COMP 6651 / Winter 2022
Dr. B. Jaumard

Lecture 8: Union-Find Partition Structures

March 11, 2022

Outline

Disjoint-set data structure

In computing, given a set of elements, it is often useful to break them up or partition them into a number of separate, non overlapping sets. A **disjoint-set data structure** is a data structure that keeps track of such a partitioning. A **union-find algorithm** is an algorithm that performs two useful operations on such a data structure:

- **Find:** Determine which set a particular element is in. Also useful for determining if two elements are in the same set.
- **Union:** Combine or merge two sets into a single set.
- **MakeSet:** Make a set containing only a given element

Because it supports the first two operations, a disjoint-set data structure is sometimes called a union-find data structure or merge-find set. With these three operations, many practical partitioning problems can be solved.

Partitions with Union-Find Operations

Given a collection $\mathcal{A} = \{A_1, A_2, \ldots, A_k\}$ of disjoint dynamic sets:

- makeSet(x): Creates a singleton set containing the element x and returns the position storing x in this set.
- union(A_i, A_j): Returns the set $A_i \cup A_j$, destroying the old sets A_i and A_j.
- find(p): Returns the set containing element p.

An Application: Determining the Connected Components of an Undirected Graph (1/3)

- An **undirected graph is connected** if every node is reachable from all other nodes.
- The **connected components of a graph** are the equivalence classes of nodes under the “reachable from” relation.
- An undirected graph is connected if it has exactly one connected component.

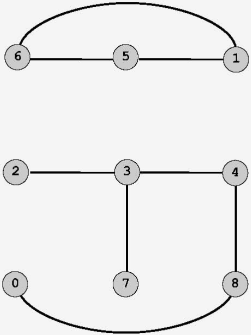
An Application: Determining the Connected Components of an Undirected Graph (2/3)

Given a set of objects

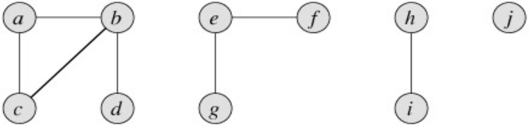
- Union: connect two objects.
- Find: is there a path connecting the two objects?

more difficult problem: find the path

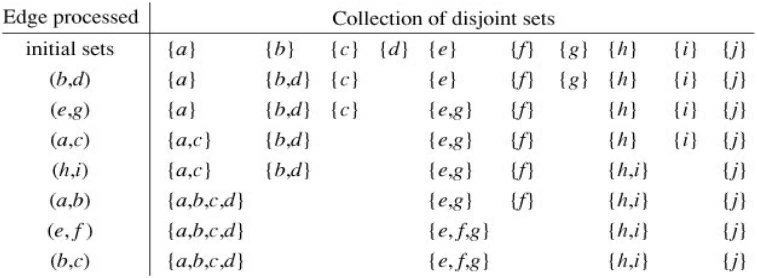
```
union(3, 4)
union(8, 0)
union(2, 3)
union(5, 6)
 find(0, 2)       no
 find(2, 4)       yes
union(5, 1)
union(7, 3)
union(1, 6)
union(4, 8)
 find(0, 2)       yes
 find(2, 4)       yes
```



An Application: Determining the Connected Components of an Undirected Graph (3/3)



(a)

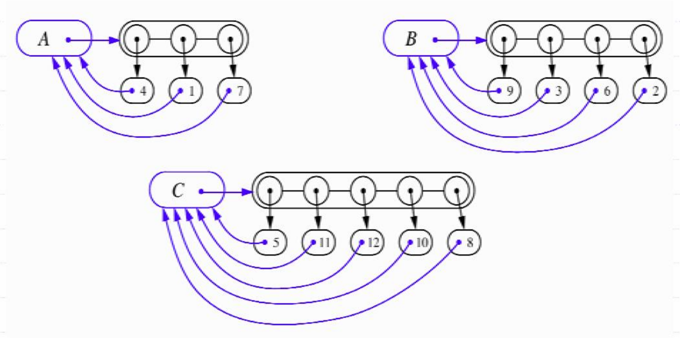


Edge processed	Collection of disjoint sets									
initial sets	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$	$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(b,d)	$\{a\}$	$\{b,d\}$	$\{c\}$		$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(e,g)	$\{a\}$	$\{b,d\}$	$\{c\}$		$\{e,g\}$	$\{f\}$		$\{h\}$	$\{i\}$	$\{j\}$
(a,c)	$\{a,c\}$	$\{b,d\}$			$\{e,g\}$	$\{f\}$		$\{h\}$	$\{i\}$	$\{j\}$
(h,i)	$\{a,c\}$	$\{b,d\}$			$\{e,g\}$	$\{f\}$		$\{h,i\}$		$\{j\}$
(a,b)	$\{a,b,c,d\}$				$\{e,g\}$	$\{f\}$		$\{h,i\}$		$\{j\}$
(e,f)	$\{a,b,c,d\}$				$\{e,f,g\}$			$\{h,i\}$		$\{j\}$
(b,c)	$\{a,b,c,d\}$				$\{e,f,g\}$			$\{h,i\}$		$\{j\}$

(b)

List-Based Implementation

- Each set is stored in a sequence represented with a linked-list
- Each node should store an object containing the element and a reference to the set name.



Quick-find [eager approach]

Data structure.

- Integer array `id[]` of size `N`.
- Interpretation: `p` and `q` are connected if they have the same id.

i	0	1	2	3	4	5	6	7	8	9
id[i]	0	1	9	9	9	6	6	7	8	9

5 and 6 are connected
2, 3, 4, and 9 are connected

Find. Check if `p` and `q` have the same id.

id[3] = 9; id[6] = 6
3 and 6 not connected

Union. To merge sets containing `p` and `q`, change all entries with `id[p]` to `id[q]`.

i	0	1	2	3	4	5	6	7	8	9
id[i]	0	1	6	6	6	6	6	7	8	6

union of 3 and 6
2, 3, 4, 5, 6, and 9 are connected

problem: many values can change

Analysis of List-based Representation

- When doing a union, always move elements from the smaller set to the larger set.
  - Each time an element is moved it goes to a set of size at least double its old set.
  - Thus, an element can be moved at most $O(\log n)$ times.

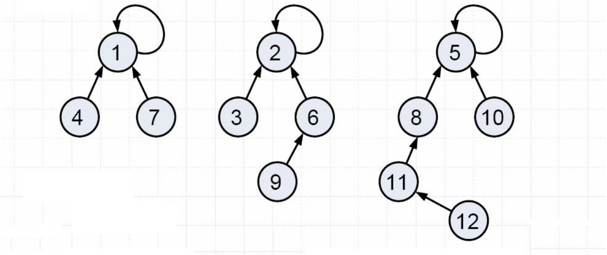
$$1 + 2 + 2^2 + 2^3 + \cdots + 2^k = 1 + 2(2^k - 1) \leq n$$

$$\hookrightarrow \qquad k \leq \log_2(\frac{n-1}{2} + 1) = O(\log_2 n)$$

- Assuming we start with n elements, total time needed to perform
  - ▶ n union or find operations: $O(n \log n)$
  - ▶ m union or find operations: $O(m \log n)$

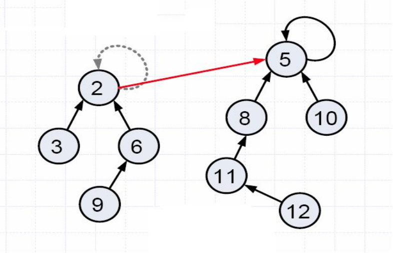
Tree-Based Implementation

- Each element is stored in a node, which contains a pointer to a **set** name.
- A node *v* whose set pointer points back to *v* is also a set name.
- Each set is a tree, rooted at a node with a self referencing set pointer.
- For example: Sets "1", "2", and "5":

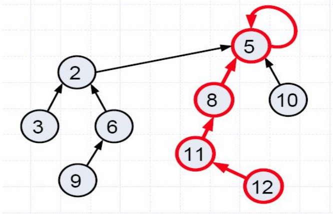


Union-Find Operations

To perform a **union**, simply make the root of one tree point to the root of the other.

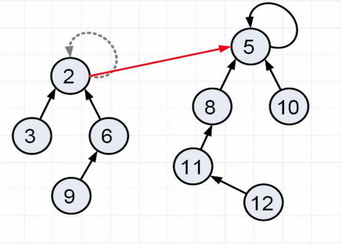


To perform a **find**, follow set name pointers from the starting node until reaching a node whose set name pointer refers back to itself.



Union-find Heuristic 1

- Union by size:
    - When performing a union, make the root of smaller tree point to the root of the larger.
- Implies $O(n \log n)$ time for performing n union find operations:
    - Each time we follow a pointer, we are going to a subtree of size at least double the size of the previous subtree.
    - Thus, we will follow at most $O(\log n)$ pointers for any find.



log star - $\log^\star$ n

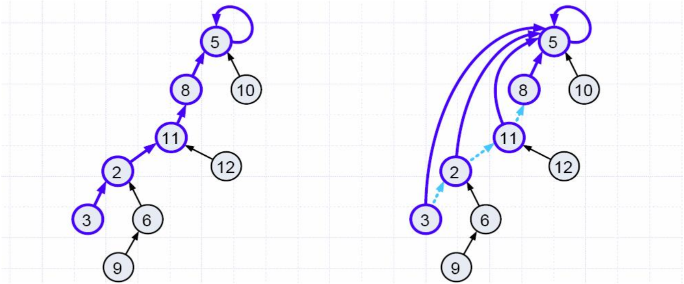
The iterated logarithm of n, written $\log^\star n$, is the number of times the logarithm function must be iteratively applied before the result is less than or equal to 1. The simplest formal definition is the result of this recursive function:

$$\log^\star n = \begin{cases} 0 & \text{if } n \leq 1 \\ 1 + \log^\star(\log n) & \text{if } n > 1 \end{cases}$$

x	$\log^\star x$
$(-\infty; 1]$	0
$(1; 2]$	1
$(2; 4]$	2
$(4; 16]$	3
$(16; 65,536]$	4
$(65,536; 2^{65,536}]$	5

Union-find Heuristic 2

- Path compression:
  - After performing a find, compress all the pointers on the path just traversed so that they all point to the root.



- Implies $O(n \log^{\star} n)$ time for performing n union find operations:
  - Proof is beyond scope of this class.

Proof of $\log^* n$ Amortized Time

- For each node v that is a root
  - define $n(v)$ to be the size of the subtree rooted at v (including v)
  - identify a set with the root of its associated tree.
- We update the size field of v each time a set is 'unioned' into v. Thus, if v is not a root, then $n(v)$ is the largest subtree rooted at v, which occurs just before we 'union' v into some other node whose size is at least as large as v's.
- For any node v, then, define the **rank** of v, which we denote as $r(v)$, as $r(v) = [\log n(v)]$:
- Thus, $n(v) \geq 2^{r(v)}$.
- Also, since there are at most n nodes on the tree of v, $r(v) = [\log n]$, for each node v.

Proof of $\log^* n$ Amortized Time (2)

- For each node v with parent w:
  - $r(v) > r(w)$
- **Claim:** There are at most $n/2^s$ of rank s
- **Proof:**
  - Since $r(v) < r(w)$, for any node v with parent w, ranks are monotonically increasing as we follow parent pointers up any tree.
  - Thus, if $r(v) = r(w)$ for two nodes v and w, then the nodes counted in $n(v)$ must be seperate and distinct from the nodes counted in $n(w)$.
  - If a node v is of rank s, then $n(v) \geq 2^s$.
  - Therefore, since there are at most n nodes total, there can be at most $n/2^s$ that are of rank s.

Proof of $\log^* n$ Amortized Time (3)

- Definition: Tower of two's function:
  - $t(i) = 2^{t(i-1)}$
- Nodes v and u are in the same rank group g if
  - $g = \log^*(r(v)) = \log^*(r(u))$:
- Since the largest rank is $\log n$, the largest rank group is
  - $\log^*(\log n) = (\log^* n) - 1$:

Proof of $\log^* n$ Amortized Time (4)

- Charge 1 cyber-dollar per pointer hop during a find:
  - If w is the root or if w is in a different ramk group than v, then charge the find operation one cyber dollar.
  - Otherwise (w is not a root and v and w are in the same rank group), charge the node v one cyber dollar.
- Since there are at most $(\log^* n) - 1$ rank groups, this rule guarantees that any find operation is charged at most $\log^* n$ cyber-dollars.

Proof of $\log^* n$ Amortized Time (5)

- After we charge a node v then v will get a new parent, which is a node higher up in v's tree.
- The rank of v's new parent will be greater than the rank of $v's$ old parent w.
- Thus, any node v can be charged at most the number of different ranks that are in v's rank group.
- If v is in rank group $g > 0$, then v can be charged at most $t(g) - t(g-1)$ times before v has a parent in a higher rank group (and from that point on, v will never be charged again). In other words, the total number, C, of cyber dollars that can ever be charged to nodes can be bounded by

$$C \leq \sum_{g=1}^{\log^*(n-1)} n(g) \times (t(g) - t(g-1))$$

Proof of $\log^* n$ Amortized Time (end)

- Bounding $n(g)$:

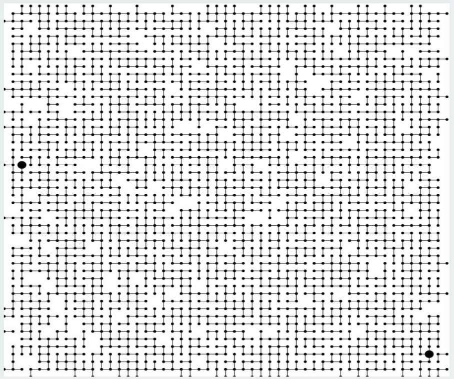
$$
\begin{aligned}
n(g) &\leq \sum_{s=t(g-1)+1}^{r(g)} \frac{n}{2^s} \\
&= \frac{n}{2^{t(g-1)+1}} \sum_{s=0}^{t(g)-t(g-1)-1} \frac{1}{2^s} \\
&< \frac{n}{2^{t(g-1)+1}} \times 2 \\
&= \frac{n}{2^{t(g-1)}} \\
&= \frac{n}{t(g)}
\end{aligned}
$$

- Returning to C:

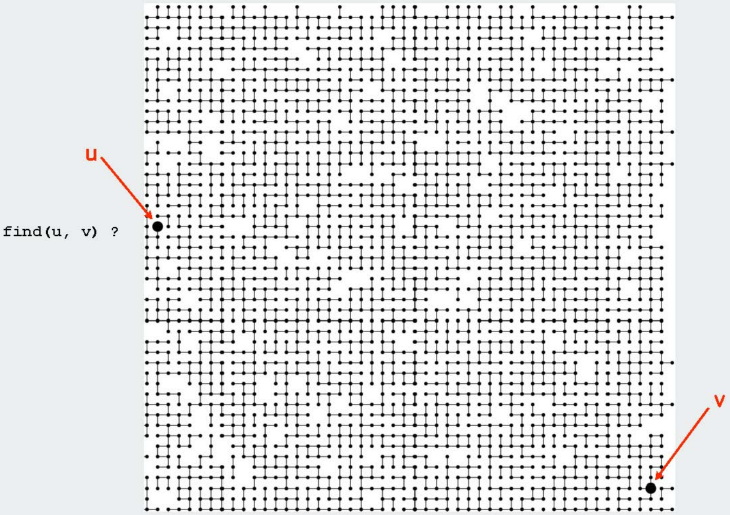
$$
\begin{aligned}
C &< \sum_{g=1}^{\log^*(n-1)} \frac{n}{t(g)}(t(g) - t(g-1)) \\
&\leq \sum_{g=1}^{\log^*(n-1)} \frac{n}{t(g)} t(g) \\
&= \sum_{g=1}^{\log^*(n-1)} n \\
&\leq n \log^* n
\end{aligned}
$$

Network Connectivity

- **union** command: connect two objects
- **find** query: is there a path connecting one object to another?



Network Connectivity



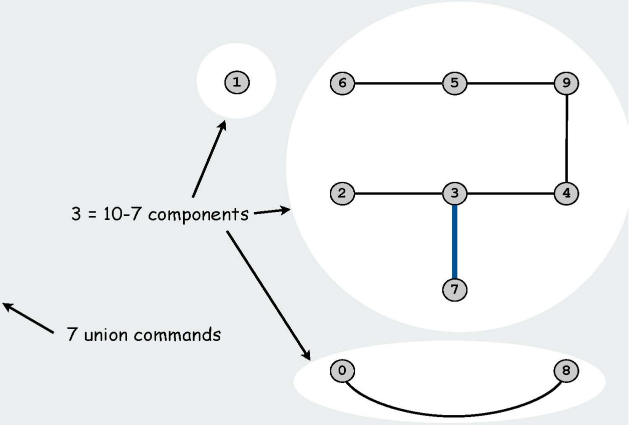
find(u, v) ?

Network Connectivity: Connected Components

Connected component: set of mutually connected vertices

Each union command reduces by 1 the number of components

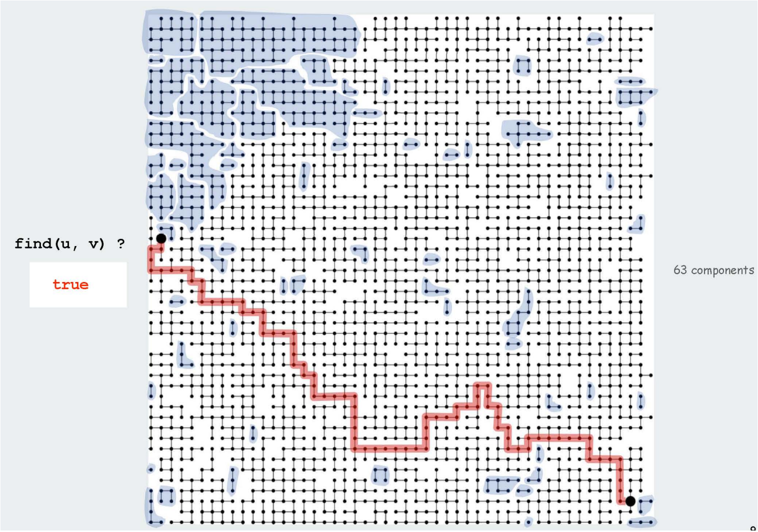
```
in      out

3 4     3 4
4 9     4 9
8 0     8 0
2 3     2 3
5 6     5 6
2 9
5 9     5 9
7 3     7 3
```



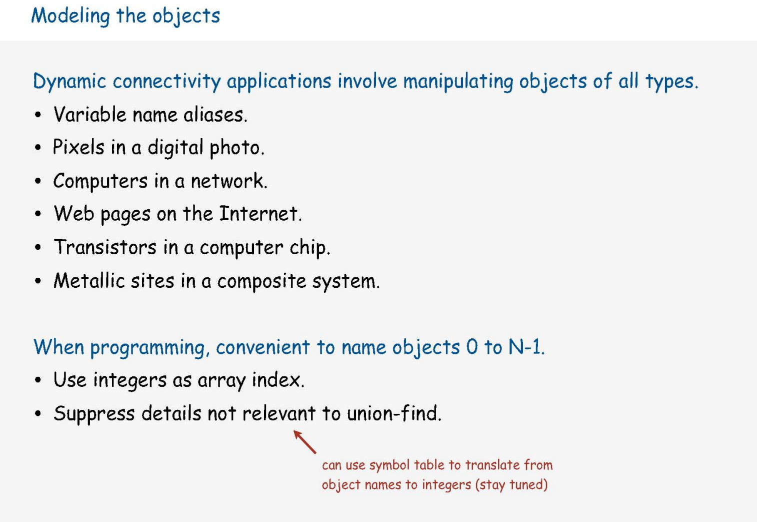
3 = 10-7 components

7 union commands

Network Connectivity



Network Connectivity Applications



Modeling the objects

Dynamic connectivity applications involve manipulating objects of all types.

- Variable name aliases.
- Pixels in a digital photo.
- Computers in a network.
- Web pages on the Internet.
- Transistors in a computer chip.
- Metallic sites in a composite system.

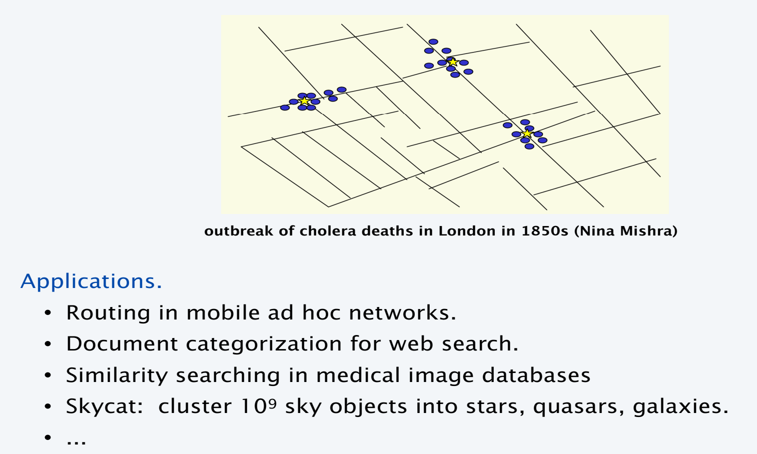
When programming, convenient to name objects 0 to N-1.

- Use integers as array index.
- Suppress details not relevant to union-find.

can use symbol table to translate from object names to integers (stay tuned)

Clustering

Given a set O of n objects labeled $O_1, O_2, \ldots, O_n$, partition into clusters so that objects in different clusters are far apart.



outbreak of cholera deaths in London in 1850s (Nina Mishra)

Applications.

- Routing in mobile ad hoc networks.
- Document categorization for web search.
- Similarity searching in medical image databases
- Skycat: cluster 10^9 sky objects into stars, quasars, galaxies.
- ...

Clustering: Single-Linkage Algorithm

- Set of N entities: $O = \{O_1, O_2, \ldots, O_N\}$ to be classified
- Matrix of dissimilarities: $D = (d_{k\ell})$ between all pairs of entities.
- Assumptions: $d_{k\ell} \geq 0, d_{k\ell} = d_{\ell k}, d_{kk}$ for $k, \ell = 1, 2, \ldots, N$
- Triangular inequality ($d_{k\ell} + d_{\ell m} \geq d_{km}$) does not necessarily hold.
- General objective of clustering: partition the entities into well-separated and homogeneous clusters.

Clustering: Split and Diameter

- Partition $P_M = \{C_1, C_2, \ldots, C_M\}$ of O into M clusters
- Split of cluster C_j:

$$s(C_j) = \min_{k,\ell:\ O_k \in C_j,\ O_\ell \notin C_j,\ j=1,2,\ldots,M} d_{k\ell}.$$

- Split of partition P_M: $s(P_M) = \min_{j=1,2,\ldots,M} s(C_j)$.
- Diameter of cluster C_j:

$$d(C_j) = \max_{k,\ell:\ O_k \in C_j,\ O_\ell \in C_j} d_{k\ell}.$$

- Diameter of partition P_M: $d(P_M) = \max_{j=1,2,\ldots,M} d(C_j)$.

Clustering

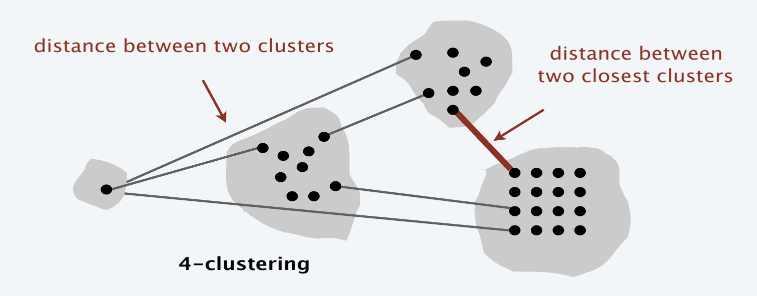
k-clustering. Divide objects into *k* non-empty groups.

Dissimilarities. Numeric value specifying "closiness" of two objects

- $d(O_i, O_i) = 0$ [identity of undiscernibles]
- $d(O_i, O_j) \geq 0$ [nonnegativity]
- $d(O_i, O_j) = d(O_j, O_i)$ [symmetry]

Spacing (Split). Min distance between any pair of points in different clusters

Goal. Given an integer *k*, find a *k* clustering of maximum spacing/split



Single-Linkage Algorithm (SLA)

(a) Let $P_N = \{C_1, C_2, \ldots, C_N\}$ where $C_j = \{O_j\}$ for $j = 1, 2, \ldots, N$ and $k = 0$.

(b) Find two clusters C_i and $C_j \in P_{N-k}$ such that $s(C_i) = s(C_j) = s(P_{N-k})$, i.e., identify the two closest clusters

(c) Obtain P_{N-k-1} from P_{N-k} by setting $C_{N+k+1} \leftarrow C_i \cup C_j$. Set $k \leftarrow k + 1$. If $k < N - 1$, return to (b).

The SLA algorithm maximizes $s(P_M)$ for all M.

Complexity:
$O(N^2 \log N)$ using Union-Find Data Structures (linked lists)
Be aware that one needs to update the split values whenever there is a union operation $\rightsquigarrow O(N)$

Single-Linkage Algorithm (SLA): Complexity Analysis

- $O(N^2 \log N)$ to sort the N^2 dissimilarities values
- After each merge iteration, the distance metric can be updated in $O(N)$.
- We pick the next pair to merge by finding the smallest distance that is still eligible for merging. If we do this by traversing the N^2 sorted list of distances, then, by the end of clustering, we will have done N^2 traversal steps.
- Adding all this up gives you $O(N^2 \log N)$.

Kruskal's Algorithm - Minimum Spanning Tree

Step 1. Set $\textsc{mst} = \emptyset$ and $F \leftarrow E$, the set of all edges.

Step 2. Choose an edge e in F of minimum weight, and check whether adding e to $\textsc{mst}$ creates a cycle.
If yes, remove e from F.
If no, move e from F to $\textsc{mst}$.

Step 3. If $F = \emptyset$, stop and output the minimal spanning tree $(V, \textsc{mst})$. Otherwise go to Step 2.

Complexity: $O(M \log M)$ using Union-Find Data Structures

Kruskal's Algorithm: Implementation

Theorem

Kruskal's Algorithm can be implemented in $O(M \log M)$ time.

- Sort edges by weights/dissimilarities
- Use union-find data structure to dynamically maintain connected component

```
KRUSKAL ($V, E, d$)
Sort $m$ edges by weight so that $w(e_1) \leq w(e_2) \leq \cdots \leq w(e_m)$
$S \leftarrow \emptyset$
For each $v \in V$: MAKESET($v$)
For $i = 1$ to $m$ do
        $(u, v) \rightarrow e_i$
        If FINDSET ($u$) $\neq$ FINDSET ($v$) $\rightsquigarrow$ are $u$ and $v$ in same component?
                $S \leftarrow S \cup \{e_i\}$
                UNION ($u, v$) $\rightsquigarrow$ make $u$ and $v$ in same component?
Return $S$
```

Kruskal's Algorithm: Complexity

- Sorting the edges: $O(m \log m)$ for m edges

$$m \leq n^2, \text{ so } \log m < \log n^2 = 2 \log n$$

Therefore sorting takes $O(m \log n)$ time.

- At most $2m$ find operations: $O(m)$ time.
- At most $n - 1$ union operations: $O(n \log n)$ time.

$\leadsto$ Total running time of $O(m \log n + 2m + n \log n)$.
The biggest term is $m \log n$ since $m > n$ if the graph is connected.
Overall complexity: $O(m \log n)$

SLA vs. Kruskal's Algorithm

SLA k-Clustering

The components $C_1, C_2, \ldots, C_k$ formed by deleting the $(k-1)$ most expensive edges of the minimum spanning tree MST constitute a k-clustering of maximum split.

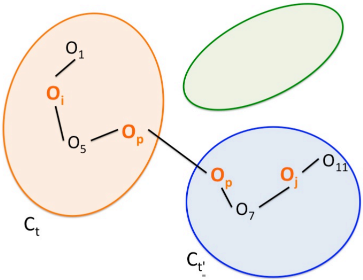
SLA vs. Kruskal's Algorithm

SLA *k*-Clustering

The components $C_1, C_2, \ldots, C_k$ formed by deleting the $(k-1)$ most expensive edges of the minimum spanning tree MST constitute a k-clustering of maximum split.

Deletion of $(k-1)$ most expensive edges in MST $\leadsto P^\star_k = \{C^\star_1, C^\star_2, \ldots, C^\star_k\}$.
Denote by $P_k = \{C_1, C_2, \ldots, C_k\}$ some other clustering.

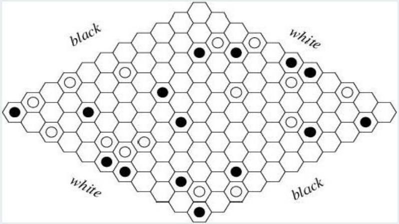
- The split of $P^\star_k$ is the length $s(P^\star_k)$ of the $(k-1)$st most expensive edge.
- Let O_i, O_j be in the same cluster in $P^\star_k$, say $C^\star_\ell$ an $C^\star_{\ell'}$, but different clusters in P_k, say C_t and $C_{t'}$.
- Some edge (O_p, O_q) on $O_i - O_j$ path in $P^\star_k$ spans two different clusters in P_k.
- All edges on $O_i - O_j$ path have length $\leq s(P^\star_k)$. since Kruskal's algorithm chose them.
- $s(P_k) \;\leq\; s(P^\star_k)$ since O_p and O_q are in different clusters



Network Connectivity

Hex. [Piet Hein 1942, John Nash 1948, Parker Brothers 1962]

- Two players alternate in picking a cell in a hex grid.
- Black: make a black path from upper left to lower right.
- White: make a white path from lower left to upper right.



Reference: http://mathworld.wolfram.com/GameofHex.html

Union-find application. Algorithm to detect when a player has won.

References

- SLA Algorithm: see, e.g., J. Kleinberg and E. Tardos, *Algorithm Design*, 2006, Pearson.