Greedy Graph Algorithms

T. M. Murali

September 16, 21, 23, and 28, 2009

Shortest Path Problem

- $G(V, E)$ is a connected directed graph. Each edge e has a length $l_e \geq 0$.
- V has n nodes and E has m edges.
- *Length of a path* P is the sum of the lengths of the edges in P.
- Goal is to determine the shortest path from a specified start node s to each node in V.
- Aside: If G is undirected, convert to a directed graph by replacing each edge in G by two directed edges.

Shortest Path Problem

- $G(V, E)$ is a connected directed graph. Each edge e has a length $l_e \geq 0$.
- V has n nodes and E has m edges.
- *Length of a path* P is the sum of the lengths of the edges in P.
- Goal is to determine the shortest path from a specified start node s to each node in V.
- Aside: If G is undirected, convert to a directed graph by replacing each edge in G by two directed edges.

SHORTEST PATHS

INSTANCE: A directed graph $G(V, E)$, a function $l : E \rightarrow \mathbb{R}^+$, and a node $s \in V$

SOLUTION: A set $\{P_u, u \in V\}$, where P_u is the shortest path in G from s to u.

Example of Dijkstra's Algorithm

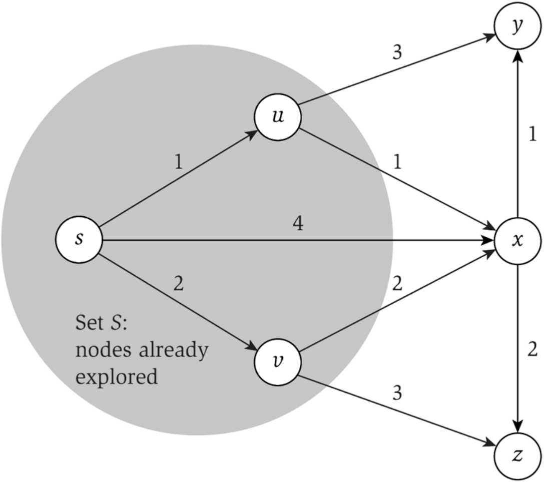

Figure 4.7 A snapshot of the execution of Dijkstra's Algorithm. The next node that will be added to the set S is x, due to the path through u.

Dijkstra's Algorithm

- Maintain a set S of explored nodes: for each node $u \in S$, we have determined the length $d(u)$ of the shortest path from s to u.
- "Greedily" add a node v to S that is closest to s.

Dijkstra's Algorithm

- Maintain a set S of explored nodes: for each node $u \in S$, we have determined the length $d(u)$ of the shortest path from s to u.
- "Greedily" add a node v to S that is closest to s.

```
Dijkstra's Algorithm (G, ℓ)
Let S be the set of explored nodes
    For each u ∈ S, we store a distance d(u)
Initially S = {s} and d(s) = 0
While S ≠ V
    Select a node v ∉ S with at least one edge from S for which
        d′(v) = min_{e=(u,v):u∈S} d(u) + ℓ_e is as small as possible
    Add v to S and define d(v) = d′(v)
EndWhile
```

Dijkstra's Algorithm

- Maintain a set S of explored nodes: for each node $u \in S$, we have determined the length $d(u)$ of the shortest path from s to u.
- "Greedily" add a node v to S that is closest to s.

Dijkstra's Algorithm (G, ℓ)

Let S be the set of explored nodes

  For each $u \in S$, we store a distance $d(u)$

Initially $S = \{s\}$ and $d(s) = 0$

While $S \neq V$

  Select a node $v \notin S$ with at least one edge from S for which

    $d'(v) = \min_{e=(u,v):u \in S} d(u) + \ell_e$ is as small as possible

  Add v to S and define $d(v) = d'(v)$

EndWhile

- $d'(v)$ = length of shortest path from s to v using only nodes in S.
- To compute the shortest paths:

Dijkstra's Algorithm

- Maintain a set S of explored nodes: for each node $u \in S$, we have determined the length $d(u)$ of the shortest path from s to u.
- "Greedily" add a node v to S that is closest to s.

Dijkstra's Algorithm (G, ℓ)

Let S be the set of explored nodes

For each $u \in S$, we store a distance $d(u)$

Initially $S = \{s\}$ and $d(s) = 0$

While $S \neq V$

Select a node $v \notin S$ with at least one edge from S for which

$d'(v) = \min_{e=(u,v):u \in S} d(u) + \ell_e$ is as small as possible

Add v to S and define $d(v) = d'(v)$

EndWhile

- $d'(v) =$ length of shortest path from s to v using only nodes in S.
- To compute the shortest paths: store the predecessor u that minimises $d'(v)$.

Proof of Correctness

- Let P_u be the path computed for a node u.
- Claim: P_u is the shortest path from s to u.
- Prove by induction on the size of S.

Proof of Correctness

- Let P_u be the path computed for a node u.
- Claim: P_u is the shortest path from s to u.
- Prove by induction on the size of S.
  - Base case: $|S| = 1$. The only node in S is s.
  - Inductive step: we add the node v to S. Let u be the v's predecessor on the path P_v. Could there be a shorter path P from s to v?

Proof of Correctness

- Let P_u be the path computed for a node u.
- Claim: P_u is the shortest path from s to u.
- Prove by induction on the size of S.
  - Base case: $|S| = 1$. The only node in S is s.
  - Inductive step: we add the node v to S. Let u be the v's predecessor on the path P_v. Could there be a shorter path P from s to v?

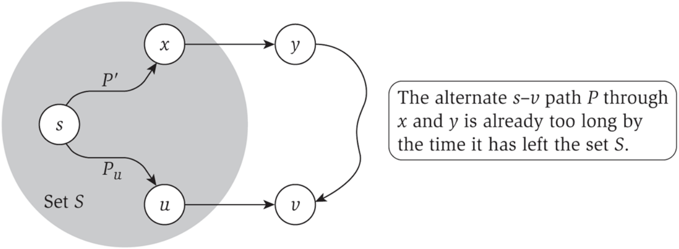

Figure 4.8 The shortest path P_v and an alternate s-v path P through the node y.

Comments about Dijkstra's Algorithm

- Algorithm cannot handle negative edge lengths. We will discuss the Bellman-Ford algorithm in a few weeks.
- Union of shortest paths output form a tree. Why?

Implementing Dijkstra's Algorithm

```
Dijkstra's Algorithm (G, ℓ)
Let S be the set of explored nodes
    For each u ∈ S, we store a distance d(u)
Initially S = {s} and d(s) = 0
While S ≠ V
    Select a node v ∉ S with at least one edge from S for which
       d'(v) = min_{e=(u,v):u∈S} d(u) + ℓ_e is as small as possible
    Add v to S and define d(v) = d'(v)
EndWhile
```

- How many iterations are there of the while loop?

Implementing Dijkstra's Algorithm

```
Dijkstra's Algorithm (G, ℓ)
Let S be the set of explored nodes
    For each u ∈ S, we store a distance d(u)
Initially S = {s} and d(s) = 0
While S ≠ V
    Select a node v ∉ S with at least one edge from S for which
      d'(v) = min_{e=(u,v):u∈S} d(u) + ℓ_e is as small as possible
    Add v to S and define d(v) = d'(v)
EndWhile
```

- How many iterations are there of the while loop? $n - 1$.

Implementing Dijkstra's Algorithm

```
Dijkstra's Algorithm (G, ℓ)
Let S be the set of explored nodes
    For each u ∈ S, we store a distance d(u)
Initially S = {s} and d(s) = 0
While S ≠ V
    Select a node v ∉ S with at least one edge from S for which
        d'(v) = min_{e=(u,v):u∈S} d(u) + ℓ_e is as small as possible
    Add v to S and define d(v) = d'(v)
EndWhile
```

- How many iterations are there of the while loop? $n - 1$.
- In each iteration, for each node $v \notin S$, compute

$$d'(v) = \min_{e=(u,v), u \in S} d(u) + l_e$$

.

Implementing Dijkstra's Algorithm

```
Dijkstra's Algorithm (G, ℓ)
Let S be the set of explored nodes
    For each u ∈ S, we store a distance d(u)
Initially S = {s} and d(s) = 0
While S ≠ V
    Select a node v ∉ S with at least one edge from S for which
        d'(v) = min_{e=(u,v):u∈S} d(u) + ℓ_e is as small as possible
    Add v to S and define d(v) = d'(v)
EndWhile
```

- How many iterations are there of the while loop? $n - 1$.
- In each iteration, for each node $v \notin S$, compute

$$d'(v) = \min_{e=(u,v), u \in S} d(u) + l_e$$

.

- Running time per iteration is

Implementing Dijkstra's Algorithm

```
Dijkstra's Algorithm (G, ℓ)
Let S be the set of explored nodes
    For each u ∈ S, we store a distance d(u)
Initially S = {s} and d(s) = 0
While S ≠ V
    Select a node v ∉ S with at least one edge from S for which
       d'(v) = min_{e=(u,v):u∈S} d(u) + ℓ_e is as small as possible
    Add v to S and define d(v) = d'(v)
EndWhile
```

- How many iterations are there of the while loop? $n-1$.
- In each iteration, for each node $v \notin S$, compute

$$d'(v) = \min_{e=(u,v), u \in S} d(u) + l_e$$

.

- Running time per iteration is $O(m) \Rightarrow$ overall running time is $O(nm)$.

A Faster implementation of Dijkstra's Algorithm

```
Dijkstra's Algorithm (G, ℓ)
Let S be the set of explored nodes
    For each u ∈ S, we store a distance d(u)
Initially S = {s} and d(s) = 0
While S ≠ V
    Select a node v ∉ S with at least one edge from S for which
        d'(v) = min_{e=(u,v):u∈S} d(u) + ℓ_e is as small as possible
    Add v to S and define d(v) = d'(v)
EndWhile
```

- Observation: If we add v to S, $d'(w)$ changes only for v's neighbours.

A Faster implementation of Dijkstra's Algorithm

```
Dijkstra's Algorithm (G, ℓ)
Let S be the set of explored nodes
    For each u ∈ S, we store a distance d(u)
Initially S = {s} and d(s) = 0
While S ≠ V
    Select a node v ∉ S with at least one edge from S for which
        d'(v) = min_{e=(u,v):u∈S} d(u) + ℓ_e is as small as possible
    Add v to S and define d(v) = d'(v)
EndWhile
```

- Observation: If we add v to S, $d'(w)$ changes only for v's neighbours.
- Store the minima $d'(v)$ for each node $v \in V - S$ in a priority queue.
- Determine the next node v to add to S using ExtractMin.
- After adding v to S, for each neighbour w of v, compute $d(v) + l_{(v,w)}$.
- If $d(v) + l_{(v,w)} < d'(w)$,
    1. Set $d'(w) = d(v) + l_{(v,w)}$.
    2. Update w's key to the new value of $d'(w)$ using ChangeKey.

A Faster implementation of Dijkstra's Algorithm

```
Dijkstra's Algorithm (G, ℓ)
Let S be the set of explored nodes
    For each u ∈ S, we store a distance d(u)
Initially S = {s} and d(s) = 0
While S ≠ V
    Select a node v ∉ S with at least one edge from S for which
        d'(v) = min_{e=(u,v):u∈S} d(u) + ℓ_e is as small as possible
    Add v to S and define d(v) = d'(v)
EndWhile
```

- Observation: If we add v to S, $d'(w)$ changes only for v's neighbours.
- Store the minima $d'(v)$ for each node $v \in V - S$ in a priority queue.
- Determine the next node v to add to S using ExtractMin.
- After adding v to S, for each neighbour w of v, compute $d(v) + l_{(v,w)}$.
- If $d(v) + l_{(v,w)} < d'(w)$,
    1. Set $d'(w) = d(v) + l_{(v,w)}$.
    2. Update w's key to the new value of $d'(w)$ using ChangeKey.
- How many times are ExtractMin and ChangeKey invoked?

A Faster implementation of Dijkstra's Algorithm

```
Dijkstra's Algorithm (G, ℓ)
Let S be the set of explored nodes
    For each u ∈ S, we store a distance d(u)
Initially S = {s} and d(s) = 0
While S ≠ V
    Select a node v ∉ S with at least one edge from S for which
        d'(v) = min_{e=(u,v):u∈S} d(u) + ℓ_e is as small as possible
    Add v to S and define d(v) = d'(v)
EndWhile
```

- Observation: If we add v to S, $d'(w)$ changes only for v's neighbours.
- Store the minima $d'(v)$ for each node $v \in V - S$ in a priority queue.
- Determine the next node v to add to S using ExtractMin.
- After adding v to S, for each neighbour w of v, compute $d(v) + l_{(v,w)}$.
- If $d(v) + l_{(v,w)} < d'(w)$,
  1. Set $d'(w) = d(v) + l_{(v,w)}$.
  2. Update w's key to the new value of $d'(w)$ using ChangeKey.
- How many times are ExtractMin and ChangeKey invoked? $n - 1$ and m times, respectively.

A Faster implementation of Dijkstra's Algorithm

```
Dijkstra's Algorithm (G, ℓ)
Let S be the set of explored nodes
    For each u ∈ S, we store a distance d(u)
Initially S = {s} and d(s) = 0
While S ≠ V
    Select a node v ∉ S with at least one edge from S for which
        d′(v) = min_{e=(u,v):u∈S} d(u) + ℓ_e is as small as possible
    Add v to S and define d(v) = d′(v)
EndWhile
```

- Observation: If we add v to S, $d'(w)$ changes only for v's neighbours.
- Store the minima $d'(v)$ for each node $v \in V - S$ in a priority queue.
- Determine the next node v to add to S using ExtractMin.
- After adding v to S, for each neighbour w of v, compute $d(v) + l_{(v,w)}$.
- If $d(v) + l_{(v,w)} < d'(w)$,
    1. Set $d'(w) = d(v) + l_{(v,w)}$.
    2. Update w's key to the new value of $d'(w)$ using ChangeKey.
- How many times are ExtractMin and ChangeKey invoked? $n - 1$ and m times, respectively. Total running time is $O(m \log n)$.

Network Design

- Connect a set of nodes using a set of edges with certain properties.
- Input is usually a graph and the desired network (the output) should use subset of edges in the graph.
- Example: connect all nodes using a cycle of shortest total length.

Network Design

- Connect a set of nodes using a set of edges with certain properties.
- Input is usually a graph and the desired network (the output) should use subset of edges in the graph.
- Example: connect all nodes using a cycle of shortest total length. This problem is the NP-complete traveling salesman problem.

Minimum Spanning Tree (MST)

- Given an undirected graph $G(V, E)$ with a cost $c_e > 0$ associated with each edge $e \in E$.
- Find a subset T of edges such that the graph (V, T) is connected and the cost $\sum_{e \in T} c_e$ is as small as possible.

Minimum Spanning Tree (MST)

- Given an undirected graph $G(V, E)$ with a cost $c_e > 0$ associated with each edge $e \in E$.
- Find a subset T of edges such that the graph (V, T) is connected and the cost $\sum_{e \in T} c_e$ is as small as possible.

MINIMUM SPANNING TREE

INSTANCE: An undirected graph $G(V, E)$ and a function $c : E \rightarrow \mathbb{R}^+$

SOLUTION: A set $T \subseteq E$ of edges such that (V, T) is connected and the $\sum_{e \in T} c_e$ is as small as possible.

Minimum Spanning Tree (MST)

- Given an undirected graph $G(V, E)$ with a cost $c_e > 0$ associated with each edge $e \in E$.
- Find a subset T of edges such that the graph (V, T) is connected and the cost $\sum_{e \in T} c_e$ is as small as possible.

MINIMUM SPANNING TREE

INSTANCE: An undirected graph $G(V, E)$ and a function $c : E \rightarrow \mathbb{R}^+$

SOLUTION: A set $T \subseteq E$ of edges such that (V, T) is connected and the $\sum_{e \in T} c_e$ is as small as possible.

- Claim: If T is a minimum-cost solution to this network design problem then (V, T) is a tree.
- A subset T of E is a *spanning tree* of G if (V, T) is a tree.

Greedy Algorithm for the MST Problem

- Template: process edges in some order. Add an edge to T if tree property is not violated.

Greedy Algorithm for the MST Problem

- Template: process edges in some order. Add an edge to T if tree property is not violated.

  Increasing cost order: Process edges in increasing order of cost. Discard an edge if it creates a cycle.

  Dijkstra-like: Start from a node s and grow T outward from s: add the node that can be attached most cheaply to current tree.

  Decreasing cost order: Delete edges in order of decreasing cost as long as graph remains connected.

Greedy Algorithm for the MST Problem

- Template: process edges in some order. Add an edge to T if tree property is not violated.

  Increasing cost order Process edges in increasing order of cost. Discard an edge if it creates a cycle.

  Dijkstra-like Start from a node s and grow T outward from s: add the node that can be attached most cheaply to current tree.

  Decreasing cost order Delete edges in order of decreasing cost as long as graph remains connected.

- Which of these algorithms works?

Greedy Algorithm for the MST Problem

- Template: process edges in some order. Add an edge to T if tree property is not violated.

  Increasing cost order: Process edges in increasing order of cost. Discard an edge if it creates a cycle. Kruskal's algorithm

  Dijkstra-like: Start from a node s and grow T outward from s: add the node that can be attached most cheaply to current tree. Prim's algorithm

  Decreasing cost order: Delete edges in order of decreasing cost as long as graph remains connected. Reverse-Delete algorithm

- Which of these algorithms works? All of them!

Greedy Algorithm for the MST Problem

- Template: process edges in some order. Add an edge to T if tree property is not violated.

  Increasing cost order: Process edges in increasing order of cost. Discard an edge if it creates a cycle. Kruskal's algorithm

  Dijkstra-like: Start from a node s and grow T outward from s: add the node that can be attached most cheaply to current tree. Prim's algorithm

  Decreasing cost order: Delete edges in order of decreasing cost as long as graph remains connected. Reverse-Delete algorithm
- Which of these algorithms works? All of them!
- Simplifying assumption: all edge costs are distinct.

Example of Prim's and Kruskal's Algorithms

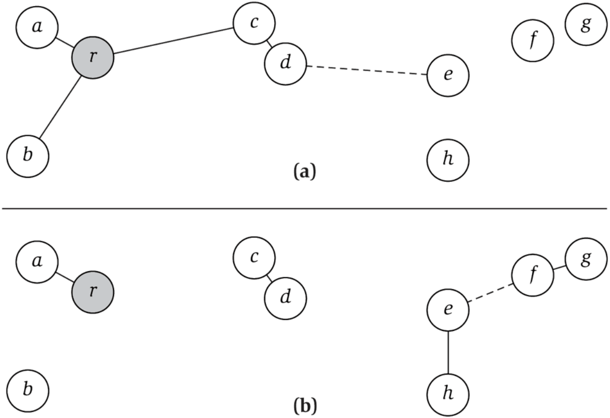

Figure 4.9 Sample run of the Minimum Spanning Tree Algorithms of (a) Prim and (b) Kruskal, on the same input. The first 4 edges added to the spanning tree are indicated by solid lines; the next edge to be added is a dashed line.

Characterising MSTs

- Does the edge of smallest cost belong to an MST?

Characterising MSTs

- Does the edge of smallest cost belong to an MST? Yes.
- Which edges must belong to an MST?

Characterising MSTs

- Does the edge of smallest cost belong to an MST? Yes.
- Which edges must belong to an MST?
  - What happens when we delete an edge from an MST?
  - MST breaks up into sub-trees.
  - Which edge should we add to join them?

Characterising MSTs

- Does the edge of smallest cost belong to an MST? Yes.
- Which edges must belong to an MST?
  - What happens when we delete an edge from an MST?
  - MST breaks up into sub-trees.
  - Which edge should we add to join them?
- Which edges cannot belong to an MST?

Characterising MSTs

- Does the edge of smallest cost belong to an MST? Yes.
- Which edges must belong to an MST?
  - What happens when we delete an edge from an MST?
  - MST breaks up into sub-trees.
  - Which edge should we add to join them?
- Which edges cannot belong to an MST?
  - What happens when we add an edge to an MST?
  - We obtain a cycle.
  - Which edge in the cycle can we be sure does not belong to an MST?

Graph Cuts

- A *cut* in a graph $G(V, E)$ is a set of edges whose removal disconnects the graph (into two or more connected components).
- Every set $S \subset V$ (S cannot be empty or the entire set V) has a corresponding cut: $\text{cut}(S)$ is the set of edges (v, w) such that $v \in S$ and $w \in V - S$.

Graph Cuts

- A *cut* in a graph $G(V, E)$ is a set of edges whose removal disconnects the graph (into two or more connected components).
- Every set $S \subset V$ (S cannot be empty or the entire set V) has a corresponding cut: $\text{cut}(S)$ is the set of edges (v, w) such that $v \in S$ and $w \in V - S$.
- $\text{cut}(S)$ is a cut because deleting the edges in $\text{cut}(S)$ disconnects S from $V - S$.

Cut Property

- When is it safe to include an edge in an MST?

Cut Property

- When is it safe to include an edge in an MST?
- Let $S \subset V$, S is not empty or equal to V.
- Let e be the cheapest edge in cut(S).
- Claim: every MST contains e.

Cut Property

- When is it safe to include an edge in an MST?
- Let $S \subset V$, S is not empty or equal to V.
- Let e be the cheapest edge in cut(S).
- Claim: every MST contains e.
- Proof: exchange argument. If a supposed MST T does not contain e, show that there is a tree with smaller cost than T that contains e.

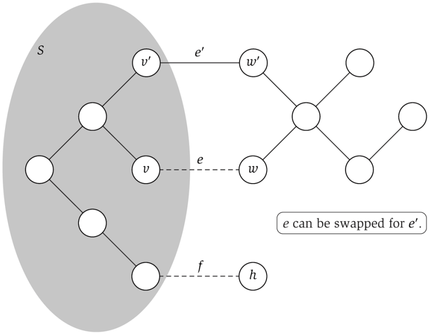

Figure 4.10 Swapping the edge e for the edge e' in the spanning tree T, as described in the proof of (4.17).

Optimality of Kruskal's Algorithm

- Kruskal's algorithm:
  1. Start with an empty set T of edges.
  2. Process edges in E in increasing order of cost.
  3. Add the next edge e to T only if adding e does not create a cycle. Discard e if it creates a cycle.
- Claim: Kruskal's algorithm outputs an MST.

Optimality of Kruskal's Algorithm

- Kruskal's algorithm:
  1. Start with an empty set T of edges.
  2. Process edges in E in increasing order of cost.
  3. Add the next edge e to T only if adding e does not create a cycle. Discard e if it creates a cycle.
- Claim: Kruskal's algorithm outputs an MST.
  1. For every edge e added, demonstrate the existence of S and $V - S$ such that e and S satisfy the cut property.
  2. Prove that the algorithm computes a spanning tree.

Optimality of Prim's Algorithm

- Prim's algorithm: Maintain a tree (S, U)
  1. Start with an arbitrary node $s \in S$ and $U = \emptyset$.
  2. Add the node v to S and the edge e to U that minimise

$$\min_{e=(u,v), u \in S, v \notin S} c_e \equiv \min_{e \in \mathrm{cut}(S)} c_e.$$

  3. Stop when $S = V$.
- Claim: Prim's algorithm outputs an MST.

Optimality of Prim's Algorithm

- Prim's algorithm: Maintain a tree (S, U)
  1. Start with an arbitrary node $s \in S$ and $U = \emptyset$.
  2. Add the node v to S and the edge e to U that minimise

$$\min_{e=(u,v), u \in S, v \notin S} c_e \equiv \min_{e \in \mathrm{cut}(S)} c_e.$$

  3. Stop when $S = V$.
- Claim: Prim's algorithm outputs an MST.
  1. Prove that every edge inserted satisfies the cut property.
  2. Prove that the graph constructed is a spanning tree.

Cycle Property

- When can we be sure that an edge cannot be in *any* MST?

Cycle Property

- When can we be sure that an edge cannot be in *any* MST?
- Let C be any cycle in G and let $e = (v, w)$ be the most expensive edge in C.
- Claim: e does not belong to any MST of G.

Cycle Property

- When can we be sure that an edge cannot be in *any* MST?
- Let C be any cycle in G and let $e = (v, w)$ be the most expensive edge in C.
- Claim: e does not belong to any MST of G.
- Proof: exchange argument. If a supposed MST T contains e, show that there is a tree with smaller cost than T that does not contain e.

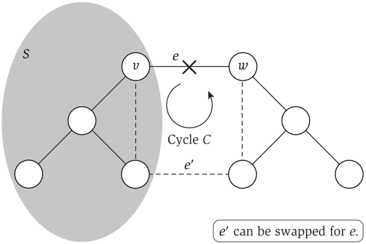

Figure 4.11 Swapping the edge e' for the edge e in the spanning tree T, as described in the proof of (4.20).

Optimality of the Reverse-Delete Algorithm

- Reverse-Delete algorithm: Maintain a set E' of edges.
  - Start with $E' = E$.
  - Process edges in decreasing order of cost.
  - Delete the next edge e from E' only if (V, E') is connected after deletion.
  - Stop after processing all the edges.
- Claim: the Reverse-Delete algorithm outputs an MST.

Optimality of the Reverse-Delete Algorithm

- Reverse-Delete algorithm: Maintain a set E' of edges.
  - Start with $E' = E$.
  - Process edges in decreasing order of cost.
  - Delete the next edge e from E' only if (V, E') is connected after deletion.
  - Stop after processing all the edges.
- Claim: the Reverse-Delete algorithm outputs an MST.
  1. Show that every edge deleted belongs to no MST.
  2. Prove that the graph remaining at the end is a spanning tree.

Comments on MST Algorithms

- To handle multiple edges with the same length, perturb each length by a random infinitesimal amount. Read the textbook.
- *Any* algorithm that constructs a spanning tree by including edges that satisfy the cut property and deleting edges that satisfy the cycle property will yield an MST!

Implementing Prim's Algorithm

- Maintain a tree (S, U).
  1. Start with an arbitrary node $s \in V$ and $U = \emptyset$.
  2. Add the node v to S and the edge e to U that minimise

$$\min_{e \in \mathrm{cut}(S)} c_e.$$

  3. Stop when $S = V$.

Implementing Prim's Algorithm

- Maintain a tree (S, U).
  1. Start with an arbitrary node $s \in V$ and $U = \emptyset$.
  2. Add the node v to S and the edge e to U that minimise
  $$\min_{e \in \mathrm{cut}(S)} c_e.$$
  3. Stop when $S = V$.
- Sorting edges takes $O(m \log n)$ time.
- Implementation is very similar to Dijkstra's algorithm.
- Maintain S and store attachment costs $a(v) = \min_{e \in \mathrm{cut}(S)} c_e$ for every node $v \in V - S$ in a priority queue.
- At each step, extract minimum v from priority queue and update the attachment costs of the neighbours of v.
- Total of $n - 1$ $\textsc{ExtractMin}$ and m $\textsc{ChangeKey}$ operations, yielding a running time of $O(m \log n)$.

Implementing Kruskal's Algorithm

1. Start with an empty set T of edges.
2. Process edges in E in increasing order of cost.
3. Add the next edge e to T only if adding e does not create a cycle.

Implementing Kruskal's Algorithm

1. Start with an empty set T of edges.
2. Process edges in E in increasing order of cost.
3. Add the next edge e to T only if adding e does not create a cycle.

- Sorting edges takes $O(m \log n)$ time.
- Key question in step 3: "Does adding $e = (u, v)$ to T create a cycle?"
- Maintain set of connected components of T in a data structure that supports:
  - $\textsc{Find}(u)$: return the name of the connected component of T containing u.
  - $\textsc{Union}(A, B)$: merge connected components A and B.
- Implementing step 3: Adding $e = (u, v)$ creates a cycle if and only if $\textsc{Find}(u) = \textsc{Find}(v)$.

  3. If $\textsc{Find}(u) \neq \textsc{Find}(v)$,

Implementing Kruskal's Algorithm

1. Start with an empty set T of edges.
2. Process edges in E in increasing order of cost.
3. Add the next edge e to T only if adding e does not create a cycle.

- Sorting edges takes $O(m \log n)$ time.
- Key question in step 3: "Does adding $e = (u, v)$ to T create a cycle?"
- Maintain set of connected components of T in a data structure that supports:
  - $\textsc{Find}(u)$: return the name of the connected component of T containing u.
  - $\textsc{Union}(A, B)$: merge connected components A and B.
- Implementing step 3: Adding $e = (u, v)$ creates a cycle if and only if $\textsc{Find}(u) = \textsc{Find}(v)$.
  3. If $\textsc{Find}(u) \neq \textsc{Find}(v)$, execute $\textsc{Union}(\textsc{Find}(u), \textsc{Find}(v))$ and add e to T.

Analysing Kruskal's Algorithm

- How many FIND invocations does Kruskal's algorithm need?

Analysing Kruskal's Algorithm

- How many FIND invocations does Kruskal's algorithm need? $2m$.
- How many UNION invocations does Kruskal's algorithm need?

Analysing Kruskal's Algorithm

- How many FIND invocations does Kruskal's algorithm need? $2m$.
- How many UNION invocations does Kruskal's algorithm need? $n - 1$.

Analysing Kruskal's Algorithm

- How many FIND invocations does Kruskal's algorithm need? $2m$.
- How many UNION invocations does Kruskal's algorithm need? $n - 1$.
- We will show two implementations of UNION-FIND:
  - Each FIND takes $O(1)$ time, k invocations of UNION take $O(k \log k)$ time in total.
  - Each FIND takes $O(\log n)$ time and each invocation of UNION takes $O(1)$ time.

Analysing Kruskal's Algorithm

- How many FIND invocations does Kruskal's algorithm need? $2m$.
- How many UNION invocations does Kruskal's algorithm need? $n - 1$.
- We will show two implementations of UNION-FIND:
  - Each FIND takes $O(1)$ time, k invocations of UNION take $O(k \log k)$ time in total.
  - Each FIND takes $O(\log n)$ time and each invocation of UNION takes $O(1)$ time.
- Total running time of Kruskal's algorithm is $O(m \log n)$.

Union-Find Data Structure

- Abstraction of the data structure needed by Kruskal's algorithm.
- Maintain disjoint subsets of elements from a universe U of n elements.
  - Think of each subset being a connected component of T.
- Each subset has a name. A subset's name will be the identity of some element in it.
- Support three operations:
  1. $\textsc{MakeUnionFind}(U)$: initialise the data structure with elements in U.
  2. $\textsc{Find}(u)$: return the identity of the subset that contains u.
  3. $\textsc{Union}(A, B)$: merge the sets named A and B into one set.

Union-Find Data Structure: Implementation 1

- Running example: three sets $\{s, u, w\}$, $\{t, v, z\}$, $\{i, j, x, y\}$ with names u, v, and j, respectively.

Union-Find Data Structure: Implementation 1

- Running example: three sets $\{s, u, w\}$, $\{t, v, z\}$, $\{i, j, x, y\}$ with names u, v, and j, respectively.
- Store all the elements of U in an array $\textsc{Component}$.
  - Assume identities of elements are integers from 1 to n.
  - $\textsc{Component}[s]$ is the name of the set containing s.
- Implementing the operations:

Union-Find Data Structure: Implementation 1

- Running example: three sets $\{s, u, w\}$, $\{t, v, z\}$, $\{i, j, x, y\}$ with names u, v, and j, respectively.
- Store all the elements of U in an array $\textsc{Component}$.
  - Assume identities of elements are integers from 1 to n.
  - $\textsc{Component}[s]$ is the name of the set containing s.
- Implementing the operations:
  1. $\textsc{MakeUnionFind}(U)$: For each $s \in U$, set $\textsc{Component}[s] = s$ in $O(n)$ time.
  2. $\textsc{Find}(s)$: return $\textsc{Component}[s]$ in $O(1)$ time.
  3. $\textsc{Union}(A, B)$: merge B into A by scanning $\textsc{Component}$ and updating each index whose value is B to the value A. Takes $O(n)$ time.

Union-Find Data Structure: Implementation 1

- Running example: three sets $\{s, u, w\}$, $\{t, v, z\}$, $\{i, j, x, y\}$ with names u, v, and j, respectively.
- Store all the elements of U in an array $\textsc{Component}$.
  - Assume identities of elements are integers from 1 to n.
  - $\textsc{Component}[s]$ is the name of the set containing s.
- Implementing the operations:
  1. $\textsc{MakeUnionFind}(U)$: For each $s \in U$, set $\textsc{Component}[s] = s$ in $O(n)$ time.
  2. $\textsc{Find}(s)$: return $\textsc{Component}[s]$ in $O(1)$ time.
  3. $\textsc{Union}(A, B)$: merge B into A by scanning $\textsc{Component}$ and updating each index whose value is B to the value A. Takes $O(n)$ time.
- $\textsc{Union}$ is very slow because

Union-Find Data Structure: Implementation 1

- Running example: three sets $\{s, u, w\}$, $\{t, v, z\}$, $\{i, j, x, y\}$ with names u, v, and j, respectively.
- Store all the elements of U in an array COMPONENT.
  - Assume identities of elements are integers from 1 to n.
  - $\text{COMPONENT}[s]$ is the name of the set containing s.
- Implementing the operations:
  1. $\text{MAKEUNIONFIND}(U)$: For each $s \in U$, set $\text{COMPONENT}[s] = s$ in $O(n)$ time.
  2. $\text{FIND}(s)$: return $\text{COMPONENT}[s]$ in $O(1)$ time.
  3. $\text{UNION}(A, B)$: merge B into A by scanning COMPONENT and updating each index whose value is B to the value A. Takes $O(n)$ time.
- UNION is very slow because we cannot efficiently find the elements that belong to a given set.

Union-Find Data Structure: Implementation 2

- Optimisation 1: Use an array $\textsc{Elements}$ in addition to $\textsc{Component}$.
  - Indices of $\textsc{Elements}$ range from 1 to n.
  - $\textsc{Elements}[s]$ stores the elements in the subset named s in a list.
- Execute $\textsc{Union}(A, B)$ by merging B into A in two steps:
  1. For every element $u \in B$, set $\textsc{Component}[u] = A$ in $O(|B|)$ time.
  2. Append $\textsc{Elements}[B]$ to $\textsc{Elements}[A]$ in $O(1)$ time.
- $\textsc{Union}$ takes $\Omega(n)$ in the worst-case.

Union-Find Data Structure: Implementation 2

- Optimisation 1: Use an array Elements in addition to Component.
  - Indices of Elements range from 1 to n.
  - $\text{Elements}[s]$ stores the elements in the subset named s in a list.
- Execute $\text{Union}(A, B)$ by merging B into A in two steps:
  1. For every element $u \in B$, set $\text{Component}[u] = A$ in $O(|B|)$ time.
  2. Append $\text{Elements}[B]$ to $\text{Elements}[A]$ in $O(1)$ time.
- Union takes $\Omega(n)$ in the worst-case.
- Optimisation 2: Store size of each set in an array Size. If $\text{Size}[B] \leq \text{Size}[A]$, merge B into A. Otherwise merge A into B. Update Size.

Union-Find Data Structure: Analysis of Implementation 2

- MakeUnionFind(S) and Find(u) are as before.

Union-Find Data Structure: Analysis of Implementation 2

- MakeUnionFind(S) and Find(u) are as before.
- Union(A, B): Running time is proportional to the size of the smaller set, which may be $\Omega(n)$.

Union-Find Data Structure: Analysis of Implementation 2

- MAKEUNIONFIND(S) and FIND(u) are as before.
- UNION(A, B): Running time is proportional to the size of the smaller set, which may be $\Omega(n)$.
- Any sequence of k UNION operations takes $O(k \log k)$ time.

Union-Find Data Structure: Analysis of Implementation 2

- MakeUnionFind(S) and Find(u) are as before.
- Union(A, B): Running time is proportional to the size of the smaller set, which may be $\Omega(n)$.
- Any sequence of k Union operations takes $O(k \log k)$ time.
  - k Union operations touch at most $2k$ elements.

Union-Find Data Structure: Analysis of Implementation 2

- MakeUnionFind(S) and Find(u) are as before.
- Union(A, B): Running time is proportional to the size of the smaller set, which may be $\Omega(n)$.
- Any sequence of k Union operations takes $O(k \log k)$ time.
  - k Union operations touch at most $2k$ elements.
  - Intuition: running time of Union is dominated by updates to Component. Charge each update to the element being updated and bound number of charges per element.

Union-Find Data Structure: Analysis of Implementation 2

- MAKEUNIONFIND(S) and FIND(u) are as before.
- UNION(A, B): Running time is proportional to the size of the smaller set, which may be $\Omega(n)$.
- Any sequence of k UNION operations takes $O(k \log k)$ time.
  - k UNION operations touch at most $2k$ elements.
  - Intuition: running time of UNION is dominated by updates to COMPONENT. Charge each update to the element being updated and bound number of charges per element.
  - Consider any element s. Every time s's set identity is updated, the size of the set containing s at least doubles $\Rightarrow$ s's set can change at most $\log(2k)$ times $\Rightarrow$ the total work done in k UNION operations is $O(k \log k)$.

Union-Find Data Structure: Analysis of Implementation 2

- MakeUnionFind(S) and Find(u) are as before.
- Union(A, B): Running time is proportional to the size of the smaller set, which may be $\Omega(n)$.
- Any sequence of k Union operations takes $O(k \log k)$ time.
  - k Union operations touch at most $2k$ elements.
  - Intuition: running time of Union is dominated by updates to Component. Charge each update to the element being updated and bound number of charges per element.
  - Consider any element s. Every time s's set identity is updated, the size of the set containing s at least doubles $\Rightarrow$ s's set can change at most $\log(2k)$ times $\Rightarrow$ the total work done in k Union operations is $O(k \log k)$.
- Find is fast in the worst case, Union is fast in an amortised sense. Can we make both operations worst-case efficient?

Union-Find Data Structure: Implementation 3

- Goal: Implement FIND in $O(\log n)$ and UNION in $O(1)$ worst-case time.

Union-Find Data Structure: Implementation 3

- Goal: Implement FIND in $O(\log n)$ and UNION in $O(1)$ worst-case time.
- Represent each subset in a tree using pointers:
  - Each tree node contains an element and a pointer to a parent.
  - The identity of the set is the identity of the element at the root.

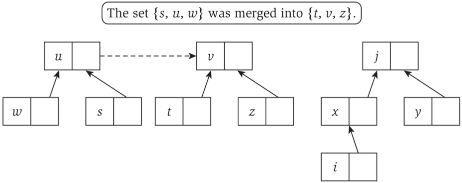

Figure 4.12 A `Union-Find` data structure using pointers. The data structure has only two sets at the moment, named after nodes v and j. The dashed arrow from u to v is the result of the last `Union` operation. To answer a `Find` query, we follow the arrows until we get to a node that has no outgoing arrow. For example, answering the query `Find`(i) would involve following the arrows i to x, and then x to j.

Union-Find Data Structure: Implementation 3

- Goal: Implement FIND in $O(\log n)$ and UNION in $O(1)$ worst-case time.
- Represent each subset in a tree using pointers:
  - Each tree node contains an element and a pointer to a parent.
  - The identity of the set is the identity of the element at the root.
- Implementing FIND(u): follow pointers from u to the root of u's tree.

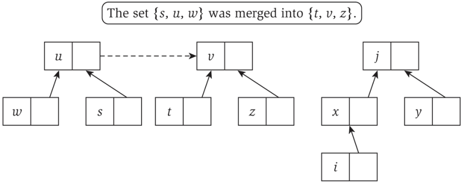

Figure 4.12 A `Union-Find` data structure using pointers. The data structure has only two sets at the moment, named after nodes v and j. The dashed arrow from u to v is the result of the last `Union` operation. To answer a `Find` query, we follow the arrows until we get to a node that has no outgoing arrow. For example, answering the query `Find`(i) would involve following the arrows i to x, and then x to j.

Union-Find Data Structure: Implementation 3

- Goal: Implement FIND in $O(\log n)$ and UNION in $O(1)$ worst-case time.
- Represent each subset in a tree using pointers:
  - Each tree node contains an element and a pointer to a parent.
  - The identity of the set is the identity of the element at the root.
- Implementing FIND(u): follow pointers from u to the root of u's tree.
- Implementing UNION(A, B): make smaller tree's root a child of the larger tree's root. Takes $O(1)$ time.

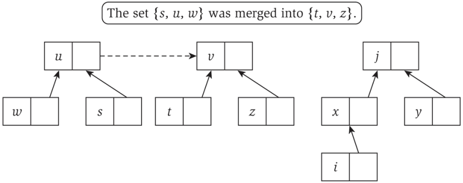

Figure 4.12 A Union-Find data structure using pointers. The data structure has only two sets at the moment, named after nodes v and j. The dashed arrow from u to v is the result of the last Union operation. To answer a Find query, we follow the arrows until we get to a node that has no outgoing arrow. For example, answering the query Find(i) would involve following the arrows i to x, and then x to j.

Union-Find Data Structure: Find in Implementation 3

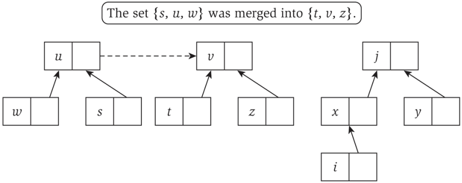

Figure 4.12 A `Union-Find` data structure using pointers. The data structure has only two sets at the moment, named after nodes v and j. The dashed arrow from u to v is the result of the last `Union` operation. To answer a `Find` query, we follow the arrows until we get to a node that has no outgoing arrow. For example, answering the query `Find`(i) would involve following the arrows i to x, and then x to j.

- Why does $\textsc{Find}(u)$ take $O(\log n)$ time?

Union-Find Data Structure: Find in Implementation 3

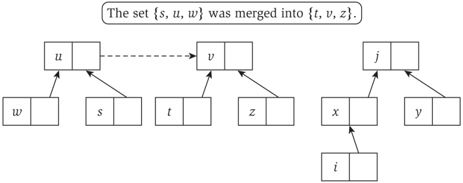

Figure 4.12 A Union-Find data structure using pointers. The data structure has only two sets at the moment, named after nodes v and j. The dashed arrow from u to v is the result of the last Union operation. To answer a Find query, we follow the arrows until we get to a node that has no outgoing arrow. For example, answering the query Find(i) would involve following the arrows i to x, and then x to j.

- Why does $\textsc{Find}(u)$ take $O(\log n)$ time?
- Number of pointers followed equals the number of times the identity of the set containing u changed.
- Every time u's set's identity changes, the set at least doubles in size $\Rightarrow$ there are $O(\log n)$ pointers followed.

Union-Find Data Structure: Improving Implementation 3

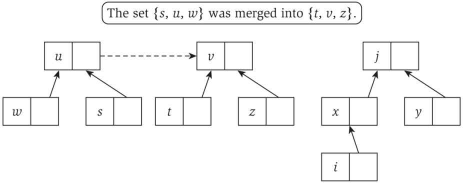

Figure 4.12 A `Union-Find` data structure using pointers. The data structure has only two sets at the moment, named after nodes v and j. The dashed arrow from u to v is the result of the last `Union` operation. To answer a `Find` query, we follow the arrows until we get to a node that has no outgoing arrow. For example, answering the query `Find`(i) would involve following the arrows i to x, and then x to j.

- Every time we invoke $\textsc{Find}(u)$, we follow the same set of pointers.

Union-Find Data Structure: Improving Implementation 3

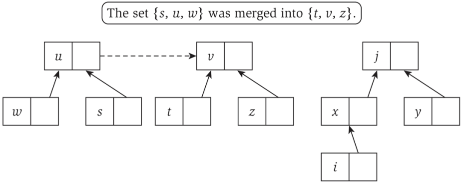

Figure 4.12 A `Union-Find` data structure using pointers. The data structure has only two sets at the moment, named after nodes v and j. The dashed arrow from u to v is the result of the last `Union` operation. To answer a `Find` query, we follow the arrows until we get to a node that has no outgoing arrow. For example, answering the query `Find`(i) would involve following the arrows i to x, and then x to j.

- Every time we invoke $\text{Find}(u)$, we follow the same set of pointers.
- Path compression: make all nodes visited by $\text{Find}(u)$ children of the root.

Union-Find Data Structure: Improving Implementation 3

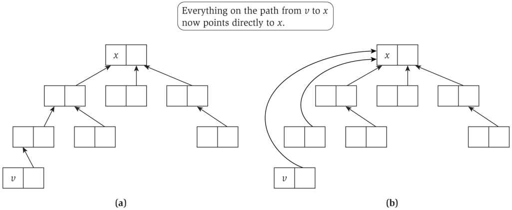

Figure 4.13 (a) An instance of a `Union-Find` data structure; and (b) the result of the operation `Find`(v) on this structure, using path compression.

- Every time we invoke $\text{FIND}(u)$, we follow the same set of pointers.
- Path compression: make all nodes visited by $\text{FIND}(u)$ children of the root.

Union-Find Data Structure: Improving Implementation 3

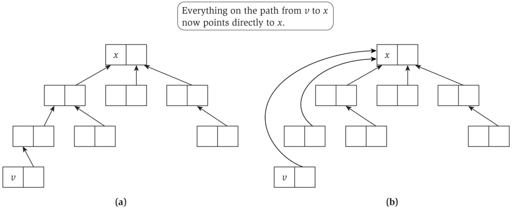

Figure 4.13 (a) An instance of a `Union-Find` data structure; and (b) the result of the operation `Find`(v) on this structure, using path compression.

- Every time we invoke $\textsc{Find}(u)$, we follow the same set of pointers.
- Path compression: make all nodes visited by $\textsc{Find}(u)$ children of the root.
- Can prove that total time taken by n $\textsc{Find}$ operations is $O(n\alpha(n))$, where $\alpha(n)$ is the inverse of the Ackermann function, and grows e-x-t-r-e-m-e-l-y s-l-o-w-l-y with n.

Comments on Union-Find and MST

- The UNION-FIND data structure is useful to maintain the connected components of a graph as edges are added to the graph.
- The data structure does not support edge deletion efficiently.
- Current best algorithm for MST runs in $O(m\alpha(m, n))$ time (Chazelle 2000) and $O(m)$ randomised time (Karger, Klein, and Tarjan, 1995).
- Holy grail: $O(m)$ deterministic algorithm for MST.