Dijkstra's Shortest Path Algorithm

DPV 4.4, CLRS 24.3

Revised, October 23, 2014

Outline of this Lecture

- Recalling the BFS solution of the shortest path problem for unweighted (di)graphs.

- The shortest path problem for weighted digraphs.

- Dijkstra's algorithm.
  Given for digraphs but easily modified to work on undirected graphs.

Recall: Shortest Path Problem for Graphs

Let $G = (V, E)$ be a (di)graph.

- The shortest path between two vertices is a path with the shortest **length** (least number of edges). Call this the **link-distance**.

- Breadth-first-search is an algorithm for finding shortest (link-distance) paths from a single source vertex to all other vertices.

- BFS processes vertices in increasing order of their distance from the root vertex.

- BFS has running time $O(|V| + |E|)$.

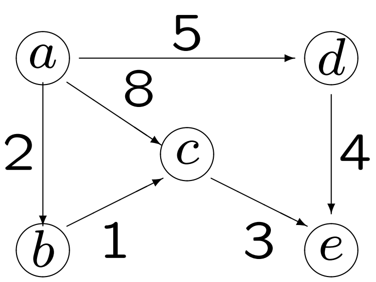
Shortest Path Problem for Weighted Graphs

Let $G = (V, E)$ be a weighted digraph, with weight function $w : E \mapsto \mathbf{R}$ mapping edges to real-valued weights. If $e = (u, v)$, we write $w(u, v)$ for $w(e)$.

- The **length** of a path $p = \langle v_0, v_1, ..., v_k \rangle$ is the sum of the weights of its constituent edges:

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

- The **distance** from u to v, denoted $\delta(u, v)$, is the length of the minimum length path if there is a path from u to v; and is ∞ otherwise.



length($\langle a, b, c, e \rangle$) $= 6$
distance from a to e is 6

Single-Source Shortest-Paths Problem

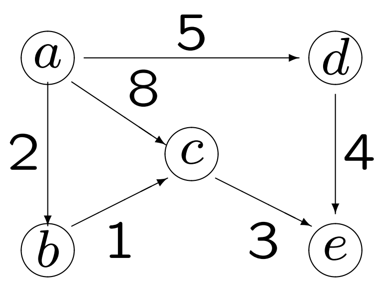
The Problem: Given a digraph with positive edge weights $G = (V, E)$
and a distinguished *source vertex*, $s \in V$,
determine the distance and a shortest path from the source vertex to every vertex in the digraph.

Question: How do you design an efficient algorithm for this problem?

Single-Source Shortest-Paths Problem

Important Observation: Any subpath of a shortest path must also be a shortest path. Why?

Example: In the following digraph, $\langle a, b, c, e\rangle$ is a shortest path. The subpath $\langle a, b, c\rangle$ is also a shortest path.



length($\langle a, b, c, e\rangle$) $= 6$
distance from a to e is 6

Observation Extending this idea we observe the existence of a *shortest path tree* in which distance from source to vertex v is length of shortest path from source to vertex in original tree.

Intuition behind Dijkstra's Algorithm

- Report the vertices in increasing order of their distance from the source vertex.

- Construct the shortest path tree edge by edge; at each step adding one new edge, corresponding to construction of shortest path to the current new vertex.

The Rough Idea of Dijkstra's Algorithm

- Maintain an *estimate* $d[v]$ of the length $\delta(s, v)$ of the shortest path for each vertex v.

- Always $d[v] \geq \delta(s, v)$ and $d[v]$ equals the length of a known path
($d[v] = \infty$ if we have no paths so far).

- Initially $d[s] = 0$ and all the other $d[v]$ values are set to ∞. The algorithm will then process the vertices one by one in some order.
The processed vertex's estimate will be validated as being real shortest distance, i.e. $d[v] = \delta(s, v)$.

  Here "processing a vertex u" means finding new paths and updating $d[v]$ for all $v \in Adj[u]$ if necessary. The process by which an estimate is updated is called **relaxation**.

  When all vertices have been processed,
$d[v] = \delta(s, v)$ for all v.

The Rough Idea of Dijkstra's Algorithm

Question 1: How does the algorithm find new paths and do the relaxation?

Question 2: In which order does the algorithm process the vertices one by one?

Answer to Question 1

- Finding new paths. When processing a vertex u, the algorithm will examine all vertices $v \in Adj[u]$. For each vertex $v \in Adj[u]$, a new path from s to v is found (path from s to u + new edge).

- Relaxation. If the new path from s to v is shorter than $d[v]$, then update $d[v]$ to the length of this new path.

Remark: Whenever we set $d[v]$ to a finite value, there exists a path of that length. Therefore $d[v] \geq \delta(s, v)$.

(Note: If $d[v] = \delta(s, v)$, then further relaxations cannot change its value.)

Implementing the Idea of Relaxation

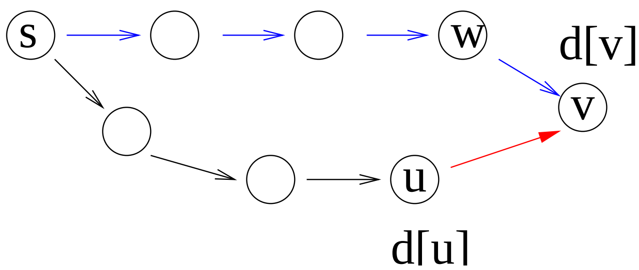
Consider an edge from a vertex u to v whose weight is $w(u,v)$. Suppose that we have already processed u so that we know $d[u] = \delta(s,u)$ and also computed a current estimate for $d[v]$. Then

- There is a (shortest) path from s to u with length $d[u]$.
- There is a path from s to v with length $d[v]$.

Combining this path from s to u with the edge (u,v), we obtain another path from s to v with length $d[u] + w(u,v)$.

If $d[u]+w(u,v) < d[v]$, then we replace the old path $\langle s,\ldots,w,v\rangle$ with the new shorter path $\langle s,\ldots,u,v\rangle$. Hence we update

- $d[v] = d[u] + w(u,v)$
- $pred[v] = u$ (originally, $pred[v] == w$).



The Algorithm for Relaxing an Edge

Relax(u,v)
$\{$
    if $(d[u] + w(u,v) < d[v])$
    $\{\quad d[v] = d[u] + w(u,v);$
        $pred[v] = u;$
    $\}$
$\}$

Remark: The predecessor pointer $pred[\,]$ is for determining the shortest paths.

Idea of Dijkstra's Algorithm: Repeated Relaxation

- Dijkstra's algorithm operates by maintaining a subset of vertices, $S \subseteq V$, for which we know the true distance, that is $d[v] = \delta(s, v)$.

- Initially $S = \emptyset$, the empty set, and we set $d[s] = 0$ and $d[v] = \infty$ for all others vertices v. One by one we select vertices from $V \setminus S$ to add to S.

- The set S can be implemented using an array of vertex colors. Initially all vertices are white, and we set $color[v] =$ black to indicate that $v \in S$.

The Selection in Dijkstra's Algorithm

Recall Question 2: What is the best order in which to process vertices, so that the estimates are guaranteed to converge to the true distances.
That is, how does the algorithm select which vertex among the vertices of $V \setminus S$ to process next?

Answer: We use a greedy algorithm. For each vertex in $u \in V \setminus S$, we have computed a distance estimate $d[u]$. The next vertex processed is always a vertex $u \in V \setminus S$ for which $d[u]$ is minimum, that is, we take the unprocessed vertex that is closest (by our estimate) to s.

Question: How do we implement this selection of vertices efficiently?

The Selection in Dijkstra's Algorithm

Question: How do we perform this selection efficiently?

Answer: We store the vertices of $V \setminus S$ in a *priority queue*, where the key value of each vertex v is $d[v]$.

[Note: if we implement the priority queue using a heap, we can perform the operations **Insert(), Extract_Min()**, and **Decrease_Key()**, each in $O(\log n)$ time.]

Review of Priority Queues

A **Priority Queue** is a data structure (can be implemented as a heap) which supports the following operations:

insert(u, key): Insert u with the key value key in Q.

u = extractMin(): Extract the item with the minimum key value in Q.

decreaseKey(u, new-key): Decrease u's key value to new-key.

Remark: *Priority Queues can be implemented such that each operation takes time $O(\log |Q|)$.*
In Class we only saw insert*,* extractMin *and* Delete*.* DecreaseKey *can either be implemented directly in $O(\log |Q|)$ time or could be implemented by a* delete *followed by an* insert*.*

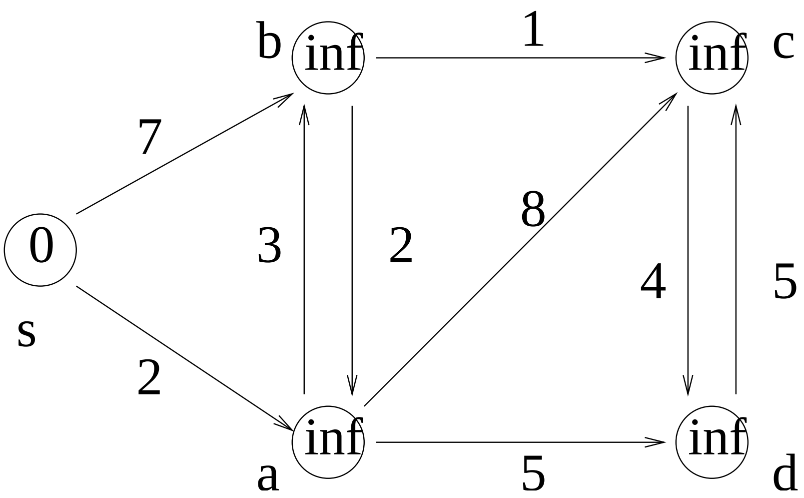
Description of Dijkstra's Algorithm

```
Dijkstra(G,w,s)
{                                              % Initialize
    for (each u ∈ V)
    {
        d[u] = ∞;
        color[u] =white;
    }
    d[s] = 0;
    pred[s] = NIL;
    Q = (queue with all vertices);

    while (Non-Empty(Q))                       % Process all vertices
    {
        u = Extract-Min(Q);                    % Find new vertex
        for (each v ∈ Adj[u])
            if (d[u] + w(u, v) < d[v])         % If estimate improves
            {
                d[v] = d[u] + w(u, v);           relax
                Decrease-Key(Q, v, d[v]);
                pred[v] = u;
            }
        color[u] =black;
    }
}
```

Dijkstra's Algorithm

Example:



Step 0: Initialization.

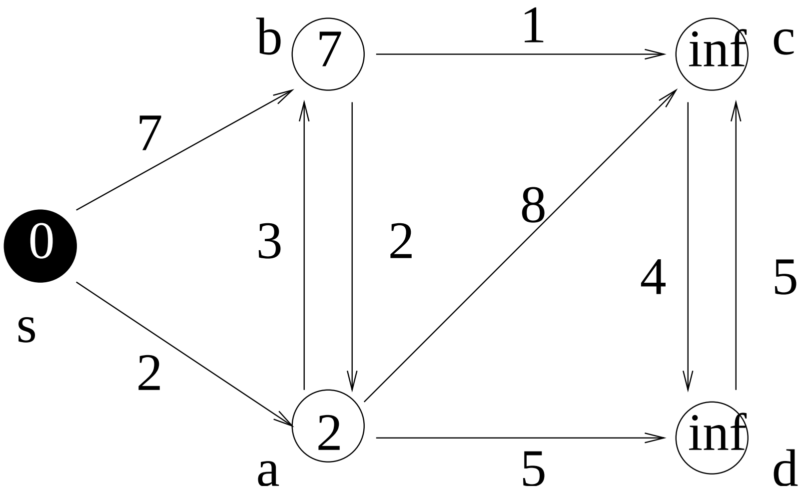
v	s	a	b	c	d
$d[v]$	0	∞	∞	∞	∞
$pred[v]$	nil	nil	nil	nil	nil
$color[v]$	W	W	W	W	W

Priority Queue:

v	s	a	b	c	d
$d[v]$	0	∞	∞	∞	∞

Dijkstra's Algorithm

Example:



Step 1: As $Adj[s] = \{a, b\}$, work on a and b and update information.

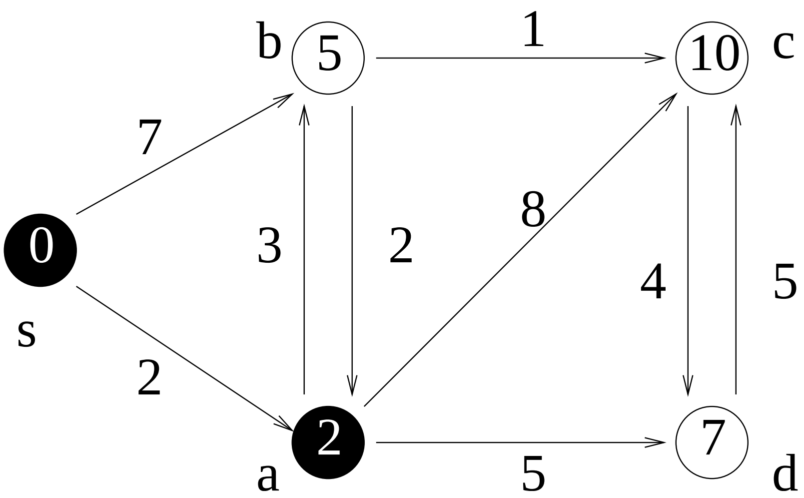
v	s	a	b	c	d
$d[v]$	0	2	7	∞	∞
$pred[v]$	nil	s	s	nil	nil
$color[v]$	B	W	W	W	W

Priority Queue:

v	a	b	c	d
$d[v]$	2	7	∞	∞

Dijkstra's Algorithm

Example:



Step 2: After Step 1, a has the minimum key in the priority queue. As $Adj[a] = \{b, c, d\}$, work on b, c, d and update information.

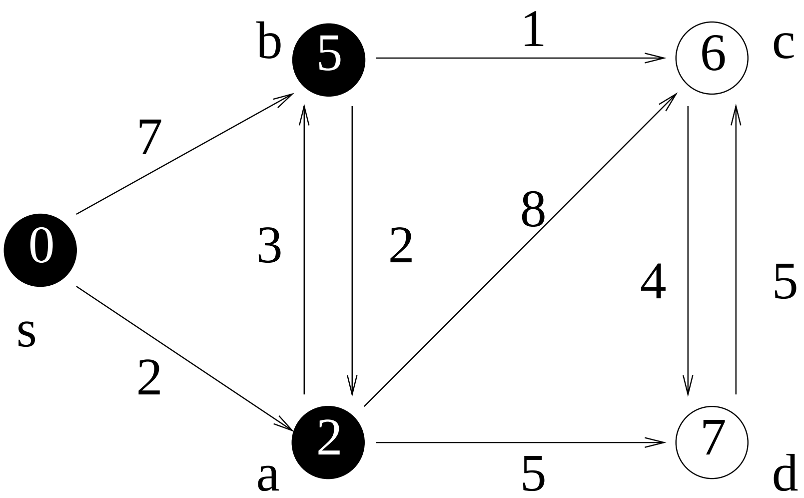
v	s	a	b	c	d
$d[v]$	0	2	5	10	7
$pred[v]$	nil	s	a	a	a
$color[v]$	B	B	W	W	W

Priority Queue:

v	b	c	d
$d[v]$	5	10	7

Dijkstra's Algorithm

Example:



Step 3: After Step 2, b has the minimum key in the priority queue. As $Adj[b] = \{a, c\}$, work on a, c and update information.

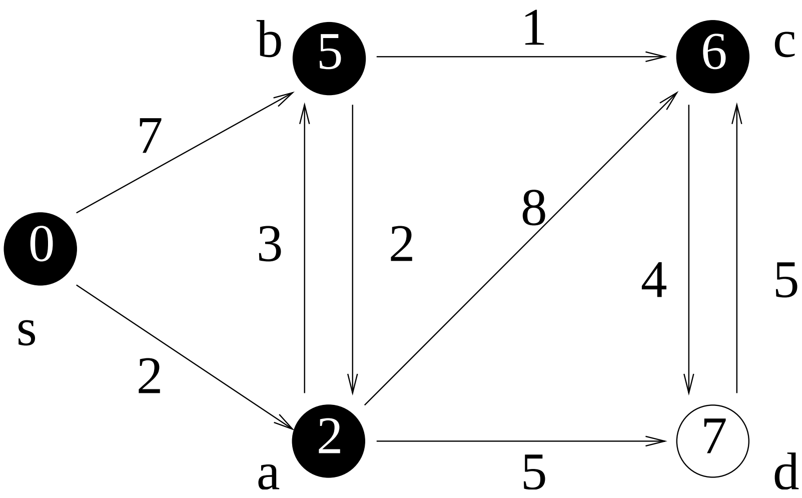
v	s	a	b	c	d
$d[v]$	0	2	5	6	7
$pred[v]$	nil	s	a	b	a
$color[v]$	B	B	B	W	W

Priority Queue:

v	c	d
$d[v]$	6	7

Dijkstra's Algorithm

Example:



Step 4: After Step 3, c has the minimum key in the priority queue. As $Adj[c] = \{d\}$, work on d and update information.

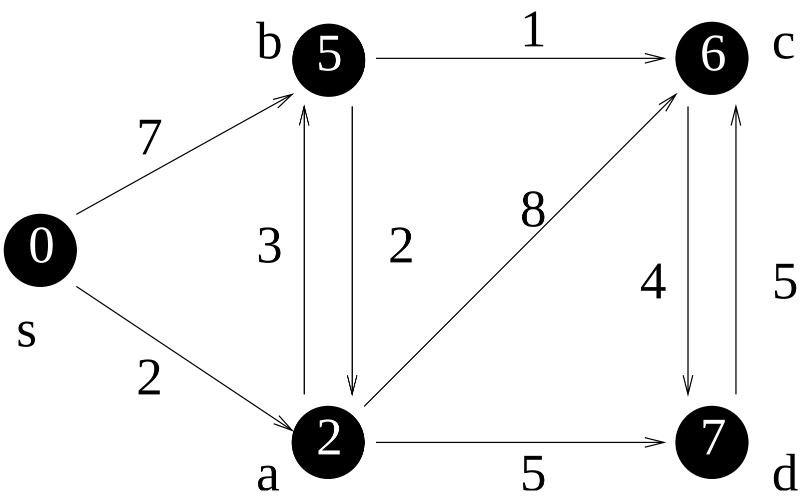
v	s	a	b	c	d
$d[v]$	0	2	5	6	7
$pred[v]$	nil	s	a	b	a
$color[v]$	B	B	B	B	W

Priority Queue:

v	d
$d[v]$	7

Dijkstra's Algorithm

Example:



Step 5: After Step 4, d has the minimum key in the priority queue. As $Adj[d] = \{c\}$, work on c and update information.

v	s	a	b	c	d
$d[v]$	0	2	5	6	7
$pred[v]$	nil	s	a	b	a
$color[v]$	B	B	B	B	B

Priority Queue: $Q = \emptyset$.

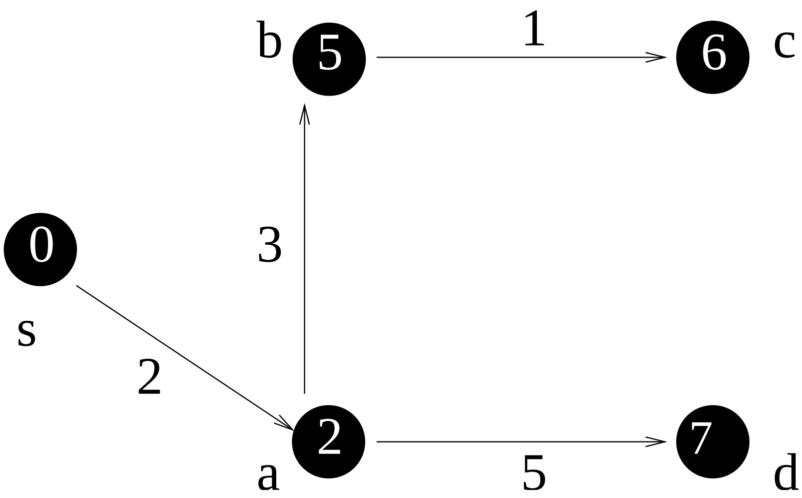
We are done.

Dijkstra's Algorithm

Shortest Path Tree: $T = (V, A)$, where

$$A = \{(pred[v], v) | v \in V \setminus \{s\}\}.$$

The array $pred[v]$ is used to build the tree.



Example:

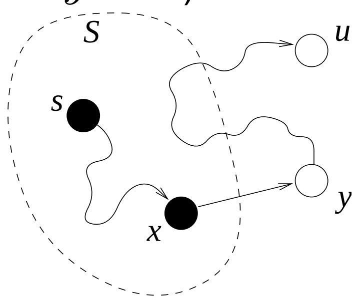
v	s	a	b	c	d
$d[v]$	0	2	5	6	7
$pred[v]$	nil	s	a	b	a

Correctness of Dijkstra's Algorithm

Lemma: When a vertex u is added to S (i.e., dequeued from the queue), $d[u] = \delta(s,u)$.

Proof: Suppose to the contrary that at some point Dijkstra's algorithm *first* attempts to add a vertex u to S for which $d[u] \neq \delta(s,u)$. By our observations about relaxation, $d[u] > \delta(s,u)$.

Consider the situation just prior to the insertion of u. Consider the true shortest path from s to u. Because $s \in S$ and $u \in V \setminus S$, at some point this path must first take a jump out of S. Let (x,y) be the edge taken by the path, where $x \in S$ and $y \in V \setminus S$ (it may be that $x = s$ and/or $y = u$).



Correctness of Dijkstra's Algorithm – Continued

We now prove that $d[y] = \delta(s, y)$. We have done relaxation when processing x, so

$$d[y] \leq d[x] + w(x, y). \tag{1}$$

Since x is added to S earlier, by hypothesis,

$$d[x] = \delta(s, x). \tag{2}$$

Since $\langle s, \ldots, x, y\rangle$ is subpath of a shortest path, by (2)

$$\delta(s, y) = \delta(s, x) + w(x, y) = d[x] + w(x, y). \tag{3}$$

By (1) and (3),

$$d[y] \leq \delta(s, y).$$

Hence

$$d[y] = \delta(s, y).$$

So $y \neq u$ (because we suppose $d[u] > \delta(s, u)$).

Now observe that since y appears midway on the path from s to u, and all subsequent edges are positive, we have $\delta(s, y) < \delta(s, u)$, and thus

$$d[y] = \delta(s, y) < \delta(s, u) \leq d[u].$$

Thus y would have been added to S *before* u, in contradiction to our assumption that u is the next vertex to be added to S.

Proof of the Correctness of Dijkstra's Algorithm

- By the lemma, $d[v] = \delta(s, v)$ when v is added into S, that is when we set $color[v] =$ black.

- At the end of the algorithm, all vertices are in S, then all distance estimates are correct.

Analysis of Dijkstra's Algorithm:

Insert: Performed once for each vertex. n times. .

Non-Empty(): Called one for each vertex. n times. Each time does one `Extract-Min()` and one color setting

inner loop for (each $v \in Adj[u]$) :
is called once for each of the m edges in the graph. $O(m)$ times.
Each call of the inner loop does $O(1)$ work plus, possibly, one `Decrease-Key` operation.

Recall that all of the priority queue operations require $O(\log |Q|) = O(\log n)$ time
we have that the algorithm uses

$$nO(\log n)+nO(\log n)+O(e \log n) = O((n+e) \log n)$$

time.

Further Analysis of Dijkstra's Algorithm:

Dijkstra's original implementation did not use a priority queue but, instead, ran through the entire list of white vertices at each step, finding the one with smallest weight. Each run through requires $O(n)$ time and there are only n steps so his implementation runs in $O(n^2)$ time.

This is slightly better than the priority queue based algorithm for *dense* graphs but much worse for *sparse* ones.

Further Improvements to Dijkstra's Algorithm:

A more advanced pririty queue data structure called a Fibonacci Heapimplements

- Insert:in $O(1)$ time
- Extract-Min(): $O(\log n)$ time
- Decrease-Key: $O(1)$ (amortized) time.

This reduces the running time down to

$$nO(\log n) + nO(\log n) + O(e) = O((n \log n + e))$$

time.

This is a huge improvement for dense graphs

Prove: Dijkstra's algorithm processes vertices in non-decreasing order of their actual distance from the source vertex.