CS 315 - Day 22, Dykstra's Algorithm

What if you wanted to travel from Juniata College's home in Huntingdon, PA, to Huntingdon College, in Montgomery, AL. How does your GPS determine the fastest route?

Your GPS contains information about the entire road network of the US, including the road distance between each pair of adjacent intersections. How does it find the shortest route?

The brute force way would be to determine all the possible routes and their associated distances, and select the shortest.

But this would be so inefficient, why even consider driving to Chicago first, and then to Montgomery, AL?

We will consider the shortest-path problem.

Over the years, the algorithms we discuss in this course have been some of the most common programming challenges Juniata students have been given during job interviews. Asymptotic limits on sorts, as well as Dijkstra's algorithm seem to be the most frequently ocurring. That's one of the reason's we will explore Dijkstra's algorithm today, besides the fact that it's very useful.

It builds on the Breadth-First-Search (BFS) we explored last class, generalizing it to weighted graphs, those graphs where the edges have an associated weight.

Dijkstra's algorithm replaces the FIFO queue used in the BFS with a minimum-priority queue of vertices, with their distances ***d*** as their key values. It requires that the weights of the edges are all non-negative.

How to find the shortest route between two points on a map.

Input:

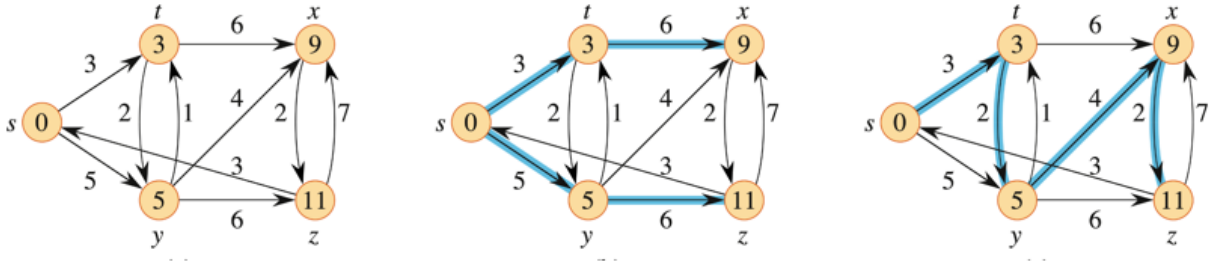
- Directed graph $G = (V, E)$
- Weight function $w : E \to \mathbb{R}$

Shortest-path weight u to v:

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

Shortest path u to v is any path p such that $w(p) = \delta(u, v)$.

shortest paths from s



This example shows that a shortest path might not be unique.

It also shows that when we look at shortest paths from one vertex to all other vertices, the shortest paths are organized as a tree.

We can disregard cycles, we can go directly from ***t*** to ***x***, and don't have to add in an extra cycle, such as: ***t → x →z → y → t***, we can get a shorter path merely by eliminating the cycles.

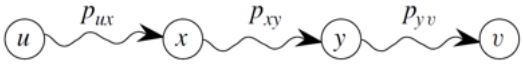
Also, we will focus on having positive weight edges, although the text covers algorithms with negative weight edges.

By the way, GPS' probably don't use just distance as the weight, since many seem to account for slow traffic etc.

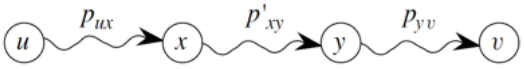
The algorithms all build on the idea that:

Any subpath of a shortest path is a shortest path.

Proof Cut-and-paste.



Now suppose there exists a shorter path $x \overset{p'_{xy}}{\rightsquigarrow} y$.
Then $w(p'_{xy}) < w(p_{xy})$.
Construct p':



Contradicts the assumption that p is a shortest path.

As we hinted at above, the output of a ***single-source*** – ***shortest-path*** algorithm is a tree:

For each vertex $v \in V$:

- $v.d = \delta(s, v)$.
  - Initially, $v.d = \infty$.
  - Reduces as algorithms progress. But always maintain $v.d \geq \delta(s, v)$.
  - Call $v.d$ a ***shortest-path estimate***.
- $v.\pi$ = predecessor of v on a shortest path from s.
  - If no predecessor, $v.\pi = \text{NIL}$.
  - π induces a tree—***shortest-path tree***.

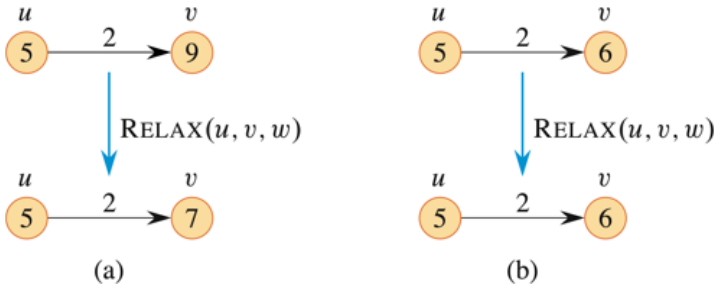
All the shortest-paths algorithms start with INITIALIZE-SINGLE-SOURCE.

```
INITIALIZE-SINGLE-SOURCE(G, s)
1  for each vertex v ∈ G.V
2      v.d = ∞
3      v.π = NIL
4  s.d = 0
```

A key concept in the algorithms is ***relaxing and edge*** between a pair of vertices (u, v):

Can the shortest-path estimate for v be improved by going through u and taking (u, v)?

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



DIJKSTRA'S ALGORITHM

No negative-weight *edges*.

Essentially a weighted version of breadth-first search.

- Instead of a FIFO queue, uses a priority queue.
- Keys are shortest-path weights $(v.d)$.
- Can think of waves, like BFS.
- A wave emanates from the source.
- The first time that a wave arrives at a vertex, a new wave emanates from that vertex.

Have two sets of vertices:

- S = vertices whose final shortest-path weights are determined,
- Q = priority queue = $V - S$.

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S = ∅
3  Q = ∅
4  for each vertex u ∈ G.V
5      INSERT(Q, u)
6  while Q ≠ ∅
7      u = EXTRACT-MIN(Q)
8      S = S ∪ {u}
9      for each vertex v in G.Adj[u]
10         RELAX(u, v, w)
11         if the call of RELAX decreased v.d
12             DECREASE-KEY(Q, v, v.d)
```

Comparing and contrasting to BFS, where the key values of a vertex were merely the number of edges from the source, now the key values are the minimum distance. This is why we have to replace the simple FIFO queue with a min-priority queue.

Let's consider an example:

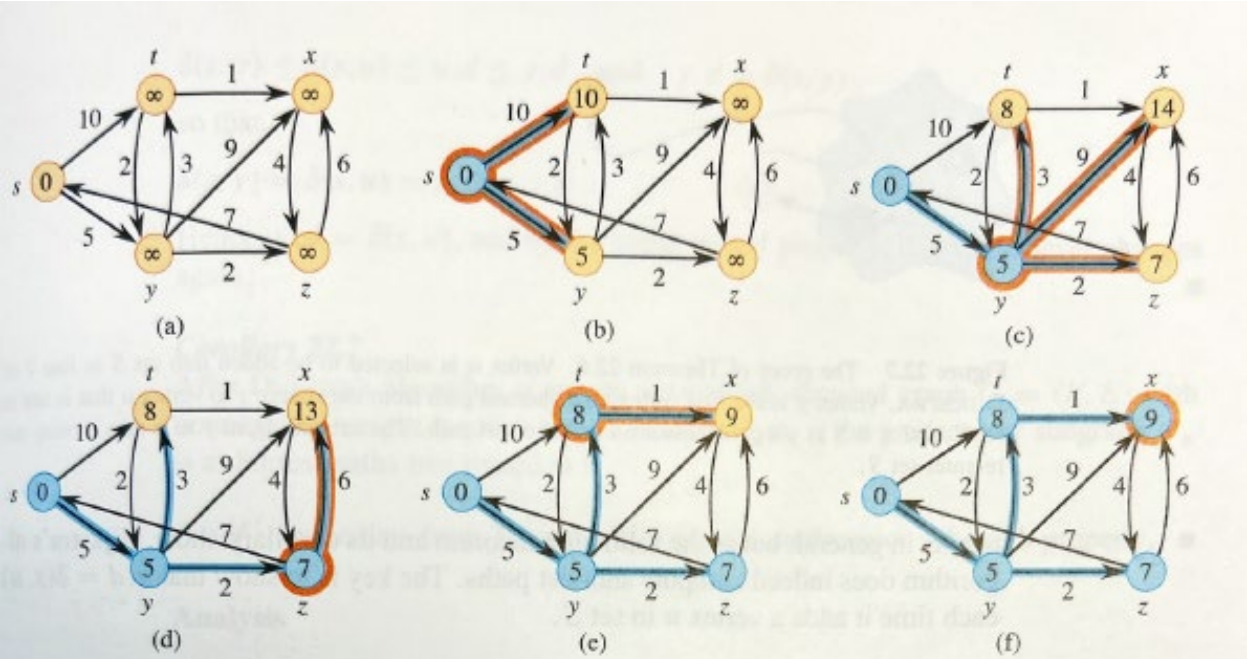


Figure 22.6 The execution of Dijkstra's algorithm. The source s is the leftmost vertex. The shortest-path estimates appear within the vertices, and blue edges indicate predecessor values. Blue vertices belong to the set S, and tan vertices are in the min-priority queue $Q = V - S$. **(a)** The situation just before the first iteration of the **while** loop of lines 6–12. **(b)–(f)** The situation after each successive iteration of the **while** loop. In each part, the vertex highlighted in orange was chosen as vertex u in line 7, and each edge highlighted in orange caused a d value and a predecessor to change when the edge was relaxed. The d values and predecessors shown in part (f) are the final values.

Dijkstra's algorithm relaxes edges as shown in Figure 22.6. Line 1 initializes the d and π values in the usual way, and line 2 initializes the set S to the empty set. The algorithm maintains the invariant that $Q = V - S$ at the start of each iteration of the **while** loop of lines 6–12. Lines 3–5 initialize the min priority queue Q to contain all the vertices in V. Since $S = \emptyset$ at that time, the invariant is true upon first reaching line 6. Each time through the **while** loop of lines 6–12, line 7 extracts a vertex u from $Q = V - S$ and line 8 adds it to set S, thereby maintaining the invariant. (The first time through this loop, $u = s$.) Vertex u, therefore, has the smallest shortest-path estimate of any vertex in $V - S$. Then, lines 9–12 relax each edge (u, v) leaving u, thus updating the estimate $v.d$ and the predecessor $v.\pi$ if the shortest path to v found so far improves by going through u. Whenever a relaxation step changes the d and π values, the call to DECREASE-KEY in line 12 updates the min-priority queue. The algorithm never inserts vertices into Q after the **for** loop of lines 4–5, and each vertex is extracted from Q and added to S exactly once, so that the **while** loop of lines 6–12 iterates exactly $|V|$ times.

Because Dijkstra's algorithm always chooses the "lightest" or "closest" vertex in $V - S$ to add to set S, you can think of it as using a greedy strategy. Chapter 15 explains greedy strategies in detail, but you need not have read that chapter to understand Dijkstra's algorithm. Greedy strategies do not always yield optimal

Run `Dijkstra.ipynb`, which includes the pseudocode above, and confirm that the output agrees with the in-class example.

In-class w/***assigned groups of four (4)***, write 0. and 1. on board as ground rule(s):

0. No devices.
1. Intro: Name, POE, Icebreaker.
2. Pick someone on the group who will report out.
3. Create a non-trivial weighted and directed graph with your team.
4. Work through Dijkstra's algorithm by hand on your graph.
5. In a new execution group, run your graph and confirm that the output agrees with your group work.