Greedy Algorithms II [Chapter 24 cormen 2001]

- Today's class will discuss Dijkstra's algorithm which is a greedy algorithm
- But first we need to talk about single-source shortest paths

Single-source shortest paths:

- In a shortest-paths problem we are given a weighted directed graph $G = (V, E)$ with weight function $w: E \to \mathbb{R}$ mapping edges to real-valued weights
- The weight $w(p)$ of path $p = \langle v_0, v_1, \ldots, v_k \rangle$ is the sum of the weights of its constituent edges:

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

- We define the shortest-path weight $\delta(u, v)$ from u to v by:

$$\delta(u, v) = \begin{cases} \min\{w(p) : u \overset{p}{\rightsquigarrow} v\}, & \text{if there is a path from } u \text{ to } v \\ \infty, & \text{otherwise} \end{cases}$$

- A shortest path from vertex u to vertex v is then defined as any path p with weight $w(p) = \delta(u, v)$

- Single-source shortest-paths problem: given a graph $G = (V, E)$, we want to find a shortest path from a given source vertex $s \in V$ to each vertex $v \in V$

- Shortest-paths algorithms typically rely on the property that a shortest path between two vertices contains other shortest paths within it
  - Optimal substructure property
    - We saw this on the class "Greedy Algorithms I"
      - Remember MSTs?

- The following Lemma states the optimal-substructure property of shortest paths more precisely:

Lemma 24.1 - Subpaths of shortest paths are shortest paths

Given a weighted, directed graph $G = (V, E)$ with weight function $w: E \to \mathbb{R}$, let $p = \langle v_0, v_1, \ldots, v_k \rangle$ be a shortest path from vertex v_0 to vertex v_k and, for any i and j such that $0 \le i \le j \le k$, let $p_{ij} = \langle v_i, v_{i+1}, \ldots, v_j \rangle$ be the subpath of p from vertex v_i to v_j. Then p_{ij} is a shortest

- Proof:

- If we decompose path p into:

$$v_0 \overset{p_{0i}}{\rightsquigarrow} v_i \overset{p_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k$$

Then we have that:

$$w(p) = w(p_{0i}) + w(p_{ij}) + w(p_{jk})$$

- Now, assume that there is a path p'_{ij} from v_i to v_j with weight $w(p'_{ij}) < w(p_{ij})$. Then:

$$v_0 \overset{p_{0i}}{\rightsquigarrow} v_i \overset{p'_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k$$

is a path from v_0 to v_k whose weight $w(p_{0i}) + w(p'_{ij}) + w(p_{jk})$ is less than $w(p)$, which contradicts the assumption that p is a shortest path from v_0 to v_k. ∎

- Issues regarding cycles (1/2):

  - Negative-weight edges: If the graph contains a negative-weight cycle reachable from s, ~~however~~ no path from s to a vertex on the cycle can be a shortest path. We can always find a path with lower weight by following the proposed "shortest" path and then traversing the negative-weight cycle. We would thus obtain another "shortest" path

- Issues regarding cycles (2/2):
  - "Normal" cycles: A shortest path cannot a normal cycle, since removing the cycle from the path produces a path with the same source and destination and a lower weight.

∴ 1 Shortest paths cannot contain: { negative, ~~positive~~ positive } cycles

- That leaves only 0-weight cycles. We can remove a 0-weight cycle from any path to produce another path whose weight is the same.

∴ 2 We can assume, without loss of generality, that when we are finding shortest paths we have no cycles.

- We often wish to compute not only shortest-path weights, but the vertices on shortest paths as well. Given a graph $G = (V, E)$ we maintain for each vertex $v \in V$ a predecessor $v.\pi$ that is either another vertex or NIL.

- Relaxation:
  - The algorithms in Chapter 24 of the book use the technique of relaxation.
  - For each vertex $v \in V$ we maintain an attribute $v.d$ which is an upper bound on the weight

- We call $v.d$ a shortest-path estimate. We initialize the shortest-path estimates and predecessors by the following $\Theta(V)$-time procedure:

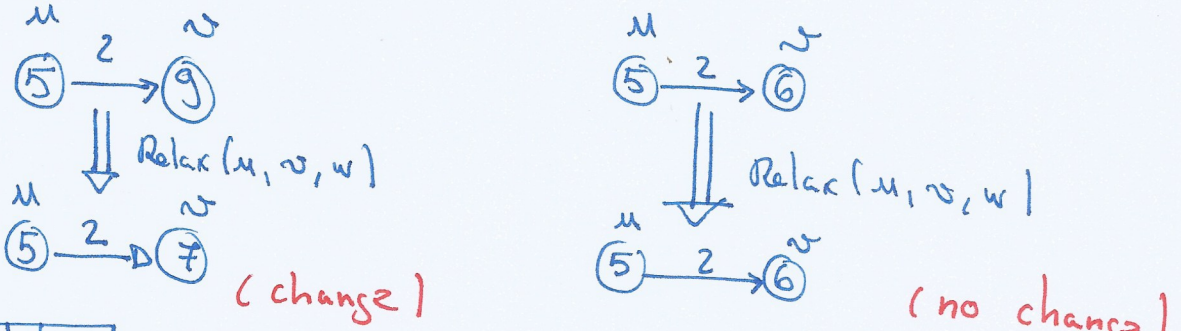
```
Initialize Single Source (G, s)
    for each vertex v ∈ G.V
        v.d = ∞
        v.π = Nil
    s.d = 0
```

- The process of relaxing an edge (u, v) consists of testing whether we can improve the shortest path to v found so far by going through u and, if so, updating $v.d$ and $v.\pi$

- A relaxation step may decrease the value of the shortest-path estimate $v.d$ and update v's predecessor attribute $v.\pi$. The following code performs a relaxation step on edge (u, v) in $O(1)$ time:

```
Relax (u, v, w)
    If v.d > u.d + w(u, v)
        v.d = u.d + w(u, v)
        v.π = u
```

- Example:



Notes:

- The shortest path estimate of each verter appears within the verter

- Relaxation is the only means by which shortest path estimate and predecessors change.

- Properties of shortest paths and relaxation
  - Proofs can be found on section 24.5 of cormen2001
  - Triangle inequality (Lemma 24.10)
    For any edge $(u,v) \in E$, we have $\delta(s,v) \leq \delta(s,u) + w(u,v)$ (shortest-path weight)
  - Upper-bound property (Lemma 24.11)
    We always have $v.d \geq \delta(s,v)$ for all vertices $v \in V$, and once $v.d$ achieves the value $\delta(s,v)$ it never changes
  - No-path property (Lemma 24.12)
    If there is no path from s to v, then we always have $v.d = \delta(s,v) = \infty$

Convergence Property (Lemma 24.14)

path to u — edge (u, v)

If $s \rightsquigarrow u \rightarrow v$ is a shortest path in G for some $u, v \in V$ and if $u.d = \delta(s, u)$ at any time prior to relaxing edge (u, v) then $v.d = \delta(s, v)$ at all times afterward

Path-Relaxation Property (Lemma 24.15)

- If $p = \langle v_0, v_1, \ldots, v_k \rangle$ is a shortest path from $s = v_0$ to v_k and we relax the edges of p in the orde $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$ then $v_k.d = \delta(s, v_k)$ This property holds regardless of any other relaxation step that occur, even if they are intermixed with relaxations the edges of p.

Predecessor-subgraph property (Lemma 24.17)

Once $v.d = \delta(s, v)$ for all $v \in V$, the predecessor subgraph is a shortest-paths tree rooted at s

24.3 Dijkstra's algorithm

- Bellman-Ford algorithm run in $O(VE)$
- Dijkstra's algorithm solves the single-source shortest-p problem on a weighted directed graph $G = (V, E)$ for th case in which all edge weights are nonnegative.
- I.e. assume $\forall_{(u,v) \in E} w(u,v) \geq 0$
- With a good implementation the running time is better than Bellman-Ford
- Dijkstra's algorithm maintains a set S of vertic whose final shortest-path weights from the source have already been determined
- The algorithm repeatedly selects the vertex $u \in V-$ with the minimum shortest-path estimate, adds u t and relaxes all edges leaving u
- The following implementation uses a min-priority queue of vertices, keyed by their d values.

Question:
Do this remind yo anything?

Dijkstra (G, w, s)

Initialize Single Source (G, s)

$S = \emptyset$

$Q = G.V$

while $Q \neq \emptyset$

u = Extract Min (Q)

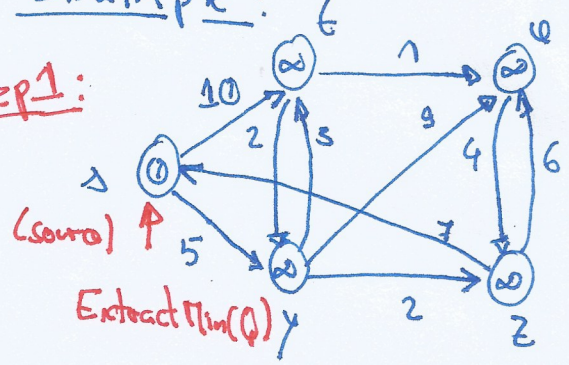
$S = S \cup \{u\}$

for each vertex $v \in G.Adj[u]$

Relax (u, v, w) /* Can we improve the shortest path estimate? */

- Example:

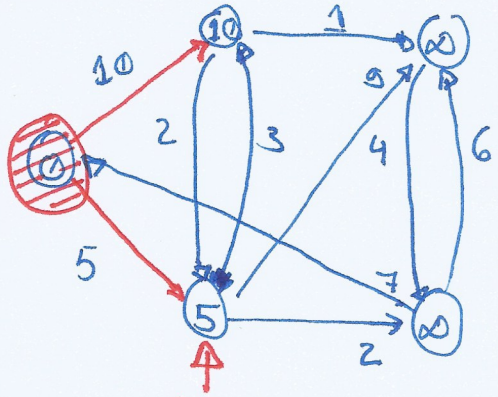
Step 1:



Caption:

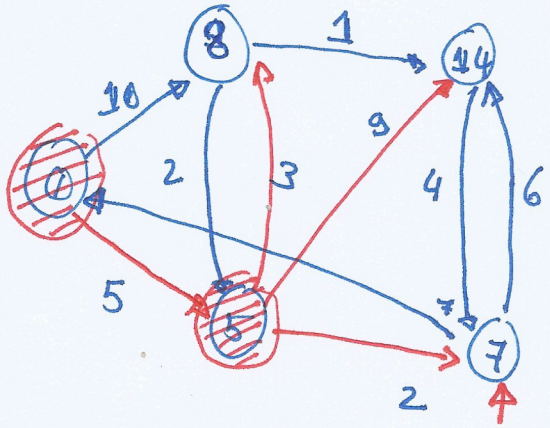
- Shortest path estimates appe within the vertices
- ~~Shaded~~ edges marked with — indicate predecess values
- Vertices marked as ◯ are in the set S
- Vertices not marked a are in the min priority queue.

Step 2:



ExtractMin(Q)

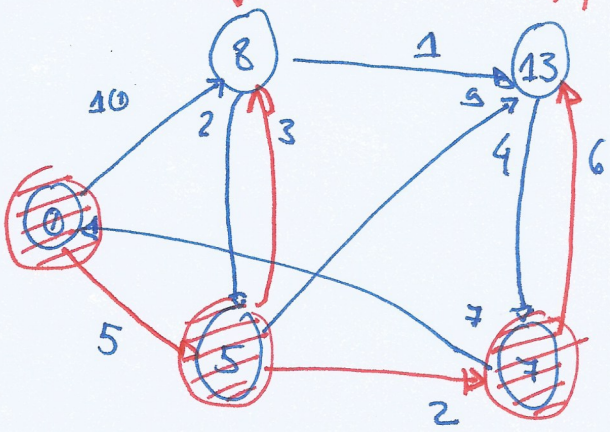
Step 3:



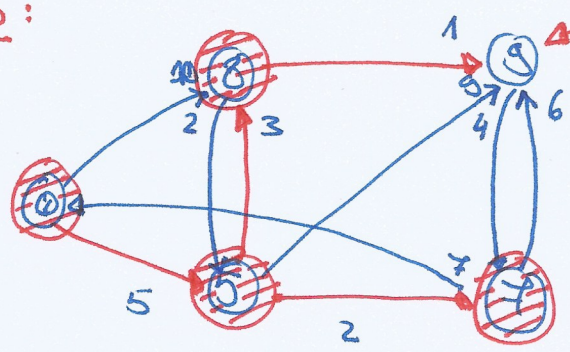
ExtractMin(Q)

Step 4:

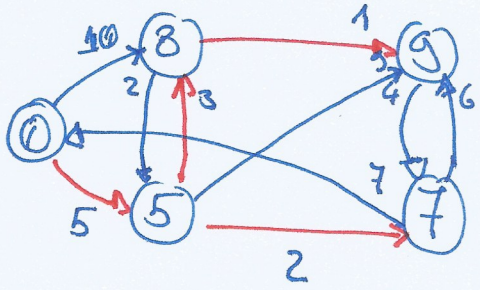
ExtractMin(Q)



Step 5:



Step 6:



Nothing to update

Notes:

- The algorithm never inserts vertices into Q after the initialization
- Each vertex is extracted from Q and added to S exactly once, so the loop iterates |V| times
- Because Dijkstra's algorithm always chooses the closest vertex in V-S to add to set S we say it uses a greedy strategy.

Question:

But how can we know that Dijkstra's algorithm does indeed compute shortest-paths?

Idea: Show that each time a vertex u is added to set S we have $u.d = \delta(s, u)$

Theorem 24.6 – Correctness of Dijkstra's algorithm

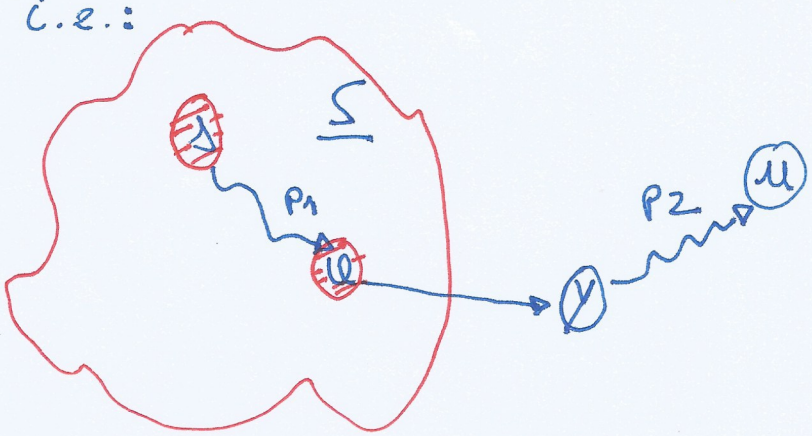
- At the start of each iteration of the loop:

  $v.d = \delta(s, v)$ $\forall v \in S$

  We need to show that this condition always holds

- First iteration (initialization): there are no vertices in S so the condition is true
- Maintenance:
  - We wish to show that $u.d = \delta(s, u)$ for the vertex added to S
  - For the purpose of contradiction, let u be the first vertex for which $u.d \neq \delta(s, u)$ when u is added to S
  - We must have $u \neq s$ because s is the first vertex added to S and $s.d = \delta(s, s) = 0$.

- Because $u \neq s$ we also have that $S \neq \emptyset$ just before u is added to S
- There must be some path from s to u for otherwise $u.d = \delta(s,u) = \infty$ by the no-path property, which would violate our assumption that $u.d \neq \delta(s,u)$
- Because there is at least one path, there is a shortest path p from s to u
- Prior to adding u to S path p connects a vertex in S, namely s, to a vertex in $V-S$, namely u.
- Let us consider the first vertex y along p such that $y \in V-S$ and let $u \in S$ by y's predecessor along p
- Then we can decompose path p into $s \overset{p_1}{\rightsquigarrow} u \rightarrow y \overset{p_2}{\rightsquigarrow}$ i.e.:



- We claim that $y.d = \delta(s,y)$ when u is added to S. This happens because we chose u as the first vertex for which $u.d \neq \delta(s,u)$ when it is added to S.
- We had $x.d = \delta(s,x)$ when x was added to S.
- Edge (x,y) was relaxed at that time and the claim follows from the convergence property (page 7 of these notes).
- We can now obtain a contradiction to prove that $u.d$ must be $\delta(s,u)$, i.e.: $u.d = \delta(s,u)$.
- Because y appears before u on a shortest path from s to u and all edge weights are non-negative we have $\delta(s,y) \leq \delta(s,u)$ and thus

$$\begin{aligned} y.d &= \delta(s,y) \\ &\leq \delta(s,u) \\ &\leq u.d \quad \text{(by the upper-bound pro} \end{aligned}$$

- But because both vertices u and y were in $V-S$ when u was chosen we have $u.d \leq y.d$. ~~I.e. we have not yet updated our estimate for u.~~ This happens because the ExtractMin(Q) operation will return vertex u

- Thus

$$\begin{cases} y.d \leq u.d \\ u.d \leq y.d \end{cases} \Rightarrow y.d = u.d \Leftrightarrow y.d = \delta(s,y) = \delta(s,u) = u.d$$

- Consequently: $u.d = \delta(s,u)$ which contradicts our choice of u.

∴ $u.d = \delta(s,u)$ when u is added to S and that this equality is maintained all times thereafter.

— Termination:

- At termination $Q = \emptyset$ but initially $Q = V - S$

$$\begin{cases} Q = \emptyset \\ Q = V - S \end{cases} \Rightarrow \emptyset = V - S \Leftrightarrow V = S$$

- Thus, when combined with our maintenance result we know that $\delta(s,u) \; \forall u \in V$

■ end of proof (finally !! ☺)

Question: How fast is Dijkstra's algorithm?

- Algorithm maintains min-priority queue Q by the operations:
  - Insert
  - ExtractMin

  } These are called once per vertex
  - DecreaseKey (From the relax operation)
- Each vertex u is added to S exactly once and each edge in the adjacency list $adj[u]$ is also examined exactly once
- Total number of edges is $|E|$
  - ↳ Therefore DecreaseKey is called $|E|$ times
- Overall complexity depends on how Q is implemented:

Q	Insert	ExtractMin	DecreaseKey	Total
Array	$O(1)$	$O(V)$	$O(1)$	$O(V^2+E)$
Heap	$O(\lg V)$	$O(\lg V)$	$O(\lg V)$	$O((V+E)\lg V)$
Fib Heap	$O(1)$	$O(\lg V)$	$O(1)$	$O(V\lg V+E)$

Total co
all verti
the conn
for loo