中山大學
SUN YAT-SEN UNIVERSITY

Lecture 2
Greedy Algorithms

Algorithm Design

zhangzizhen@gmail.com

QQ group: 117282780

Introduction

- Definition:

  A *greedy algorithm* is an algorithm in which at each stage a locally optimal choice is made.

- Characteristics:

  1. Greedy-choice property: A global optimum can be arrived at by selecting a local optimum.

  2. Optimal substructure: An optimal solution to the problem contains an optimal solution to subproblems.

- Greedy algorithms are usually extremely efficient, but they can only be applied to a small number of problems.

Introduction

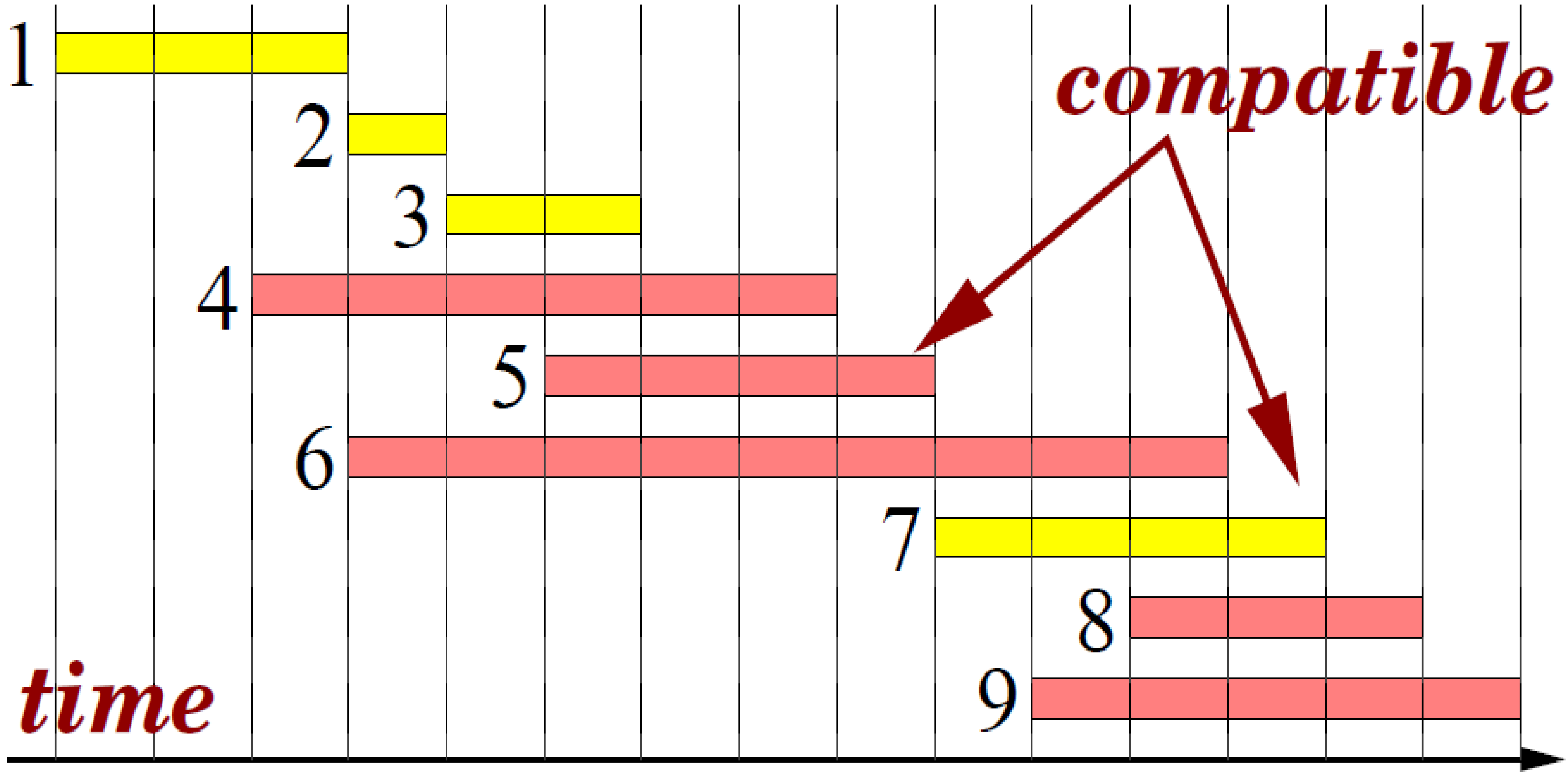
- 贪心算法总是作出在当前看来最好的选择。也就是说贪心算法并不从整体最优考虑，它所作出的选择只是在某种意义上的**局部最优**选择。当然，希望贪心算法得到的最终结果也是整体最优的。虽然贪心算法不能对所有问题都得到整体最优解，但对许多问题它能产生整体最优解。如单源最短路经问题，最小生成树问题等。在一些情况下，即使贪心算法不能得到整体最优解，其最终结果却是最优解的很好近似。

Activity Selection Problem

- Let $S = \{1, 2, . . . , n\}$ be the set of activities that compete for a resource. Each activity i has its starting time s_i and finish time f_i with $s_i \leq f_i$, namely, if selected, i takes place during time $[s_i, f_i)$. No two activities can share the resource at any time point. We say that activities i and j are compatible if their time periods are disjoint.

- The *activity selection problem* is the problem of selecting the largest set of mutually compatible activities.

Activity Selection Problem



Activity Selection Problem

- **Greedy template**. Consider activities in some natural order. Take each activity provided it's compatible with the ones already taken.

  [Earliest start time] Consider jobs in ascending order of s_i.

  [Earliest finish time] Consider jobs in ascending order of f_i.

  [Shortest interval] Consider jobs in ascending order of f_i-s_i.

  [Fewest conflicts] For each job i, count the number of conflicting jobs c_i. Schedule in ascending order of c_i.

Activity Selection Problem

- Counterexamples:

counterexample for earliest start time

counterexample for shortest interval

counterexample for fewest conflicts

Activity Selection Problem

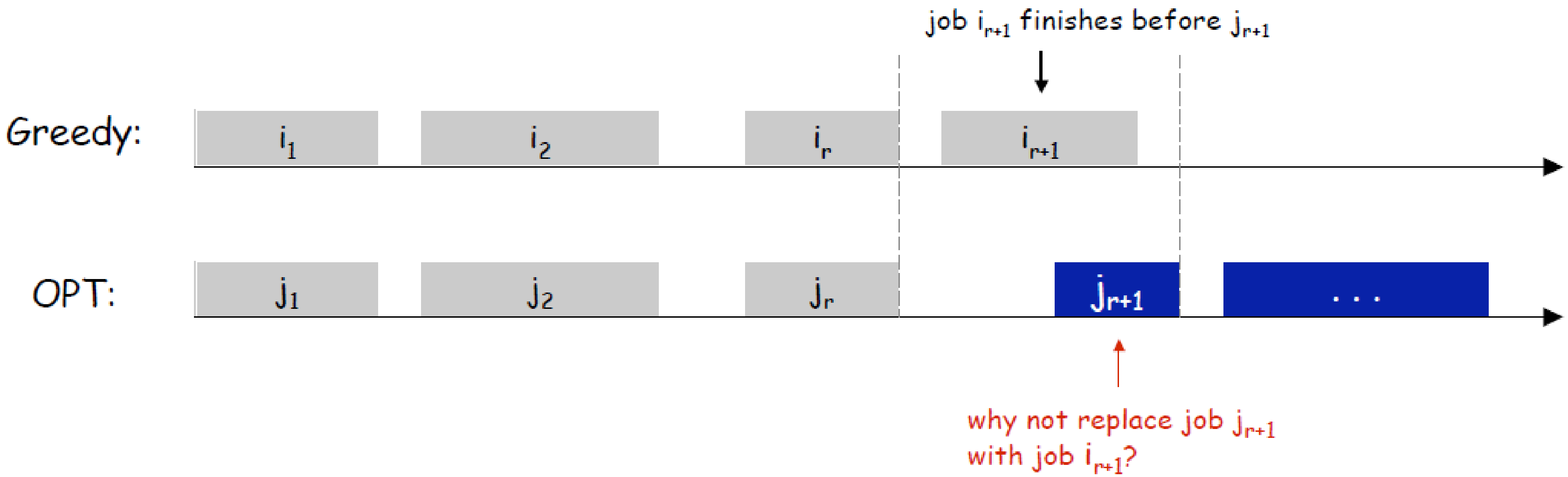
- Greedy algorithm. Consider jobs in increasing order of finish time. Take each job provided it's compatible with the ones already taken.

```
Sort activities by finish times so that f1 <= f2 <= ... <= fn
A = Φ
for j = 1 to n {
    if (activity j compatible with A) A = A ∪ {j}
}
return A
```

- Implementation. O(nlogn)+O(n)

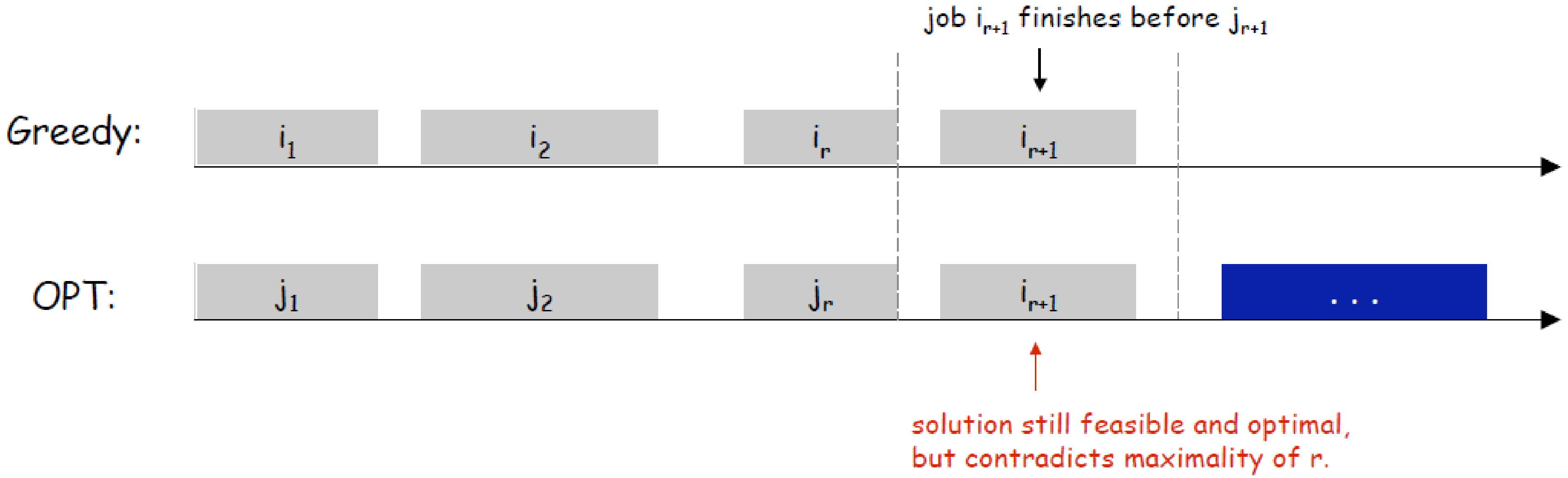
Activity Selection Problem

- **Theorem**: Greedy algorithm is optimal for the activity selection problem.
- **Proof: (by contradiction)**
  - Assume greedy is not optimal.
  - Let $i_1, i_2, \ldots i_k$ denote set of jobs selected by greedy.
  - Let $j_1, j_2, \ldots j_m$ denote set of jobs in the optimal solution with $i_1 = j_1$, $i_2 = j_2$, ..., $i_r = j_r$ for the largest possible value of r.



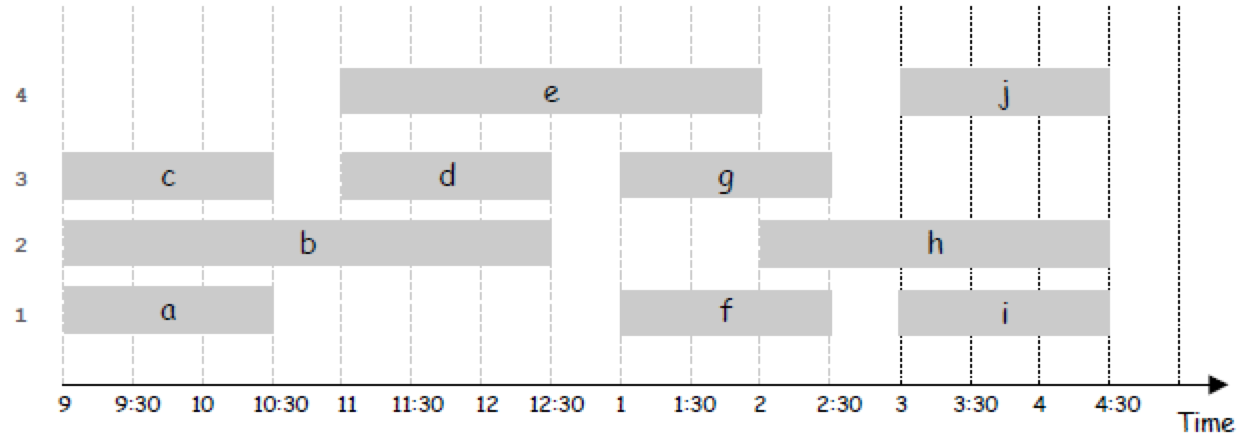
Activity Selection Problem

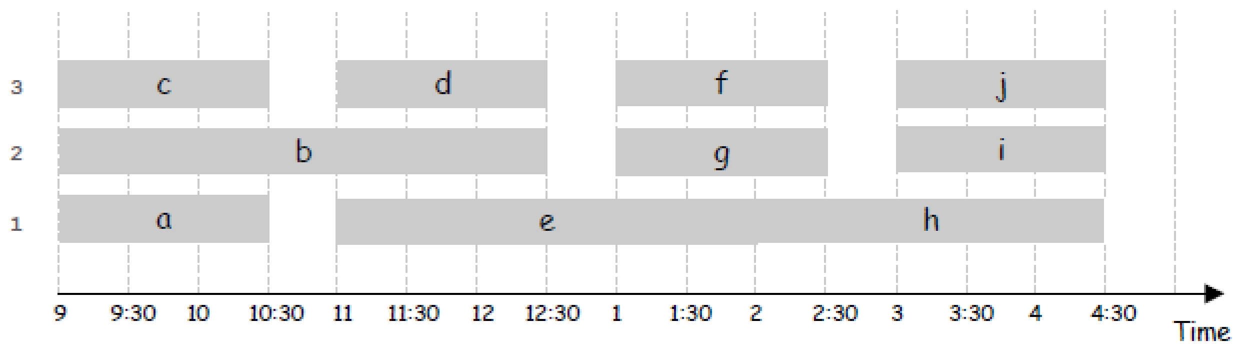
- **Theorem**: Greedy algorithm is optimal for the activity selection problem.
- **Proof: (by contradiction)**
  - Assume greedy is not optimal.
  - Let $i_1, i_2, \ldots i_k$ denote set of jobs selected by greedy.
  - Let $j_1, j_2, \ldots j_m$ denote set of jobs in the optimal solution with $i_1 = j_1$, $i_2 = j_2$, ..., $i_r = j_r$ for the largest possible value of r.



Interval Partitioning Problem

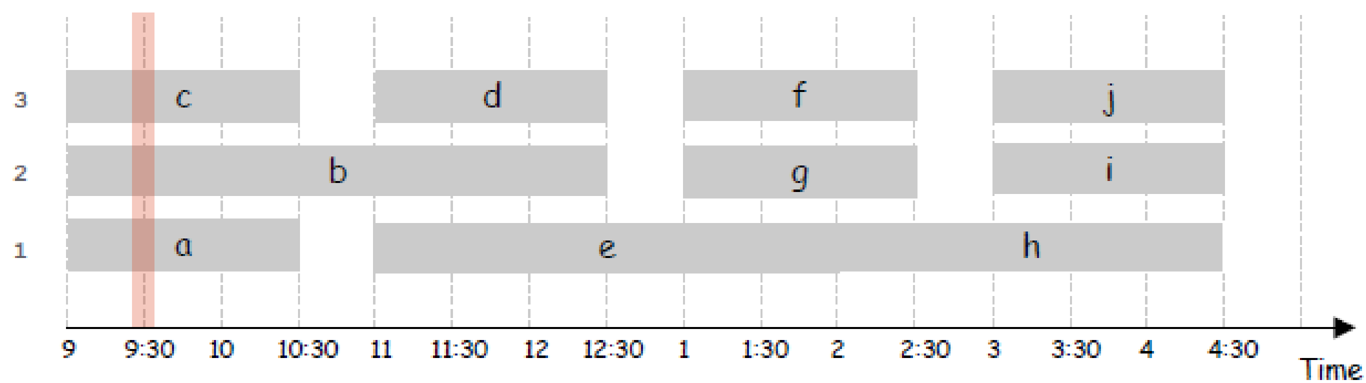
- Lecture j starts at s_j and finishes at f_j.
- Goal: find minimum number of classrooms to schedule all lectures so that no two occur at the same time in the same room.





Interval Partitioning Problem

- **Definition**. The depth of a set of open intervals is the maximum number that contain any given time.
- **Key observation**. Number of classrooms needed ≥ depth.
- **Question**. Does there always exist a schedule equal to depth of intervals?



Interval Partitioning Problem

- **Greedy algorithm**. Consider lectures in increasing order of start time: assign lecture to any compatible classroom.

```
Sort intervals by starting time so that s1 ≤ s2 ≤ ... ≤ sn.
d ← 0
for j = 1 to n {
    if (lecture j is compatible with some classroom k)
        schedule lecture j in classroom k
    else
        allocate a new classroom d + 1
        schedule lecture j in classroom d + 1
        d ← d + 1
}
```

- Implementation. O(*n* log *n*).
- For each classroom *k*, maintain the finish time of the last job added.
- Keep the classrooms in a priority queue.

Interval Partitioning Problem

- **Theorem**. Greedy algorithm is optimal.
- **Proof**.
  - Let *d* = number of classrooms that the greedy algorithm allocates.
  - Classroom *d* is opened because we needed to schedule a job, say *j*, that is incompatible with all *d*-1 other classrooms.
  - These *d* jobs each end after s_j.
  - Since we sorted by start time, all these incompatibilities are caused by lectures that start no later than s_j.
  - Thus, we have *d* lectures overlapping at time $s_j + \varepsilon$.

Fractional Knapsack Problem

- We have n objects and a knapsack. The i-th object has positive weight w_i and positive value v_i. The knapsack capacity is C. We wish to select a set of proportions of objects to put in the knapsack so that the total values is maximum and without breaking the knapsack.

- Example:
- $n = 5$, $C = 100$

w	10	20	30	40	50
v	20	30	66	40	60

$$\max. \sum_{i=1}^{n} v_i x_i$$

$$s.t. \sum_{i=1}^{n} w_i x_i \le W$$

$$0 \le x_i \le 1$$

Fractional Knapsack Problem

- **Greedy template**.

  [Select always the lighter object] Total selected weight 100 and total value 156.

object	1	2	3	4	5
selected	1	1	1	1	0

  [Select always the most valuable object] Total selected weight 100 and total value 146.

object	1	2	3	4	5
selected	0	0	1	0.5	1

  [Select always the object with highest ratio value/weight] Total selected weight 100 and total value 164.

object	1	2	3	4	5
ratio	2.0	1.5	2.2	1.0	1.2
selected	1	1	1	0	0.8

Fractional Knapsack Problem

- **Theorem:** The greedy algorithm that always selects the object with better ratio value/weight always finds an optimal solution to the Fractional Knapsack problem.
- **Proof:**

Assume that the objects are {1, ... , n} and that

$$\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq \ldots \geq \frac{v_n}{w_n}$$

Let $X = (x_1, . . . , x_n)$ be the solution computed by the greedy algorithm.

If $x_i = 1$ for all i, the solution is optimal. Otherwise, let j be the smallest value for which $x_j < 1$. According to the algorithm we have: If $i < j$ then $x_i = 1$, and if $i > j$ then $x_i = 0$.

Furthermore, $\sum_{i=1}^{n} w_i x_i = W$

Fractional Knapsack Problem

- Let $Y = (y_1, \dots , y_n)$ be any feasible solution, we have

$$\sum_{i=1}^{n} w_i y_i \le W = \sum_{i=1}^{n} w_i x_i$$

so, $\sum_{i=1}^{n} w_i (x_i - y_i) \ge 0$

Let $V(Z)$ denotes the total value of a feasible solution.

$$V(X) - V(Y) = \sum_{i=1}^{n} v_i (x_i - y_i) = \sum_{i=1}^{n} w_i \frac{v_i}{w_i} (x_i - y_i)$$

If $i<j$, $x_i=1$, then $x_i-y_i>=0$ and $v_i/w_i>=v_j/w_j$, we have

$$(x_i - y_i)\frac{v_i}{w_i} \ge (x_i - y_i)\frac{v_j}{w_j}$$

If $i>j$, $x_i=0$, then $x_i-y_i<=0$ but $v_i/w_i<=v_j/w_j$, we also have

$$(x_i - y_i)\frac{v_i}{w_i} \ge (x_i - y_i)\frac{v_j}{w_j}$$

Fractional Knapsack Problem

Plugging the inequality we have,

$$V(X)-V(Y)=\sum_{i=1}^{n} w_i \frac{v_i}{w_i}(x_i - y_i) \geq \sum_{i=1}^{n} w_i \frac{v_j}{w_j}(x_i - y_i)$$

$$= \frac{v_j}{w_j} \sum_{i=1}^{n} w_i (x_i - y_i) \geq 0$$

Therefore, *X* is an optimal solution.

0-1 Knapsack Problem

$$\max . \sum_{i=1}^{n} v_i x_i$$

$$s.t. \ \sum_{i=1}^{n} w_i x_i \le W$$

$$x_i \in \{0,1\}$$

$$c[i,w] = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0 \\ c[i-1,w] & \text{if } w_i > w \\ \max(v_i + c[i-1, w-w_i], c[i-1,w]) & \text{if } i > 0 \text{ and } w \geqslant w_i \end{cases}$$

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

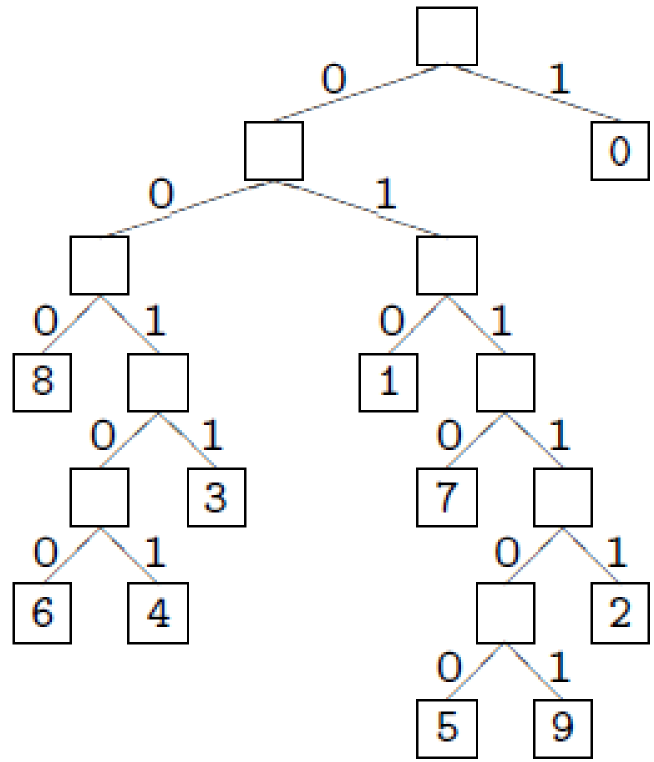
- What would happen if we used the following code to store the data rather than the fixed length code?

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

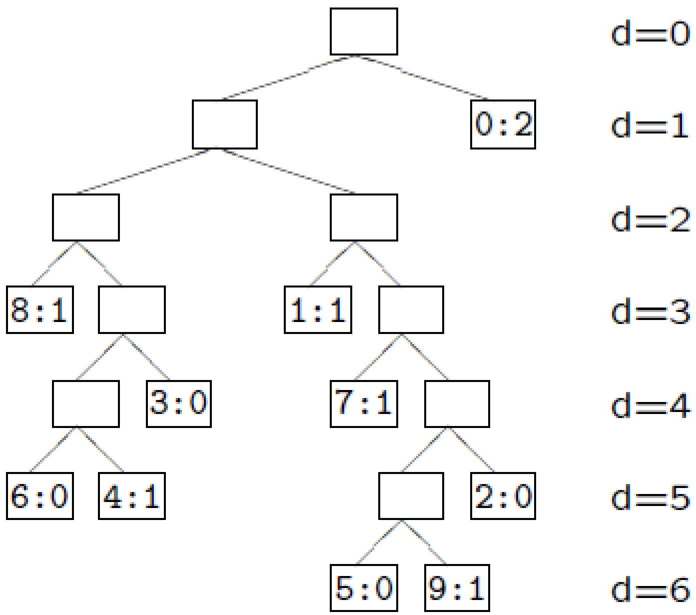
- Now assign to each leaf of the tree a value, *f(c)*, which is the frequency of occurrence of the character *c* represented by the leaf.
- Let $d_T(c)$ be the depth of character *c*'s leaf in the tree *T*.
- Then the number of bits required to encode a file is

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

- 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

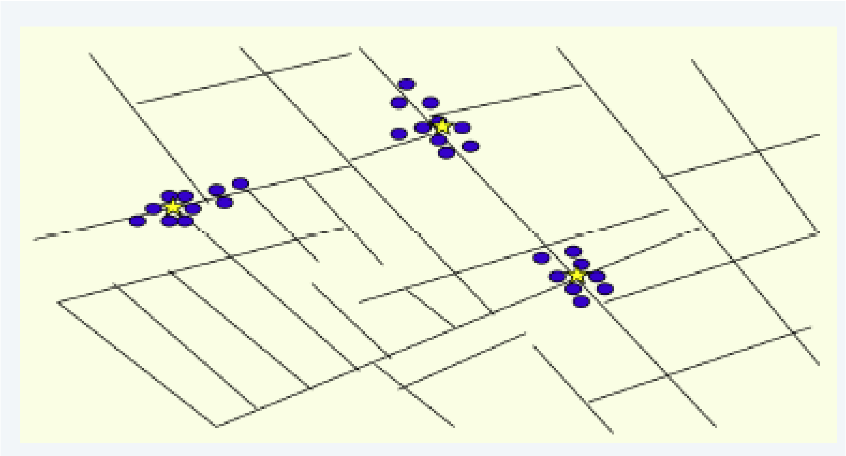
- Then repeat the following procedure $s - 1$ times:
- Remove from the priority queue the two nodes L and R with the lowest values, and create a internal node of the binary tree whose left child is L and right child R.
- Compute the value of the new node as the sum of the values of L and R and insert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

Huffman Coding

- Huffman算法用最小堆实现优先队列Q。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

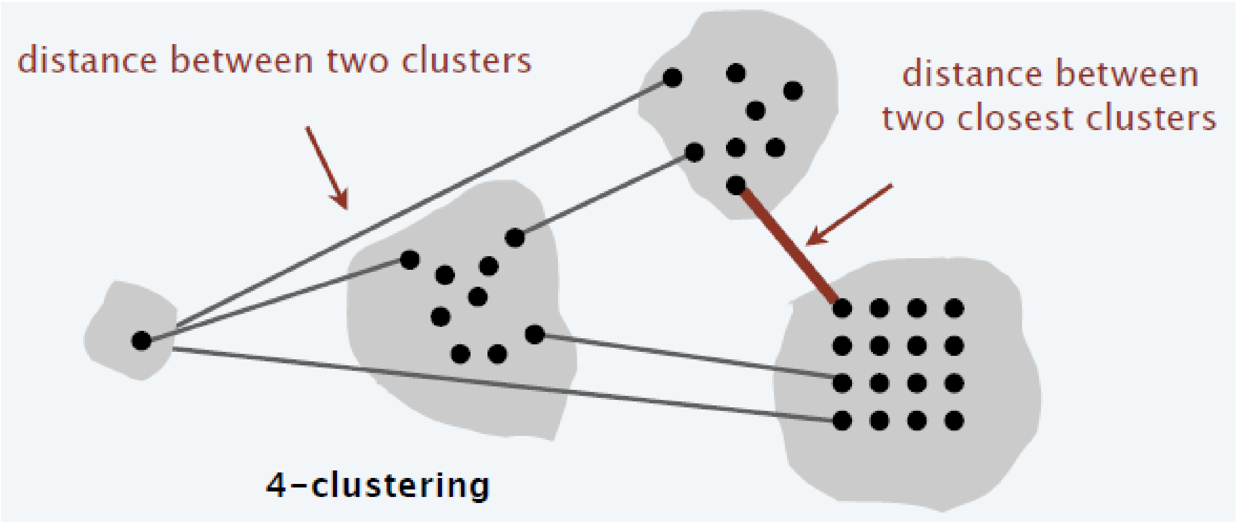
- Given a set U of n objects labeled $p_1, \ldots, p_n$, partition into clusters so that objects in different clusters are far apart.



- 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.
- ...

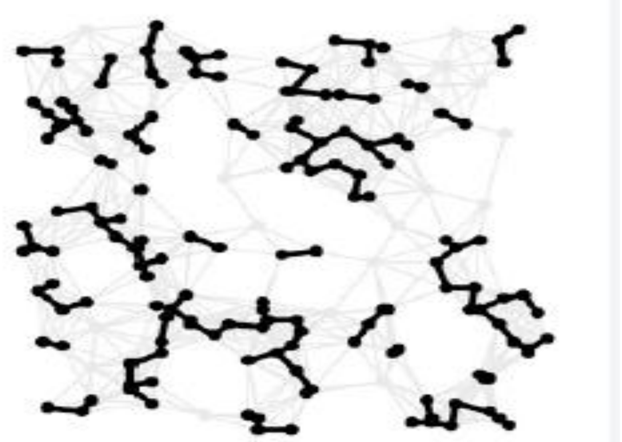
Single-link Clustering

- ***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

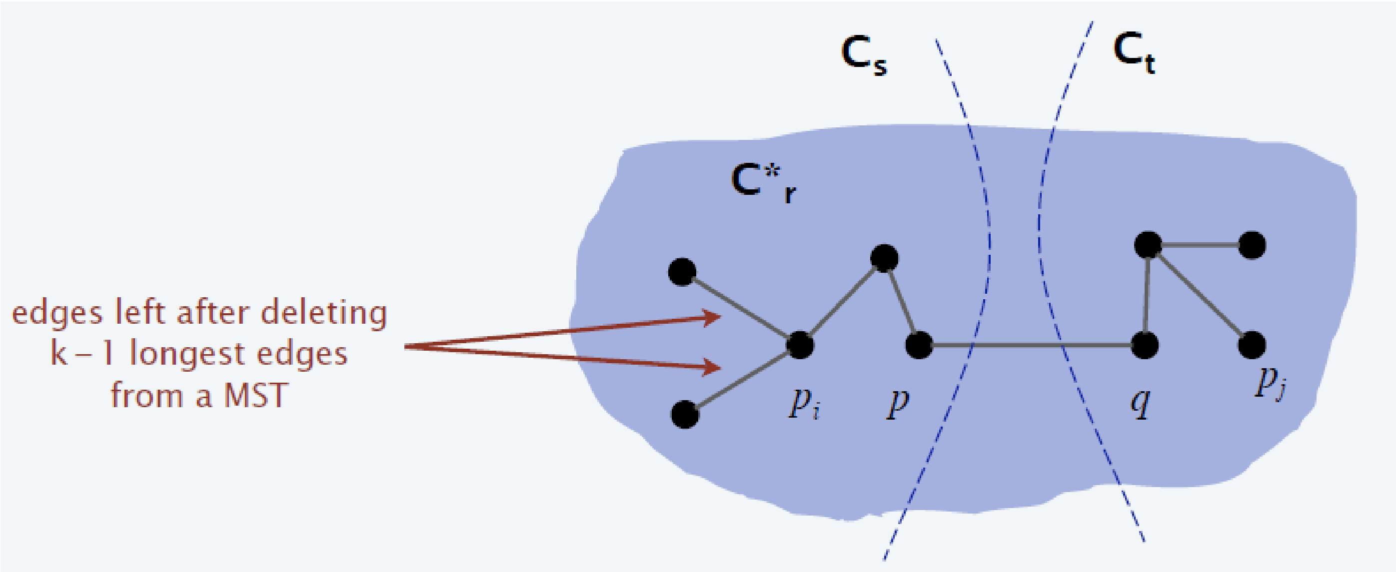
- "Well-known" algorithm in science literature for single-linkage *k*-clustering:
  - Form a graph on the node set *U*, corresponding to *n* clusters.
  - Find the closest pair of objects such that each object is in a different cluster, and add an edge between them.
  - Repeat *n* – *k* times until there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



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

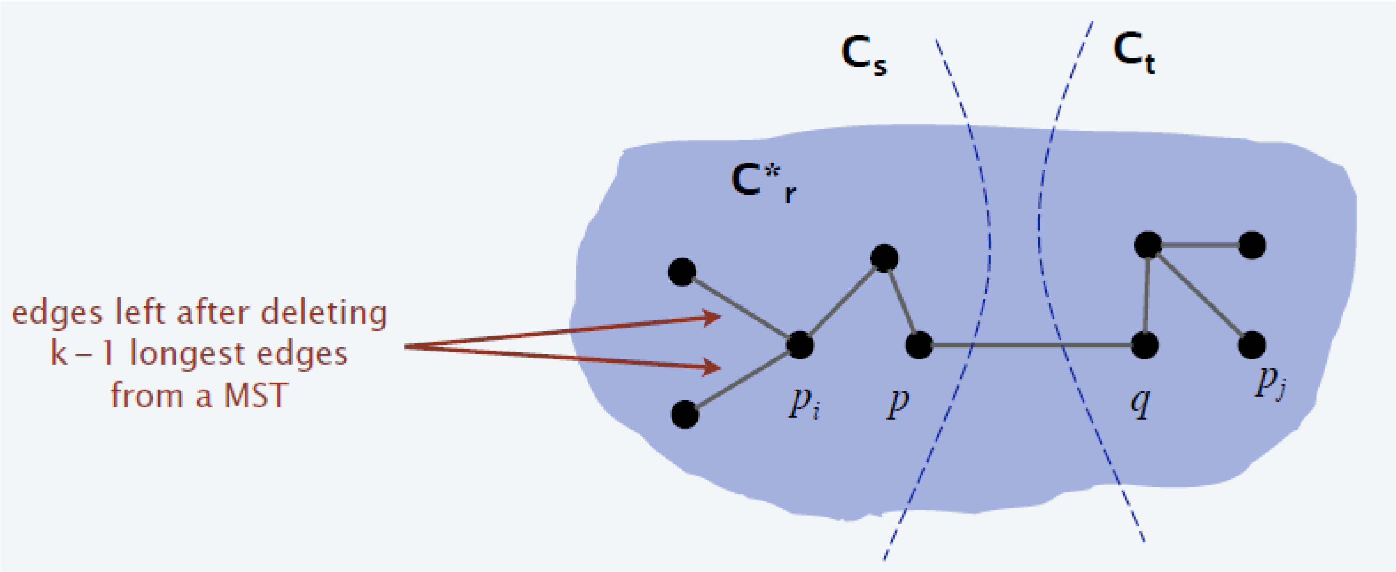
Single-link Clustering

- **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.



Thank you!