Efficient algorithms and data structures

Gregory Kucherov G.Kucherov@skoltech.ru

TA: Ilya Vorobyev I.Vorobyev@skoltech.ru

Course

- *What the course is:*
  - a selection of topics on the design and analysis of algorithms
  - with emphasis on rigorous analysis (Ph.Flajolet: "mathematically oriented engineering")
  - dealing with basic data structures (graphs, strings, trees, tables, …)

- *What the course is not:*
  - a programming course
  - a course oriented to a specific programming language
  - a course oriented to a specific application area
  - a math course

Why study algorithms?

Authors: L.Page, S.Brin, R.Motwani, T.Winograd

Discrete Mathematics 27 (1979) 47–57.
© North-Holland Publishing Company

BOUNDS FOR SORTING BY PREFIX REVERSAL

William H. GATES
Microsoft, Albuquerque, New Mexico

Christos H. PAPADIMITRIOU*†
Department of Electrical Engineering, University of California, Berkeley, CA 94720, U.S.A.

Received 18 January 1978
Revised 28 August 1978

For a permutation σ of the integers from 1 to n, let $f(\sigma)$ be the smallest number of prefix reversals that will transform σ to the identity permutation, and let $f(n)$ be the largest such $f(\sigma)$ for all σ in (the symmetric group) S_n. We show that $f(n) \leq (5n+5)/3$, and that $f(n) \geq 17n/16$ for n a multiple of 16. If, furthermore, each integer is required to participate in an even number of reversed prefixes, the corresponding function $g(n)$ is shown to obey $3n/2 - 1 \leq g(n) \leq 2n+3$.

1. Introduction

We introduce our problem by the following quotation from [1]

> The chef in our place is sloppy, and when he prepares a stack of pancakes they come out all different sizes. Therefore, when I deliver them to a customer, on the way to the table I rearrange them (so that the smallest winds up on top, and so on, down to the largest at the bottom) by grabbing several from the top and flipping them over, repeating this (varying the number I flip) as many times as necessary. If there are n pancakes, what is the maximum number of flips (as a function $f(n)$ of n) that I will ever have to use to rearrange them?

In this paper we derive upper and lower bounds for $f(n)$. Certain bounds were already known. For example, consider any stack of pancakes. An *adjacency* in this stack is a pair of pancakes that are adjacent in the stack, and such that no other pancake has size intermediate between the two. If the largest pancake is on the bottom, this also counts as one extra adjacency. Now, for $n \geq 4$ there are stacks of n pancakes that have no adjacencies whatsoever. On the other hand, a sorted stack must have all n adjacencies and each move (flip) can create at most one adjacency. Consequently, for $n \geq 4$, $f(n) \geq n$. By elaborating on this argument, M.R. Garey, D.S. Johnson and S. Lin [2] showed that $f(n) \geq n+1$ for $n \geq 6$.

For upper bounds—algorithms, that is—it was known that $f(n) \leq 2n$. This can be seen as follows. Given any stack we may start by bringing the largest pancake on top and then flip the whole stack: the largest pancake is now at the bottom,

* Research supported by NSF Grant MCS 77-01193.
† Current address: Laboratory for Computer Science, Massachusets, Institute of Technology, Cambridge, Ma 02139, USA.

The PageRank Citation Ranking: Bringing Order to the Web

January 29, 1998

Abstract

The importance of a Web page is an inherently subjective matter, which depends on the readers interests, knowledge and attitudes. But there is still much that can be said objectively about the relative importance of Web pages. This paper describes PageRank, a method for rating Web pages objectively and mechanically, effectively measuring the human interest and attention devoted to them.

We compare PageRank to an idealized random Web surfer. We show how to efficiently compute PageRank for large numbers of pages. And, we show how to apply PageRank to search and to user navigation.

1 Introduction and Motivation

The World Wide Web creates many new challenges for information retrieval. It is very large and heterogeneous. Current estimates are that there are over 150 million web pages with a doubling life of less than one year. More importantly, the web pages are extremely diverse, ranging from "What is Joe having for lunch today?" to journals about information retrieval. In addition to these major challenges, search engines on the Web must also contend with inexperienced users and pages engineered to manipulate search engine ranking functions.

However, unlike "flat" document collections, the World Wide Web is hypertext and provides considerable auxiliary information on top of the text of the web pages, such as link structure and link text. In this paper, we take advantage of the link structure of the Web to produce a global "importance" ranking of every web page. This ranking, called PageRank, helps search engines and users quickly make sense of the vast heterogeneity of the World Wide Web.

1.1 Diversity of Web Pages

Although there is already a large literature on academic citation analysis, there are a number of significant differences between web pages and academic publications. Unlike academic papers which are scrupulously reviewed, web pages proliferate free of quality control or publishing costs. With a simple program, huge numbers of pages can be created easily, artificially inflating citation counts. Because the Web environment contains competing profit seeking ventures, attention getting strategies evolve in response to search engine algorithms. For this reason, any evaluation strategy which counts replicable features of web pages is prone to manipulation. Further, academic papers are well defined units of work, roughly similar in quality and number of citations, as well as in their purpose - to extend the body of knowledge. Web pages vary on a much wider scale than academic papers in quality, usage, citations, and length. A random archived message posting

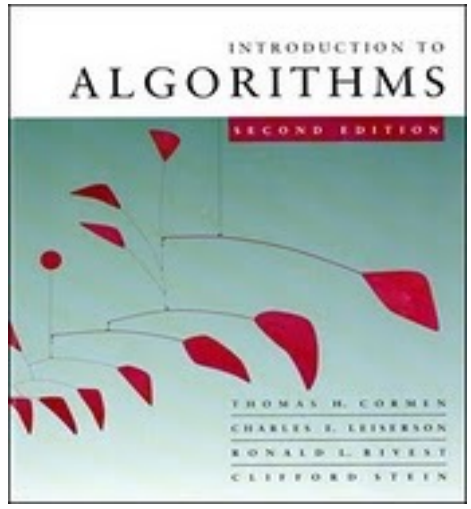
Course

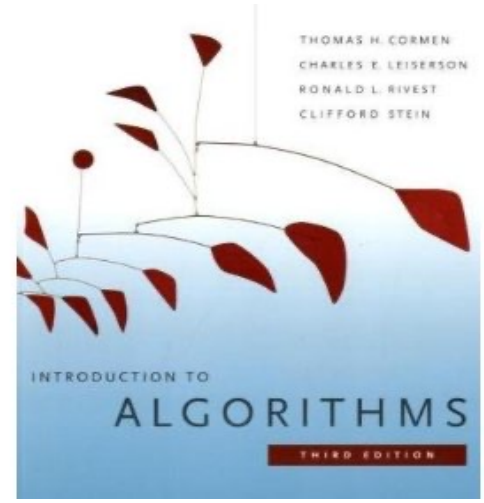
- *Varying level of difficulty*
- *Prerequisites:*
  - imperative programming (C, C++, Java, …)
  - basic probability and discrete maths
  - Basic data structures: lists, arrays, stacks, queues
  - Recursion, Big-Oh notation
  - Sorting, …
- *"Free-style" pseudo-code*

Course organization and grading

- Lectures
  - "control quizzes" during the lectures (5%)
  - obligatory attendance
- Homeworks: programming exercises (60%)
  - five exercises are planned
  - testing platform: codeforces.com – register during coming days! questions to I.Vorobyev@skoltech.ru (with Cc to me) or start discussion on Canvas
- "Exam quizzes"
  - mid-term (15%)
  - final (20%)

Recommended books





CLRS = Cormen & Leiserson & Rivest &Stein

***Some other good algorithm textbooks**:*

- *Steven Skiena*, The Algorithm Design Manual, 2nd Edition, Springer, 2008 [less formal]
- *Jon Kleinberg* and *Éva Tardos*, Algorithm Design, MIT Press 2005 [formal but with examples]
- *Robert Sedgewick* and *Kevin Wayne*, Algorithms, Addison-Wesley, 4th Edition, 2011 [for beginners, Java-oriented]

How to measure the efficiency of algorithms?

- Efficiency (*in this course*) = TIME and SPACE
  - other possible measure of efficiency: accuracy

- *Classical model*: RAM model of computation
  - all memory accesses have equal cost
  - no parallel execution
  - unit cost ($O(1)$) of basic operations (unless we want to explicitely count individual bits operations)
  - space = # of computer words (unless bit complexity is considered); each computer word contains $\Theta(\log n)$ bits
  - other possible measures can be considered: disk accesses, cache misses, probe model, query complexity …

How to measure the efficiency of algorithms?

- Algorithms solve mass problems
  - n: input size (in computer words or bits)
  - time/space as a function of n

- Different *complexity analyses*:
  - ***worst-case*** complexity
  - average-case complexity
  - smoothed analysis
  - query (probe) complexity
  - …

Graphs

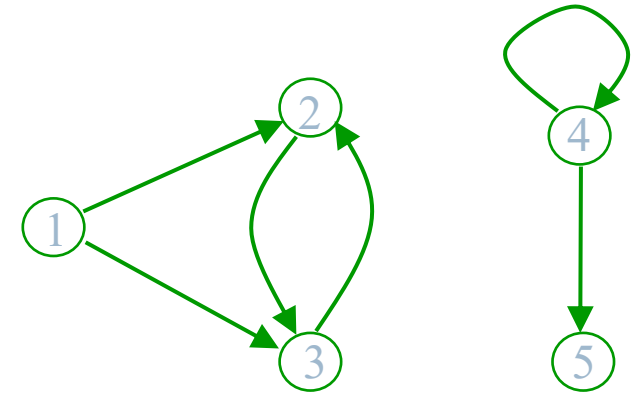
Graphs

Directed graph $G = (V, E)$

V finite set of *nodes (vertices)*

$E \subseteq V \times V$ set of *edges (arcs)*,

i.e., a relation on V



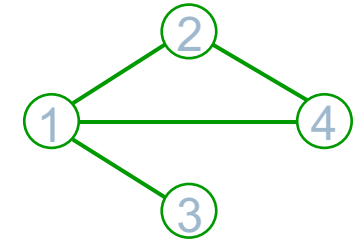
$V = \{ 1, 2, 3, 4, 5 \}$

$E = \{ (1, 2), (1, 3), (2, 3), (3, 2), (4, 4), (4, 5) \}$

Undirected graph $G = (V, E)$

E set of *edges (arcs)*,

symmetric relation



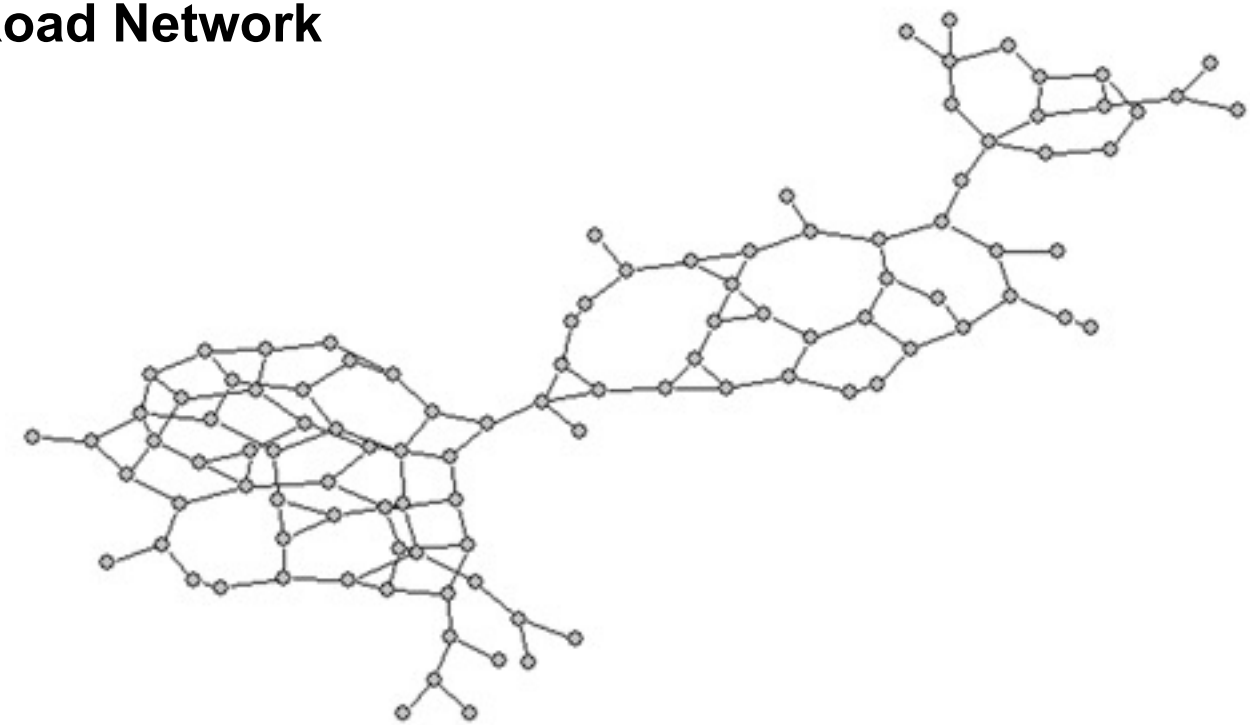
$E = \{ 1, 2, 3, 4 \}$

$V = \{ \{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 4\} \}$

Weighted/Unweighted graphs

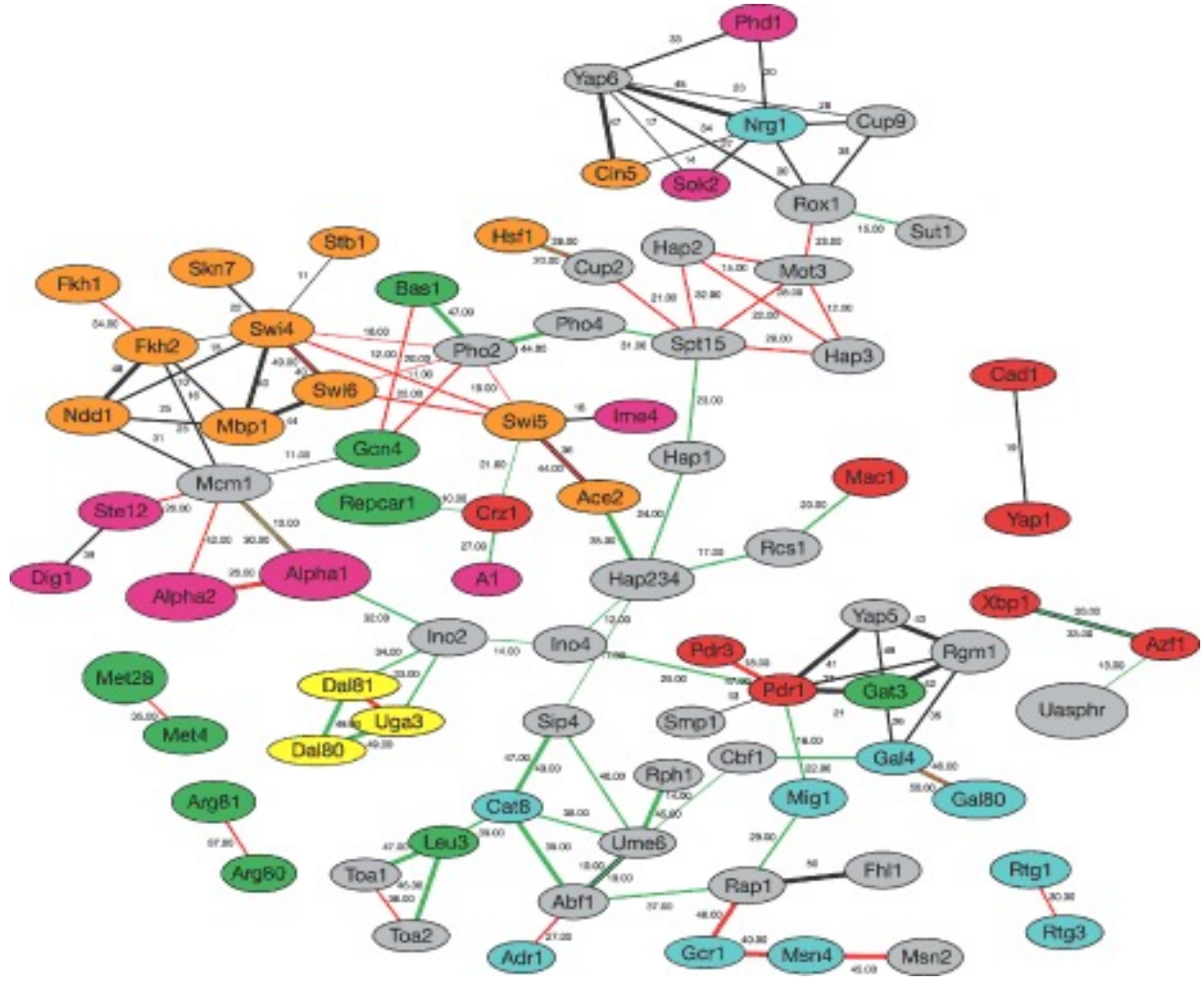
Graphs are everywhere

Road Network



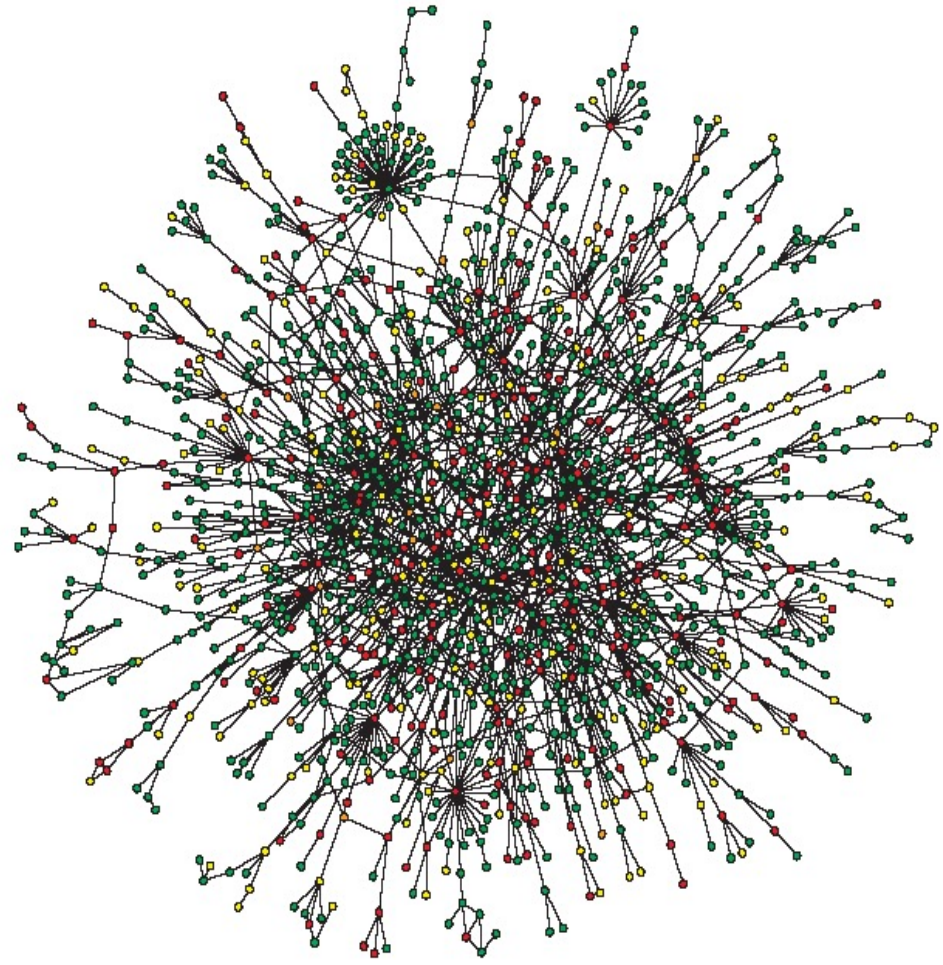
source: [Demšar *et al.* ISCRAM 2007]

Gene regulation network in biology



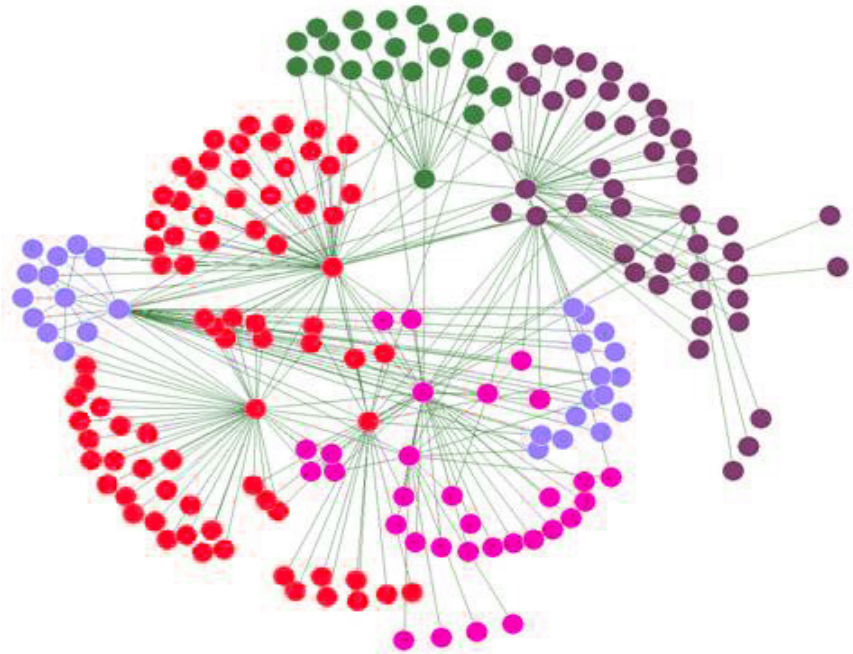
source: J.Galagan, Broad MSC

Protein-protein interaction network (in yeast)



source: [Jeong et al. Nature (2001)]

Social networks



Graph representations

$G = (V, E)$ $V = \{1,2, \ldots, n\}$

Adjacency list

reduces the size if $|E| \ll |V|^2$

reading time : $O(|V| + |E|)$

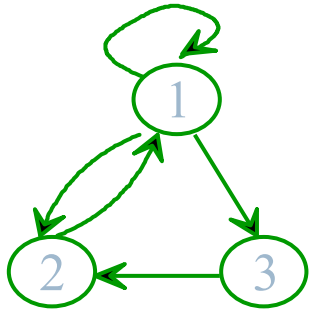
Adjacency matrix

using matrix operations

reading time $O(|V|^2)$

Other representations possible

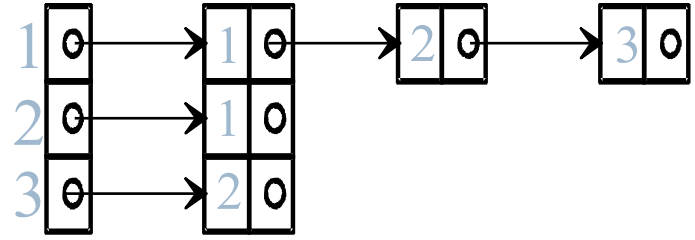
Adjacency lists

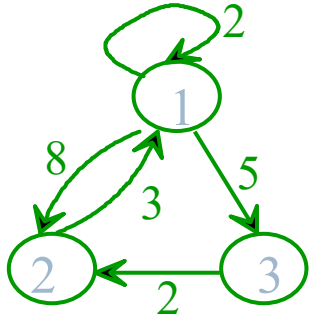


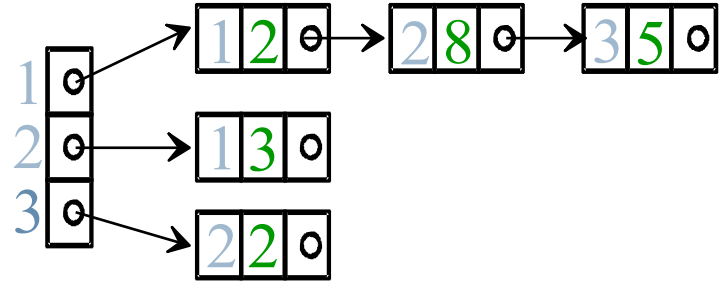
Lists of $E(s)$

$V = \{ 1, 2, 3 \}$

$E = \{ (1,1), (1, 2), (1, 3), (2, 1), (3, 2)\}$

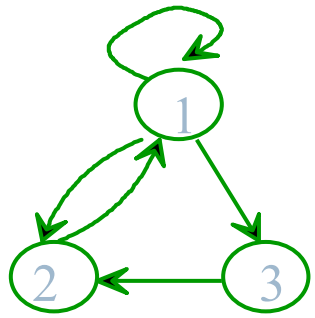






weight: $w: E \to X$

Adjacency matrix

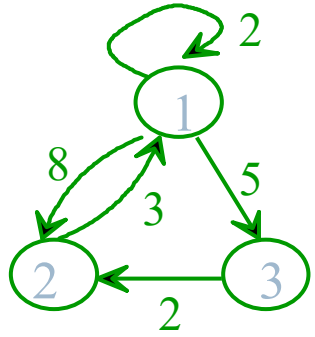


$V = \{ 1, 2, 3 \}$

$E = \{ (1,1), (1, 2), (1, 3), (2, 1), (3, 2) \}$

$$M = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

$M[i, j] = 1$ iff j is adjacent to i



$$W = \begin{pmatrix} 2 & 8 & 5 \\ 3 & 0 & 0 \\ 0 & 2 & 0 \end{pmatrix}$$

weight: $w: E \to X$

Graph algorithms

- Exploration
  - Depth-first or breadth-first search
  - Topological sorting
  - Strongly connected components
- Path computation
  - Shortest path
  - Transitive closure
  - Eulerian and Hamiltonian paths
- Minimum spanning trees
  - Kruskal's and Prim's algorithms
- Networks
  - Maximum flow
- Others
  - Matching
  - Graph coloring
  - Planarity testing
  - …

Shortest paths in graphs

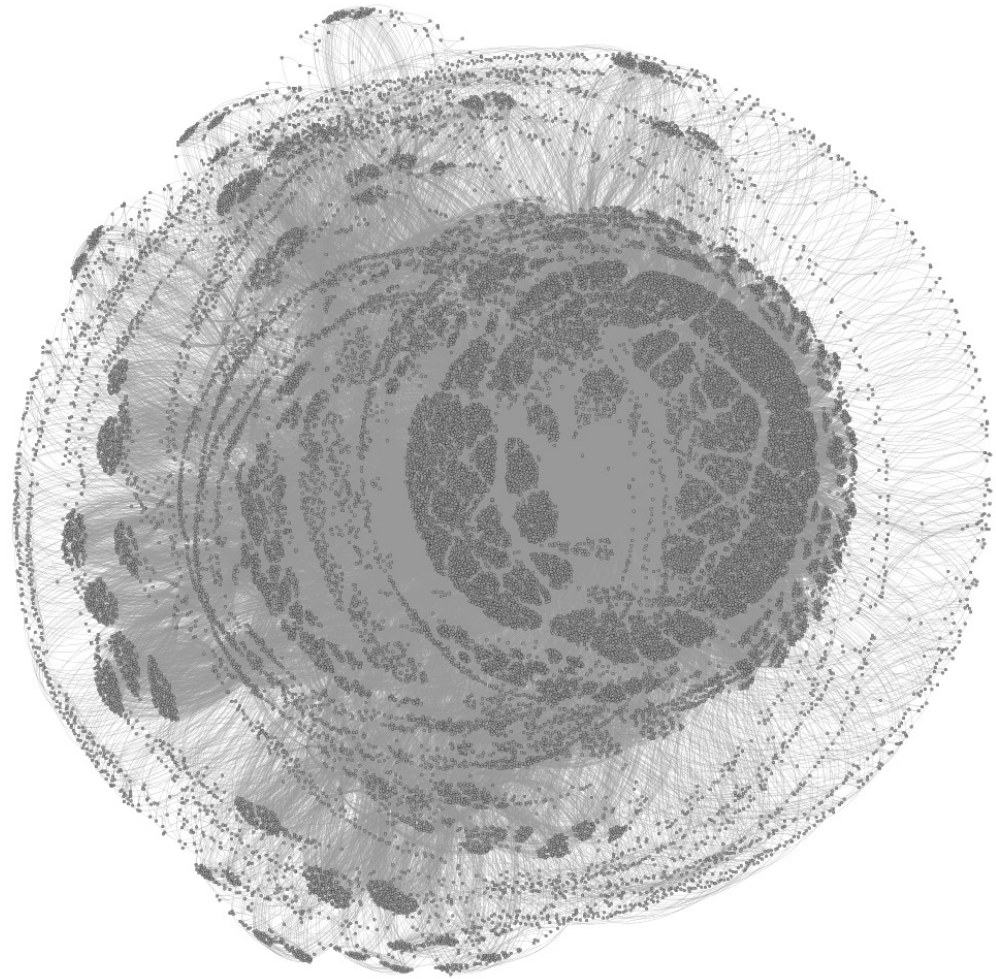
BFS and Dijkstra's algorithm

Single-source shortest path: unweighted case

- Path length = number of edges
- Distance between two nodes = length of the shortest path

- *Problem*: given a (directed or undirected) graph $G = (V, E)$ and a *source node* $s \in V$, compute the distance from s to each reachable node

Single-source shortest path: unweighted case



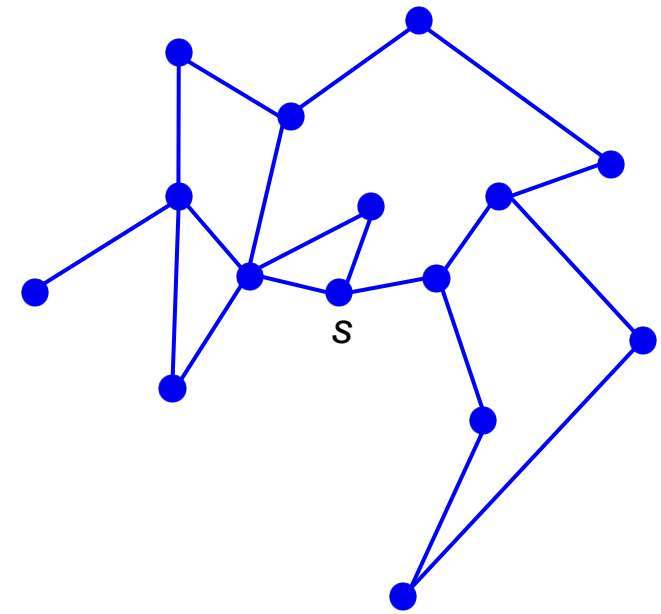
a subgraph (29,160 nodes) of the graph of Rubik's mini cube (2x2x2) configurations (3,674,160 nodes)

https://miscellaneouscoder.wordpress.com/2014/07/28/working-with-rubiks-group-cycle-graphs/

Breadth-first search (BFS)

Given a source node s,

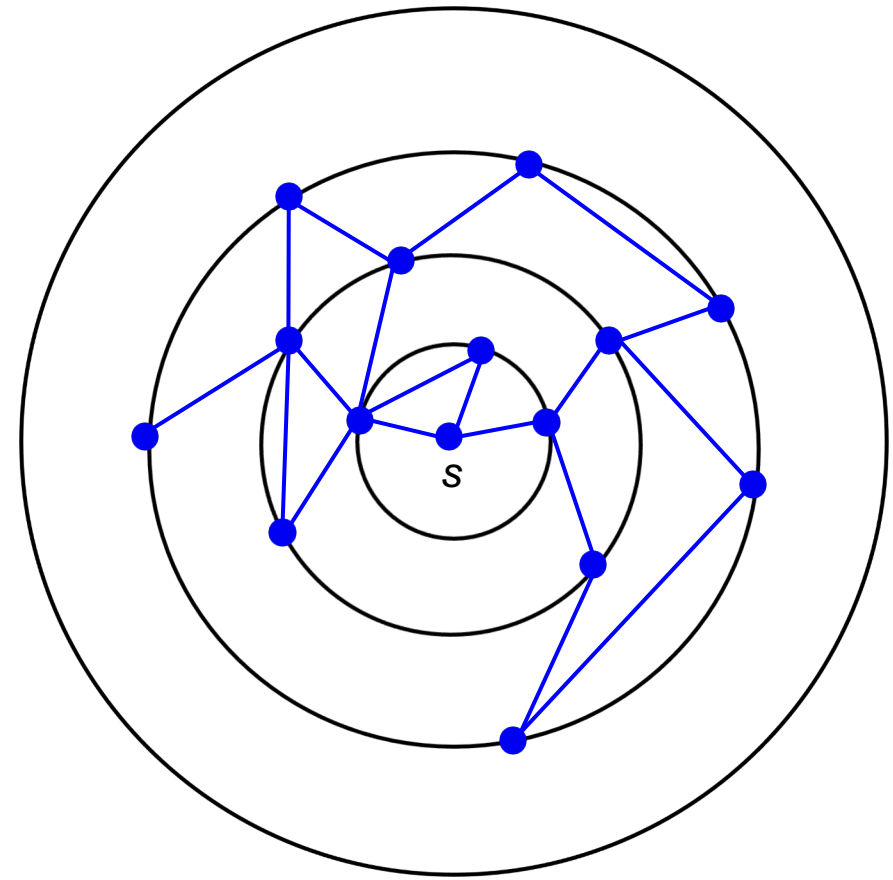
- "Discovers" all nodes reachable from s



Breadth-first search (BFS)

Given a source node s,

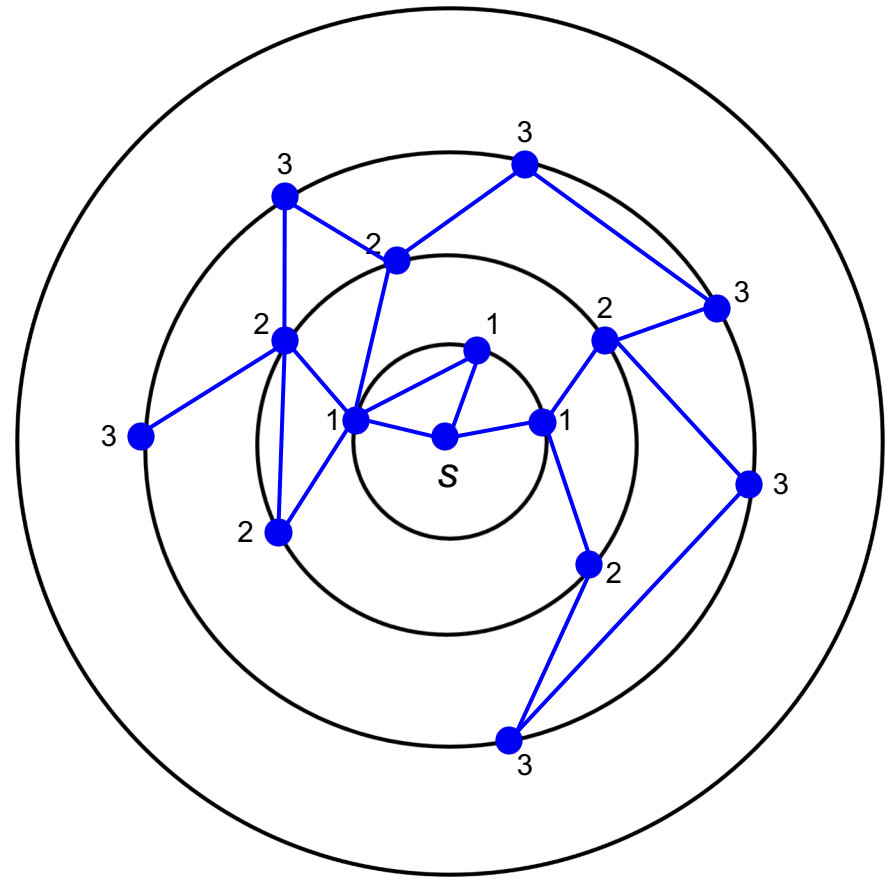
- "Discovers" all nodes reachable from s
- Proceeds by "concentric circles"
- Discovers all nodes at distance d from s before discovering any nodes at distance $d + 1$



Breadth-first search (BFS)

Given a source node s,

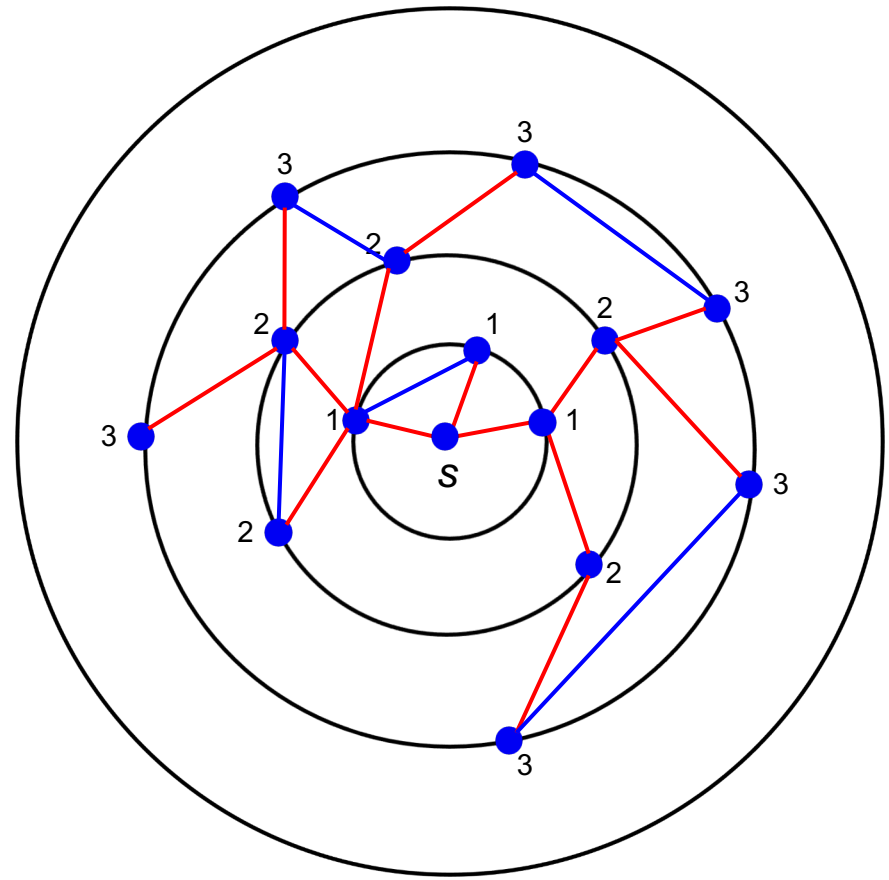
- "Discovers" all nodes reachable from s
- Proceeds by "concentric circles"
- Discovers all nodes at distance d from s before discovering any nodes at distance $d + 1$
- Computes the distances from s



Breadth-first search (BFS)

Given a source node s,

- "Discovers" all nodes reachable from s
- Proceeds by "concentric circles"
- Discovers all nodes at distance d from s before discovering any nodes at distance $d + 1$
- Computes the distances from s
- Computes a *breadth-first tree* encoding one shortest path for each node

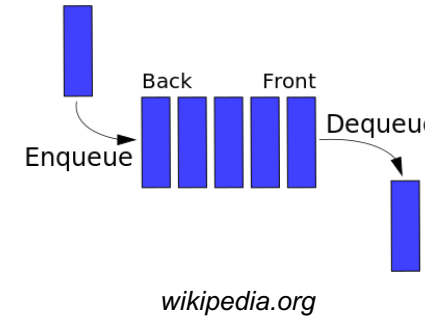


How it works?

- "colors" every node *white* (not yet discovered), *yellow* (discovered but may have white adjacent nodes), or *red* (discovered and all adjacent nodes discovered)
- yellow nodes = "active frontier" (nodes under processing)
- for each node t, the algorithm will compute $d[t]$ – its distance from s – and $\pi(t)$ – its parent in the breadth-first tree
- when processing a (yellow) node, determine all white neighbors, set their distance to be larger by 1, color them yellow. After that, color the node red.

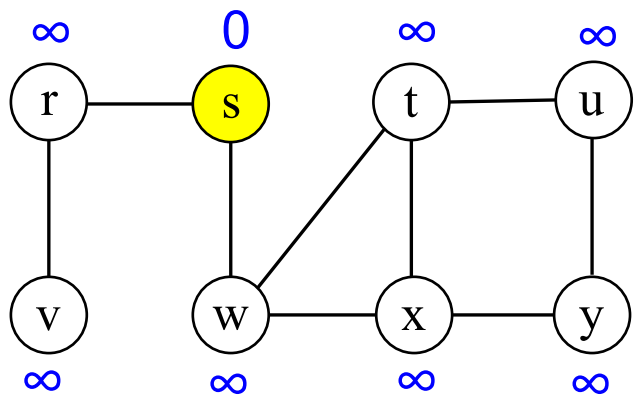
Breadth-first search (BFS)

```
BFS (V, s) ;
for every node v of V  do {
        visited[v] = false ; //s is white
        d[v] = ∞ ; π(v) = nil
        }
visited[s]=true ; //s becomes yellow
d[s]=0 ;
Queue = enqueue (empty-queue, s) ;
while not empty (Queue)  do {
        u = dequeue (Queue) ;
        for  t = first to last successor of u do
                if not visited [ t ] then
                        visited[ t ]=true ; //t becomes yellow
                        d[ t ] = d[ u ]+1 ; π(t) = u
                        Queue = enqueue (Queue, t) ;
        //u becomes red
        }
```



wikipedia.org

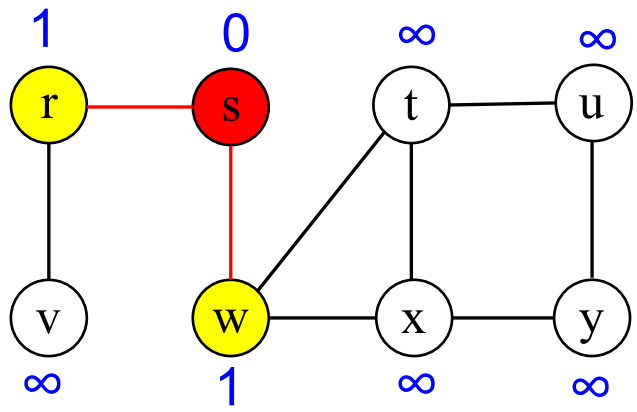
BFS: example



Queue : s

d[.] : 0

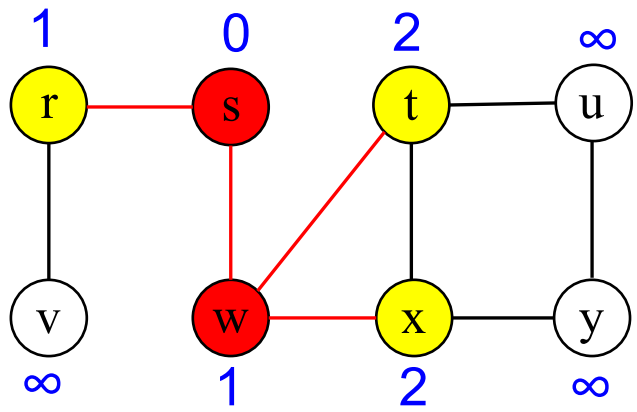
BFS: example



Queue : s w r

d[.] : 0 1 1

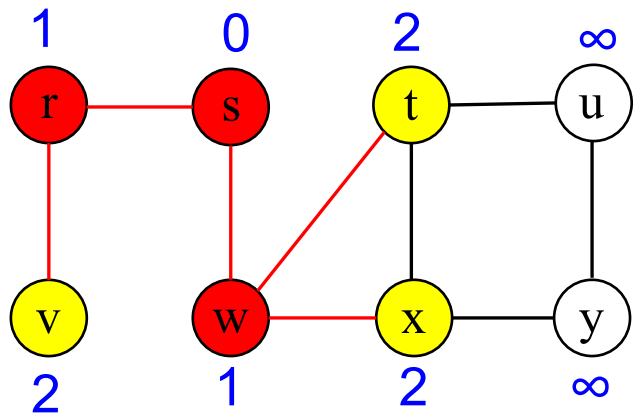
BFS: example



Queue : s w r t x

d[.] : 0 1 1 2 2

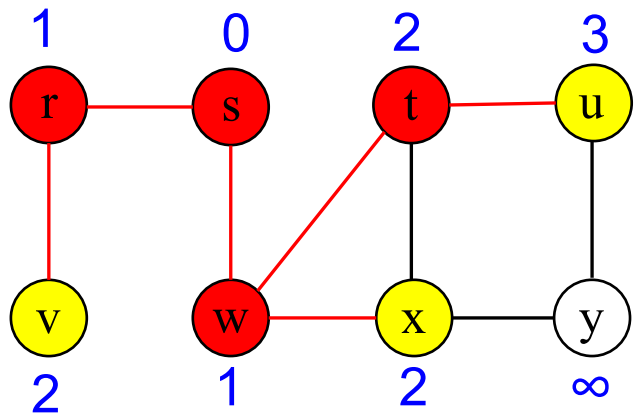
BFS: example



Queue : s w r t x v

d[.] : 0 1 1 2 2 2

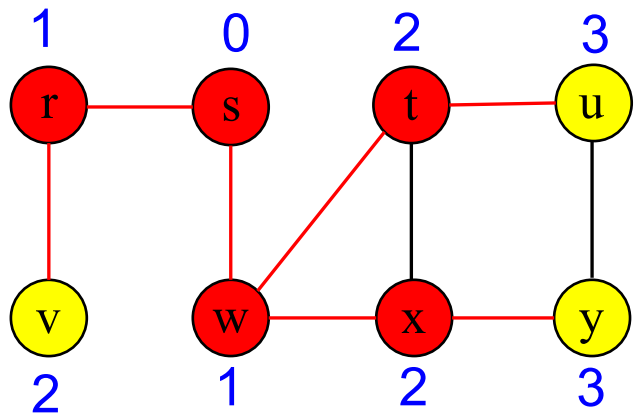
BFS: example



Queue : s w r t x v u

d[.] : 0 1 1 2 2 2 3

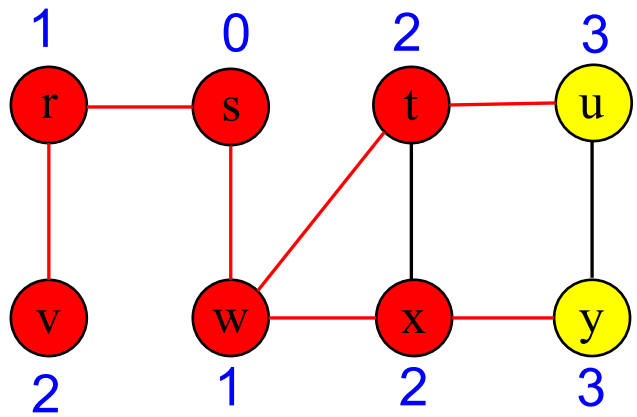
BFS: example



Queue : s w r t x v u y

d[.] : 0 1 1 2 2 2 3 3

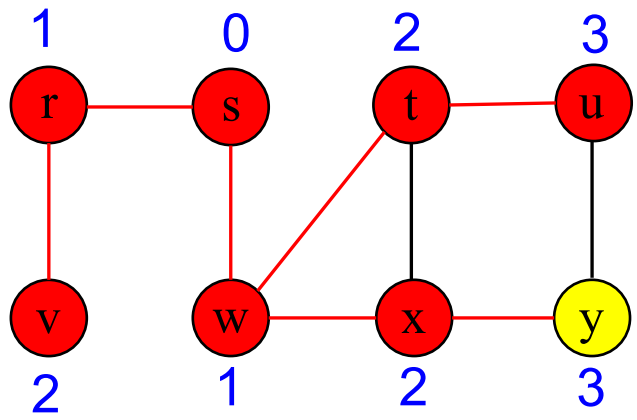
BFS: example



Queue : s w r t x v u y

d[.] : 0 1 1 2 2 2 3 3

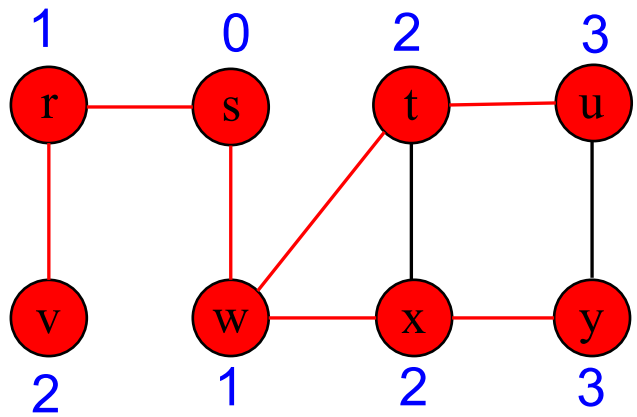
BFS: example



Queue : s w r t x v u y

d[.] : 0 1 1 2 2 2 3 3

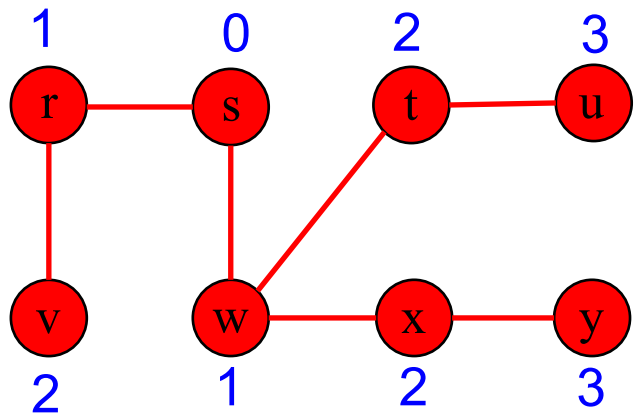
BFS: example



Queue : s w r t x v u y

d[.] : 0 1 1 2 2 2 3 3

BFS: example



breadth-first tree

Queue : s w r t x v u y

d[.] : 0 1 1 2 2 2 3 3

Properties of BFS

- Show that BFS runs in time $O(n+m)$ (assuming the graph is represented by adjacency lists), $n = |V|, m = |E|$

- Show that if $(v_1, v_2, \ldots, v_r)$ is the state of the Queue, then $d[v_r] \leq d[v_1]+1$ and $d[v_i] \leq d[v_{i+1}]$ for all i

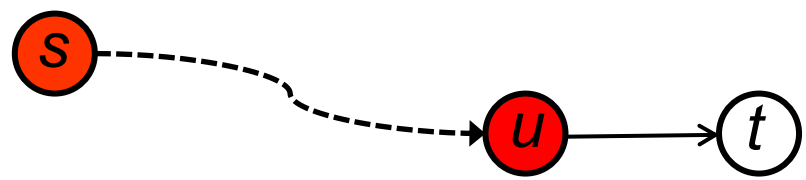
- Show that upon termination $d[v]=\delta(s,v)$, where $\delta(s,v)$ is the length of the shortest path from s to v

Correctness of BFS

Proposition : After the execution of BFS on a graph $G = (V, E)$, $d[t] = \delta(s, t)$ for all $t \in V$.

Proof by contradiction: let t be the closest to s node with $d[t] > \delta(s, t)$
consider a shortest path s to t
u predecessor of t ($\delta(s, t)=\delta(s, u)+1$)
consider the moment when u was dequeued ($d[u]=\delta(s, u)$)

- *t is white* ⇒ $d[t]=\delta(s,t)$ ⇒ *contradiction*
- *t is red* ⇒ $d[t]\leq d[u]$ ⇒ *contradiction*



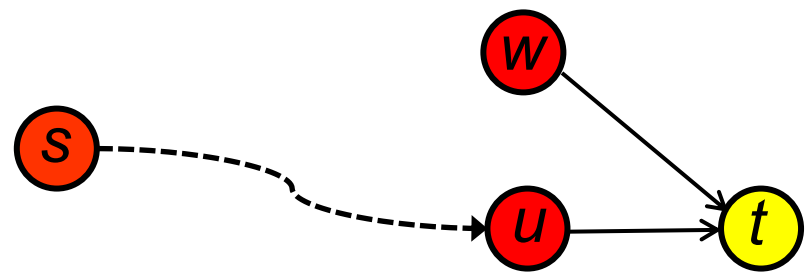
$$\delta(s,t) = \delta(s,u) + 1 = d[u] + 1$$

Correctness of BFS

Proposition : After the execution of BFS on a graph $G = (V, E)$, $d[t] = \delta(s, t)$ for all $t \in V$.

Proof by contradiction: let t be the closest to s node with $d[t] > \delta(s, t)$
consider the shortest path s to t
u predecessor of t ($\delta(s, t)=\delta(s, u)+1$)
consider the moment when u was dequeued ($d[u]=\delta(s, u)$)

- *t is white* ⇒ $d[t]=\delta(s,t)$ ⇒ *contradiction*
- *t is red* ⇒ $d[t]\leq d[u]$ ⇒ *contradiction*
- *t is yellow* ⇒ $\exists(w,t)$ with $d[w]\leq d[u]$ ⇒ $d[t]=d[w]+1\leq d[u]+1$ ⇒ *contradiction*



$$\delta(s,t) = \delta(s,u) + 1 = d[u] + 1$$

Space efficient BFS

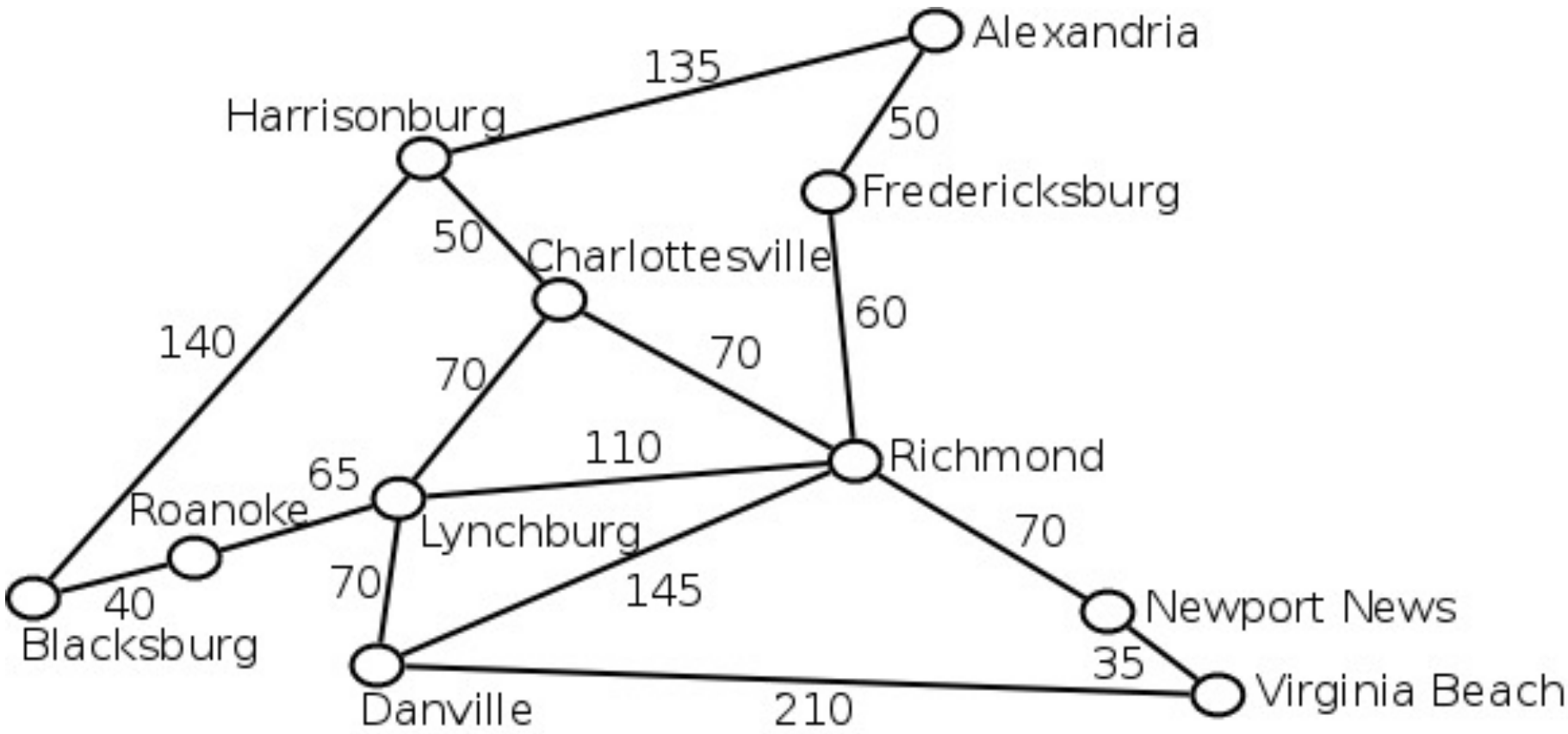
- BFS stores the queue which (in the worst case) can contain $O(n)$ nodes, i.e. $O(n \log n)$ bits
- Can we implement BFS with $o(n \log n)$ bits?

- *Example of a result*: There exists an algorithm that outputs vertices in the BFS order in time $O(n + m)$ and uses $2n + o(n)$ bits
  [N. Banerjee, S. Chakraborty, V. Raman, and S. R. Satti. Space efficient linear time algorithms for BFS, DFS and applications. Theory of Computing Systems, 2018]

Quiz 1

Single-source shortest path: weighted case

Single-source shortest path: weighted case



Shortest path problem

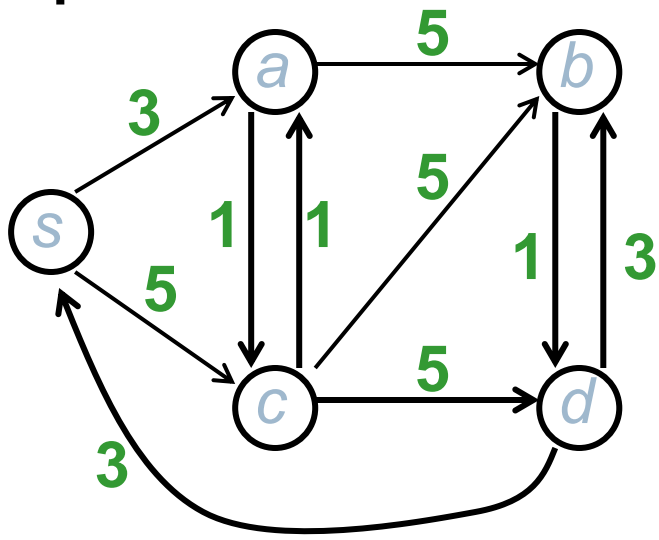
Weighted (directed or undirected) graph: $G = (V, E, w)$ where $w : E \rightarrow \mathbf{R}$ (weight/cost)

Source : $s \in V$

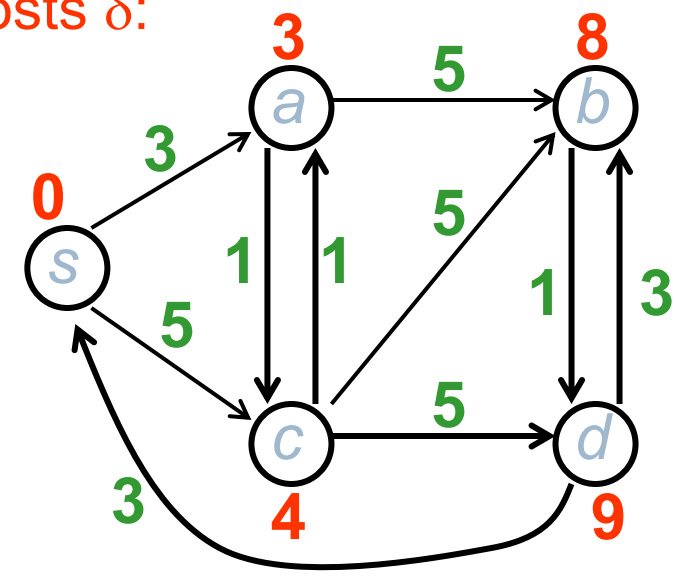
Problem: for all $t \in V$, compute

$$\delta(s, t) = \min \{\{ w(c) \; ; \; c \text{ path from } s \text{ to } t \} \cup \{+\infty\}\}$$

Example:

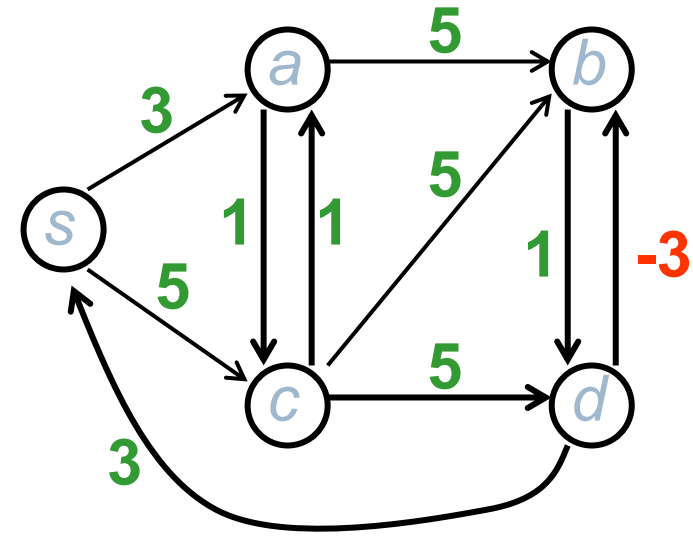


Costs δ:



Properties of the shortest paths

Proposition 1 (existence):
shortest paths are well-defined (i.e. for all $t \in V$, $\delta(s, t) > -\infty$) **iff** the graph does not have a cycle of cost < 0 reachable from *s*

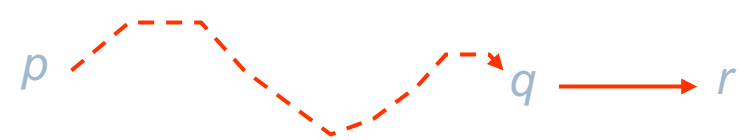


Proposition 2: if there exists a shortest path from *s* to *t*, then there exists one without a cycle

Proposition 3: if there exists a shortest path from *s* to *t*, then there exists one with no more than |*V*|-1 edges

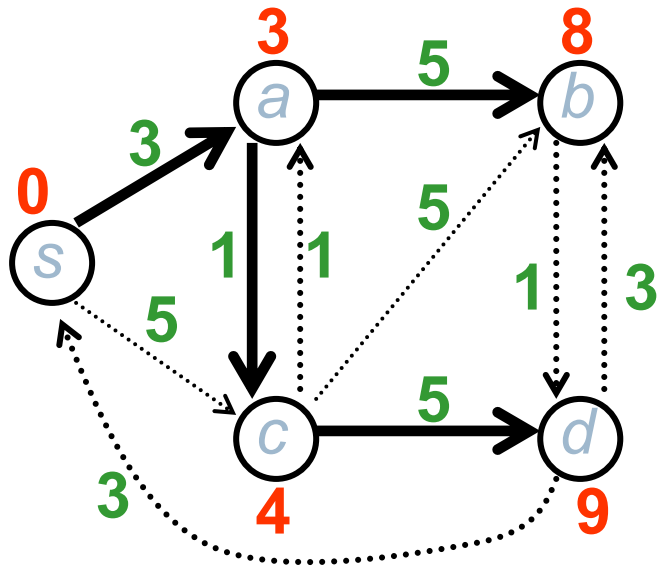
Main properties

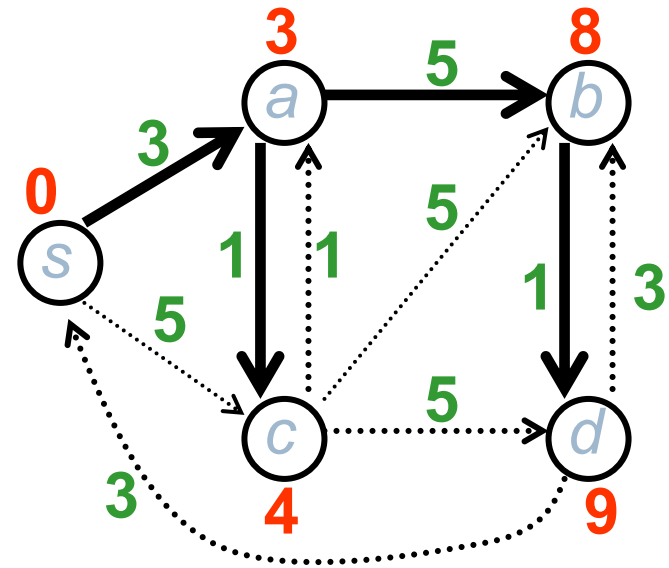
Property 1: $G = (V, E, w)$
let c be a shortest path from p to r
and q be the node preceding r in c.
Then $\delta(p, r) = \delta(p, q) + w(q, r)$.



Property 2: A subpath of a shortest path is a shortest path

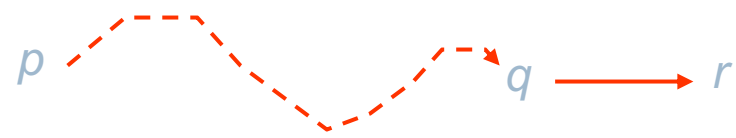
Shortest path tree: tree rooted at s representing shortest paths





Main properties (cont)

Property 3: $G = (V, E, w)$ let c be a path from p to r and q be the node preceding r in c. Then $\delta(p, r) \leq \delta(p, q) + w(q, r)$.



Dijkstra's algorithm: Relaxation

Compute $\delta(s,t)$ by successive approximations

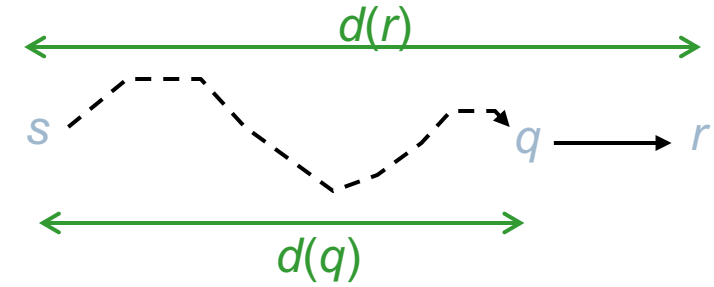
$t \in V$ $d[t]$ = estimate (from above) of $\delta(s, t)$

$\pi[t]$ = predecessor of t on

a path from s to t of cost $d[t]$

Initialization of d and π

```
INIT
        for all  t ∈ V do
                {d[t] = ∞ ;  π[t] = nil}
        d[s] = 0 ;
```



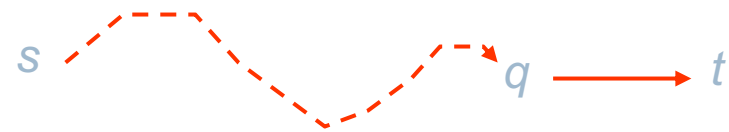
Relaxation of the edge (q, r)

```
RELAX(q, r)
        if  d[q] + w(q, r) < d[r]
        then { d[r] = d[q] + w(q, r) ;  π[r] = q }
```

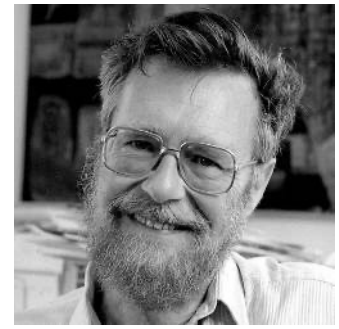
Relaxation (cont)

Proposition :
the following property is an invariant of **RELAX**: for all $t \in V$, $d[t] \geq \delta(s, t)$

Proof: by induction on the number of executions of **RELAX**



Dijkstra's algorithm

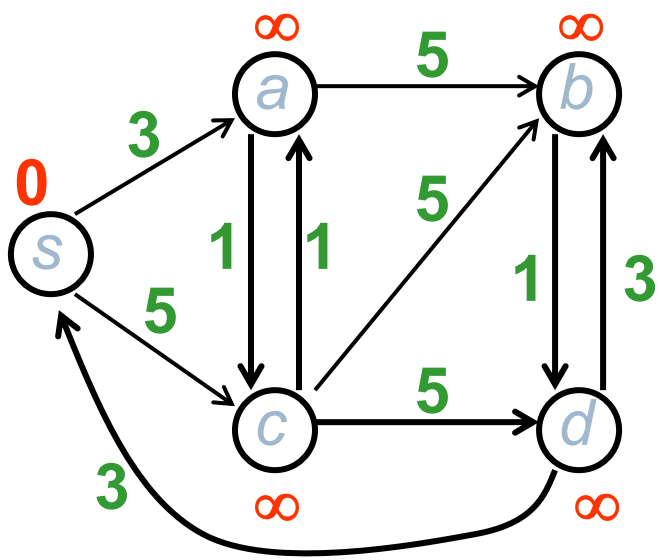


Assumption: $w(p, q) \geq 0$ for all edges (p, q)

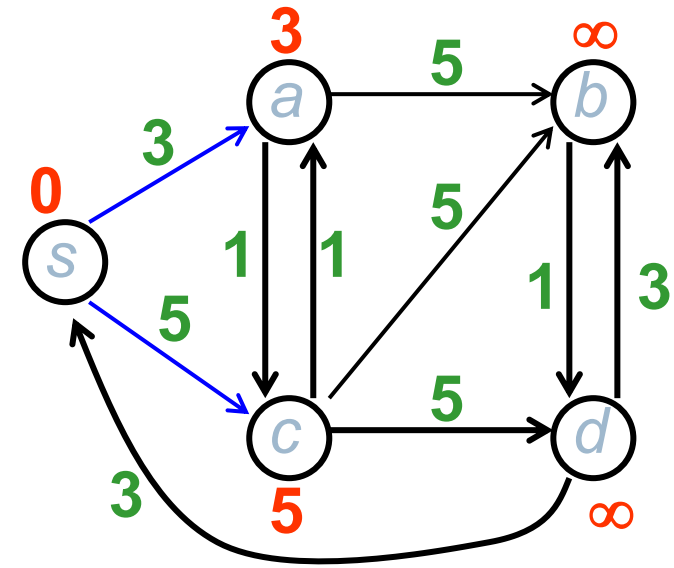
```
DIJKSTRA(G,w,s)
INIT;
S = ∅ ;  Q = V ;
while Q ≠ ∅  do {
      q = MIN_d(Q) ;  Q = Q \ {q} ;  S = S∪{q} ;
      for all r successor of q  do
            RELAX(q, r) ;
}
```

- At each iteration, the algorithm extracts a node from Q that is never returned to Q
- **RELAX**(q, r) may change $d[r]$

Example 1

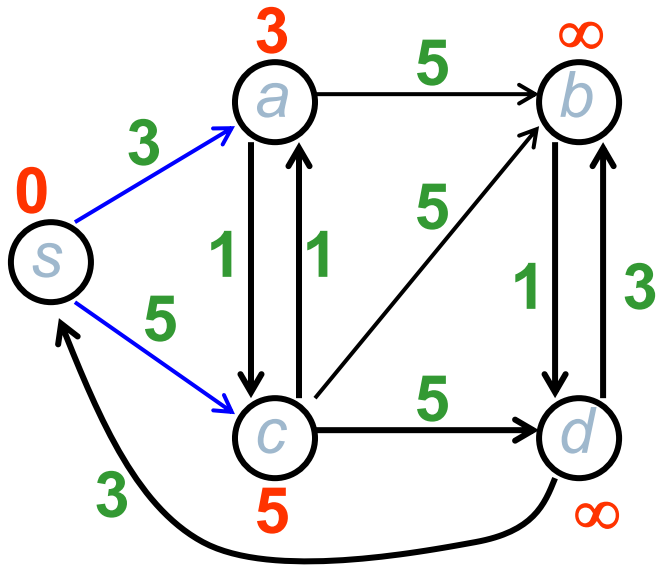


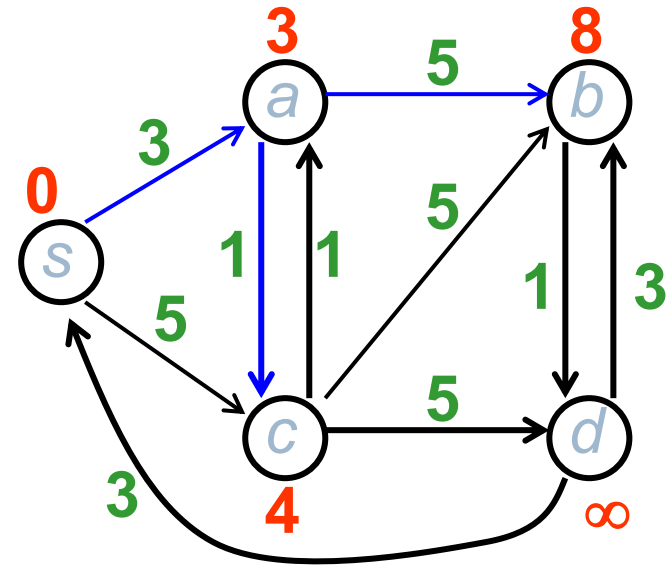
$S = \{\varnothing\}$
$Q = \{s, a, b, c, d\}$
$\pi[s] = \text{nil}$
$\pi[a] = \text{nil}$
$\pi[b] = \text{nil}$
$\pi[c] = \text{nil}$
$\pi[d] = \text{nil}$



$S = \{s\}$
$Q = \{ a, b, c, d \}$
$\pi[s] = \text{nil}$
$\pi[a] = s$
$\pi[b] = \text{nil}$
$\pi[c] = s$
$\pi[d] = \text{nil}$

Example 1 (cont)

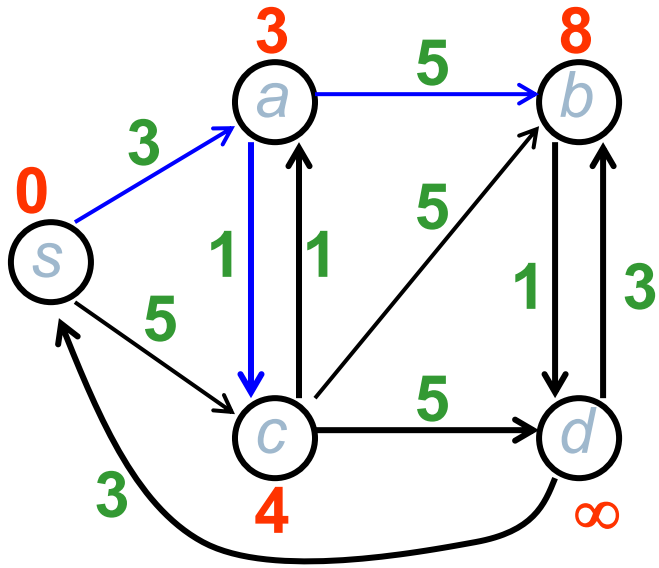


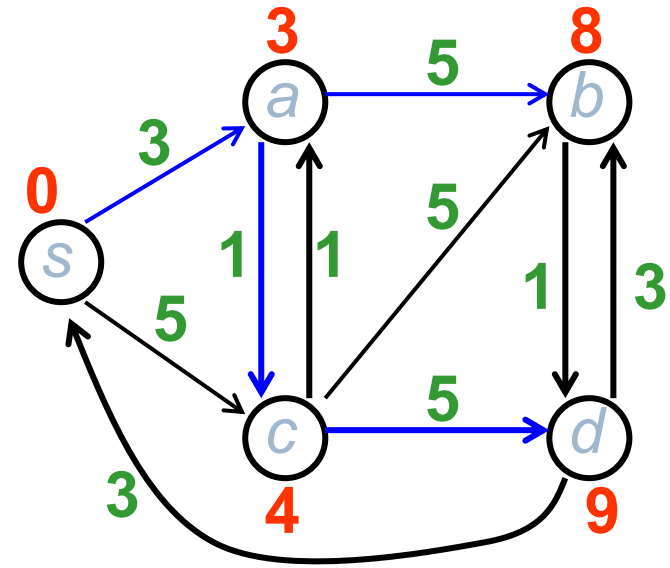


$S = \{s\}$
$Q = \{ a, b, c, d \}$
$\pi[s] = \text{nil}$
$\pi[a] = s$
$\pi[b] = \text{nil}$
$\pi[c] = s$
$\pi[d] = \text{nil}$

$S = \{s, a\}$
$Q = \{ b, c, d \}$
$\pi[s] = \text{nil}$
$\pi[a] = s$
$\pi[b] = a$
$\pi[c] = a$
$\pi[d] = \text{nil}$

Example 1 (cont)

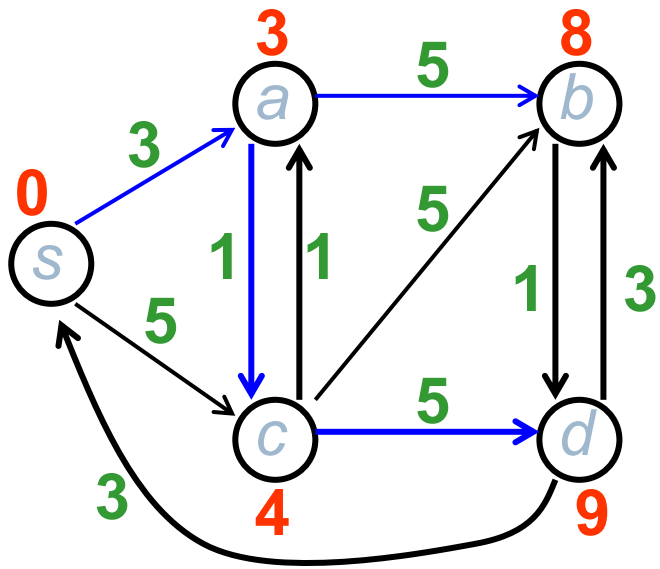


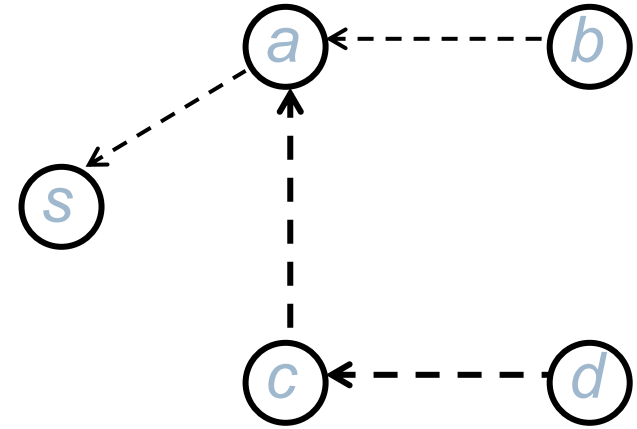


$S = \{s, a\}$
$Q = \{ b, c, d \}$
$\pi[s] = \text{nil}$
$\pi[a] = s$
$\pi[b] = a$
$\pi[c] = a$
$\pi[d] = \text{nil}$

$S = \{s, a, c\}$
$Q = \{ b, d \}$
$\pi[s] = \text{nil}$
$\pi[a] = s$
$\pi[b] = a$
$\pi[c] = a$
$\pi[d] = c$

Example 2 (cont)





$S = \{s, a, c\}$

$Q = \{ b, d \}$, $Q = \{ d \}$ then $Q = \varnothing$

$\pi[s] =$ nil

$\pi[a] = s$

$\pi[b] = a$

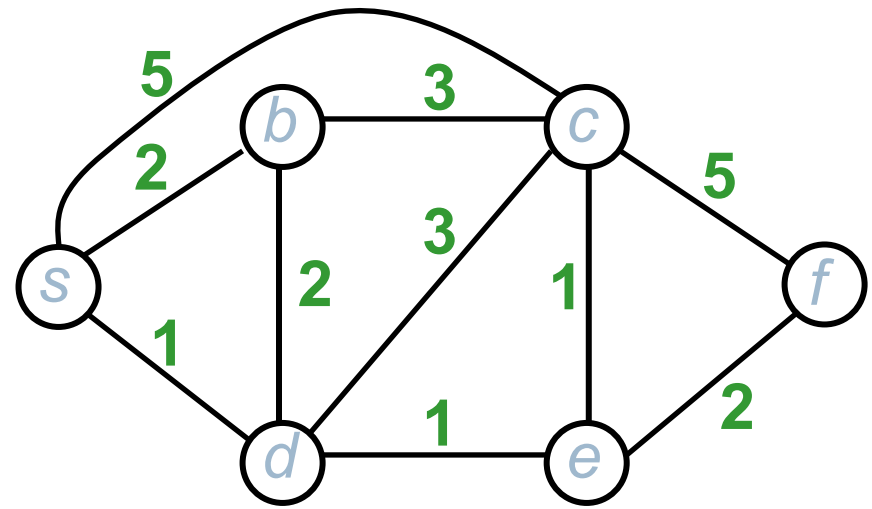
$\pi[c] = a$

$\pi[d] = c$

Extraction order of vertices (in which vertices are extracted from Q and become red):
s, a, c, b, d

Example 2

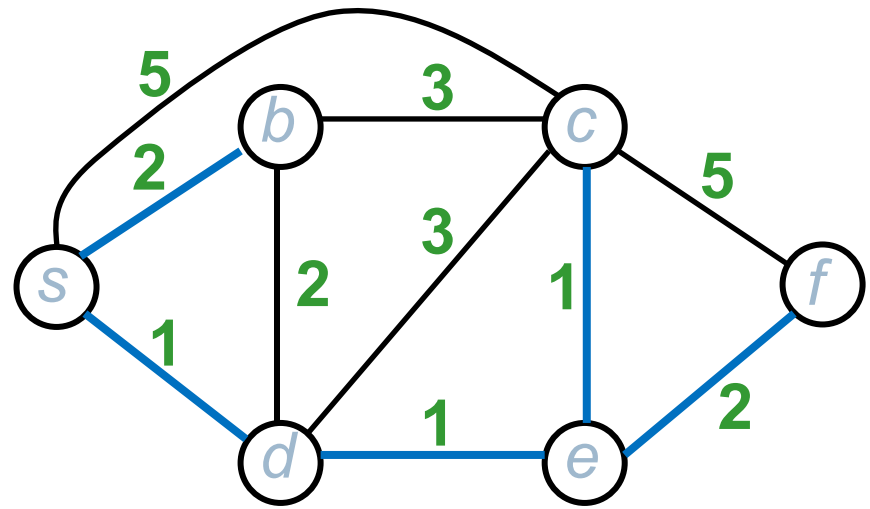
Step	S	d[*s*]	d[*b*],π[*b*]	d[*c*],π[*c*]	d[*d*],π[*d*]	d[*e*],π[*e*]	d[*f*],π[*f*]
0	∅	0					
1	*s*		2,*s*	5,*s*	1,*s*		
2	*sd*		2,*s*	4,*d*		2,*d*	
3	*sde*		2,*s*	3,*e*			4,*e*
4	*sdeb*			3,*e*			4,*e*
5	*sdebc*						4,*e*
6	*sdebcf*						



Example 2

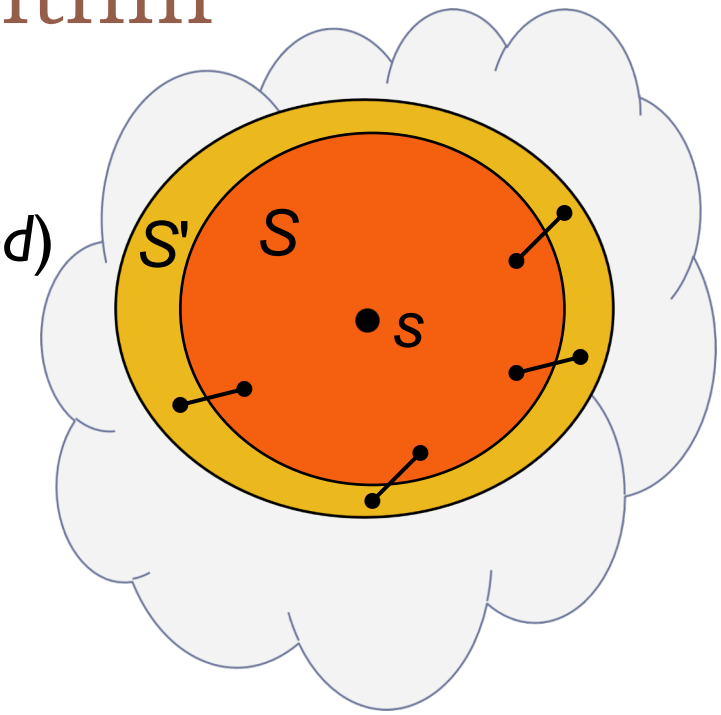
Step	S	d[*s*]	d[*b*],π[*b*]	d[*c*],π[*c*]	d[*d*],π[*d*]	d[*e*],π[*e*]	d[*f*],π[*f*]
0	∅	0					
1	*s*		2,*s*	5,*s*	1,*s*		
2	*sd*		2,*s*	4,*d*		2,*d*	
3	*sde*		2,*s*	3,*e*			4,*e*
4	*sdeb*			3,*e*			4,*e*
5	*sdebc*						4,*e*
6	*sdebcf*						

extraction order of vertices



Properties of Dijkstra's algorithm

- Algorithm maintains three sets:
  - *S*: *finished* nodes, for which $d[t] = \delta(s, t)$ (*red*)
  - S': nodes of *Q* with $d[t]<\infty$ (*yellow*)
  - nodes of *Q* with $d[t]=\infty$ (*white*)

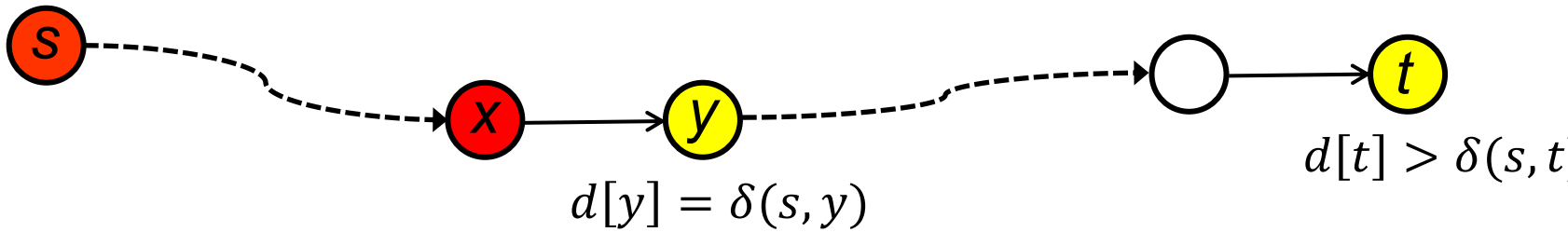


- Algorithm can be seen as expanding a ball centered at s following a *greedy strategy*

Correctness of Dijkstra's algorithm

Proposition : After the execution of Dijkstra's algorithm on a graph $G = (V, E, w)$, $d[t] = \delta(s, t)$ for all $t \in V$.

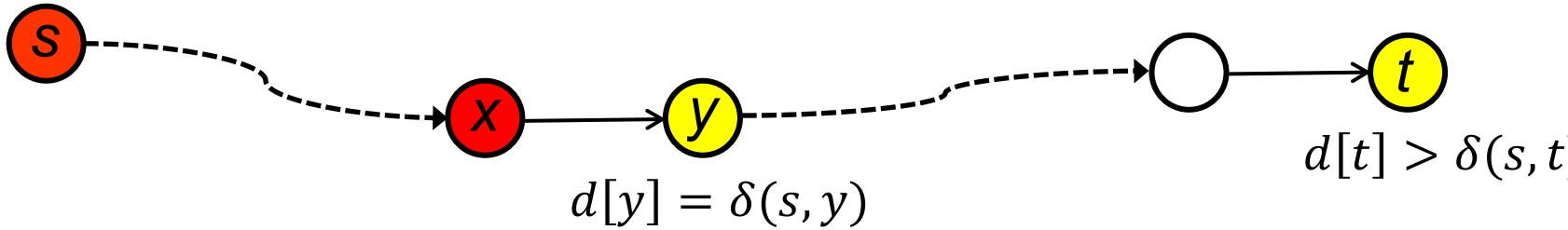
Proof by contradiction: let $d[t] \neq \delta(s, t)$



Correctness of Dijkstra's algorithm

Proposition : After the execution of Dijkstra's algorithm on a graph $G = (V, E, w)$, $d[t] = \delta(s, t)$ for all $t \in V$.

Proof by contradiction: let $d[t] \neq \delta(s, t)$



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

Quiz 2

Complexity of Dijkstra's algorithm

Complexity of Dijkstra's algorithm

With adjacency matrix

time $O(n^2)$ (where $n = |V|$)

With adjacency lists

depends on the data structure for *Q*

we need to support operations:

- insert an element to *Q*
- extract an element with minimum *d* value
- modify (decrease) the *d* value of an element (when relaxing)

Complexity of Dijkstra's algorithm

With adjacency matrix

time $O(n^2)$ (where $n = |V|$)

With adjacency lists

depends on the data structure for *Q*

we need to support operations:

- insert an element to *Q*
- extract an element with minimum *d* value
- modify (decrease) the *d* value of an element (when relaxing)

⇒ (min-)priority queue

Priority Queues

- (max-)Priority Queue is a data structure that supports operations
  - INSERT(S,*x*)
  - MAX(S)
  - EXTRACT-MAX(S)
  - INCREASE-KEY(S,*x*,*k*): increase the key of *x* to *k*
- Priority Queues are used in
  - Dijkstra's algorithm for shortest paths
  - Prim's algorithm for minimum spanning tree
  - other greedy algorithms
- Implemented using heaps

Priority Queues: time bounds

- MAX: $O(1)$
- EXTRACT-MAX, INCREASE-KEY, INSERT: $O(\log(n))$

Various improvements have been proposed

- *Fibonacci heaps* take $O(1)$ *amortized* time for INSERT and INCREASE-KEY
- if keys are integers bounded by C, van Emde Boas trees support INSERT, DELETE, MAX, MIN, SUCC, PRED in time $O(\log \log(C))$

Back to Dijkstra's algorithm

With adjacency matrix

time $O(n^2)$ (where $n = |V|$)

With adjacency lists

Q : priority queue

if implemented by binary heaps:

n building a heap of n elements: $O(n)$

n operations $\mathbf{MIN_d}$: $O(n \log n)$

m operations **RELAX** : $O(m \log n)$ (where $m = |E|$)

total time $O((n + m) \log n)$

improves over $O(n^2)$ if $m = o(n^2 / \log n)$

time can be improved to $O(n \log n + m)$ using *Fibonacci heaps,*

as decreasing the key takes $O(1)$ amortized