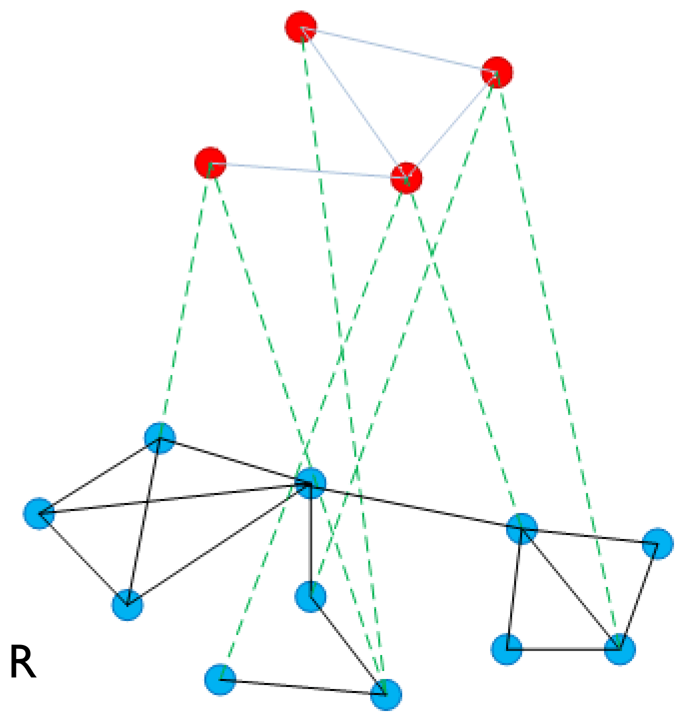
Analysis of Updates

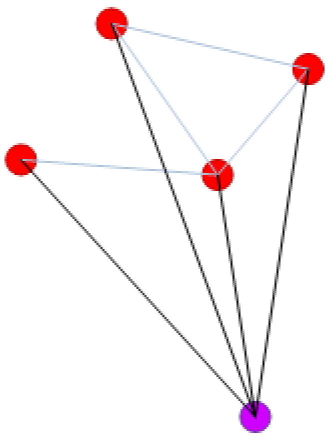
- When deleting a vertex of a low component in P:
  - Since the number of edges associate with that low component is at most $m^{1/3}$, we may need to update $O(m^{2/3})$ edges in G*, thus will take $\tilde{O}(m^{2/3})$ time.

Analysis of Updates

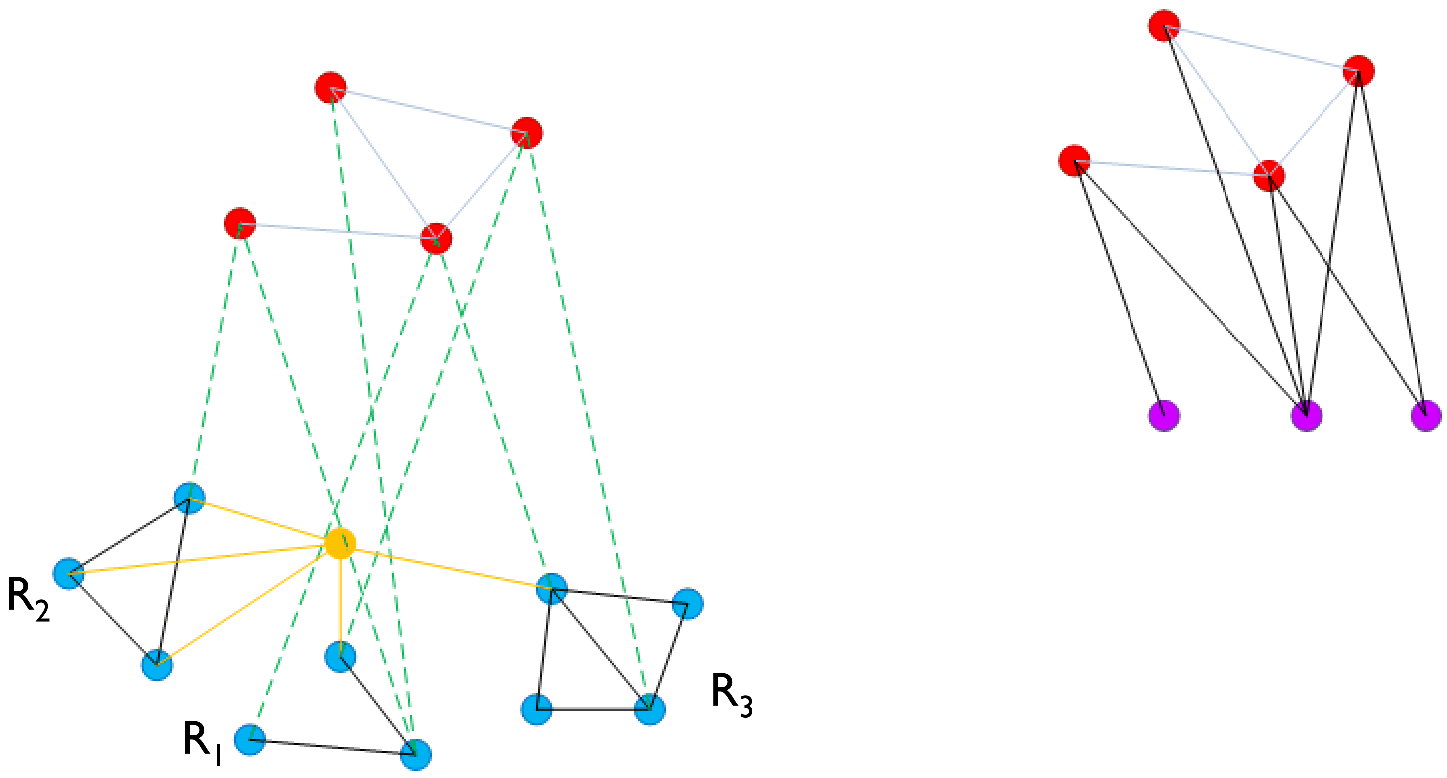
- When deleting a vertex of a high component in P:
  - The new components it generates may be high or low.
  - Rank the new components by the sum of degrees: $R_1, R_2, \ldots, R_k$ (from high to low).
  - Consider the new high components

Example



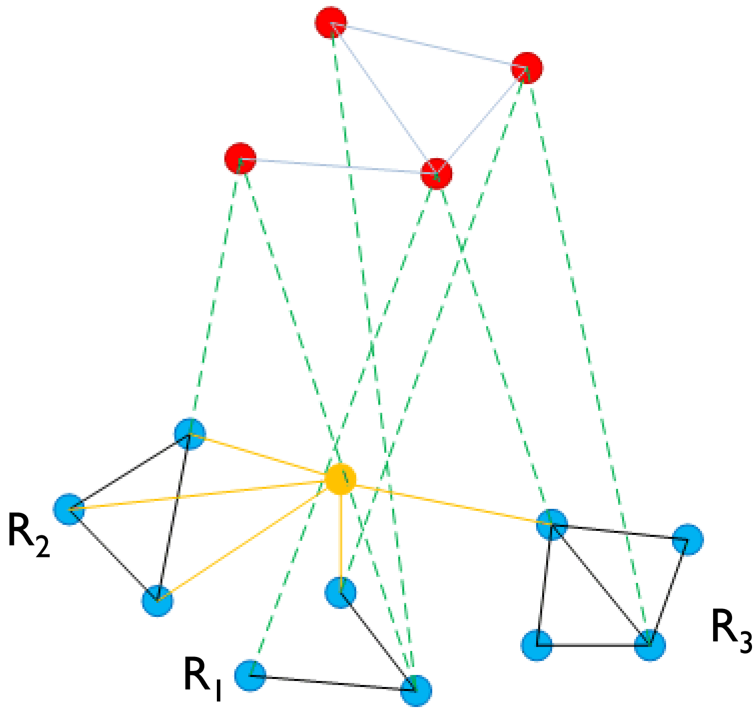


Example

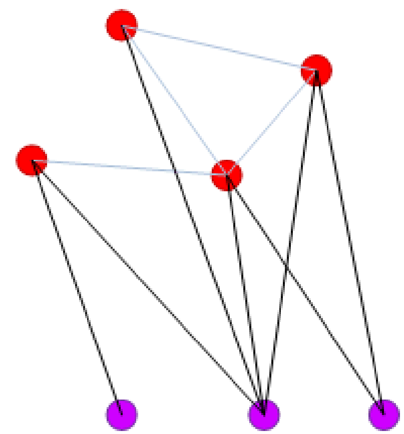


Cost of deleting edges in P:
In total of one phase is $\tilde{O}(m)$, so it is $\tilde{O}(m^{1/3})$ per updates.

Example



Cost of deleting edges in P:
In total of one phase is Õ(m), so it is Õ($m^{1/3}$) per updates.



Time needed to update G*:
O(deg(R_2)+deg(R_3)+…+deg(R_k))
Since deg(R_2),deg(R_3),…,deg(R_k) are at most half of deg(r), every edge can be moved at most *log*n times, so the total time per phase is still Õ(m).

Analysis of Updates

- For the new low components:
  - Compute the edges of Γ generated by them.
  - Since an edge can be in a new low component from a high component only once, so the total cost of time is also $\tilde{O}(m^{4/3})$, absorbed by the preprocess cost.

Conclusion

- Preprocessing Time: $\tilde{O}(m^{4/3})$.
- Amortized Update Time: $\tilde{O}(m^{2/3})$.
- Query Time: $\tilde{O}(m^{1/3})$
- Space: $\tilde{O}(m^{4/3})$ (The space needed to store Γ).

Conclusion

- Preprocessing Time: $\tilde{O}(m^{4/3})$.
- Amortized Update Time: $\tilde{O}(m^{2/3})$.
- Query Time: $\tilde{O}(m^{1/3})$
- Space: $\tilde{O}(m^{4/3})$ (We have improved it to $O(m)$).

Thank you!