Greedy graph algorithms

- Dijkstra’s algorithm
- Prim’s algorithm
- Kruskal’s algorithm
- Union-find data structure with path compression
What is the shortest path from a to n?

To every other node?

How can we find these paths efficiently?

For navigation, the edge weights are positive distances (obviously)

For some other graphs the weights can be a positive or negative cost

The problem is easier with positive weights
Dijkstra’s algorithm

- Given a directed graph \( G(V, E) \), a weight function \( w : E \rightarrow R \), and a node \( s \in V \), Dijkstra’s algorithm computes the shortest paths from \( s \) to every other node.
- The sum of all edge weights on a path should be minimized.
- A weight can e.g. mean metric distance, cost, or travelling time.
- For this algorithm, we assume the weights are nonnegative numbers.
Dijkstra's algorithm — overview

- input \( w(e) \) weight of edge \( e = (u, v) \). We also write \( w(u, v) \)
- output \( d(v) \) shortest path distance from \( s \) to \( v \) for \( v \in V \)
- output \( \text{pred}(v) \) predecessor of \( v \) in shortest path from \( s \) to \( v \in V \)
- A set \( Q \) of nodes for which we have not yet found the shortest path
- A set \( S \) of nodes for which we have already found the shortest path

procedure \( \text{dijkstra}(G, s) \)

\[
\begin{align*}
&d(s) \leftarrow 0 \\
&Q \leftarrow V - \{s\} \\
&S \leftarrow \{s\} \\
\text{while } Q \neq \emptyset &\\
&\text{select } v \text{ which minimizes } d(u) + w(e) \text{ where } u \in S, v \notin S, e = (u, v) \\
&d(v) \leftarrow d(u) + w(e) \\
&\text{pred}(v) \leftarrow u \\
&\text{remove } v \text{ from } Q \\
&\text{add } v \text{ to } S
\end{align*}
\]
Shortest paths

- Only $b$ has a predecessor in $S$
- $d(b) \leftarrow 4$
- $\text{pred}(b) \leftarrow a$
- $S \leftarrow \{a, b\}$
Shortest paths

- $d(b) + w(b, d) = 4 + 2 = 6$
- $d(b) + w(b, h) = 4 + 21 = 25$
- $d$ minimizes $d(u) + w(u, v)$
- $d(d) \leftarrow 6$
- $pred(d) \leftarrow b$
- $S \leftarrow \{a, b, d\}$
Shortest paths

- $d(b) + w(b, h) = 4 + 21 = 25$
- $d(d) + w(d, c) = 6 + 8 = 14$
- $d(d) + w(d, g) = 6 + 13 = 19$
- $c$ minimizes $d(u) + w(u, v)$
- $d(c) \leftarrow 14$
- $\text{pred}(c) \leftarrow d$
- $S \leftarrow \{a, b, c, d\}$
Shortest paths

- $d(b) + w(b, h) = 4 + 21 = 25$
- $d(d) + w(d, g) = 6 + 13 = 19$
- $d(c) + w(c, e) = 14 + 3 = 17$
- $e$ minimizes $d(u) + w(u, v)$
- $d(e) \leftarrow 17$
- $pred(e) \leftarrow c$
- $S \leftarrow \{a, b, c, d, e\}$
Shortest paths

- $d(b) + w(b, h) = 4 + 21 = 25$
- $d(d) + w(d, g) = 6 + 13 = 19$
- $d(e) + w(e, f) = 17 + 9 = 26$
- $g$ minimizes $d(u) + w(u, v)$
- $d(g) \leftarrow 19$
- $\text{pred}(g) \leftarrow d$
- $S \leftarrow \{a, b, c, d, e, g\}$
Shortest paths

- \( d(b) + w(b, h) = 4 + 21 = 25 \)
- \( d(e) + w(e, f) = 17 + 9 = 26 \)
- \( d(g) + w(g, h) = 19 + 7 = 26 \)
- \( d(g) + w(g, j) = 19 + 3 = 22 \)
- \( j \) minimizes \( d(u) + w(u, v) \)
- \( d(j) \leftarrow 22 \)
- \( \text{pred}(j) \leftarrow g \)
- \( S \leftarrow \{ a, b, c, d, e, g, j \} \)
Shortest paths

\begin{itemize}
  \item \(d(b) + w(b, h) = 4 + 21 = 25\)
  \item \(d(e) + w(e, f) = 17 + 9 = 26\)
  \item \(d(g) + w(g, h) = 19 + 7 = 26\)
  \item \(d(j) + w(j, m) = 22 + 3 = 25\)
  \item \(h\) and \(m\) minimize
    \[d(u) + w(u, v)\]
  \item We can take any of them
  \item \(d(h) \leftarrow 25\)
  \item \(\text{pred}(h) \leftarrow b\)
  \item \(S \leftarrow \{a, b, c, d, e, g, h, j\}\)
\end{itemize}
Shortest paths

- $d(e) + w(e, f) = 17 + 9 = 26$
- $d(j) + w(j, m) = 22 + 3 = 25$
- $d(h) + w(h, k) = 25 + 6 = 27$
- $m$ minimizes $d(u) + w(u, v)$
- $d(m) \leftarrow 25$
- $pred(m) \leftarrow j$
- $S \leftarrow \{ a, b, c, d, e, g, h, j, m \}$
Shortest paths

- $d(e) + w(e, f) = 17 + 9 = 26$
- $d(h) + w(h, k) = 25 + 6 = 27$
- $d(m) + w(m, n) = 25 + 5 = 30$
- $f$ minimizes $d(u) + w(u, v)$
- $d(f) \leftarrow 26$
- $pred(f) \leftarrow e$
- $S \leftarrow \{a, b, c, d, e, f, g, h, j, m\}$
Shortest paths

- $d(h) + w(h, k) = 25 + 6 = 27$
- $d(m) + w(m, n) = 25 + 5 = 30$
- $d(f) + w(f, i) = 26 + 6 = 32$
- $k$ minimizes $d(u) + w(u, v)$
- $d(k) \leftarrow 27$
- $\text{pred}(k) \leftarrow h$
- $S \leftarrow \{a - h, j, k, m\}$
Shortest paths

- $d(m) + w(m, n) = 25 + 5 = 30$
- $d(f) + w(f, i) = 26 + 6 = 32$
- $n$ minimizes $d(u) + w(u, v)$
- $d(n) \leftarrow 30$
- $pred(k) \leftarrow h$
- $S \leftarrow \{a - k, m, n\}$
Shortest paths

\[ d(f) + w(f, i) = 26 + 6 = 32 \]

Only \( i \) possible

\[ d(i) \leftarrow 32 \]

\[ \text{pred}(i) \leftarrow f \]

\[ S \leftarrow \{ a - k, m, n \} \]
Shortest paths

\[ d(i) + w(i, l) = 32 + 1 = 33 \]

Only \( l \) possible

\[ d(l) \leftarrow 33 \]

\[ \text{pred}(l) \leftarrow i \]

\[ S \leftarrow \{ a - n \} \]
Observations about Dijkstra’s algorithm

- We only add an edge when it really is to the node which is closest to the start vertex.
- To print the shortest path from $s$ to any node $v$, simply print $v$ and follow the $\text{pred}(v)$ attributes.
Dijkstra’s algorithm

Theorem

For each node \( v \in S \), \( d(v) \) is the length of the shortest path from \( s \) to \( v \).

Proof.

- We use induction with base case \( |S| = 1 \) which is true since \( S = \{s\} \) and \( d(s) = 0 \).
- Inductive hypothesis: Assume theorem is true for \( |S| \geq 1 \).
- Let \( v \) be the next node added to \( S \), and \( \text{pred}(v) = u \).
- \( d(v) = d(u) + w(e) \) where \( e = (u, v) \).
- Assume in contradiction there exists a shorter path from \( s \) to \( v \) containing the edge \( (x, y) \) with \( x \in S \) and \( y \not\in S \), followed by the subpath from \( y \) to \( v \).
- Since the path via \( y \) to \( v \) is shorter than the path from \( u \) to \( v \), \( d(y) < d(v) \) but it is not since \( v \) is chosen and not \( y \). A contradiction which means no shorter path to \( v \) exists.
Recall

**procedure** $dijkstra(G, s)$

$$d(s) \leftarrow 0$$

$Q \leftarrow V - \{s\}$

$S \leftarrow \{s\}$

while $Q \neq \emptyset$

- select $v$ which minimizes $d(u) + w(e)$ where $u \in S$, $v \not\in S$, $e = (u, v)$

- $d(v) \leftarrow d(u) + w(e)$

- $pred(v) \leftarrow u$

- remove $v$ from $Q$

- add $v$ to $S$

- We use a heap priority queue for $Q$ with $d(v)$ as keys.

- For $v \neq s$ we initially set $d(v) \leftarrow \infty$ and then decrease it.
Running time of Dijkstra’s algorithm

- Assume \( n \) nodes and \( m \) edges
- Constructing \( Q \): \( O(n) \) using heapify (but \( O(n \log n) \) using \( n \) inserts)
- Heapify is called init_heap in C and pseudo-code in the book
- \( O(n) \) iterations of the while loop
- Each selected node must check each neighbor not in \( S \) and possibly reduce its key
- \( O(m \log n) \) operations for reducing keys
- With all nodes reachable from \( s \), we have \( m \geq n - 1 \)
- Therefore \( (m \log n) \) running time
Assume the nodes are cities and a country wants to build an electrical network.

- The edge weights are the costs of connecting two cities.
- We want to find a subset of the edges so that all cities are connected, and which minimizes the cost.
- This problem was suggested to the Czech mathematician Otakar Borůvka during World War I for Mähren.
The minimum spanning tree problem

- In 1926 Borůvka published the first paper on finding the **minimum spanning tree**.
- It is an abbreviation of **minimum-weight spanning tree**.
- It has been regarded as the cradle of combinatorial optimization.
- Borůvka’s algorithm has been rediscovered several times: Choquet 1938, by Florek, Lukasiewicz, Steinhaus, and Zubrzycki 1951 and by Sollin 1965.
- We will study two classic algorithms for this problem:
  - Prim’s algorithm, and
  - Kruskal’s algorithm
- One of the currently fastest MST algorithm by Chazelle 2000 is based on Borůvka’s algorithm.
Consider a connected undirected graph $G(V, E)$

If $T \subseteq E$ and $(V, T)$ is a tree, it is called a spanning tree of $G(V, E)$

Given edge costs $c(e)$, a $(V, T)$ is a minimum spanning tree, or MST of $G$ such that the sum of the edge costs is minimized.

Prim’s algorithm is similar to Dijkstra’s and grows one MST

Kruskal’s algorithm instead creates a forest which eventually becomes one MST
A root node $s$ must first be selected.
Any will do.
How can we know which edge to add next?
Is it possible to do it with a greedy algorithm?
Compare with the Traveling Salesman Problem! (JS/Section 6.6)
TSP searches a path from one node which visits all nodes and returns.
TSP asks if there is such a tour of cost at most $x$?
We will next learn a rule which Prim’s and Kruskal’s algorithm rely on. It determines when it is safe to add a certain edge \((u, v)\).

A partition \((S, V - S)\) of the nodes \(V\) is called a cut.

An edge \((u, v)\) crosses the cut if \(u \in S\) and \(v \in V - S\).

Let \(A \subseteq E\) and \(A\) be a subset of the edges in some minimum spanning tree of \(G\).

\(A\) does not necessarily create a connected graph — \(A\) is applicable to both Prim’s and Kruskal’s algorithms and represents the edges selected so far.

An edge \((u, v)\) is safe if \(A \cup \{(u, v)\}\) is also a subset of the edges in some MST.

So how can we determine if an edge is safe?
Safe edges

Lemma

Assume $A$ is a subset of the edges in some minimum spanning tree of $G$, $(S, V - S)$ is any cut of $V$, and no edge in $A$ crosses $(S, V - S)$. Then every edge $(u, v)$ with minimum weight, $u \in S$, and $v \in V - S$ is safe.

Proof.

- Assume $T \subseteq E$ is a minimum spanning tree of $G$.
- We have either $(u, v) \in T$ (in which case we are done) or $(u, v) \notin T$.
- Without loss of generality we can assume $u \in S$ and $v \in V - S$.
- There is a path $p$ in $T$ which connects $u$ and $v$.
- Therefore $T \cup \{(u, v)\}$ creates a cycle with $p$.
- There is an edge $(x, y) \in T$ which also crosses $(S, V - S)$ and by assumption $(x, y) \notin A$. 
Proof.

- Since $T$ is a minimum spanning tree, it has only one path from $u$ to $v$.
- Removing $(x, y)$ from $T$ partitions $V$ and adding $(u, v)$ creates a new spanning tree $U$
  \[ U = (T - \{(x, y)\}) \cup \{(u, v)\} \]
- Since $(u, v)$ has minimum weight, $w(U) \leq w(T)$, and since $T$ is a minimum spanning tree, $w(U) = w(T)$
- Since $A \cup (u, v) \subseteq U$, $(u, v)$ is safe for $A$
Prim’s algorithm — overview

- input $w(e)$ weight of edge $e = (u, v)$. We also write $w(u, v)$
- a root node $r \in V$
- output minimum spanning tree $T$

procedure $prim(G, r)$

- $T \leftarrow \emptyset$
- $Q \leftarrow V - \{r\}$
- while $Q \neq \emptyset$
  - select a $v$ which minimizes $w(e)$ where $u \notin Q$, $v \in Q$, $e = (u, v)$
  - remove $v$ from $Q$
  - add $(u, v)$ to $T$
- return $T$

We use a heap priority queue for $Q$ with $d(v)$, the distance to any node in $V - Q$, as keys.
Prim’s algorithm has the same running time as for Dijkstra’s algorithm
Assume $n$ nodes and $m$ edges
Constructing $Q$: $O(n)$ using heapify (but $O(n \log n)$ using $n$ inserts)
$O(n)$ iterations of the while loop
Each selected node must check each neighbor not in $S$ and possibly reduce its key
$O(m \log n)$ operations for reducing keys
With all nodes reachable from $s$, we have $m \geq n - 1$
Therefore $(m \log n)$ running time
Kruskal’s algorithm — overview

- input $w(e)$ weight of edge $e = (u, v)$. We also write $w(u, v)$
- output minimum spanning tree $T$

**procedure** $\text{kruskal}(G)$

$T \leftarrow \emptyset$

$B \leftarrow E$

**while** $B \neq \emptyset$

select an edge $e$ with minimal weight

**if** $T \cup \{e\}$ does not create a cycle **then**

add $e$ to $T$

remove $e$ from $B$

**return** $T$

- How can we detect cycles?
The union-find data structure

- Consider a set, such as with $n$ nodes of a graph
- A union-find data structure lets us:
  - Create an initial partitioning $\{p_0, p_1, \ldots, p_{n-1}\}$ with $n$ sets consisting of one element each
  - Merge two sets $p_i$ and $p_j$
  - Check which set an elements belongs to
- The merge operation is called **union**
- The check set operation is called **find**
- We can use this as follows:
  - A set represents a connected subgraph and initially consists of one node
  - When we add an edge $(u, v)$ to the minimum spanning tree, we need to
    - Find the set $p_u$ with $u$
    - Find the set $p_v$ with $v$
    - Ignore $(u, v)$ if $\text{find}(u) = \text{find}(v)$
    - Note that the two subgraphs are connected using union otherwise
How should the sets $p_i$ be "named"?

It is only essential that two different sets have different names.

It is suitable to let the node $v$ be the initial name of $p_v$.

Thus no extra data type is needed. We simply add an attribute to the node.

Then after a union operation with $u$ and $v$ we set one of the nodes as the name of the merged set.

Assume we use $u$ as the name. Then $v$ needs a way to find $u$.

For this the node attribute $parent(v) = u$.

Code for find: if $parent(v) == \text{null}$ then $v$ else $parent(v)$.
Refer to Section 3.7 of JS.

Using both path compression and union-by-size (or union-by-rank), the time complexity of $m$ find and $n$ union operations is:

\[
\begin{align*}
\Theta(m\alpha(m, n)) & \quad m \geq n \\
\Theta(n + m\alpha(m, n)) & \quad m < n
\end{align*}
\]

\[\alpha(m, n) \leq 4 \text{ for all practical values of } m \text{ and } n\]
Running time of Kruskal’s algorithm

- Assume \( n \) nodes and \( m \) edges and \( m > n \)
- Sorting the edges: \( O(m \log m) \)
- Adding an edge \((v, w)\) would create a cycle if \( \text{find}(v) = \text{find}(w) \)
- There are \( m \) edges so we do at most \( 2m \) find operations
- A tree has \( n - 1 \) edges so we do \( n - 1 \) union operations
- From previous slide the complexity of these union-find operations is \( \Theta(m\alpha(m, n)) \)
- We can conclude that sorting the edges is more costly than the union-find operations so the running time of Kruskal’s algorithm is \( O(m \log m) \)
- We have \( m \leq n^2 \)
- Therefore \( O(m \log m) = O(m \log n^2) = O(m^2 \log n) = O(m \log n) \)
- I.e. the same as for Prim’s algorithm.