Efficient algorithms and data structures

Gregory Kucherov G.Kucherov@skoltech.ru

TA: Stanislav Protasov

Programming exercises:

Ilya Vorobyov I.Vorobyov@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, …)
  - including programming assignments and in-class projects
- *What the course is not:*
  - a programming course
  - a course oriented to a specific programming language (an imperative programming language is assumed, one of Python, C, C++, Java)
  - a course oriented to a specific application area
  - a math course

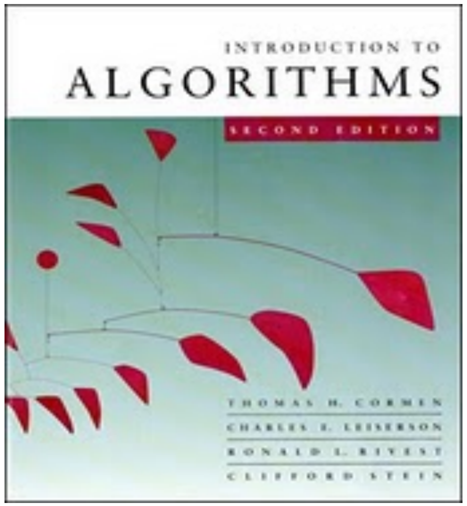
Course

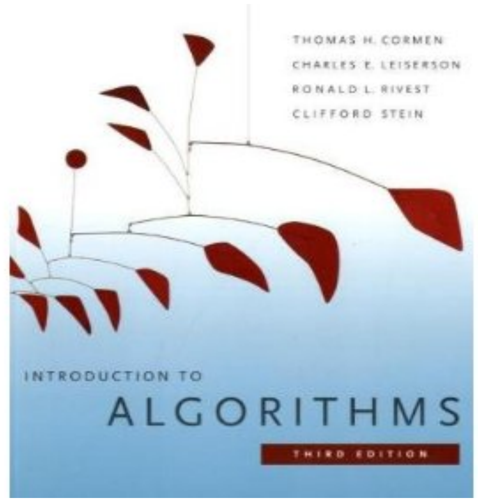
- *Varying level of difficulty*
- *Prerequisites:*
  - imperative programming (C, C++, Java, …)
  - Basic data structures: lists, arrays, stacks, queues
  - Recursion, Big-Oh notation
  - Sorting, …
- *“Free-style” pseudo-code*
- Having a laptop assumed

Grading

- participation in class 10%
  - full attendance is expected
  - in-class projects
- programming exercises 40%
  - one every ~2 weeks
  - plagiarism is not tolerated
- exam 50%

Useful books





CLRS = Cormen & Leiserson & Rivest &Stein

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

• *Steven Skiena*, The Algorithm Design Manual, 2nd Edition, Springer, 2008 [a bit advanced?]
• *Jon Kleinberg* and *Éva Tardos*, Algorithm Design, MIT Press 2005
• *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

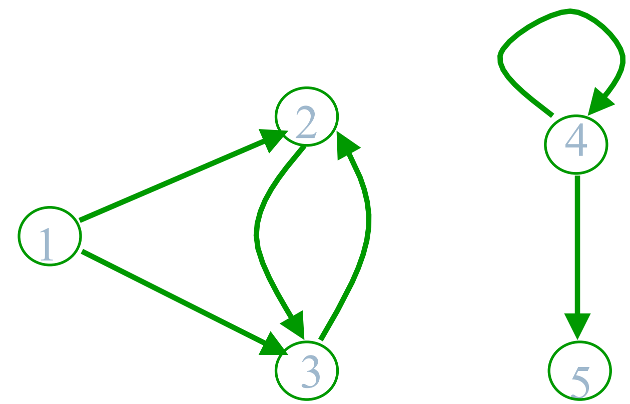
Graphs

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



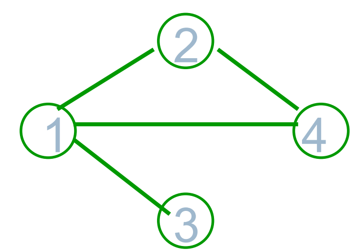
$V = \{ 1, 2, 3, 4, 5 \}$

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

Undirected graph $G = (V, E)$

E set of *edges (arcs)*,

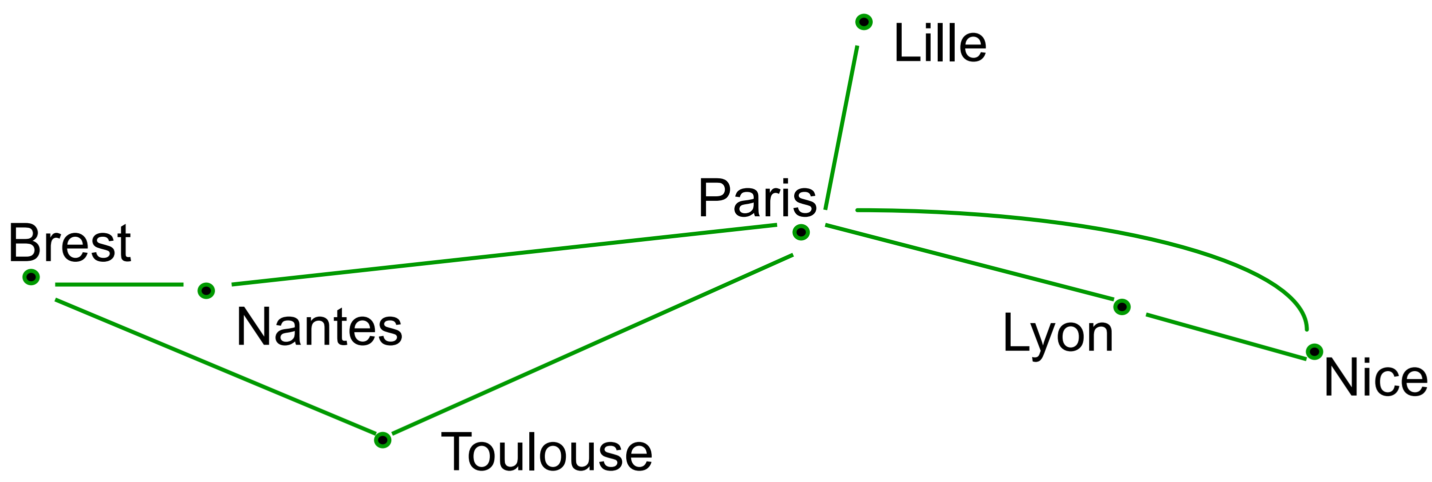
symmetric relation



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

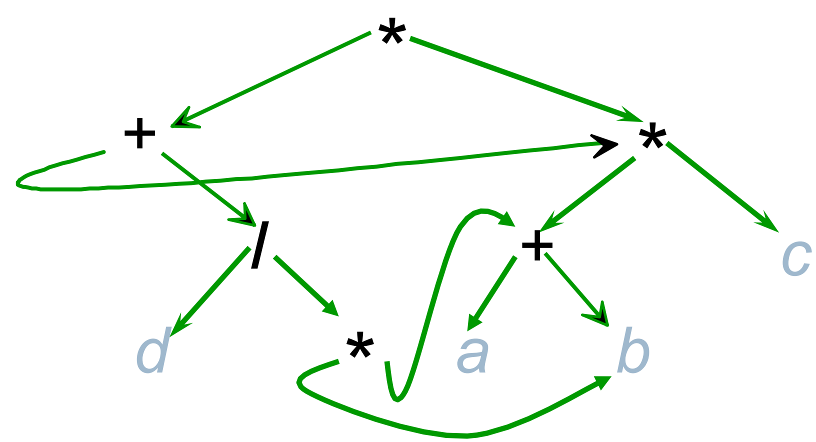
$V = \{ \{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 4\} \}$

Graphs are everywhere



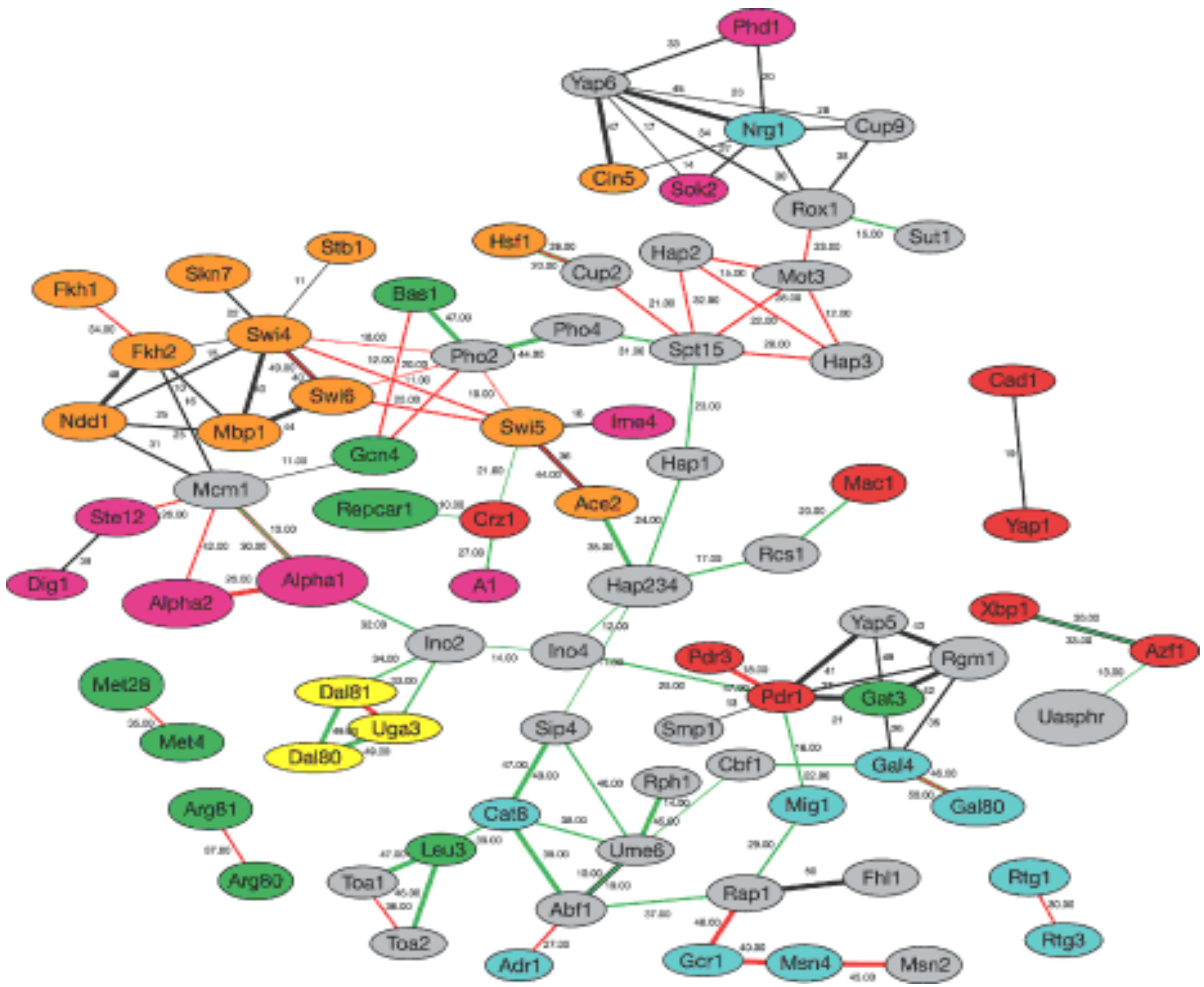
Road Network

Acyclic graph of an expression (DAG)

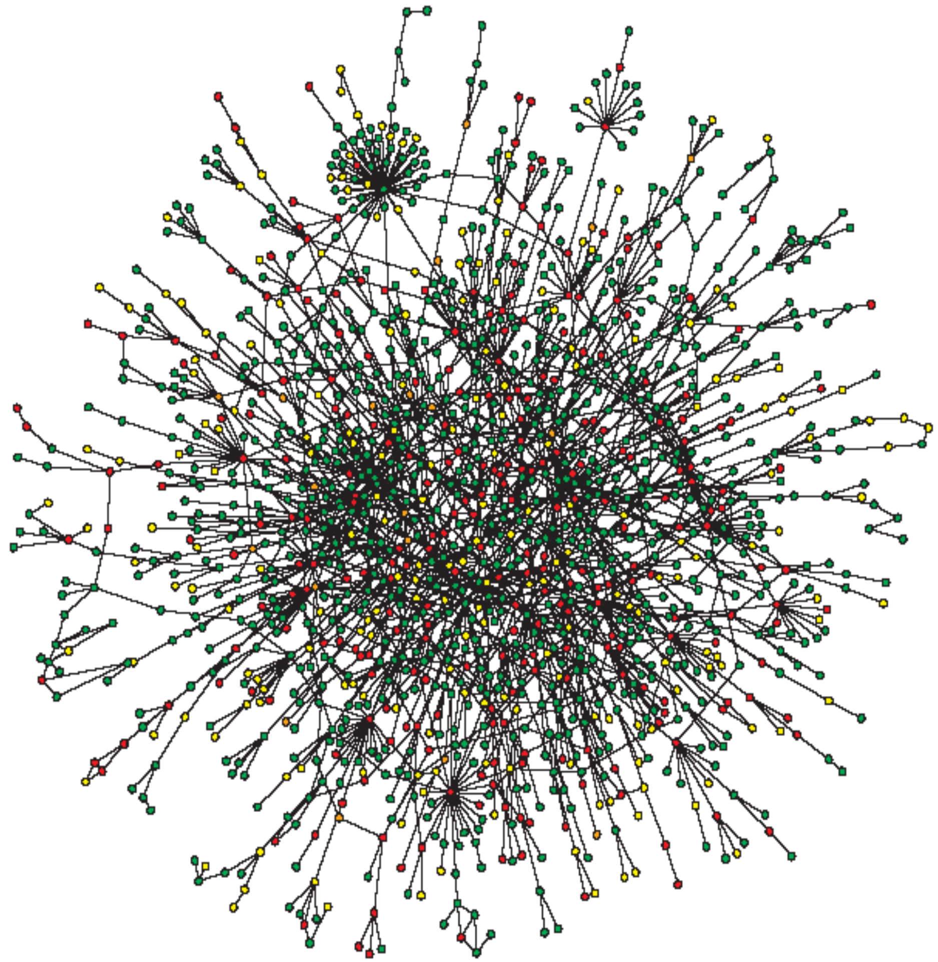


((a+b)*c+d/(b*(a+b))) * (a+b)*c

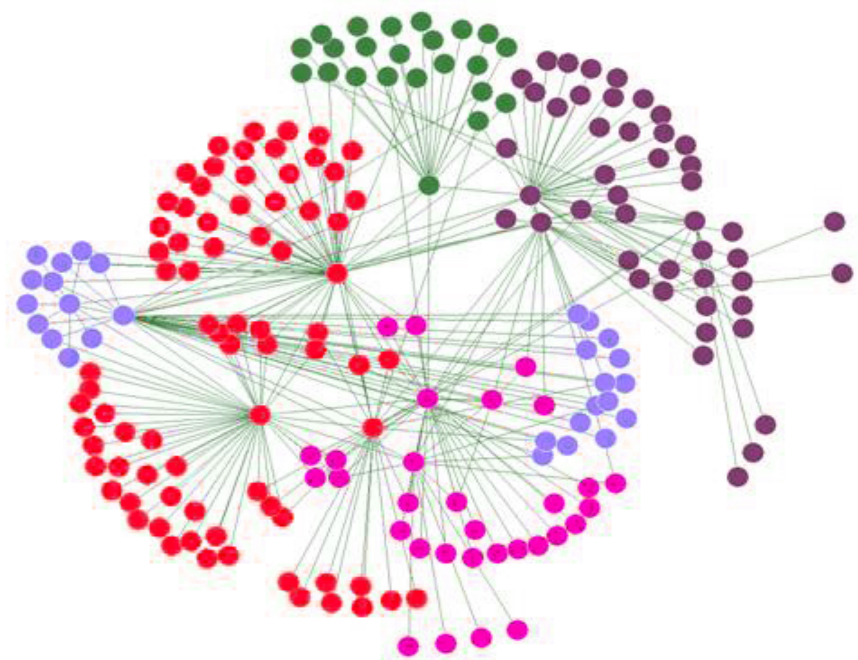
Gene regulation network in biology



Protein-protein interaction network (in yeast)



Social networks



Graph representations

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

Adjacency list

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

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

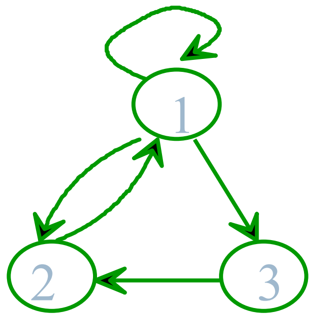
Adjacency matrix

using matrix operations

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

Other representations possible

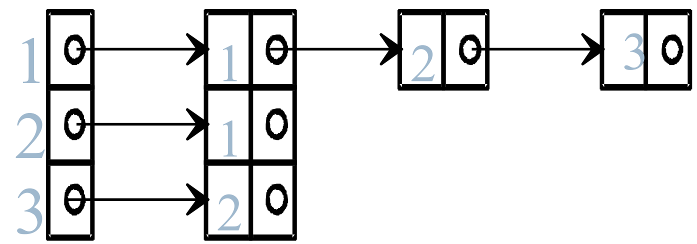
Adjacency lists

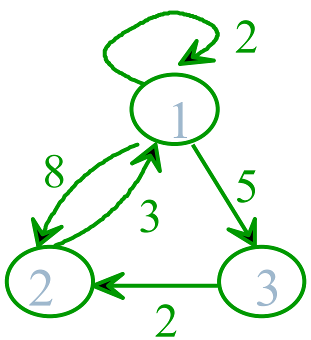


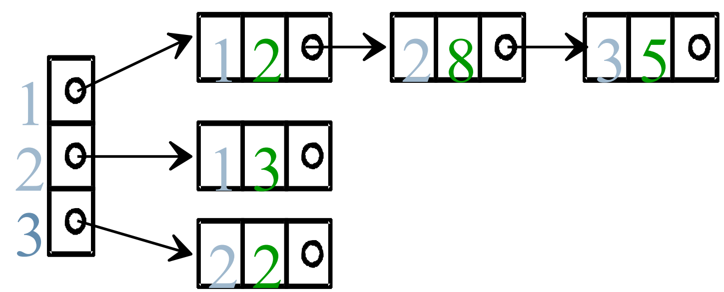
Lists of $A(s)$

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

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



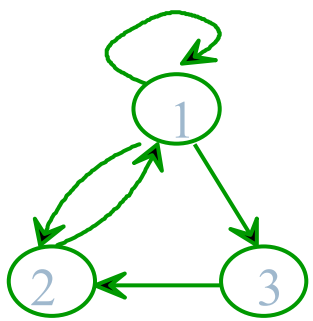




weight: $w : A \longrightarrow X$

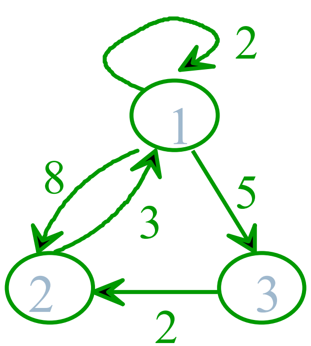
Adjacency matrix

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



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

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



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

weight: $w : A \longrightarrow X$

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
  - Graph coloring
  - Planarity testing
  - …

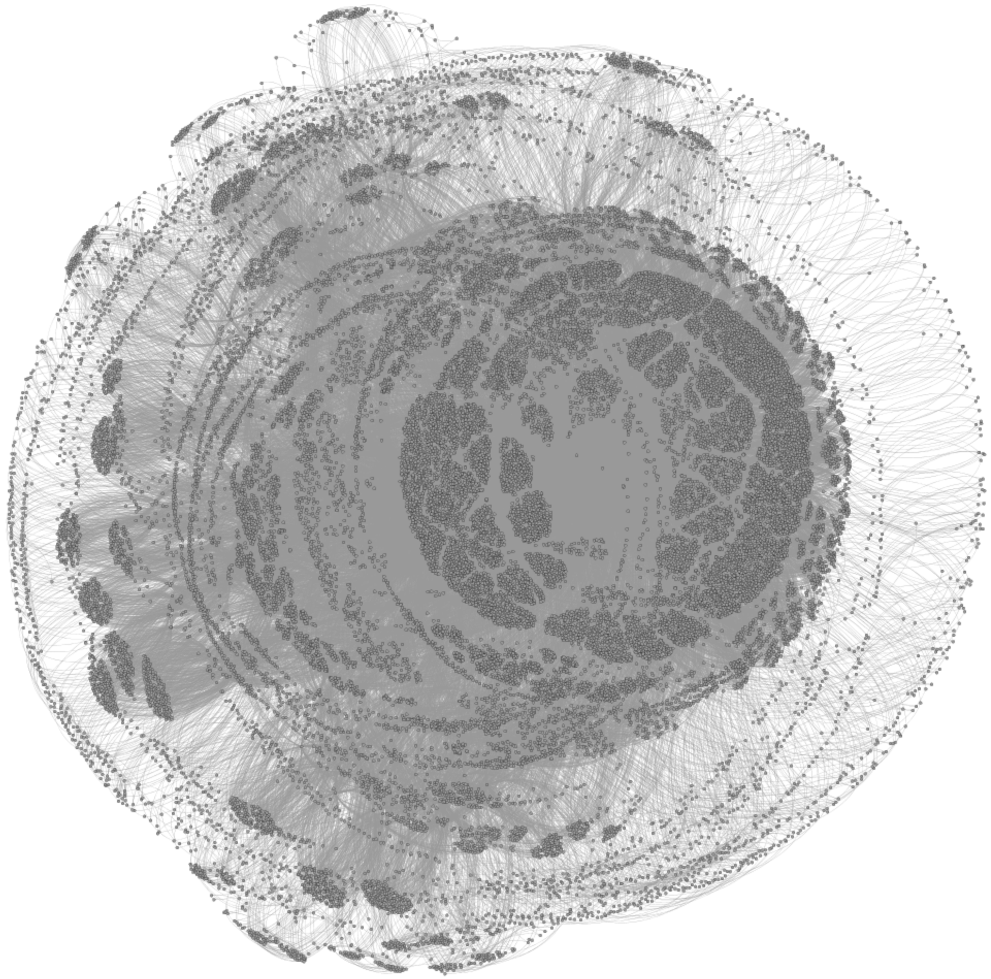
Shortest paths in graphs

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



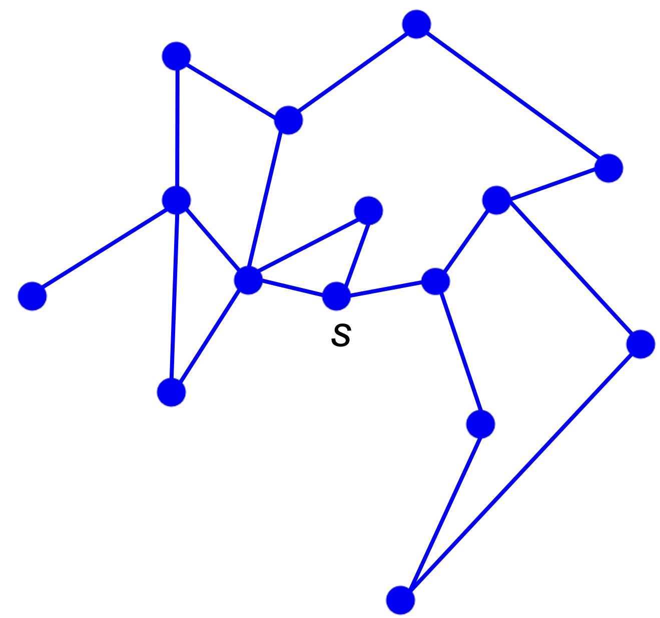
a subgraph (29,160 nodes) of the graph of Rubik's mini cube (2x2x2) configuraitons (3,674,160 nodes)

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

Breadth-first search (BFS)

Given a source node *s*,

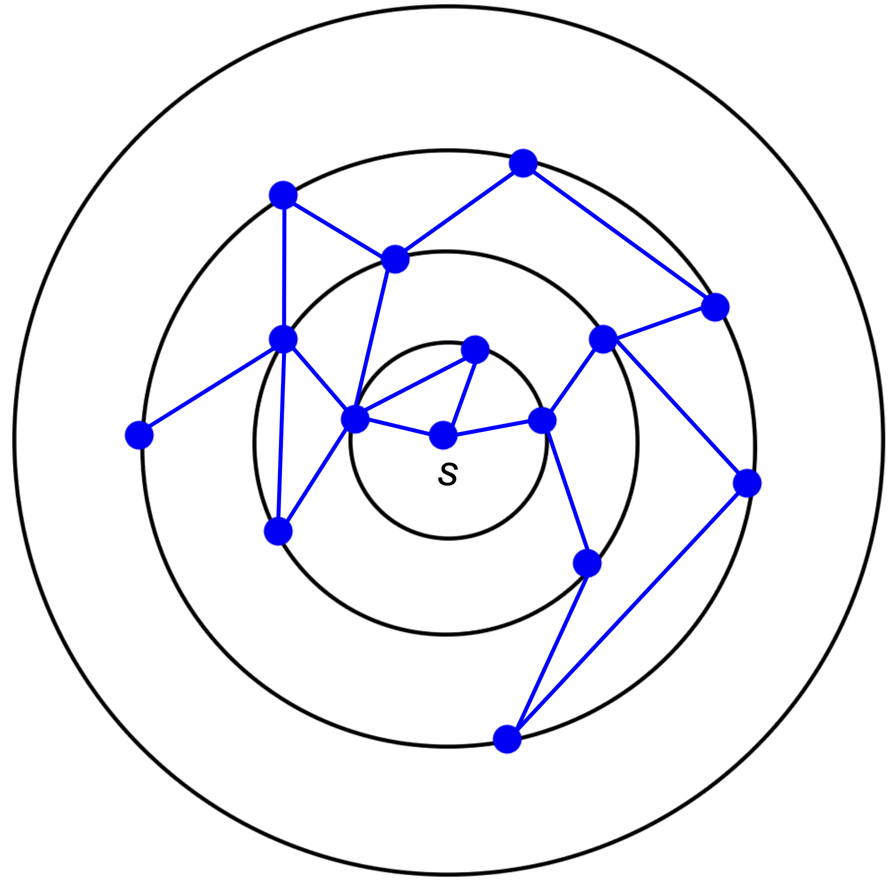
- "Discovers" all nodes reachable from *s*



Breadth-first search (BFS)

Given a source node *s*,

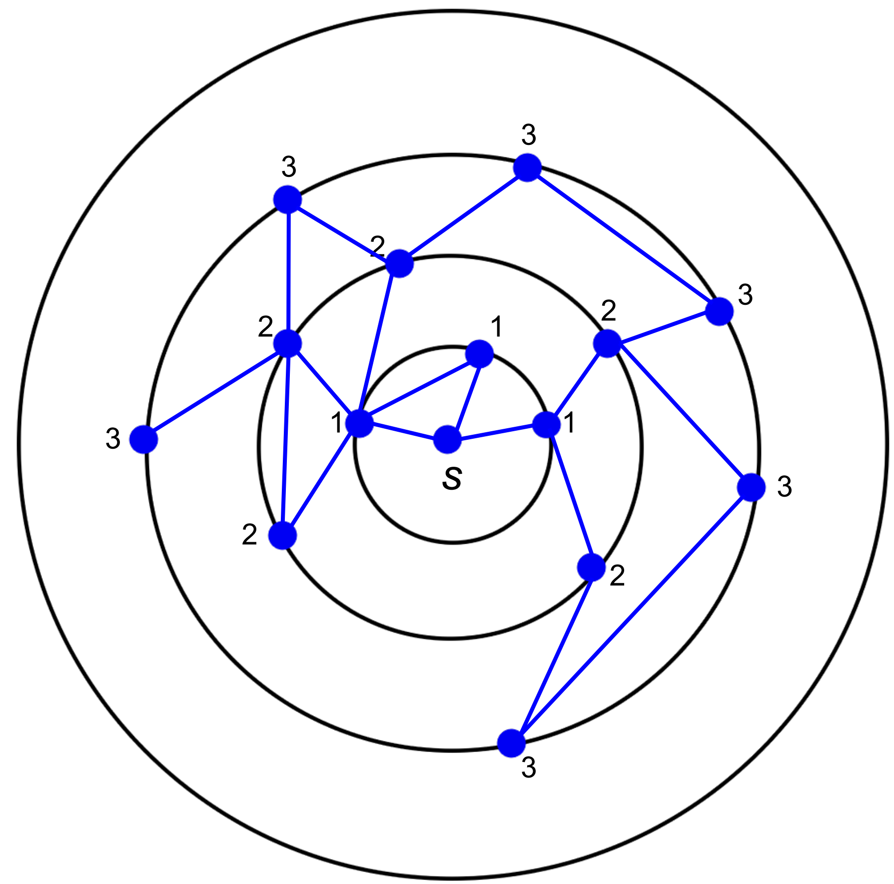
- "Discovers" all nodes reachable from *s*
- Proceeds by "concentric circles"
- Discovers all nodes at distance *d* from s before discovering any nodes at distance *d*+1



Breadth-first search (BFS)

Given a source node *s*,

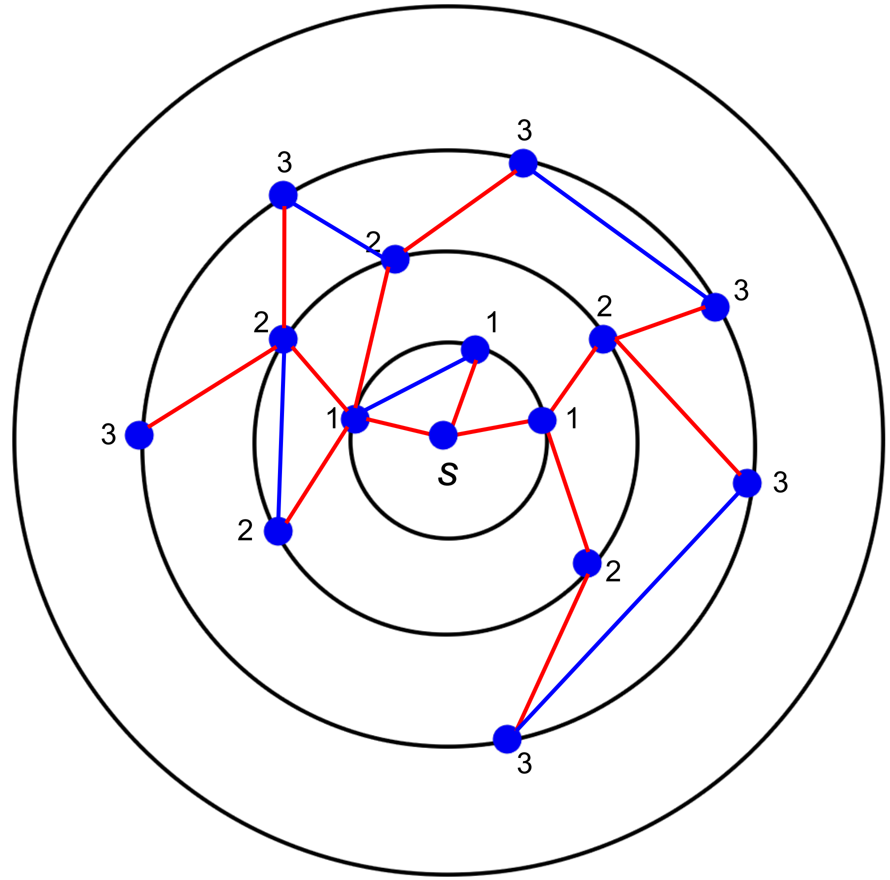
- "Discovers" all nodes reachable from *s*
- Proceeds by "concentric circles"
- Discovers all nodes at distance *d* from s before discovering any nodes at distance $d+1$
- Computes the distances from *s*
- Computes a *breadth-first tree* encoding one shortest path for each node



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

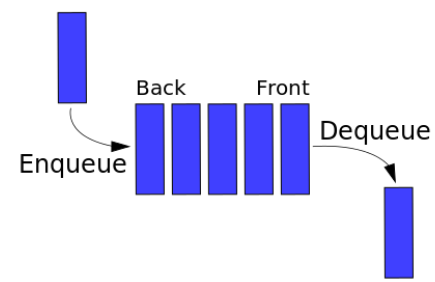


How it works?

- 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)
- 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.

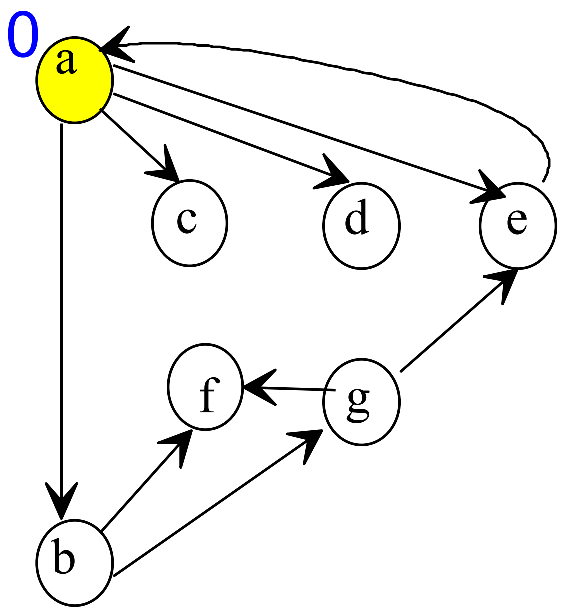
Breadth-first search (BFS)

```
procedure BFT (s node of V) ;
begin
for each 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) ;
        //s' becomes red
        }
end
```



wikipedia.org

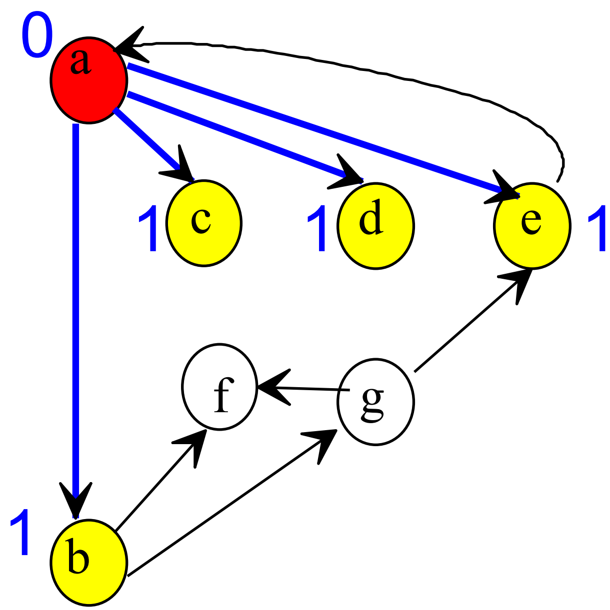
BFS: example



Queue : a

Order of traversal:

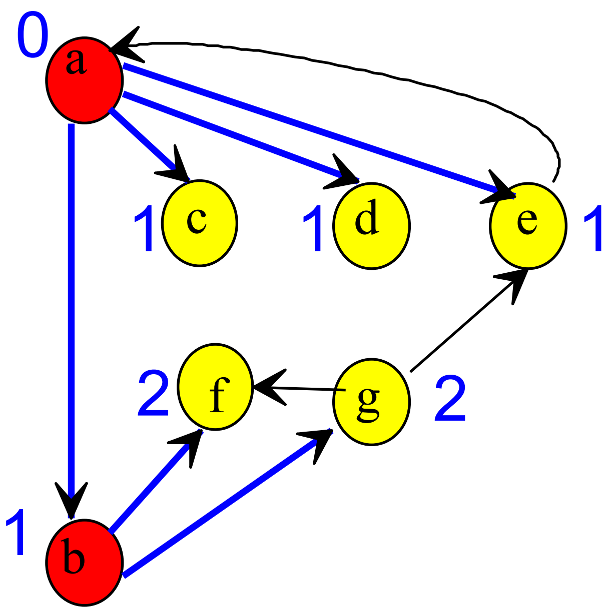
BFS: example



Queue : a b c d e

Order of traversal: a

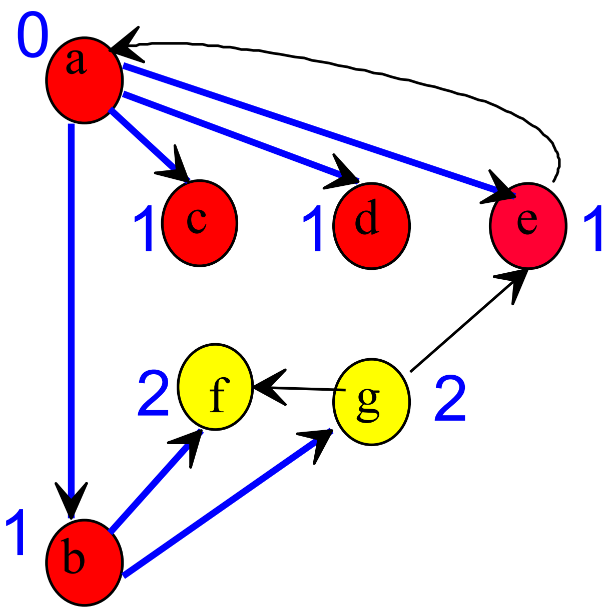
BFS: example



Queue : a b c d e f g

Order of traversal: a b

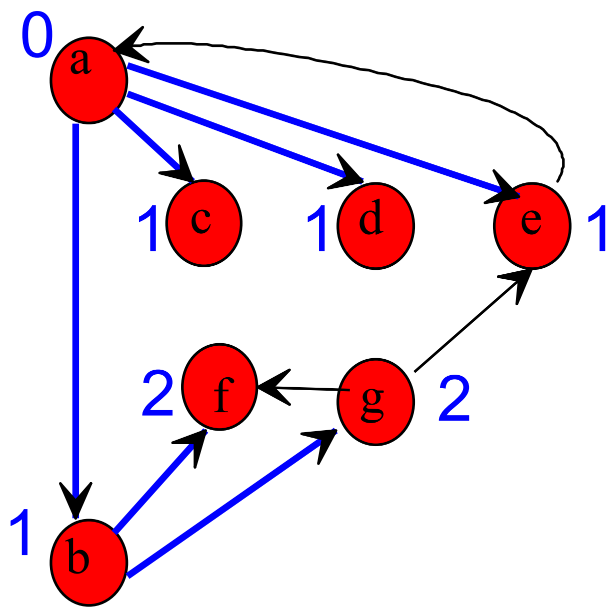
BFS: example



Queue : a b c d e f g

Order of traversal: a b c d e

BFS: example



Queue : a b c d e f g

Order of traversal: a b c d e f g

Questions

- 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 $d[v_r] \le d[v_1]+1$ and $d[v_i] \le d[v_{i+1}]$ 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

$d[v]=\delta(s,v)$: sketch of the proof

- by contradiction, let v be the closest to s node with $d[v]>\delta(s,v)$
- consider a shortest path from s to v, and let u be the node preceding v in this path
- $\delta(s,v)=\delta(s,u)+1$ (by properties of shortest paths)
- consider the moment when u was dequeued ($d[u]=\delta(s,u)$)
- if v was white then, we have $d[v]=\delta(s,v)$ $\Rightarrow$ *contradiction*
- if v was yellow then, it was visited earlier by exploring the successors of some w with $d[w]\le d[u]$. Then $d[v]=d[w]+1\le d[u]+1$ $\Rightarrow$ *contradiction*
- if v was red, then $d[v]\le d[u]$ $\Rightarrow$ *contradiction*

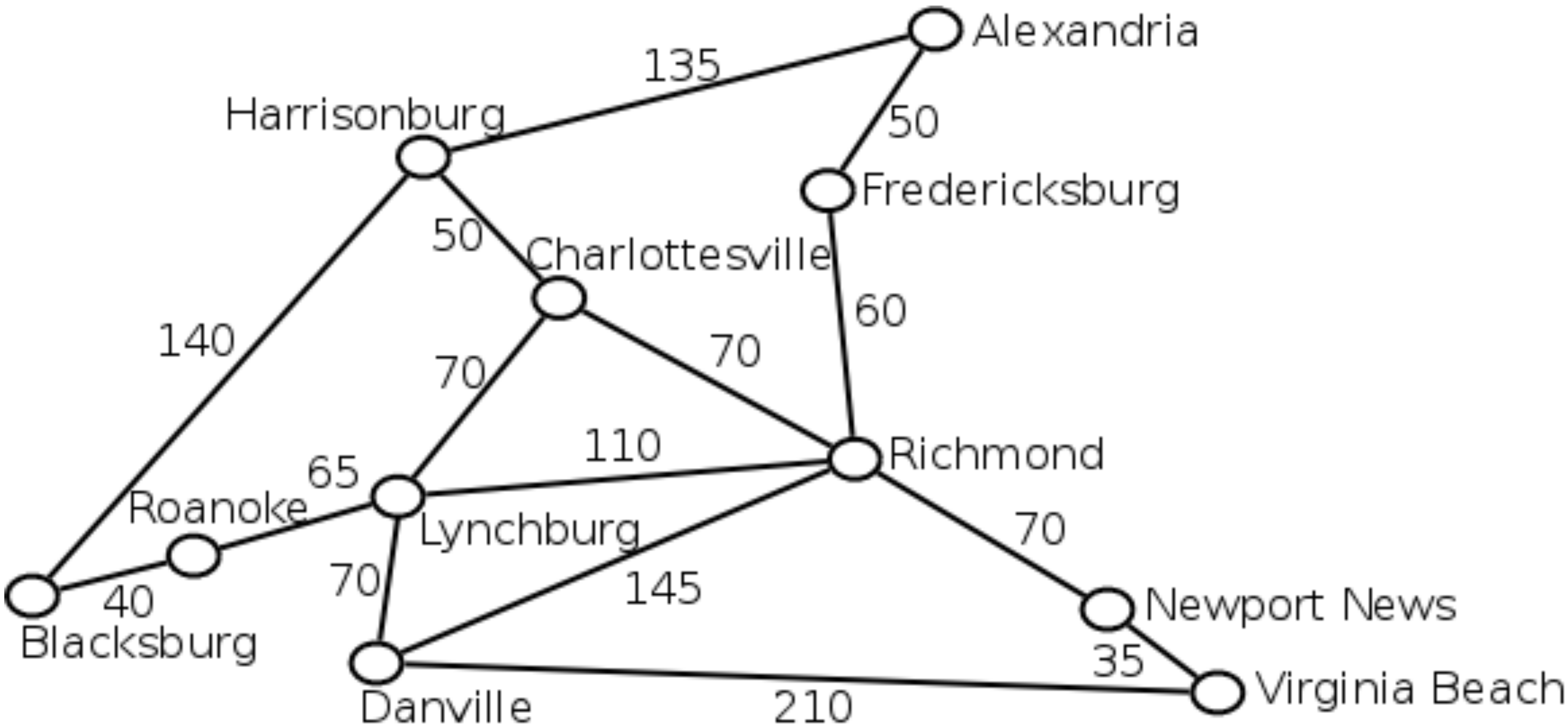
Space efficient BFS

- 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, Jan 2018]

Single-source shortest path: weighted case

Single-source shortest path: weighted case



Shortest path problem

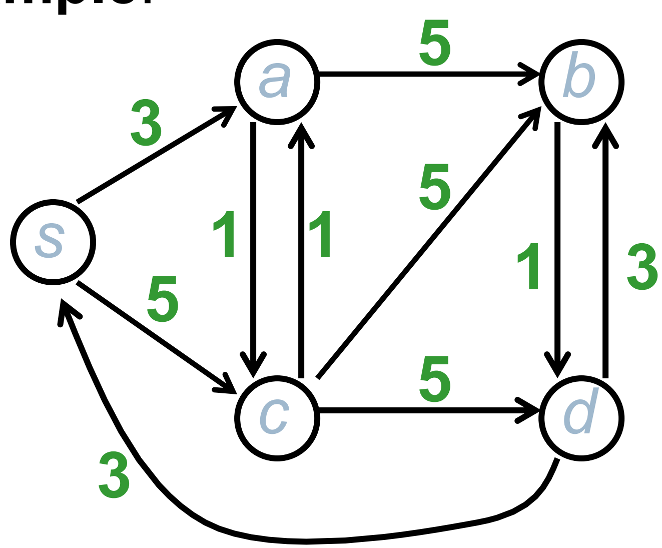
Weighted (directed or undirected) graph: $G = (V, E, w)$ where
$w : E \to \mathbf{R}$ (weight/cost)

Source : $s \in V$

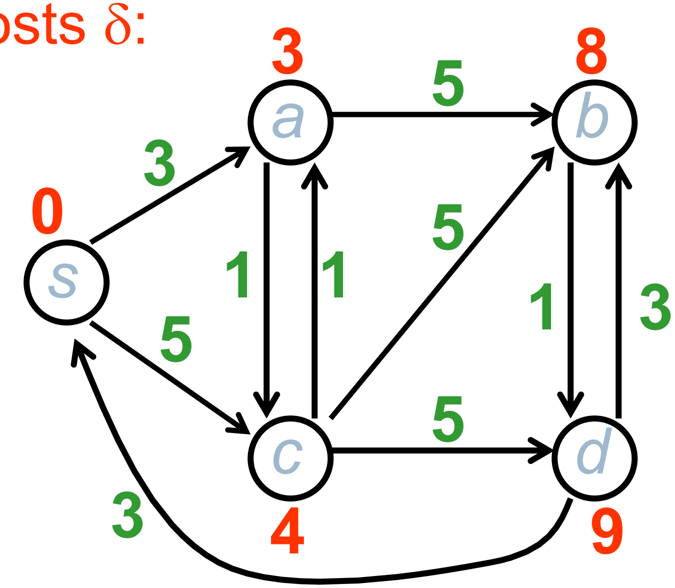
Problem: for all $t \in V$, compute

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

Example:

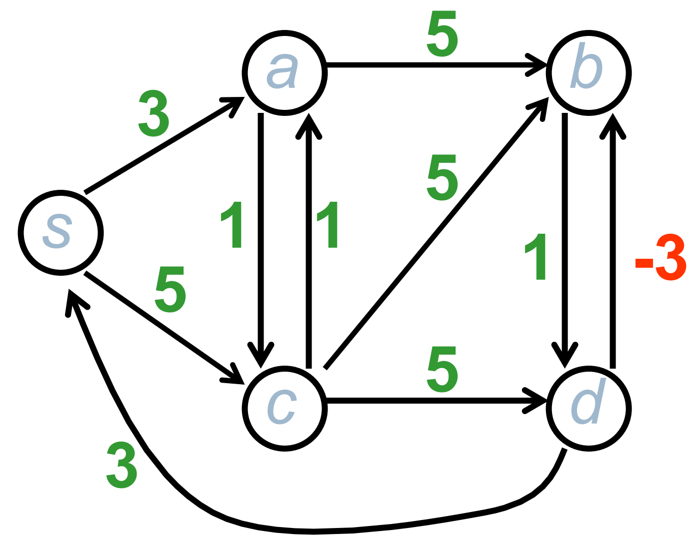


Costs δ:



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*

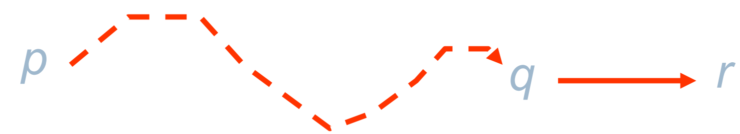


Proposition 2: if there exists a shortest path from *s* to *t*, then there exists one without a cycle

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

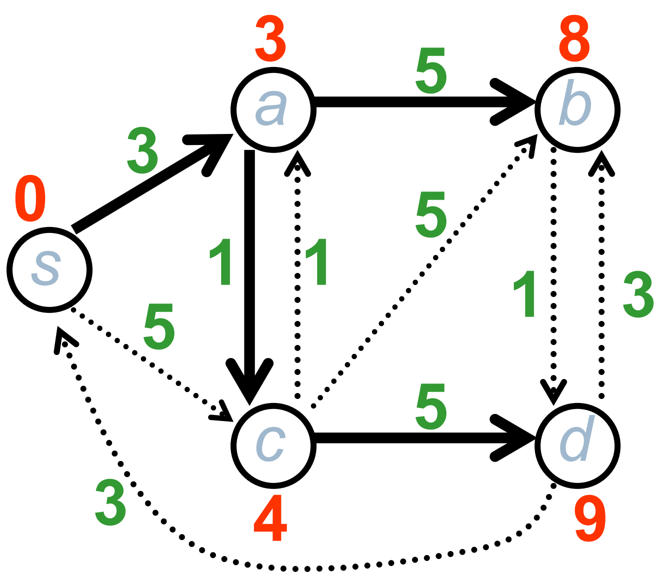
Main properties

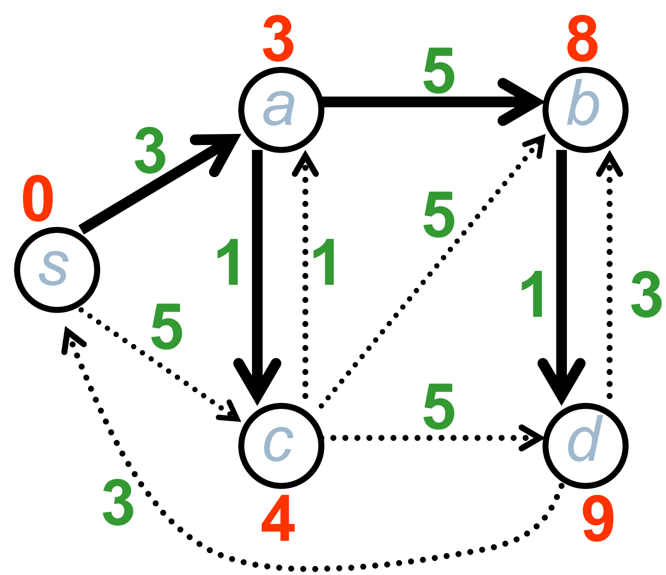
Property 1: $G = (V, E, w)$
let c be a shortest path from p to r
and q be the node preceding r in c.
Then $\delta(p, r) = \delta(p, q) + w(q, r)$.



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

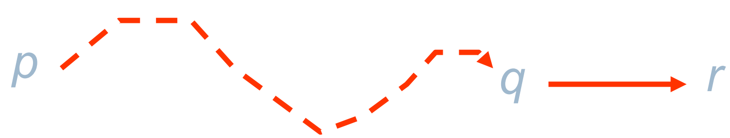
Shortest path tree: tree rooted at s representing shortest paths





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 $\delta(p, r) \le \delta(p, q) + w(q, r)$.



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 $d[t]$

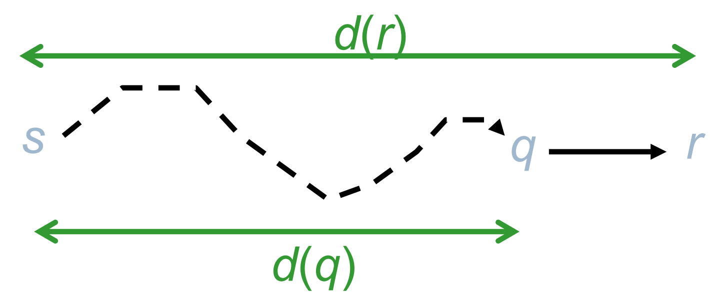
Initialization of d and π

INIT

for all $t \in V$ **do**

{ $d[t] = \infty$; $\pi[t]$ = **nil** }

$d[s] = 0$;



Relaxation of the edge (q, r)

RELAX(q, r)

if $d[q] + w(q, r) < d[r]$

then { $d[r] = d[q] + w(q, r)$; $\pi[r] = q$ }

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`**

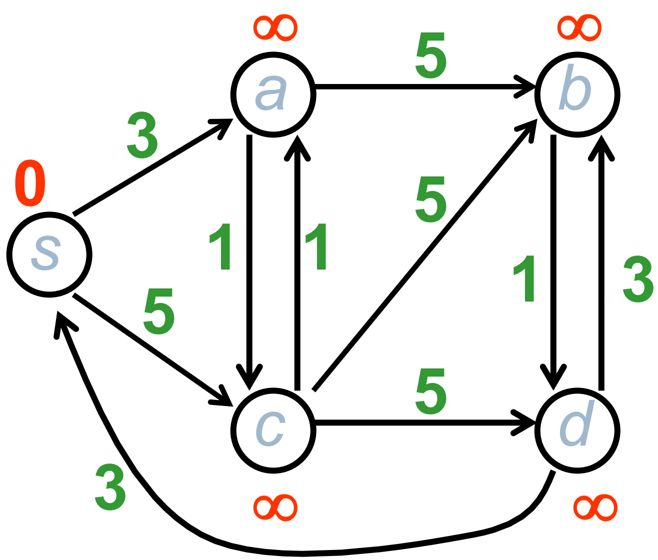
Dijkstra's algorithm

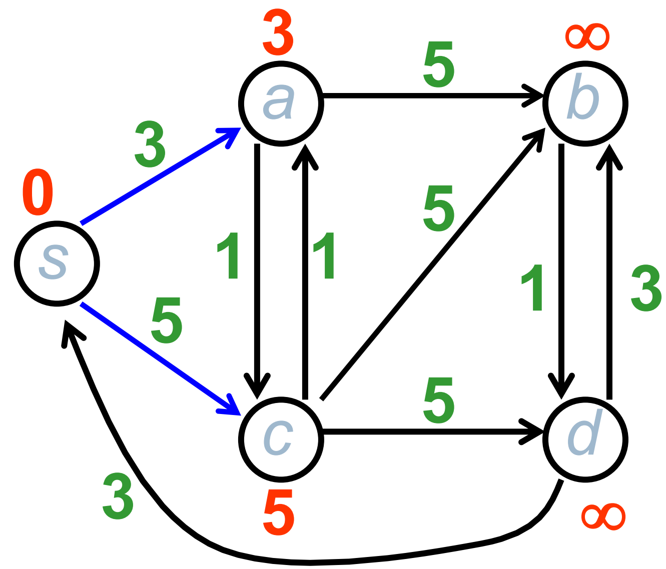
Assumption: $w(p, q) \geq 0$ for all edges (p, q)

```
begin
      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) ;
      }
end
```

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

Example





$S = \{\varnothing\}$

$Q = \{s, a, b, c, d\}$

$\pi[s]$ = nil

$\pi[a]$ = nil

$\pi[b]$ = nil

$\pi[c]$ = nil

$\pi[d]$ = nil

$S = \{s\}$

$Q = \{a, b, c, d\}$

$\pi[s]$ = nil

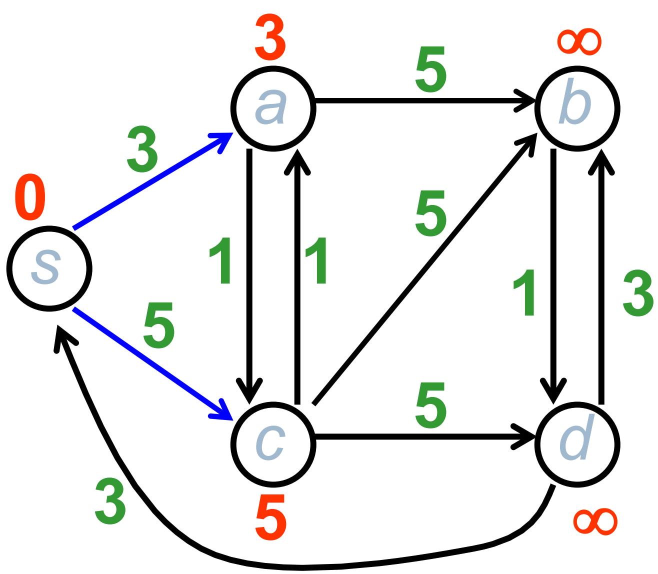
$\pi[a] = s$

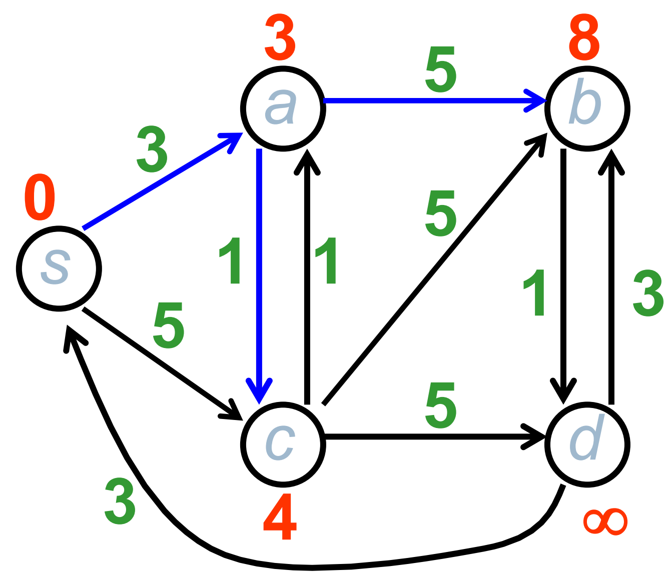
$\pi[b]$ = nil

$\pi[c] = s$

$\pi[d]$ = nil

Example (cont)

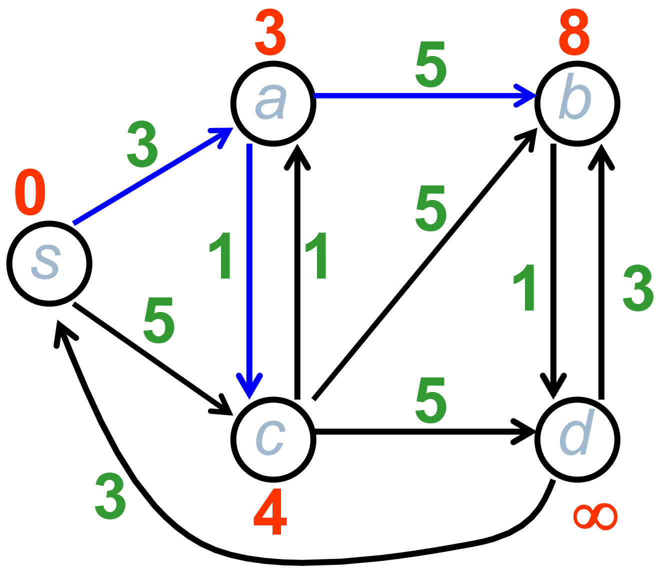


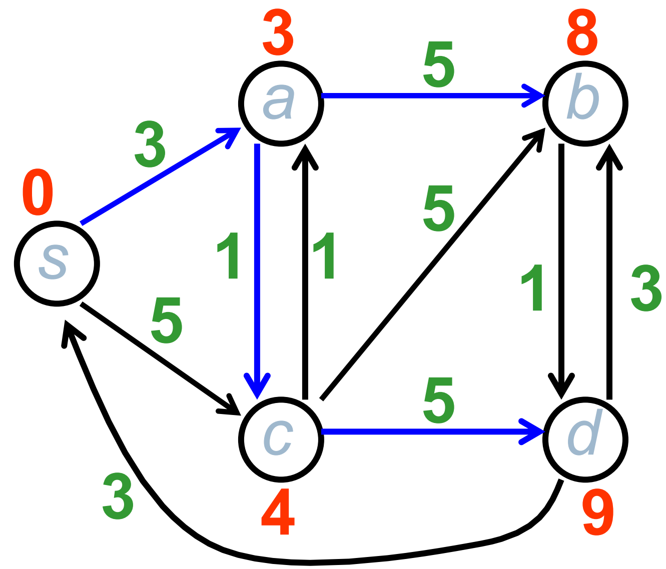


$S = \{s\}$
$Q = \{ a, b, c, d \}$
$\pi[s]$ = nil
$\pi[a] = s$
$\pi[b]$ = nil
$\pi[c] = s$
$\pi[d]$ = nil

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

Example (cont)

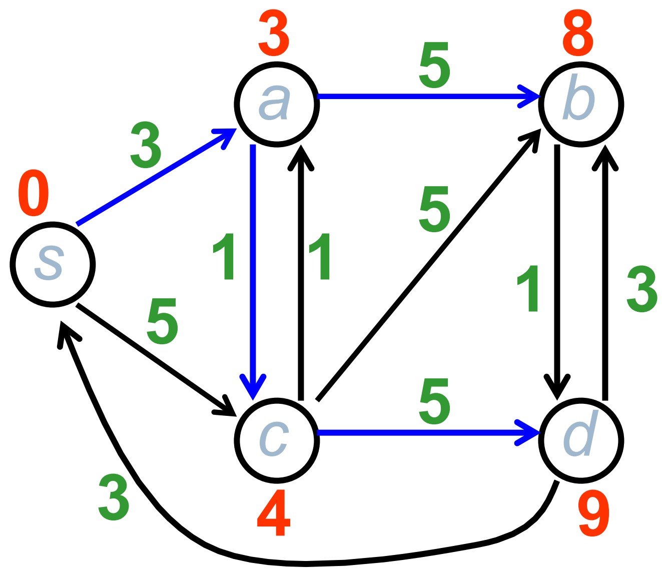


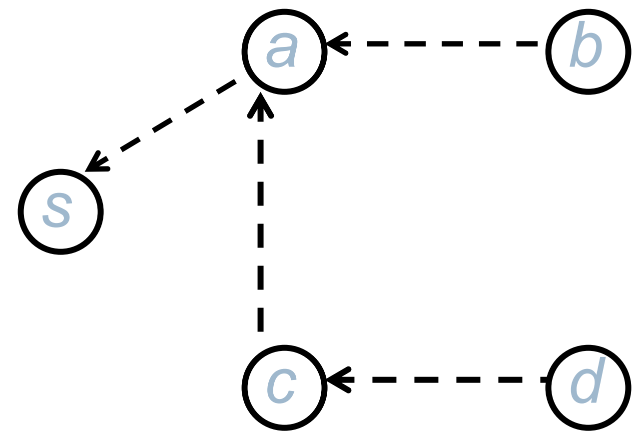


$S = \{s, a\}$
$Q = \{ b, c, d \}$
$\pi[s]$ = nil
$\pi[a] = s$
$\pi[b] = a$
$\pi[c] = a$
$\pi[d]$ = nil

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

Example (cont)





S = {s, a, c}

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

$\pi[s]$ = nil

$\pi[a] = s$

$\pi[b] = a$

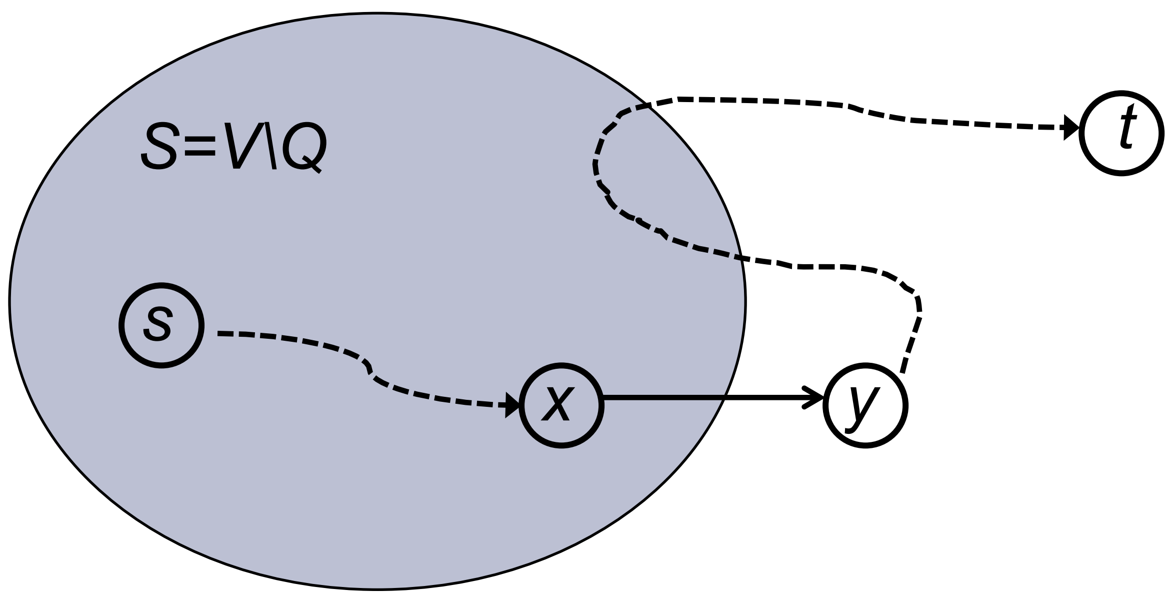
$\pi[c] = a$

$\pi[d] = c$

Correctness of Dijkstra's algorithm

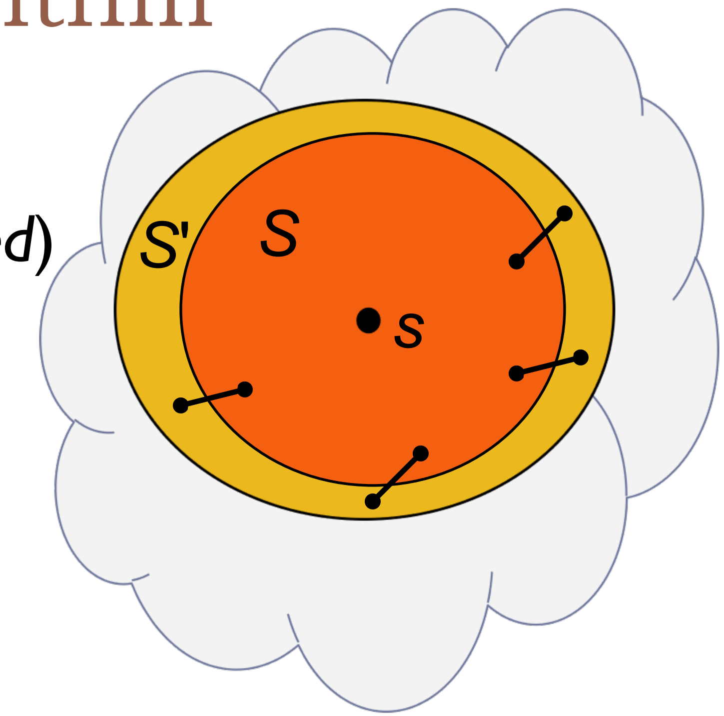
Proposition : After the execution of Dijkstra's algorithm on a graph $G = (V, E, w)$, $d[t] = \delta(s, t)$ for all $t \in V$.

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



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*

Implementation

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)

$\Rightarrow$ (min-)priority queue

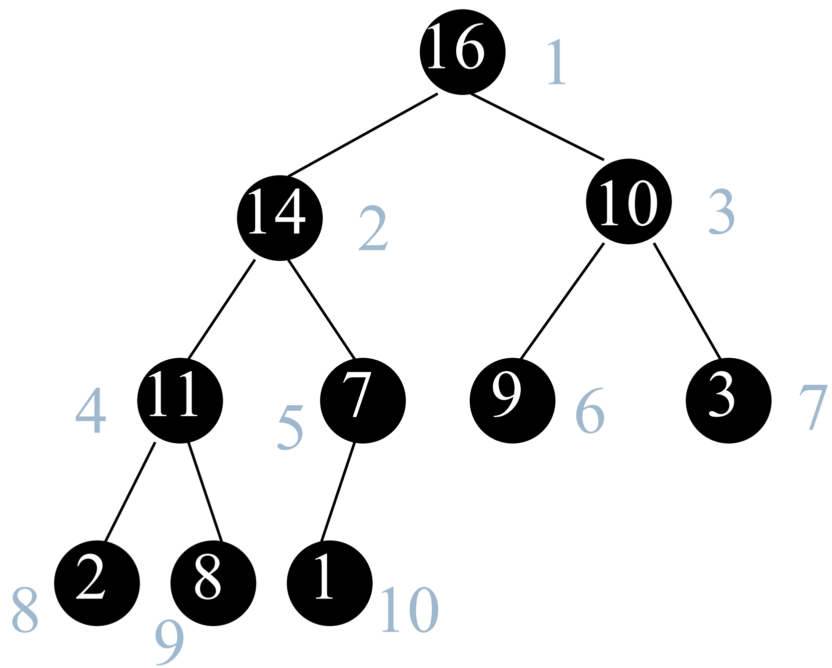
Priority Queues

- (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

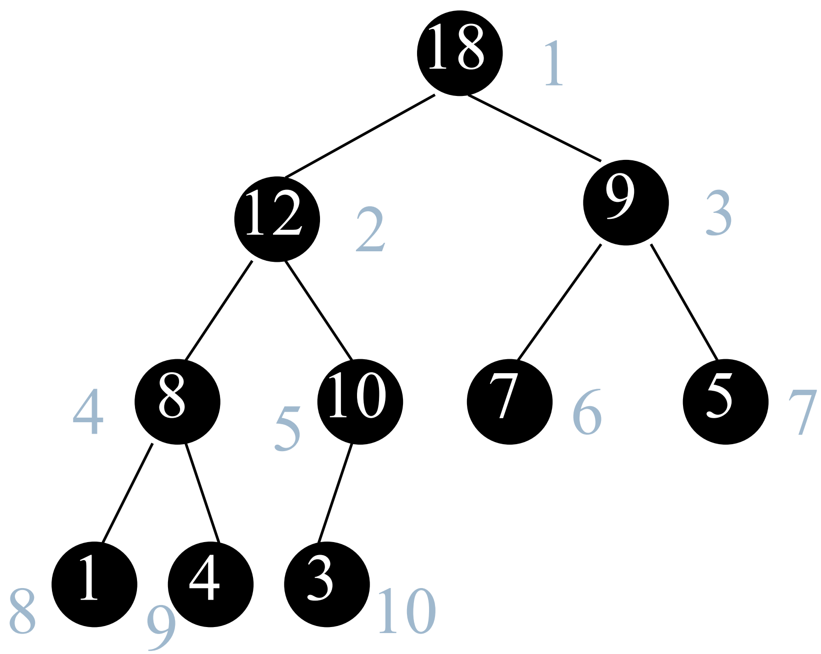
Binary Heaps

- Binary heap:
  - a **binary tree** that is
  - **complete**: every level except possibly the bottom one is completely filled and the leaves in the bottom level are as far left as possible
  - satisfies the **(max-)heap property**: the key stored in every node is greater than or equal to the keys stored in its children
    - If the key at each node is smaller than or equal to the keys of its children, then we have a **min-heap**

Binary (max-)heap: example



Binary heaps stored in arrays



Due to their regular structure, binary heaps are easily stored in arrays

Given index i of a node,

- the index of its parent is $\lfloor i / 2 \rfloor$
- the indices of its children are $2i$ and $2i+1$

18	12	9	8	10	7	5	1	4	3	*element key A*[i]
1	2	3	4	5	6	7	8	9	10	*index* i

Binary heaps: some properties

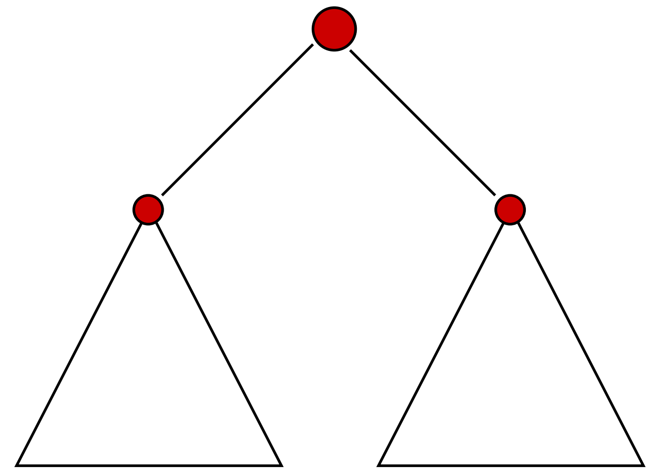
- The height of a heap is $\lfloor \log(n) \rfloor$

- Not every array represents a heap

- In a max-heap, the largest element is at the root and the smallest element is in a leaf

Heapify

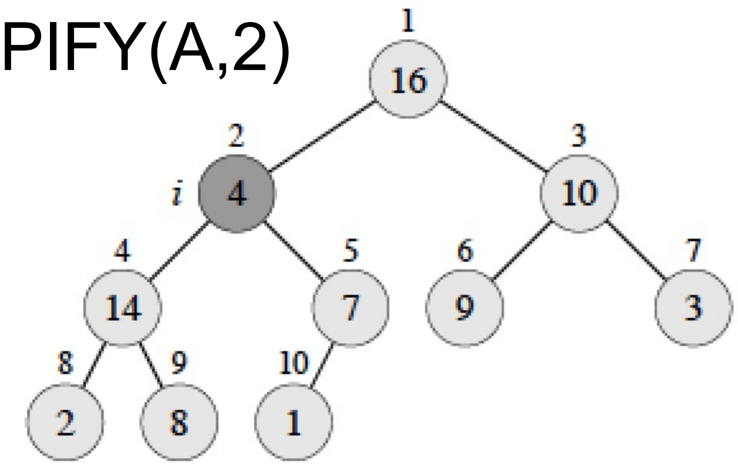
- Assume that node i violates the heap property, but the children nodes $2i$ and $2i+1$ (if exist) are heaps.



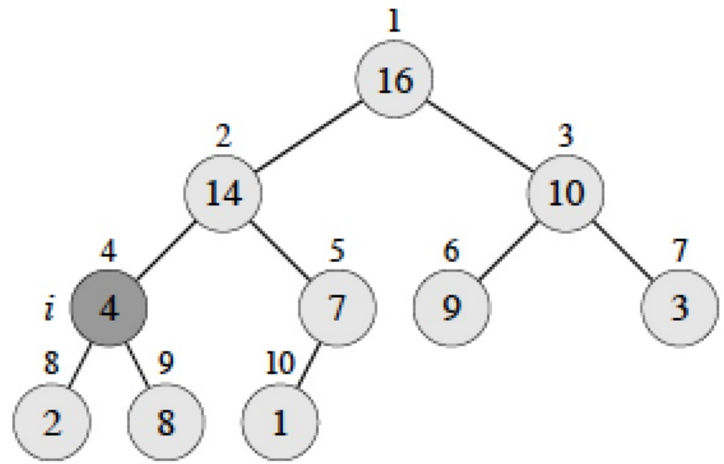
```
HEAPIFY(A,i)
  if A[2i]>A[i] or A[2i+1]>A[i] then
      if A[2i+1]>A[2i] then
            exchange A[i] and A[2i+1];
            HEAPIFY(A, 2i+1)
      else
            exchange A[i] and A[2i];
            HEAPIFY(A, 2i)
      end
  end
```

Heapify: example

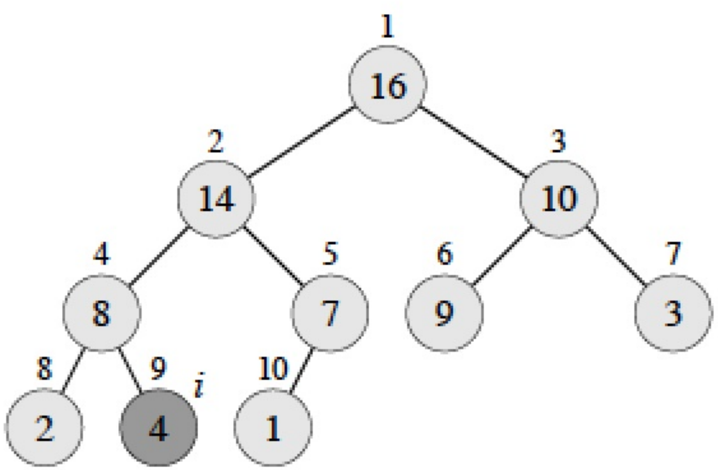
HEAPIFY(A,2)



(a)



(b)



(c)

Building a binary heap

- Given an array A[1..n], build a binary heap for array elements

BUILD-HEAP(A,n)

for i=$\lfloor n/2 \rfloor$ **downto** 1 **do** HEAPIFY(A,i);

- *Exercise*: build the heap for A=[4,1,3,2,16,9,10,14,8,7]

BUILD-HEAP: complexity

- Straightforward estimation $O(n \cdot \log(n))$

- Refined analysis:
  - Cost of a call to HEAPIFY at a node depends on the height, *h*, of the node – *O*(*h*).
  - Height of most nodes smaller $\lfloor \log(n) \rfloor$
  - Height of nodes *h* ranges from 0 to $\lfloor \log(n) \rfloor$
  - number of nodes of height *h* is at most $\lceil n / 2^{h+1} \rceil$?

Heap Characteristics

- Height = $\lfloor \log n \rfloor$
- Number of leaves = $\lceil n/2 \rceil$
- Number of nodes of height $h \leq \lceil n/2^{h+1} \rceil$

- *Proof by induction*:
  - remove all leaves from the heap
  - there remains $n - \lceil n/2 \rceil = \lfloor n/2 \rfloor$ nodes
  - the height of each node is decremented by 1
  - nb of nodes of height h-1 is (by induction) $\lceil \lfloor n/2 \rfloor / 2^h \rceil \leq \lceil n/2^{h+1} \rceil$

Tighter bound for BUILD-HEAP: $O(n)$

time of BUILD-HEAP is $\sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h}\right)$

note that $\sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h} \leq \sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2$

therefore the time is $O(n)$

Priority Queue

- MAX(A): return the heap root
- EXTRACT-MAX(A):

Priority Queue

- MAX(A): return the heap root
- EXTRACT-MAX(A): exchange A[1] and A[n], discard element n, and apply HEAPIFY(A,1)
- INCREASE-KEY(A,i,k):

Priority Queue

- MAX(A): return the heap root
- EXTRACT-MAX(A): exchange A[1] and A[n], discard element n, and apply HEAPIFY(A,1)
- INCREASE-KEY(A,i,k):

  A[i]←k;

  while A[$\lfloor i/2 \rfloor$]<A[i] **do**

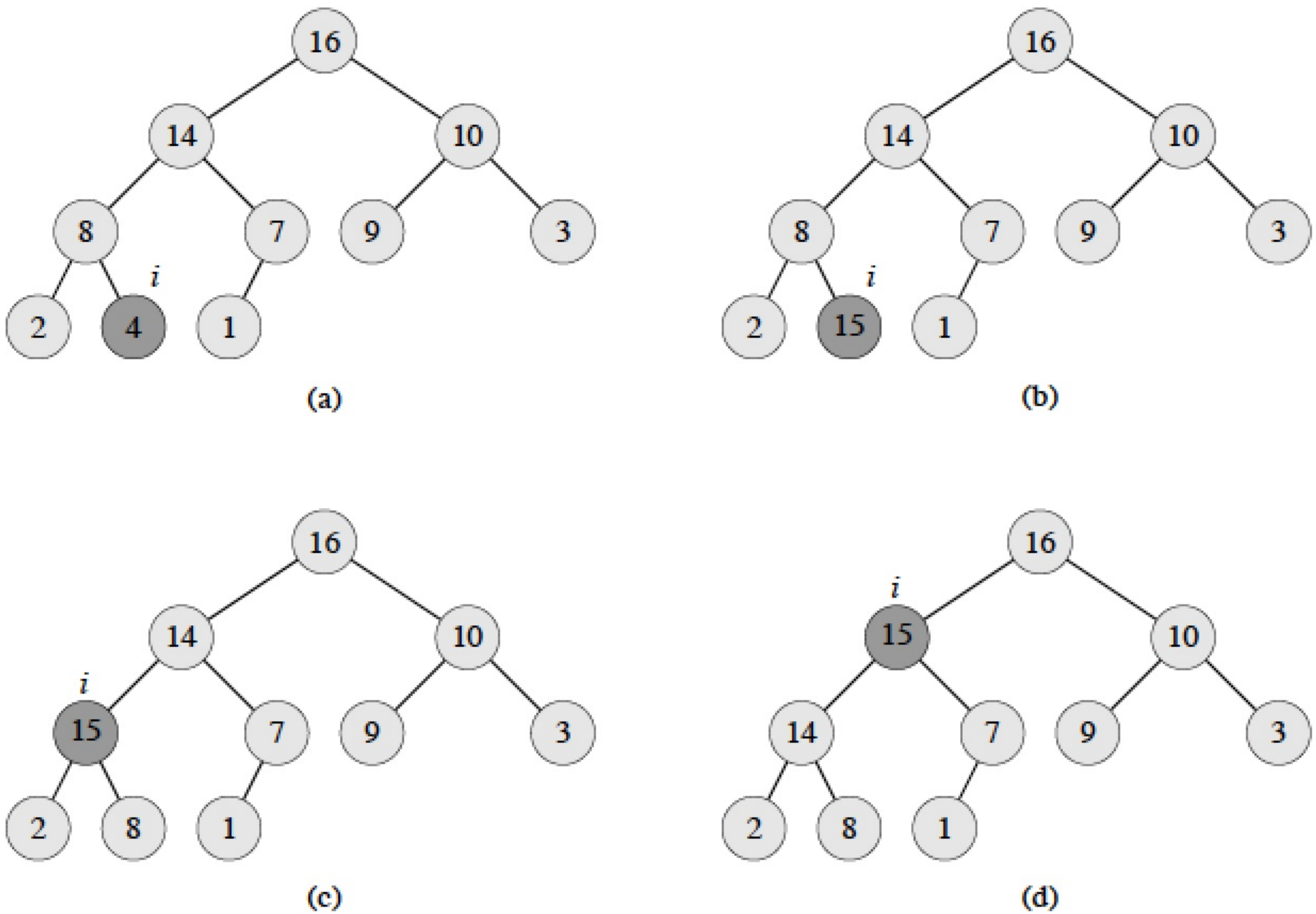
  exchange A[$\lfloor i/2 \rfloor$] and A[i];

  i←parent(i)

  end
- INSERT(A,i): insert a new leaf n+1 with key -∞; call INCREASE-KEY(A,n+1,k)

INCREASE-KEY: example

INCREASE-KEY(A,9,15)



Priority Queues: time bounds

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

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 $O(n^2)$

With adjacency lists

Q : priority queue

if implemented by binary heaps:

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

n operations **MIN$_d$** : $O(n \cdot \log n)$

m operations **RELAX** : $O(m \cdot \log n)$

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

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

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

as decreasing the key takes O(1) amortized