Introduction to Algorithms

Department of Computer Science and Engineering

East China University of Science and Technology

Lecture 08

Greedy Algorithm
&Dynamic Programming

Topic:

- **Huffman codes**
- **Single-Source Shortest Paths**
- **Minimum Spanning Trees**

Huffman codes

Data Compression via Huffman Coding

- Human codes are used for data compression.
  - **Reducing time to transmit large files**
  - **Reducing the space required to store them on disk or tape.**
- Huffman codes are a widely used and very effective technique for compressing data, savings of 20% to 90% are typical, depending on the characteristics of the data.

Huffman codes

- Problem
  - Suppose that you have a file of 100,000-character. To keep the example simple, suppose that each character is one of the 6 letters from *a* through *f*. Since we have just 6 characters, we need just 3-bit to represent a character, so the file requires 300,000 bits to store. Can we do better?
  - Suppose that we have more information about the file:
    - **the frequency which each character appears.**
- Solution
  - The idea is that we will use a variable length code instead of a fixed length code (3-bit for each character), with fewer bits to store the common characters, and more bits to store the rare characters.

Huffman codes---- Prefix code

- In a Prefix code no codeword is a prefix of another code word.
  - Easy encoding and decoding.
- To encode, we need only concatenate the codes of consecutive characters in the message.
  - The string 110001001101 parses uniquely as 1100−0−100−1101, which decodes to FACE.
- To decode, we have to decide where each code begins and ends.
  - Easy, since, no codes share a prefix.

Huffman codes

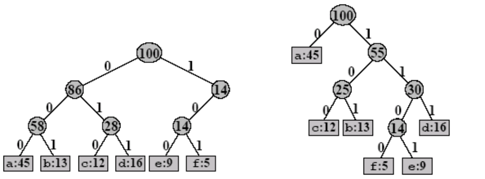
- For example, suppose that the characters appear with the following frequencies, and following codes:

	a	b	c	d	e	f
Frequency (in thousand)	45	13	12	16	9	5
Fixed length codeword	000	001	010	011	100	101
Variable length codeword	0	101	100	111	1101	1100

- Then the variable-length coded version will take not 300,000 bits but 45*1 + 13*3 + 12*3 + 16*3 + 9*4 + 5*4 = 224,000 bits to store, a 25% saving. In fact this is the optimal way to encode the 6 characters present, as we shall see.

Huffman codes

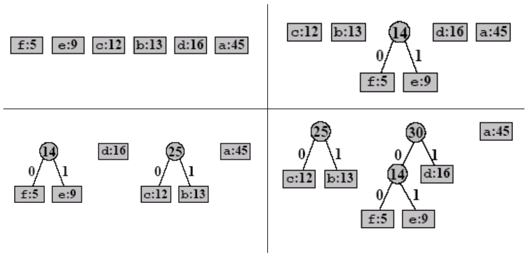
- Represented as a binary tree whose leaves are the given characters.
- In an optimal code each non-leaf node has two children.



(a) fixed-length codeword (b)variable-length codeword

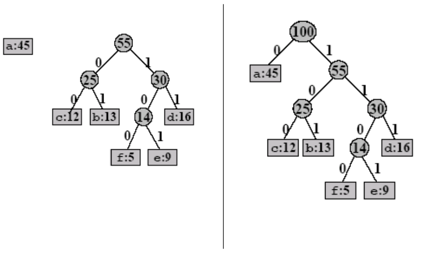
Huffman codes

- Example of Huffman codes (a)



Huffman codes

- Example of Huffman codes (b)

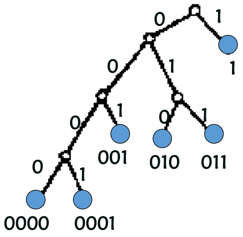


Huffman codes

- The greedy algorithm for computing the optimal Human coding tree *T* is as follows.
  - It starts with a forest of one-node trees representing each *c* ∈*C*, and merges them in a greedy style, using a priority queue *Q*, sorted by the smallest frequency:

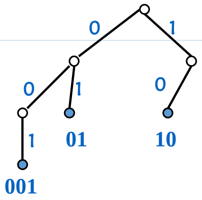
Binary Tree v.s. Prefix Code

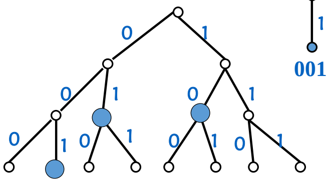
Example



{0000，0001，001，010，011，1}

Example {001， 01， 10}



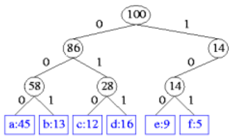


Optimal Prefix Code Design

- **Coding Cost** of T

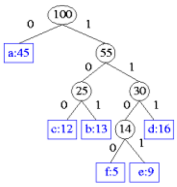
$$B(T) = \sum_{c \in C} f(c) d_T(c)$$

  - c : each character in the alphabet C
  - $f(c)$: frequency of c
  - $d_T(c)$: depth of c's leaf (length of the codeword of c)
- **Code design:** Given $f(c_1)$, $f(c_2)$, …, $f(c_n)$, construct a binary tree with n leaves such that $B(T)$ is minimized.
  - **Idea: more frequently used characters use shorter depth.**



Fixed–length cost: 3 * 100 = 300

optimal code —> full binary tree!!



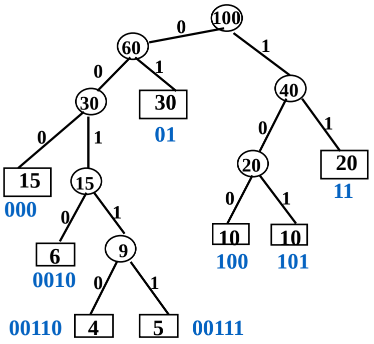
Variable–length cost = 224

Huffman's Procedure

- **Pair** two nodes with the least costs at each step.

Frequencies as weights:

4 , 5 , 6 , 10, 10, 15 , 20, 30



Huffman's Algorithm

```
• Huffman(C)
• 1. n ← |C|;
• 2. Q ← C;
• 3. for I ← 1 to n-1
• 4.     z ← Allocate-Node();
• 5.     x ← left[z] ← Extract-Min(Q);
• 6.     y ← right[z] ← Extract-Min(Q);
• 7.         f[z] ← f[x]+f[y];
• 8.         Insert(Q, z);
• 9. return Extract-Min(Q)
```

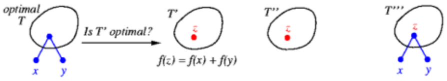
Time complexity: $O(n \lg n)$.

Extract-Min(Q) needs $O(\lg n)$ by a heap operation.

Requires initially $O(n \lg n)$ time to build a binary heap.

Huffman Algorithm:Optimal substucture

- **Optimal substructure:** Let T be a full binary tree for an optimal prefix code over C. Let z be the parent of two leaf characters x and y. If $f[z]=f[x]+f[y]$, tree $T' = T - \{x, y\}$ represents an optimal prefix code for $C' = C - \{x, y\} \cup \{z\}$.



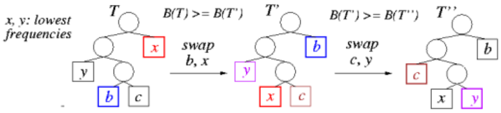
$B(T) = B(T') + f(x) + f(y)$

If T' is not optimal, find T'' s.t. $B(T'') < B(T')$.
z in C' => z is a leaf of T''.
Add x, y as z's children (T''')

$$
\begin{aligned}
=> B(T''') &= B(T'') + f(x) + f(y)\\
&< B(T') + f(x) + f(y)\\
&= B(T) \quad \textit{a contradiction!!}
\end{aligned}
$$

Huffman Algorithm: Greedy Choice

- **Greedy choice:** Two characters x and y with the lowest frequencies must have *the same length* and differ only in the last bit.



T is an optimal tree → T' is an optimal tree → T'' is an optimal tree

Huffman codes

```
HUFFMAN( C )
n ← |C|
Q ← C
for  i ← 1 to  n – 1
    do allocate a new node z
        left[z] ← x ← EXTRACT-MIN(Q)
        right[z] ← y ← EXTRACT-MIN(Q)
        f[z] ← f[x] + f[y]
        INSERT(Q,Z)
return EXTRACT-MIN(Q)
```

Single-Source Shortest Paths

- The problem:

A motorist wishes to find the shortest possible route from Chicago to Boston. Given a road map of the United States on which the distance between each pair of adjacent intersections is marked, how can we determine this shortest route?

Single-Source Shortest Paths

- Given a weighted, directed graph $G = (V, E)$, with weight function $w : E \rightarrow R$ mapping edges to real-valued-weights. Given a vertex in V, called the **source vertex** . Now we need to calculate all other vertices from the source to the length of the shortest path. **The weight of path p = v0, v1, ..., vk** is the sum of the weights of its constituent edges. This problem is usually referred to as **Single-Source Shortest Paths.**

Single-Source Shortest Paths

Dijkstra algorithm is the greedy algorithm for the solution of single-source shortest path problem.

The basic idea is to set the vertex set S and expand the collection by continually making greedy choices. A vertex belonging to the set S if and only if the known length of the shortest path from the source to the vertex.

The initial, S contains only source vertex.Assume u is a vertex in G,the road that from source to u and through the S vertices is called special path from source to u,and use the array dist to record the corresponding shortest special path length of each vertex.

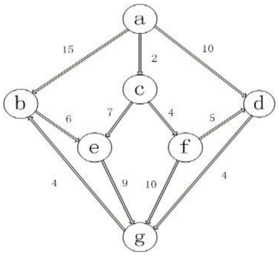
Single-Source Shortest Paths

- Dijkstra algorithm everytime takes out vertex u that has the shortest special path length from the V-S , put u add to S,and at the same time do some necessary modifications on the array dist.
- Once S contains all of V vertices,the array dist has recorded the length of the shortest path from the source to all other vertices.

Single-Source Shortest Paths

for instance:

a directed graph,use Dijkstra algorithm to calculate the Shortest path from the source vertex **a** to other vertices.



Single-Source Shortest Paths

Destination	**b**	**c**	**d**	**e**	**f**	**g**
Dist	15	2	10	9	6	14
Path	ab	ac	ad	ace	acf	adg

Single-Source Shortest Paths

- min ←selected the shortest path length
- // Update other vertices shortest path

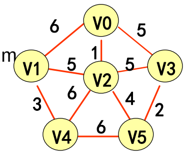
```
If(!final[w] && (min + G.arcs[v][w]<Dist[w]))
Dist[w] ←min +G.arcs[v][w]
Path[w] ←Path[v]+<v,w>
```

Minimum Spanning Tree

- *optimal substructure* property？
- overlapping subproblem?

Minimum Spanning Trees

In all spanning trees from a connected, undirected graph,the Minimum Spanning Trees is the minimum sum of the weights of its constituent edges.



For example:

To create a transport network between n cities, to consider the problem of how to ensure the most savings under the premise of the n-point connectivity?

Problem: connectivity to six cities and least costly transit line?

Solution: **Prim's algorithm** and **Kruskal's algorithm**.

Minimum Spanning Trees

1. Prime algorithm

Assume G=(V,E) is a connected, undirected graph with a real-valued weight. V={1,2,…,n}.

（1） Let S={1},if S is the proper subset of V,then make greedy choices as follow:

（2） Select meet the conditions of i ∈S,

j ∈ V-S,and c[i][j] is the minimum edge and put the vertex j add to the S.

（3） Repeat （2） Until S=V.

In this process, all edges selected exactly constitute a minimum spanning tree of G,

Minimum Spanning Trees

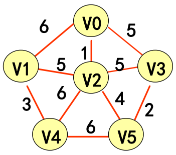
1. Prime algorithm

```
void Prim(int n,Type *c) {
    T ← Ø
    S ←{1}
    while(S != V){
        (i,j) ← the minimum edge of i ∈S and j ∈V-S
        T ← T ∪{ (i,j) };
        S ← S ∪{j} ;
    }
}
```

Minimum Spanning Trees

1. Prime algorithm

Example: Calculate the minimum spanning tree of the following figure.



Prim's Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```

Review: Prim's Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```

What is the hidden cost in this code?

Review: Prim's Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                DecreaseKey(v, w(u,v));
```

How often is ExtractMin() called?

How often is DecreaseKey() called?

Review: Prim’s Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```

What will be the running time?

A: Depends on queue

Minimum Spanning Trees

2. Kruskal algorithm

Assume G=(V,E) is a connected, undirected graph with a real-valued weight. V={1,2,…,n}.according to the weight of G from small to large order;

（1） Treat N vertices as n sets；

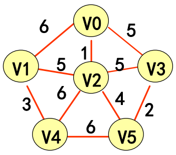
（2） According to the weight from small to large order selecte the edge, the selected edge should satisfy that two vertices are not at the same vertex set inside, put the edge into the tree of the set. At the same time merger the two vertices of this edge into vertex set；

（3）Repeat (2), until all vertices are in the same vertex sets in.

Minimum Spanning Trees

2. Kruskal algorithm

Example: Calculate the minimum spanning tree of the following figure.



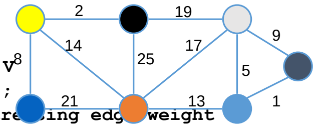
Minimum Spanning Trees

2. Kruskal algorithm

```
MST-KRUSKAL(G, w)
1   A ← Ø
2   for each vertex v □ V[G]
3        do MAKE-SET(v)
4   sort the edges of E into nondecreasing order by weight w
5   for each edge (u, v) □ E, taken in nondecreasing order by weight
6        do if FIND-SET(u) ≠ FIND-SET(v)
7              then A ← A □ {(u, v)}
8                   UNION(u, v)
9   return A
```

Kruskal’s Algorithm

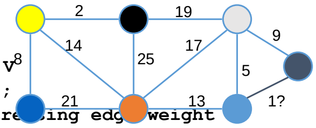
Run the algorithm:



```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T 🗐 {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Kruskal’s Algorithm

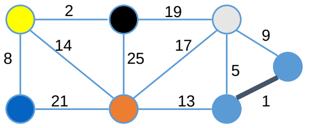
Run the algorithm:



```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T 🗐 {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

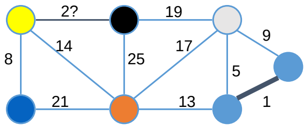
Run the algorithm:



```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T 🗐 {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

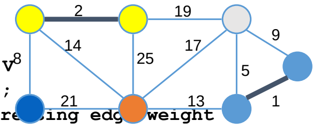
Run the algorithm:



```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T 🗐 {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

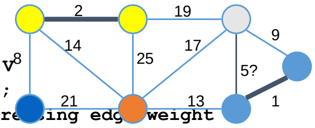
Run the algorithm:



```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by incre ing edg  weight
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T 🗐 {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

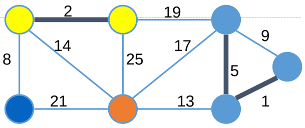
Run the algorithm:



```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by incre ing edg  weight
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T 🗐 {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

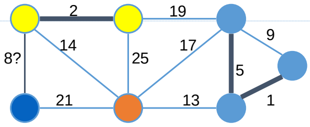
Run the algorithm:



```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T 🗐 {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

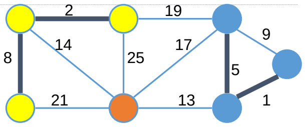
Run the algorithm:



```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T 🗐 {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

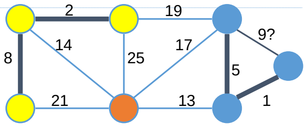
Run the algorithm:



```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T 🗐 {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

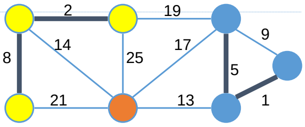
Run the algorithm:



```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T 🗐 {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

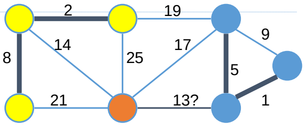
Run the algorithm:



```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T 🗐 {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

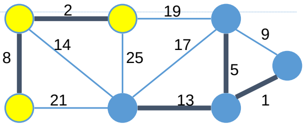
Run the algorithm:



```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T 🗐 {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

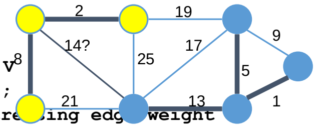
Run the algorithm:



```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T 🗐 {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

Run the algorithm:

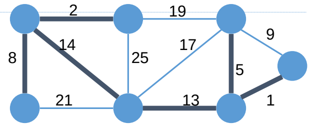


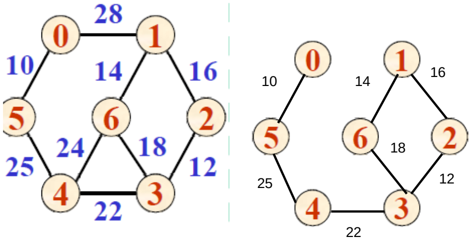
```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T 🗐 {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Kruskal’s Algorithm

```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T 🗐 {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Run the algorithm:





Thanks!