Graphs and MST

Pierre Aboulker - pierreaboulker@gmail.com
1 - Graph Basics
Definitions

A graph $G = (V, E)$:
- $V$ is the set of vertices
- $E \subseteq V \times V$ is the set of edges.

All along the course, particularly for complexity analysis,
- $n$ is the number of vertices,
- $m$ is the number of edges.

A bridgeless cubic graph with chromatic index 4 and girth 5
Why do we study graphs

- Thousands of practical applications.
- Challenging branch of computer science and discrete math.
- Hundreds of graph algorithms known.
- Rich combinatorial properties.

figure by Kevin Wayne
Born of graph theory: Euler and the Koenigsberg bridges
Graphs: a modelization tool

(a) Königsberg in 1736

(b) Euler's graphical representation
Praeter illam Geometriae parrem, quae circa quantitates versatur, et omni tempore summo studio, est exculta, alterius partis etiamque admodum ignotae primum mentionem fecit Leibnitzius, quam Geometriam situs vocavit. Istam pars ab ipso singularem determinandae, singulique proprietatis distincte occupata esse statuit; in quo negotio necesse ad quantitates respicienda, neque calculo quantitatum versusum est. Cuiusmodi autem problemata ad hanc situs Geometriam pertinent, et qualis methodo in iis resolueendis vti operetur, non fas est definirem. Quamobrem, cum super problematis cujusdam mention esset facta, quod quidem ad geometriam pertinentem videatur, at ita est comparatum, ut neque determinationem quantitatum requireret, neque solutionem calulii quantitatum ope admitteret, id ad geometriam situs referre hanc dubitavi: praetertim quod in eius solutione solus situs in considerationem veniat, calculus vero nullius prorsus sit vasis. Methodum ergo meam quam ad huius generis problematica
## Graph applications

<table>
<thead>
<tr>
<th>graph</th>
<th>vertex</th>
<th>edge</th>
</tr>
</thead>
<tbody>
<tr>
<td>communication</td>
<td>telephone, computer</td>
<td>fiber optic cable</td>
</tr>
<tr>
<td>circuit</td>
<td>gate, register, processor</td>
<td>wire</td>
</tr>
<tr>
<td>mechanical</td>
<td>joint</td>
<td>rod, beam, spring</td>
</tr>
<tr>
<td>financial</td>
<td>stock, currency</td>
<td>transactions</td>
</tr>
<tr>
<td>transportation</td>
<td>street intersection, airport</td>
<td>highway, airway route</td>
</tr>
<tr>
<td>internet</td>
<td>class C network</td>
<td>connection</td>
</tr>
<tr>
<td>game</td>
<td>board position</td>
<td>legal move</td>
</tr>
<tr>
<td>social relationship</td>
<td>person, actor</td>
<td>friendship, movie cast</td>
</tr>
<tr>
<td>neural network</td>
<td>neuron</td>
<td>synapse</td>
</tr>
<tr>
<td>protein network</td>
<td>protein</td>
<td>protein-protein interaction</td>
</tr>
<tr>
<td>chemical compound</td>
<td>molecule</td>
<td>bond</td>
</tr>
</tbody>
</table>
Graph terminology

- **Path**: sequence of vertices connected by an edge.
- We write $u \rightsquigarrow v$ if there is a path between $u$ and $v$.
- A graph is **connected** if there is a path between each pair of vertices.
- **Connected component**: maximum connected subgraph.
- **Cycle**: path whose first and last vertices are the same.
- **Degree** of a vertex $u$, $d(u) =$ number of edges incident to $u$.
- **Neighborhood** of a vertex $u$, $N(u) = \{v : uv \in E\}$.

**Property:** $\sum_{u \in V} d(u) = 2|E|$
figure by Kevin Wayne
Complete graphs and complete bipartite graphs

Notation: $G = (U, V, E)$ a bipartite graph, means $U$ and $V$ is the partition.
Classes of graphs

- Complete graphs
- Bipartite graphs
- Planar graphs
- Interval graphs
- Chordal graphs
- Perfect graphs
- Distance hereditary graphs
- Claw-free graphs
- Intersection graphs
- Snarks
- Even-hole free graphs
Graph parameters

- $\delta(G)$: minimum degree.
- $\Delta(G)$: maximum degree.
- $\omega(G)$: clique number.
- $\alpha(G)$: size of a maximum independent set.
- $\chi(G)$: chromatic number.
- $\chi'(G)$: chromatic index (edge coloring).
- $\mu(G)$: size of a maximum matching (disjoint edges).
- $\tau(G)$: vertex cover.
- $\kappa(G)$: vertex connectivity.
- $\lambda(G)$: edge connectivity.
- Girth: length of a shortest cycle of $G$.
- $tw(G)$: treewidth, measure how much a graph looks like a tree.
Some graphs problems

- **Path.** Is there a path between \( s \) and \( t \)?
- **Shortest path.** What is the shortest path between \( s \) and \( t \)?
- **Cycle.** Is there a cycle in the graph?
- **Euler tour.** Is there a cycle that uses each edge exactly once?
- **Hamiltonian cycle.** Is there a cycle that uses each vertex exactly once.
- **Connectivity.** Is the graph connected?
- **MST.** What is the best way to connect all of the vertices?
- **k-connectivity.** Is there a set of at most \( k \) vertices whose removal disconnects the graph?
- **Planarity.** Can you draw the graph in the plane with no crossing edges?
- **Graph isomorphism.** Do two adjacency lists represent the same graph?

**Questions.** Which of these problems are easy? difficult? intractable?
How do we represent a graph?

- To a human?
- To a machine?
Graph representation: drawing

Graph drawing: provides intuition about the structure of the graph.

two drawings of the same graph

Caveat: can be misleading...
Representation for a machine

When we encode a graph in a machine we typically care about:

- The space it takes,
- Time to decide if two vertices are adjacent or not,
- Time to iterate over the neighborhood of a vertex,
- Time to add an edge.

Often, we assume that vertices are ordered, i.e. $V = \{1, \ldots, n\}$
Graph representation: list of edges

Maintain a list of the edges (linked-list or array)
Graph representation: adjacency matrix

Maintain a two-dimensional $V \times V$ boolean array $M$; for each edge $uv$ in the graph: $M[u][v] = M[v][u] = true$. 

![Graph representation with adjacency matrix](figure by Kevin Wayne)
Graph representation: adjacency list

Maintain vertex-indexed array of lists containing the neighbors of each vertex.
Graph representation: which one should we use?

- **Average degree** = \(\sum_{v \in V} d(v)/n = 2|E|/n\)
- A graph is **dense** if it has a lot of edges: \(\Omega(n^2)\) i.e. average degree is \(\Omega(n)\)
- A graph is **sparse** if it has not a lot of edges: \(o(n^2)\) i.e. average degree is \(o(n)\)

<table>
<thead>
<tr>
<th>representation</th>
<th>space</th>
<th>add edge</th>
<th>edge between v and w?</th>
<th>iterate over vertices adjacent to v?</th>
</tr>
</thead>
<tbody>
<tr>
<td>list of edges</td>
<td>E</td>
<td>1</td>
<td>E</td>
<td>E</td>
</tr>
<tr>
<td>adjacency matrix</td>
<td>(V^2)</td>
<td>1 *</td>
<td>1</td>
<td>V</td>
</tr>
<tr>
<td>adjacency lists</td>
<td>(E + V)</td>
<td>1</td>
<td>degree(v)</td>
<td>degree(v)</td>
</tr>
</tbody>
</table>

figure by Kevin Wayne
2 - Minimum Weighted Spanning Tree (MST)
Trees

A tree is a graph that is:

- connected and
- acyclic.

A forest is an acyclic graph (i.e. its connected component are trees).
**Trees**

A **tree** is a graph that is:
- **connected** and
- **acyclic**.

A **forest** is an acyclic graph (i.e. its connected component are trees).

**Tree’s characterizations:**
Let $G = (V, E)$ be a graph. The following are equivalent:
1. $G$ is a tree.
2. Any two vertices in $G$ are linked by a *unique path*.
3. $G$ is connected, and $|E| = |V| - 1$ ($m = n - 1$).
4. $G$ is acyclic, and $|E| = |V| - 1$ ($m = n - 1$).
5. $G$ is connected, but for every edge $e = uv$, $G - \{uv\}$ has exactly two connected components, one containing $u$ the other $v$.
6. $G$ is acyclic, but if any edge is added to $E$, the resulting graph contains a unique cycle (going through this added edge).

For a proof see Introduction to Algorithms, 3rd edition, Cormen, Leiserson, Rivest, Stein, Theorem B.2
Definitions

- Let $G = (V, E)$ be a graph.
- Let $\omega : E \to \mathbb{R}$ be a weight function on the edges of $G$.
- Then $G = (V, E, \omega)$ is called an (edge-)weighted graph.
Definitions

- Let $G = (V, E)$ be a graph.
- Let $\omega : E \to \mathbb{R}$ be a weight function on the edges of $G$.
- Then $G = (V, E, \omega)$ is called an (edge-)weighted graph.

- **Spanning tree**: subgraph of $G$ that is a tree and contains all vertices of $G$.
- The weight of a tree $T$ is the sum of the weights of its edges:

$$w(T) = \sum_{(u, v) \in T} w(u, v)$$
**Problem** (Minimum Weighted Spanning Tree (MST))

**Input**: A weighted graph $G = (V, E, \omega)$.

**Output**: A spanning tree of minimum weight.

---

![graph G](image)

*figure by Kevin Wayne*
**Problem (Minimum Weighted Spanning Tree (MST))**

**Input**: A weighted graph \( G = (V, E, \omega) \).

**Output**: A spanning tree of minimum weight.
Problem (Minimum Weighted Spanning Tree (MST))

Input: A weighted graph $G = (V, E, \omega)$.
Output: A spanning tree of minimum weight.

figure by Kevin Wayne
Problem (Minimum Weighted Spanning Tree (MST))

Input: A weighted graph $G = (V, E, \omega)$.

Output: A spanning tree of minimum weight.

spanning tree $T$: cost $= 50 = 4 + 6 + 8 + 5 + 11 + 9 + 7$

figure by Kevin Wayne
Combinatorial Optimization

In operations research, applied mathematics and theoretical computer science, **combinatorial optimization** is a topic that consists of finding an optimal object from a finite set of objects. In many such problems, exhaustive search is not tractable *according to wikipedia*. 

Famous combinatorial problems:
- Minimum Spanning Tree problem (“MST”),
- Knapsack problem,
- Travelling Salesman Problem (“TSP”).

Reference:
On the history of combinatorial optimization by Alexander Schriver
Combinatorial Optimization

In operations research, applied mathematics and theoretical computer science, **combinatorial optimization** is a topic that consists of finding an optimal object from a finite set of objects. In many such problems, exhaustive search is not tractable according to Wikipedia.

An **optimization problem** is a problem where you have to maximize (or minimize) a quantity.

Famous combinatorial problems:
- Minimum Spanning Tree problem ("MST"),
- Knapsack problem.
- Travelling Salesman Problem ("TSP").

Reference: *On the history of combinatorial optimization* by Alexander Schriver
Applications of MST

Many applications:

- Electrical, communication, road etc network design.
- Data coding and clustering.
- Approximate NP-complete graph optimisation.
  - Travelling salesman problem: the MST is within a factor of two of the optimal path.
- Image analysis.

http://www.geeksforgeeks.org/applications-of-minimum-spanning-tree/
If I give you a graph, how would you find a Minimum Spanning Tree?
The generic method

- A set of edges \( A \) is said to be promising if it can be completed into a minimum spanning tree.

Prior to each iteration, \( A \) is promising.

At each step, we determine an edge \( uv \) that we can add to \( A \) without violating this invariant, that is \( A \cup \{uv\} \) is still a promising tree.

Such an edge \( uv \) is said to be safe with respect to \( A \).
The generic method

- A set of edges $A$ is said to be promising if it can be completed into a minimum spanning tree.

The generic method manages a set of edges $A$, maintaining the following loop invariant:

Prior to each iteration, $A$ is promising

- At each step, we determine an edge $uv$ that we can add to $A$ without violating this invariant, that is $A \cup \{uv\}$ is still a promising tree.
- Such an edge $uv$ is said to be safe with respect to $A$. 
Pseudocode for the generic method

**Algorithm 1** Generic-MST\((G = (V, E, \omega))\)

\begin{enumerate}
\item \(A = \emptyset\)
\item \textbf{while} \(A\) is not a spanning tree \textbf{do}
\item \quad Find an edge \(uv\) that is safe for \(A\)
\item \quad \(A \leftarrow A \cup \{uv\}\)
\item \textbf{return} \(A\)
\end{enumerate}

**Proof of correctness:**

- **Initialization:** After line 1, the set \(A\) trivially satisfies the loop invariant.
- **Maintenance:** The loop in lines 2–4 maintains the invariant by adding only safe edges.
- **Termination:** All edges added to \(A\) are in a minimum spanning tree, and so the set \(A\) returned in line 5 must be a minimum spanning tree.

The tricky part is of course to find a safe edge.
How can we be sure that an edge is safe

We first need some definitions. Let $G = (V, E)$ be a graph.

- Let $S \subseteq V$, a cut $(S, V - S)$ is a partition of $V$.
- An edge is crossing the cut if it has one end in $S$ and the other in $V - S$. 
How can we be sure that an edge is safe

We first need some definitions. Let $G = (V, E)$ be a graph.

- Let $S \subseteq V$, a cut $(S, V - S)$ is a partition of $V$.
- An edge is crossing the cut if it has one end in $S$ and the other in $V - S$.
- We say that a cut respect a set of edges $A$ if no edge of $A$ is a crossing edge.
- A crossing edge is light if it is a crossing edge of minimum weight (there can be several light edges).
The safe property

**Safe Property:**
Let \( G = (V, E, \omega) \) and let \( A \) be a promising set of edges. Let \( (S, V – S) \) be a cut respecting \( A \) and let \( uv \) be a light crossing edge. Then \( uv \) is safe.

- The shaded edges form the promising set \( A \).
- The black vertices are in \( S \), the white in \( V – S \). Observe that it respects \( A \).
**Safe Property:**

Let $G = (V, E, \omega)$ and let $A$ be a promising set of edges. Let $(S, V - S)$ be a cut respecting $A$ and let $uv$ be a light crossing edge. Then $uv$ is safe.

- The shaded edges form the promising set $A$.
- The black vertices are in $S$, the white in $V - S$. Observe that it respects $A$.
- $dc$ is a light edge of the cut, it is claimed to be safe by the property.
Proof of the safe property

**Safe Property:**
Let $G = (V, E, \omega)$ and let $A$ be a promising set of edges. Let $(S, V - S)$ be a cut respecting $A$ and let $uv$ be a light crossing edge. Then $uv$ is safe.
Proof of the safe property

**Safe Property:**
Let $G = (V, E, \omega)$ and let $A$ be a promising set of edges. Let $(S, V - S)$ be a cut respecting $A$ and let $uv$ be a light crossing edge. Then $uv$ is safe.

**Proof** Let $T$ be a minimum spanning tree that includes $A$, and assume that $T$ does not contain the light edge $(u, v)$, since if it does, we are done. We shall construct another minimum spanning tree $T'$ that includes $A \cup \{(u, v)\}$ by using a cut-and-paste technique, thereby showing that $(u, v)$ is a safe edge for $A$.

The edge $(u, v)$ forms a cycle with the edges on the simple path $p$ from $u$ to $v$ in $T$, as Figure 23.3 illustrates. Since $u$ and $v$ are on opposite sides of the cut $(S, V - S)$, at least one edge in $T$ lies on the simple path $p$ and also crosses the cut. Let $(x, y)$ be any such edge. The edge $(x, y)$ is not in $A$, because the cut respects $A$. Since $(x, y)$ is on the unique simple path from $u$ to $v$ in $T$, removing $(x, y)$ breaks $T$ into two components. Adding $(u, v)$ reconnects them to form a new spanning tree $T' = T - \{(x, y)\} \cup \{(u, v)\}$.

It now suffices to prove that $\omega(T') \leq \omega(T)$ (on board).
Let $G = (V, E, \omega)$ be a weighted graph.
A spanning tree $T$ of $G$ is a minimum spanning tree if and only if for every edge $e \in E \setminus T$:

$$\omega(e) \geq \omega(f)$$

for every edge $f$ of the unique cycle of $T \cup \{e\}$

Hidden here: a matroid structure
Two greedy algorithms

We are going to see two greedy algorithms that solves MST:

- **Kruskal’s algorithm**: grows a forest whose trees coalesce into one spanning tree
- **Prim’s algorithm**: grows a tree until it becomes a spanning tree
Kruskal’s Algorithm
Kruskal’s algorithm

Kruskal algorithm is a greedy algorithm that grows a promising forest $A$:

- Start with the empty set (or more precisely with $(V, \emptyset)$)
- Add a minimum weighted edge that does not create a cycle.
  ⇔
  Add a minimum weighted edge that connects two connected components of the growing forest.

Show an execution
Kruskal’s algorithm

Kruskal algorithm is a greedy algorithm that grows a promising forest $A$:

- Start with the empty set (or more precisely with $(V, \emptyset)$)
- Add a minimum weighted edge that does not create a cycle.
  ⇔
  Add a minimum weighted edge that connects two connected components of the growing forest.

Show an execution

**Disjoint Set Data Structure** is perfect to implement it!
Algorithm 2 MST-Kruskal($G = (V, E, \omega)$)

1: $A = \emptyset$
2: for each vertex $u \in V$ do
3:   \hspace{1em} MAKE-SET($u$)
4: Sort the edges by non-decreasing order of weight
5: for each edge $uv$ taken in non-decreasing order do
6:   \hspace{1em} if FIND($u$) $\neq$ FIND($v$) then
7:     \hspace{2em} $A = A \cup \{uv\}$
8:     \hspace{2em} UNION($u$, $v$)
9: return $A$
Algorithm 3 MST-Kruskal\((G = (V, E, \omega))\)

1: \(A = \emptyset\)
2: \textbf{for} each vertex \(u \in V\) \textbf{do}
3: \hspace{1em} \textsc{Make-Set}(u)
4: Sort the edges by non-decreasing order of weight
5: \textbf{for} each edge \(uv\) taken in non-decreasing order \textbf{do}
6: \hspace{1em} \textbf{if} \textsc{Find}(u) \neq \textsc{Find}(v) \textbf{then}
7: \hspace{2em} \(A = A \cup \{uv\}\)
8: \hspace{1em} \textsc{Union}(u, v)
9: \textbf{return} \(A\)

---

Case 1: adding \(v-w\) creates a cycle

Case 2: add \(v-w\) to \(T\) and merge sets containing \(v\) and \(w\)

figure by Kevin Wayne
Algorithm 4 MST-Kruskal\((G = (V, E, \omega))\)

1. \(A = \emptyset\)
2. for each vertex \(u \in V\) do
3. \hspace{1em} Make-Set\((u)\)
4. Sort the edges by non-decreasing order of weight
5. for each edge \(uv\) taken in non-decreasing order do
6. \hspace{1em} if \(\text{Find}(u) \neq \text{Find}(v)\) then
7. \hspace{2em} \(A = A \cup \{uv\}\)
8. \hspace{2em} Union\((u, v)\)

return \(A\)

Complexity analyse (recall that \(|V| = n\) and \(|E| = m\)):

- \(\ell.\) 4: \(O(m \log m) = O(m \log(n))\) because \(m = O(n^2)\).
- \(\ell.\) 2-3 and 5-8: \(O(m + n) \alpha(n)^1\)
  - \(O(m)\) \text{Find-Set} and
  - \(O(n)\) \text{Union} and \text{Make-Set}

All together, the sorting win: \(O(m \log(n))\)

\(^1\)We implement the \textit{union-find} operations using the disjoint-set-forest implementation with the union-by-rank and path-compression heuristics
Correctness of Kruskal

Loop invariant:

Prior to each iteration, $A$ is promising
Correctness of Kruskal

Loop invariant:

Prior to each iteration, $A$ is promising

- Assume $A$ is the growing forest and the algorithm tells you to add the edge $uv$. 
Correctness of Kruskal

Loop invariant:

Prior to each iteration, $A$ is promising

- Assume $A$ is the growing forest and the algorithm tells you to add the edge $uv$.
- So $u$ and $v$ are in two distinct connected components of $A$. 
Correctness of Kruskal

Loop invariant:

Prior to each iteration, A is promising

- Assume A is the growing forest and the algorithm tells you to add the edge \( uv \).
- So \( u \) and \( v \) are in two distinct connected components of A.
- Take a cut \((S, V - S)\) such that:

\[\begin{align*}
\text{The connected component of } A \text{ containing } u \text{ is in } S, \\
\text{The connected component of } A \text{ containing } v \text{ is in } V - S, \\
\text{Put all the other connected components in } S.
\end{align*}\]

Then \((S, V - S)\) is a cut respecting A and \( uv \) is the lightest crossing edge.

So, by the Safe Property, \( uv \) is safe and thus \( A \cup \{uv\} \) is promising.
Correctness of Kruskal

Loop invariant:

Prior to each iteration, $A$ is promising

- Assume $A$ is the growing forest and the algorithm tells you to add the edge $uv$.
- So $u$ and $v$ are in two distinct connected components of $A$.
- Take a cut $(S, V - S)$ such that:
  - The connected component of $A$ containing $u$ is in $S$
Correctness of Kruskal

Loop invariant:

Prior to each iteration, $A$ is promising

- Assume $A$ is the growing forest and the algorithm tells you to add the edge $uv$.
- So $u$ and $v$ are in two distinct connected components of $A$.
- Take a cut $(S, V \setminus S)$ such that:
  - The connected component of $A$ containing $u$ is in $S$
  - The connected component of $A$ containing $v$ is in $V \setminus S$.
Correctness of Kruskal

Loop invariant:

Prior to each iteration, $A$ is promising

- Assume $A$ is the growing forest and the algorithm tells you to add the edge $uv$.
- So $u$ and $v$ are in two distinct connected components of $A$.
- Take a cut $(S, V - S)$ such that:
  - The connected component of $A$ containing $u$ is in $S$
  - The connected component of $A$ containing $v$ is in $V - S$.
  - Put all the other connected components in $S$. 
Correctness of Kruskal

Loop invariant:

Prior to each iteration, \( A \) is promising

- Assume \( A \) is the growing forest and the algorithm tells you to add the edge \( uv \).
- So \( u \) and \( v \) are in two distinct connected components of \( A \).
- Take a cut \((S, V - S)\) such that:
  - The connected component of \( A \) containing \( u \) is in \( S \)
  - The connected component of \( A \) containing \( v \) is in \( V - S \).
  - Put all the other connected components in \( S \).
- Then \((S, V \setminus S)\) is a cut respecting \( A \) and \( uv \) is the lightest crossing edge.
Correctness of Kruskal

Loop invariant:

Prior to each iteration, \( A \) is promising

- Assume \( A \) is the growing forest and the algorithm tells you to add the edge \( uv \).
- So \( u \) and \( v \) are in two distinct connected components of \( A \).
- Take a cut \( (S, V - S) \) such that:
  - The connected component of \( A \) containing \( u \) is in \( S \)
  - The connected component of \( A \) containing \( v \) is in \( V - S \).
  - Put all the other connected components in \( S \).
- Then \( (S, V \setminus S) \) is a cut respecting \( A \) and \( uv \) is the lightest crossing edge.
- So, by the Safe Property, \( uv \) is safe and thus \( A \cup \{uv\} \) is promising.
Prim’s Algorithm
Prim’s algorithm

Prim’s algorithm is like Kruskal, but instead of growing a forest, it grows a tree $A$.

- Start with a vertex
- **Repeat:** Add the lightest edge with exactly one extremity in $V(A)$
Priority-Queue

To find the next edge to add, we use a priority queue.
Priority Queue

Collection. Insert and Delete elements. Which item to delete.

Stack. Delete the element most recently added.

Queue. Delete the element the least recently added.

Priority Queue. Delete the largest (or smallest) items.

It generalises stacks and queues.
## Priority Queue Applications

- Event-driven simulation.  
  - customers in a line, colliding particles
- Numerical computation.  
  - reducing roundoff error
- Data compression.  
  - Huffman codes
- Graph searching.  
  - Dijkstra's algorithm, Prim's algorithm
- Number theory.  
  - sum of powers
- Artificial intelligence.  
  - A* search
- Statistics.  
  - maintain largest M values in a sequence
- Operating systems.  
  - load balancing, interrupt handling
- Discrete optimization.  
  - bin packing, scheduling
- Spam filtering.  
  - Bayesian spam filter

*Picture from Kevin Wayne*
A **priority queue** is a data structure that maintains a set $S$ of elements, each with an associated value called a *key*.

\[2\text{I let you guess the operations supported by a max-priority queue.}\]
Priority queue

A priority queue is a data structure that maintains a set $S$ of elements, each with an associated value called a key.

As for heap, there is min-priority queue and max-priority queue.

A min-priority queue supports the following operations$^2$:

- **Insert**($S, x$) inserts the element $x$ into the set $S$; i.e. $S \leftarrow S \cup \{x\}$.
- **Minimum**($S$) returns the element of $S$ with the smallest key.
- **Extract-min**($S$) removes and returns the element of $S$ with the smallest key.
- **Decrease-Key**($S, x, k$) does $x.key \leftarrow k$ ($k \leq x.key$ is assumed).

Efficient way to implement it: **min-heap**.

---

$^2$I let you guess the operations supported by a max-priority queue.
How to implement a priority queue efficiently
How to implement a priority queue efficiently

<table>
<thead>
<tr>
<th>implementation</th>
<th>insert</th>
<th>del max</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>unordered array</td>
<td>1</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>ordered array</td>
<td>N</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>goal</td>
<td>log N</td>
<td>log N</td>
<td>log N</td>
</tr>
</tbody>
</table>

Picture from Kevin Wayne
Max-heap

You already know:

- a max-heap can be implemented in an array.
- \( \text{Parent}(A, i) = \lfloor \frac{i}{2} \rfloor, \text{Right}(A, i) = 2i, \text{Left}(A, i) = 2i + 1: O(1). \)
- \( \text{Build-Heap}(A): O(n). \)
- \( \text{Max-Heapify}(A, i): O(\text{height}(i)) = O(\log n). \)

Figure from *Introduction to Algorithms*, 3rd edition, Cormen, Leiserson, Rivest, Stein
A wild complete binary tree
Max-Heapify(A, i)

Figure from *Introduction to Algorithms*, 3rd edition, Cormen, Leiserson, Rivest, Stein
Algorithm 5 Extract-Max(A)

1:  if A.size < 1 then
2:      error: "heap under flow"
3:  max ← A[1]
5:  A.size ← A.size − 1
6:  Heapify(A, 1)  ▷ Bubble down
7:  return max

Algorithm 6 Increase-Key(A, i, key)

1:  if A[i] < key then
2:      error: "new key is smaller than current key"
3:  while i > 1 and A[i] > A[parent(i)] do
4:      swap (A[i] and A[parent(i)])  ▷ Buble up
5:      i ← parent(i)
Algorithm 7  \textbf{Insert}(A, key)

1: \texttt{A.size} $\leftarrow$ \texttt{A.size} + 1
2: \texttt{A[A.size]} $\leftarrow$ $-\infty$
3: \textbf{Increase-Key}(A, A.size, key)

Hence:

- \textbf{Insert}(A, key): $O(\log n)$
- \textbf{Maximum}(A): $O(1)$
- \textbf{Extract-Max}(A): $O(\log n)$
- \textbf{Increase-Key}(A, i, key): $O(\log n)$
Many ways to implement a priority queue

<table>
<thead>
<tr>
<th>implementation</th>
<th>insert</th>
<th>del max</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>unordered array</td>
<td>1</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>ordered array</td>
<td>N</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>binary heap</td>
<td>log N</td>
<td>log N</td>
<td>1</td>
</tr>
<tr>
<td>d-ary heap</td>
<td>log d N</td>
<td>d log d N</td>
<td>1</td>
</tr>
<tr>
<td>Fibonacci</td>
<td>1</td>
<td>log N †</td>
<td>1</td>
</tr>
<tr>
<td>Brodal queue</td>
<td>1</td>
<td>log N</td>
<td>1</td>
</tr>
<tr>
<td>impossible</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

† amortized

order-of-growth of running time for priority queue with N items

Figure by Kevin Wayne
Back to Prim’s Algorithm
Pseudocode of Prim’s algorithm

- $Q$ is the priority queue, it contains the set of vertices not yet in the growing tree.
- $V - Q$ is the set of vertices covered by the growing tree.
- $v.key$ is the weight of the lightest edge connecting the growing tree with $u$.
- $v.\pi$ is the parent of $v$. 

```
Algorithm 8 MST-Prim (G = (V, E, ω), r)
1: for Each $v \in V$ do
2: $v.\pi \leftarrow \text{NIL}$ and $v.key \leftarrow +\infty$
3: $r.key \leftarrow 0$
4: $Q \leftarrow V \setminus \{r\}$
5: while $Q \neq \emptyset$ do
6: $u \leftarrow \text{Extract-Min}(Q)$
7: for each $v \in \text{Adj}[u]$ do
8: if $v \in Q$ and $v.key > \omega(uv)$ then
9: $v.key \leftarrow \omega(uv)$
10: $v.\pi \leftarrow u$
```
Pseudocode of Prim’s algorithm

- $Q$ is the priority queue, it contains the set of vertices not yet in the growing tree.
- $V - Q$ is the set of vertices covered by the growing the tree
- $v.key$ is the weight of the lightest edge connecting the growing tree with $u$.
- $v.\pi$ is the parent of $v$.

### Algorithm 9 MST-Prim($G = (V, E, \omega), r$)

1: for Each $v \in V$ do
2:   $v.\pi = NIL$ and $v.key = +\infty$
3: $r.key \leftarrow 0$
4: $Q \leftarrow V$
5: while $Q \neq \emptyset$ do
6:   $u \leftarrow$ Extract-Min($Q$)
7:   for each $v \in Adj[u]$ do
8:     if $v \in Q$ and $v.key > \omega(uv)$ then
9:       $v.key \leftarrow \omega(uv)$
10:      $v.\pi \leftarrow u$

\[ \text{▷ BUILD the priority queue} \]
\[ \text{▷ } u \text{ is added to the tree} \]
\[ \text{▷ For each neighbor } v \text{ of } u \]
\[ \text{▷ DECREASE-KEY}(Q, v, \omega(uv)) \]

- The algo maintains $A = \{(v, v.\pi) : v \in V - \{r\} - Q\}$
Proof of correctness and running time

**Algorithm 10** \( \text{MST-Prim}(G = (V, E, \omega), r) \)

1. **for** Each \( v \in V \) **do**
2. \( v.\pi = \infty \)
3. \( v.\text{key} = -\infty \)
4. \( r.\text{key} \leftarrow 0 \)
5. \( Q \leftarrow V \)
6. **while** \( Q \neq \emptyset \) **do**
7. \( u \leftarrow \text{Extract-Min}(Q) \)
8. **for** each \( v \in \text{Adj}[u] \) **do**
9. \( \text{if } v \in Q \text{ and } v.\text{key} > \omega(uv) \text{ then} \)
10. \( v.\text{key} \leftarrow \omega(uv) \)
11. \( v.\pi \leftarrow u \)

**Proof of correctness:** by the *safe property*

**Complexity:**
- \( O(m + n \log n) \) with a Fibonacci heap
- \( O(m \log n) \) with a min-heap
Recap
Comparing Primm’s and Kruskal’s Algorithms

Both algorithms choose and add at each step a min-weight edge from the remaining edges, subject to constraints.

Prim’s MST algorithm:
- Start at a root vertex.
- Two rules for a new edge:
  (a) No cycle in the subgraph built so far.
  (b) The connected subgraph built so far.
- Terminate if no more edges to add can be found.

At each step: an acyclic connected subgraph being a tree.

Kruskal’s MST algorithm:
- Start at a min-weight edge.
- One rule for a new edge:
  (a) No cycle in a forest of trees built so far.
- Terminate if no more edges to add can be found.

At each step: a forest of trees merging as the algorithm progresses (can find a spanning forest for a disconnected graph).
Recap

We are going to see two greedy algorithm that solves MST:

- **Kruskal’s algorithm:**
  - maintains a forest whose trees coalesce into one spanning tree,
  - can be implemented using a disjoint set data structure, runs in $O(m \log n)$.

- **Prim’s algorithm:**
  - Grows a tree (choose the lightest edge that does not create a cycle and maintain the constructing solution connected)
  - Prim’s algorithm implemented with a priority queue can run in $O(m + n \log n)$. 