CSE 202: Design and Analysis of Algorithms

Lecture 2
Greedy Algorithms

- Minimum Spanning Trees
- The Union/Find Data Structure
A Network Design Problem

**Problem:** Given distances between a set of computers, find the cheapest set of pairwise connections so that they are all connected.

**Graph-Theoretic Formulation:**

- Node = Computer
- Edge = Pair of computers
- Edge Cost(u,v) = Distance(u,v)

Find a subset of edges T such that the cost of T is minimum and all nodes are connected in (V,T)

Can T contain a cycle?
Solution is connected and acyclic, so a **tree**
A connected, undirected and acyclic graph is called a tree

Property 1. A tree on n nodes has exactly n - 1 edges
A connected, undirected and acyclic graph is called a tree

Property 1. A tree on n nodes has exactly n - 1 edges

Proof. By induction.

Base Case:
n nodes, no edges,
n connected components

Inductive Case:
Add edge between two connected components
No cycle created
#components decreases by 1

At the end: 1 component
How many edges were added?
Trees

A connected, undirected and acyclic graph is called a tree

Property 1. A tree on $n$ nodes has exactly $n - 1$ edges

Is any graph on $n$ nodes and $n - 1$ edges a tree?

Property 2. Any connected, undirected graph on $n$ nodes and $n - 1$ edges is a tree
A connected, undirected and acyclic graph is called a tree

**Property 1.** A tree on $n$ nodes has exactly $n - 1$ edges

**Property 2.** Any connected, undirected graph on $n$ nodes and $n - 1$ edges is a tree

**Proof:** Suppose $G$ is connected, undirected, has some cycles. While $G$ has a cycle, remove an edge from this cycle. Result: $G' = (V, E')$ where $E'$ is a tree. So $|E'| = n - 1$

Thus, $E = E'$, and $G$ is a tree
Minimum Spanning Trees (MST)

**Problem:** Given distances between a set of computers, find the cheapest set of pairwise connections so that they are all connected.

**Graph-Theoretic Formulation:**
- Node = Computer
- Edge = Pair of computers
- Edge Cost\((u,v)\) = Distance\((u,v)\)

Find a subset of edges \(T\) such that the cost of \(T\) is minimum and all nodes are connected in \((V,T)\)

**Goal:** Find a spanning tree \(T\) of the graph \(G\) with minimum total cost

We’ll see a greedy algorithm to construct \(T\)
Properties of MSTs

For a cut \((S, V\setminus S)\), the lightest edge in the cut is the minimum cost edge that has one end in \(S\) and the other in \(V\setminus S\).

Property 1. A lightest edge in any cut always belongs to an MST

Proof. Suppose not.

Let \(e = \) lightest edge in \((S, V\setminus S)\), \(T = \) MST, \(e\) is not in \(T\)
\(T \cup \{e\}\) has a cycle with edge \(e'\) across \((S, V\setminus S)\)
Let \(T' = T \setminus \{e'\} \cup \{e\}\)
\(\text{cost}(T') = \text{cost}(T) + \text{cost}(e) - \text{cost}(e') < \text{cost}(T)\)
Properties of MSTs

The heaviest edge in a cycle is the maximum cost edge in the cycle.

**Property 2.** The heaviest edge in a cycle never belongs to an MST.

**Proof.** Suppose not. Let $T = \text{MST}$, $e = \text{heaviest edge in some cycle, } e \in T$.

Delete $e$ from $T$ to get subtrees $T_1$ and $T_2$.

Let $e' = \text{lightest edge in the cut } (T_1, V \setminus T_1)$.

Then, $\text{cost}(e') < \text{cost}(e)$.

Let $T' = T \setminus \{e\} + \{e'\}$.

$\text{cost}(T') = \text{cost}(T) + \text{cost}(e) - \text{cost}(e') < \text{cost}(T)$.
Summary: Properties of MSTs

Property 1. A lightest edge in any cut always belongs to an MST

Property 2. The heaviest edge in a cycle never belongs to an MST
A Generic MST Algorithm

\[ X = \{ \} \]

While there is a cut \((S, V \setminus S)\) s.t. \(X\) has no edges across it
\[ X = X + \{e\}, \text{ where } e \text{ is the lightest edge across } (S, V \setminus S) \]

Does this output a tree?
- At each step, no cycle is created
- Continues while there are disconnected components

Why does this produce a MST?
A Generic MST Algorithm

Proof of correctness by induction.
Base Case: At \( t=0 \), \( X \) is in some MST \( T \)

Induction: Assume at \( t=k \), \( X \) is in some MST \( T \)
Suppose we add \( e \) to \( X \) at \( t=k+1 \)
Suppose \( e \) is not in \( T \). Adding \( e \) to \( T \) forms a cycle \( C \)
Let \( e' \) = another edge in \( C \) across \((S, V\backslash S)\), \( T' = T \backslash \{e'\} \cup \{e\} \)
\[ \text{cost}(T') = \text{cost}(T) - \text{cost}(e') + \text{cost}(e) \leq \text{cost}(T) \]
Thus, \( T' \) is a MST that contains \( X \)
Kruskal’s Algorithm

\( X = \{ \} \)
For each edge \( e \) in increasing order of weight:
   If the end-points of \( e \) lie in different components in \( X \),
   Add \( e \) to \( X \)

Why does this work correctly?

Efficient Implementation: Need a data structure with properties:
   - Maintain disjoint sets of nodes
   - Merge sets of nodes (union)
   - Find if two nodes are in the same set (find)

The Union-Find data structure
The Union-Find Data Structure

procedure makeset(x)
    p[x] = x
    rank[x] = 0

procedure find(x)
    if x \neq p[x]:
        p[x] = find(p[x])
    return p[x]

procedure union(x,y)
    rootx = find(x)
    rooty = find(y)
    if rootx = rooty: return
    if rank[rootx] > rank[rooty]:
        p[rooty] = rootx
    else:
        p[rootx] = rooty
    if rank[rootx] = rank[rooty]:
        rank[rooty]++
The Union-Find Data Structure

**procedure makeset(x)**

\[ p[x] = x \]
\[ \text{rank}[x] = 0 \]

**procedure find(x)**

\[ \text{if } x \neq p[x]: \]
\[ \quad p[x] = \text{find}(p[x]) \]
\[ \text{return } p[x] \]

**procedure union(x, y)**

\[ \text{root}_x = \text{find}(x) \]
\[ \text{root}_y = \text{find}(y) \]
\[ \text{if } \text{root}_x = \text{root}_y: \text{return} \]
\[ \quad \text{if } \text{rank}[	ext{root}_x] > \text{rank}[	ext{root}_y]: \]
\[ \quad \quad p[\text{root}_y] = \text{root}_x \]
\[ \quad \text{else:} \]
\[ \quad \quad p[\text{root}_x] = \text{root}_y \]
\[ \quad \quad \text{if } \text{rank}[	ext{root}_x] = \text{rank}[	ext{root}_y]: \]
\[ \quad \quad \quad \text{rank}[	ext{root}_y]++ \]
The Union-Find Data Structure

makeset(a), ..., makeset(h)
union(a, b), union(c, d), union(e, f), union(g, h), union(f, g), union(b, c), union(h, d), find(e)

procedure makeset(x)
p[x] = x
rank[x] = 0

procedure find(x)
if x ≠ p[x]:
    p[x] = find(p[x])
return p[x]

procedure union(x, y)
rootx = find(x)
rooty = find(y)
if rootx = rooty: return
if rank[rootx] > rank[rooty]:
    p[rooty] = rootx
else:
    p[rootx] = rooty
    if rank[rootx] = rank[rooty]:
        rank[rooty]++
The Union-Find Data Structure

**procedure** makeset(x)

\[ p[x] = x \]
\[ \text{rank}[x] = 0 \]

**procedure** find(x)

if \( x \neq p[x] \):
    \[ p[x] = \text{find}(p[x]) \]
return \( p[x] \)

**procedure** union(x, y)

rootx = \text{find}(x)
rooty = \text{find}(y)

if rootx = rooty: return
if rank[rootx] > rank[rooty]:
    \[ p[rooty] = \text{rootx} \]
else:
    \[ p[rootx] = \text{rooty} \]
    if rank[rootx] = rank[rooty]:
        rank[rooty]++

makeset(a), ..., makeset(h)
union(a, b), union(c, d), union(e, f), union(g, h),
union(f, g), union(b, c), union(h, d), find(e)
**The Union-Find Data Structure**

**procedure makeset(x)**

p[x] = x  
rank[x] = 0

**procedure find(x)**

if x ≠ p[x]:
    p[x] = find(p[x])
return p[x]

**procedure union(x,y)**

rootx = find(x)  
rooty = find(y)
if rootx = rooty: return
if rank[rootx] > rank[rooty]:
    p[rooty] = rootx
else:
    p[rootx] = rooty
    if rank[rootx] = rank[rooty]:
        rank[rooty]++
The Union-Find Data Structure

procedure makeset(x)
  p[x] = x
  rank[x] = 0

procedure find(x)
  if x ≠ p[x]:
    p[x] = find(p[x])
  return p[x]

procedure union(x,y)
  rootx = find(x)
  rooty = find(y)
  if rootx = rooty: return
  if rank[rootx] > rank[rooty]:
    p[rooty] = rootx
  else:
    p[rootx] = rooty
  if rank[rootx] = rank[rooty]:
    rank[rooty]++

makeset(a), ..., makeset(h)
union(a, b), union(c, d), union(e, f), union(g, h),
union(f, g), union(b, c), union(h, d), find(e)
The Union-Find Data Structure

**procedure** makeset(x)

\[
p[x] = x \\
\text{rank}[x] = 0
\]

**procedure** find(x)

\[
\text{if } x \neq p[x]: \\
p[x] = \text{find}(p[x]) \\
\text{return } p[x]
\]

**procedure** union(x, y)

\[
\text{root}_x = \text{find}(x) \\
\text{root}_y = \text{find}(y) \\
\text{if } \text{root}_x = \text{root}_y: \text{return} \\
\text{if } \text{rank}[	ext{root}_x] > \text{rank}[	ext{root}_y]: \\
\quad \text{p[root}_y] = \text{root}_x \\
\text{else:} \\
\quad \text{p[root}_x] = \text{root}_y \\
\text{if } \text{rank}[	ext{root}_x] = \text{rank}[	ext{root}_y]: \\
\quad \text{rank}[	ext{root}_y]++
\]

makeset(a), ..., makeset(h)
union(a, b), union(c, d), union(e, f), union(g, h),
union(f, g), union(b, c), union(h, d), find(e)
The Union-Find Data Structure

procedure makeset(x)
p[x] = x
rank[x] = 0

procedure find(x)
if x ≠ p[x]:
    p[x] = find(p[x])
return p[x]

procedure union(x, y)
rootx = find(x)
rooty = find(y)
if rootx = rooty: return
if rank[rootx] > rank[rooty]:
    p[rooty] = rootx
else:
    p[rootx] = rooty
    if rank[rootx] = rank[rooty]:
        rank[rooty]++

makeset(a), ..., makeset(h)
union(a, b), union(c, d), union(e, f), union(g, h),
union(f, g), union(b, c), union(h, d), find(e)
The Union-Find Data Structure

**Fact 1:** Total time for \( m \) find operations = \( O((m+n) \log^* n) \)

**Fact 2:** Time for each union operation = \( O(1) + \text{Time(find)} \)

**Fact 3:** Total time for \( m \) find and \( n \) union ops = \( O((m+n)\log^* n) \)

\[
\log^* n = \min\{ k \mid \log \log \ldots (k \text{ times}) n = 1 \}
\]
The Union-Find Data Structure

Property 1: If x is not a root, then rank[p[x]] > rank[x]

Proof: By property of union

Property 2: For root x, if rank[x] = k, then subtree at x has size >= 2^k

Proof: By induction

Property 3: There are at most n/2^k nodes of rank k

Proof: Combining properties 1 and 2

procedure makeset(x)
    p[x] = x
    rank[x] = 0

procedure find(x)
    if x ≠ p[x]:
        p[x] = find(p[x])
    return p[x]

procedure union(x,y)
    rootx = find(x)
    rooty = find(y)
    if rootx = rooty: return
    if rank[rootx] > rank[rooty]:
        p[rooty] = rootx
    else:
        p[rootx] = rooty
    if rank[rootx] = rank[rooty]:
        rank[rooty]++
The Union-Find Data Structure

**Property 1:** If x is not a root, then
\[ \text{rank}[p[x]] > \text{rank}[x] \]

**Property 2:** For root x, if rank[x] = k, then subtree at x has size \( \geq 2^k \)

**Property 3:** There are at most \( \frac{n}{2^k} \) nodes of rank k

Interval \( I_k = [k+1, k+2, \ldots, 2^k] \)

Break up 1..n into intervals \( I_k = [k+1, k+2, \ldots, 2^k] \)

**Example:** [1], [2], [3, 4], [5,..,16], [17,..,65536],...

How many such intervals? \( \log^*n \)

**Charging Scheme:** For non-root x, if rank[x] is in \( I_k \), set \( t(x) = 2^k \)
Running time of m find operations

**Property 1:** If x is not a root, then rank[p[x]] > rank[x]

**Property 2:** For root x, if rank[x] = k, then subtree at x has size >= 2^k

**Property 3:** There are at most n/2^k nodes of rank k

Two types of nodes in a find operation:

1. rank[x], rank[p[x]] lie in different intervals
2. rank[x], rank[p[x]] lie in same interval

When a **type 2** node is touched, its parent has higher rank

Time on a **type 2** node before it becomes **type 1** <= 2^k
Running time of m find operations

**Property 1:** If \( x \) is not a root, then
\[
\text{rank}[p[x]] > \text{rank}[x]
\]

**Property 2:** For root \( x \), if \( \text{rank}[x] = k \), then subtree at \( x \) has size \( \geq 2^k \)

**Property 3:** There are at most \( n/2^k \) nodes of rank \( k \)

Total time on m find operations <= m log*\( n+\sum t(x) \)

---

Two types of nodes in a find operation:

1. \( \text{rank}[x], \text{rank}[p[x]] \) lie in different intervals
2. \( \text{rank}[x], \text{rank}[p[x]] \) lie in same interval

Interval \( I_k = [k+1, k+2, ..., 2^k] \)

#intervals = log*\( n \)

Total time on type 1 nodes <= m log*\( n \)
Total time on type 2 node \( x \) <= \( t(x) = 2^k \)
**The Union-Find Data Structure**

**Property 1:** If \( x \) is not a root, then \( \text{rank}[\text{p}[x]] > \text{rank}[x] \)

**Property 2:** For root \( x \), if \( \text{rank}[x] = k \), then subtree at \( x \) has size \( \geq 2^k \)

**Property 3:** There are at most \( n/2^k \) nodes of rank \( k \)

Interval \( I_k = [k+1, k+2, \ldots, 2^k] \)  

Number of intervals = \( \log^*n \)

Break up \( 1 \ldots n \) into intervals \( I_k = [k+1, k+2, \ldots, 2^k] \)

**Charging Scheme:** If \( \text{rank}[x] \) is in \( I_k \), set \( t(x) = 2^k \)

Total time on \( m \) find operations \( \leq m \log^*n + \sum t(x) \)

Therefore, we need to estimate \( \sum t(x) \)
The Union-Find Data Structure

**Property 1:** If $x$ is not a root, then $\text{rank}[p[x]] > \text{rank}[x]$

**Property 2:** For root $x$, if $\text{rank}[x] = k$, then subtree at $x$ has size $\geq 2^k$

**Property 3:** There are at most $\frac{n}{2^k}$ nodes of rank $k$

Interval $I_k = [k+1, k+2, \ldots, 2^k]$  

- #intervals $= \log^* n$

Break up $1..n$ into intervals $I_k = [k+1, k+2, \ldots, 2^k]$

**Charging Scheme:** If $\text{rank}[x]$ is in $I_k$, set $t(x) = 2^k$

Total time on $m$ find operations $\leq m\log^* n + \sum t(x)$

From **Property 3**, #nodes with rank in $I_k$ is at most:

$n/2^{k+1} + n/2^{k+2} + \ldots < n/2^k$

Therefore, for each interval $I_k$, $\sum_{x \text{ in } I_k} t(x) \leq n$

As #intervals $= \log^* n$, $\sum t(x) \leq n \log^* n$
procedure makeset(x)
  p[x] = x
  rank[x] = 0

procedure find(x)
  if x ≠ p[x]:
    p[x] = find(p[x])
  return p[x]

procedure union(x, y)
  rootx = find(x)
  rooty = find(y)
  if rootx = rooty: return
  if rank[rootx] > rank[rooty]:
    p[rooty] = rootx
  else:
    p[rootx] = rooty
  if rank[rootx] = rank[rooty]:
    rank[rooty]++

Property 1: Total time for m find operations = O((m+n) log^* n)

Property 2: Time for each union operation = O(1) + Time(find)
Summary: Kruskal’s Algorithm
Running Time

X = {}
For each edge e in increasing order of weight:
    If the end-points of e lie in different components in X,
    Add e to X

Sort the edges = O(m log m) = O(m log n)
Add e to X = Union Operation = O(1) + Time(Find)
Check if end-points of e lie in different components = Find Operation

Total time = Sort + O(n) Unions + O(m) Finds = O(m log n)
With sorted edges, time = O(n) Unions + O(m) Finds = O(m log* n)
MST Algorithms

- Kruskal’s Algorithm: Union-Find Data Structure
- Prim’s Algorithm: How to Implement?
Prim’s Algorithm

$X = \{ \}, S = \{ r \}$
Repeat until $S$ has $n$ nodes:
  - Pick the **lightest** edge $e$ in the cut $(S, V - S)$
  - Add $e$ to $X$
  - Add $v$, the end-point of $e$ in $V - S$ to $S$
Prim’s Algorithm

\[ X = \emptyset, S = \{r\} \]
Repeat until S has n nodes:
- Pick the lightest edge \( e \) in the cut \((S, V - S)\)
- Add \( e \) to \( X \)
- Add \( v \), the end-point of \( e \) in \( V - S \) to \( S \)

How to implement Prim’s algorithm?

Need data structure for edges with the operations:
1. **Add** an edge
2. **Delete** an edge
3. **Report** the edge with min weight
Data Structure: Heap

Heap Property: If $x$ is the parent of $y$, then $\text{key}(x) \leq \text{key}(y)$

A heap is stored as a balanced binary tree

Height = $O(\log n)$, where $n = \# \text{ nodes}$
Heap: Reporting the min

Heap Property: If x is the parent of y, then key(x) <= key(y)
**Heap: Reporting the min**

Heap Property: If $x$ is the parent of $y$, then $\text{key}(x) \leq \text{key}(y)$

Report the root node

Time = $O(1)$
Heap Property: If $x$ is the parent of $y$, then $\text{key}(x) \leq \text{key}(y)$

Add item $u$ to the end of the heap

If heap property is violated, swap $u$ with its parent
Heap: Add an item

Heap Property: If $x$ is the parent of $y$, then $\text{key}(x) \leq \text{key}(y)$

Add item $u$ to the end of the heap

If heap property is violated, swap $u$ with its parent
Heap Property: If x is the parent of y, then key(x) <= key(y)

Add item u to the end of the heap
If heap property is violated, swap u with its parent
Heap: Add an item

Heap Property: If x is the parent of y, then key(x) ≤ key(y)

Add item u to the end of the heap
If heap property is violated, swap u with its parent
Heap: Add an item

Heap Property: If x is the parent of y, then key(x) <= key(y)

Add item u to the end of the heap
If heap property is violated, swap u with its parent
Heap: Add an item

Heap Property: If x is the parent of y, then key(x) <= key(y)

Add item u to the end of the heap
If heap property is violated, swap u with its parent

Time = O(log n)
Heap: Delete an item

Heap Property: If $x$ is the parent of $y$, then $\text{key}(x) \leq \text{key}(y)$

Delete item $u$

Move $v$, the last item to $u$'s position
Heap Property: If x is the parent of y, then key(x) \leq key(y)

If heap property is violated:

Case 1. key[v] > key[child[v]]

Case 2. key[v] < key[parent[v]]
Heap Property: If x is the parent of y, then key(x) <= key(y)
If heap property is violated:
   Case 1. key[v] > key[child[v]]
       Swap v with its lowest key child
Heap: Delete an item

**Heap Property:** If x is the parent of y, then \( \text{key}(x) \leq \text{key}(y) \)

If heap property is violated:

**Case 1.** \( \text{key}[v] > \text{key}[\text{child}[v]] \)

Swap v with its **lowest key** child

Continue until heap property holds

Time = \( O(\log n) \)
Heap Property: If x is the parent of y, then key(x) <= key(y)

If heap property is violated:

Case 2. key[v] < key[parent[v]]
Swap v with its parent
Continue till heap property holds

Time = O(log n)
Heap: Delete an item

**Heap Property:** If x is the parent of y, then key(x) <= key(y)

If heap property is violated:

Case 2. key[v] < key[parent[v]]

Swap v with its parent
Continue till heap property holds

Time = O(log n)
Heap Property: If x is the parent of y, then key(x) <= key(y)

If heap property is violated:

Case 2. key[v] < key[parent[v]]

Swap v with its parent
Continue till heap property holds

Time = O(log n)
Heap: Delete an item

Heap Property: If x is the parent of y, then key(x) <= key(y)

If heap property is violated:

Case 2. key[v] < key[parent[v]]
   Swap v with its parent
   Continue till heap property holds

Time = O(log n)
Heap Property: If $x$ is the parent of $y$, then $\text{key}(x) \leq \text{key}(y)$

Operations:
- Add an element: $O(\log n)$
- Delete an element: $O(\log n)$
- Report min: $O(1)$
Prim’s Algorithm

X = {}, S = {r}
Repeat until S has n nodes:
  1. Pick lightest edge e in the cut (S, V - S)
  2. Delete all edges b/w v and S from heap
  3. Add all edges b/w v and V - S - {v}

#edge additions and deletions = O(m) (Why?)
#report mins = O(n)

Use a heap to store edges between S and V - S
At each step:
  1. Pick lightest edge with a report-min
  2. Delete all edges b/w v and S from heap
  3. Add all edges b/w v and V - S - {v}

Black edges = in heap
Prim’s Algorithm

X = { }, S = {r}
Repeat until S has n nodes:
  1. Pick the lightest edge e in the cut (S, V - S)
  2. Delete all edges b/w v and S from heap
  3. Add all edges b/w v and V - S - {v}

#edge additions and deletions = O(m)
#report mins = O(n)
Total running time = O(m log n)

Heap Ops:
  Add: O(log n)
  Delete: O(log n)
  Report min: O(1)
Summary: Prim’s Algorithms

$X = \{ \}, S = \{r\}$
Repeat until $S$ has $n$ nodes:

- Pick the **lightest** edge $e$ in the cut $(S, V - S)$
- Add $e$ to $X$
- Add $v$, the end-point of $e$ in $V - S$, to $S$

**Implementation**: Store edges from $S$ to $V - S$ using a **heap**

**Running Time**: $O(m \log n)$
MST Algorithms

- Kruskal’s Algorithm: Union-Find Data Structure
- Prim’s Algorithm: How to Implement?
- An Application of MST: Single Linkage Clustering
Single Linkage Clustering

**Problem:** Given a set of points, build a hierarchical clustering

**Procedure:**
Initialize: each node is a cluster
Until we have one cluster:
Pick two closest clusters $C, C^*$
Merge $S = C \cup C^*$

Distance between two clusters:
$$d(C, C^*) = \min_{x \in C, y \in C^*} d(x, y)$$

Can you recognize this algorithm?
Greedy Algorithms

- Direct argument - MST
- Exchange argument - Caching
- Greedy approximation algorithms
Optimal Caching

Given a sequence of memory accesses, limited cache: How do you decide which cache element to evict?

**Note:** We are given *future memory accesses* for this problem, which is usually not the case. This is for an application of greedy algorithms.
Optimal Caching: Example

Given a sequence of memory accesses, limited cache size, How do you decide which cache element to evict?

Goal: Minimize #main memory fetches
Optimal Caching: Example

<table>
<thead>
<tr>
<th>M</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>b</th>
<th>c</th>
<th>b</th>
<th>a</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>S₁</td>
<td>a</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>b</td>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Memory Access Sequence

Cache Contents

Evicted items

Given a sequence of memory accesses, limited cache size, how do you decide which cache element to evict?

**Goal**: Minimize #main memory fetches
Optimal Caching: Example

Given a sequence of memory accesses, limited cache size, How do you decide which cache element to evict?

Goal: Minimize #main memory fetches
Optimal Caching: Example

Given a sequence of memory accesses, limited cache size, How do you decide which cache element to evict?

Goal: Minimize #main memory fetches
Optimal Caching: Example

Given a sequence of memory accesses, limited cache size, how do you decide which cache element to evict?

**Goal:** Minimize \#main memory fetches
Optimal Caching: Example

Given a sequence of memory accesses, limited cache size, How do you decide which cache element to evict?

Goal: Minimize #main memory fetches
## Optimal Caching: Example

<table>
<thead>
<tr>
<th>M</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>b</th>
<th>c</th>
<th>b</th>
<th>a</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>S₁</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
</tr>
<tr>
<td>E</td>
<td>-</td>
<td>-</td>
<td>b</td>
<td>a</td>
<td>-</td>
<td>c</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

### Memory Access Sequence

### Cache Contents

### Evicted items

Given a sequence of memory accesses, limited cache size, How do you decide which cache element to evict?

**Goal:** Minimize #main memory fetches
Optimal Caching

Farthest-First (FF) Schedule: Evict an item when needed. Evict the element which is accessed farthest down in the future

Theorem: The FF algorithm minimizes \#fetches
Optimal Caching

Farthest-First (FF) Schedule: Evict an item when needed. Evict the element which is accessed farthest down in the future.

Theorem: The FF algorithm minimizes #fetches
Optimal Caching

Farthest-First (FF) Schedule: Evict an item when needed. Evict the element which is accessed farthest down in the future.

Theorem: The FF algorithm minimizes \#fetches
Optimal Caching

Farthest-First (FF) Schedule: Evict an item when needed. Evict the element which is accessed farthest down in the future.

Theorem: The FF algorithm minimizes \#fetches.
Optimal Caching

Farthest-First (FF) Schedule: Evict an item when needed. Evict the element which is accessed farthest down in the future.

Theorem: The FF algorithm minimizes #fetches.
Optimal Caching

Farthest-First (FF) Schedule: Evict an item when needed. Evict the element which is accessed farthest down in the future.

Theorem: The FF algorithm minimizes \#fetches.
**Optimal Caching**

<table>
<thead>
<tr>
<th>M</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>b</th>
<th>c</th>
<th>b</th>
<th>a</th>
<th>a</th>
</tr>
</thead>
</table>

Memory Access Sequence

<table>
<thead>
<tr>
<th>S₁</th>
<th>a</th>
<th>a</th>
<th>c</th>
<th>c</th>
<th>c</th>
<th>b</th>
<th>b</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>c</td>
<td>a</td>
</tr>
</tbody>
</table>

Cache Contents

| E | - | - | a | - | - | - | - | c |

Evicted items

**Farthest-First (FF) Schedule:** Evict an item when needed. Evict the element which is accessed farthest down in the future.

**Theorem:** The FF algorithm minimizes \#fetches
Optimal Caching

**Farthest-First (FF) Schedule:** Evict an item when needed. Evict the element which is accessed farthest down in the future.

**Theorem:** The FF algorithm minimizes #fetches
Caching: Reduced Schedule

An eviction schedule is **reduced** if it fetches an item $x$ only when it is accessed.

**Fact:** For any $S$, there is a reduced schedule $S^*$ which makes at most as many fetches as $S$. 
Caching: Reduced Schedule

An eviction schedule is **reduced** if it fetches an item \( x \) only when it is accessed.

Fact: For any \( S \), there is a reduced schedule \( S^* \) with at most as many fetches as \( S \).

To convert \( S \) to \( S^* \): Be lazy!
Caching: FF Schedules

Theorem: Suppose a reduced schedule $S_j$ makes the same decisions as SFF from $t=1$ to $t=j$. Then, there exists a reduced schedule $S_{j+1}$ s.t:

1. $S_{j+1}$ makes **same decision** as SFF from $t=1$ to $t=j+1$
2. $\#\text{fetches}(S_{j+1}) \leq \#\text{fetches}(S_j)$
Theorem: Suppose a reduced schedule $S_j$ makes the same decisions as SFF from $t=1$ to $t=j$. Then, there exists a reduced schedule $S_{j+1}$ s.t:

1. $S_{j+1}$ makes same decision as SFF from $t=1$ to $t=j+1$
2. $\#fetches(S_{j+1}) \leq \#fetches(S_j)$

Case 1: No cache miss at $t=j+1$. $S_{j+1} = S_j$
Caching: FF Schedules

**Theorem:** Suppose a reduced schedule $S_j$ makes the same decisions as SFF from $t=1$ to $t=j$. Then, there exists a reduced schedule $S_{j+1}$ s.t:

1. $S_{j+1}$ makes **same decision** as SFF from $t=1$ to $t=j+1$
2. $\#\text{fetches}(S_{j+1}) \leq \#\text{fetches}(S_j)$

**Case 2:** Cache miss at $t=j+1$, $S_j$ and SFF evict same item. $S_{j+1} = S_j$
Caching: FF Schedules

**Theorem:** Suppose a reduced schedule \(S_j\) makes the same decisions as SFF from \(t=1\) to \(t=j\). Then, there exists a reduced schedule \(S_{j+1}\) s.t:
1. \(S_{j+1}\) makes **same decision** as SFF from \(t=1\) to \(t=j+1\)
2. \(\#\text{fetches}(S_{j+1}) \leq \#\text{fetches}(S_j)\)

**Case 3a:** Cache miss at \(t=j+1\). \(S_j\) evicts a, SFF evicts b. \(S_{j+1}\) also evicts b. Next there is a request to d, and \(S_j\) evicts b. Make \(S_{j+1}\) evict a, bring in d.
Caching: FF Schedules

**Theorem:** Suppose a reduced schedule $S_j$ makes the same decisions as SFF from $t=1$ to $t=j$. Then, there exists a reduced schedule $S_{j+1}$ s.t:

1. $S_{j+1}$ makes **same decision** as SFF from $t=1$ to $t=j+1$
2. $\#\text{fetches}(S_{j+1}) \leq \#\text{fetches}(S_j)$

**Case 3b:** Cache miss at $t=j+1$. $S_j$ evicts $a$, SFF evicts $b$. $S_{j+1}$ also evicts $b$

Next there is a request to $a$, and $S_j$ evicts $b$. $S_{j+1}$ does nothing.
Caching: FF Schedules

Theorem: Suppose a reduced schedule $S_j$ makes the same decisions as SFF from $t=1$ to $t=j$. Then, there exists a reduced schedule $S_{j+1}$ s.t:

1. $S_{j+1}$ makes same decision as SFF from $t=1$ to $t=j+1$
2. $\#fetches(S_{j+1}) \leq \#fetches(S_j)$

Case 3c: Cache miss at $t=j+1$. $S_j$ evicts a, SFF evicts b. $S_{j+1}$ also evicts b
Next there is a request to a, and $S_j$ evicts d. $S_{j+1}$ evicts d and brings in b.
Now convert $S_{j+1}$ to the reduced version of this schedule.
Theorem: Suppose a reduced schedule $S_j$ makes the same decisions as SFF from $t=1$ to $t=j$. Then, there exists a reduced schedule $S_{j+1}$ s.t:
1. $S_{j+1}$ makes same decision as SFF from $t=1$ to $t=j+1$
2. $\#fetches(S_{j+1}) \leq \#fetches(S_j)$

Case 3d: Cache miss at $t=j+1$. $S_j$ evicts a, SFF evicts b. $S_{j+1}$ also evicts b
Next there is a request to b. Cannot happen as a is accessed before b!
Summary: Optimal Caching

Theorem: Suppose a reduced schedule $S_j$ makes the same decisions as SFF from $t=1$ to $t=j$. Then, there exists a reduced schedule $S_{j+1}$ s.t:
1. $S_{j+1}$ makes same decision as SFF from $t=1$ to $t=j+1$
2. $\#\text{fetches}(S_{j+1}) \leq \#\text{fetches}(S_j)$

Case 1: No cache miss at $t=j+1$. $S_{j+1} = S_j$

Case 2: Cache miss at $t=j+1$, $S_j$ and SFF evict same item. $S_{j+1} = S_j$

Case 3a: Cache miss at $t=j+1$. $S_j$ evicts $a$, SFF evicts $b$. $S_{j+1}$ also evicts $b$. Next there is a request to $d$, and $S_j$ evicts $b$. Make $S_{j+1}$ evict $a$, bring in $d$.

Case 3b: Cache miss at $t=j+1$. $S_j$ evicts $a$, SFF evicts $b$. $S_{j+1}$ also evicts $b$. Next there is a request to $a$, and $S_j$ evicts $b$. $S_{j+1}$ does nothing.

Case 3c: Cache miss at $t=j+1$. $S_j$ evicts $a$, SFF evicts $b$. $S_{j+1}$ also evicts $b$. Next there is a request to $a$, and $S_j$ evicts $d$. $S_{j+1}$ evicts $d$ and brings in $b$. Now convert $S_{j+1}$ to the reduced version of this schedule.

Case 3d: Cache miss at $t=j+1$. $S_j$ evicts $a$, SFF evicts $b$. $S_{j+1}$ also evicts $b$. Next there is a request to $b$. Cannot happen as $a$ is accessed before $b$!
**Summary: Optimal Caching**

**Theorem:** Suppose a reduced schedule $S_j$ makes the same decisions as SFF from $t=1$ to $t=j$. Then, there exists a reduced schedule $S_{j+1}$ s.t:

1. $S_{j+1}$ makes **same decision** as SFF from $t=1$ to $t=j+1$
2. $\text{#fetches}(S_{j+1}) \leq \text{#fetches}(S_j)$

Suppose you claim a magic schedule schedule $S_M$ makes less fetches than SFF. Then, we can construct a sequence of schedules:

$S_M = S_0, S_1, S_2, ..., S_n = SFF$ such that:

1. $S_j$ agrees with SFF from $t=1$ to $t = j$
2. $\text{#fetches}(S_{j+1}) \leq \text{#fetches}(S_j)$

What does this say about $\text{#fetches}(SFF)$ relative to $\text{#fetches}(S_M)$?
Greedy Algorithms

- Direct argument - MST
- Exchange argument - Caching
- Greedy approximation algorithms
Greedy Approximation Algorithms

- k-Center
- Set Cover
Approximation Algorithms

- Optimization problems, e.g., MST, Shortest paths
- For an instance $I$, let:
  - $A(I) =$ value of solution by algorithm $A$
  - $OPT(I) =$ value of optimal solution
- Approximation ratio($A$) = $\max_I A(I)/OPT(I)$
- $A$ is an approx. algorithm if approx-ratio($A$) is bounded
k-Center Problem

Given $n$ towns on a map
Find how to place $k$ shopping malls such that:
Drive to the nearest mall from any town is shortest
k-Center Problem

Given \text{n towns} on a map
Find how to place \text{k shopping malls} such that:
Drive to the nearest mall from any town is shortest
k-Center Problem

Given $n$ points in a metric space
Find $k$ centers such that distance between any point and its closest center is as small as possible

Metric Space:
Point set with distance function $d$

Properties of $d$:
- $d(x, y) \geq 0$
- $d(x, y) = d(y, x)$
- $d(x, y) \leq d(x, z) + d(y, z)$

NP Hard in general
Greedy Algorithm: Furthest-first traversal

1. Pick $C = \{x\}$, for an arbitrary point $x$
2. Repeat until $C$ has $k$ centers:
   - Let $y$ maximize $d(y, C)$, where
     $$d(y, C) = \min_{x \in C} d(x, y)$$
   - $C = C \cup \{y\}$
Greedy Algorithm: Furthest-first traversal

1. Pick $C = \{x\}$, for an arbitrary point $x$
2. Repeat until $C$ has $k$ centers:
   
   Let $y$ maximize $d(y, C)$, where
   
   $d(y, C) = \min_{x \in C} d(x, y)$
   
   $C = C \cup \{y\}$
Greedy Algorithm: Furthest-first traversal

1. Pick $C = \{x\}$, for an arbitrary point $x$
2. Repeat until $C$ has $k$ centers:
   
   Let $y$ maximize $d(y, C)$, where
   
   $$d(y, C) = \min_{x \in C} d(x, y)$$

   $C = C \cup \{y\}$
Greedy Algorithm: Furthest-first traversal

1. Pick $C = \{x\}$, for an arbitrary point $x$
2. Repeat until $C$ has $k$ centers:
   
   Let $y$ maximize $d(y, C)$, where
   
   $$d(y, C) = \min_{x \in C} d(x, y)$$

   $C = C \cup \{y\}$
Greedy Algorithm: Furthest-first traversal

1. Pick $C = \{x\}$, for an arbitrary point $x$
2. Repeat until $C$ has $k$ centers:
   Let $y$ maximize $d(y, C)$, where
   $d(y, C) = \min_{x \in C} d(x, y)$
   $C = C \cup \{y\}$
Furthest-first Traversal

Is furthest-first traversal always optimal?

Theorem: Approx. ratio of furthest-first traversal is 2
**Furthest-first (FF) Traversal**

**Metric Space:**
Point set w/ distance fn \( d \)

**Properties of \( d \):**
- \( d(x, y) \geq 0 \)
- \( d(x, y) = d(y, x) \)
- \( d(x, y) \leq d(x, z) + d(y, z) \)

For a set \( S \),
\[
d(x, S) = \min_{y \in S} d(x, y)
\]

**FF-traversal:**
Pick \( C = \{x\} \), arbitrary \( x \)
Repeat until \( C \) has \( k \) centers:
- Let \( y \) maximize \( d(y, C) \)
- \( C = C \cup \{y\} \)

**Theorem:** Approx. ratio of FF-traversal is 2

Define, for any instance: \( r = \max_x d(x, C) \)

**Property 1.** Solution value of FF-traversal = \( r \)

**Property 2.** There are at least \( k+1 \) points \( S \) s.t.
each pair has distance \( \geq r \)

**Property 3.** Any \( k \)-center solution must assign at least two points \( x, y \) in \( S \) to the same center \( c \)

What is \( \max(d(x, c), d(y, c)) \)?
**Furthest-first (FF) Traversal**

**Metric Space:**
Point set w/ distance fn $d$

**Properties of $d$:**

- $d(x, y) \geq 0$
- $d(x, y) = d(y, x)$
- $d(x, y) \leq d(x, z) + d(y, z)$

For a set $S$,

$$d(x, S) = \min_{y \in S} d(x, y)$$

**FF-traversal:**
Pick $C = \{x\}$, arbitrary $x$
Repeat until $C$ has $k$ centers:

- Let $y$ maximize $d(y, C)$
- $C = C \cup \{y\}$

**Theorem:** Approx. ratio of FF-traversal is 2

Define, for any instance: $r = \max_x d(x, C)$

**Property 3.** Any $k$-center solution must assign at least two points $x, y$ in $S$ to the same center $c$.

What is $\max(d(x, c), d(y, c))$?

From property of $d$,

$$d(x, y) \geq d(x, c) + d(y, c)$$

$$\max(d(x, c), d(y, c)) \geq d(x, y)/2$$
**Furthest-first (FF) Traversal**

**Theorem:** Approx. ratio of FF-traversal is 2

Define, for any instance: \( r = \max_x d(x, C) \)

**Metric Space:**
Point set w/ distance fn \( d \)

**Properties of \( d \):**
- \( d(x, y) \geq 0 \)
- \( d(x, y) = d(y, x) \)
- \( d(x, y) \leq d(x, z) + d(y, z) \)

For a set \( S \),
\[
d(x, S) = \min_{y \in S} d(x, y) \]

**FF-traversal:**
Pick \( C = \{x\} \), arbitrary \( x \)
Repeat until \( C \) has \( k \) centers:
- Let \( y \) maximize \( d(y, C) \)
- \( C = C \cup \{y\} \)

**Property 1.** Solution value of FF-traversal = \( r \)

**Property 2.** There are at least \( k+1 \) points \( S \) s.t each pair has distance \( \geq r \)

**Property 3.** Any \( k \)-center solution must assign at least two points \( x, y \) in \( S \) to the same center \( c \)
\[
\max(d(x, c), d(y, c)) \geq \frac{d(x,y)}{2} \geq \frac{r}{2}
\]

**Property 4.** Any other solution has value \( \geq \frac{r}{2} \)
Applications:

- Facility-location problems
- Clustering
- Initialization step in clustering problems e.g, k-means++
Greedy Approximation Algorithms

• k-Center

• Set Cover
Set Cover Problem

Given:
- Universe $U$ with $n$ elements
- Collection $C$ of sets of elements of $U$

Find the smallest subset $C^*$ of $C$ that covers all of $U$

NP Hard in general
Set Cover Problem

Given:
- Universe $U$ with $n$ elements
- Collection $C$ of sets of elements of $U$

Find the smallest subset $C^*$ of $C$ that covers all of $U$

NP Hard in general
Applications

• Sensor placing problems

• Facility location problems
A Greedy Set-Cover Algorithm

\[ C^* = \{ \} \]

Repeat until all of \( U \) is covered:
- Pick the set \( S \) in \( C \) with highest \# of uncovered elements
- Add \( S \) to \( C^* \)
A Greedy Set-Cover Algorithm

$C^* = \{ \} $

Repeat until all of $U$ is covered:

Pick the set $S$ in $C$ with highest # of uncovered elements
Add $S$ to $C^*$
A Greedy Set-Cover Algorithm

C* = { }  
Repeat until all of U is covered:
  Pick the set S in C with highest # of uncovered elements
  Add S to C*

Diagram of a set-cover problem with two sets and three elements per set.
A Greedy Set-Cover Algorithm

\[ C^* = \{ \} \]
Repeat until all of U is covered:
   Pick the set \( S \) in \( C \) with highest \# of uncovered elements
   Add \( S \) to \( C^* \)
A Greedy Set-Cover Algorithm

C* = { }  
Repeat until all of U is covered:
  Pick the set S in C with highest # of uncovered elements
  Add S to C*
A Greedy Set-Cover Algorithm

C* = { }
Repeat until all of U is covered:
   Pick the set S in C with highest # of uncovered elements
   Add S to C*

Diagram showing set covering algorithm.
A Greedy Set-Cover Algorithm

\[ C^* = \{ \} \]
Repeat until all of \( U \) is covered:
Pick the set \( S \) in \( C \) with highest \# of uncovered elements
Add \( S \) to \( C^* \)
A Greedy Set-Cover Algorithm

C* = { }
Repeat until all of U is covered:
   Pick the set S in C with highest # of uncovered elements
   Add S to C*

Greedy: #sets=7
A Greedy Set-Cover Algorithm

C* = { }
Repeat until all of U is covered:
   Pick the set S in C with highest # of uncovered elements
   Add S to C*

Greedy: #sets=7
OPT: #sets=5
Greedy Set-Cover Algorithm

**Theorem:** If optimal set cover has \( k \) sets, then greedy selects \( \leq k \ln n \) sets.

**Greedy Algorithm:**

\[
C^* = \{ \} \\
\text{Repeat until } U \text{ is covered:} \\
\quad \text{Pick } S \text{ in } C \text{ with highest # of uncovered elements} \\
\quad \text{Add } S \text{ to } C^*
\]

Define:

\[ n(t) = \text{#uncovered elements after step } t \text{ in greedy} \]

**Property 1:** There is some \( S \) that covers at least \( n(t)/k \) of the uncovered elements.

**Property 2:** \( n(t+1) \leq n(t)(1 - 1/k) \)

**Property 3:** \( n(T) \leq n(1 - 1/k)^T < 1 \),
when \( T = k \ln n \)
Greedy Algorithms

• Direct argument - MST
• Exchange argument - Caching
• Greedy approximation algorithms
  • k-center, set-cover