



# LECTURE 07

## GREEDY ALGORITHM

Spring 2023  
Zhihua Jiang

# Content

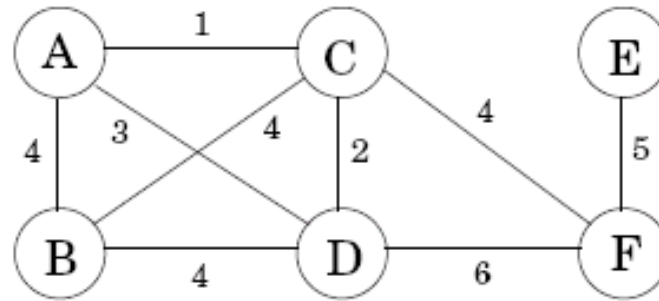
- Applications of greedy algorithms
  - Minimum spanning trees
  - Huffman encoding
  - Set cover
  - .....
- Optimality proof
  - Optimal substructure (最优子结构)+Greedy choice (贪心选择)
  - Matroid (拟阵)

# Characteristics of greedy algorithm

- ◆ Efficient
- ◆ Easy to implement
- ◆ Not guarantee an optimal solution
- ◆ Often used as approximated algorithm

# Minimum spanning trees

- Introductory example:
  - how to find the cheapest network?



- **Property 1** Removing a cycle edge cannot disconnect a graph.
- The solution:
  - Tree (connected and acyclic)
  - Minimum spanning tree (with minimum total weight)

# Minimum spanning trees

## ○ Trees

- **Property 1** A tree on  $n$  nodes has  $n-1$  edges.
- **Property 2** Any connected, undirected graph  $G=(V,E)$  with  $|E|=|V|-1$  is a tree.
- **Property 3** An undirected graph is a tree if and only if there is a unique path between any pair of nodes

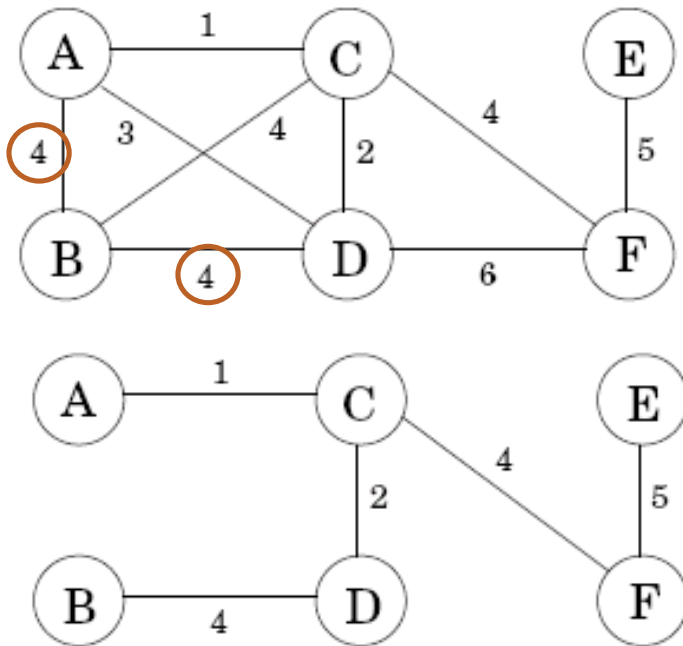
# Minimum spanning trees

- Minimum spanning tree (MST)

Input: An undirected graph  $G=(V,E)$ ; edge weights  $W_e$ .

Output: A tree  $T=(V,E')$ , with  $E' \subseteq E$ , that minimizes

$$\text{weight}(T) = \sum_{e \in E'} W_e$$

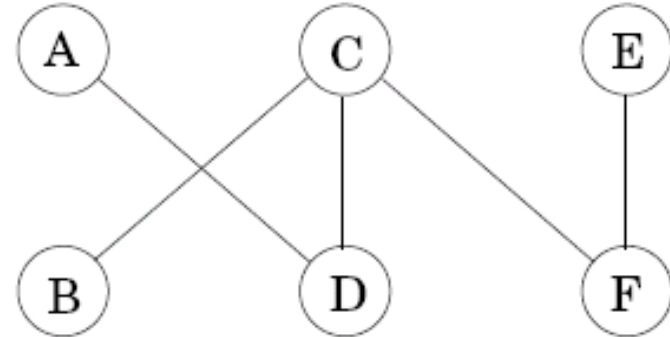
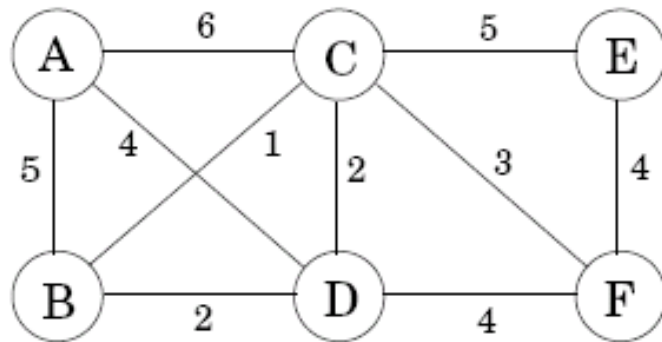


The MST is not unique!

# A greedy approach

## ○ Kruskal's MST algorithm:

- Start with the empty graph
- Repeatedly *add the next lightest edge* that doesn't produce a cycle



## ○ Example: Add edges in increasing order of weight

$B - C$ ,  $C - D$ ,  $A \times D$ ,  $C - F$ ,  $D \times F$ ,  $E - F$ ,  $A - D$ ,  $A \times B$ ,  $C \times E$ ,  $A \times C$ .

# Kruskal's algorithm

20

procedure kruskal( $G, w$ )

Input: A connected undirected graph  $G = (V, E)$  with edge weights  $w_e$

Output: A minimum spanning tree defined by the edges  $X$

for all  $u \in V$ :  
    makeset( $u$ )

Makeset( $x$ ): create a singleton set containing just  $x$   
Find( $x$ ): to which set does  $x$  belong?  
Union( $x, y$ ): merge the sets containing  $x$  and  $y$   
Set: connected component

$X = \{\}$

Sort the edges  $E$  by weight

for all edges  $\{u, v\} \in E$ , in increasing order of weight:

    if find( $u$ )  $\neq$  find( $v$ ):

        add edge  $\{u, v\}$  to  $X$

        union( $u, v$ )

✓ Total work:

$|V|$  makeset,  $2|E|$  find,  $|V|-1$  union

✓ The total time:

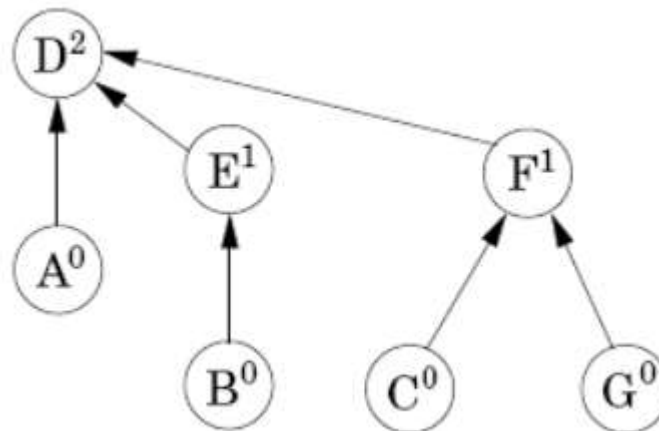
$O((|V|+|E|)\log|V|)$ ,

    if implemented with disjoint sets



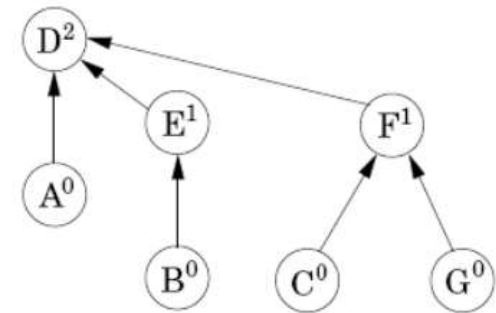
# Disjoint set

- Why use disjoint sets?
  - Contain the nodes of a particular component
- How to store disjoint sets?
  - A directed tree
  - Nodes of the tree are elements of the set
  - Each node has a parent pointer that eventually leads up to the root of the tree



# Disjoint set

- Each node has a *parent pointer*  $\pi$  and a *rank* (i.e., the height of the subtree rooted at that node)
- Three operations on disjoint sets:
  - Makeset( $x$ )
    - Make a singleton tree
    - $O(1)$
  - Find( $x$ )
    - Follow parent pointer to the root of the tree
    - $O(\log n)$
  - Union( $x, y$ )
    - Make the root of the shorter tree point to the root of the taller tree
    - The overall height increases only if equally tall
    - $O(\log n)$



procedure makeset( $x$ )

$\pi(x) = x$

$\text{rank}(x) = 0$

function find( $x$ )

while  $x \neq \pi(x)$ :  $x = \pi(x)$

return  $x$

procedure union( $x, y$ )

$r_x = \text{find}(x)$

$r_y = \text{find}(y)$

if  $r_x = r_y$ : return

if  $\text{rank}(r_x) > \text{rank}(r_y)$ :

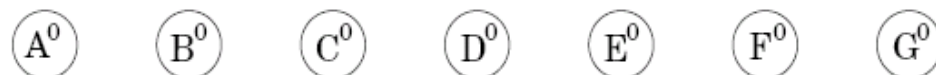
$\pi(r_y) = r_x$

else:

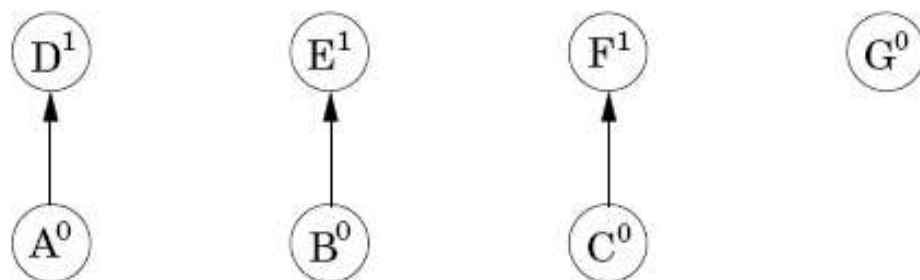
$\pi(r_x) = r_y$

if  $\text{rank}(r_x) = \text{rank}(r_y)$ :  $\text{rank}(r_y) = \text{rank}(r_y) + 1$

After makeset( $A$ ), makeset( $B$ ), ..., makeset( $G$ ):



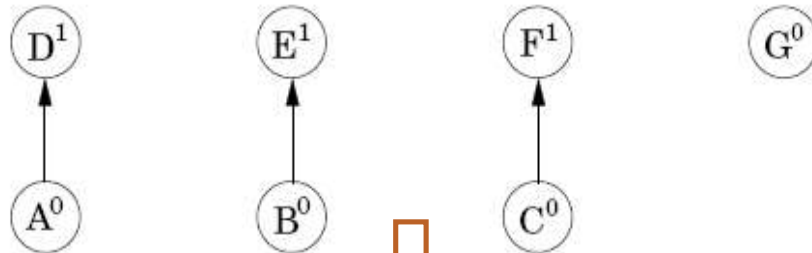
After union( $A, D$ ), union( $B, E$ ), union( $C, F$ ):



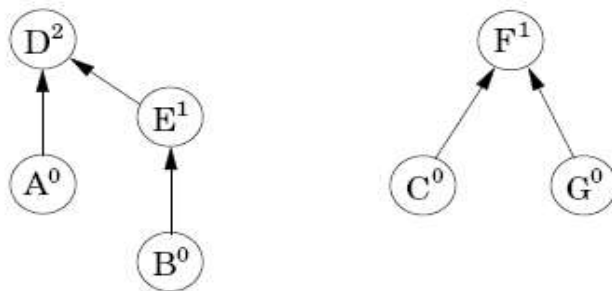
After  $\text{makeset}(A), \text{makeset}(B), \dots, \text{makeset}(G)$ :



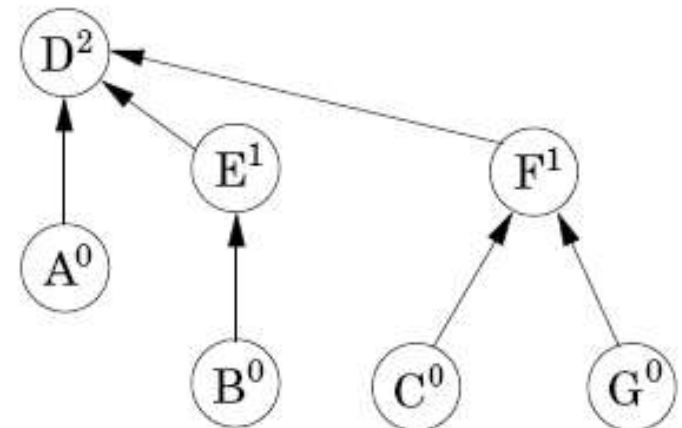
After  $\text{union}(A, D), \text{union}(B, E), \text{union}(C, F)$ :



After  $\text{union}(C, G), \text{union}(E, A)$ :



After  $\text{union}(B, G)$ :

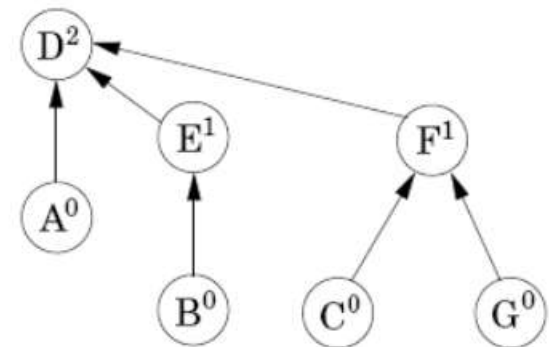


# Disjoint set

- **Property 1** For any  $x \neq \pi(x)$ ,  $\text{rank}(x) < \text{rank}(\pi(x))$ .
- **Property 2** Any root node of rank  $k$  has *at least*  $2^k$  nodes in its tree.
- **Property 3** If there are  $n$  elements overall, there can be *at most*  $n/2^k$  nodes of rank  $k$ . (All the trees have height  $\leq \log n$ )

Hints:

- Property 2 (at least):
  - merge two trees with height  $k$ .
- Property 3 (at most):
  - $k=0$ : forest of  $n$  singleton trees with height 0.
  - $k=1$ :  $n/2$  single-child trees with height 1.
  - assume when rank =  $k$ , the property holds
  - how to produce the most nodes at rank  $k+1$ ? Merge equal-height trees as many as possible so that #nodes of rank  $k+1$  is at most  $(n/2^k)/2$ .

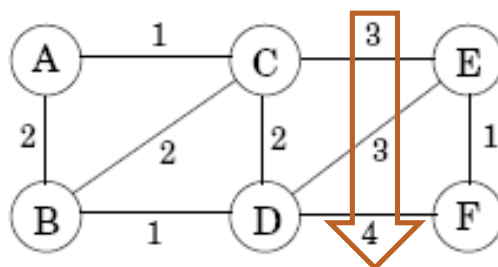


# The cut property

- **Cut**: any partition of the vertices into two groups,  $S$  and  $V-S$ .
- **Cut property**: Suppose edges  $X$  are part of a minimum spanning tree of  $G=(V,E)$ . Pick any subset of nodes  $S$  for which  $X$  does not cross between  $S$  and  $V-S$ , and let  $e$  be the lightest edge across this partition. Then  $X \cup \{e\}$  is a part of some *MST*.
- The correctness of Kruskal's method: it is always safe to add the lightest edge across any cut, provided  $X$  has no edges across the cut.

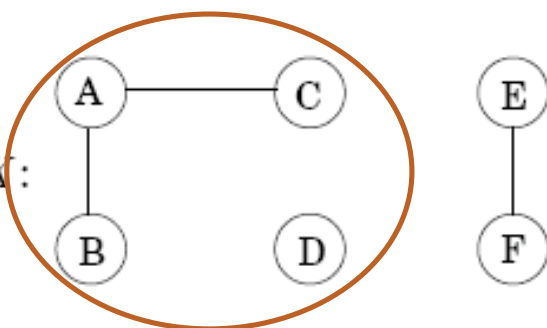
**Figure 5.3** The cut property at work. (a) An undirected graph. (b) Set  $X$  has three edges, and is part of the MST  $T$  on the right. (c) If  $S = \{A, B, C, D\}$ , then one of the minimum-weight edges across the cut  $(S, V - S)$  is  $e = \{D, E\}$ .  $X \cup \{e\}$  is part of MST  $T'$ , shown on the right.

(a)

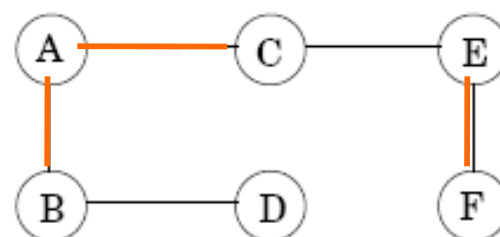


(b)

Edges  $X$ :

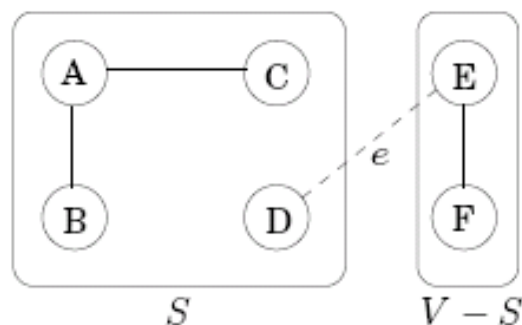


MST  $T$ :

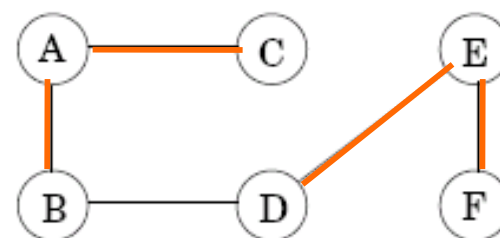


(c)

The cut:



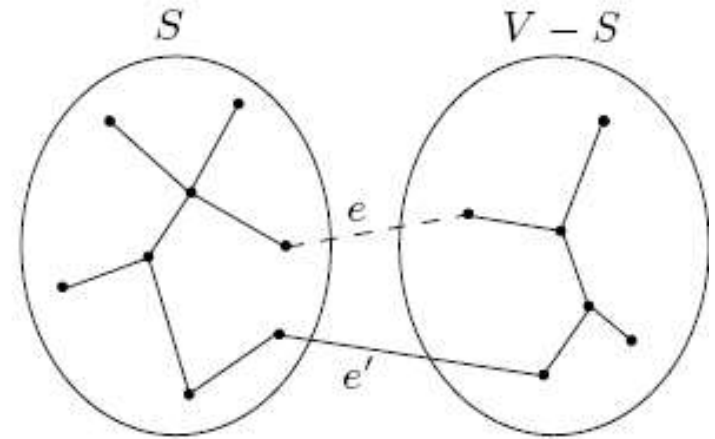
MST  $T'$ :



# The cut property

## ○ *Proof of the cut property*

- Assume  $X$  is part of some MST  $T$ , and  $e$  is not in  $T$
- **Construct a different MST  $T'$**  containing  $X \cup \{e\}$ 
  - Adding  $e$  to  $T$  will create a cycle
  - $T' = T \cup \{e\} - \{e'\}$
  - $T'$  has the same number of edges as  $T$ , so  $T'$  is a tree
  - $\text{weight}(T') = \text{weight}(T) + w(e) - w(e')$ , since  $w(e) \leq w(e')$ ,  $\text{weight}(T') \leq \text{weight}(T)$ .  $T'$  is also a MST.





# Prim's algorithm

- The following greedy schema always works for finding MST:

$X = \{ \}$  (edges picked so far)

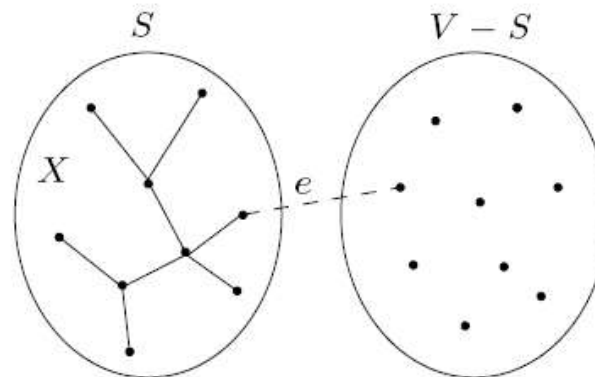
repeat until  $|X| = |V| - 1$ :

    pick a set  $S \subset V$  for which  $X$  has no edges between  $S$  and  $V - S$

    let  $e \in E$  be the minimum-weight edge between  $S$  and  $V - S$

$X = X \cup \{e\}$

- Prim's algorithm:  $X$  always forms a subtree, and  $S$  is the set of this tree's vertices



procedure prim( $G, w$ )

Input: A connected undirected graph  $G = (V, E)$  with edge weights  $w_e$

Output: A minimum spanning tree defined by the array prev

for all  $u \in V$ :

$\text{cost}(u) = \infty$

$\text{prev}(u) = \text{nil}$

Pick any initial node  $u_0$

$\text{cost}(u_0) = 0$

$$\text{cost}(v) = \min_{u \in S} w(u, v).$$

$H = \text{makequeue}(V)$  (priority queue, using cost-values as keys)

while  $H$  is not empty:

$v = \text{deletemin}(H)$

    for each  $\{v, z\} \in E$ :

        if  $\text{cost}(z) > w(v, z)$ :

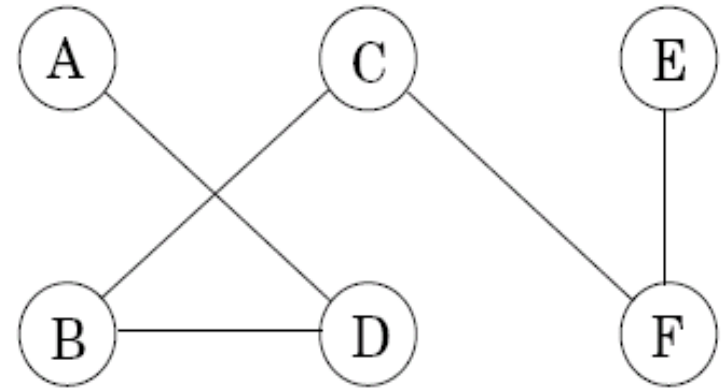
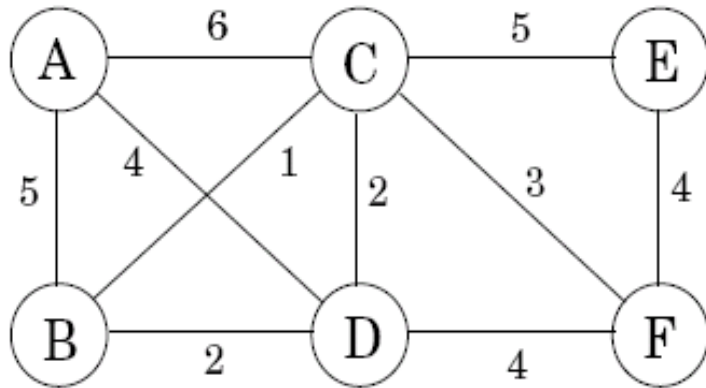
$\text{cost}(z) = w(v, z)$

$\text{prev}(z) = v$

$\text{decreasekey}(H, z)$

- Similar to Dijkstra's algorithm, except that the key value is the weight of the lightest outgoing edge from set  $S$ , while in Dijkstra's it is the length of an entire path from start node to the node
- The time complexity:  $O((|V|+|E|)\log|V|)$

## 5.1.5 Prim's algorithm

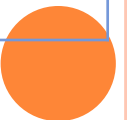


Set $S$	$A$	$B$	$C$	$D$	$E$	$F$
$\{ \}$	0/nil	$\infty$ /nil	$\infty$ /nil	$\infty$ /nil	$\infty$ /nil	$\infty$ /nil
$A$		5/ $A$	6/ $A$	4/ $A$	$\infty$ /nil	$\infty$ /nil
$A, D$		2/ $D$	2/ $D$		$\infty$ /nil	4/ $D$
$A, D, B$			1/ $B$		$\infty$ /nil	4/ $D$
$A, D, B, C$					5/ $C$	3/ $C$
$A, D, B, C, F$					4/ $F$	

# Comparison

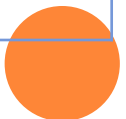
	<b>Kruskal</b>	<b>prim</b>
<b>sort all edges?</b>		
<b>mimimum</b>		
<b>data structure</b>		
<b>time complexity</b>		

2023/4/10



# Comparison

	<b>Kruskal</b>	<b>prim</b>
<b>sort all edges?</b>	<b>yes</b>	<b>no</b>
<b>mimimum</b>	<b>the lightest edge in the remaining edges</b>	<b>the lightest edge among the cross edges</b>
<b>data structure</b>	<b>disjoint set (connected component)</b>	<b>binary heap (priority queue)</b>
<b>time complexity</b>	$O(( V  +  E )\log  V )$	$O(( V  +  E )\log  V )$



# Huffman encoding

- The introductory example:

A long string of length  $T = 130$  million, an alphabet  $\Gamma$  consists of just four symbols  $A, B, C, D$ , how to encode in the most economical way?

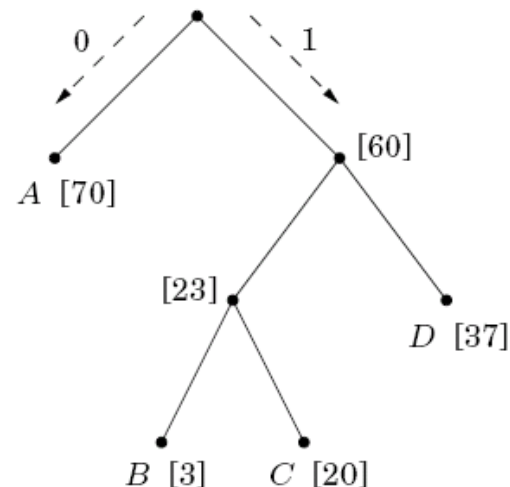
- Uniform-length encoding
  - 00 for  $A$ , 01 for  $B$ , 10 for  $C$ , 11 for  $D$
  - 260 megabits are needed in total
- Variable-length encoding
  - One bit for the frequently occurring symbol, while two or more bits for less common symbols
  - Cause ambiguity
  - *Prefix-free property*: no codeword can be a prefix of another codeword

# Huffman encoding

## ○ Prefix-free encoding:

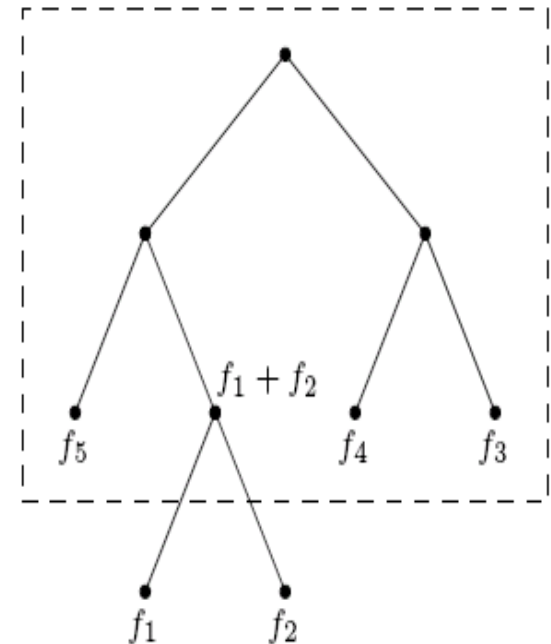
- Represented by a full binary tree
- Symbols are at the leaves
- Codeword: a path from root to leaf, interpreting left as 0 and right as 1
- For the introductory example, drop to 213 megabits, a  $(260-213)/260 = 18\%$  improvement

Symbol	Codeword
<i>A</i>	0
<i>B</i>	100
<i>C</i>	101
<i>D</i>	11



# Huffman encoding

- How to find the optimal coding tree, given the frequencies  $f_1, f_2, \dots, f_n$  of  $n$  symbols?
  - Define the frequency of any internal node to be the sum of the frequencies of its descendant leaves
  - Construct the tree **greedily**:
    - Find the two symbols with the smallest frequencies  $f_1$  and  $f_2$
    - Pull  $f_1$  and  $f_2$  off the list of frequencies, insert  $(f_1 + f_2)$ , and loop





# Huffman encoding

procedure Huffman( $f$ )

Input: An array  $f[1 \cdots n]$  of frequencies

Output: An encoding tree with  $n$  leaves

let  $H$  be a priority queue of integers, ordered by  $f$

for  $i=1$  to  $n$ : insert( $H, i$ )

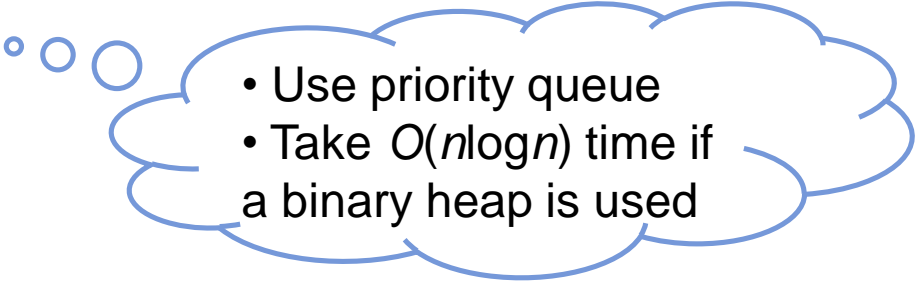
for  $k=n+1$  to  $2n-1$ :

$i = \text{deletemin}(H)$ ,  $j = \text{deletemin}(H)$

    create a node numbered  $k$  with children  $i, j$

$f[k] = f[i] + f[j]$

    insert( $H, k$ )

- 
- Use priority queue
  - Take  $O(n \log n)$  time if a binary heap is used

# Set cover

- The set cover problem:

*Input:* A set of elements  $B$ ; sets  $S_1, \dots, S_m \subseteq B$

*Output:* A selection of the  $S_i$  whose union is  $B$ .

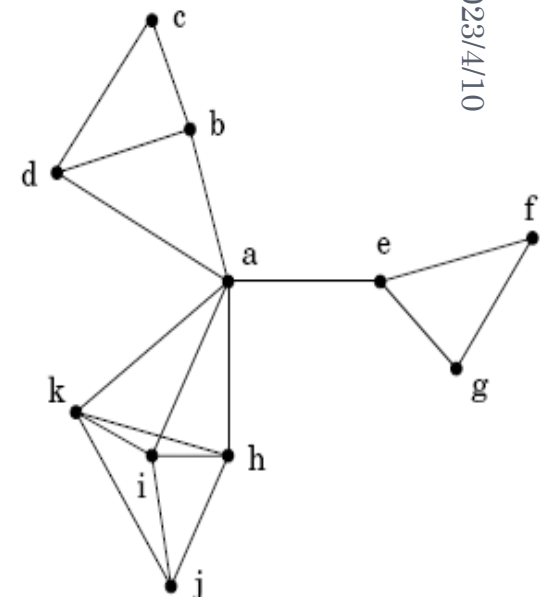
*Cost:* Number of sets picked.

- A greedy algorithm:

Repeat until all elements of  $B$  are covered:

Pick the set  $S_i$  with the largest number of uncovered elements.

- A greedy solution  $\{S_a, S_c, S_j, S_f\}$  where  $S_a = \{v | (v, a) \in E\} \cup \{a\}$
- An optimal solution  $\{S_b, S_e, S_i\}$



## Set cover

- **Claim:** Suppose  $B$  contains  $n$  elements and that the optimal cover consists of  $k$  sets. Then the greedy algorithm will use at most  $k \ln n$  sets.
- Approximation factor: the ratio between the optimal value  $c$  of the approximation solution and that  $c^*$  of the optimal solution.

$$\Delta = \max \left\{ \left| \frac{c}{c^*} \right|, \left| \frac{c^*}{c} \right| \right\}$$

- The closer the approximation factor is to one, the better the approximation algorithm performs.

Claim: Suppose  $B$  contains  $n$  elements and that the optimal cover consists of  $k$  sets. Then the greedy algorithm will use at most  $k \ln n$  sets.

### ○ *Proof*

Let  $n_t$  be the number of elements still not covered after  $t$  iterations of the greedy algorithm (so  $n_0 = n$ ). Since these remaining elements are covered by the optimal  $k$  sets, there must be some set with at least  $n_t/k$  of them. Therefore, the greedy strategy will ensure that

$$n_{t+1} \leq n_t - \frac{n_t}{k} = n_t \left(1 - \frac{1}{k}\right),$$

which by repeated application implies  $n_t \leq n_0(1 - 1/k)^t$ . A more convenient bound can be obtained from the useful inequality

$$1 - x \leq e^{-x} \text{ for all } x, \text{ with equality if and only if } x = 0,$$

Thus

$$n_t \leq n_0 \left(1 - \frac{1}{k}\right)^t < n_0 (e^{-1/k})^t = n e^{-t/k}.$$

At  $t = k \ln n$ , therefore,  $n_t$  is strictly less than  $n e^{-\ln n} = 1$ , which means no elements remain to be covered.

each greedy choice covered at least  $n_t / k$  :

*E.g.*,  $greedy = \{S_a, S_f, S_c, S_j\}, optimal = \{S_b, S_e, S_i\}, k = 3$

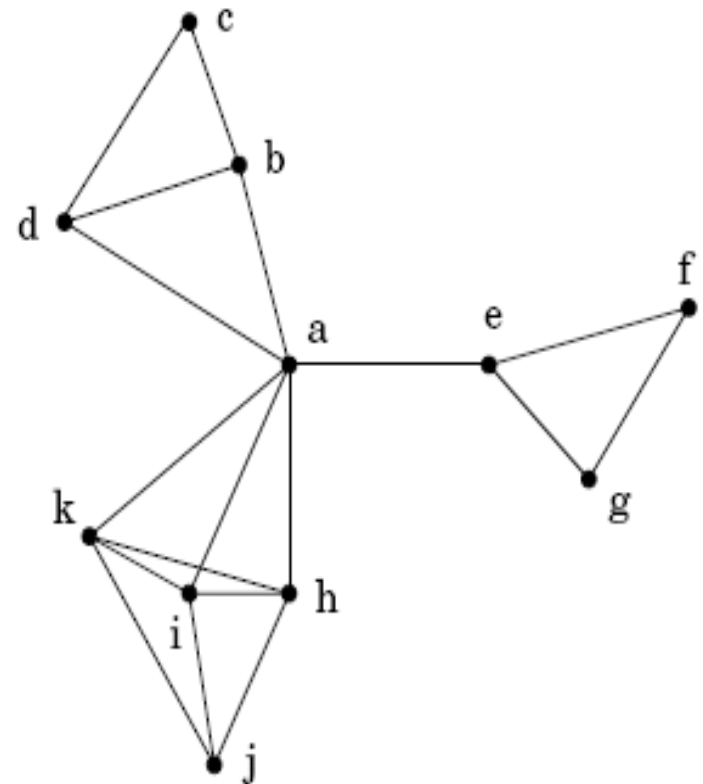
$n_0 = 11$ , (*greedy* choice:  $S_a$ )  $\Rightarrow n_0 / 3 = 11 / 3 = 3.7 < |S_a| = 7$

$n_1 = 4$ , (*greedy* choice:  $S_f$ )  $\Rightarrow n_1 / 3 = 4 / 3 = 1.3 < |S_f| = 2$

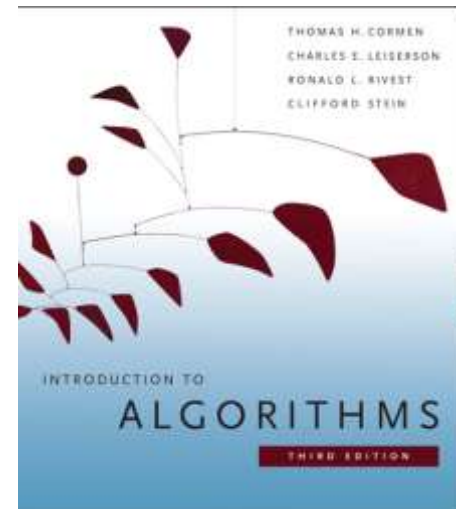
$n_2 = 2$ , (*greedy* choice:  $S_c$ )  $\Rightarrow n_2 / 3 = 2 / 3 = 0.7 < |S_c| = 1$

$n_3 = 1$ , (*greedy* choice:  $S_j$ )  $\Rightarrow n_3 / 3 = 1 / 3 = 0.3 < |S_j| = 1$

$n_4 = 0$ , (*solved*)



# Matroid theory: See CLRS book (Chapter 16.4)



2023/4/10

A **matroid** is an ordered pair  $M = (S, \mathcal{I})$  satisfying the following conditions.

1.  $S$  is a finite set.
2.  $\mathcal{I}$  is a nonempty family of subsets of  $S$ , called the *independent* subsets of  $S$ , such that if  $B \in \mathcal{I}$  and  $A \subseteq B$ , then  $A \in \mathcal{I}$ . We say that  $\mathcal{I}$  is *hereditary* if it satisfies this property. Note that the empty set  $\emptyset$  is necessarily a member of  $\mathcal{I}$ .
3. If  $A \in \mathcal{I}$ ,  $B \in \mathcal{I}$ , and  $|A| < |B|$ , then there exists some element  $x \in B - A$  such that  $A \cup \{x\} \in \mathcal{I}$ . We say that  $M$  satisfies the *exchange property*.



As another example of matroids, consider the **graphic matroid**  $M_G = (S_G, \mathcal{I}_G)$  defined in terms of a given undirected graph  $G = (V, E)$  as follows:

- The set  $S_G$  is defined to be  $E$ , the set of edges of  $G$ .
- If  $A$  is a subset of  $E$ , then  $A \in \mathcal{I}_G$  if and only if  $A$  is acyclic. That is, a set of edges  $A$  is independent if and only if the subgraph  $G_A = (V, A)$  forms a forest.

Given a matroid  $M = (S, \mathcal{I})$ , we call an element  $x \notin A$  an extension of  $A \in \mathcal{I}$  if we can add  $x$  to  $A$  while preserving independence; that is,  $x$  is an extension of  $A$  if  $A \cup \{x\} \in \mathcal{I}$ . As an example, consider a graphic matroid  $M_G$ . If  $A$  is an independent set of edges, then edge  $e$  is an extension of  $A$  if and only if  $e$  is not in  $A$  and the addition of  $e$  to  $A$  does not create a cycle.

If  $A$  is an independent subset in a matroid  $M$ , we say that  $A$  is maximal if it has no extensions. That is,  $A$  is maximal if it is not contained in any larger independent subset of  $M$ . The following property is often useful.

**Theorem 16.6**

All maximal independent subsets in a matroid have the same size.

**Proof** Suppose to the contrary that  $A$  is a maximal independent subset of  $M$  and there exists another larger maximal independent subset  $B$  of  $M$ . Then, the exchange property implies that for some  $x \in B - A$ , we can extend  $A$  to a larger independent set  $A \cup \{x\}$ , contradicting the assumption that  $A$  is maximal. ■

We say that a matroid  $M = (S, \mathcal{I})$  is **weighted** if it is associated with a weight function  $w$  that assigns a strictly positive weight  $w(x)$  to each element  $x \in S$ . The weight function  $w$  extends to subsets of  $S$  by summation:

$$w(A) = \sum_{x \in A} w(x)$$



Many problems for which a greedy approach provides optimal solutions can be formulated in terms of finding a maximum-weight independent subset in a weighted matroid. That is, we are given a weighted matroid  $M = (S, \mathcal{I})$ , and we wish to find an independent set  $A \in \mathcal{I}$  such that  $w(A)$  is maximized. We call such a subset that is independent and has maximum possible weight an *optimal* subset of the matroid. Because the weight  $w(x)$  of any element  $x \in S$  is positive, an optimal subset is always a maximal independent subset—it always helps to make  $A$  as large as possible.

GREEDY( $M, w$ )

```
1   $A = \emptyset$ 
2  sort  $M.S$  into monotonically decreasing order by weight  $w$ 
3  for each  $x \in M.S$ , taken in monotonically decreasing order by weight  $w(x)$ 
4      if  $A \cup \{x\} \in M.\mathcal{I}$ 
5           $A = A \cup \{x\}$ 
6  return  $A$ 
```

The running time of GREEDY is easy to analyze. Let  $n$  denote  $|S|$ . The sorting phase of GREEDY takes time  $O(n \lg n)$ . Line 4 executes exactly  $n$  times, once for each element of  $S$ . Each execution of line 4 requires a check on whether or not the set  $A \cup \{x\}$  is independent. If each such check takes time  $O(f(n))$ , the entire algorithm runs in time  $O(n \lg n + nf(n))$ .

**Lemma 16.7 (Matroids exhibit the greedy-choice property)**

Suppose that  $M = (S, \mathcal{I})$  is a weighted matroid with weight function  $w$  and that  $S$  is sorted into monotonically decreasing order by weight. Let  $x$  be the first element of  $S$  such that  $\{x\}$  is independent, if any such  $x$  exists. If  $x$  exists, then there exists an optimal subset  $A$  of  $S$  that contains  $x$ .

**Lemma 16.10 (Matroids exhibit the optimal-substructure property)**

Let  $x$  be the first element of  $S$  chosen by GREEDY for the weighted matroid  $M = (S, \mathcal{I})$ . The remaining problem of finding a maximum-weight independent subset containing  $x$  reduces to finding a maximum-weight independent subset of the weighted matroid  $M' = (S', \mathcal{I}')$ , where

$$\begin{aligned} S' &= \{y \in S : \{x, y\} \in \mathcal{I}\} , \\ \mathcal{I}' &= \{B \subseteq S - \{x\} : B \cup \{x\} \in \mathcal{I}\} , \end{aligned}$$

and the weight function for  $M'$  is the weight function for  $M$ , restricted to  $S'$ . (We call  $M'$  the *contraction* of  $M$  by the element  $x$ .)

**Theorem 16.11 (Correctness of the greedy algorithm on matroids)**

If  $M = (S, \mathcal{I})$  is a weighted matroid with weight function  $w$ , then  $\text{GREEDY}(M, w)$  returns an optimal subset.

## Ex. 1

Design a linear-time algorithm for the following task.

*Input:* A connected, undirected graph  $G$ .

*Question:* Is there an edge you can remove from  $G$  while still leaving  $G$  connected?

Can you reduce the running time of your algorithm to  $O(|V|)$ ?

Since the graph is given to be connected, it will have an edge whose removal still leaves it connected, if and only if it is not a tree i.e. has more than  $|V| - 1$  edges. We perform a DFS on the graph until we see  $|V|$  edges. If we can find  $|V|$  edges then the answer is “yes” else it is “no”. In either case, the time taken is  $O(|V|)$ .



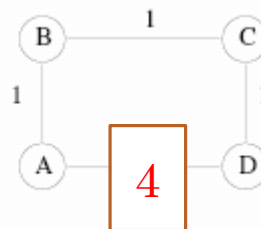
## Ex. 2

Consider an undirected graph  $G = (V, E)$  with nonnegative edge weights  $w_e \geq 0$ . Suppose that you have computed a minimum spanning tree of  $G$ , and that you have also computed shortest paths to all nodes from a particular node  $s \in V$ .

Now suppose each edge weight is increased by 1: the new weights are  $w'_e = w_e + 1$ .

- (a) Does the minimum spanning tree change? Give an example where it changes or prove it cannot change.
- (b) Do the shortest paths change? Give an example where they change or prove they cannot change.

- (a) The minimum spanning tree does not change. Since, each spanning tree contains exactly  $n - 1$  edges, the cost of each tree is increased  $n - 1$  and hence the minimum is unchanged.
- (b) The shortest paths may change. In the following graph, the shortest path from  $A$  to  $D$  changes from  $AB - BC - CD$  to  $AD$  if each edge weight is increased by 1.



## Ex. 3

Show how to find the *maximum* spanning tree of a graph, that is, the spanning tree of largest total weight.

Multiply the weights of all the edges by -1. Since both Kruskal's and Prim's algorithms work for positive as well as negative weights, we can find the minimum spanning tree of the new graph. This is the same as the maximum spanning tree of the original graph.

## Ex. 4

Suppose the symbols  $a, b, c, d, e$  occur with frequencies  $1/2, 1/4, 1/8, 1/16, 1/16$ , respectively.

- (a) What is the Huffman encoding of the alphabet?
- (b) If this encoding is applied to a file consisting of 1,000,000 characters with the given frequencies, what is the length of the encoded file in bits?

(a)  $a \rightarrow 0, b \rightarrow 10, c \rightarrow 110, d \rightarrow 1110, e \rightarrow 1111$ .

(b)  $\text{length} = \frac{1000000}{2} \cdot 1 + \frac{1000000}{4} \cdot 2 + \frac{1000000}{8} \cdot 3 + 2 \cdot \frac{1000000}{16} \cdot 4 = 1875000$

## Ex. 5

We use Huffman's algorithm to obtain an encoding of alphabet  $\{a, b, c\}$  with frequencies  $f_a, f_b, f_c$ . In each of the following cases, either give an example of frequencies  $(f_a, f_b, f_c)$  that would yield the specified code, or explain why the code cannot possibly be obtained (no matter what the frequencies are).

- (a) Code:  $\{0, 10, 11\}$
- (b) Code:  $\{0, 1, 00\}$
- (c) Code:  $\{10, 01, 00\}$

- (a)  $(f_a, f_b, f_c) = (2/3, 1/6, 1/6)$  gives the code  $\{0, 10, 11\}$ .
- (b) This encoding is not possible, since the code for  $a$  (0), is a prefix of the code for  $c$  (00).
- (c) This code is not optimal since  $\{1, 01, 00\}$  gives a shorter encoding. Also, it does not correspond to a *full* binary tree and hence cannot be given by the Huffman algorithm.