Union-Find

Algorithms for Big Data
Andrei Bulatov

Previous Lecture

- Kruskal’s Algorithm
- Soundness of Kraskal’s algorithm
- Implementation details
- Prim’s algorithm
- Clustering

Kruskal's Algorithm

Input: graph G with weights c_e

Output: a minimum spanning tree of G

Method:

$T \coloneqq \emptyset$

while $|T| < |V| - 1$ do

pick an edge e with minimum weight such that

it is not from T and

$T \cup \{e\}$ does not contain cycles

set $T \coloneqq T \cup \{e\}$

The Union-Find Data Structure

- To work efficiently Kruskal's algorithm requires a data structure to store the collection of connected components of a graph and merge then when necessary
- More precisely, the data structure has to support the following operations:
  - `makeset`(x) create a singleton set containing just x
  - `find`(x) returns a label to which set x belongs
  - `union`(x, y) merge the sets containing x and y

Kruskal's Algorithm with Union-Find

- To avoid searching for an edge of minimum weight, sort the edges in the beginning of the algorithm
- If T is the current set of selected edges, the data structure contains the collection of connected components of (V,T)
- To check whether edge (v,w) forms a cycle just check if $\mathtt{find}(v) = \mathtt{find}(w)$
- Follow the sorted list of edges. Every time for edge (v,w), if it forms a cycle, remove it from the list. If it does not, merge the sets containing v and w, and remove the edge from the list

Kruskal's Algorithm with Union-Find (Running Time)

- Running time:
  - $O(|E|\log|E|) = O(|E|\log|V|)$ for sorting
  - Need to consider $|E|$ edges
- Goal: perform `find` and `merge` in $O(\log|V|)$ time

Kruskal's Algorithm

Input: graph $G = (V, E)$ with weights c_e

Output: a minimum spanning tree of G

Method:

```
sort E according to c_e
for v ∈ V do
   makeset(v)
T := ∅
while E ≠ ∅ do
  pick the first edge e = (u, v) from E
  remove e from E
  if find(u) ≠ find(v) then
     union(u, v)
     set T := T ∪ {e}
```

Implementation

- Represent every set as a tree: Equip every element of the set with a pointer to the predecessor. Label the set with its root
- Also we define the rank of vertices, which is for now the height of the subtree rooted at the vertex in the data structure

makeset(v)
  set $\pi(v) = v$
  set rank$(v) = 0$

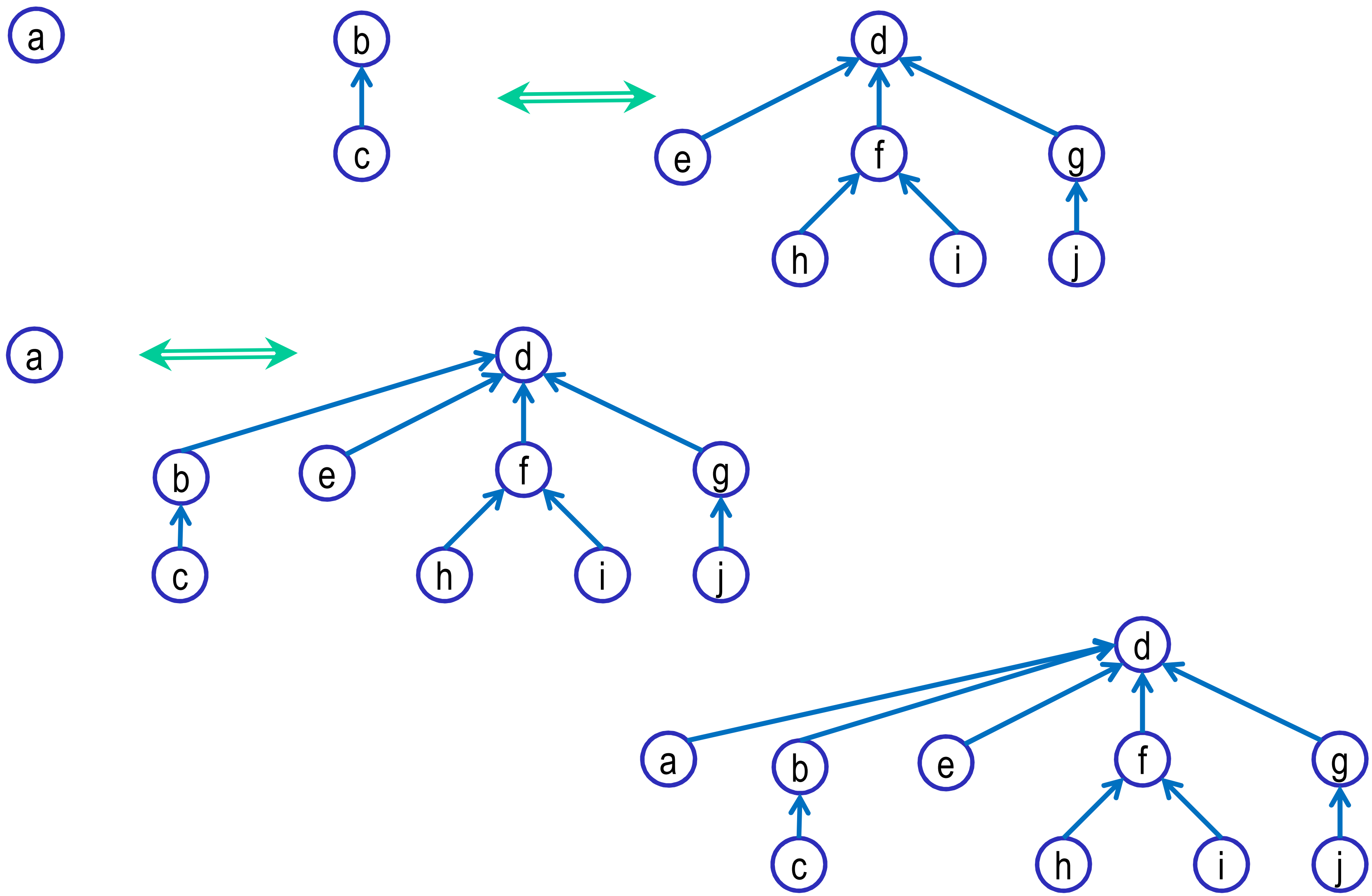
find(v)
  while $\pi(v) \neq v$ do $v = \pi(v)$
  return v

Implementation (cntd)

- Merge by connecting the root of one set to the root of the other
- Have to do it carefully to keep the rank as low as possible

union(v, w)
  set $r_v =$ find(v)
  set $r_w =$ find(w)
  if $r_v = r_w$ then return
  if rank(v) $>$ rank(w) then
    set $\pi(r_w) = r_v$
  else
    set $\pi(r_v) = r_w$
    if rank(r_v) $=$ rank(r_w) then
      set rank(r_w) $=$ rank(r_w) $+ 1$

Example



Properties of Rank

- **Property 1**.

  For any v (except a root) $\mathtt{rank}(v) < \mathtt{rank}(\pi(v))$

- **Property 2.**

  Any vertex of rank k has at least 2^k vertices in the subtree rooted at that vertex

- **Property 3.**

  If there are n vertices overall, there can be at most $\frac{n}{2^k}$ vertices of rank k

Running Time

- Property 2 implies that the maximal length of a sequence
$$v \to \pi(v) \to \pi(\pi(v)) \to \cdots \to \mathrm{root}$$
can be at most $O(\log n)$
- Therefore the running time of `find` is $O(\log n)$, and we get the desired running time for Kruskal's algorithm

Path Compression

- In the general case there is no sense to improve the running time of the Union-Find data structure, because sorting edges in the beginning of Kruskal's algorithm takes $O(|E|\log|V|)$ time, and this time dominates the overall running time of the algorithm
- However there may be cases when sorting can be done faster, say, in linear time
- Improve the `find` procedure

```
find(v)
  if π(v) ≠ v then
     set π(v) = find(π(v))
  return π(v)
```

Amortized Running Time

- Although we cannot improve the running time of any single `find` operation, these small changes will improve the running time of the sequence of ALL `find` operations during the execution of the algorithm
- This is the idea of the amortized running time (or amortized complexity)
- Let $\log^* n$ denote the smallest number k such that

$$\underbrace{\log(\log(\log(\ldots\log}_{k \text{ times}} n))) \leq 1$$

Amortized Running Time (cntd)

- $\log^* n = 0$ for $n = 1$

  $\log^* n = 1$ for $n = 2$

  $\log^* n = 2$ for $n = 3,4$

  $\log^* n = 3$ for $n = 5, \dots, 16$

  $\log^* n = 4$ for $n = 17, \dots, 2^{16} = 65536$

  $\log^* n = 5$ for $n = 65537, \dots, 2^{65536}$

Claim.

With path compression a sequence of $|E|$ `find` operations can be completed in time $O(|E| \log^* |V|)$

Amortized Running Time (cntd)

- Let $n = |V|$. Subdivide the set $\{1, \dots, n\}$ into intervals according to the value of $\log^* k$:
$\{1\}, \{2\}, \{3,4\}, \{5, \dots, 16\}, \{17, \dots, 2^{16}\}, \{65536, \dots, 2^{65536}\}$
- Every vertex belonging to the interval $\{k+1, \dots, 2^k\}$ receives a budget of 2^k dollars
- By Property 3 there are at most $\frac{n}{2^k}$ vertices of rank k. Therefore the total budget of vertices from the interval $\{k+1, \dots, 2^k\}$ is at most

$$2^k \left(\frac{n}{2^{k+1}} + \frac{n}{2^{k+2}} + \cdots \right) \le 2^k \cdot \frac{n}{2^k} = n$$

- Therefore the total budget of all vertices is at most $n \log^* n$.

Amortized Running Time (cntd)

- Consider the execution of $\mathtt{find}(v)$
- It produces a sequence of vertices
$$v \to v_1 \to v_2 \to \cdots \to \mathrm{root}$$
where $v_{i+1} = \pi(v_i)$ and $\mathrm{rank}(v_i) < \mathrm{rank}(v_{i+1})$
- Vertices v_i can be of two types:
  - $\log^*(\mathrm{rank}(v_i)) < \log^*(\mathrm{rank}(v_{i+1}))$
  - the rest
- There are at most $\log^* n$ vertices of the first type
- For them making step to the predecessor $\pi(v_i)$ is free
- Vertices of the second type pay \$1 for each step

Amortized Running Time (cntd)

- Note that since after a vertex v_i whose rank is in the interval $\{k+1,\dots,2^k\}$ pays its way to the beginning of the interval, $\pi(v_i)$ is assigned to be a vertex whose rank is in the higher interval, and therefore v_i never pays again
- Vertex v_i has to pay at most 2^k times, and so it stays within its budget
- Therefore, every `find` operation runs for at most $\log^* n$ `free' steps, plus all `find` operations together require at most $n \log^* n$ `paid' steps
- The total running time of all `find` operations is therefore $O(|E| \log^* n + n \log^* n) = O(|E| \log^* n)$

Homework and Reading

Exercises from the Book:

page 149-150 5.11, 5.12

Reading

Chapters 5.2