CSE 202: Design and Analysis of Algorithms

Lecture 3

Instructor: Kamalika Chaudhuri
Announcement

• Homework 1 out
• Due on **Tue Jan 24** in class
• No late homeworks will be accepted
Greedy Algorithms

- Direct argument - MST
- Exchange argument - Caching
- Greedy approximation algorithms
Last Class: MST Algorithms

• Kruskal’s Algorithm: Union-Find Data Structure
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

```plaintext
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 \( \text{rank}[\text{p}[x]] > \text{rank}[x] \)

**Proof:** By property of union

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

**Proof:** By induction

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

**Proof:** Combining properties 1 and 2
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 \( n/2^k \) nodes of rank \( k \)

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

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

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

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

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

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

**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 $k$

Interval $I_k = [k+1, k+2, \ldots, 2^k]$  
$\#\text{intervals} = \log^*n$

**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

$\#\text{nodes of type I} \leq \log^*n$
Time spent on nodes of **type I** in $m$ finds $\leq m \log^*n$
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 $k$

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 $\leq 2^k$
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 \( k \)

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

Total time on **type 1** nodes \( \leq m \log^*n \)
Total time on **type 2** node \( x \) \( \leq t(x) = 2^k \)
Total time on m find operations \( \leq m \log^*n + \sum t(x) \)

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

**#intervals = \log^*n**
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 \( n/2^k \) nodes of rank \( k \)

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

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

**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 \( n/2^k \) nodes of rank \( k \)

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

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

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

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

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 = \( \text{log}^*n \), \( \sum t(x) \leq n \text{log}^*n \)
**Summary: 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)**

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

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

\textbf{Sort} the edges
\textbf{Add} \( e \) to \( X \)
\textbf{Check} if end-points of \( e \) lie in different components
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 = \( O(1) + \text{Time(Find)} \)

**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
Prim’s Algorithm

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

**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

\[ X = \{ \}, \quad S = \{r\} \]
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 = {}, 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?
Prim’s Algorithm

\[ X = \{ \}, \quad S = \{r\} \]
Repeat until \( S \) has \( n \) nodes:
   1. Pick the \textbf{lightest} edge \( e \) in the cut \((S, V - S)\)
   2. Add \( e \) to \( X \)
   3. 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. \textbf{Add} an edge
   2. \textbf{Delete} an edge
   3. \textbf{Report} the edge with \textbf{min} weight
Data Structure: Heap

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

A heap is stored as a balanced binary tree
Height = \( O(\log n) \), where \( n = \# \) nodes
**Heap Property:** If $x$ is the parent of $y$, then $\text{key}(x) \leq \text{key}(y)$
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: Add an item

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

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: 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 $\text{key}(x) \leq \text{key}(y)$

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

Time = $O(\log n)$
**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) <= 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 $\text{key}(x) \leq \text{key}(y)$

If heap property is violated:

Case 1. $\text{key}[v] > \text{key}[$child$[v]]$

Swap $v$ with its lowest key child
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 $\text{key}(x) \leq \text{key}(y)$

If heap property is violated:

Case 2. $\text{key}[v] < \text{key}[\text{parent}[v]]$
- Swap $v$ with its parent
- Continue till heap property holds

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

If heap property is violated:

**Case 2.** \( \text{key}[v] < \text{key}[\text{parent}[v]] \)

Swap \( v \) with its **parent**
Continue till heap property holds

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

If heap property is violated:

**Case 2.** \( \text{key}[v] < \text{key}[\text{parent}[v]] \)

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)
**Summary: Heap**

**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:

- 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 \( S \)

Use a **heap** to store edges between \( S \) and \( V - S \)

[Diagram: S and V - S with black edges indicated as in heap]
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 \( S \)

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 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

\[ X = \{ \}, \quad S = \{r\} \]
Repeat until \( S \) has \( n \) nodes:

1. Pick the \textbf{lightest} edge \( e \) in the cut \((S, V - S)\)
2. Add \( e \) to \( X \)
3. Add \( v \), the end-point of \( e \) in \( V - S \) to \( S \)

Use a \textbf{heap} to store edges b/w \( S \) 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 \( V - S \) - \( \{v\} \)

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

\textbf{Heap Ops:}
Add: \( O(\log n) \)
Delete: \( O(\log n) \)
Report min: \( O(1) \)
Prim's Algorithm

\[ X = \{ \}, \quad S = \{r\} \]
Repeat until \( S \) has \( n \) nodes:

1. Pick the **lightest** edge \( e \) in the cut \((S,V - S)\)
2. Add \( e \) to \( X \)
3. Add \( v \), the end-point of \( e \) in \( V - S \) to \( S \)

Use a **heap** to store edges b/w \( S \) 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 \( V - S - \{v\} \)

#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

<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>S</td>
<td>a</td>
<td></td>
<td>b</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

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