中山大學
SUN YAT-SEN UNIVERSITY

Lecture 4
Greedy Algorithms

Algorithm Design and Analysis

zhangzizhen@gmail.com

Huffman Coding

- A nice application of a greedy algorithm is found in an approach to data compression called Huffman coding.
- Suppose that we have a large amount of text that we wish to store on a computer disk in an efficient way. The simplest way to do this is simply to assign a binary code to each character, and then store the binary codes consecutively in the computer memory.
- The ASCII system for example, uses a fixed 8-bit code to represent each character. Storing n characters as ASCII text requires *8n* bits of memory.

Huffman Coding

- Let C be the set of characters we are working with. To simplify things, let us suppose that we are storing only the 10 numeric characters 0, 1, . . ., 9. That is, set C = {0, 1, … , 9}.
- A fixed length code to store these 10 characters would require at least 4 bits per character. For example we might use a code like this:

Char	Code
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

- However in any non-random piece of text, some characters occur far more frequently than others, and hence it is possible to save space by using a variable length code where the more frequently occurring characters are given shorter codes.

Huffman Coding

- Consider the following data, which is taken from a Postscript file.

Char	Freq
5	1294
9	1525
6	2260
4	2561
2	4442
3	5960
7	6878
8	8865
1	11610
0	70784

- Notice that there are many more occurrences of 0 and 1 than the other characters.

Huffman Coding

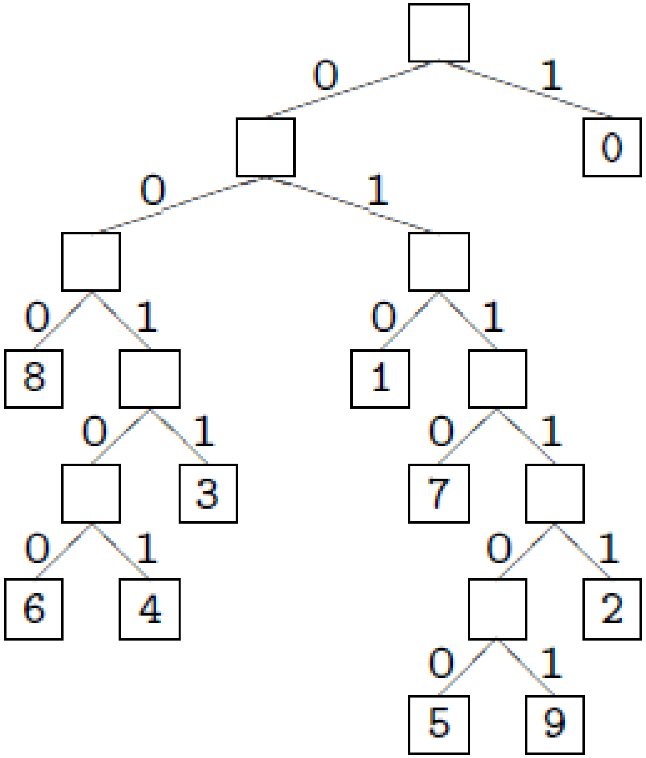
- What would happen if we used the following code to store the data rather than the fixed length code?

Char	Code
0	1
1	010
2	01111
3	0011
4	00101
5	011100
6	00100
7	0110
8	000
9	011101

- To store the string 0748901 we would get 0000011101001000100100000001 using the fixed length code and 10110001010000111011010 using the variable length code.

Huffman Coding

- In order to be able to decode the variable length code properly it is necessary that it be a prefix code — that is, a code in which no codeword is a prefix of any other codeword.
- Decoding such a code is done using a binary tree.



Huffman Coding

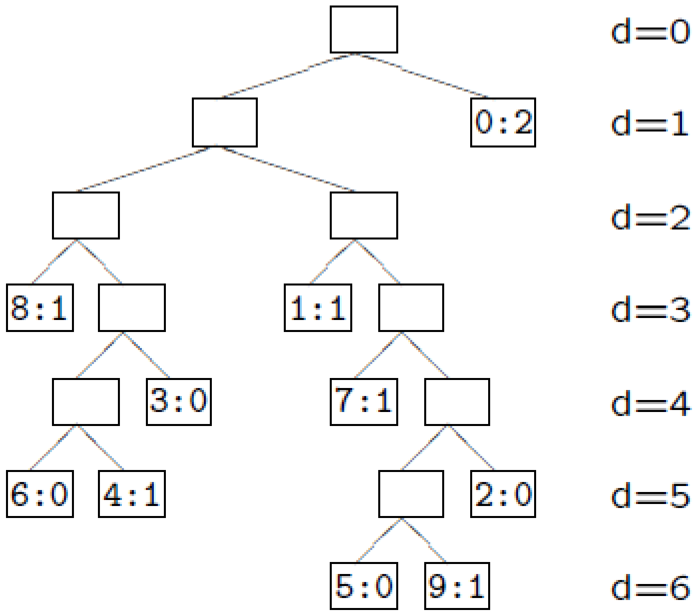
- Now assign to each leaf of the tree a value, *f(c)*, which is the frequency of occurrence of the character *c* represented by the leaf.
- Let $d_T(c)$ be the depth of character *c*'s leaf in the tree *T*.
- Then the number of bits required to encode a file is

$$B(T) = \sum_{c \in C} f(c) d_T(\mathrm{c})$$

- which we define as the cost of the tree T.

Huffman Coding

- For example, the number of bits required to store the string 0748901 can be computed from the tree *T*:



giving B(T) = $2\times1+1\times3+1\times3+1\times4+1\times5+1\times6 = 23$.

Thus, the cost of the tree *T* is 23.

Huffman Coding

- A tree representing an optimal code for a file is always a full binary tree (note, full v.s. complete, perfect) — namely, one where every node is either a leaf or has precisely two children.
- Therefore if we are dealing with an alphabet of s symbols we can be sure that our tree has precisely s leaves and $s-1$ internal nodes, each with two children.
- Huffman invented a greedy algorithm to construct such an optimal tree. The resulting code is called a Huffman code for that file.

Huffman Coding

- **Huffman's algorithm**. The algorithm starts by creating a forest of s single nodes, each representing one character, and each with an associated value, being the frequency of occurrence of that character. These values are placed into a priority queue (implemented as a linear array).

5:1294	9:1525	6:2260	4:2561	2:4442
3:5960	7:6878	8:8865	1:11610	0:70784

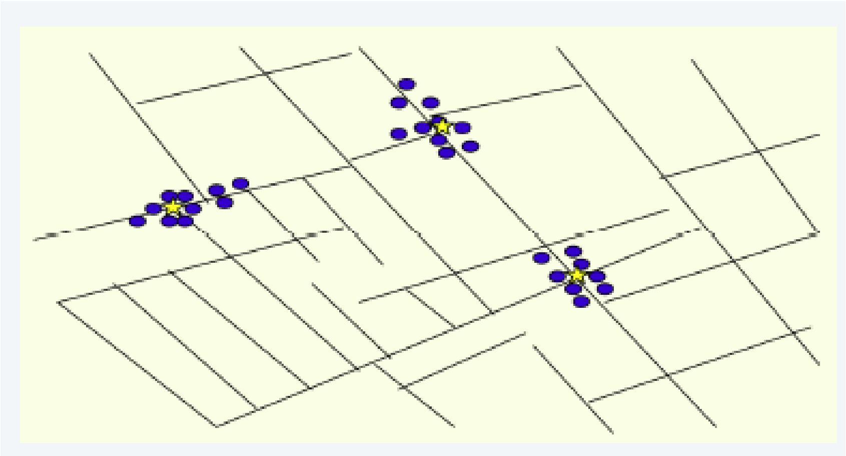
- Then repeat the following procedure $s - 1$ times:
- Remove from the priority queue the two nodes *L* and *R* with the lowest values, and create a internal node of the binary tree whose left child is *L* and right child *R*.
- Compute the value of the new node as the sum of the values of *L* and *R* and insert this into the priority queue.

Huffman Coding

- Huffman算法用最小堆实现(优先队列)。初始化优先队列需要O(n)计算时间，由于最小堆的removeMin和insert运算均需O(logn)时间，n－1次的合并总共需要O(nlogn)计算时间。因此，关于n个字符的哈夫曼算法的计算时间为O(nlogn)。

Single-link Clustering

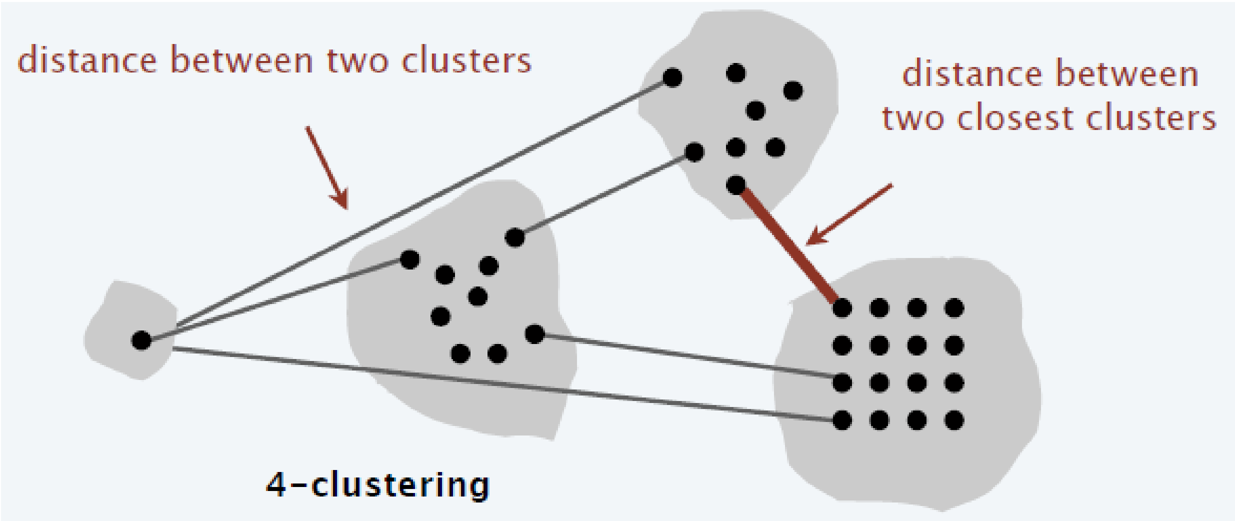
- Given a set U of n objects labeled $p_1, \ldots, p_n$, partition into clusters so that objects in different clusters are far apart.



- 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.
- ...

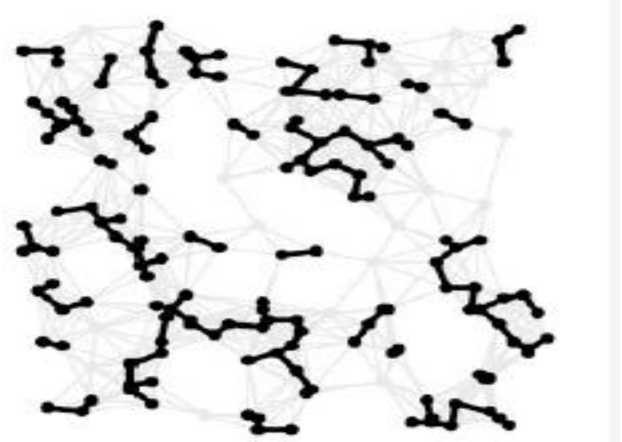
Single-link Clustering

- ***k*-clustering**: Divide objects into *k* non-empty groups.
- **Distance function**: Numeric value specifying "closeness" of two objects.
- **Spacing**: Minimum distance between any pair of points in different clusters.
- **Goal**: Given an integer *k*, find a *k*-clustering of maximum spacing.



Single-link Clustering

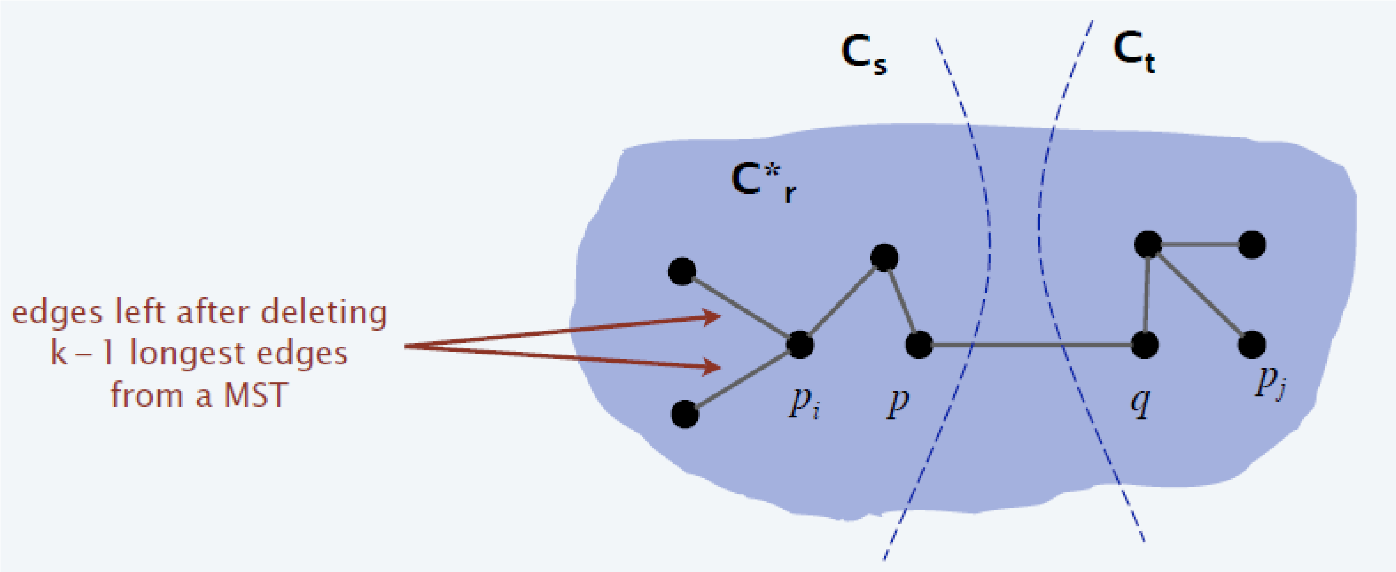
- "Well-known" algorithm in science literature for single-linkage k-clustering:
  - Form a graph on the node set U, corresponding to n clusters.
  - Find the closest pair of objects such that each object is in a different cluster, and add an edge between them.
  - Repeat $n - k$ times until there are exactly k clusters.



- This procedure is precisely Kruskal's algorithm.
- Alternative. Find an MST and delete the $k - 1$ longest edges.

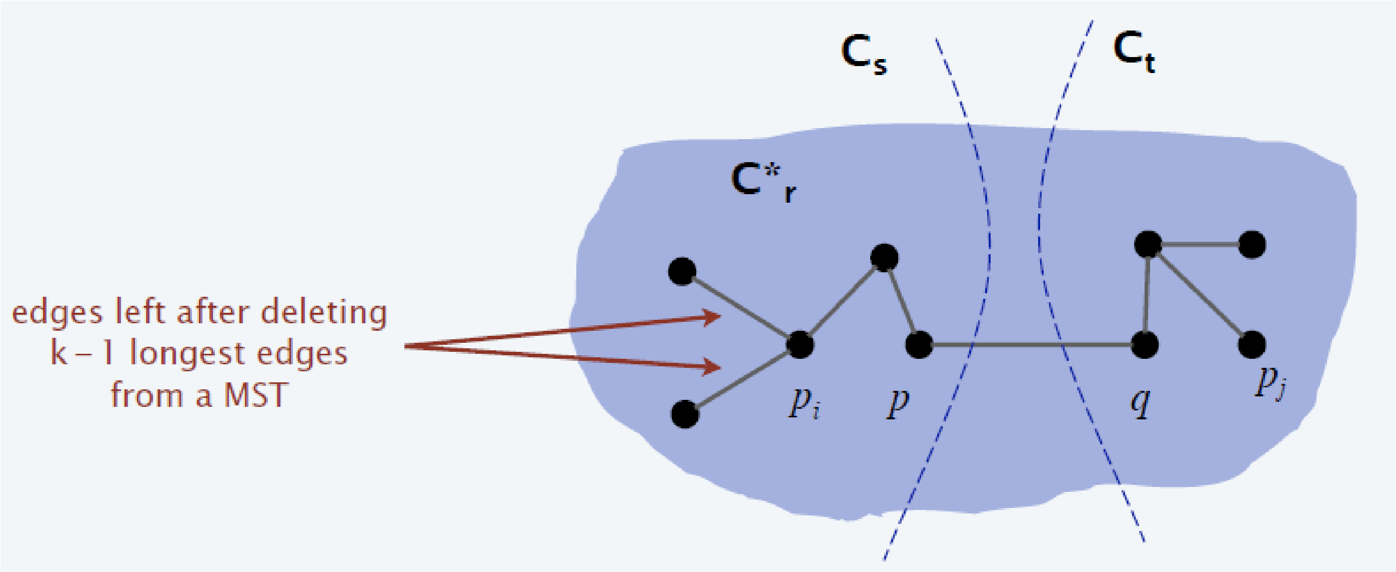
Single-link Clustering

- **Theorem**: Let C^* denote the clustering $C^*_1, \ldots, C^*_k$ formed by deleting the $k - 1$ longest edges of an MST. Then, C^* is a k-clustering of max spacing.
- **Proof**: Let C denote some other clustering $C_1, \ldots, C_k$.
  - The spacing of C^* is the length d^* of the $(k - 1)$-st longest edge in MST.



Single-link Clustering

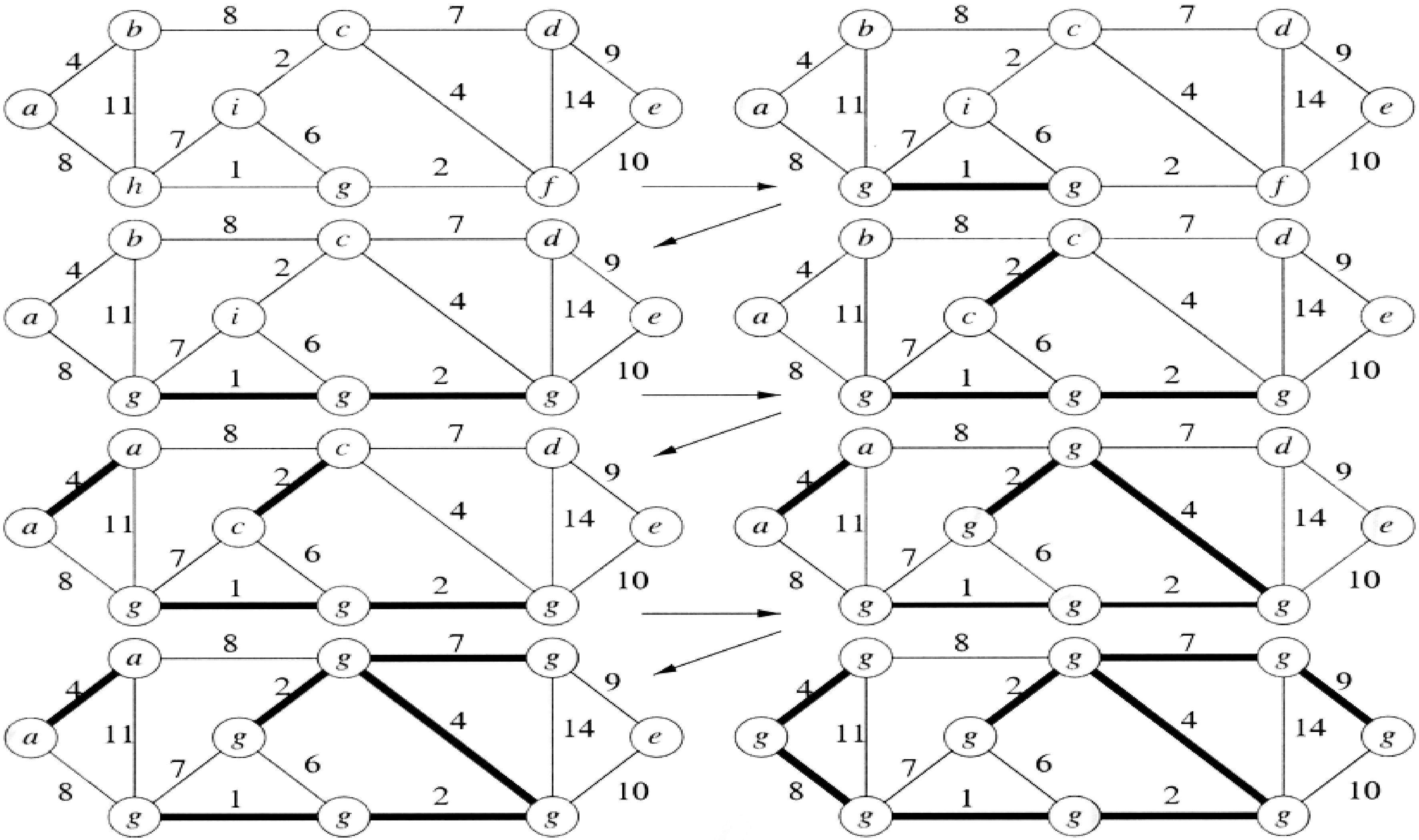
- Let p_i and p_j be in the same cluster in C^*, say C^*_r , but different clusters in C, say C_s and C_t.
- Some edge (p, q) on $p_i \sim p_j$ path in C^*_r spans two different clusters in C.
- Edge (p, q) *has length* $\leq d^*$ since it wasn't deleted.
- Spacing of C *is* $\leq d^*$ since p *and* q are in different clusters.



Kruskal's Algorithm for Minimum Spanning Trees

- Kruskal's Algorithm is directly based on the generic MST algorithm. It builds the MST in forest.
- Initially, each vertex is in its own tree in forest. Then, the algorithm considers each edge in turn, order by increasing weight.
- If an edge (u, v) connects two different trees, then (u, v) is added to the set of edges of the MST, and two trees connected by an edge (u, v) are merged into a single tree.
- On the other hand, if an edge (u, v) connects two vertices in the same tree, then edge (u, v) is discarded.

Kruskal's Algorithm for Minimum Spanning Trees



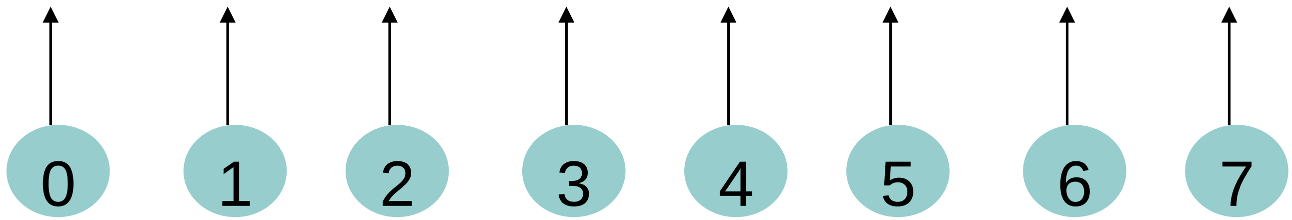
Implementation of Kruskal's Algorithm

1 $A \leftarrow \varnothing$ // initially A is empty
2 **for** each vertex $v \in V[G]$ // line 2-3 takes $O(V)$ time
3 do Make-Set(v) // create set for each vertex
4 sort the edges of E into non-decreasing order by weight w
5 **for** each edge $(u,v) \in E$, taken in non-decreasing order by weight **do**
6 **if** Find-Set(u) $\neq$ Find-Set(v) // u&v on different trees
7 **then** $A \leftarrow A \cup \{(u,v)\}$
8 Union(u,v)
9 return A

- **Make_Set**(*v*): Create a new set whose only member is pointed to by *v*. Note that for this operation *v* must already be in a set.
- **Find_Set**(*v*): Returns a pointer to the set containing *v*.
- **Union**(*u*, *v*): Unites the dynamic sets that contain *u* and *v* into a new set that is union of these two sets.

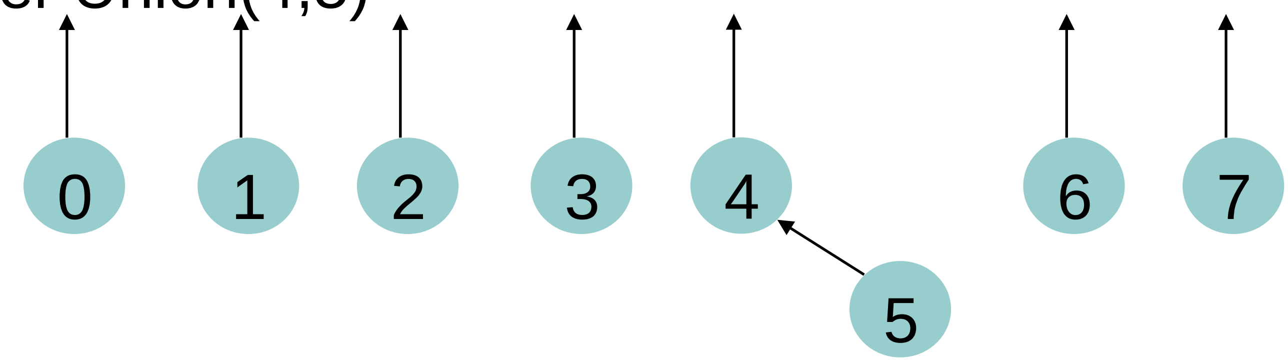
Disjoint-set Operations: Union

- Eight elements, initially in different sets



−1	−1	−1	−1	−1	−1	−1	−1
0	1	2	3	4	5	6	7

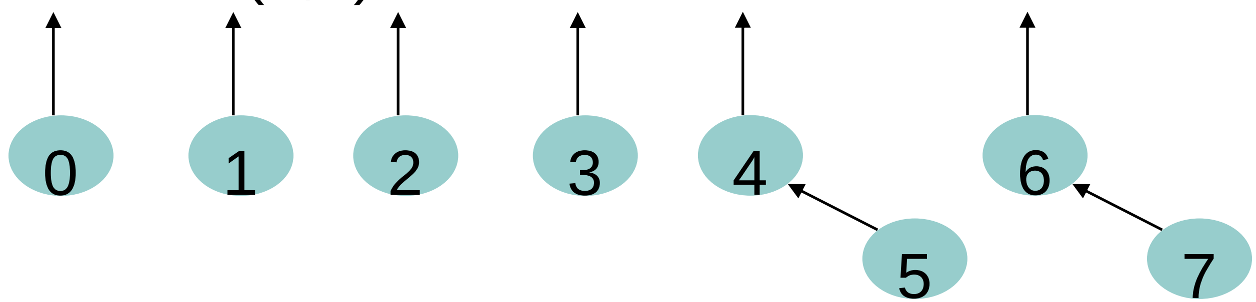
- After Union(4,5)



−1	−1	−1	−1	−1	4	−1	−1
0	1	2	3	4	5	6	7

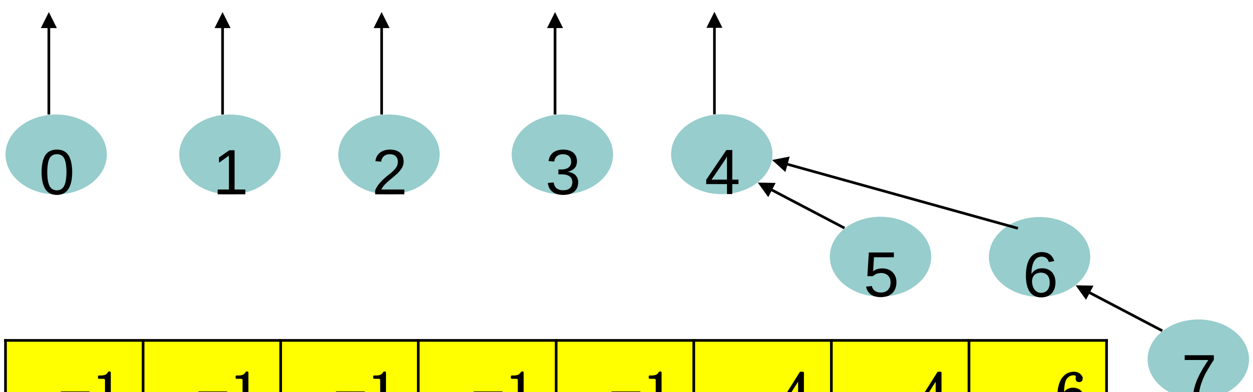
Disjoint-set Operations: Union

- After union(6,7)



−1	−1	−1	−1	−1	4	−1	6
0	1	2	3	4	5	6	7

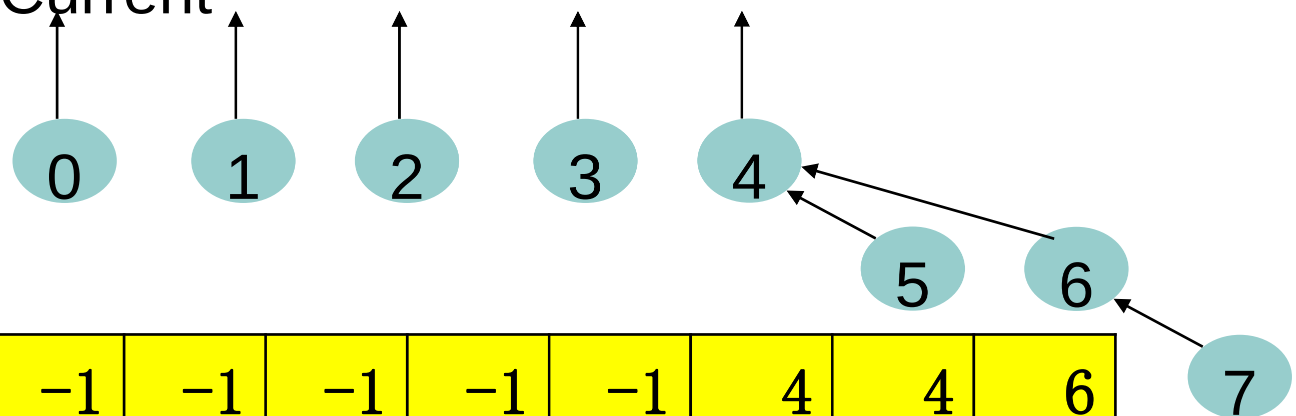
- After union(4,6)



−1	−1	−1	−1	−1	4	4	6
0	1	2	3	4	5	6	7

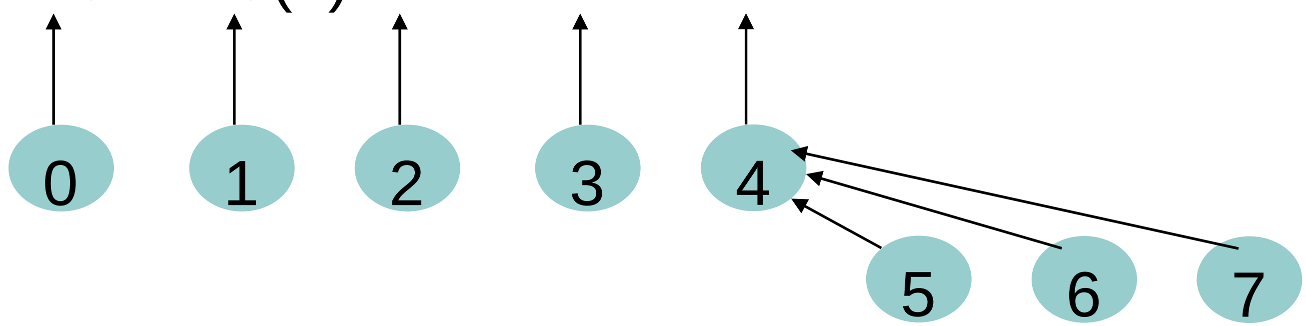
Disjoint-set Operations: Quick-Find

- Current



−1	−1	−1	−1	−1	4	4	6
0	1	2	3	4	5	6	7

- After Find(7)



−1	−1	−1	−1	−1	4	4	4
0	1	2	3	4	5	6	7

Implementation of Kruskal's Algorithm

```
int p[maxn],r[maxm];

int cmp(int i, int j) {return w[i]<w[j]};

int find(int x)
{return (p[x] == -1) ? x : p[x]=find(p[x]);}

int union(int x, int y) { p[y] = x;}

int Kruskal() {
  int ans = 0;
  for(int i=0;i<n;i++) p[i] = -1;
  for(int i=0;i<m;i++) r[i] = i;
  sort(r,r+m,cmp);
  for(int i=0;i<m;i++) {
    int e = r[i], x = find(u[e]), y = find(v[e]);
    if (x!=y) {ans += w[e]; union(x,y);}
  }
  return ans;
}
```

Time Complexity

- MAKE-SETs requires $O(V)$
- Edge sorting requires $O(E \log E)$
- FIND-SETs and UNIONs perform $O(E)$ time. Assuming the implementation of disjoint-set data structure that uses union by rank and path compression, so amortized in $O(E\, \alpha(V))$. (α为反ackerman函数)
- $\alpha(V) = O(\log V) = O(\log E)$
- Total Time: $O(E \log E) = O(E \log V)$
- If edges are already sorted, $O(E\, \alpha(V))$, which is almost linear.

Thank you!