We’ve done

- Matroid Theory
  - Matroids and weighted matroids
  - Generic matroid algorithms
  - Minimum spanning trees

Now

- Task scheduling problem (another matroid example)
- Dijkstra’s algorithm (another greedy example)

Next

- Dynamic programming
A Task-Scheduling Problem

Input

- A set $S = \{1, \ldots, n\}$ of $n$ unit-time tasks;
- A set $D = \{d_1, \ldots, d_n\}$ of integer deadlines for the tasks $(1 \leq d_i \leq n)$;
- A set $W = \{w_1, \ldots, w_n\}$ of (positive) penalties for each task, i.e. task $i$ is penalized $w_i$ if it’s not finished by time $d_i$.

Output

- A schedule, i.e. a permutation of tasks, which minimizes the sum of penalties.
A Key Observation

Given a scheduling $\pi$ (just a permutation),

Let $P(\pi)$ denote the total penalty for $\pi$

Call a task *early* if it finishes at or before the deadline.

We can always transform $\pi$ into another schedule $\pi'$ in which the early tasks precedes the late tasks with $P(\pi') = P(\pi)$.

The problem reduces to finding a set $A$ of early tasks which minimizes total penalty.

(Equivalently, maximizes total penalty in $S - A$!)

Define a pair $M = (S, I)$ as follows

- $S$ is the set of tasks
- $A \in I$ if and only if there exists a schedule in which no tasks in $A$ is late

**Theorem 1.** $M$ is a matroid

**Proof.**
- **Hereditary:** obvious!
- **Exchange:** slightly less obvious. Use the next Lemma.
A Key Lemma

For each task set $A$, and a time $t$, let

$$S_t(A) = \{ i \in A \mid d_i \leq t \}$$

**Lemma 2.** $A$ is independent if and only if $|S_t(A)| \leq t$, for all $t = 1, 2, \ldots, n$. Moreover, we can schedule $A$ in increasing deadlines.

Now back to the exchange property:

Consider $A, B$, where $|A| < |B|$. We have $|S_n(A)| < |S_n(B)|$.

Let $k \geq 0$ be the least index for which:

$$|S_j(A)| < |S_j(B)| \text{ for all } j = k + 1, \ldots, n$$

$$|S_k(A)| \geq |S_k(B)|$$

Thus, we can take a task $x$ in $B - A$ with deadline $k + 1$ and add to $A$. The set $A \cup \{x\}$ is independent (why?).
Algorithm for Task Scheduling

We use MATROID-GREEDY in this context: \( D \) – corresponding deadlines, and \( W \) – corresponding penalties.

**Task-Scheduling** \((D, W, n)\)

1: \( A \leftarrow \emptyset \)
2: Sort \( W \) in **decreasing** (why?) order of penalty
3: Simultaneously move the deadlines in \( D \) correspondingly
4: // Now initialize array \( N \), where \( N[t] = |S_t(A)| \)
5: for \( t = 1 \) to \( n \) do
6: \( N[t] \leftarrow 0 \)
7: end for
8: for \( i = 1 \) to \( n \) do
9: \( OK \leftarrow \text{TRUE} // \) check if \( A \cup \{i\} \in \mathcal{I} \)
10: for \( j = 1 \) to \( D[i] \) do
11: if \( N[j] + 1 > j \) then
12: \( OK \leftarrow \text{FALSE} \)
13: end if
14: end for
15: if \( OK \) then
16: \( A \leftarrow A \cup \{s_i\} \)
17: end if
18: end for
A Small Summary on Priority Queues

A priority queue is a data structure

- maintains a set $S$ of objects
- each $s \in S$ has a key $key[s] \in \mathbb{R}$

Two types of priority queues: min-priority queue and max-priority queue

**Min-Priority Queue** – denoted by $Q$

- $\text{INSERT}(Q, x)$: insert $x$ into $S$
- $\text{MINIMUM}(Q)$: returns element with min key
- $\text{EXTRACT-MIN}(Q)$: removes and returns element with min key
- $\text{DECREASE-KEY}(Q, x, k)$: change the key of $x$ in $S$ into new key $k$, where $k \leq key[x]$

Using Heap, Min-PQ can be implemented so that:

- Building a $Q$ from an array takes $O(n)$
- Each of the operations takes $O(\lg n)$
Single Source Shortest-Paths Problem

Terminologies:

- Strictly speaking, a **path** in a graph $G$ is a sequence of vertices $P = (v_0, v_1, \ldots, v_k)$, where $(v_i, v_{i+1}) \in E$, and no vertex is repeated in the sequence.

- A **walk** is the same kind of sequences with repeated vertices allowed.

- If $w : E \rightarrow \mathbb{R}$, then
  
  $$w(P) = w(v_0v_1) + w(v_1v_2) + \cdots + w(v_{k-1}v_k).$$

Given a directed graph $G = (V, E)$, a source vertex $s \in V$.

A weight function $w : E \rightarrow \mathbb{R}^+$

Find a “shortest” path from $s$ to each vertex $v \in V$

Note:

- “Shortest” means least weight
- In general, $w : E \rightarrow \mathbb{R}$, we’ll discuss this later
- We want to find $n - 1$ shortest paths, not one
- Note also that the graph $G$ is directed (what if it wasn’t?)
Representing Shortest Paths

Given a directed graph $G = (V, E)$ and a source vertex $s \in V$.
We would like to represent the shortest paths from $s$ to each vertex $v \in V$.

**Lemma 3.** If a shortest path from $s$ to $v$ is $P = (s, \ldots, u, v)$,
then the part of $P$ from $s$ to $u$ is a shortest path from $s$ to $u$.

Hence, a representation of all shortest paths is as follows:

For each $v \in V$, maintain a pointer $\pi[v]$ to the previous vertex along a shortest path from $s$ to $v$
\[
\pi[s] = \text{NIL}
\]
\[
\pi[v] = \text{NIL} \text{ if } v \text{ is not reachable from } s
\]

Note:

- There could be multiple shortest paths to the same vertex
- The representation gives one set of shortest paths
- All SSSP algorithms we shall discuss produce a shortest paths tree
Shortest-Paths Trees

Given a directed graph $G = (V, E)$, a source $s$ and a weight function $w$

A shortest-paths tree rooted at $s$ is a directed subgraph $T = (V', E')$, where

- $V' \subseteq V$, $E' \subseteq E$ (part of being a subgraph)
- $V'$ is a set of vertices reachable from $s$
- $T$ forms a rooted tree with root $s$
- for all $v \in V'$, the unique simple path from $s$ to $v$ in $T$ is a shortest path from $s$ to $v$ in $G$

Note:

- We’ve noted that shortest paths are not necessarily unique
- SPTs are also not necessarily unique
Important Data Structures and Sub-routines

For each vertex \( v \in V(G) \):

- \( d[v] \): current estimate of the weight of a shortest path to \( v \)
- \( \pi[v] \): pointer to the previous vertex on the shortest path to \( v \)

\textbf{INITIALIZE-SINGLE-SOURCE}(G, s)

1: \textbf{for each} \( v \in V(G) \) \textbf{do}
2: \hspace{1em} \( d[v] \leftarrow \infty \)
3: \hspace{1em} \( \pi[v] \leftarrow \text{NIL} \)
4: \hspace{1em} \textbf{end for}
5: \hspace{1em} \( d[s] \leftarrow 0 \)

\textbf{RELAX}(u, v, w)

1: \textbf{if} \( d[v] > d[u] + w(u, v) \) \textbf{then}
2: \hspace{1em} \( d[v] \leftarrow d[u] + w(u, v) \)
3: \hspace{1em} \( \pi[v] \leftarrow u \)
4: \hspace{1em} \textbf{end if}
Dijkstra’s Algorithm

Pictorially, it operates very similar to Prim’s Algorithm

\[
\text{Dijkstra}(G, s, w)
\]

1. \text{INITIALIZE-SINGLE-SOURCE}(G, s)
2. \( S \leftarrow \emptyset \)  // set of vertices considered so far
3. \( Q \leftarrow V(G) \)  // \( \forall v, \text{key}[v] = d[v] \) after initialization
4. \textbf{while} \( Q \) is not empty \textbf{do}
5. \( u \leftarrow \text{EXTRACT-MIN}(Q) \)
6. \( S \leftarrow S \cup \{u\} \)
7. \textbf{for} each \( v \in \text{Adj}[u] \) \textbf{do}
8. \( \text{RELAX}(u, v, w) \)
9. \textbf{end for}
10. \textbf{end while}

This is a greedy algorithm because at every step, we add a vertex in \( V - S \) “closest” to \( S \) into \( S \).
Analysis of Dijkstra’s Algorithm

Let $n = |V(G)|$, and $m = |E(G)|$

- **INITIALIZE-SINGLE-SOURCE** takes $O(n)$
- Building the queue takes $O(n)$
- The while loop is done $n$ times, so **EXTRACT-MIN** is called $n$ times for a total of $O(n \log n)$
- For each $u$ extracted, and each $v$ adjacent to $u$, **RELAX**$(u, v, w)$ is called, hence totally $|E|$ calls to **RELAX** were made
- Each call to **RELAX** implicitly implies a call to **DECREASE-KEY**, which takes $O(\log n)$; hence, totally $O(m \log n)$-time on **DECREASE-KEY**

In total, we have $O((m + n) \log n)$, which could be improved using **FIBONACCI-HEAP** to implement the priority queue
Correctness of Dijkstra’s Algorithm

Let $\delta(s, v)$ denote the weight of a shortest path from $s$ to $v$.

We note the following facts

- $d[v] \geq \delta(s, v)$, and once $d[v] = \delta(s, v)$ it never changes.
- If $v$ is not reachable from $s$, then $d[v] = \infty$ always.
- If there is a path from $s$ to $u$, and there is an edge $uv$, and $d[u] = \delta(s, u)$ at any time before the call to \text{RELAX}(u, v, w), then $d[v] = \delta(s, v)$ after the call.

**Theorem 4.** Dijkstra’s Algorithm terminates with $d[v] = \delta(s, v)$ for all $v \in V$.

**Proof.** We show that at the start of each iteration, $d[v] = \delta(s, v)$ for each $v \in S$.

(Note: this is like induction on the number of steps of the algorithm.)

**Lemma 5.** The predecessor subgraph produced by the $\pi$’s is a shortest-paths tree.