Introduction to Algorithms and Data Structures

Lecture 16: Dijkstra's Algorithm (for shortest paths)

Mary Cryan

School of Informatics
University of Edinburgh

Welcome back!

Directed and Undirected Graphs

We return to the world of graphs and directed graphs.

- A *graph* is a mathematical structure consisting of a set of *vertices* and a set of *edges* connecting the vertices.

  Formally: $G = (V, E)$, where V is a set and $E \subseteq V \times V$.

- $G = (V, E)$ *undirected* if for all $v, w \in V$:

$$(v, w) \in E \iff (w, v) \in E.$$

  Otherwise *directed*.

  Directed $\sim$ *arrows* (one-way)
  Undirected $\sim$ *lines* (two-way)

Road Networks

The weighted case is a very natural graph model - eg, road network where vertices represent intersections, edges represent road segments, and the weight of an edge is the distance of that road segment.

Shortest paths in graphs

In this lecture we will consider *weighted* graphs (and digraphs) $G = (V, E)$ where there is a weight function $w : E \to \mathbb{R}$ defining weights for all arcs/edges.

We are interested in evaluating the cost of shortest paths (from specific node u to specific node v) in the given weighted graph.

> *We will focus on single-source shortest paths, where we want to find the minimum path from node s to node v, for every v.*

Input: Graph $G = (V, E), w : E \to \mathbb{R}^+$ a weighted graph/digraph (no negative weights), $s \in V$ a specific source vertex.

Single-source shortest paths

unweighted graphs and digraphs

We can use breadth-first search to explore a graph $G = (V, E)$ from a specific vertex $s \in V$. $\Theta(|V| + |E|)$ running-time.

(in the unweighted case, the shortest path is the one with fewest edges)

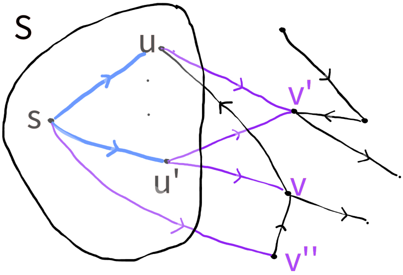
weighted graphs and digraphs

- Dijkstra's algorithm, will compute the single source shortest paths (and their values) for any graph or directed graph without negative weights.
- Dijkstra's Algorithm is a greedy algorithm.
- Makes use of a Priority Queue (as introduced at the end of L11 in s1).
- We will see that (with the use of a (Min) Heap to deliver the Priority Queue), Dijkstra can achieve running-time $O((|V| + |E|)\lg(|V|))$, or $O((m + n)\lg(n))$.

Dijkstra's Algorithm

A Greedy algorithm which grows the set S of "shortest path solved" vertices.

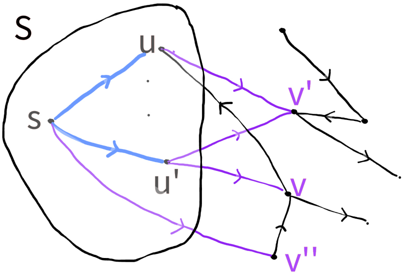
- S has some vertices where shortest path from S is known (blue edges).
- S has some outgoing edges (from S to outside S) (fringe edges in purple) (fringe vertices are those accessible by a fringe edge)



At each iteration, Dijkstra's Algorithm will add the fringe vertex $v \in V \setminus S$ with the shortest candidate path into S.

Dijkstra - how to select the next fringe vertex?

- We have some vertices already in S ($\{s, u, u'\}$ in picture).
  We already know the optimum shortest path from s ($d[v]$) for every $v \in S$.
- We need to consider the fringe vertices (v, v', v'' in picture) and add the one with shortest candidate path into S. For our picture ...
  v's candidate path is $d[u'] + w(u', v)$
  v' has two candidate paths: $d[u] + w(u, v')$ and $d[u'] + w(u', v')$
  v'' has candidate path $d[s] + w(s, v'') = w(s, v'')$ (as $d[s]$ is 0)



Dijkstra - rules for selecting next fringe vertex

Arrays: We use arrays d and π of length $n = |V|$ each:

$d[v]$ to (eventually) hold shortest-path distance $d_G(s, v)$ from s to v
$\pi[v]$ to (eventually) be v's predecessor along that shortest path.

Initialisation:

$$d[v] \leftarrow \begin{cases} 0 & v = s \\ \infty & v \in V \setminus \{s\}. \end{cases}$$

We initialise predecessor array π by $\pi[v] \leftarrow \mathrm{NIL}$ for every $v \in V$.

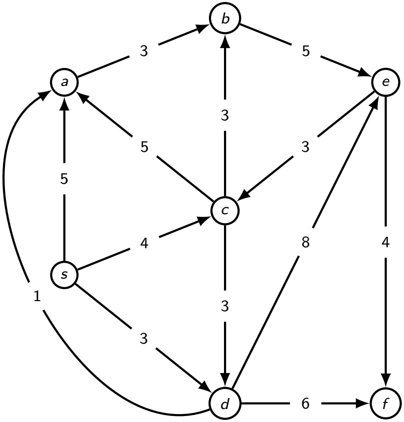
Induction step: (while S still has fringe edges to $V \setminus S$), then for every fringe vertex $v \in V \setminus S$, compute v's (current) shortest candidate path/predecessor

$$\begin{aligned} d[v] &\leftarrow \min_{u \in S} \{d[u] + w(u, v)\} \\ \pi[v] &\leftarrow \arg\min_{u \in S} \{d[u] + w(u, v)\}. \end{aligned}$$

- Let $v^* \in V \setminus S$ be the fringe vertex with $\min_{\text{over all fringe vertices}} d[v]$.
- Update $S \leftarrow S \cup \{v^*\}$, then $d[v^*]$ and $\pi[v^*]$ become fixed from now on.

Terminate when S no longer has fringe edges.

Worked example - initialisation



To start we have $S = \{s\}$, and fringe vertices are $\{a, c, d\}$. Arrays are:

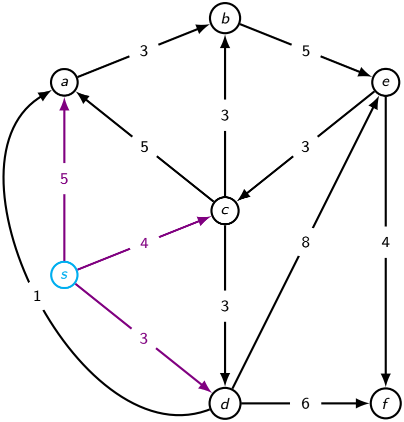
d :

s	a	b	c	d	e	f
0	∞	∞	∞	∞	∞	∞

π :

s	a	b	c	d	e	f
-	-	-	-	-	-	-

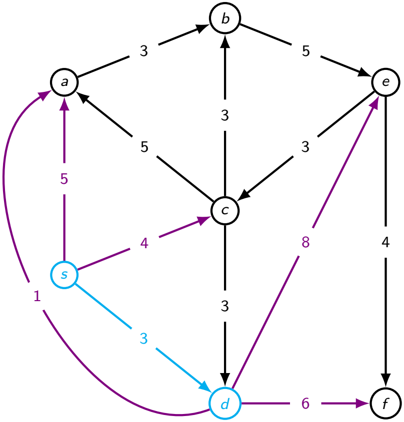
Worked example - adding 2nd vertex



Fringe vertices are $\{a, c, d\}$

- a has candidate path value $d[s] + w(s, a) = 0 + 5 = 5$
- c has candidate path value $d[s] + w(s, c) = 0 + 4 = 4$
- d has candidate path value $d[s] + w(s, d) = 0 + 3 = 3 \ldots \Rightarrow S \leftarrow S \cup \{d\}$

Worked example - $S = \{s, d\}$



$S = \{s, d\}$, fringe vertices are now $\{a, c, e, f\}$. Arrays are:

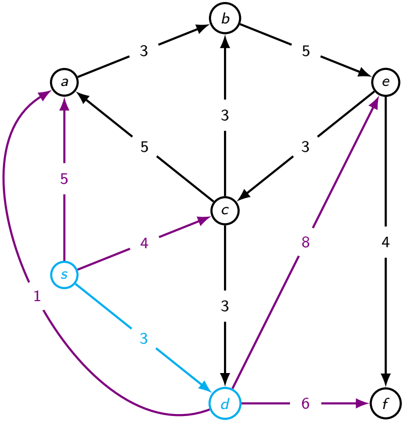
d :

0	∞	∞	∞	3	∞	∞
s	a	b	c	d	e	f

π :

-	-	-	-	s	-	-
s	a	b	c	d	e	f

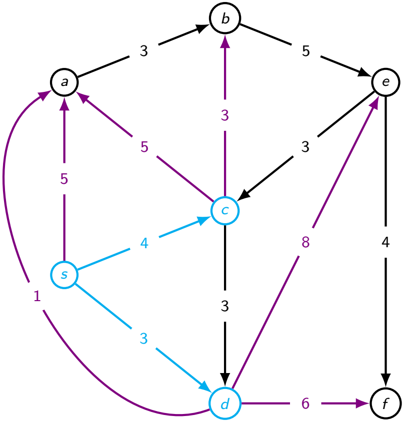
Worked example - adding 3rd vertex



$S = \{s, d\}$, fringe vertices are now $\{a, c, e, f\}$.

- a has extra candidate path with $\pi[a] = d$, better value $d[d] + w(d, a) = 4$.
- c's existing candidate path, still available, has value 4.
- New fringe vertices e, f have paths ($\pi[\cdot] = d$) with values 11, 9 resp.
  $\Rightarrow$ EITHER $S \leftarrow S \cup \{c\}$ OR $S \leftarrow S \cup \{a\}$

Worked example - $S = \{s, d, c\}$



$S = \{s, d, c\}$. Arrays are:

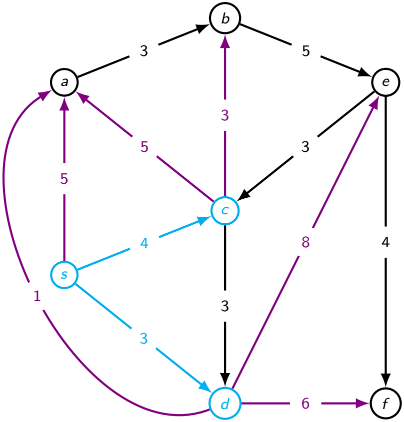
d :

0	∞	∞	4	3	∞	∞
s	a	b	c	d	e	f

π :

-	-	-	s	s	-	-
s	a	b	c	d	e	f

Worked example - adding 4th vertex

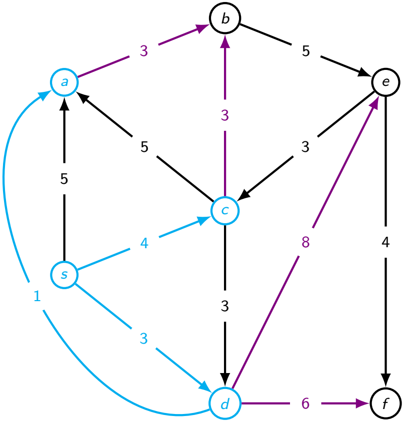


$S = \{s, d, c\}$, fringe vertices are now $\{a, b, e, f\}$.

- We know a has candidate path value 4 (via d), e value 11, f's value 9.
- New fringe vertex b has candidate path value $d[c] + w(c, b) = 4 + 3 = 7$
- a has an extra candidate path with value $d[c] + w(c, a) = 4 + 5 = 9 > 5$

$\ldots \Rightarrow S \leftarrow S \cup \{a\}$ with $\pi[a] \leftarrow d$.

Worked example - $S = \{s, d, c, a\}$



$S = \{s, d, c, a\}$. Arrays are:

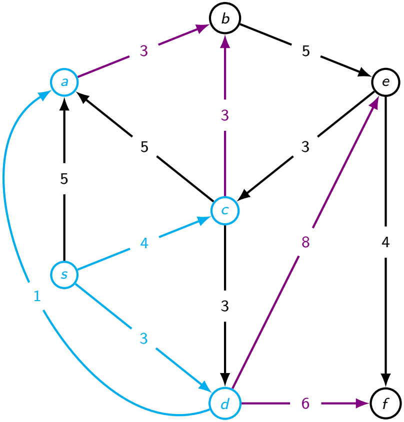
d :

0	4	∞	4	3	∞	∞
s	a	b	c	d	e	f

π :

-	d	-	s	s	-	-
s	a	b	c	d	e	f

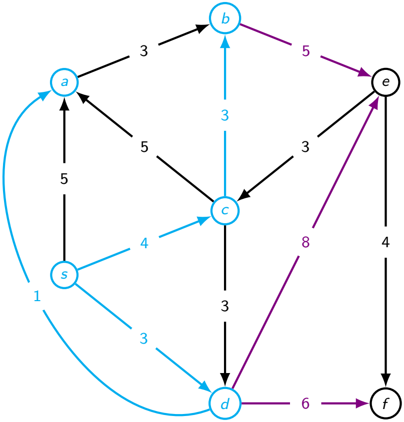
Worked example - adding 5th vertex



$S = \{s, d, c, a\}$, fringe vertices are now $\{b, e, f\}$.

- We know e and f have candidate paths $(\pi[\cdot] = d)$ with values 11, 9.
- Fringe vertex b has a new candidate path value $d[a] + w(a, b) = 4 + 3 = 7$, same value as existing path via c.
  $\Rightarrow S \leftarrow S \cup \{b\}$ with $\pi[b] \leftarrow c/a$.

Worked example - $S = \{s, d, c, a, b\}$



$S = \{s, d, c, a, b\}$. Arrays are:

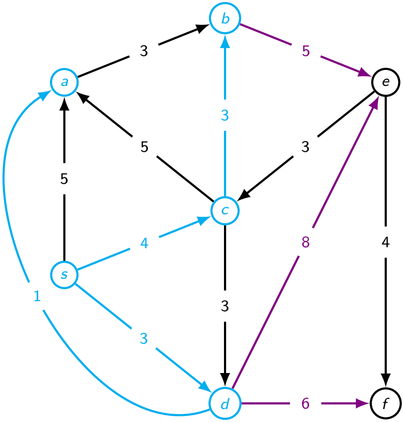
d :

0	4	7	4	3	∞	∞
s	a	b	c	d	e	f

π :

-	d	c	s	s	-	-
s	a	b	c	d	e	f

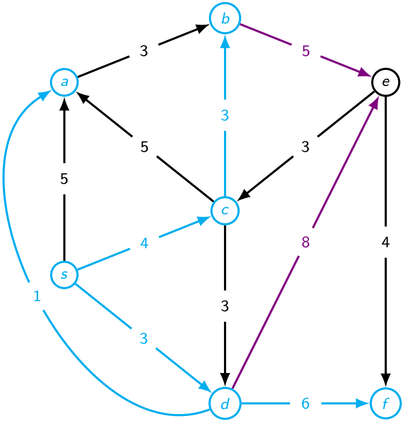
Worked example - adding 6th vertex



$S = \{s, d, c, a, b\}$, fringe vertices are now $\{e, f\}$.

- f has existing candidate path ($\pi[f] = d$) with value 9.
- e has a new candidate path ($\pi[e] = b$) with value $d[b] + w(b, e) = 7 + 5 = 12$, worse than existing candidate path via d. $\Rightarrow S \leftarrow S \cup \{f\}$ with $\pi[f] \leftarrow d$.

Worked example - $S = \{s, d, c, a, b, f\}$



$S = \{s, d, c, a, b, f\}$. Arrays are:

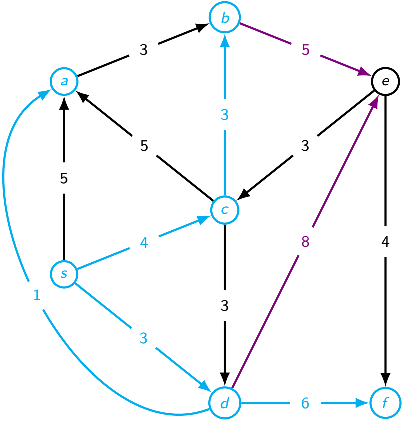
d :

0	4	7	4	3	∞	9
s	a	b	c	d	e	f

π :

-	d	c	s	s	-	d
s	a	b	c	d	e	f

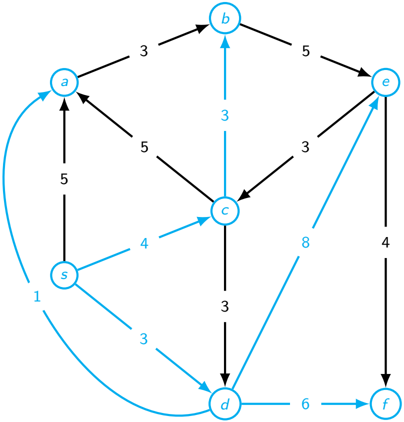
Worked example - adding last vertex



$S = \{s, d, c, a, b, f\}$, fringe vertex set is just $\{e\}$.

- e's best candidate path is with $\pi[e] = d$ with value 11 $\Rightarrow S \leftarrow S \cup \{e\}$ with $\pi[e] \leftarrow d$.

Example - final "shortest path tree" and arrays



$S = \{s, d, c, a, b, f, e\}$, no fringe edges/vertices.

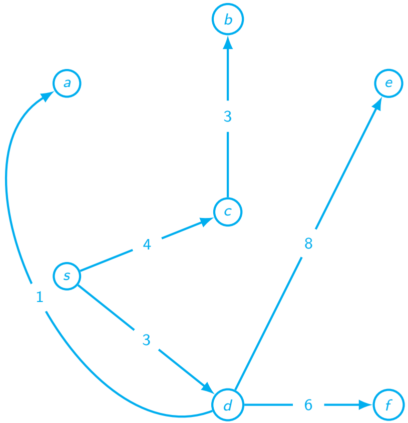
d :

s	a	b	c	d	e	f
0	4	7	4	3	11	9

π :

s	a	b	c	d	e	f
-	d	c	s	s	d	d

final “shortest path tree”



Observe that the collection of edges contributing to all shortest paths forms a “shortest path tree” (a directed arborescence out of the source vertex s)

Simple Implementation of Dijkstra

Algorithm InitializeSingleSource(G, s)

for each vertex $v \in V[G]$
    do $d[v] \leftarrow \infty$
        $\pi[v] \leftarrow \text{NIL}$

Algorithm DijkstraSimple(G, s)

InitializeSingleSource(G, s)
$d[s] \leftarrow 0$, $S \leftarrow \{s\}$
while $V[G] \setminus S \neq \emptyset$ **and** there are fringe edges
    $min_u \leftarrow s, min_d \leftarrow \infty, \min_v \leftarrow \text{NIL}$
    do for $u \in S, v \in V[G] \setminus S, (u, v) \in E(G)$
        if $d[u] + w(u, v) < min_d$
            $min_d \leftarrow d[u] + w(u, v)$, $min_u \leftarrow u$, $min_v \leftarrow v$
    $S \leftarrow S \cup \{min_v\}, d[min_v] \leftarrow min_d, \pi[min_v] \leftarrow min_u$
return d, π

Recovering the shortest paths

(in a graph/digraph with non-negative weights)
In practice, we will want the short paths themselves, not just the values.

Some facts that help us:

- No shortest path from s to any v can contain a cycle.
  why?: If a path p contains a cycle, cycle's weight is ≥ 0, we could delete it to get another $s \rightarrow v$ with fewer edges, and distance no greater.
- Every shortest path has at most $n - 1$ edges.
  why?: no cycles, so can visit any node at most once.
- If $s = v_0, v_1, \ldots, v_k$ is a shortest path to v_k, then every prefix $s = v_0, v_1, \ldots, v_i$ is a shortest path to v_i.

  why?: If we had a shorter path for one of the v_i, we could replace section $s = v_0, v_1, \ldots, v_i$ to get a shorter path for v_k too.

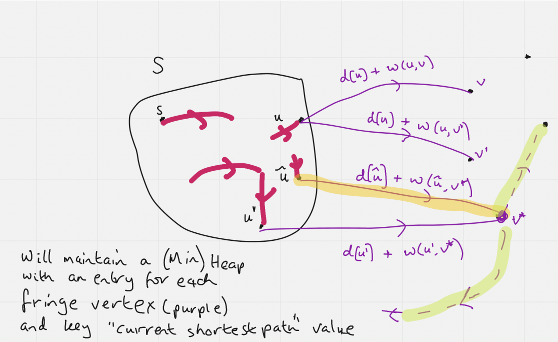
The third point allows us to use the π array to recursively build the short path for any $v \in S$ (lookup $\pi[v]$ to get last edge $(\pi[v], v)$, lookup $\pi[\pi[v]]$, ...)

A more efficient implementation

- Dijkstra is "all about" the ranking/management of the fringe edges/vertices.
- The DijkstraSimple implementation has in-efficiency, in that it reconsiders/recalculates existing fringe edges at later steps.
- Improvement: Eliminate "re-calculation" for fringe edges:
  - Let $d[v], \pi[v]$ store the "shortest so far" $d[\cdot] + w(\cdot, v)$ for every current fringe vertex v.
  - After a new change ($S \leftarrow S \cup \{v^*\}$), limit calculation to the new fringe edges: consider the (v^*, w) edges for $w \in V[G] \setminus (S \cup \{v^*\})$ and possibly update the $d[w], \pi[w]$ entries.

    This can reduce the work for adding 1 vertex to become $O(n)$ (in fact $O(out(v^*))$) rather than potentially $\Omega(m)$ (as with DijkstraSimple). However, we need to be able to store the fringe vertices in a Data Structure which will allow us to identify/access the optimum fringe vertex quickly.

Dijkstra's Algorithm using a (Min) Heap



- G as Adjacency list - can visit "outgoing edges from v" in $O(out(v))$.
- Maintain a (Min) Heap priority queue Q of current fringe vertices, with their current shortest path value (so far) as key. (we will need to be able to *update/reduce* keys, after a successful Relax operation).
- Q.extractMin() $\Leftrightarrow$ "add the best fringe vertex v" to S.

Implementation using (Min) Heap

Algorithm InitializeSingleSource(G, s)

```
for each vertex v ∈ V[G]
    do d[v] ← ∞
         π[v] ← NIL
```

Algorithm Relax$(G, (u, v))$

```
if d[v] = ∞
    then d[v] ← d[u] + w(u, v)
         π[v] ← u
         Q.insertItem(d[v], v)
if(d[v] > d[u] + w(u, v))
    then d[v] ← d[u] + w(u, v)
         π[v] ← u
         Q.reduceKey(d[v], v)
```

Implementation using (Min) Heap

Edsger Dijkstra

Algorithm Dijkstra(G, s)

InitializeSingleSource$(G,\ s)$
Q.insertItem$(0, s)$
$d[s] \leftarrow 0$
while $\neg$(Q.isEmpty())
  do $(d^*, u) \leftarrow$ Q.extractMin()
    for $x \in Out(u)$
      Relax$(G, (u, x))$

(Min) Heaps

In Lecture 11 we saw how we can use a Heap to implement a Priority Queue with n items, so operations have the following worst-case running-times:

Q.isEmpty()	$\Theta(1)$
Q.minElement()	$\Theta(1)$
Q.extractMin()	$O(\lg(n))$
Q.insertItem(d,v)	$O(\lg(n))$
Q.reduceKey(d',v)	$O(\lg(n))$

Strictly speaking, we demonstrated this for a Max Heap - however, by exchanging $>$ and $<$ we can transform a Max Heap implementation into a Min Heap structure, same running-times.

updates: We can also add the operation Q.reduceKey(d', v) (to replace v's current key by a smaller d') to operate in $O(\lg(n))$ worst-case time.

- Use the "bubble up" of insertItem, but may start higher than a leaf.
- (we assume we have an index supporting jumps to v's cell of the Heap)

Running-time analysis for (Heap) Dijkstra

- InitializeSingleSource takes $O(n)$ time at most.
- lines 2.-3. take $O(1)$.
- *Take an "aggregated" approach to bounding run-time of the* **while**
- A vertex v can be *added* to the Heap only once (need $d[v] = \infty$ in Relax) and hence, only removed once
  $\Rightarrow O(n. \lg(n))$ covers *all* the Q.extractMin() and Q.insertItem(d, v) calls.
- *Apart* from the insertItem calls, a call to Relax takes
  $O(1) + T_{\text{reduceKey}}(n) = O(1) + O(\lg(n))$ time.
- We might call Relax at most *twice* for every edge $e \in E$ ... as we only call Relax$(G, (u, v))$ immediately after an endpoint has joined S.
  Hence total for *all* Relax calls is $O(m + m \cdot \lg(n))$ time.
- Other work done by the **while** is at most $O(n)$.

$$O((n + m) \lg(n)) \text{ time overall}$$

Reading

- CLRS, Ed 4: Sections 20.1 (graph rep.), Section 22.2 (shortest-path reps) and Sections 22.3 for Dijkstra. Some proofs in 22.5.
- "Algorithms Illuminated" by Roughgarden: Sections 9.1, 9.2, 9.4 and (for the faster Heap implementation) 10.4, 10.5.

Our Heap version of Dijkstra's Alg is most similar (but *slightly* different) to the [CLRS] presentation.