Dynamic Connectivity

Ran Duan
In this talk

- Concept of dynamic algorithms
- Dynamic connectivity of $O(\log^2 n)$ amortized update time
- Decremental minimum spanning tree
Dynamic Problems

- Find algorithms and data structures to answer a certain query about a set of input objects where each time the input data is modified.
Dynamic Graph

- Fully dynamic model: we can insert and delete edges to the graph G
- Decremental model: only deletions
- Incremental model: only insertions
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
Amortized analysis

- For a sequence of updates, count the average time needed per each update.
  - Some updates may require much longer time
  - Only happen infrequently
Connectivity Problem

- In an undirected graph $G$, judge whether any two vertices are connected by a path.
Dynamic Connectivity

- We can insert or delete edges in this graph, and still find the connectivity of any pair of vertices.
Dynamic Connectivity

- We can insert or delete edges in this graph, and still find the connectivity of any pair of vertices.
Connectivity and spanning forest

- Spanning forest F: there is a spanning tree in each connected component
- Connectivity: check whether u, v are in the same spanning tree of F.
Dynamic Connectivity

- Maintain the spanning forest dynamically

  - Inserting (u,v):
    - When u,v are in the same tree, F do not change
    - When u,v are not in the same tree, connect these trees to a bigger tree
Dynamic Connectivity

- Maintain the spanning forest dynamically
- Deleting a tree edge \((u,v)\):
  - The tree will be split into two parts
  - We need to find other edges reconnecting these two parts
Dynamic Connectivity

- Maintain the spanning forest dynamically
- Deleting a tree edge \((u,v)\):
  - The tree will be split into two parts
  - We need to find other edges reconnecting these two parts
Dynamic Connectivity

- Maintain the spanning forest dynamically
- Deleting a tree edge \((u,v)\):
  - The tree will be split into two parts
  - We need to find other edges reconnecting these two parts
Dynamic Connectivity

- Maintain the spanning forest dynamically
- Deleting a tree edge \((u,v)\):
  - The tree will be split into two parts
  - We need to find other edges reconnecting these two parts
Holm, Lichtenberg & Thorup’s structure

- $O(\log^2 n)$ amortized update time
- Best amortized update time so far.
- Appears in STOC’98
High-level description

- Each edge $e$ is assigned a level $l(e)$. ($0 \leq l(e) \leq l_{\text{max}}$)
- $E_i = \{\text{edges of level} \geq i\}$
- So $E = E_0 \supseteq E_1 \supseteq \ldots \supseteq E_{l_{\text{max}}}$
High-level description

- We keep the set of spanning forest $F_0 \supseteq F_1 \supseteq \ldots \supseteq F_{\text{max}}$ on $E_0, E_1, \ldots, E_{\text{max}}$
  - if $e=(u,v)$ is a non-tree edge in $E_i$, $u$ and $v$ are connected in $F_i$
  - if $e$ is a tree edge in $F_i$, it must be a tree edge in $F_j$ ($j<i$)
- Also, the number of vertices of a tree in $F_i$ is at most $n/2^i$

- These properties are maintained throughout the algorithm.
High-level description

- We keep the set of spanning forest $F = F_0 \supseteq F_1 \supseteq \ldots \supseteq F_{l_{\text{max}}}$ on $E_0, E_1, \ldots, E_{l_{\text{max}}}$
  - if $e = (u, v)$ is a non-tree edge in $E_i$, $u$ and $v$ are connected in $F_i$
  - if $e$ is a tree edge in $F_i$, it must be tree edge in $F_j$ ($j < i$)

- Also, the number of vertices of a tree in $F_i$ is at most $n/2^i$
  - The sizes of connected components decrease by a half when level increases
  - So $l_{\text{max}} = O(\log n)$

- These properties are maintained throughout the algorithm.
Example – tree edge

- **level ≥2**
- **level 1**
- **level 0**
Remind

- **Inserting** \((u,v)\):
  - When \(u,v\) are in the same tree, \(F\) do not change
  - When \(u,v\) are not in the same tree, connect these trees to a bigger tree

- **Deleting a tree edge** \((u,v)\):
  - The tree will be split into two parts
  - We need to find other edges reconnecting these two parts
Algorithm

- Initially the graph is empty
- Level of an edge only increases, never decreases
  - When we have checked the edge, its level increases
  - Only increases for $l_{\text{max}} = O(\log n)$ times
  - So the amortized time for an edge is very small.
Algorithm

- **Insert(e):**
  - $l(e) = 0$, if its two ends are not connected in $F_0$, $e$ is added to $F_0$

- **Delete(e):**
  - If $e$ is not a tree edge at level $l(e)$, simply delete $e$
  - If $e$ is a tree edge, delete it in $F_0$, $F_1$, ..., $F_{l(e)}$, and call Reconnect($e$, $l(e)$)
Algorithm

- **Insert(e):**
  - \( l(e) = 0, \) if its two ends are not connected in \( F_0, \) \( e \) is added to \( F_0 \)

- **Delete(e):**
  - If \( e \) is not a tree edge at level \( l(e) \), simply delete \( e \)
  - If \( e \) is a tree edge, delete it in \( F_0, F_1, \ldots, F_{l(e)}, \)
    and call \( \text{Reconnect}(e, l(e)) \)

**Spanning forests** \( F = F_0 \supseteq F_1 \supseteq \ldots \supseteq F_{l_{\max}} \) on \( E_0, E_1, \ldots, E_{l_{\max}} \)

So when \( e \) is not a tree edge at its level \( l(e) \), it can not be a tree edge at other levels.
Algorithm

- **Reconnect((u,v),i)** – reconnect trees containing u and v by edges of level i
  - T – original tree in $F_i$ containing (u,v),
  - $T(u), T(v)$ – trees in $F_i$ containing u,v after deletion of (u,v)
  - One of $T(u), T(v)$ has at most a half as many vertices as T, assume it is $T(u)$, move $T(u)$ to level i+1
  - Check level i edges $f$ incident to $T(u)$ one by one, either:
    - $f$ does not connect $T(u)$ and $T(v)$, then it must be included in $T(u)$, increase its level to i+1
    - $f$ connect $T(u)$ and $T(v)$, stop the search, and add $f$ to $F_0, F_1, \ldots, F_i$
  - If no such edges are found, call **Reconnect((u,v),i-1)**
  - If i=0, we conclude that there is no reconnecting edges.
Algorithm

- If \( f \) does not connect \( T(u) \) and \( T(v) \), then it must be included in \( T(u) \), increase its level to \( i+1 \) (since \( |T(u)| \leq \frac{1}{2}|T| \))
- If \( f \) connect \( T(u) \) and \( T(v) \), stop the search, and add \( f \) to \( F_0, F_1, \ldots, F_i \)
Algorithm

- If \( f \) does not connect \( T(u) \) and \( T(v) \), then it must be included in \( T(u) \), increase its level to \( i + 1 \).
- If \( f \) connect \( T(u) \) and \( T(v) \), stop the search, and add \( f \) to \( F_0, F_1, \ldots, F_i \).
Algorithm

- If $f$ does not connect $T(u)$ and $T(v)$, then it must be included in $T(u)$, increase its level to $i+1$.
- If $f$ connects $T(u)$ and $T(v)$, stop the search, and add $f$ to $F_0, F_1, \ldots, F_i$. 
Bound the reconnecting time

- In one update we may need to check all the edges associated with a subtree $T(u)$
- But after checking an edge, its level increases, so every edge can be checked $O(\log n)$ times
- If initially the graph is empty, the number of edges is at most the number of update, so we need to check $O(\log n)$ edges per update.
We keep the set of spanning forest $F = F_0 \supseteq F_1 \supseteq \ldots \supseteq F_{l_{\text{max}}}$ on $E_0, E_1, \ldots, E_{l_{\text{max}}}$

- if $e = (u, v)$ is a non-tree edge in $E_i$, $u$ and $v$ are connected in $F_i$
- if $e$ is a tree edge in $F_i$, it must be a tree edge in $F_j$ ($j < i$)

Also, the number of vertices of a tree in $F_i$ is at most $n/2^i$

These properties hold after the update algorithm
Example

- $F_0, F_1, F_2$: (non-tree edges are shown only in their levels)
Example

- Deleting a tree edge:
Example

- Call Reconnect(e,l(e))
Example

- Check for an edge whether it can reconnect them
Example

- Remove it to higher level
Example

- Call reconnect in lower level

\[ F_0 \]

\[ F_1 \]

\[ F_2 \]
Implementation

- 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 an ET-list:
  
  \[ v_1, v_2, \ldots, v_n \]
When we delete a tree edge, the ET-list will be divided into \( \leq 3 \) parts, and we need to merge two lists.
When we connect two trees by an edge, we need to split the ET-lists of the two trees from the vertices on that edge ...

$(v_1, v_2, v_3, v_4, v_5), (v_6, v_7)$

$(u_1), (u_2, u_3, u_4)$
When we connect two trees by an edge, we need to split the ET-lists of the two trees from the vertices on that edge, and merge them in the right order.

\[(v_1, v_2, v_3, v_4, v_5), (v_6, v_7); (u_1), (u_2, u_3, u_4)\]

\[(v_1, v_2, v_3, v_4, v_5, u_2, u_3, u_4, u_1, v_6, v_7)\]
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.

However, we need **balanced binary trees** to keep the ET-lists, so it takes $O(\log n)$ time to rebalancing after a update,
Self-balancing binary search tree

- Automatically keep its height $O(\log n)$
Self-balancing binary search tree

- Need $O(\log n)$ time to rebalancing
- $O(\log n)$ time to find the root from a vertex
- Every vertex can store the size or min-key of its subtree, so these information can be maintained in $O(\log n)$ time per update.
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.
If initially the graph is empty, the number of edges is at most the number of update, so we need to check $O(\log n)$ edges per update.

Since merging two trees takes $O(\log n)$ time, and an edge can merge trees in $O(\log n)$ levels, so the amortized update time is $O(\log^2 n)$
Back to dynamic connectivity

- If initially the graph is empty, the number of edges is at most the number of update, so we need to check $O(\log n)$ edges per update.
- Since merging two trees takes $O(\log n)$ time, and an edge can merge trees in $O(\log n)$ levels, so the amortized update time is $O(\log^2 n)$.
- Deletion can cost $O(\log^2 n)$ time.
  - Delete an edge in $l_{\text{max}}$ trees

- Query time: $O(\log n/\log\log n)$
- Space: $O(m+n\log n)$ (almost linear)
Dynamic Minimum Spanning Tree

- Much more complicated since we need to consider the order of edges
- Decremental minimum spanning tree
  - Only a modification from dynamic connectivity structure
  - Only support deletions
Algorithm

- Originally we have a MST $F_0$ at level 0
- **Delete(e):**
  - If $e$ is not a tree edge at level $l(e)$, simply delete $e$
  - If $e$ is a tree edge, delete it in $F_0, F_1, \ldots, F_{l(e)}$ and call $\text{Reconnect}(e, l(e))$

Spanning forests $F = F_0 \supseteq F_1 \supseteq \ldots \supseteq F_{l_{\text{max}}}$ on $E_0, E_1, \ldots, E_{l_{\text{max}}}$
So when $e$ is not a tree edge at level $l(e)$, it can not be a tree edge at other levels.
Algorithm

- **Reconnect**((u,v),i) – reconnect trees containing u and v by edges of level i
  - T – original tree containing (u,v),
  - T(u),T(v)– trees containing u,v after deletion of (u,v)
  - One of T(u),T(v) has at most a half as many vertices as T, assume it is T(u), move T(u) to level i+1
  - Check level i edges f incident to T(u) **in increasing order**, 
    - f does not connect T(u) and T(v), then it must be included in T(u), increase its level to i+1
    - f connect T(u) and T(v), stop the search, and add f to F₀, F₁,…,Fᵢ
  - If no such edges are found, call Reconnect((u,v),i-1)
  - If i=0, we conclude that there is no reconnecting edges.
Algorithm

- **Reconnect((u,v),i)** — reconnect trees containing u and v by edges of level i
  - T — original tree containing (u,v),
  - T(u), T(v) — trees containing u, v after deletion of (u,v)
  - One of T(u), T(v) has at most a half as many vertices as T, assume it is T(u), move T(u) to level i+1
  - Check level i edges f incident to T(u) in increasing order,
    - f does not connect T(u) and T(v), then it must be included in T(u), increase its level to i+1
    - f connect T(u) and T(v), stop the search, and add f to F₀, F₁, ..., Fᵢ
  - If no such edges are found, call Reconnect((u,v),i-1)
  - If i=0, we conclude that there is no reconnecting edges.

- Intuitively, we can see we find the minimum edge which reconnects the two subtrees.
Example
Example
Example

$F_1$

$F_0$
Invariants

1. We keep the set of spanning forest $F = F_0 \supseteq F_1 \supseteq \ldots \supseteq F_{\max}$ on $E_0, E_1, \ldots, E_{\max}$

2. The number of vertices of a tree in $F_i$ is at most $n/2^i$

3. Every cycle $C$ has a non-tree edge $e$ with:

   $$w(e) = \max_{f \in C} w(f)$$

   $$l(e) = \min_{f \in C} l(f)$$
Invariants

1. We keep the set of spanning forest \( F = F_0 \supseteq F_1 \supseteq \ldots \supseteq F_{l_{\text{max}}} \) on \( E_0, E_1, \ldots, E_{l_{\text{max}}} \)

2. The number of vertices of a tree in \( F_i \) is at most \( n/2^i \)

3. Every cycle \( C \) has a non-tree edge \( e \) with:

\[
  w(e) = \max_{f \in C} w(f)
\]

\[
  l(e) = \min_{f \in C} l(f)
\]

\( F_0 \)

\( F_1 \)
Proof of correctness

- Assume (3), the lightest replacement edge is on the maximum level

- The algorithm maintains (3)
Proof of correctness

- Assume (3), the lightest replacement edge is on the maximum level
  - Compare two replacement edges $e_1, e_2$, if $w(e_1) \leq w(e_2)$, we need to prove $l(e_1) \geq l(e_2)$
  - $e_1, e_2$ can form cycles $C_1, C_2$ with the original tree
Proof of correctness

- Assume (3), the lightest replacement edge is on the maximum level
  - Compare two replacement edges $e_1, e_2$, if $w(e_1) \leq w(e_2)$, we need to prove $l(e_1) \geq l(e_2)$
  - $e_1, e_2$ can form cycles $C_1, C_2$ with the original tree
    - $e_1, e_2$ must be largest edges in $C_1, C_2$, resp. Otherwise original tree is not minimum
Assume (3), the lightest replacement edge is on the maximum level

- Compare two replacement edges \( e_1, e_2 \), if \( w(e_1) < w(e_2) \), we need to prove \( l(e_1) \geq l(e_2) \)
- \( e_1, e_2 \) can form cycles \( C_1, C_2 \) with the original tree
  - \( e_1, e_2 \) must be largest edges in \( C_1, C_2 \), resp. Otherwise original tree is not minimum
- \( C = C_1 \oplus C_2 \) is also a cycle with \( e_1 \) and \( e_2 \), and \( w(e_2) \) is the largest in \( C \), so \( l(e_2) \) is lowest.

(3) Every cycle \( C \) has a non-tree edge \( e \) with largest weight and lowest level
Proof of correctness

- The algorithm maintains (3):

- When the level of $e$ increases, $e$ is in $T(u)$
  - Assume $e$ is the unique lowest largest edge on some cycle $C$
  - All other edges of $C$ incident to $T(u)$ have level $>l(e)$
  - $C$ cannot leave $T(u)$
  - So all other edges in $C$ have level $>l(e)$, so (3) is maintained when $l(e)$ increases by 1

(3) Every cycle $C$ has a non-tree edge $e$ with largest weight and lowest level
Proof of correctness

- The algorithm maintains (3):

- When the level of e increases, e is in T(u)
  - Assume e is the unique lowest largest edge on some cycle C
  - All other edge of C incident to T(u) have level > l(e)
  - C cannot leave T(u) (Otherwise there will be a replacement found.)
  - So all other edges in C have level > l(e), so (3) is maintained when l(e) increases by 1

(3) Every cycle C has a non-tree edge e with largest weight and lowest level
Update time

- Only need to maintain min-key in ET-tree structure

- Update time for this decremental MST is still $O(\log^2 n)$
Discussion

- Why is it hard to extend this to fully dynamic MST?
  - Unlike connectivity structures, we may need to change the forest when inserting an edge.
  - Totally breaking the order of the structure
Invariants

1. We keep the set of spanning forest \( F = F_0 \supseteq F_1 \supseteq \ldots \supseteq F_{\text{max}} \) on \( E_0, E_1, \ldots, E_{\text{max}} \)
2. The number of vertices of a tree in \( F_i \) is at most \( n/2^i \)
3. Every cycle \( C \) has a non-tree edge \( e \) with:
   \[
   w(e) = \max_{f \in C} w(f)
   \]
   \[
   l(e) = \max_{f \in C} l(f)
   \]

- If we insert an edge with very small weight:
Invariants

1. We keep the set of spanning forest $F=F_0\supseteq F_1\supseteq\ldots\supseteq F_{l_{\text{max}}}$ on $E_0,E_1,\ldots,E_{l_{\text{max}}}$

2. The number of vertices of a tree in $F_i$ is at most $n/2^i$

3. Every cycle $C$ has a non-tree edge $e$ with:

   \[ w(e) = \max_{f \in C} w(f) \]

   \[ l(e) = \max_{f \in C} l(f) \]

   If we insert an edge with very small weight:
   The MST will change, so as MST in higher levels
Invariants

1. We keep the set of spanning forest $F = F_0 \supseteq F_1 \supseteq \ldots \supseteq F_{\text{max}}$.
2. The number of vertices of a tree in $F_i$ is at most $n/2^i$.
3. Every cycle $C$ has a non-tree edge $e$ with:
   \[ w(e) = \max_{f \in C} w(f) \]
   \[ l(e) = \max_{f \in C} l(f) \]
   
   - If we insert an edge with very small weight:
     - The MST will change, so as MST in higher levels
     - Level decreasing will destroy the hierarchy

   Originally this edge at level 2, but we need to decrease this level after update.

Too large for this level if we add the new edge here.
Fully dynamic MST

- An $O(\log^4 n)$ amortized update time structure is given in:
  - “Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity”
  - By Holm, Lichtenberg, Thorup, *Journal of ACM* 2001

- Construct smaller decremental structure every time
- Complicated analysis
Overview of Dynamic Connectivity Results

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

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

<table>
<thead>
<tr>
<th></th>
<th>Edge Updates</th>
<th>Vertex Updates (Subgraph)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Amortized</strong></td>
<td>$O(\log^2 n)$</td>
<td>$\tilde{O}(m^{2/3})$, with query time $\tilde{O}(m^{1/3})$ [Chan, Pătraşcu &amp; Roditty ‘2008]</td>
</tr>
<tr>
<td></td>
<td>[Holm, Lichtenberg &amp; Thorup ‘1998]</td>
<td></td>
</tr>
<tr>
<td><strong>Worst-Case</strong></td>
<td>$O(n^{1/2})$</td>
<td>$\tilde{O}(m^{4/5})$, with query time $\tilde{O}(m^{1/5})$ [Duan 2010]</td>
</tr>
<tr>
<td></td>
<td>[O($m^{1/2}$) by Frederickson ‘1985]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[Improved by Eppstein, Galil, Italiano, Nissenzweig ‘1992]</td>
<td></td>
</tr>
</tbody>
</table>
d-failure Model

- **d-failure model:**
  - The number of “failed” vertices/edges is bounded by d.
  - It can be seen as a static structure, in which the query \((u,v)\) is given with a set \(D\) of “failed” vertices/edges and \(|D| \leq d\).
Next lecture

- Worst-case dynamic connectivity