Lecture 9: Dijkstra's Shortest Path Algorithm

CLRS 24.3

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 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 .

3/EXP/MOT/DEF/3/A

Shortest Path Problem for Weighted Graphs

Let be a weighted digraph, with weight function mapping edges to real-valued weights. If , we write for .

The **length** of a path is the sum of the weights of its constituent edges:

length

The **distance** from to , denoted , is the length of the minimum length path if there is a path from to ; and is otherwise.

length(
distance from to is

Single-Source Shortest-Paths Problem

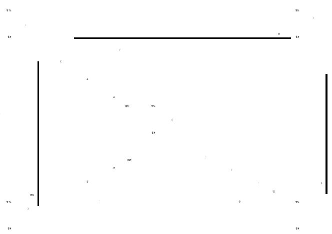
The Problem: Given a digraph with positive edge weights
and a distinguished source vertex ,
determine the distance and a shortest path from the source vertex to every vertex in the digraph.

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, is a shortest path. The subpath is also a shortest path.



length(
distance from to is

Observation Extending this idea we observe the existence of a shortest path tree in which distance from source to vertex 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 of the length of the shortest path for each vertex .

Always and equals the length of a known path
(if we have no paths so far).

Initially and all the other 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.

Here "processing a vertex " means finding new paths and updating for all if necessary. The process by which an estimate is updated is called **relaxation**.

When all vertices have been processed,
for all .

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 , the algorithm will examine all vertices . For each vertex , a new path from to is found (path from to + new edge).

Relaxation. If the new path from to is shorter than , then update to the length of this new path.

Remark: Whenever we set to a finite value, there exists a path of that length. Therefore .

(Note: If , then further relaxations cannot change its value.)

Implementing the Idea of Relaxation

Consider an edge from a vertex to whose weight is . Suppose that we have already processed so that we know and also computed a current estimate for . Then

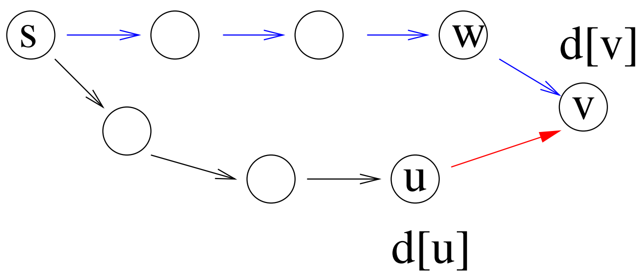
There is a (shortest) path from to with length .

There is a path from to with length .

Combining this path from to with the edge , we obtain another path from to with length .

If , then we replace the old path with the new shorter path . Hence we update

(originally,).



The Algorithm for Relaxing an Edge

Relax(u,v)

if ( )
;
;

Remark: The predecessor pointer is for determining the shortest paths.

Idea of Dijkstra's Algorithm: Repeated Relaxation

Dijkstra's algorithm operates by maintaining a subset of vertices, , for which we know the true distance, that is .

Initially , the empty set, and we set and for all others vertices . One by one we select vertices from to add to .

The set can be implemented using an array of vertex colors. Initially all vertices are white, and we set black to indicate that .

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 to process next?

Answer: We use a greedy algorithm. For each vertex in , we have computed a distance estimate . The next vertex processed is always a vertex for which is minimum, that is, we take the unprocessed vertex that is closest (by our estimate) to .

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 in a priority queue, where the key value of each vertex is .

[Note: if we implement the priority queue using a heap, we can perform the operations **Insert(), Extract Min()**, and **Decrease_Key()**, each in time.]

Review of Priority Queues

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

insert( ): Insert with the key value in .

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

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

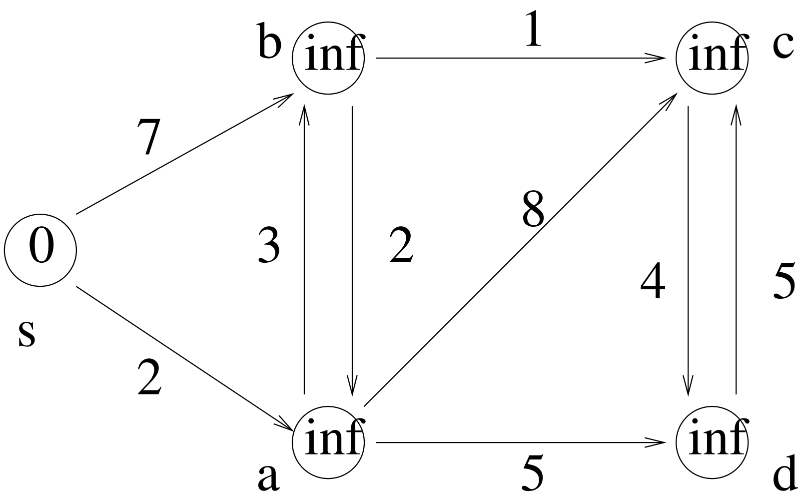
Remark: Priority Queues can be implementad such that each operation takes time . See CLRS!

Description of Dijkstra's Algorithm

```
Dijkstra(G,w,s)
                                              % Initialize
    for (each          )

                      ;
                       white;

                  ;
                   NIL;
              (queue with all vertices);

    while (Non-Empty(   ))                    % Process all vertices

              Extract-Min     ;               % Find new vertex
         for (each                  )
              if (                        )   % If estimate improves

                                      ;          relax
                   Decrease-Key              ;
                             ;

                       black;
```

Dijkstra's Algorithm

Example:



Step 0: Initialization.

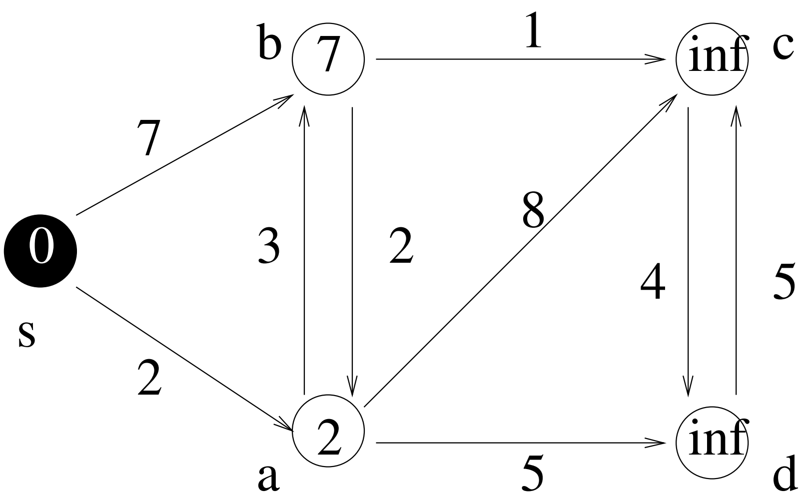
	s	a	b	c	d
	0				
	nil	nil	nil	nil	nil
	W	W	W	W	W

Priority Queue:

s	a	b	c	d
0				

Dijkstra's Algorithm

Example:



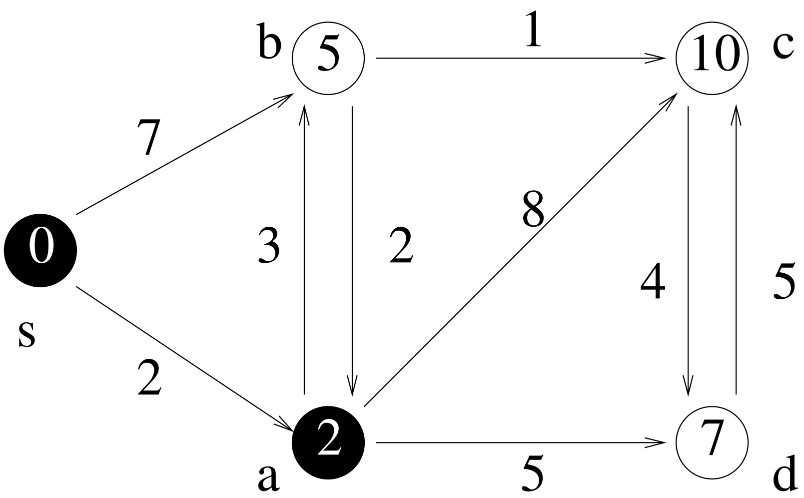
Step 1: As , work on and and update information.

	s	a	b	c	d
	0				
	nil	s	s	nil	nil
	B	W	W	W	W

Priority Queue:	a	b	c	d

Dijkstra's Algorithm

Example:



Step 2: After Step 1, has the minimum key in the priority queue. As , work on , , and update information.

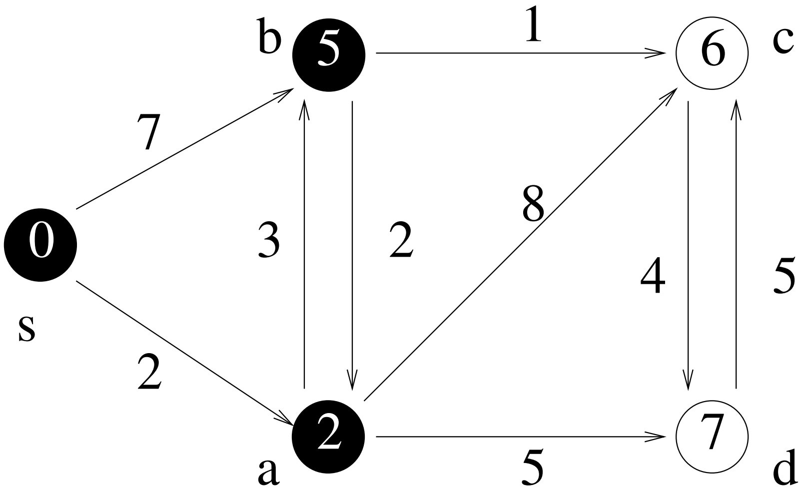
	s	a	b	c	d
	0				
	nil	s	a	a	a
	B	B	W	W	W

Priority Queue:

b	c	d

Dijkstra's Algorithm

Example:



Step 3: After Step 2, has the minimum key in the priority queue. As , work on , and update information.

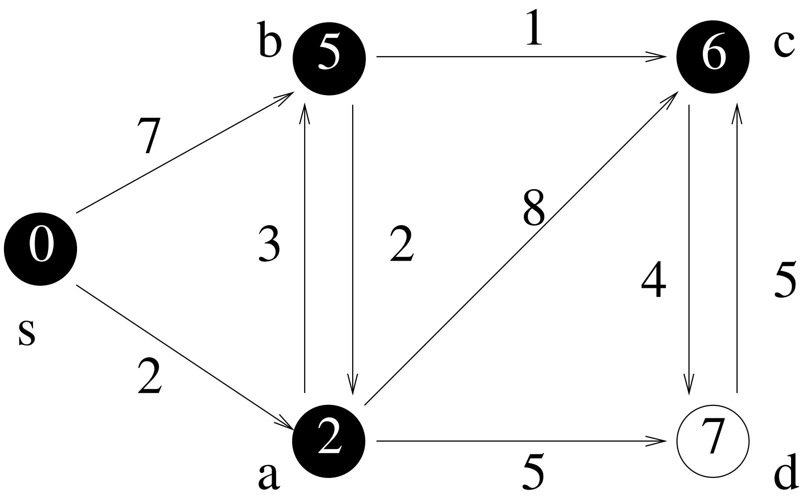
	s	a	b	c	d
	0				
	nil	s	a	b	a
	B	B	B	W	W

Priority Queue:

	c	d

Dijkstra's Algorithm

Example:



Step 4: After Step 3, has the minimum key in the priority queue. As , work on and update information.

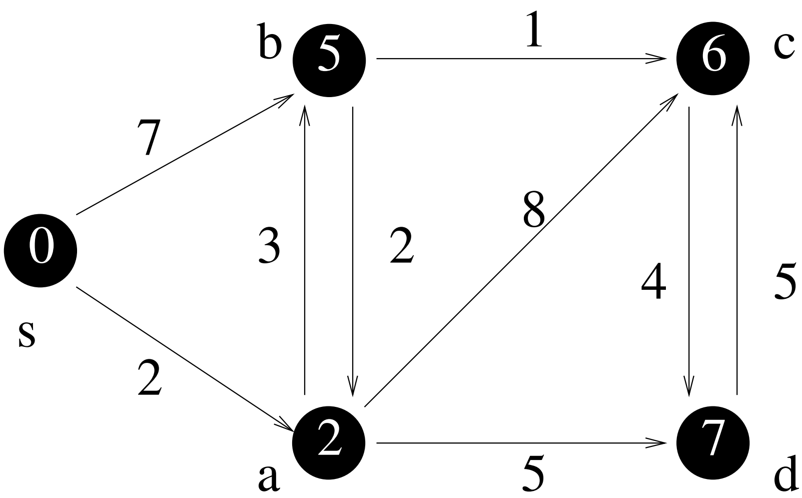
	s	a	b	c	d
	0				
	nil	s	a	b	a
	B	B	B	B	W

Priority Queue:

d

Dijkstra's Algorithm

Example:



Step 5: After Step 4, has the minimum key in the priority queue. As , work on and update information.

	s	a	b	c	d
	0				
	nil	s	a	b	a
	B	B	B	B	B

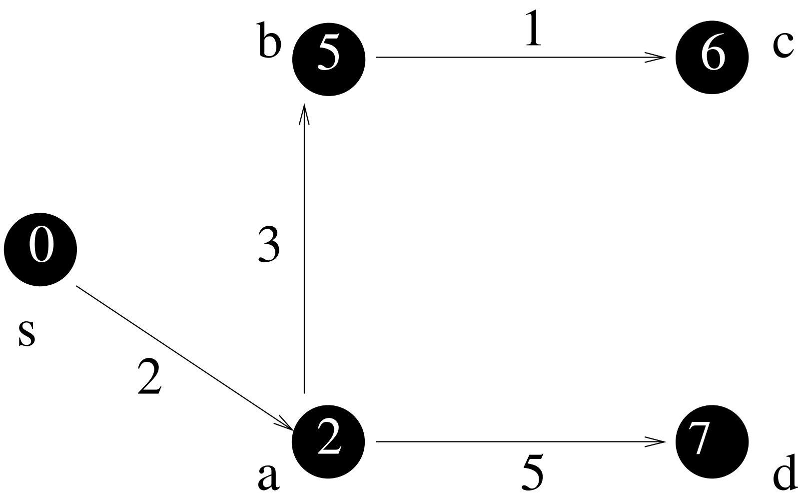
Priority Queue: .

We are done.

Dijkstra's Algorithm

Shortest Path Tree: , where

The array is used to build the tree.



Example:

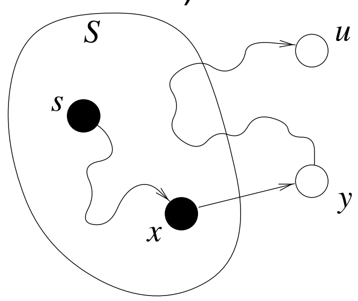
	s	a	b	c	d
	0				
	nil	s	a	b	a

Correctness of Dijkstra's Algorithm

Lemma: When a vertex is added to (i.e., dequeued from the queue), .

Proof: Suppose to the contrary that at some point Dijkstra's algorithm first attempts to add a vertex to for which . By our observations about relaxation, .

Consider the situation just prior to the insertion of . Consider the true shortest path from to . Because and , at some point this path must first take a jump out of . Let be the edge taken by the path, where and (it may be that and/or).



Correctness of Dijkstra's Algorithm – Continued

We now prove that . We have done relaxation when processing , so

(1)

Since is added to earlier, by hypothesis,

(2)

Since is subpath of a shortest path, by (2)

(3)

By (1) and (3),

Hence

So (because we suppose).

Now observe that since appears midway on the path from to , and all subsequent edges are positive, we have , and thus

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

Proof of the Correctness of Dijkstra's Algorithm

By the lemma, when is added into , that is when we set black.

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

Analysis of Dijkstra's Algorithm:

The initialization uses only time.

Each vertex is processed exactly once so `Non-Empty()` and `Extract-Min()` are called exactly once, e.g., times in total.

The inner loop for (each) is called once for each edge in the graph. Each call of the inner loop does work plus, possibly, one `Decrease-Key` operation.

Recalling that all of the priority queue operations require time we have that the algorithm uses

time.

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