CHAPTER 4
GREEDY ALGORITHMS

Outline

Content:
- Interval scheduling: The greedy algorithm stays ahead
- Scheduling to minimize lateness: An exchange argument
- Shortest paths
- The minimum spanning tree problem
- Implementing Kruskal's algorithm: Union-find

Reading:
- Chapter 4

Greedy Algorithms

An algorithm is greedy if it builds up a solution in small steps, choosing a decision at each step myopically to optimize some underlying criterion.

It's easy to invent greedy algorithms for almost any problem.
- Intuitive and fast
- Usually not optimal

It's challenging to prove greedy algorithms succeed in solving a nontrivial problem optimally.
1. The greedy algorithm stays ahead.
2. An exchange argument.

Interval Scheduling

The greedy algorithm stays ahead
The Interval Scheduling Problem

- Given: Set of requests \( \{1, 2, \ldots, n\} \), \( i \)th request corresponds an interval with start time \( s(i) \) and finish time \( f(i) \)
  - interval \( i: [s(i), f(i)] \) requests don’t overlap
- Goal: Find a compatible subset of the requests with maximum size

Greedy algorithms

The Greedy Algorithm

- The 4th greedy rule leads to the optimal solution.
  - We first accept the request that finish first
  - Natural idea: Our resource becomes free ASAP
- The greedy algorithm:
  
  ```
  Interval-Scheduling(R)
  // R: undetermined requests; A: accepted requests
  1. A = ∅;
  2. while (R is not empty) do
     3. choose a request \( i \in R \) with minimum \( f(i) \) // greedy rule
     4. A = A + \{i\}
     5. R = R – \{i\} – X, where X = \{j: j \in R and j is not compatible with i\}
  6. return A
  ```

Greedy algorithms

The Greedy Algorithm Stays Ahead

- Q: How to prove optimality?
  - Let \( O \) be an optimal solution. Prove \( A = O \)? Prove \( |A| = |O| \)?
- The greedy algorithm stays ahead.
  - We will compare the partial solutions of \( A \) and \( O \), and show that the greedy algorithm is doing better in a step-by-step fashion.
  - Let \( A \) be the output of the greedy algorithm, \( A = \{i_1, \ldots, i_k\} \), in the order they were added. Let \( O \) be the optimal solution, \( O = \{j_1, \ldots, j_m\} \) in the ascending order of start (finish) times.
    - For all indices \( r \leq k \), we have \( f(i_r) \leq f(j_r) \).
- Pf: Proof by induction!
  - Basis step: true for \( r = 1 \), \( f(i_1) \leq f(j_1) \).
  - Induction step: hypothesis: true for \( r-1 \).
    - \( f(i_{r-1}) \leq f(j_{r-1}) \)
    - \( O \) is compatible, \( f(i_{r-1}) \leq s(j_{r-1}) \)
    - Hence, \( f(i_{r-1}) \leq s(j_{r}) \) \( j_{r} \in R \) after \( i_{r-1} \) is selected in line 5.
    - According to line 3, \( f(i_r) \leq f(j_r) \).
The Greedy Algorithm Is Optimal

- The greedy algorithm returns an optimal set $A$.
- Pf: Proof by contradiction.
  - If $A$ is not optimal, then an optimal set $O$ must have more requests, i.e., $|O| = m > k = |A|$.
  - Since $f(i_k) \leq f(j_k)$ and $m > k$, there is a request $j_{k+1}$ in $O$.
  - $f(j_k) \leq s(j_{k+1})$: $f(i_k) < s(i_{k+1})$.
  - Hence, $j_{k+1}$ is compatible with $i_k$. $R$ should contain $j_{k+1}$.
  - However, the greedy algorithm stops with request $i_k$, and it is only supposed to stop when $R$ is empty.

Implementation: The Greedy Algorithm

- Interval-Scheduling($R$)
  // $R$: undetermined requests; $A$: accepted requests
  1. $A = \emptyset$
  2. while ($R$ is not empty) do
  3. choose a request $i \in R$ with minimum $f(i)$ \ // greedy rule
  4. $A = A + \{i\}$
  5. $R = R - \{i\} - X$, where $X = \{j: j \in R$ and $j$ is not compatible with $i\}$
  6. return $A$

- Running time: From $O(n^2)$ to $O(n \log n)$
  - Initialization:
    - $O(n \log n)$: sort $R$ in ascending order of $f(i)$
    - $O(n)$: construct $S$, $S[i] = s(i)$
  - Lines 3 and 5:
    - $O(n)$: scan $R$ once
    - We do not delete all incompatible requests in line 5, only those finish after the next selected request finishes and start before the next selected request finishes.
What if We Have Multiple Resources?

- The interval partitioning problem:
  - A.K.A. the interval coloring problem: one resource = one color
  - Use as few resources as possible

- Given: Set of requests \( \{1, 2, \ldots, n\} \), \( i^{th} \) request corresponds to an interval with start time \( s(i) \) and finish time \( f(i) \)
  - Interval \( i: [s(i), f(i)] \)

- Goal: Partition these requests into a minimum number of compatible subsets, each corresponds to one resource

Can We Reach the Lower Bound?

- The depth \( d \) is the lower bound on the number of required resources.
- Q: Can we always use \( d \) resources to schedule all requests?
  - A: Yes.

- Q: How to prove the optimality?
  - A: One finds a bound that every possible solution must have at least a certain value, and then one shows that the algorithm under consideration always achieves this bound.

The Greedy Algorithm

- Assign a label to each interval. Possible labels: \( \{1, 2, \ldots, d\} \).
- Assign different labels for overlapping intervals.

```
Interval-Partitioning(R)
1. \( \{I_1, \ldots, I_n\} = \) sort intervals in ascending order of their start times
2. for \( j \) from 1 to \( n \) do
3.   exclude the labels of all assigned intervals that are not compatible with \( I_j \)
4.   if there is a nonexcluded label from \( \{1, 2, \ldots, d\} \) then
5.     assign a nonexcluded label to \( I_j \)
6.   else leave \( I_j \) unlabeled
```

- Implementation:
  - Lines 3–5: find a resource compatible with \( I_j \), assign this label
    - Record the finish time of the last added interval for each label
    - Compatibility checking: \( s(I_j) \geq f(\text{label}) \)
    - Use priority queue to maintain labels
The Interval Partitioning Problem

Greedy algorithms

Optimality (1/2)

- The greedy algorithm assigns every interval a label, and no two overlapping intervals receive the same label.

- Pf:
  1. No interval ends up unlabeled.
     - Suppose interval \( I_j \) overlaps \( t \) intervals earlier in the sorted list.
     - These \( t + 1 \) intervals pass over a common point, namely \( s(I_j) \).
     - Hence, \( t + 1 \leq d \). Thus, \( t \leq d - 1 \); at least one of the \( d \) labels is not excluded, and so there is a label that can be assigned to \( I_j \).
     - i.e., line 6 never occurs!

Interval-Partitioning(\( R \))
1. \( \{I_1, \ldots, I_n\} = \) sort intervals in ascending order of their start times
2. \( \text{for } j \text{ from } 1 \text{ to } n \) do
3. \( \text{exclude the labels of all assigned intervals that are not compatible with } I_j \)
4. \( \text{if (there is a nonexcluded label from } \{1, 2, \ldots, d\} \text{) then} \)
5. \( \text{assign a nonexcluded label to } I_j \)
6. \( \text{else leave } I_j \text{ unlabeled} \)

Optimality (2/2)

- Pf (cont’d)
  2. No two overlapping intervals are assigned with the same label.
     - Consider any two intervals \( I_i \) and \( I_j \) that overlap, \( i<j \).
     - When \( I_j \) is considered (in line 2), \( I_i \) is in the set of intervals whose labels are excluded (in line 3).
     - Hence, the algorithm will not assign to \( I_j \) the label used for \( I_i \).
     - Since the algorithm uses \( d \) labels, we can conclude that the greedy algorithm always uses the minimum possible number of labels, i.e., it is optimal!

Interval-Partitioning(\( R \))
1. \( \{I_1, \ldots, I_n\} = \) sort intervals in ascending order of their start times
2. \( \text{for } j \text{ from } 1 \text{ to } n \) do
3. \( \text{exclude the labels of all assigned intervals that are not compatible with } I_j \)
4. \( \text{if (there is a nonexcluded label from } \{1, 2, \ldots, d\} \text{) then} \)
5. \( \text{assign a nonexcluded label to } I_j \)
6. \( \text{else leave } I_j \text{ unlabeled} \)

Scheduling to Minimize Lateness

An exchange argument

Greedy algorithms
What If Each Request Has a Deadline?

- **Given**: A single resource is available starting at time $s$. A set of requests $\{1, 2, \ldots, n\}$, request $i$ requires a contiguous interval of length $t_i$ and has a deadline $d_i$.
- **Goal**: Schedule all requests without overlapping so as to minimize the maximum lateness.

  - Lateness: $l_i = \max\{0, f(i) - d_i\}$.

Greedy Rule

- **Consider requests in some order.**
  1. **Shortest interval first**: Process requests in ascending order of $t_i$.
     
    | $i$ | $t_i$ | $d_i$  |
    |-----|-------|-------|
    | 1   | 1     | 10    |
    | 2   | 2     | 10    |
  2. **Smallest slack**: Process requests in ascending order of $d_i - t_i$.
    
    | $i$ | $t_i$ | $d_i$  |
    |-----|-------|-------|
    | 1   | 1     | 10    |
    | 2   | 2     | 10    |
  3. **Earliest deadline first**: Process requests in ascending order of $d_i$.

Minimizing Lateness

- **Greedy rule: Earliest deadline first!**

Min-Lateness($R, s$)

- $f$: the finishing time of the last scheduled request

1. $(d_1, \ldots, d_n) = \text{sort requests in ascending order of their deadlines}$
2. $f = s$
3. for $i$ from 1 to $n$
   4. assign request $i$ to the time interval from $s(i) = f$ to $f(i) = f + t_i$
   5. $f = f + t_i$
4. return the set of scheduled intervals $[s(i), f(i))$ for all $i = 1..n$

No Idle Time

- **Observation**: The greedy schedule has no idle time.

- **Line 4!**

Greedy algorithms

- **There is an optimal schedule with no idle time.**

Min-Lateness($R, s$)

- $f$: the finishing time of the last scheduled request

1. $(d_1, \ldots, d_n) = \text{sort requests in ascending order of their deadlines}$
2. $f = s$
3. for $i$ from 1 to $n$
   4. assign request $i$ to the time interval from $s(i) = f$ to $f(i) = f + t_i$
   5. $f = f + t_i$
4. return the set of scheduled intervals $[s(i), f(i))$ for all $i = 1..n$

Greedy algorithms

- **Jobs with earlier deadlines get completed earlier.**
No Inversions

- **Exchange argument:** Gradually transform an optimal solution to the one found by the greedy algorithm without hurting its quality.
- An inversion in schedule $S$ is a pair of requests $i$ and $j$ such that $s(i) < s(j)$ but $d_j < d_i$.
- All schedules without inversions and without idle time have the same maximum lateness.

**Pf:**
- If two different schedules have neither inversions nor idle time, then they can only differ in the order in which requests with identical deadlines are scheduled.
- Consider such a deadline $d$. In both schedules, the jobs with deadline $d$ are all scheduled consecutively (after all jobs with earlier deadlines and before all jobs with later deadlines).
- Among them, the last one has the greatest lateness, and this lateness does not depend on the order of the requests.

Optimality

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

**Pf:**
- There is an optimal schedule $O$ without idle time. (done!)
- If $O$ has an inversion, there is a pair of jobs $i$ and $j$ such that $j$ is scheduled immediately after $i$ and has $d_j < d_i$.
- After swapping $i$ and $j$ we get a schedule with one less inversion.
- The new swapped schedule has a maximum lateness no larger than that of $O$.
- Other requests have the same lateness.

Optimality: Exchange Argument

- **Theorem:** The greedy schedule $S$ is optimal.
- **Pf:** Proof by contradiction
  - Let $O$ be an optimal schedule with inversions.
  - Assume $O$ has no idle time.
  - If $O$ has no inversions, then $S = O$. done!
  - If $O$ has an inversion, let $i$-$j$ be an adjacent inversion.
    - Swapping $i$ and $j$ does not increase the maximum lateness and strictly decreases the number of inversions.
    - This contradicts definition of $O$.

Summary: Greedy Analysis Strategies

- An algorithm is greedy if it builds up a solution in small steps, choosing a decision at each step myopically to optimize some underlying criterion.
- It's challenging to prove greedy algorithms succeed in solving a nontrivial problem optimally.
  1. The greedy algorithm stays ahead: Show that after each step of the greedy algorithm, its partial solution is better than the optimal.
  2. An exchange argument: Gradually transform an optimal solution to the one found by the greedy algorithm without hurting its quality.
Edsger W. Dijkstra 1959

Edsger W. Dijkstra (1930—2002)

- 1972 recipient of the ACM Turing Award

If you want more effective programmers, you will discover that they should not waste their time debugging, they should not introduce the bugs to start with.

Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence.

-- Turing Award Lecture 1972, the humble programmer

Google Map

Shortest path from San Francisco Shopping Centre to Yosemite National Park

Floor Guide in a Shopping Mall

Direct shoppers to their destinations in real-time

---

http://www.cs.utexas.edu/users/EWD/obituary.html
The Shortest Path Problem

- Given:
  - Directed graph \( G = (V, E) \)
  - Length \( l_e \): length of edge \( e = (u, v) \in E \)
  - Distance; time; cost
  - \( l_e \geq 0 \)
  - Q: what if undirected?
  - A: 1 undirected edge = 2 directed ones
  - Source \( s \)

- Goal:
  - Shortest path \( P_v \) from \( s \) to each other node \( v \in V - \{s\} \)
  - Length of path \( P \): \( l(P) = \sum_{e \in P} l_e \)

---

Dijkstra’s Algorithm

- \( \text{Dijkstra}(G, l) \)
- \( S \): the set of explored nodes
- \( \forall u \in S \), we store a shortest path distance \( d(u) \) from \( s \) to \( u \)
  1. initialize \( S = \{s\} \), \( d(s) = 0 \)
  2. while \( S \neq V \) do
  3. select a node \( v \notin S \) with at least one edge from \( S \) for which
  4. \( d'(v) = \min_{e = (u, v), u \in S} d(u) + l_e \)
  5. add \( v \) to \( S \) and define \( d(v) = d'(v) \)

---

Correctness

- Loop invariant: Consider the set \( S \) at any point in the algorithm’s execution. For each node \( u \in S \), \( d(u) \) is the length of the shortest \( s-u \) path \( P_u \).
- \( \text{Pf: Proof by induction on } |S| \)
  - Basis step: trivial for \( |S| = 1 \).
  - Induction step: hypothesis: true for \( k \geq 1 \).
    - Grow \( S \) by adding \( v \):
      - let \( (u, v) \) be the final edge on our \( s-v \) path \( P_v \).
      - By induction hypothesis, \( P_u \) is the shortest \( s-u \) path.
      - Consider any other \( s-v \) path \( P \); \( P \) must leave \( S \) somewhere; let \( y \) be the first node on \( P \) that is not in \( S \), and \( x \in S \) be the node just before \( y \).
      - \( P \) cannot be shorter than \( P_v \) because it is already at least as long as \( P_v \) by the time it has left the set \( S \).
      - At iteration \( k+1 \), \( d(v) = d'(v) = d(u) + l_{e=(u, v)} \leq d(x) + l_{e=(x,y)} \leq l(P) \)

---

Implementation

- Q: How to do line 4 efficiently? \( d'(v) = \min_{e = (u, v), u \in S} d(u) + l_e \)
- A: Explicitly maintain \( d'(v) \) in the view of each unexplored node \( v \) instead of \( S \)
  - Next node to explore = node with minimum \( d'(v) \).
  - When exploring \( v \), update \( d'(w) \) for each outgoing edge \((v, w)\), \( w \not\in S \).
- Q: How?
- A: Implement a min priority queue nicely

<table>
<thead>
<tr>
<th>Operation</th>
<th>Dijkstra Array</th>
<th>Binary heap</th>
<th>Fibonacci heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ExtractMin</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ChangeKey</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>isEmpty</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

Greedy algorithms

**Example**

Greedy algorithms

**Priority Queue**

- In a priority queue (PQ)
  - Each element has a priority (key)
  - Only the element with highest (or lowest) priority can be deleted
    - Max priority queue, or min priority queue
  - An element with arbitrary priority can be inserted into the queue at any time

<table>
<thead>
<tr>
<th>Operation</th>
<th>Binary heap</th>
<th>Fibonacci heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>FindMin</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>ExtractMin</td>
<td>$\Theta(lg n)$</td>
<td>$O(lg n)$</td>
</tr>
<tr>
<td>Insert</td>
<td>$\Theta(lg n)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>ChangeKey</td>
<td>$\Theta(lg n)$</td>
<td>$\Theta(1)$</td>
</tr>
</tbody>
</table>

The time complexities are worst-case time for binary heap, and amortized time complexity for Fibonacci heap.

Greedy algorithms

**Heap**

- Definition: A max (min) heap is
  - A max (min) tree: key[parent] >= (<=) key[children]
  - A complete binary tree
- Corollary: Who has the largest (smallest) key in a max (min) heap?
  - Root!
- Example

<table>
<thead>
<tr>
<th>Max heap</th>
<th>Min heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>12</td>
<td>7</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

Greedy algorithms

**Recap Heaps: Priority Queues**

Binary Tree Application

Greedy algorithms

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein

Class MaxHeap

- Implementation?
  - Complete binary tree ⇒ array representation

```
14
12 7
10 8 6
```

Insertion into a Max Heap (1/3)

- Maintain heap property all the times
  - Push(1)

```
14
10 2
```

Insertion into a Max Heap (2/3)

- Push(21)

```
20
15 2
```

Insertion into a Max Heap (3/3)

- Time complexity?
  - How many times to bubble up in the worst case?
  - Tree height: $\Theta(\log n)$
Deletion from a Max Heap (1/3)

- Maintain heap ⇒ trickle down if needed!
- Pop()

Deletion from a Max Heap (2/3)

- Maintain heap ⇒ trickle down if needed!
- Pop()

Deletion from a Max Heap (3/3)

- Time complexity?
  - How many times to trickle down in the worst case? Θ(lg n)

Max Heapify

- Max (min) heapify = maintain the max (min) heap property
  - What we do to trickle down the root in deletion
  - Assume i's left & right subtrees are heaps
    - But key[i] may be < (> key[children]
  - Heapify i = trickle down key[i]
    ⇒ the tree rooted at i is a heap
How to Build a Max Heap?

- How to convert any complete binary tree to a max heap?
- Intuition: Max heapify in a bottom-up manner
  - Induction basis: Leaves are already heaps
  - Induction steps: Start at parents of leaves, work upward till root
- Time complexity: $O(n \log n)$

Greedy algorithms

Minimum Spanning Trees

- Robert C. Prim 1957 (Dijkstra 1959)
- Joseph B. Kruskal 1956
- Reverse-delete

Minimum Spanning Graphs

- Q: How can a cable TV company lay cable to a new neighborhood, of course, as cheaply as possible?
- A: Curiously and fortunately, this problem is a case where many greedy algorithms optimally solve.

- Given
  - Undirected graph $G = (V, E)$
  - Nonnegative cost $c_e$ for each edge $e = (u, v) \in V$
  - $c_e \geq 0$
- Goal
  - Find a subset of edges $T \subseteq E$ so that
    - The subgraph $(V, T)$ is connected
    - Total cost $\sum_{e \in V} c_e$ is minimized

Minimum Spanning ?????

- Q: Let $T$ be a minimum-cost solution. What should $(V, T)$ be?
- A:
  - By definition, $(V, T)$ must be connected.
  - We show that it also contains no cycles.
  - Suppose it contained a cycle $C$, and let $e$ be any edge on $C$.
  - We claim that $(V, T \setminus \{e\})$ is still connected.
  - Any path previously used $e$ can now go path $C \setminus \{e\}$ instead.
  - It follows that $(V, T \setminus \{e\})$ is also a valid solution, but cheaper.
  - Hence, $(V, T)$ is a tree.
- Goal
  - Find a subset of edges $T \subseteq E$ so that
    - $(V, T)$ is a tree,
    - Total cost $\sum_{e \in V} c_e$ is minimized.
Greedy Algorithms (1/3)

Q: What will you do?

- All three greedy algorithms produce an MST.
- Kruskal’s algorithm:
  - Start with \( T = \{ \} \).
  - Consider edges in ascending order of cost.
  - Insert edge \( e \) in \( T \) as long as it does not create a cycle; otherwise, discard \( e \) and continue.

Greedy Algorithms (2/3)

Q: What will you do?

- All three greedy algorithms produce an MST.
- Prim’s algorithm: (c.f. Dijkstra’s algorithm)
  - Start with a root node \( s \).
  - Greedily grow a tree \( T \) from \( s \) outward.
  - At each step, add the cheapest edge \( e \) to the partial tree \( T \) that has exactly one endpoint in \( T \).

Greedy Algorithms (3/3)

Q: What will you do?

- All three greedy algorithms produce an MST.
- Reverse-delete algorithm: (reverse of Kruskal’s algorithm)
  - Start with \( T = E \).
  - Consider edges in descending order of cost.
  - Delete edge \( e \) from \( T \) unless doing so would disconnect \( T \).

Cut Property (1/2)

Q: What’s wrong with this proof?

- A: Take care about the definition!

- Simplifying assumption: All edge costs \( c_e \) are distinct.
- Q: When is it safe to include an edge in the MST?

- Cut Property: Let \( S \) be any subset of nodes, and let \( e = (v, w) \) be the minimum cost edge with one end in \( S \) and the other in \( V – S \). Then every MST contains \( e \).

- Pf: Exchange argument!
  - Let \( T \) be a spanning tree that does not contain \( e \). We need to show that \( T \) does not have the minimum possible cost.
  - Since \( T \) is a spanning tree, it must contain an edge \( f \) with one end in \( S \) and the other in \( V – S \).
  - Since \( e \) is the cheapest edge with this property, we have \( c_e < c_f \).
  - Hence, \( T – \{f\} + \{e\} \) is a spanning tree that is cheaper than \( T \).

- Q: What’s wrong with this proof?
- A: Take care about the definition!
**Cut Property (2/2)**

**Cut Property:** Let \( S \) be any subset of nodes, and let \( e = (v, w) \) be the minimum cost edge with one end in \( S \) and the other in \( V - S \). Then every MST contains \( e \).

**Pf:** Exchange argument!
- Let \( T \) be a spanning tree that does not contain \( e \).
- \( T \) is a spanning tree; \( \exists \) path \( P \in T \) from \( v \) to \( w \).
- Let \( e' = (v', w') \) on \( P \), \( v' \in S \) and \( w' \in V - S \).
- \( T' = T - \{ e' \} + \{ e \} \) is a spanning tree
  - \( (V, T') \) must be connected:
    - \( (V, T') \) is connected, any path in \( (V, T) \) using \( e' \) can be rerouted in \( (V, T) \) by \( v' \rightarrow v \), \( (v, w) \), \( w \rightarrow w' \).
  - \( (V, T') \) must be acyclic:
    - The only cycle in \( (V, T' + \{ e \}) \) is \( e + P \), it isn’t in \( (V, T) \)
- Since \( c_e < c_{e'} \), \( T' \) is cheaper than \( T \).

**Cycle Property**

**Q:** When is it safe to exclude an edge out?

**Cycle Property:** Let \( C \) be any cycle in \( G \), and let \( e = (v, w) \) be the maximum cost edge in \( C \). Then \( e \) does not belong to any MST.

**Pf:** Exchange argument! (Similar to Cut Property)

---

**Prim’s Example**

**R. C. Prim, 1957**

**Procedure:**
- Start with a root node \( s \).
- Greedily grow a tree \( T \) from \( s \) outward.
- At each step, add the cheapest edge \( e \) to the partial tree \( T \) that has exactly one endpoint in \( T \).
**Prim’s Algorithm**

Dijkstra(G, l)
1. // S: the set of explored nodes
2. // for each u ∈ S, we store a shortest path distance d(u) from s to u
3. initialize S = {s}, d(s) = 0
4. while S ≠ V do
5.  select a node v ≠ S with at least one edge from S for which
6.  d'(v) = min_{e = (u, v), u ∈ S} d(u) + l_e
7.  add v to S and define d(v) = d'(v)

Q: How to change Dijkstra’s algorithm to Prim’s?
Q: How to implement?

**Kruskal’s Algorithm**

J. B. Kruskal, 1956

Procedure:
1. Start with T = {}.
2. Consider edges in ascending order of cost.
3. Insert edge e in T as long as it does not create a cycle; otherwise, discard e and continue.

Kruskal(G, c)
1. \{e_1, e_2, ..., e_n\} = sort edges in ascending order of their costs
2. T = {}.
3. for each e_i = (u, v) do
4.  if (u and v are not connected by edges in T) then // different subtrees
5.    T = T + {e_i} // merge these two corresponding subtrees

**The Union-Find Data Structure (1/2)**

- Union-find data structure represents disjoint sets
  - Disjoint sets: elements are disjoint
  - Each set has a representative
  - Operations:
    - MakeUnionFind(S): initialize a set for each element in S
    - Find(u): return the representative of the set containing u
    - Union(A, B): merge sets A and B

**The Union-Find Data Structure (2/2)**

- Implementation: disjoint-set forest
  - Representative is the root; link: from children to parent
  - Union: attach the smaller to the larger one (union by rank)
  - Find: trace back to root and redirect the link (path compression)
- Running time: union by rank + path compression
  - The amortized running time per operation is O(α(n)), α(n) < 5 !!
  - Average running time of a sequence of n operations

**Greedy algorithms**


Implementing Kruskal’s Algorithm

Kruskal\((G,c)\)
1. \(\{e_1, e_2, \ldots, e_m\}\) = sort edges in ascending order of their costs
2. \(T = \{\}\\)
3. for each \(e_i = (u, v)\) do
4. if \((u\) and \(v\) are not connected by edges in \(T\)) then // different subtrees
5. \(T = T + \{e_i\}\) // merge these two corresponding subtrees

- **Use the union-find data structure**
  - Maintain a disjoint set for each connected component (subtree)

- Line 1: sort edge costs
- Line 4: “Find” twice for each edge (total \(m\) edges in \(G\))
- Line 5: “Union” possibly once for each edge (total \(n-1\) edges in \(T\))

- Comparison sort + simple disjoint set: \(O(m \log n)\)
- Linear sort + union-find: \(O(m \alpha(m, n))\)

Greedy algorithms