Greedy Algorithms

It is hard to define exactly but can give general properties:
- Solution is built in small steps
- Decisions on how to build the solution are made to maximize some criterion without looking to the future
  - Want the ‘best’ current partial solution as if the current step were the last step
- May be more than one greedy algorithm using different criteria to solve a given problem

Greedy Algorithms
- Easy to produce
- Fast running times
- Work only on certain classes of problems
- Two methods for proving that greedy algorithms do work
- Greedy algorithm stays ahead
  - At each step any other algorithm will have a worse value for the criterion
- Exchange Argument
  - Can transform any other solution to the greedy solution at no loss in quality

Interval Scheduling
- Single resource
- Reservation requests
  - Of form “Can I reserve it from start time $s$ to finish time $f$?”
  - $s < f$
- Find: maximum number of requests that can be scheduled so that no two reservations have the resource at the same time

Interval scheduling
- Formally
  - Requests $1, 2, \ldots, n$
    - Request $i$ has start time $s_i$ and finish time $f_i$, $s_i < f_i$
    - Requests $i$ and $j$ are compatible iff either
      - request $i$ is for a time entirely before request $j$
      - $f_i \leq s_j$
      - or, request $j$ is for a time entirely before request $i$
      - $f_j \leq s_i$
    - Set $A$ of requests is compatible iff every pair of requests $i,j \in A$, $i \neq j$ is compatible
    - Goal: Find maximum size subset $A$ of compatible requests
Greedy Algorithms for Interval Scheduling

- What criterion should we try?
  - Earliest start time $s_i$
  - Shortest request time $f_i - s_i$
  - Earliest finish time $f_i$

What criterion should we try?

- Earliest start time $s_i$
  - Doesn't work

- Shortest request time $f_i - s_i$
  - Doesn't work

- Even fewest conflicts doesn't work

- Earliest finish time $f_i$
  - Works

Greedy Algorithm for Interval Scheduling

1. $R \leftarrow$ set of all requests
2. $A \leftarrow \emptyset$
3. While $R \neq \emptyset$
   - Choose request $i \in R$ with smallest finishing time $f_i$
   - Add request $i$ to $A$
   - Delete all requests in $R$ that are not compatible with request $i$
4. Return $A$

Claim: $A$ is a compatible set of requests and these are added to $A$ in order of finish time.

When we add a request to $A$ we delete all incompatible ones from $R$.

Claim: For any other set $O \subseteq R$ of compatible requests then if we order requests in $A$ and $O$ by finish time then for each $k$:

- If $O$ contains a $k$th request then so does $A$
- The finish time of the $k$th request in $A$ is at least the finish time of the $k$th request in $O$, i.e., $a_k \leq o_k$ where $a_k$ and $o_k$ are the respective finish times.

Inductive Proof of Claim: $a_k \leq o_k$

- Base Case: This is true for the first request in $A$ since that is the one with the smallest finish time.
- Inductive Step: Suppose $a_k \leq o_k$.
  - By definition of compatibility:
    - If $O$ contains a $k$th request then the start time of that request must be after $o_k$, and thus after $a_k$.
    - Thus $r$ is compatible with the first $k$ requests in $A$.
    - Therefore $A$ has at least $k+1$ requests since a compatible one is available after the first $k$ are chosen.
  - $r$ was among those considered by the greedy algorithm for that $k+1$th request in $A$.
- Therefore by the greedy choice the finish time of $r$ which is $a_{k+1}$ is at least the finish time of that $k+1$th request in $A$ which is $a_{k+1}$.

Interval Scheduling: Analysis

Therefore we have:

- Theorem: Greedy algorithm is optimal.
- Alternative Proof: (by contradiction)
  - Assume greedy is not optimal, and let's see what happens.
  - Let $a_1, a_2, \ldots, a_k$ denote set of jobs selected by greedy.
  - Let $o_1, o_2, \ldots, o_k$ denote set of jobs in the optimal solution with $a_1 \leq o_1, a_2 \leq o_2, \ldots, a_k \leq o_k$ for the largest possible value of $k$.

Greedy: $a_1, a_2, a_3, a_k, \text{finish before } o_{k+1}$

OPT: $o_1, o_2, o_3, o_k, \text{finish before } o_{k+1}$

why not replace job $a_{k+1}$ with job $o_{k+1}$?
Implementing the Greedy Algorithm

- Sort the requests by finish time
  - \(O(n \log n)\) time
- Maintain current latest finish time scheduled
- Keep array of start times indexed by request number
- Only eliminate incompatible requests as needed
- Walk along array of requests sorted by finish times skipping those whose start time is before current latest finish time scheduled
- \(O(n)\) additional time for greedy algorithm

```
Sort jobs by finish times so that \(0 \leq f_1 \leq f_2 \leq \ldots \leq f_n\).
A ← φ
last ← 0
for \(j = 1\) to \(n\) {
    if (last ≤ \(s_j\))
        \(A ← A \cup \{j\}\)
    last ← \(f_j\)
}
return A
```

Scheduling All Intervals: Interval Partitioning

- Interval partitioning
  - Lecture \(j\) starts at \(s_j\) and finishes at \(f_j\)
  - Goal: find minimum number of classrooms to schedule all lectures so that no two occur at the same time in the same room.
- Example: This schedule uses 4 classrooms to schedule 10 lectures.

```
Example: This schedule uses only 3 classrooms
```

Scheduling all intervals

- Interval Partitioning Problem: We have resources to serve more than one request at once and want to schedule all the intervals using as few of our resources as possible
- Obvious requirement: At least the depth of the set of requests

```
```

Interval Partitioning: Lower Bound on Optimal Solution

- Definition. The depth of a set of open intervals is the maximum number that contain any given time.
- Key observation. Number of classrooms needed \(\geq\) depth.
- Ex: Depth of schedule below = 3 \(\rightarrow\) schedule below is optimal.
- Q. Does there always exist a schedule equal to depth of intervals?
A simple greedy algorithm

Sort requests in increasing order of start times \((s_1, f_1), \ldots, (s_n, f_n)\)

For \(i = 1\) to \(n\)
  \(j \leftarrow 1\)
  While (request \(i\) not scheduled)
    \(last_j \leftarrow \text{finish time of the last request currently scheduled on resource } j\)
    If \(s_i \leq last_j\), then schedule request \(i\) on resource \(j\)
    \(j \leftarrow j + 1\)
  End While
End For

Interval Partitioning: Greedy Analysis

- Observation. Greedy algorithm never schedules two incompatible lectures in the same classroom.
- Theorem. Greedy algorithm is optimal.
- Proof.
  - Let \(d\) = number of classrooms that the greedy algorithm allocates.
  - Classroom \(d\) is opened because we needed to schedule a job, say \(j\), that is incompatible with all \(d-1\) other classrooms.
  - Since we sorted by start time, all these incompatibilities are caused by lectures that start no later than \(s_j\).
  - Thus, we have \(d\) lectures overlapping at time \(s_j\).
  - Key observation \(\Rightarrow\) all schedules use \(d\) classrooms.

A more efficient implementation

Sort requests in increasing order of start times \((s_1, f_1), \ldots, (s_n, f_n)\)

For \(i = 1\) to \(n\)
  \(j \leftarrow 1\)
  While (request \(i\) not scheduled)
    \(last_j \leftarrow \text{finish time of the last request currently scheduled on resource } j\)
    If \(s_i \leq last_j\), then schedule request \(i\) on resource \(j\)
    \(j \leftarrow j + 1\)
  End While
End For

Greedy Analysis Strategies

- Greedy algorithm stays ahead. Show that after each step of the greedy algorithm, its solution is at least as good as any other algorithm’s.
- Exchange argument. Gradually transform any solution to the one found by the greedy algorithm without hurting its quality.
- Structural. Discover a simple “structural” bound asserting that every possible solution must have a certain value. Then show that your algorithm always achieves this bound.

Scheduling to Minimize Lateness

**Scheduling to minimize lateness**

- Single resource as in interval scheduling but instead of start and finish times request \(i\) has
  - Time requirement \(t_i\), which must be scheduled in a contiguous block
  - Target deadline \(d_i\), by which time the request would like to be finished
  - Overall start time \(s_i\)
- Requests are scheduled by the algorithm into time intervals \((s_i, f_i)\) such that \(L_i = f_i - s_i\)
- Lateness of schedule for request \(i\) is
  - \(0\) if \(d_i < t_i\), then request \(i\) is late by \(L_i = f_i - d_i\), otherwise its latency \(L_i = 0\)
  - Maximum lateness \(L_{\text{max}}\)
- Goal: Find a schedule for all requests (values of \(s_i\) and \(t_i\) for each request \(i\)) to minimize the maximum latency, \(L_{\text{max}}\)
Scheduling to Minimizing Lateness

- Example:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
</tbody>
</table>

Minimizing Lateness: Greedy Algorithms

- Greedy template. Consider jobs in some order.
  - [Shortest processing time first] Consider jobs in ascending order of processing time $t_i$.
  - [Earliest deadline first] Consider jobs in ascending order of deadline $d_i$.
  - [Smallest slack] Consider jobs in ascending order of slack $d_i - t_i$.

Greedy Algorithm: Earliest Deadline First

- Order requests in increasing order of deadlines
- Schedule the request with the earliest deadline as soon as the resource becomes available

Minimizing Lateness: Greedy Algorithm

- Greedy algorithm. Earliest deadline first.

**Proof for Greedy Algorithm: Exchange Argument**

We will show that if there is another schedule O (think optimal schedule) then we can gradually change O so that
- at each step the maximum lateness in O never gets worse
- it eventually becomes the same cost as A
Minimizing Lateness: No Idle Time

- Observation. There exists an optimal schedule with no idle time.

- Observation. The greedy schedule has no idle time.

Minimizing Lateness: Inversions

- Definition. An inversion in schedule $S$ is a pair of jobs $i$ and $j$ such that $d_i < d_j$ but $j$ scheduled before $i$.

- Observation. Greedy schedule has no inversions.

- Observation. If a schedule (with no idle time) has an inversion, it has one with a pair of inverted jobs scheduled consecutively (by transitivity of $<$).

Minimizing Lateness: Inversions

- Claim. Swapping two adjacent, inverted jobs reduces the number of inversions by one and does not increase the max lateness.

- Claim: There is an optimal schedule with no idle time and no inversions

- Proof:
  - By previous argument there is an optimal schedule $O$ with no idle time
  - If $O$ has an inversion then it has a consecutive pair of requests in its schedule that are inverted and can be swapped without increasing lateness

Optimal schedules and inversions

- Eventually these swaps will produce an optimal schedule with no inversions
  - Each swap decreases the number of inversions by 1
  - There are a bounded number of (at most $n(n-1)/2$) inversions (we only care that this is finite.)
Idleness and Inversions are the only issue
- **Claim:** All schedules with no inversions and no idle time have the same maximum lateness
- **Proof**
  - Schedules can differ only in how they order requests with equal deadlines
  - Consider all requests having some common deadline \( d \)
  - Maximum lateness of these jobs is based on the finish time of the last of these jobs but the set of these requests occupies the same time segment in both schedules
  - Last of these requests finishes at the same time in any such schedule.

Earliest Deadline First is optimal
- **We know that**
  - There is an optimal schedule with no idle time or inversions
  - All schedules with no idle time or inversions have the same maximum lateness
  - EDF produces a schedule with no idle time or inversions
- **Therefore**
  - EDF produces an optimal schedule

Optimal Caching/Paging
- **Memory systems**
  - Many levels of storage with different access times
  - Smaller storage has shorter access time
  - To access an item it must be brought to the lowest level of the memory system
- **Consider the management problem between adjacent levels**
  - Main memory with \( n \) data items from a set \( U \)
  - Cache can hold \( k < n \) items
  - Simplest version with no direct-mapping or other restrictions about where items can be
  - Suppose cache is full initially
    - Holds \( k \) data items to start with

Optimal Caching/Paging
- **Given a memory request \( d \) from \( U \)**
  - If \( d \) is stored in the cache we can access it quickly
  - If not then we call it a cache miss and (since the cache is full)
    - We must bring it into cache and evict some other data item from the cache
    - Which one to evict?
  - **Given** a sequence \( D = d_1, d_2, \ldots, d_m \) of elements from \( U \) corresponding to memory requests
  - **Find** a sequence of evictions (an eviction schedule) that has as few cache misses as possible

Caching Example
- **\( n=3, k=2, U = \{a, b, c\} \)**
- Cache initially contains \( \{a, b\} \)
- \( D = a \ b \ c \ b \ c \ a \ b \)
- \( S = a \ c \)
- \( C = \begin{array}{ccc} a & b & a \\ b & c & b \end{array} \)
- This is optimal

A Note on Optimal Caching
- In real operating conditions one typically needs an on-line algorithm
  - Make the eviction decisions as each memory request arrives
- However to design and analyze these algorithms it is also important to understand how the best possible decisions can be made if one did know the future
  - Field of on-line algorithms compares the quality of on-line decisions to that of the optimal schedule
  - What does an optimal schedule look like?
Belady’s Greedy Algorithm: Farthest-In-Future
- Given sequence \( D = d_1, d_2, \ldots, d_m \)
- When \( d_i \) needs to be brought into the cache evict the item that is needed farthest in the future
  - Let \( \text{NextAccess}_i(d) = \min \{ j : d_j = d_i \} \) be the next point in \( D \) that item \( d \) will be requested
  - Evict \( d \) such that \( \text{NextAccess}_i(d) \) is largest

Other Algorithms
- Often there is flexibility, e.g.
  - \( k = 3, C = \{a, b, c\} \)
  - \( D = \) \( a \ b \ c \ d \ a \ d \ e \ a \ d \ b \ c \)
  - \( S_{\mathsf{FF}} = \) \( c \ b \ e \ d \)
  - \( S = \) \( b \ c \ d \ e \)
- Why aren’t other algorithms better?
  - Least-Frequently-Used-In-Future?
  - Exchange Argument
    - We can swap choices to convert other schedules to Farthest-In-Future without losing quality

Optimal Offline Caching
- Caching:
  - 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 (actually, # of evictions).
- Example: \( k = 2 \), initial cache = \( ab \),
  - requests: \( a, b, c, b, c, a, b \).
  - Optimal eviction schedule: 2 cache misses.

Optimal Offline Caching: Farthest-In-Future
- Farthest-in-future. Evict item in the cache that is not requested until farthest in the future.
  - Current cache: \( a \ b \ c \ d \ e \ f \ g \ h \ ... \)
  - Future queries: \( a \ b \ c \ d \ e \ f \ g \ h \ ... \)
  - Cache exist
  - Evict That one
- Theorem. [Belady, 1960s] FIF is an optimal eviction schedule.
- Proof. Algorithm and theorem are intuitive; proof is subtle.

Reduced Eviction Schedules
- 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.
- Claim. Given any unreduced schedule \( S \), can transform it into a reduced schedule \( S' \) with no more cache misses.
- Proof. (by induction on number of unreduced items)
  - Suppose \( S \) brings \( d \) into the cache at time \( t \), without a request.
  - Let \( c \) be the item \( S \) evicts when it brings \( d \) into the cache.
  - Case 1: \( d \) existed at time \( t \), before next request for \( d \).
  - Case 2: \( d \) requested at time \( t \) before \( d \) is evicted.
**Farthest-In-Future: Analysis**

- **Theorem.** FIF is optimal eviction algorithm.
- **Proof.** (by induction on number of requests $j$)

  **Invariant:** There exists an optimal reduced schedule $S$ that makes the same eviction schedule as $S_{\text{FIF}}$ 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 $S_{\text{FIF}}$ 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 $S_{\text{FIF}}$ evict the same element).

  $S' = S$ satisfies invariant.

  **Invariant:** There exists an optimal reduced schedule $S$ that makes the same eviction schedule as $S_{\text{FIF}}$ through the first $j+1$ requests.

  **Proof.** (continued)

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

  begin construction of $S'$ from $S$ by evicting $e$ instead of $f$

  now $S'$ agrees with $S_{\text{FIF}}$ on first $j+1$ requests; we show that having element $f$ in cache is no worse than having element $e$

  **Note:** $S'$ is no longer reduced, but can be transformed into a reduced schedule that agrees with $S_{\text{FIF}}$ through step $j$.

**Farthest-In-Future: Analysis**

- Let $j'$ be the first time after $j+1$ that $S$ and $S'$ take a different action, and let $g$ be item requested at time $j'$.

  **Case 3c:** $g \neq e, f$. $S$ must evict $e$. Make $S'$ evict $f$; now $S$ and $S'$ have the same cache.

**Caching Perspective**

- **Online vs. offline algorithms.**
  - Offline: full sequence of requests is known a priori.
  - Online (reality): requests are not known in advance.
  - Caching is among most fundamental online problems in CS.

- **LIFO.** Evict page brought in most recently.
- **LRU.** Evict page whose most recent access was earliest.

- **Theorem.** FF is optimal offline eviction algorithm.
  - Provides basis for understanding and analyzing online algorithms.
  - LRU is $k$-competitive. [Section 13.8]
  - LIFO is arbitrarily bad.

**Single-source shortest paths**

- Given an (un)directed graph $G=(V,E)$ with each edge $e$ having a non-negative weight $w(e)$ and a vertex $v$

  Find length of shortest paths from $v$ to each vertex in $G$
A greedy algorithm

Dijkstra's Algorithm:
- Maintain a set $S$ of vertices whose shortest paths are known
  - initially $S = \{s\}$
- Maintaining current best lengths of paths that only go through $S$ to each of the vertices in $G$
  - path-lengths to elements of $S$ will be right, to $V-S$ they might not be right
- Repeatedly add vertex $v$ to $S$ that has the shortest path-length of any vertex in $V-S$
  - update path lengths based on new paths through $v$

Dijkstra's Algorithm

Dijkstra($G, w, s$)
$S \leftarrow \{s\}$
$d[s] \leftarrow 0$

while $S \neq V$
do
  of all edges $e = (u, v)$ s.t. $v \in S$ and $u \not\in S$ select* one with the minimum value of $d[u] + w(e)$
  $S \leftarrow S \cup \{v\}$
  $d[v] \leftarrow d[u] + w(e)$
  $\text{pred}[v] \leftarrow u$

*For each $v \in S$ maintain $d'[v] = \text{minimum value of } d[u] + w(e)$ over all vertices $ue S$ s.t. $e = (u,v)$ is in $G$
Dijkstra's Algorithm

Update distances

Add to S

Update distances

Add to S

Update distances

Add to S

Update distances
Update distances

Add to S

Dijkstra's Algorithm

Update distances

Add to S

Dijkstra's Algorithm

Update distances

Add to S

Dijkstra's Algorithm

Update distances

Add to S

Dijkstra's Algorithm

Update distances

Add to S
Dijkstra's Algorithm

Update distances

Dijkstra's Algorithm

Add to S
Dijkstra's Algorithm

Update distances

Dijkstra's Algorithm

Add to S

Dijkstra's Algorithm Correctness

Suppose all distances to vertices in \( S \) are correct and \( u \) has smallest current value in \( V-S \)

\[ d'(v) \leq d'(x) \]

Suppose some other path to \( v \) and \( x= \) first vertex on this path not in \( S \)

\[ x-v \text{ path length} \geq 0 \]

\[ \therefore \text{ other path is longer} \]

Therefore adding \( v \) to \( S \) keeps correct distances

Implementing Dijkstra's Algorithm

Need to

- keep current distance values for nodes in \( V-S \)
- find minimum current distance value
- reduce distances when vertex moved to \( S \)

Data Structure Review

Priority Queue:
- Elements each with an associated key
- Operations
  - Insert
  - Find-min
  - Delete-min
- Return the element with the smallest key
  - Return the element with the smallest key and delete it from the data structure
- Decrease-key
- Decrease the key value of some element

Implementations
- Arrays: \( O(n) \) time find/delete-min, \( O(1) \) time insert/decrease-key
- Heaps: \( O(\log n) \) time insert/decrease-key/delete-min, \( O(1) \) time find-min
Dijkstra’s Algorithm with Priority Queues

- For each vertex u not in tree maintain cost of current cheapest path through tree to u
- Store u in priority queue with key = length of this path
- Operations:
  - n-1 insertions (each vertex added once)
  - n-1 delete-mins (each vertex deleted once)
  - pick the vertex of smallest key, remove it from the priority queue and add its edge to the graph
  - <m decrease-keys (each edge updates one vertex)

Priority queue implementations

- Array
  - insert O(1), delete-min O(n), decrease-key O(1)
  - total O(n+n^2+m)=O(n^2)
- Heap
  - insert, delete-min, decrease-key all O(log n)
  - total O(m log n)
- d-Heap (d=min)
  - insert, decrease-key O(log_{m/n} n)
  - delete-min O((m/n) log_{m/n} n)
  - total O(m log_{m/n} n)

Minimum Spanning Trees (Forests)

- Given an undirected graph G=(V,E) with each edge e having a weight w(e)
- Find a subgraph T of G of minimum total weight s.t. every pair of vertices connected in G are also connected in T
- if G is connected then T is a tree otherwise it is a forest

Weighted Undirected Graph

Greedy Algorithm

- Prim’s Algorithm:
  - start at a vertex s
  - add the cheapest edge adjacent to s
  - repeatedly add the cheapest edge that joins the vertices explored so far to the rest of the graph
  - Exactly like Dijkstra’s Algorithm but with a different metric

Dijsktra’s Algorithm

Dijkstra(G,w,s)

S←{s}
d[s]←0
while S≠V do
  if all edges e=(u,v) s.t. v∈S and u∈S select* one with the minimum value of d[u]+w(e)
    S←S∪{v}
    d[v]←d[u]+w(e)
pred[v]←u

*For each v∈S maintain d[v]=minimum value of d[u]+w(e) over all vertices u∈S s.t. e=(u,v) is in of G
Prim’s Algorithm

Prim(G,w,s)
S←{s}
while S≠V do
    of all edges e=(u,v) s.t. v∉S and u∈S select* one with the minimum value of w(e)
    S←S∪{v}
    pred[v]←u

*For each v∈S maintain small[v]=minimum value of w(e) over all vertices u∈S s.t. e=(u,v) is in of G

Second Greedy Algorithm

Kruskal’s Algorithm
- Start with the vertices and no edges
- Repeatedly add the cheapest edge that joins two different components. i.e. that doesn’t create a cycle

Why greed is good

Definition: Given a graph G=(V,E), a cut of G is a partition of V into two non-empty pieces, S and V-S

Lemma: For every cut (S,V-S) of G, there is a minimum spanning tree (or forest) containing any cheapest edge crossing the cut, i.e. connecting some node in S with some node in V-S.
- call such an edge safe

The greedy algorithms always choose safe edges

Prim’s Algorithm
- Always chooses cheapest edge from current tree to rest of the graph
- This is cheapest edge across a cut which has the vertices of that tree on one side.
The greedy algorithms always choose safe edges

- Kruskal's Algorithm
  - Always chooses cheapest edge connecting two pieces of the graph that aren't yet connected
  - This is the cheapest edge across any cut which has those two pieces on different sides and doesn't split any current pieces.

Proof of Lemma:

Suppose you have an MST not using cheapest edge $e$

Endpoints of $e$, $u$ and $v$ must be connected in $T$
Proof of Lemma

Suppose you have an MST not using cheapest edge $e$

Endpoints of $e, u$ and $v$ must be connected in $T$

Proof of Lemma

Replacing $h$ by $e$ does not increase weight of $T$

All the same points are connected by the new tree

Kruskal’s Algorithm

Implementation & Analysis

- First sort the edges by weight $O(m \log m)$
- Go through edges from smallest to largest
  - if endpoints of edge $e$ are currently in different components
  - then add to the graph
  - else skip
- Union-find data structure handles last part
- Total cost of last part: $O(m \alpha(n))$ where $\alpha(n) \ll \log m$
- Overall $O(m \log n)$

Union-find disjoint sets data structure

- Maintaining components
  - start with $n$ different components
    - one per vertex
  - find components of the two endpoints of $e$
    - $2m$ finds
  - union two components when edge connecting them is added
    - $n-1$ unions

Prim’s Algorithm with Priority Queues

- For each vertex $u$ not in tree maintain current cheapest edge from tree to $u$
  - Store $u$ in priority queue with key = weight of this edge
- Operations:
  - $n-1$ insertions (each vertex added once)
  - $n-1$ delete-mins (each vertex deleted once)
    - pick the vertex of smallest key, remove it from the p.q. and add its edge to the graph
  - $<m$ decrease-keys (each edge updates one vertex)

Priority queue implementations

- Array
  - insert $O(1)$, delete-min $O(n)$, decrease-key $O(1)$
  - total $O(n^2 + m) = O(n^2)$
- Heap
  - insert, delete-min, decrease-key all $O(\log n)$
  - total $O(m \log n)$
- $d$-Heap ($d = m/n$)
  - insert, decrease-key $O(\log_{m/n} n)$
  - delete-min $O((m/n) \log_{m/n} n)$
  - total $O(m \log_{m/n} n)$
Boruvka’s Algorithm (1927)
- A bit like Kruskal’s Algorithm
  - Start with \( n \) components consisting of a single vertex each
  - At each step, each component chooses its cheapest outgoing edge to add to the spanning forest
    - Two components may choose to add the same edge
  - Useful for parallel algorithms since components may be processed (almost) independently

Many other minimum spanning tree algorithms, most of them greedy
- Cheriton & Tarjan
  - \( O(m \log \log n) \) time using a queue of components
- Chazelle
  - \( O(m \alpha(m) \log \alpha(m)) \) time
    - Incredibly hairy algorithm
- Karger, Klein & Tarjan
  - \( O(m+n) \) time randomized algorithm that works most of the time

Applications of Minimum Spanning Tree Algorithms
- Minimum cost network design:
  - Build a network to connect all locations \( \{v_1, \ldots, v_n\} \)
  - Cost of connecting \( v_i \) to \( v_j \) is \( w(v_i, v_j) > 0 \)
  - Choose a collection of links to create that will be as cheap as possible
  - Any minimum cost solution is an MST
    - If there is a solution containing a cycle then we can remove any edge and get a cheaper solution

Applications of Minimum Spanning Tree Algorithms
- Maximum Spacing Clustering
  - Given
    - a collection \( U \) of \( n \) objects \( \{p_1, \ldots, p_n\} \)
    - Distance measure \( d(p_i, p_j) \) satisfying
      - \( d(p_i, p_j) = 0 \)
      - \( d(p_i, p_j) > 0 \) for \( i \neq j \)
      - \( d(p_i, p_j) = d(p_j, p_i) \)
    - Positive integer \( k \leq n \)
  - Find a \( k \)-clustering, i.e. partition of \( U \) into \( k \) clusters \( C_1, \ldots, C_k \) such that the spacing between the clusters is as large possible where
    - \( \text{spacing} = \min \{d(p_i, p_j) : p_i \text{ and } p_j \text{ in different clusters}\} \)

Greedy Algorithm
- Start with \( n \) clusters each consisting of a single point
  - Repeatedly find the closest pair of points in different clusters under distance \( d \) and merge their clusters until only \( k \) clusters remain
  - Gets the same components as Kruskal’s Algorithm does!
    - The sequence of closest pairs is exactly the MST
  - Alternatively we could run Kruskal’s algorithm once and for any \( k \) we could get the maximum spacing \( k \)-clustering by deleting the \( k-1 \) most expensive edges

Proof that this works
- Removing the \( k-1 \) most expensive edges from an MST yields \( k \) components \( C_1, \ldots, C_k \) and the spacing for them is precisely the cost \( d^* \) of the \( k-1 \) most expensive edge in the tree
  - Consider any other \( k \)-clustering \( C_1', \ldots, C_k' \)
    - Since they are different and cover the same set of points there is some pair of points \( p_i, p_j \) such that \( p_i, p_j \) are in some cluster \( C \), but \( p_i, p_j \) are in different clusters \( C_i \) and \( C_j \)
      - Since \( p_i, p_j \in C \), and \( p_i, p_j \) have a path between them all of whose edges have distance at most \( d^* \)
        - This path must cross between clusters in the \( C \) clustering so the spacing in \( C' \) is at most \( d^* \)