1) define & draw the graph (what is vertices, what is edges)
2) what algorithm will you use
3) write pseudocode

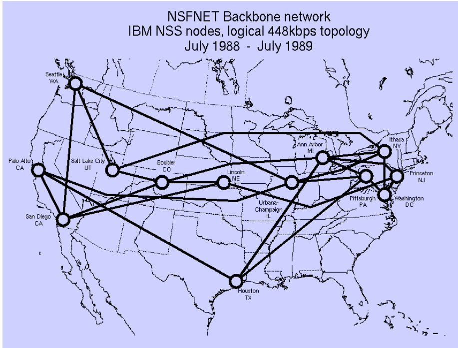
Shortest-Paths Algorithm Overview

BFS(Breadth first search) is good for finding shortest path

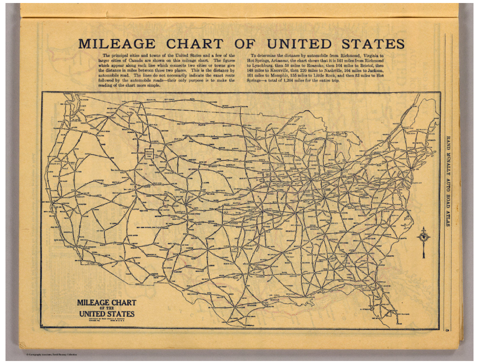
What Makes Shortest-Paths Algorithms Possible?

Shortest-Paths Problem

- Representative motivating example
  - You use it everyday on online maps, GPS navigation, internet packet routing, …



http://hpwren.ucsd.edu/~hwb/NSFNET/NSFNET-200711Summary/



From http://www.davidrumsey.com

Shortest-Paths Problem

- Representative motivating example
  - You use it everyday on online maps, GPS navigation, internet packet routing, …
- Formal definition of a shortest-paths problem:
  - Given a weighted, directed graph $G = (V, E)$, with weight function $w : E \to \mathbb{R}$,
  - The ***weight*** $w(p)$ ***of path*** $p = \langle v_0, v_1, \ldots, v_k \rangle$ is the sum of 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:
    - $\min\{w(p)$: for any path p from u to $v\}$ if there exists a path from u to v.
    - ∞ otherwise.
  - A ***shortest path*** from vertex u to vertex v is defined as any path p with weight $w(p) = \delta(u, v)$.

http://hpwren.ucsd.edu/~hwb/NSFNET/NSFNET-200711Summary/

From http://www.davidrumsey.com

Variants Of Shortest-Paths Problem

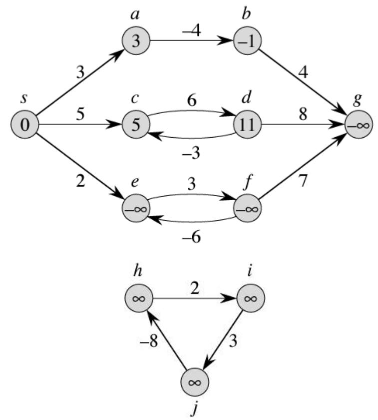
- On unweighted graphs
  - Use BFS (Breadth-First Search) that we learned earlier.
- Single-source shortest-paths problem
  - Find a shortest path from a given ***source*** vertex to each vertex
  - Seemingly doing more than what's needed for a ***single-pair*** shortest-path problem, but all known single-pair shortest-path algorithms have the same worst-case asymptotic running time as single-source ones.
- Single-destination shortest-paths problem
  - Reverse all edges and solve single-source shortest-paths problem
- All-pairs shortest-paths problem
  - Find a shortest path between every pair of two vertices.
  - Can solve this by running single-source algorithm once from each vertex.
  - Can solve it faster. See CLRS Ch. 25.

Optimal Substructure Of Shortest Path

- Subpaths of a shortest path are shortest paths!
  - Given $G = (V, E)$ with $w : E \to \mathbb{R}$,
  - Let $p = \langle v_0, v_1, \dots, v_k \rangle$ be a shortest path from v_0 to v_k
  - And for any i and j such that $0 \le i \le j \le k$, let $p_{ij} = \langle v_i, v_{i+1}, \dots, v_j \rangle$ the sub-path of p from v_i to v_j.
  - Then, p_{ij} is a shortest path from v_i to v_j.
- Proof: Usual cut-and-paste technique with contradiction applies here as well.

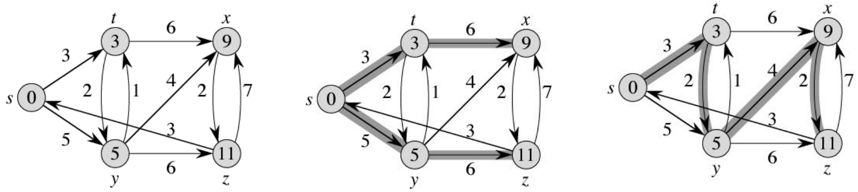
Negative-Weight Edges And Cycles

- Negative-weight edges are OK as long as there's no negative-weight cycle (CLRS Fig. 24.1).
- A shortest path can't have a cycle.
  - If there's a reachable negative-weight cycle, no shortest path can be defined.
  - If there are positive-weight cycles in a shortest path, it can't be a shortest path.
  - If there are 0-weight cycles in a shortest path, we can always remove them for same shortest-path weight.



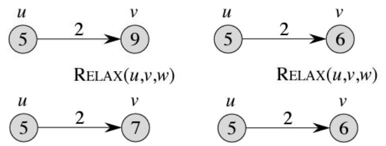
Shortest-Paths Tree

- Maintain a predecessor $v.\pi$ for each vertex v as in MST (Minimum Spanning Tree) algorithms.
- Predecessor subgraph induced by the π values form a ***shortest-paths tree*** rooted at the source vertex (CLRS Fig. 24.2).



Relaxation

- Key technique for shortest-path finding algorithms.
- Maintain a ***shortest-path distance estimate*** $v.d$ for each vertex v.
- Initialize them properly: 0 for the source vertex, ∞ for all other vertices.
- For any edge $(u, v) \in E$, we can reduce $v.d$ to $u.d + w(u, v)$ if currently $v.d > u.d + w(u, v)$.
  - In that case, v's shortest-path distance estimate can be reduced by going to u first and then taking the edge (u, v).
  - $v.\pi$ should be updated as well, because the new $v.d$ was obtained by taking (u, v).




```
RELAX(u, v, w)
if v.d > u.d + w(u, v)
    v.d = u.d + w(u, v)
    v.π = u
```

Properties of Shortest Paths and Relaxation

- Triangle inequality (CLRS Lemma 24.10)
  - For any edge $(u, v) \in E$, we have $\delta(s, v) \leq \delta(s, u) + w(u, v)$.
- Upper-bound property (Lemma 24.11)
  - Always $v.d \geq \delta(s, v)$ for all vertices $v \in V$.
  - Once $v.d$ reaches $\delta(s, v)$, it never changes.
- No-path property (Corollary 24.12)
  - $v.d = \delta(s, v) = \infty$ always if there's no path from s to v.
- Convergence property (Lemma 24.14)
  - 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 time afterward.

Properties of Shortest Paths and Relaxation (Continued)

- 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 order $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$,
  - Then $v_k.d = \delta(s, v_k)$.
  - This is true regardless of any other relaxation steps that may occur, even if they are intermixed with relaxations of the edges of p.
- Predecessor-subgraph property (Lemma 24.17)
  - The predecessor subgraph is a shortest-paths tree rooted at s, once $v.d = \delta(s, v)$ for all $v \in V$.

question on what graph algorithm should apply: bellman-ford or dijkstra?

compare these two algorithms? which one is faster? what does it suit for?

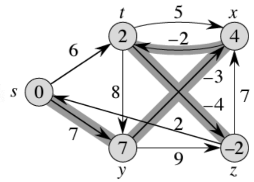
Bellman-Ford Algorithm

Shortest-Paths Finding Algorithm For General Cases (Negative Weight Edges Allowed)

Bellman-Ford Algorithm: Shortest-Paths Even With Negative Weight Edges

- Recall path-relaxation property: It tells us that we will get a shortest path if we relax edges in the shortest path in that order.
- Can be achieved surprisingly simply
  - Relax each and every edge in one pass, repeat this pass $|V| - 1$ times
- Lemma 24.2 proves the correctness of this idea for reachable vertices
  - Using path-relaxation property
- Non-reachable vertices will have $v.d = \infty$ (no-path property)
- Theorem 24.4 proves cases of negative-weight cycles (return false)
  - Proof by contradiction

Bellman-Ford Code And Example (CLRS Fig. 24.4)



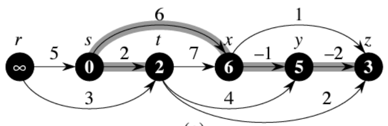

```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| − 1          v times
3      for each edge (u, v) ∈ G.E   e times
4          RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6      if v.d > u.d + w(u, v)
7          return FALSE
8  return TRUE
```

Time complexity: $\Theta(V)$ (line 1) + $\Theta(VE)$ (lines 2-4) + $O(E)$ (lines 5-7) = $\Theta(VE)$

Special Cases: Directed-Acyclic Graphs

- There are no cycles.
- All edges are in the forwarding direction.
- Therefore, vertices can be scanned from left to right, relaxing outgoing edges.
- At the time of starting a vertex (to relax all its outgoing edges), the vertex's shortest distance estimate is indeed the shortest distance.

DAG Shortest Paths Algorithm And Example (CLRS Fig. 24.5)



(g)

```
DAG-SHORTEST-PATHS(G, w, s)
1  topologically sort the vertices of G
2  INITIALIZE-SINGLE-SOURCE(G, s)
3  for each vertex u, taken in topologically sorted order
4      for each vertex v ∈ G.Adj[u]
5          RELAX(u, v, w)
```

Time complexity:
$\Theta(V+E)$ (line 1)
$+\Theta(V)$ (line 2)
$+\Theta(V+E)$ (lines 3-5, amortized)
$=\Theta(V+E)$

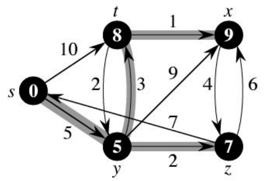
Dijkstra's Algorithm

Finding Shortest Paths On Graphs With Nonnegative Edge Weights

Shortest Paths On Graphs With Nonnegative Edge Weights

- Find a vertex with the shortest-path distance, mark it as visited, and repeat this $|V|$ times.
  - We can certainly start from s (source vertex) for its shortest-path distance 0.
- When we mark such a vertex, shortest-path distance estimates of all its adjacent vertices may be reduced (relaxed).
  - That is, relax all outgoing edges of that marked vertex.
- The next vertex to mark for its shortest-path distance is the vertex with the smallest shortest-path distance estimate, among all unmarked vertices.
  - Obvious for the next (second) marked vertex after s.
  - Need to prove in general (CLRS Theorem 24.6)

Dijkstra's Algorithm Example And Code

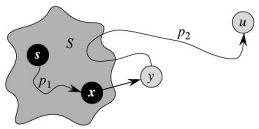


DIJKSTRA(G, w, s)

```
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S = ∅
3  Q = G.V
4  while Q ≠ ∅
5      u = EXTRACT-MIN(Q)
6      S = S ∪ {u}
7      for each vertex v ∈ G.Adj[u]
8          RELAX(u, v, w)
```

- Time complexity analysis:
  - Line 1, 3: $\Theta(V)$
  - While loop repeats $|V|$ times
  - Line 5: $O(V)$ if using array, $O(\lg V)$ if using heap
  - For loop repeats $|E|$ times (amortized)
  - Line 8: $O(1)$ if using array, $O(\lg V)$ if using heap (DECREASE-KEY)
- $\therefore O(V^2 + E) = O(V^2)$ if using array
- $O((V + E)\lg V)$ if using heap
  - $O(E \lg V)$ if connected V logV + E logV = E log V
- Note $O(E \lg V) < O(V^2)$ iff $E = o(\frac{V^2}{\lg V})$
  - Don't use heap for a dense graph!

Formal Proof of Dijkstra's Algorithm



- When u (= EXTRACT-MIN(Q)) is added to S, $u.d = \delta(u)$!
  - Obvious for first two such vertices.
- General proof: CLRS Theorem 24.6, by contradiction
  - Suppose $u.d \neq \delta(s, u)$ at the time of adding u to S, then derive a contradiction.
  - There must exist a shortest path $s \to \cdots \to x \to y \to \cdots \to u$ from s to u where $s \to \cdots \to x$ is entirely in S (marked/solved vertices), but y is not in S.
  - We claim that $y.d = \delta(s, y)$ at the time (because $x.d = \delta(s, x)$ by induction hypothesis and the convergence property on a shortest path).
  - Then we see $y.d = \delta(s, y) \leq \delta(s, u) \leq u.d$ (sub-path & upper-bound property)
  - And also see $u.d \leq y.d$ (because u = EXTRACT-MIN(Q)).
  - This gives a desired contradiction: $u.d = \delta(s, u)$

Intuition

- Pause the video after the animation is played, visit the web page below and read through.
- A good animation and intuitive proof discussions from https://www.quora.com/What-is-the-simplest-intuitive-proof-of-Dijkstra%E2%80%99s-shortest-path-algorithm