COL702: Advanced Data Structures and Algorithms

Ragesh Jaiswal

CSE, IIT Delhi

Kruskal using DSDS

- Kruskal’s algorithm uses a sequence of DSDS operations.
- The usual strategy for calculating the running time:
  - Count the number of each operation and multiply it with the worst-case times for these operations.
- Advantage: Easy
- Disadvantage: Pessimistic estimate
  - Each Find/Union operation may not take the same time.
  - Even more serious for data structures that get dynamically restructured.

Amortized Analysis

- Starting from a base configuration of the dynamic data structure, we are interested in finding the time for a sequence of m operations.
- Amortized analysis: Instead of computing the worst-case running time of an operation, compute the amortized time (*how much does a "typical" operation cost instead of the worst-case time*).
  - Averaging method: Let T be the total time for a sequence of m operations. Then, the amortized time for an operation is $\frac{T}{m}$.
  - Accounting method: Distribute the cost of a few time-taking operations among many fast operations.

Amortized analysis: Accounting method

- Accounting method:
  - Open a bank account at the very beginning.
  - Every operation has an associated payment. Every basic computational step is charged one unit of money.
  - If the payment exceeds the computational steps involved in the operation, the extra money goes to the bank account (to be used later).
  - On the other hand, if the steps involved are more than the payment, one can pay the difference from the bank account.
  - The bank balance should be ≥ 0 at every step.
  - The payment associated with an operation is the amortized time for that operation.
  - As long as the bank balance does not become negative, the sum of amortized time is an upper bound on the sum of real times of the operations.

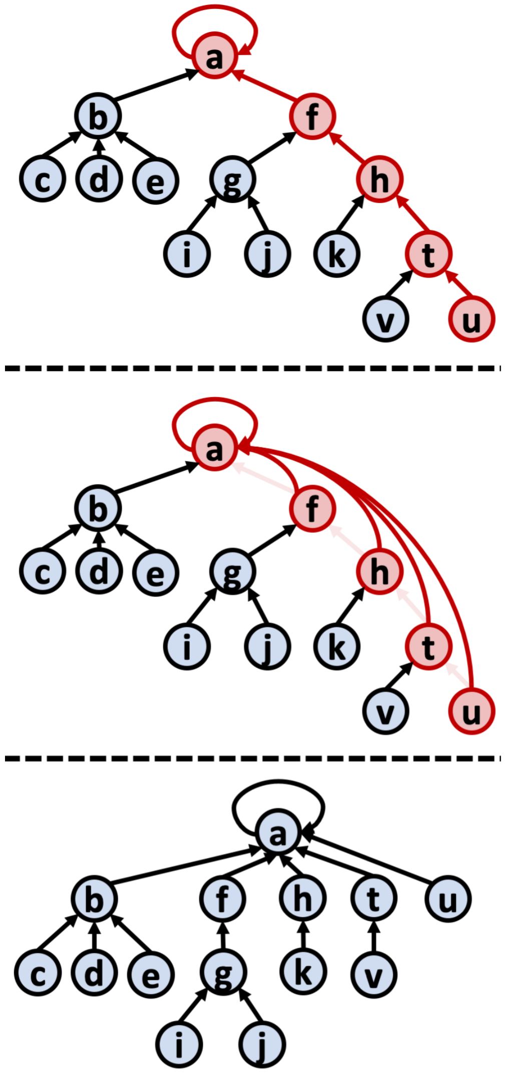
DSDS with path compression

`Makeset`(S)
- For every $u \in S$:
  - $parent[u] = u; rank[u] = 0$

`Union`(u,v)
- ru =`Find`(u); rv =`Find`(v)
- `Union-by-rank`(ru,rv)

`Union-by-rank`(ru,rv)
- if ($rank[ru] > rank[rv]$)
  - $parent[rv] = ru$
- if ($rank[ru] < rank[rv]$)
  - $parent[ru] = rv$
- if ($rank[ru] = rank[rv]$)
  - $parent[rv] = ru$
  - $rank[ru] = rank[ru] + 1$

`Find`(u)
- if ($parent[u] = u$)return(u)
- $parent[u]$ = `Find`($parent[u]$)
- return($parent[u]$))



Accounting for path-compression

- The bank account money is stored in the tree nodes.
- *Money restructuring*: When the parent ru is made the parent of root rv, half the money in rv is moved to ru. Rounding is done using one unit of extra payment.
- Makeset(u):
  - *Payment*: 3 units. One unit is used to pay for setting up the node. Two units are stored in the node.
- Union-by-rank(ru, rv):
  - *Payment*: 2 units. One unit for changing pointer. One unit for rounding.
- Find(u): Pull out one unit of money stored at all nodes whose pointer changes on executing the Find(u) operation.
  - *Payment*: Number of "broke" nodes.

Accounting for path-compression

- The bank account money is stored in the tree nodes.
- *Money restructuring*: When the parent ru is made the parent of root rv, half the money in rv is moved to ru. Rounding is done using one unit of extra payment.
- Makeset(u):
  - *Payment*: 3 units. One unit is used to pay for setting up the node. Two units are stored in the node.
- Union-by-rank(ru, rv):
  - *Payment*: 2 units. One unit for changing pointer. One unit for rounding.
- Find(u): Pull out one unit of money stored at all nodes whose pointer changes on executing the Find(u) operation.
  - *Payment*: Number of "broke" nodes.
- Amortized time for Find and Union: O(number of broke nodes)

Number of broke nodes

- *<u>Note 1</u>:* rank[u] does not necessarily store the depth of the tree rooted at u.
- <u>Lemma 1</u>: A root node with rank r holds at least $2 \cdot \left(\frac{3}{2}\right)^r$ units of money.
- *<u>Note 2</u>:* Rank of a non-root node does not change.

- <u>Corollary of Lemma 1</u>: At the time when a node with rank r becomes a non-root, it has accumulated at least $\left(\frac{3}{2}\right)^r$ amount of money.

Number of broke nodes

- Lemma 2: The following three properties hold:
  a) $rank[parent[u]] > rank[u]$ for any non-root node u.
  b) Every time a node u spends one unit of money, the rank of its parent increases.
  c) If a node v of rank r has gone broke, $rank[parent[v]] \geq \left(\frac{3}{2}\right)^r$.

Number of broke nodes

- Lemma 2: The following three properties hold:
  a) $rank[parent[u]] > rank[u]$ for any non-root node u.
  b) Every time a node u spends one unit of money, the rank of its parent increases.
  c) If a node v of rank r has gone broke, $rank[parent[v]] \geq \left(\frac{3}{2}\right)^r$.

- Theorem 1: The number of broke vertices in the path from any node to the root of its tree is $O(\log^*_{\frac{3}{2}} n)$.

Number of broke nodes

- Lemma 2: The following three properties hold:
  a) $rank[parent[u]] > rank[u]$ for any non-root node u.
  b) Every time a node u spends one unit of money, the rank of its parent increases.
  c) If a node v of rank r has gone broke, $rank[parent[v]] \geq \left(\frac{3}{2}\right)^r$.

- Theorem 1: The number of broke vertices in the path from any node to the root of its tree is $O(\log^*_{\frac{3}{2}} n)$.

- Running time of Kruskal's algorithm using DSDS with path-compression: $|\mathrm{E}| \cdot O(\log^* |V|)$.