Outline

▶ Optimal cashing
Outline

- Optimal cashing
- Shortest paths in graphs
Optimal cashing
Shortest paths in graphs
Minimum spanning trees
Outline

- Optimal cashing
- Shortest paths in graphs
- Minimum spanning trees
- Clustering
Outline

- Optimal cashing
- Shortest paths in graphs
- Minimum spanning trees
- Clustering
- Huffman codes and data compression
Optimal Cashing: problem

- **Given:**
  - Cache with capacity to store $k$ items.
  - Sequence of $m$ item requests $d_1, d_2, \ldots, d_m$.
  - Cache hit: item already in cache when requested.
  - Cache miss: item not already in cache when requested: must bring requested item into cache, and evict some existing item, if full.
  - **Goal:** Eviction schedule that minimizes number of cache misses.
Optimal Cashing: problem

- **Given:**
  - Cache with capacity to store $k$ items.

- **Cache hit:** item already in cache when requested.

- **Cache miss:** item not already in cache when requested: must bring requested item into cache, and evict some existing item, if full.

- **Goal:** Eviction schedule that minimizes number of cache misses.
Optimal Cashing: problem

Given:

- Cache with capacity to store $k$ items.
- Sequence of $m$ item requests $d_1, d_2, \ldots, d_m$. 

Cache hit: item already in cache when requested.

Cache miss: item not already in cache when requested: must bring requested item into cache, and evict some existing item, if full.

Goal: Eviction schedule that minimizes number of cache misses.
Optimal Cashing: problem

- **Given:**
  - Cache with capacity to store $k$ items.
  - Sequence of $m$ item requests $d_1, d_2, \ldots, d_m$.
  - **Cache hit:** item already in cache when requested.
  - **Cache miss:** item not already in cache when requested.

- **Goal:** Eviction schedule that minimizes number of cache misses.
Optimal Cashing: problem

- **Given:**
  - Cache with capacity to store $k$ items.
  - Sequence of $m$ item requests $d_1, d_2, \ldots, d_m$.
  - **Cache hit:** item already in cache when requested.
  - **Cache miss:** item not already in cache when requested: must bring requested item into cache, and evict some existing item, if full.
Optimal Cashing: problem

- **Given:**
  - Cache with capacity to store $k$ items.
  - Sequence of $m$ item requests $d_1, d_2, \ldots, d_m$.
  - **Cache hit:** item already in cache when requested.
  - **Cache miss:** item not already in cache when requested: must bring requested item into cache, and evict some existing item, if full.

- **Goal:** Eviction schedule that minimizes number of cache misses.
Optimal Cashing: example

- \( k = 2 \) items.
Optimal Cashing: example

- $k = 2$ items.
- Initial cache has $ab$
Optimal Cashing: example

- $k = 2$ items.
- Initial cache has $ab$
- requests: $a, b, c, b, c, a, a, b$. 
Optimal Cashing: example

- $k = 2$ items.
- Initial cache has $ab$
- Requests: $a, b, c, b, c, a, a, b$.

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>a</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
<td>b</td>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>b</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>c</td>
<td>c</td>
<td>b</td>
<td>c</td>
<td>b</td>
</tr>
<tr>
<td>b</td>
<td>c</td>
<td>b</td>
<td>c</td>
<td>b</td>
</tr>
<tr>
<td>a</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>a</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>b</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
</tr>
</tbody>
</table>

- Requests cache

Prof. Dr. Brahim Hnich  Algorithm Design  Greedy Algorithms (Part II)
Optimal Cashing: example

- $k = 2$ items.
- Initial cache has $ab$
- Requests: $a, b, c, b, c, a, a, b$.

Optimal eviction schedule: 2 cache misses.
Optimal Cashing: greedy algorithm

**Farthest-in-future.** Evict item in the cache that is not requested until farthest in the future.
**Farthest-in-future.** Evict item in the cache that is not requested until farthest in the future.

When $d_i$ needs to be brought into the cache, evict the item that is needed the farthest into the future.
Optimal Cashing: greedy algorithm

Farthest-in-future. Evict item in the cache that is not requested until farthest in the future.

When \( d_i \) needs to be brought into the cache, evict the item that is needed the farthest into the future.

current cache: \[\text{a b c d e f}\]

future queries: \[\text{g a b c e d a b b a c d e a f a d e f g h ...}\]

\[\text{↑}\] cache miss

\[\text{↑}\] eject this one
Definition. A *reduced schedule* is a schedule that only inserts an item into the cache in a step in which that item is requested.
Optimal Cashing: analysis

**Definition.** A *reduced schedule* is a schedule that only inserts an item into the cache in a step in which that item is requested.

**Intuition.** Can transform an unreduced schedule into a reduced one with no more cache misses.
**Definition.** A *reduced schedule* is a schedule that only inserts an item into the cache in a step in which that item is requested.

**Intuition.** Can transform an unreduced schedule into a reduced one with no more cache misses.

An unreduced schedule:

```
a a b c
a a x c
c a d c
d a d b
a a c b
b a x b
```  

A reduced schedule:

```
a a b c
a a b c
c a b c
d a d c
a a d c
b a d b
```
Theorem

Given any unreduced schedule $S$, we can transform it into a reduced schedule $S'$ with no more cache misses.

Proof.

In any step $i$ where $S$ brings in an item $d$ that has not been requested, our construction of $S'$ ”pretends” to do this but actually leaves $d$ in main memory and brings $d$ into the cache when its requested in step $j$. The cache miss incurred by $S'$ in step $j$ is counter-balanced with the cache operation performed by $S$ in step $i$. 

□
Optimal Cashing: analysis

\[ S \quad \mid \quad c \quad \mid \quad S' \]
\[ t \quad \rightarrow \quad d \quad \rightarrow \quad t' \]
\[ d \text{ requested at time } t' \]
Theorem
Farthest-in-future is an optimal eviction schedule.

Proof.
(by induction on number of requests $j$). **Invariant:** There exists an optimal reduced schedule $S$ that makes the same eviction schedule as $S' = S_{FF}$ through the first $j + 1$ requests.
Let $S$ be reduced schedule that satisfies invariant through $j$ requests. We produce $S'$ that satisfies invariant after $j + 1$ requests. Consider $(j + 1)^{st}$ request $d = d_{j+1}$. Since $S$ and $S'$ have agreed up until now, they have the same cache contents before request $j + 1$.

**Case 1:** ($d$ is already in the cache). $S' = S$ satisfies invariant.

**Case 2:** ($d$ is not in the cache and $S$ and $S'$ evict the same element). $S' = S$ satisfies invariant.
Theorem

Farthest-in-future is an optimal eviction schedule.

Proof.

... 

Case 3: \((d \text{ is not in the cache; } S' \text{ evicts } e; S \text{ evicts } f \neq e)\).

\begin{align*}
\text{j} & \quad \text{same} & \text{e} & \text{f} & \quad \text{same} & \text{e} & \text{f} & \quad S & \quad S' \\
\text{j+1} & \quad \text{same} & \text{e} & \text{d} & \quad \text{same} & \text{d} & \text{f} & \quad S & \quad S' \\
\end{align*}
Theorem

Farthest-in-future is an optimal eviction schedule.

Proof.

(Case3: d is not in the cache; \( S' \) evicts e; \( S \) evicts \( f \neq e \)). From request \( j + 2 \) onwards, \( S' \) should behave exactly like \( S \) until

Case3a: There is a request for \( g \neq e, f \) not in the cache of \( S \). \( S \) evicts e. Since \( S \) and \( S' \) only differ on e and f, it must be that \( g \) is not in the cache of \( S' \) either. Now, we can make \( S' \) evict f and now both caches are the same.
Optimal Cashing: analysis

Theorem
Farthest-in-future is an optimal eviction schedule.

Proof.
(Case 3: d is not in the cache; $S_{FF}$ evicts $e$; $S$ evicts $f \neq e$). From request $j + 2$ onwards, $S'$ should behave exactly like $S$ until

Case 3b: There is a request to $f$, and $S$ evicts an item $e'$. If $e' = e$, then we are all set: $S'$ can simply access $f$ from cache, and after this step both caches are the same. If $e' \neq e$, then we have $S'$ evict $e'$ and bring in $e$ to have both caches the same. But, now $S'$ is no longer a reduced schedule. We can use the previous result to convert $S'$ into a reduced schedule without incurring more misses and still agreeing with $S'$ through step $j + 1$. 
Theorem

Farthest-in-future is an optimal eviction schedule.

Proof.

(Case3: d is not in the cache; $S_{FF}$ evicts $e$; $S$ evicts $f \neq e$). From request $j + 2$ onwards, $S'$ should behave exactly like $S$ until

Case3c: $g = e$. Can’t happen with Farthest-In-Future since there must be a request for $f$ before $e$. 

□
Caching is among most fundamental online problems in CS.
Optimal Cashing: perspectives

- Caching is among most fundamental online problems in CS.
- Offline: full sequence of requests is known a priori.
Optimal Cashing: perspectives

- Caching is among most fundamental online problems in CS.
- Offline: full sequence of requests is known a priori.
  - FF is an optimal offline eviction algorithm.
Caching is among most fundamental online problems in CS.

Offline: full sequence of requests is known a priori.
  - FF is an optimal offline eviction algorithm.

Online (reality): requests are not known in advance.
Caching is among most fundamental online problems in CS.

Offline: full sequence of requests is known a priori.
- FF is an optimal offline eviction algorithm.

Online (reality): requests are not known in advance.
- variant of the Least-Recently-Used principle
Caching is among most fundamental online problems in CS.

- Offline: full sequence of requests is known a priori.
  - FF is an optimal offline eviction algorithm.

- Online (reality): requests are not known in advance.
  - variant of the *Least-Recently-Used* principle
  - evict the item that was referenced *longest ago*
Shortest path: problem

shortest path from Princeton CS department to Einstein's house
Shortest path: problem

Given:

- A directed graph \( G = (V, E) \).
- A source \( s \) and a destination \( t \).
- The length \( \ell(e) \) of each edge \( e \).

The goal is to find the shortest directed path from \( s \) to \( t \).
Shortest path: problem

Given:
- Directed graph $G = (V, E)$.
Shortest path: problem

- **Given:**
  - Directed graph $G = (V, E)$.
  - Source $s$, destination $t$. 
Shortest path: problem

- **Given:**
  - Directed graph $G = (V, E)$.
  - Source $s$, destination $t$.
  - Length $\ell_e = \text{length of edge } e$. 
Shortest path: problem

- **Given:**
  - Directed graph $G = (V, E)$.
  - Source $s$, destination $t$.
  - Length $\ell_e = \text{length of edge } e$.

- **Goal:** find shortest directed path from $s$ to $t$. 
Shortest path: example

Cost of path $s - 2 - 3 - 5 - t = 9 + 23 + 2 + 16 = 48$. 
Dijkstra's Algorithm $(G, \ell)$
Let $S$ be the set of explored nodes
For each $u \in S$, we store a distance $d(u)$
Initially $S = \{s\}$ and $d(s) = 0$
While $S \neq V$
    Select a node $v \notin S$ with at least one edge from $S$ for which
    $d'(v) = \min_{e=(u,v) : u \in S} d(u) + \ell_e$ is as small as possible
    Add $v$ to $S$ and define $d(v) = d'(v)$
EndWhile
Shortest path: greedy algorithm

**Figure 4.7** A snapshot of the execution of Dijkstra’s Algorithm. The next node that will be added to the set $S$ is $x$, due to the path through $u$. 
Shortest path: greedy algorithm
Shortest path: greedy algorithm
Shortest path: demo
Shortest path: demo
Shortest path: demo
Shortest path: demo

decrease key

distance label
Shortest path: demo

distance label  ➔  15

delmin
Shortest path: demo

decrease key
Shortest path: demo

delmin
Shortest path: demo

![Graph showing a network with nodes and edges labeled with weights. The node labeled 's' has edges to nodes 0, 2, and 6. Node 0 has an edge to node 2 with weight 9, node 2 to node 6 with weight 14, and node 6 to node 7 with weight 15. The path from node s to node 7 is highlighted with a red arrow labeled 'delmin'.]
Shortest path: demo
Shortest path: demo
Shortest path: demo
Shortest path: demo

Prof. Dr. Brahim Hnich  Algorithm Design  Greedy Algorithms (Part II)
Shortest path: demo
Shortest path: demo
Shortest path: demo
Shortest path: demo
Shortest path: demo
Theorem

For each node \( u \in S \), \( d(u) \) is the length of the shortest \( s - u \) path.

Proof.

(by induction on \( |S| \))

**Base case:** \( |S| = 1 \) is trivial.

**Inductive hypothesis:** Assume true for \( |S| = k \geq 1 \). Let \( v \) be next node added to \( S \), and let \( u - v \) be the chosen edge. The shortest \( s - u \) path plus \((u, v)\) is an \( s - v \) path of length \( d(v) \).

Consider any other \( s - v \) path \( P \). We’ll see that it’s no shorter than \( d(v) \). Let \( x - y \) be the first edge in \( P \) that leaves \( S \), and let \( P' \) be the subpath to \( x \). Since, the Dijkstra’s algorithm chose \( v \), we have that

\[
    d(v) = d(u) + \ell(u, v) \leq d(x) + \ell(x, y)
\]

which makes the length of \( P' \) already larger than that of \( P \) (see next figure).
Theorem

For each node \( u \in S \), \( d(u) \) is the length of the shortest \( s - u \) path.

\[\text{Figure 4.8} \] The shortest path \( P_v \) and an alternate \( s-v \) path \( P \) through the node \( y \).
Minimum spanning tree. Given a connected graph $G = (V, E)$ with real-valued edge weights $c_e$, an MST is a subset of the edges $T \in E$ such that $T$ is a spanning tree whose sum of edge weights is minimized.
Minimum spanning tree: example

\[ G = (V, E) \]

\[ T, \sum_{e \in T} c_e = 50 \]
Minimum spanning tree: applications

- Network design
Minimum spanning tree: applications

- Network design
  - telephone, electrical, hydraulic, TV cable, computer, road

Approximation algorithms for NP-hard problems
- traveling salesperson problem, Steiner tree

Indirect applications
- max bottleneck paths
- reducing data storage in sequencing amino acids in a protein

Cluster analysis
Minimum spanning tree: applications

- Network design
  - telephone, electrical, hydraulic, TV cable, computer, road
- Approximation algorithms for NP-hard problems
Minimum spanning tree: applications

- Network design
  - telephone, electrical, hydraulic, TV cable, computer, road
- Approximation algorithms for NP-hard problems
  - traveling salesperson problem, Steiner tree
Minimum spanning tree: applications

- Network design
  - telephone, electrical, hydraulic, TV cable, computer, road
- Approximation algorithms for NP-hard problems
  - traveling salesperson problem, Steiner tree
- Indirect applications
Minimum spanning tree: applications

- Network design
  - telephone, electrical, hydraulic, TV cable, computer, road
- Approximation algorithms for NP-hard problems
  - traveling salesperson problem, Steiner tree
- Indirect applications
  - max bottleneck paths
Minimum spanning tree: applications

- **Network design**
  - telephone, electrical, hydraulic, TV cable, computer, road
- **Approximation algorithms for NP-hard problems**
  - traveling salesperson problem, Steiner tree
- **Indirect applications**
  - max bottleneck paths
  - reducing data storage in sequencing amino acids in a protein
Minimum spanning tree: applications

- Network design
  - telephone, electrical, hydraulic, TV cable, computer, road
- Approximation algorithms for NP-hard problems
  - traveling salesperson problem, Steiner tree
- Indirect applications
  - max bottleneck paths
  - reducing data storage in sequencing amino acids in a protein
- Cluster analysis
Minimum spanning tree: greedy algorithms

Kruskal’s algorithm. Start with $T = \emptyset$. Consider edges in ascending order of cost. Insert edge $e$ in $T$ unless doing so would create a cycle.
Minimum spanning tree: greedy algorithms

**Kruskal’s algorithm.** Start with $T = \emptyset$. Consider edges in ascending order of cost. Insert edge $e$ in $T$ unless doing so would create a cycle.

**Prim’s algorithm.** Start with some root node $s$ and greedily grow a tree $T$ from $s$ outward. At each step, add the cheapest edge $e$ to $T$ that has exactly one endpoint in $T$. 
Kruskal’s algorithm. Start with $T = \emptyset$. Consider edges in ascending order of cost. Insert edge $e$ in $T$ unless doing so would create a cycle.

Prim’s algorithm. Start with some root node $s$ and greedily grow a tree $T$ from $s$ outward. At each step, add the cheapest edge $e$ to $T$ that has exactly one endpoint in $T$.

Reverse-Delete algorithm. Start with $T = E$. Consider edges in descending order of cost. Delete edge $e$ from $T$ unless doing so would disconnect $T$. 
Minimum spanning tree: greedy algorithms

Kruskal’s algorithm. Start with $T = \emptyset$. Consider edges in ascending order of cost. Insert edge $e$ in $T$ unless doing so would create a cycle.

Prim’s algorithm. Start with some root node $s$ and greedily grow a tree $T$ from $s$ outward. At each step, add the cheapest edge $e$ to $T$ that has exactly one endpoint in $T$.

Reverse-Delete algorithm. Start with $T = E$. Consider edges in descending order of cost. Delete edge $e$ from $T$ unless doing so would disconnect $T$.

Remark. All three algorithms produce an MST.
Simplifying assumption. All edge costs $c_e$ are distinct.
Simplifying assumption. All edge costs $c_e$ are distinct.

Cut property. Let $S$ be any subset of nodes, and let $e$ be the min cost edge with exactly one endpoint in $S$. Then the MST contains $e$. 
**Simplifying assumption.** All edge costs $c_e$ are distinct.

**Cut property.** Let $S$ be any subset of nodes, and let $e$ be the min cost edge with exactly one endpoint in $S$. Then the MST contains $e$.

**Cycle property.** Let $C$ be any cycle, and let $f$ be the max cost edge belonging to $C$. Then the MST does not contain $f$. 
**Simplifying assumption.** All edge costs $c_e$ are distinct.

**Cut property.** Let $S$ be any subset of nodes, and let $e$ be the min cost edge with exactly one endpoint in $S$. Then the MST contains $e$.

**Cycle property.** Let $C$ be any cycle, and let $f$ be the max cost edge belonging to $C$. Then the MST does not contain $f$. 

![Diagram](image-url)
Cycle. Set of edges the form
\[ a - b, b - c, c - d, \ldots, y - z, z - a. \]
**Cycle.** Set of edges the form
\[ a - b, b - c, c - d, \ldots, y - z, z - a. \]
Minimum spanning tree: analysis

**Cycle.** Set of edges the form
\[ a - b, b - c, c - d, \ldots, y - z, z - a. \]

**Cutset.** A cut is a subset of nodes S. The corresponding cutset \( D \) is the subset of edges with exactly one endpoint in \( S \).
**Cycle.** Set of edges the form 
\[ a - b, b - c, c - d, \ldots, y - z, z - a. \]

**Cutset.** A cut is a subset of nodes \( S \). The corresponding cutset \( D \) is the subset of edges with exactly one endpoint in \( S \).

- **Cycle \( C \):** 1-2, 2-3, 3-4, 4-5, 5-6, 6-1
- **Cut \( S \):** \{4, 5, 8\}
- **Cutset \( D \):** 5-6, 5-7, 3-4, 3-5, 7-8
Claim. A cycle and a cutset intersect in an even number of edges.
Claim. A cycle and a cutset intersect in an even number of edges.

Cycle $C = 1-2, 2-3, 3-4, 4-5, 5-6, 6-1$
Cutset $D = 3-4, 3-5, 5-6, 5-7, 7-8$
Intersection $= 3-4, 5-6$
Claim. A cycle and a cutset intersect in an even number of edges.

Proof. by picture.
Claim. A cycle and a cutset intersect in an even number of edges.

Proof. by picture.
Theorem

**Cut property.** Let $S$ be any subset of nodes, and let $e$ be the minimum cost edge with exactly one endpoint in $S$. Then the MST $T^*$ contains $e$.

Proof.

**Exchange argument.** Suppose $e$ does not belong to $T^*$, and let's see what happens. Adding $e$ to $T^*$ creates a cycle $C$ in $T^*$. Edge $e$ is both in the cycle $C$ and in the cutset $D$ corresponding to $S$. Thus, there exists another edge, say $f$, that is in both $C$ and $D$. $T' = T^* \cup \{e\} - \{f\}$ is also a spanning tree. Since $c_e < c_f$, $\text{cost}(T') < \text{cost}(T^*)$. This is a contradiction. \qed
Theorem

**Cycle property.** Let $C$ be any cycle in $G$, and let $f$ be the max cost edge belonging to $C$. Then the MST $T^*$ does not contain $f$.

**Proof.**

**Exchange argument.** Suppose $f$ belongs to $T^*$, and let’s see what happens. Deleting $f$ from $T^*$ creates a cut $S$ in $T^*$. Edge $f$ is both in the cycle $C$ and in the cutset $D$ corresponding to $S$. Thus, there exists another edge, say $e$, that is in both $C$ and $D$. $T' = T^* \cup \{e\} - \{f\}$ is also a spanning tree. Since $c_e < c_f$, $\text{cost}(T') < \text{cost}(T^*)$. This is a contradiction.
Claim. Krushal’s algorithm produces a MST.
Claim. Krushal’s algorithm produces a MST.
Claim. Prim’s algorithm produces a MST.
Claim. Krushal’s algorithm produces a MST.

Claim. Prim’s algorithm produces a MST.

Proof(Sketch). Both algorithms only include an edge when it is justified by the cut property.
Minimum spanning tree: Correctness of Krushal’s algorithm

- Consider edges in ascending order of weight.
Minimum spanning tree: Correctness of Krushal’s algorithm

- Consider edges in ascending order of weight.
- Case 1: If adding $e$ to $T$ creates a cycle, discard $e$ according to cycle property.
Minimum spanning tree: Correctness of Krushal’s algorithm

- Consider edges in ascending order of weight.
- Case 1: If adding $e$ to $T$ creates a cycle, discard $e$ according to cycle property.
- Case 2: Otherwise, insert $e = (u, v)$ into $T$ according to cut property where $S =$ set of nodes in $u$’s connected component.
Minimum spanning tree: Correctness of Krushal’s algorithm

- Consider edges in ascending order of weight.
- Case 1: If adding $e$ to $T$ creates a cycle, discard $e$ according to cycle property.
- Case 2: Otherwise, insert $e = (u, v)$ into $T$ according to cut property where $S =$ set of nodes in $u$’s connected component.
1. Initialize $S = $ any node
Minimum spanning tree: Correctness of Prim’s algorithm

1. Initialize $S = \text{any node}$
2. Apply cut property to $S$. 
1. Initialize $S = \text{any node}$
2. Apply cut property to $S$.
   2.1 Add min cost edge in cutset corresponding to $S$ to $T$
Minimum spanning tree: Correctness of Prim’s algorithm

1. Initialize $S = \text{any node}$
2. Apply cut property to $S$.
   2.1 Add min cost edge in cutset corresponding to $S$ to $T$
   2.2 Add one new explored node $u$ to $S$
Minimum spanning tree: Correctness of Prim’s algorithm

1. Initialize $S = \text{any node}$

2. Apply cut property to $S$.
   
   2.1 Add min cost edge in cutset corresponding to $S$ to $T$
   
   2.2 Add one new explored node $u$ to $S$

3. 

Diagram:

- $S$
Outbreak of cholera deaths in London in 1850s.
Reference: Nina Mishra, HP Labs
Clustering: definition

**Clustering.** Given a set $U$ of $n$ objects labeled $p_1, \ldots, p_n$, classify into coherent groups.
Clustering. Given a set $U$ of $n$ objects labeled $p_1, \ldots, p_n$, classify into coherent groups.

Distance function. Numeric value specifying "closeness" of two objects.
Clustering. Given a set $U$ of $n$ objects labeled $p_1, \ldots, p_n$, classify into coherent groups.

Distance function. Numeric value specifying "closeness" of two objects.

Fundamental problem. Divide into clusters so that points in different clusters are far apart.
Clustering: definition

Clustering. Given a set $U$ of $n$ objects labeled $p_1, \ldots, p_n$, classify into coherent groups.

Distance function. Numeric value specifying "closeness" of two objects.

Fundamental problem. Divide into clusters so that points in different clusters are far apart.
  ▶ Routing in mobile ad hoc networks.
Clustering. Given a set $U$ of $n$ objects labeled $p_1, \ldots, p_n$, classify into coherent groups.

Distance function. Numeric value specifying "closeness" of two objects.

Fundamental problem. Divide into clusters so that points in different clusters are far apart.

- Routing in mobile ad hoc networks.
- Identify patterns in gene expression.

Document categorization for web search.

Similarity searching in medical image databases.

Skycat: cluster 10 sky objects into stars, quasars, galaxies.
Clustering. Given a set $U$ of $n$ objects labeled $p_1, \ldots, p_n$, classify into coherent groups.

Distance function. Numeric value specifying "closeness" of two objects.

Fundamental problem. Divide into clusters so that points in different clusters are far apart.

- Routing in mobile ad hoc networks.
- Identify patterns in gene expression.
- Document categorization for web search.
Clustering: definition

Clustering. Given a set $U$ of $n$ objects labeled $p_1, \ldots, p_n$, classify into coherent groups.

Distance function. Numeric value specifying "closeness" of two objects.

Fundamental problem. Divide into clusters so that points in different clusters are far apart.

- Routing in mobile ad hoc networks.
- Identify patterns in gene expression.
- Document categorization for web search.
- Similarity searching in medical image databases
Clustering. Given a set $U$ of $n$ objects labeled $p_1, \ldots, p_n$, classify into coherent groups.

Distance function. Numeric value specifying "closeness" of two objects.

Fundamental problem. Divide into clusters so that points in different clusters are far apart.

- Routing in mobile ad hoc networks.
- Identify patterns in gene expression.
- Document categorization for web search.
- Similarity searching in medical image databases
- Skycat: cluster $10^9$ sky objects into stars, quasars, galaxies.
Clustering of maximum spacing: definition

$k$-clustering. Divide objects into $k$ non-empty groups.
Clustering of maximum spacing: definition

\emph{k-clustering}. Divide objects into \( k \) non-empty groups.

\emph{Distance function}. Assume it satisfies several natural properties.

- \( d(p_i, p_j) = 0 \) iff \( p_i = p_j \) (identity of indiscernibles)
- \( d(p_i, p_j) \geq 0 \) (nonnegativity)
- \( d(p_i, p_j) = d(p_j, p_i) \) (symmetry)

\textbf{Spacing.} Min distance between any pair of points lying in different clusters.

Clustering of maximum spacing. Given an integer \( k \), find a \( k \)-clustering of maximum spacing.
**Clustering of maximum spacing: definition**

**k-clustering.** Divide objects into \( k \) non-empty groups.

**Distance function.** Assume it satisfies several natural properties.

- \( d(p_i, p_j) = 0 \) iff \( p_i = p_j \) (identity of indiscernibles)
- \( d(p_i, p_j) \geq 0 \) (nonnegativity)
- \( d(p_i, p_j) = d(p_j, p_i) \) (symmetry)

**Spacing.** Min distance between any pair of points lying in different clusters.

**Clustering of maximum spacing.** Given an integer \( k \), find a \( k \)-clustering of maximum spacing.
Clustering of maximum spacing: definition

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

**Distance function.** Assume it satisfies several natural properties.

- \( d(p_i, p_j) = 0 \) iff \( p_i = p_j \) (identity of indiscernibles)
- \( d(p_i, p_j) \geq 0 \) (nonnegativity)

Spacing. Min distance between any pair of points lying in different clusters.

Clustering of maximum spacing. Given an integer *k*, find a *k*-clustering of maximum spacing.
Clustering of maximum spacing: definition

**k-clustering.** Divide objects into \( k \) non-empty groups.

**Distance function.** Assume it satisfies several natural properties.

- \( d(p_i, p_j) = 0 \) iff \( p_i = p_j \) (identity of indiscernibles)
- \( d(p_i, p_j) \geq 0 \) (nonnegativity)
- \( d(p_i, p_j) = d(p_j, p_i) \) (symmetry)
Clustering of maximum spacing: definition

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

**Distance function.** Assume it satisfies several natural properties.

- $d(p_i, p_j) = 0$ iff $p_i = p_j$ (identity of indiscernibles)
- $d(p_i, p_j) \geq 0$ (nonnegativity)
- $d(p_i, p_j) = d(p_j, p_i)$ (symmetry)

**Spacing.** Min distance between any pair of points lying in different clusters.
**Clustering of maximum spacing: definition**

**k-clustering.** Divide objects into \( k \) non-empty groups.

**Distance function.** Assume it satisfies several natural properties.

- \( d(p_i, p_j) = 0 \) iff \( p_i = p_j \) (identity of indiscernibles)
- \( d(p_i, p_j) \geq 0 \) (nonnegativity)
- \( d(p_i, p_j) = d(p_j, p_i) \) (symmetry)

**Spacing.** Min distance between any pair of points lying in different clusters.

**Clustering of maximum spacing.** Given an integer \( k \), find a \( k \)-clustering of maximum spacing.
Clustering of maximum spacing: definition

\textit{k-clustering.} Divide objects into \textit{k} non-empty groups.

\textit{Distance function.} Assume it satisfies several natural properties.

\begin{itemize}
\item $d(p_i, p_j) = 0$ iff $p_i = p_j$ (identity of indiscernibles)
\item $d(p_i, p_j) \geq 0$ (nonnegativity)
\item $d(p_i, p_j) = d(p_j, p_i)$ (symmetry)
\end{itemize}

\textit{Spacing.} Min distance between any pair of points lying in different clusters.

\textbf{Clustering of maximum spacing.} Given an integer \textit{k}, find a \textit{k}-clustering of maximum spacing.
Figure 4.14 An example of single-linkage clustering with $k = 3$ clusters. The clusters are formed by adding edges between points in order of increasing distance.
Clustering of maximum spacing: greedy algorithm

Single-link k-clustering algorithm. steps:

1. Form a graph on the vertex set $U$, corresponding to $n$ clusters.
2. Find the closest pair of objects such that each object is in a different cluster, and add an edge between them.
3. Repeat $n-k$ times until there are exactly $k$ clusters.

Key observation. This procedure is precisely Kruskal's algorithm (except we stop when there are $k$ connected components).

Remark. Equivalent to finding an MST and deleting the $k-1$ most expensive edges.
Clustering of maximum spacing: greedy algorithm

Single-link k-clustering algorithm. steps:

▶ Form a graph on the vertex set $U$, corresponding to $n$ clusters.
Clustering of maximum spacing: greedy algorithm

Single-link k-clustering algorithm. steps:

- Form a graph on the vertex 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.

Key observation. This procedure is precisely Kruskal's algorithm (except we stop when there are $k$ connected components).

Remark. Equivalent to finding an MST and deleting the $k - 1$ most expensive edges.
Clustering of maximum spacing: greedy algorithm

Single-link $k$-clustering algorithm. steps:

- Form a graph on the vertex 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.
Single-link k-clustering algorithm. steps:

▶ Form a graph on the vertex 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.

Key observation. This procedure is precisely Kruskal’s algorithm (except we stop when there are $k$ connected components).
Clustering of maximum spacing: greedy algorithm

Single-link $k$-clustering algorithm. steps:

- Form a graph on the vertex 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.

Key observation. This procedure is precisely Kruskal’s algorithm (except we stop when there are $k$ connected components).

Remark. Equivalent to finding an MST and deleting the $k - 1$ most expensive edges.
Clustering of maximum spacing: analysis

Theorem
Let $C$ denote the clustering $C_1, \ldots, C_k$ formed by deleting the $k-1$ most expensive edges of a MST. $C$ 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 $(k-1)^{st}$ most expensive edge. Let $p_i, p_j$ be in the same cluster in $C$, say $C_r$, but different clusters in $C'$, say $C'_s$ and $C'_t$. Some edge $(p, p')$ on $p_i - p_j$ path in $C_r$ spans two different clusters in $C'$. All edges on $p_i - p_j$ path have length $\leq d^*$ since Kruskal chose them. Spacing of $C'$ is $\leq d^*$ since $p$ and $p'$ are in different clusters. \qed
Figure 4.15 An illustration of the proof of (4.26), showing that the spacing of any other clustering can be no larger than that of the clustering found by the single-linkage algorithm.
Definition. Reduce size of data. I.e., number of bits needed to represent data
Data compression: definition

**Definition.** Reduce size of data. I.e., number of bits needed to represent data

**Benefits.** Reduce storage needed; Reduce transmission cost or latency or bandwidth
Data compression: Huffman codes

**Definition.** Use variable length codes based on frequency.
Definition. Use variable length codes based on frequency.

Approach. as follows.
Data compression: Huffman codes

Definition. Use variable length codes based on frequency

Approach. as follows.

▶ Variable length encoding of symbols
Definition. Use variable length codes based on frequency

Approach. as follows.

- Variable length encoding of symbols
- Exploit statistical frequency of symbols
Definition. Use variable length codes based on frequency
Approach. as follows.
  ▶ Variable length encoding of symbols
  ▶ Exploit statistical frequency of symbols
  ▶ Efficient when symbol probabilities vary widely
Data compression: Huffman codes

**Definition.** Use variable length codes based on frequency

**Approach.** as follows.

- Variable length encoding of symbols
- Exploit statistical frequency of symbols
- Efficient when symbol probabilities vary widely

**Principle.** Use fewer bits to represent frequent symbols; Use more bits to represent infrequent symbols
Huffman codes: example

<table>
<thead>
<tr>
<th>Symbol</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency</td>
<td>13%</td>
<td>25%</td>
<td>50%</td>
<td>12%</td>
</tr>
<tr>
<td>Original</td>
<td>00</td>
<td>01</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>Encoding</td>
<td>2 bits</td>
<td>2 bits</td>
<td>2 bits</td>
<td>2 bits</td>
</tr>
<tr>
<td>Huffman</td>
<td>110</td>
<td>10</td>
<td>0</td>
<td>111</td>
</tr>
<tr>
<td>Encoding</td>
<td>3 bits</td>
<td>2 bits</td>
<td>1 bit</td>
<td>3 bits</td>
</tr>
</tbody>
</table>
Huffman codes: example

<table>
<thead>
<tr>
<th>Symbol</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency</td>
<td>13%</td>
<td>25%</td>
<td>50%</td>
<td>12%</td>
</tr>
<tr>
<td>Original Encoding</td>
<td>00</td>
<td>01</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>2 bits</td>
<td>2 bits</td>
<td>2 bits</td>
<td>2 bits</td>
</tr>
<tr>
<td>Huffman Encoding</td>
<td>110</td>
<td>10</td>
<td>0</td>
<td>111</td>
</tr>
<tr>
<td></td>
<td>3 bits</td>
<td>2 bits</td>
<td>1 bit</td>
<td>3 bits</td>
</tr>
</tbody>
</table>

Expected size:
### Huffman codes: example

<table>
<thead>
<tr>
<th>Symbol</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency</td>
<td>13%</td>
<td>25%</td>
<td>50%</td>
<td>12%</td>
</tr>
<tr>
<td>Original Encoding</td>
<td>00</td>
<td>01</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>Huffman Encoding</td>
<td>110</td>
<td>10</td>
<td>0</td>
<td>111</td>
</tr>
</tbody>
</table>

#### Expected size:
- Original: \( \frac{1}{8} \times 2 + \frac{1}{4} \times 2 + \frac{1}{2} \times 2 + \frac{1}{8} \times 2 = 2 \) bits per symbol
Huffman codes: example

<table>
<thead>
<tr>
<th>Symbol</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency</td>
<td>13%</td>
<td>25%</td>
<td>50%</td>
<td>12%</td>
</tr>
<tr>
<td>Original Encoding</td>
<td>00</td>
<td>01</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>Huffman Encoding</td>
<td>110</td>
<td>10</td>
<td>0</td>
<td>111</td>
</tr>
<tr>
<td>Expected size:</td>
<td>3 bits</td>
<td>2 bits</td>
<td>1 bit</td>
<td>3 bits</td>
</tr>
</tbody>
</table>

Original: $1/8 \times 2 + 1/4 \times 2 + 1/2 \times 2 + 1/8 \times 2 = 2$ bits per symbol

Huffman: $1/8 \times 3 + 1/4 \times 2 + 1/2 \times 1 + 1/8 \times 3 = 1.75$ bits per symbol
Binary (Huffman) tree. Represents Huffman code:

- Edge: code (0 or 1)
- Leaf: symbol
- Path to leaf: encoding
Huffman codes: binary tree

Binary (Huffman) tree. Represents Huffman code:

- Edge: code (0 or 1)
- Leaf: symbol
- Path to leaf: encoding
- Example: A = 110, B = 10, C = 0
To construct a prefix code for an alphabet $S$, with given frequencies:

If $S$ has two letters then
    Encode one letter using 0 and the other letter using 1
Else
    Let $y^*$ and $z^*$ be the two lowest-frequency letters
    Form a new alphabet $S'$ by deleting $y^*$ and $z^*$ and
    replacing them with a new letter $\omega$ of frequency $f_{y^*} + f_{z^*}$
    Recursively construct a prefix code $\gamma'$ for $S'$, with tree $T'$
Define a prefix code for $S$ as follows:
    Start with $T'$
    Take the leaf labeled $\omega$ and add two children below it
        labeled $y^*$ and $z^*$
Endif
Huffman codes: greedy algorithm

**Figure 4.17** There is an optimal solution in which the two lowest-frequency letters label sibling leaves; deleting them and labeling their parent with a new letter having the combined frequency yields an instance with a smaller alphabet.
Huffman codes: greedy algorithm

A: 3
C: 5
E: 8
H: 2
I: 7
Huffman codes: greedy algorithm
Huffman codes: greedy algorithm
Huffman codes: greedy algorithm

Diagram showing a Huffman tree with characters A, H, C, E, I and their frequencies 3, 2, 5, 8, 7, 15.
Huffman codes: greedy algorithm

\[ \begin{align*}
E &= 01 \\
I &= 00 \\
C &= 10 \\
A &= 111 \\
H &= 110
\end{align*} \]
Huffman codes: properties

Prefix code properties:

▶ No code is a prefix of another code
▶ Can stop as soon as complete code found
▶ No need for end-of-code marker
Prefix code. properties:

▶ No code is a prefix of another code
Prefix code properties:

- No code is a prefix of another code
- Can stop as soon as complete code found
Prefix code properties:

- No code is a prefix of another code
- Can stop as soon as complete code found
- No need for end-of-code marker
Theorem

*The huffman code for a given alphabet achieves the minimum average number of bits per letter of any prefix code*