Greedy Graph Algorithms

September 25, 2014
Graphs

- Model pairwise relationships (edges) between objects (nodes).
- Undirected graph $G = (V, E)$: set $V$ of nodes and set $E$ of edges, where $E \subseteq V \times V$. Elements of $E$ are unordered pairs.
- Directed graph $G = (V, E)$: set $V$ of nodes and set $E$ of edges, where $E \subseteq V \times V$. Elements of $E$ are ordered pairs.
Applications of Graphs

- Useful in a large number of applications:
Applications of Graphs

- Useful in a large number of applications: computer networks, the World Wide Web, ecology (food webs), social networks, software systems, job scheduling, VLSI circuits, cellular networks, ...  
- Problems involving graphs have a rich history dating back to Euler.
Applications of Graphs

- Useful in a large number of applications: computer networks, the World Wide Web, ecology (food webs), social networks, software systems, job scheduling, VLSI circuits, cellular networks, . . .
- Problems involving graphs have a rich history dating back to Euler.
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?

---

**Simplifying assumption:** all edge costs are distinct.
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 two or more 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?
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 two or more sub-trees.
  - Which edge should we add to join them?
- Which edges cannot belong to an MST?

September 25, 2014 Greedy Graph Algorithms
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 two or more 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 $S \subseteq V$ (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 $V - S$. 

September 25, 2014 Greedy Graph Algorithms
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 $e$. 

September 25, 2014 Greedy Graph Algorithms
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 $e$. 

September 25, 2014  Greedy Graph Algorithms
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 $\text{cut}(S)$.
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 $e$.

\[ e \text{ can be swapped for } e'. \]

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:
  - Start with an empty set $T$ of edges.
  - Process edges in $E$ in increasing order of cost.
  - 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:
  - Start with an empty set $T$ of edges.
  - Process edges in $E$ in increasing order of cost.
  - 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 Kruskal’s Algorithm

- Kruskal’s algorithm:
  - Start with an empty set $T$ of edges.
  - Process edges in $E$ in increasing order of cost.
  - 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.
    - If $e = (u, v)$, let $S$ be the set of nodes connected to $u$ in the current graph $T$.
  2. Prove that the algorithm computes a spanning tree.
Optimality of Kruskal’s Algorithm

- Kruskal’s algorithm:
  - Start with an empty set $T$ of edges.
  - Process edges in $E$ in increasing order of cost.
  - 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.
     - If $e = (u, v)$, let $S$ be the set of nodes connected to $u$ in the current graph $T$.
     - Why is $e$ the cheapest edge in $\text{cut}(S)$?
  2. Prove that the algorithm computes a spanning tree.
Optimality of Kruskal’s Algorithm

- **Kruskal’s algorithm:**
  - Start with an empty set $T$ of edges.
  - Process edges in $E$ in increasing order of cost.
  - 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.
     - If $e = (u, v)$, let $S$ be the set of nodes connected to $u$ in the current graph $T$.
     - Why is $e$ the cheapest edge in cut($S$)?
  2. Prove that the algorithm computes a spanning tree.
     - $(V, T)$ contains no cycles by construction.
Optimality of Kruskal’s Algorithm

▷ Kruskal’s algorithm:
  ▷ Start with an empty set $T$ of edges.
  ▷ Process edges in $E$ in increasing order of cost.
  ▷ 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.
     ▷ If $e = (u, v)$, let $S$ be the set of nodes connected to $u$ in the current graph $T$.
     ▷ Why is $e$ the cheapest edge in cut($S$)?
  2. Prove that the algorithm computes a spanning tree.
     ▷ $(V, T)$ contains no cycles by construction.
     ▷ If $(V, T)$ is not connected, then exists a subset $S$ of nodes not connected to $V - S$. What is the contradiction?
Optimality of Prim’s Algorithm

- Prim’s algorithm: Maintain a tree \((S, U)\)
  - Start with an arbitrary node \(s \in S\) and \(U = \emptyset\).
  - Add the node \(v\) to \(S\) and the edge \(e\) to \(U\) that minimise
    \[
    \min_{e = (u, v), u \in S, v \notin S} c_e \equiv \min_{e \in \text{cut}(S)} c_e.
    \]
  - Stop when \(S = V\).
- Claim: Prim’s algorithm outputs an MST.
Optimality of Prim’s Algorithm

- Prim’s algorithm: Maintain a tree \((S, U)\)
  - Start with an arbitrary node \(s \in S\) and \(U = \emptyset\).
  - Add the node \(v\) to \(S\) and the edge \(e\) to \(U\) that minimise
    \[
    \min_{e = (u, v), u \in S, v \notin S} c_e \equiv \min_{e \in \text{cut}(S)} c_e.
    \]
- 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.
Optimality of Prim’s Algorithm

- Prim’s algorithm: Maintain a tree \((S, U)\)
  - Start with an arbitrary node \(s \in S\) and \(U = \emptyset\).
  - Add the node \(v\) to \(S\) and the edge \(e\) to \(U\) that minimise
    \[
    \min_{e=(u,v), u \in S, v \notin S} c_e \equiv \min_{e \in \text{cut}(S)} c_e.
    \]
- Stop when \(S = V\).

- Claim: Prim’s algorithm outputs an MST.
  1. Prove that every edge inserted satisfies the cut property.
     - In each iteration, \(S\) is the set added in the algorithm and \(e\) is the cheapest edge in \(\text{cut}(S)\) by construction.
  2. Prove that the graph constructed is a spanning tree.
Optimality of Prim’s Algorithm

- Prim’s algorithm: Maintain a tree \((S, U)\)
  - Start with an arbitrary node \(s \in S\) and \(U = \emptyset\).
  - Add the node \(v\) to \(S\) and the edge \(e\) to \(U\) that minimise
    \[
    \min_{e=(u, v), u \in S, v \notin S} c_e \equiv \min_{e \in \text{cut}(S)} c_e.
    \]
- Stop when \(S = V\).

- Claim: Prim’s algorithm outputs an MST.
  1. Prove that every edge inserted satisfies the cut property.
     - In each iteration, \(S\) is the set added in the algorithm and \(e\) is the cheapest edge in \(\text{cut}(S)\) by construction.
  2. Prove that the graph constructed is a spanning tree.
     - Why are there no cycles in \((V, T)\)?
Optimality of Prim’s Algorithm

- Prim’s algorithm: Maintain a tree \((S, U)\)
  - Start with an arbitrary node \(s \in S\) and \(U = \emptyset\).
  - Add the node \(v\) to \(S\) and the edge \(e\) to \(U\) that minimise
    \[
    \min_{e = (u, v), u \in S, v \notin S} c_e \equiv \min_{e \in \text{cut}(S)} c_e.
    \]
  - Stop when \(S = V\).

- Claim: Prim’s algorithm outputs an MST.
  1. Prove that every edge inserted satisfies the cut property.
     - In each iteration, \(S\) is the set added in the algorithm and \(e\) is the cheapest edge in \(\text{cut}(S)\) by construction.
  2. Prove that the graph constructed is a spanning tree.
     - Why are there no cycles in \((V, T)\)?
     - Why is \((V, T)\) connected?
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 $G$. 

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 $e$. 

September 25, 2014 Greedy Graph Algorithms
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.

1. Show that every edge deleted belongs to no MST.
   - A deleted edge must belong to some cycle $C$.
   - Since the edge is the first encountered by the algorithm, it is the most expensive edge in $C$.

2. Prove that the graph remaining at the end is a spanning tree.
   - $(V, E')$ is connected at the end, by construction.
   - If $(V, E')$ contains a cycle, consider the costliest edge in that cycle. The algorithm would have deleted that edge.
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.
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.
     - A deleted edge must belong to some cycle $C$.
     - Since the edge is the first encountered by the algorithm, it is the most expensive edge in $C$.
  2. Prove that the graph remaining at the end is a spanning tree.
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.
     - A deleted edge must belong to some cycle $C$.
     - Since the edge is the first encountered by the algorithm, it is the most expensive edge in $C$.
  2. Prove that the graph remaining at the end is a spanning tree.
     - $(V, E')$ is connected at the end, by construction.
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.
     - A deleted edge must belong to some cycle $C$.
     - Since the edge is the first encountered by the algorithm, it is the most expensive edge in $C$.
  2. Prove that the graph remaining at the end is a spanning tree.
     - $(V, E')$ is connected at the end, by construction.
     - If $(V, E')$ contains a cycle, consider the costliest edge in that cycle. The algorithm would have deleted that edge.
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 \((S, U)\).
  - Start with an arbitrary node \(s \in V\) and \(U = \emptyset\).
  - Add the node \(v\) to \(S\) and the edge \(e\) to \(U\) that minimise
    \[
    \min_{e \in \text{cut}(S)} c_e.
    \]
  - Stop when \(S = V\).
Implementing Prim’s Algorithm

- Maintain a tree $(S, U)$.
  - Start with an arbitrary node $s \in V$ and $U = \emptyset$.
  - Add the node $v$ to $S$ and the edge $e$ to $U$ that minimise
    \[
    \min_{e \in \text{cut}(S)} c_e.
    \]
  - 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 $a(v) = \min_{e \in \text{cut}(S)} c_e$ 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 $n - 1$ ExtractMin and $m$ ChangeKey operations, yielding a running time of $O(m \log n)$. 

Implementing Kruskal’s Algorithm

- Start with an empty set \( T \) of edges.
- Process edges in \( E \) in increasing order of cost.
- Add the next edge \( e \) to \( T \) only if adding \( e \) does not create a cycle.
Implementing Kruskal’s Algorithm

- Start with an empty set $T$ of edges.
- Process edges in $E$ in increasing order of cost.
- 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: “Does adding $e = (u, v)$ to $T$ create a cycle?”
  - Maintain set of connected components of $T$.
  - $\text{FIND}(u)$: return the name of the connected component of $T$ that $u$ belongs to.
  - $\text{UNION}(A, B)$: merge connected components $A$ and $B$. 
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 \texttt{FIND} invocations does Kruskal’s algorithm need? 2m.
- How many \texttt{UNION} invocations does Kruskal’s algorithm need? n − 1.
- Textbook describes two implementations of \texttt{UNION-FIND}: (see appendix to this set of slides)
  - Each \texttt{FIND} takes $O(1)$ time, $k$ invocations of \texttt{UNION} take $O(k \log k)$ time in total.
  - Each \texttt{FIND} takes $O(\log n)$ time and each invocation of \texttt{UNION} takes $O(1)$ time.

Total running time of Kruskal's algorithm is $O(m \log n)$. 

September 25, 2014 Greedy Graph Algorithms
Analysing Kruskal’s Algorithm

- How many $\text{FIND}$ invocations does Kruskal’s algorithm need? $2m$.
- How many $\text{UNION}$ invocations does Kruskal’s algorithm need? $n - 1$.
- Textbook describes two implementations of $\text{UNION-FIND}$: (see appendix to this set of slides)
  - Each $\text{FIND}$ takes $O(1)$ time, $k$ invocations of $\text{UNION}$ take $O(k \log k)$ time in total.
  - Each $\text{FIND}$ takes $O(\log n)$ time and each invocation of $\text{UNION}$ takes $O(1)$ time.
- Total running time of Kruskal’s algorithm is $O(m \log n)$. 
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.
Appendix: Union-Find

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.
- Each subset has a name. We will set a set’s name to be the identity of some element in it.
- Support three operations:
  1. \texttt{MAKEUNIONFIND}(\( U \)): initialise the data structure with elements in \( U \).
  2. \texttt{FIND}(\( u \)): return the identity of the subset that contains \( u \).
  3. \texttt{UNION}(\( A, B \)): merge the sets named \( A \) and \( B \) into one set.
Union-Find Data Structure: Implementation 1

- Store all the elements of $U$ in an array $\text{COMPONENT}$.
  - Assume identities of elements are integers from 1 to $n$.
  - $\text{COMPONENT}[s]$ is the name of the set containing $s$.
- Implementing the operations:
Appendix: Union-Find

Union-Find Data Structure: Implementation 1

- Store all the elements of $U$ in an array $\text{COMPONENT}$.
  - 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 $\text{COMPONENT}$ and updating each index whose value is $B$ to the value $A$. Takes $O(n)$ time.
Appendix: Union-Find

Union-Find Data Structure: Implementation 1

- Store all the elements of $U$ in an array $\text{Component}$.
  - 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 $\text{Component}$ and updating each index whose value is $B$ to the value $A$. Takes $O(n)$ time.

$\text{Union}$ is very slow because
Union-Find Data Structure: Implementation 1

- Store all the elements of $U$ in an array $\text{COMPONENT}$.
  - 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 $\text{COMPONENT}$ and updating each index whose value is $B$ to the value $A$. Takes $O(n)$ time.

- $\text{UNION}$ is very slow because we cannot efficiently find the elements that belong to a set.
Appendix: Union-Find

Union-Find Data Structure: Implementation 2

- Optimisation 1: Use an array \texttt{ELEMENTS}
  - Indices of \texttt{ELEMENTS} range from 1 to \( n \).
  - \texttt{ELEMENTS}[s] stores the elements in the subset named \( s \) in a list.

- Execute \texttt{UNION}(A, B) by merging \( B \) into \( A \) in two steps:
  1. Updating \texttt{COMPONENT} for elements of \( B \) in \( O(|B|) \) time.
  2. Append \texttt{ELEMENTS}[B] to \texttt{ELEMENTS}[A] in \( O(1) \) time.

- \texttt{UNION} takes \( \Omega(n) \) in the worst-case.
Appendix: Union-Find

Union-Find Data Structure: Implementation 2

- Optimisation 1: Use an array \texttt{ELEMENTS}
  - Indices of \texttt{ELEMENTS} range from 1 to \(n\).
  - \texttt{ELEMENTS}[s] stores the elements in the subset named \(s\) in a list.

- Execute \texttt{UNION}(A, B) by merging \(B\) into \(A\) in two steps:
  1. Updating \texttt{COMPONENT} for elements of \(B\) in \(O(|B|)\) time.
  2. Append \texttt{ELEMENTS}[B] to \texttt{ELEMENTS}[A] in \(O(1)\) time.

- \texttt{UNION} takes \(\Omega(n)\) in the worst-case.

- Optimisation 2: Store size of each set in an array (say, \texttt{SIZE}). If \(\texttt{SIZE}[B] \leq \texttt{SIZE}[A]\), merge \(B\) into \(A\). Otherwise merge \(A\) into \(B\). Update \texttt{SIZE}.
Appendix: Union-Find

Union-Find Data Structure: Analysis of Implementation

- `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 $\log(2^k)$ 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: Analysis of Implementation

- **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

- **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.
Appendix: Union-Find

Union-Find Data Structure: Analysis of Implementation

- `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.
Appendix: Union-Find

Union-Find Data Structure: Analysis of Implementation

- **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.

- **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: Analysis of Implementation

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

Union-Find Data Structure: Analysis of Implementation

- **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 \texttt{FIND} in $O(\log n)$ and \texttt{UNION} in $O(1)$ worst-case time.
Union-Find Data Structure: Implementation 3

- Goal: Implement \texttt{FIND} in $O(\log n)$ and \texttt{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.

![Diagram of Union-Find data structure with nodes and arrows indicating set membership and hierarchy.](image)

\textbf{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 \texttt{Find}(i) would involve following the arrows $i$ to $x$, and then $x$ to $j$. 
Goal: Implement \texttt{Find} in $O(\log n)$ and \texttt{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 \texttt{Find}(u): follow pointers from $u$ to the root of $u$’s tree.

\begin{figure}
\centering
\includegraphics[width=\textwidth]{union-find-diagram.png}
\caption{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 $j$.}
\end{figure}
Appendix: Union-Find

Union-Find Data Structure: Implementation 3

- **Goal**: Implement \texttt{FIND} in $O(\log n)$ and \texttt{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 \texttt{FIND}(u)**: follow pointers from $u$ to the root of $u$’s tree.
- **Implementing \texttt{UNION}(A, B)**: make smaller tree’s root a child of the larger tree’s root. Takes $O(1)$ time.

![Diagram](image)

**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 $j$. 

September 25, 2014 Greedy Graph Algorithms
Why does $\text{FIND}(u)$ take $O(\log n)$ 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 $\text{Find}(i)$ would involve following the arrows $i$ to $x$, and then $x$ to $j$. 

The set $\{s, u, w\}$ was merged into $\{t, v, z\}$. 

```plaintext
1. $u$ is merged into $v$.
2. $w$ is merged into $s$.
3. $t$ is merged into $s$.
4. $j$ is merged into $t$.
5. $x$ is merged into $w$.
6. $y$ is merged into $x$.
7. $i$ is merged into $x$.
```
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.
Every time we invoke $\text{FIND}(u)$, we follow the same set of pointers.

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 $\text{Find}(i)$ would involve following the arrows $i$ to $x$, and then $x$ to $j$. 

The set $\{s, u, w\}$ was merged into $\{t, v, z\}$. 
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.
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.
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.

Can prove that total time taken by $n$ $\text{FIND}$ operations is $O(n\alpha(n))$, where $\alpha(n)$ is the inverse of the Ackermann function, and grows extremely slowly with $n$. 