



# ***Design and analysis of algorithms***

## ***Lecture 16 & 17***

Edyta Szymańska

edyta@cc.gatech.edu

# *Binary heap data structure*

- ⑥ nearly complete binary tree (last level may not be complete) stored in an array

# Binary heap data structure

- ⑥ nearly complete binary tree (last level may not be complete) stored in an array
- ⑥ heap property:  $A[\text{Parent}(i)] \geq A[i]$

# Binary heap data structure

- ⑥ nearly complete binary tree (last level may not be complete) stored in an array
- ⑥ heap property:  $A[\text{Parent}(i)] \geq A[i]$
- ⑥ How to find the maximum element of a heap?
  - △ at root (proof: induction + transitivity of  $\leq$ )
  - △ extracting the maximum element takes  $\Theta(1)$ (returning the root)

# Binary heap data structure

- ⑥ nearly complete binary tree (last level may not be complete) stored in an array
- ⑥ heap property:  $A[\text{Parent}(i)] \geq A[i]$
- ⑥ How to find the maximum element of a heap?
  - △ at root (proof: induction + transitivity of  $\leq$ )
  - △ extracting the maximum element takes  $\Theta(1)$ (returning the root)

Example:

10	7	9	4	7	5	2	2	1	6
----	---	---	---	---	---	---	---	---	---

# *Binary heap*



Storage of a heap in an array:

# Binary heap

Storage of a heap in an array:  
children of  $i$  are  $2i$  and  $2i + 1$

# Binary heap

Storage of a heap in an array:

children of  $i$  are  $2i$  and  $2i + 1$

It is fast to compute using binary representation

# Binary heap

Storage of a heap in an array:

children of  $i$  are  $2i$  and  $2i + 1$

It is fast to compute using binary representation

Example:

children of  $A[3] = 9$  are  $A[6] = 5$  and  $A[7] = 2$

# Operations on heaps

## I. Extracting the maximum element

HEAP\_EXTRACT\_MAX( $A$ )

- ⑥ remove  $A[1]$
- ⑥  $A[1] := A[n]$
- ⑥  $n := n - 1$
- ⑥ HEAPIFY( $A, 1, n$ ) {*maintain the heap*}

# Operations on heaps

## I. Extracting the maximum element

HEAP\_EXTRACT\_MAX( $A$ )

- ⑥ remove  $A[1]$
- ⑥  $A[1] := A[n]$
- ⑥  $n := n - 1$
- ⑥ HEAPIFY( $A, 1, n$ ) {*maintain the heap*}

## II. Maintaining the heap

# Operations on heaps

## II. Maintaining the heap

HEAPIFY( $A, i, n$ )

if  $2i \leq n$  and  $A[2i] > A[i]$  {makes  $i$ 's subtree a heap}

⌚ then  $largest := 2i$

⌚ else  $largest := i$

if  $2i + 1 \leq n$  and  $A[2i + 1] > A[largest]$

⌚ then  $largest := 2i + 1$

if  $largest \neq i$

⌚ then  $A[i] := A[largest]$

⌚ HEAPIFY( $A, largest, n$ )

# Operations on heaps

## II. Maintaining the heap

HEAPIFY( $A, i, n$ )

if  $2i \leq n$  and  $A[2i] > A[i]$  {makes  $i$ 's subtree a heap}

⌚ then  $largest := 2i$

⌚ else  $largest := i$

if  $2i + 1 \leq n$  and  $A[2i + 1] > A[largest]$

⌚ then  $largest := 2i + 1$

if  $largest \neq i$

⌚ then  $A[i] := A[largest]$

⌚ HEAPIFY( $A, largest, n$ )

Running time:  $O(\log n)$ , intuition: must process all levels of the binary tree

# Operations on heaps

## II. Maintaining the heap

HEAPIFY( $A, i, n$ )

if  $2i \leq n$  and  $A[2i] > A[i]$  {makes  $i$ 's subtree a heap}

⌚ then  $largest := 2i$

⌚ else  $largest := i$

if  $2i + 1 \leq n$  and  $A[2i + 1] > A[largest]$

⌚ then  $largest := 2i + 1$

if  $largest \neq i$

⌚ then  $A[i] := A[largest]$

⌚ HEAPIFY( $A, largest, n$ )

Running time:  $O(\log n)$ , intuition: must process all levels of the binary tree

# Operations on heaps

## III. Building a heap from an unsorted array

BUILD\_HEAP( $A$ )

from  $i := n$  downto 1 do

- ⑥ HEAPIFY( $A, i, n$ ) {*maintain the heap rooted at  $i$* }

# Operations on heaps

## III. Building a heap from an unsorted array

BUILD\_HEAP( $A$ )

from  $i := n$  downto 1 do

    HEAPIFY( $A, i, n$ ) {*maintain the heap rooted at  $i$* }

Running time :  $O(n \log n)$ , but a more careful analysis gives  $O(n)$

# Operations on heaps

Running time :  $O(n \log n)$ , but a more careful analysis gives  $O(n)$

The cost of HEAPIFY is proportional to the number of levels visited.

# Operations on heaps

Running time :  $O(n \log n)$ , but a more careful analysis gives  $O(n)$

The cost of HEAPIFY is proportional to the number of levels visited.

Assume  $n = 2^k - 1$ , where  $k$  is the number of levels in the complete binary tree.

# Operations on heaps

Running time :  $O(n \log n)$ , but a more careful analysis gives  $O(n)$

The cost of HEAPIFY is proportional to the number of levels visited.

Assume  $n = 2^k - 1$ , where  $k$  is the number of levels in the complete binary tree.

on level  $i$  (bottom up) we have  $\frac{n+1}{2^i}$  nodes and for every such node HEAPIFY( $A, i, n$ ) visits at most  $i$  levels

# Operations on heaps

Running time :  $O(n \log n)$ , but a more careful analysis gives  $O(n)$

The cost of HEAPIFY is proportional to the number of levels visited.

Assume  $n = 2^k - 1$ , where  $k$  is the number of levels in the complete binary tree.

on level  $i$  (bottom up) we have  $\frac{n+1}{2^i}$  nodes and for every such node HEAPIFY( $A, i, n$ ) visits at most  $i$  levels

Thus

$$T(n) = \sum_{i=0}^{\log(n+1)} \frac{i}{2^i} (n+1) = (n+1) \sum_{i=0}^{\log(n+1)} \frac{i}{2^i} \leq 2(n+1) = O(n)$$

using  $\sum_{i=0}^{\infty} \frac{i}{2^i} = \frac{\frac{1}{2}}{(1-\frac{1}{2})^2} = 2$

# *Applications and generalization of a binary heap*

Heapsort - sorting using heaps (optional reading).

# Applications and generalization of a binary heap

Heapsort - sorting using heaps (optional reading).  
Generalization:  $d$ -ary heap defined as a  $d$ -ary almost complete tree (every node has  $d$  children.) with the same heap property.

# ***Applications and generalization of a binary heap***

Heapsort - sorting using heaps (optional reading).

Generalization: *d*-ary heap defined as a *d*-ary almost complete tree (every node has *d* children.) with the same heap property.

Fibonacci heap: consists of min-heap-ordered trees

# Heap implementation of priority queue

A heap with MIN at root ( $A[\text{Parent}(i)] \leq A[i]$ ) makes excellent priority queue,

# Heap implementation of priority queue

A heap with MIN at root ( $A[\text{Parent}(i)] \leq A[i]$ ) makes excellent priority queue, good compromise between fast INSERTION and slow EXTRACT\_MIN, both  $O(\log n)$

# Heap implementation of priority queue

A heap with MIN at root ( $A[\text{Parent}(i)] \leq A[i]$ ) makes excellent priority queue, good compromise between fast INSERTION and slow EXTRACT\_MIN, both  $O(\log n)$

HEAP\_INSERT( $A, key$ )

$n := n + 1$

$i := n$

while  $i > 1$  and  $A[\lfloor \frac{i}{2} \rfloor] > key$   $\{O(\log n)\}$

do  $A[i] := A[\lfloor \frac{i}{2} \rfloor]$

$i := \lfloor \frac{i}{2} \rfloor$  {go to parent}

$A[i] := key$

# Heap implementation of priority queue

HEAP\_INSERT( $A, key$ )

$n := n + 1$

$i := n$

while  $i > 1$  and  $A[\lfloor \frac{i}{2} \rfloor] > key$   $\{O(\log n)\}$


do  $A[i] := A[\lfloor \frac{i}{2} \rfloor]$

$i := \lfloor \frac{i}{2} \rfloor$  {go to parent}

$A[i] := key$


- HEAP\_INSERT traverses  $O(\log n)$  nodes as HEAPIFY but makes fewer comparisons
- EXTRACT\_MIN takes  $O(\log n)$  like E\_MAX
- DECREASE\_KEY is analogous to insertion

# *MIN\_heap : example*



3	10	5	11	12	6	8	15	20	13
---	----	---	----	----	---	---	----	----	----


## *MIN\_heap : example*



3	10	5	11	12	6	8	15	20	13
---	----	---	----	----	---	---	----	----	----

Perform  $\text{HEAP\_INSERT}(A, 7)$  (*bubble-up*)

# MIN\_heap : example



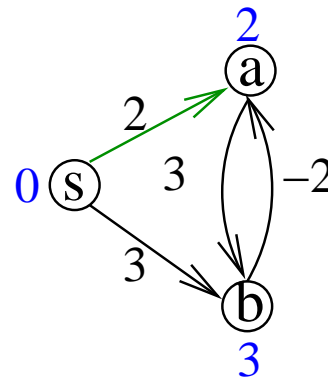
3	10	5	11	12	6	8	15	20	13
---	----	---	----	----	---	---	----	----	----

Perform HEAP\_INSERT( $A, 7$ ) (*bubble-up*)

DECREASE\_KEY( $A, i, key$ )

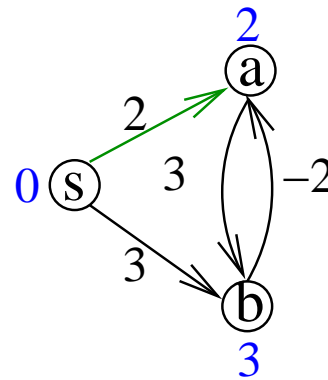
# General shortest path algorithm

Consider again the problematic instance for Dijkstra's

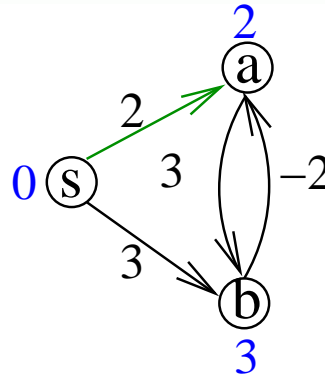


# General shortest path algorithm

Consider again the problematic instance for Dijkstra's



# General shortest path algorithm



In Dijkstra's algorithm the shortest path from  $s$  to  $v$  passes only through vertices which are **closer** to  $s$  than  $v$ . ( not true in the example above,  $s \rightarrow b \rightarrow a$  !)

# General shortest path algorithm

In Dijkstra's algorithm the shortest path from  $s$  to  $v$  passes only through vertices which are **closer** to  $s$  than  $v$ . ( not true in the example above,  $s \rightarrow b \rightarrow a$  !)

Dijkstra's algorithm is a sequence of the following updates:

# General shortest path algorithm

In Dijkstra's algorithm the shortest path from  $s$  to  $v$  passes only through vertices which are **closer** to  $s$  than  $v$ . ( not true in the example above,  $s \rightarrow b \rightarrow a$  !)

Dijkstra's algorithm is a sequence of the following updates:

UPDATE( $e = (u, v)$ )

if  $d[v] > d[u] + w(u, v)$  then

$d[v] := d[u] + w(u, v)$

$prev[v] := u$

# General shortest path algorithm

In Dijkstra's algorithm the shortest path from  $s$  to  $v$  passes only through vertices which are **closer** to  $s$  than  $v$ . ( not true in the example above,  $s \rightarrow b \rightarrow a$  !)

Dijkstra's algorithm is a sequence of the following updates:

UPDATE( $e = (u, v)$ )

if  $d[v] > d[u] + w(u, v)$  then

$d[v] := d[u] + w(u, v)$

$prev[v] := u$

Properties of UPDATE:

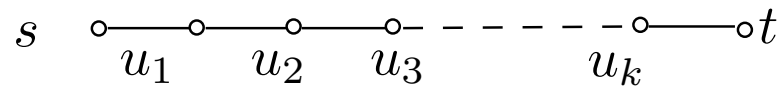
1. assigns the correct distance to  $v$  if  $u$  is the second-last vertex on the shortest path  $s \rightsquigarrow v$  and  $d[u]$  is correctly set
2. it will never make  $d[v]$  too small

# ***General shortest path algorithm***

Consider a shortest path from  $s$  to  $t$ .

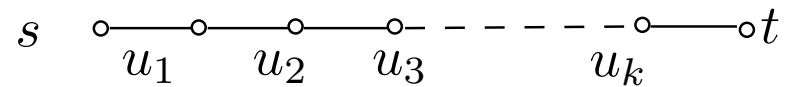
# General shortest path algorithm

Consider a shortest path from  $s$  to  $t$ .



# General shortest path algorithm

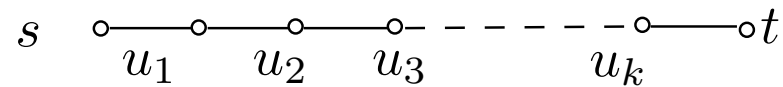
Consider a shortest path from  $s$  to  $t$ .



The path  $s \rightsquigarrow t$  has at most  $n - 1$  edges.

# General shortest path algorithm

Consider a shortest path from  $s$  to  $t$ .



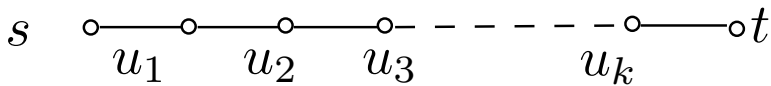
The path  $s \rightsquigarrow t$  has at most  $n - 1$  edges.

If we update them in the order

$(s, u_1), (u_1, u_2), (u_2, u_3), \dots, (u_k, t)$  then, by Property 1,  $d[t]$  will be computed correctly.

# General shortest path algorithm

Consider a shortest path from  $s$  to  $t$ .



The path  $s \rightsquigarrow t$  has at most  $n - 1$  edges.

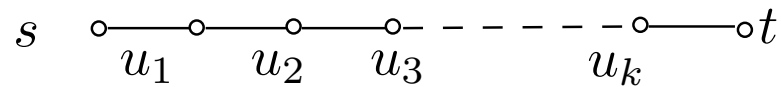
If we update them in the order

$(s, u_1), (u_1, u_2), (u_2, u_3), \dots, (u_k, t)$  then, by Property 1,  $d[t]$  will be computed correctly.

But we do not know the order beforehand.

# General shortest path algorithm

Consider a shortest path from  $s$  to  $t$ .



The path  $s \rightsquigarrow t$  has at most  $n - 1$  edges.

If we update them in the order

$(s, u_1), (u_1, u_2), (u_2, u_3), \dots, (u_k, t)$  then, by Property 1,  $d[t]$  will be computed correctly.

But we do not know the order beforehand.

Solution: **Update all edges  $n - 1$  times!**

# General shortest path algorithm

```
BELLMAN_FORD( $G, w, s$ )  
for all  $v \in V - \{s\}$  do  
   $d[v] := \infty; prev[v] := nil$   
 $d[s] := 0$   
for  $k := 1$  to  $n - 1$  do  
  Ⓞ for each edge  $e \in E(G)$   
    ⚠ UPDATE( $e$ )
```

# General shortest path algorithm

```
BELLMAN_FORD( $G, w, s$ )  
for all  $v \in V - \{s\}$  do  
   $d[v] := \infty; prev[v] := nil$   
 $d[s] := 0$   
for  $k := 1$  to  $n - 1$  do  
  Ⓞ for each edge  $e \in E(G)$   
    ⚠ UPDATE( $e$ )
```

Time:  $O(nm)$

# General shortest path algorithm

BELLMAN\_FORD( $G, w, s$ )

for all  $v \in V - \{s\}$  do

$d[v] := \infty; prev[v] := nil$

$d[s] := 0$

for  $k := 1$  to  $n - 1$  do

⊗ for each edge  $e \in E(G)$

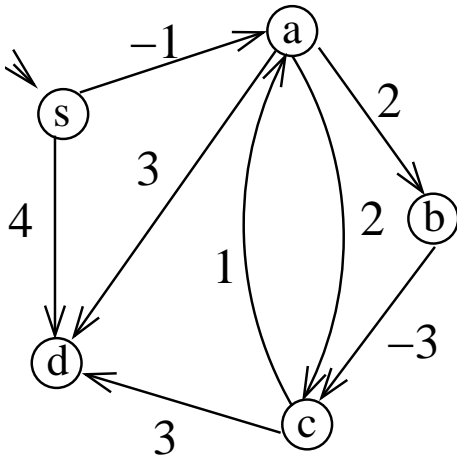
△ UPDATE( $e$ )

for each edge  $e = (u, v) \in E(G)$

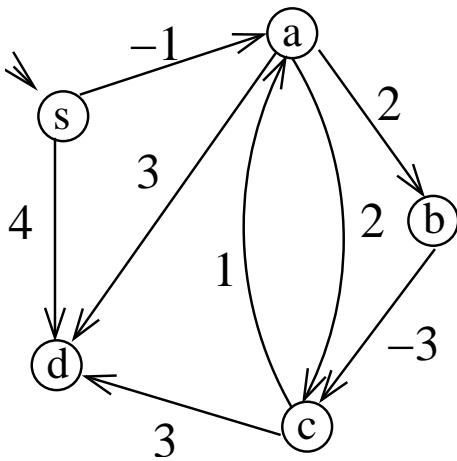
⊗ do if  $d[v] > d[u] + w(u, v)$  then **NEGATIVE CYCLE**

Time:  $O(nm)$

# Bellman-Ford algorithm

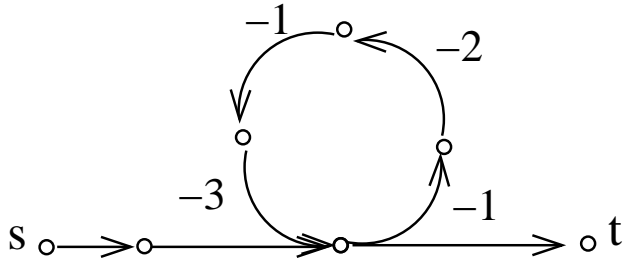


# Bellman-Ford algorithm

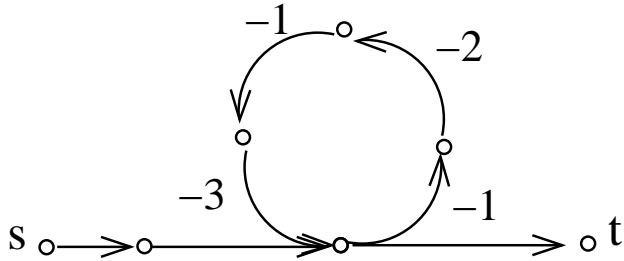


	Iteration				
Node	0	1	2	3	4
s	0	0	0	0	0
a	$\infty$	-1	-1	-1	-1
b	$\infty$	$\infty$	1	1	1
c	$\infty$	$\infty$	1	-2	-2
d	$\infty$	4	2	2	1

# Negative cycles

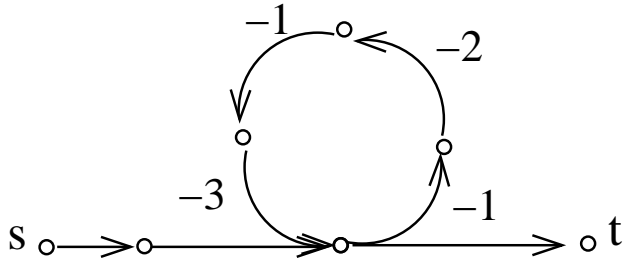


# Negative cycles



The shortest path from  $s$  to  $t$  does not exist.

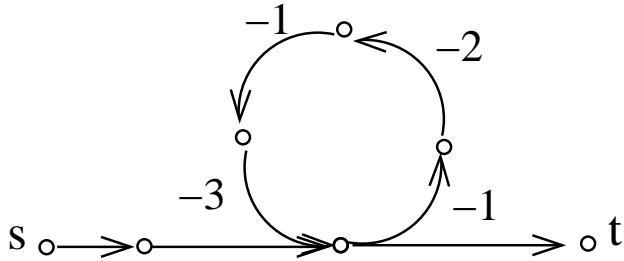
# Negative cycles



The shortest path from  $s$  to  $t$  does not exist.

An extra round in `BELLMAN_FORD` will detect it. Namely, if the value of  $d[v]$  for any  $v$  decreases in this extra round then there is a negative cycle in  $G$ .

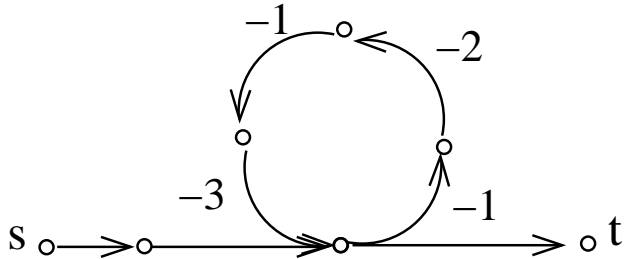
# Negative cycles



The shortest path from  $s$  to  $t$  does not exist.

An extra round in `BELLMAN_FORD` will detect it. Namely, if the value of  $d[v]$  for any  $v$  decreases in this extra round then there is a negative cycle in  $G$ .

# Negative cycles



The shortest path from  $s$  to  $t$  does not exist.

An extra round in `BELLMAN_FORD` will detect it. Namely, if the value of  $d[v]$  for any  $v$  decreases in this extra round then there is a negative cycle in  $G$ .