Initialization for Shortest Paths

All shortest-paths algorithms start with

```
Init-Single-Source(V,s)
for each v \in V do
  d[v]=\infty
  \pi[v]=\text{NIL}
  d[s]=0
```
Relaxation

Can we improve the shortest-path estimated for \( v \) going through \( u \) and taking \((u,v)\)?

\[
\text{Relax}(u,v,w) \\
\text{if } d[v] > d[u] + w(u,v) \\
\text{then } d[v] = d[u] + w(u,v) \\
\pi[v] = u
\]
Scheme for Single-Source Shortest-Paths Algorithms

- Start by calling Init-Single-Source
- Relax edges
- Different algorithms differ on
  - Number of relaxations
  - Order of relaxations
- Bellman-Ford: \(|V|-1\) consecutive relaxations
- Dijkstra: “greedy” relaxation
Dijkstra Algorithm for Shortest Paths

INPUT:

- A directed graph $G=(V,E)$
- Source $s$
- A weight function $w: E \rightarrow \mathbb{R}^+$
  - $w(u,v) \geq 0$, $(u,v) \in E$

No problem with negative-weight cycles

Maintain a set $S \subseteq V$ whose final shortest-path weights from $s$ have been determined
Dijkstra Algorithm

Dijkstra(G,w,s)
Initialize-Single-Source(G,s)
S = 0
Q = V
while Q ≠ 0 do
  u = Extract-Min(Q)
  S = S ∪ {u}
  for each vertex v ∈ Adj[u] do Relax(u,v,w)
Shortest Paths Properties

- Upper-bound Property: Always have $d[v] \geq \delta(s,v)$ for all $v$. When $d[v] = \delta(s,v)$ it never changes.

- No-path property: If $\delta(s,v) = \infty$ then $d[v] = \infty$ always.

- Convergence property: If $s \rightarrow u \rightarrow v$ is a shortest path, $d[u] = \delta(s,u)$ and we call Relax($u,v,w$) then $d[v] = \delta(s,v)$ afterward.
Dijkstra Correctness, 1

Dijkstra maintains the invariant $Q = V \setminus S$ at the start of each iteration of the while loop:

- **Initialization:** It is clearly true before the while ($S = 0$ and $Q = V$)
- **Maintenance:** $u$ is extracted from $Q = V \setminus S$ and inserted in $S$ (first time, $u = s$)
- **Termination:** $Q = 0$, and $S = V$
Correctness, 2

Theorem: Dijkstra algorithm, run on a weighted, directed graph $G=(V,E)$ with weight function $w$ and source $s$, terminates with $d[v]=d(s,v)$ for all vertices $v \in V$. 
Correctness, 3

**Proof:** We use the following invariant:

At the start of each iteration of the while loop \( d[v] = d(s, v) \) for all \( v \in S \)

It suffices to show that \( d[v] = d(s, v) \) at the time \( v \) is added to \( S \). Once \( d[v] = d(s, v) \) we use the upper-bound property to show that the equality holds at all times thereafter.
Correctness, 4

Initialization: It is $S=0$, so, true

Maintenance: By contradiction, let $u \neq s$ be the first vertex such that $d[u] \neq d(s,u)$ when it is added to $S$. Right before $u$ is added to $S$, it is $S \neq 0$. Then there must be a path from $s$ to $u$, otherwise $d[u] = d(s,u) = \infty$ by the no-path property which would violate $d[u] \neq d(s,u)$. 

Correctness, 5

If there is at least one path, there is a shortest path $p = s \rightarrow u$ which connects $s \in S$ to $u \in V \setminus S$. Let us decompose $p$ in $p_1 = s \rightarrow x$ and $p_2 = y \rightarrow u$ with $x$ the predecessor of $y$: $p = s \rightarrow x \rightarrow y \rightarrow u$ (either $p_1$ or $p_2$ may have no edges) with $x \in S$ and $y$ the first vertex in $V \setminus S$. Claim: $d[y] \neq d(s, y)$ when $u$ is added to $S$. 
Correctness, 6

Since $x \in S$ and $u$ was chosen as the first vertex such that $d[u] \neq d(s,u)$ when $is$ was added to $S$, it is $d[x] \neq d(s,x)$ when $x$ was added to $S$. Edge $(x,y)$ was relaxed at that time, so the claim follows from the convergence property.
Correctness, 7

Since p is a shortest path form s to u and y comes before u, and since there are no negative edges it is $d(s,y) \leq d(s,u)$. Thus: $d[y] = d(s,y) \leq d(s,u) \leq d[u]$ (upper bound property). But because both vertices were in $V \setminus S$ when u was chosen we have $d[u] \leq d[y]$, which imposes $d(s,u) = d[u]$, a contradiction.
Correctness, 8

**Termination:** At termination, $Q=0$, which, along with the invariant $Q=V \setminus S$, implies $S=V$. Thus, $d[v]=d(s,v)$ for each vertex $v$ in $V$.

**Corollary:** At termination the predecessor subgraph $G_\pi$ is a shortest path tree rooted at $s$. 
Binary Heaps

- A binary heap is an (array) object that can be seen as a nearly complete binary tree.
- The tree is completely filled on all levels except, possibly, the lowest, which is partially filled from the left.

- Two kind of binary heaps:
  - Max-heaps, and
  - Min-heaps
Priority Queues

A priority queue is a data structure for maintaining a set $S$ of elements, each with a key.

A min-priority queue supports the operations:

- $\text{Insert}(S, x)$, insertion
- $\text{Minimum}(S)$, returns the element with the largest key
- $\text{Extract-Min}(S)$, removes and returns the min
- $\text{Decrease-Key}(S, x, k)$ decreases the key of $x$ to the new value $k$ (assumed smaller than $\text{key}[x]$)
Heaps for Priority Queues

Given the operations on binary heaps, the operations on a priority queue cost:

- Insert: $O(\log n)$
- Minimum: $O(1)$
- Extract-Min: $O(\log n)$
- Decrease-Key: $O(\log n)$

A heap can support any priority queue operations on a set of size $n$ in $O(\log n)$ time (worst case)
Fibonacci Heaps

- Heap operations that do not involve deletion are implemented in $O(1)$ amortized time
- Desirable when Extract-Min and Delete are small compared to other operations
- A Fibonacci heap is a collection of trees
- (Not of practical use sometimes ... )
Dijkstra Analysis, 1

Dijkstra maintains a min-priority queue by calling three operations:

- Insert (implicit in Q=V)
- Extract-Min
- Decrease-Key (implicit in Relax)

Insert and Extract-Min are invoked one per vertex

Decrease-Key is executed |E| times (once per edge)
Dijkstra running time depends on how we implement the priority queue.

Being the vertices number from 1 to \(|V|\) we can store \(d[v]\) in the \(v\)-th position if an array:

- Insert and Decrease-Key takes \(O(1)\)
- Extract-Min takes \(O(V)\)

\(O(V^2 + E) = O(V^2)\)
If the graph is sparse the priority queue can be implemented by a binary min-heap
- Insert: $O(\log V) \rightarrow O(V \log V)$ to build the heap
- Decrease-Key takes $O(\log V)$
- Extract-Min takes $O(\log V)$

$O((V+E) \log V) = O(E \log V)$ if all vertices are reachable from the source

Improvement over $O(V^2)$ when $|E| = o(V^2/\log V)$
Analysis, 4

- Using a Fibonacci heap
- Still $O(V)$ to build the heap
- Amortized cost of each of the $|V|$ Extract-Min is $O(\log V)$
- Amortized cost of each of the $|E|$ Decrease-Key is $O(1)$
- Dijkstra cost: $O(V \log V + E)$
Assignments

- Textbook, Chapter 24, pages 595—614
- Updated information on the class web page:
  www.ece.neu.edu/courses/eceg205/2003fa