Lecture 7
Data Structures (DAT037)

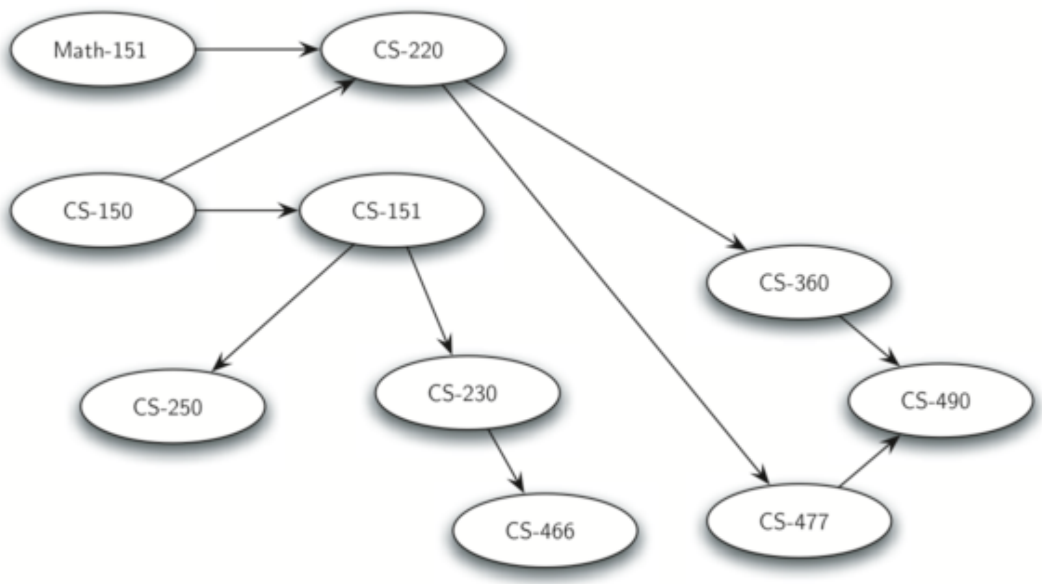
Ramona Enache
(with slides from Nick Smallbone and Nils Anders Danielsson)

Directed Acyclic Graphs

A **DAG** is a directed graph without cycles.

That means:
once you follow an edge there is no way back to the source node

(we can say that one node
Is after another in the graph)



Topologic Sort

An example:

- nodes are tasks, and an edge (u, v) means “task u must be done before task v”
- the graph is a DAG,

It means that there are no impossible dependencies between tasks
A topological sort gives a valid order to do the tasks in

Implementing Topologic Sort

```
r = new empty list
while V ≠ ∅
  do
     if any v ∈ V with indegree(v) = 0
        then  r.add-last(v)
              remove v from G
      else
          raise error: cycle found
return r // Nodes, topologically sorted.
```

Implementing Topologic Sort

```
r = new empty list
while V ≠ ∅
  do
    if any v ∈ V with indegree(v) = 0
       then  r.add-last(v)
             remove v from G
     else
          raise error: cycle found
return r // Nodes, topologically sorted.
```

How can we avoid removing v from G ?

Implementing Topologic Sort

```
r = new empty list
d = map from vertices to their indegrees // null for nodes in r.
repeat |V| times
   if d[v] == 0 for some v
      then  r.add-last(v)
            d[v] = null
            for each direct successor v' of v
              do
                 decrease d[v'] by 1
     else
        raise error: cycle found
return r // Nodes, topologically sorted.
```

Implementing Topologic Sort

Complexity Analysis
(Need more info because of pseudocode representation).

- Nodes: 0, 1,…|V|-1
- *adjacent*: array with V positions

adjacent[i] – contains list of neighbours for node I

- *r*: dynamic array,
- *d*: array.

Implementing Topologic Sort

// d (indegree) is a map from nodes to the indegrees that it would have
// easier for deleting a node from the graph – indegree would be null

```
d = new array of size |V|
for i in [0,...,|V|-1]
  do
    d[i] = 0
for i in [0,...,|V|-1]
  do
   for each direct successor j of i
     do
      d[j]++
```

O(|V|)

O(|E|)

Implementing Topologic Sort

r = new empty list
O(1)
d = map from vertices to their indegrees // null for nodes in r.
O(|V| + |E|)
repeat |V| times
O(|V|)
    if d[v] == 0 for some v
O(|V|)
        then r.add-last(v)
O(1)
                d[v] = null
O(1)
                for each direct successor v' of v
O(|E|)
                  do
                      decrease d[v'] by 1
O(1)
      else
          raise error: cycle found
return r // Nodes, topologically sorted.
Total : O(|V|^2 + |E|)

Implementing Topologic Sort

```
r = new empty list
d = map from vertices to their indegrees
q = queue with all nodes of indegree 0
while q is non-empty
  do
     v = q.dequeue()
     r.add-last(v)
     for each direct successor v' of v
       do
          decrease d[v'] by 1
          if d[v'] = 0 then  q.enqueue(v')
if r.length() < |V| then  raise error: cycle found
return r // Nodes, topologically sorted.
```

Question

What is the time complexity of this solution if adjacency lists are used to represent edges ?

1. $O(|V|)$
2. $O(|E|)$
3. $O(|V|+|E|)$
4. $O(|V|^2)$

govote.at
Code 771118

Implementing Topologic Sort

r = new empty list — O(1)

d = map from vertices to their indegrees — O(|V|+|E|)

q = queue with all nodes of indegree 0 — O(|V|)

while q is non-empty — O(|V|)

  do

    v = q.dequeue() — O(1)

    r.add-last(v) — O(1)

    for each direct successor v' of v — O(E) in total

      do

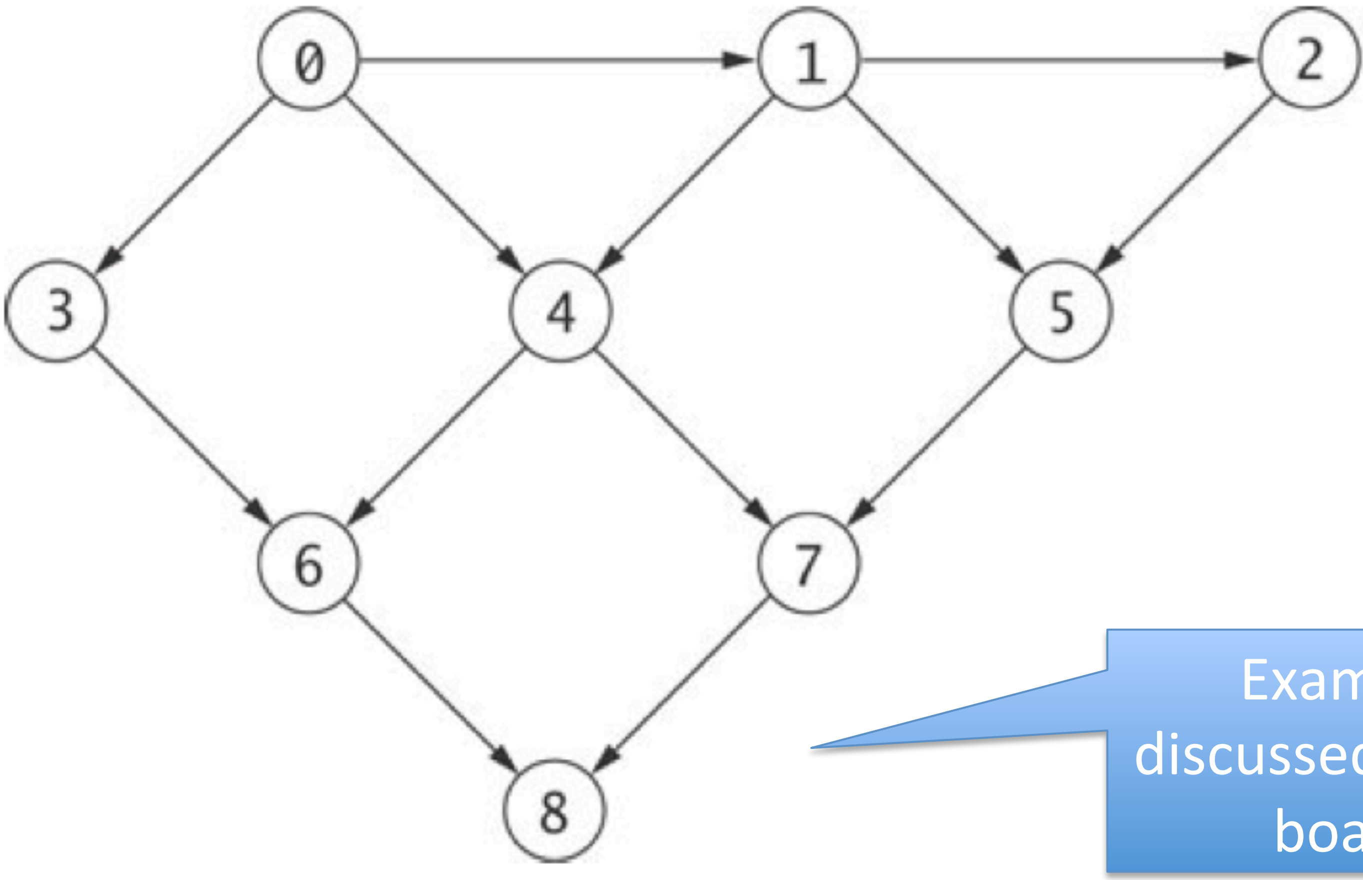
        decrease d[v'] by 1 — O(1)

        if d[v'] = 0 **then** q.enqueue(v') — O(1)

if r.length() < |V| **then** raise error: cycle found — O(1)

return r // Nodes, topologically sorted. — O(1)

Example of Topologic Sort



Example discussed on the board

Depth-first Search – Again!

- **Depth-first search** is an alternative search order that's easier to implement
- To do a DFS starting from a node:

  - visit the node

  - recursively DFS all adjacent nodes (skipping any already-visited nodes)
- Much simpler ☺

Applications of DFS

- Checking if a graph has a cycle
- Checking if the graph is connected
- Alternative algorithm for topological sort
- …

Topologic Sort with DFS

L ← Empty list that will contain the sorted nodes
while there are unmarked nodes
do
select an unmarked node n
visit(n)
function visit(node n)
if n has a temporary mark **then** stop (not a DAG)
if n is not marked (i.e. has not been visited yet)
then mark n temporarily **for each** node m with an edge from n to m **do** visit(m) mark n permanently unmark n temporarily add n to *head* of L

Strong Connection in Graphs

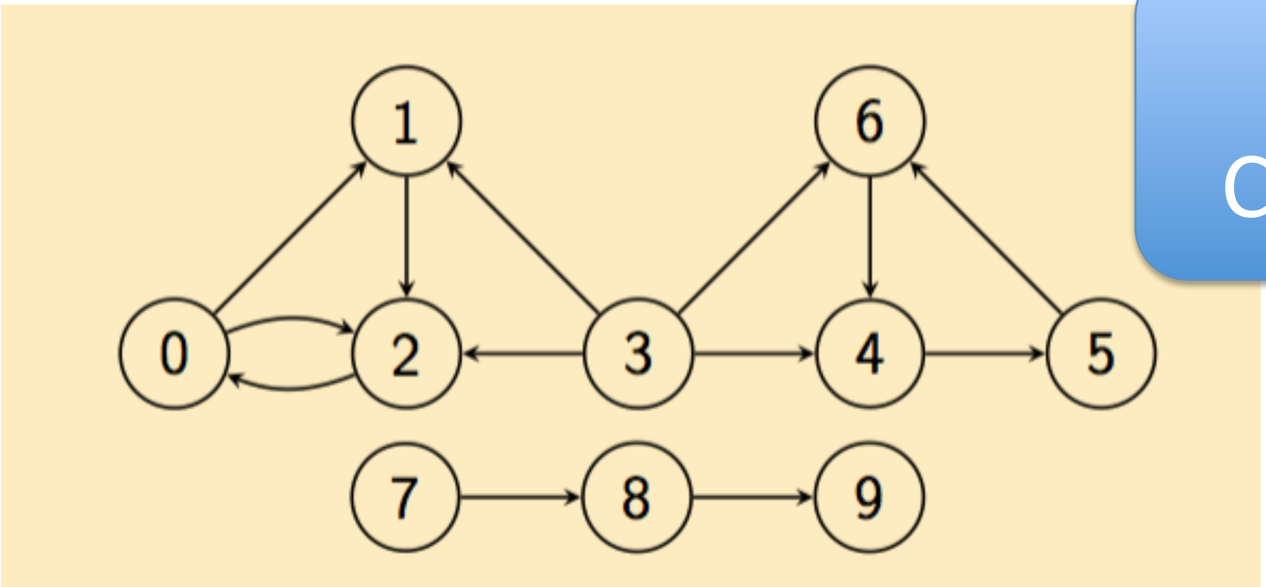
A directed graph where there is a path between any two nodes in both directions is called **strongly connected**.

A graph can be split into **strongly connected components** (subgraphs that are strongly connected) with the help of DFS

Strongly Connected Components

Question

How many strongly connected components are here ?



govote.at
Code 510490

Strongly Connected Components

Kosaraju's algorithm

S ← empty stack
while S does not contain all vertices
  do
    choose an arbitrary vertex *v* not in S
    dfs(v)
      - for each vertex *u* found, push *u* onto S.
reverse the directions of all arcs to obtain the transpose graph.
while S is nonempty
  pop the top vertex *v* from S.
  dfs(v) in the transposed graph
The set of visited vertices will give the strongly connected component containing *v*; record this and remove all these vertices from the graph G and the stack S.

Shortest paths

Typical problems

- Find the shortest path between
- u and v
- Find the shortest path between u and all other nodes from the graph
- Find the shortest path between any two nodes from the graph

Shortest Path – Based on BFS

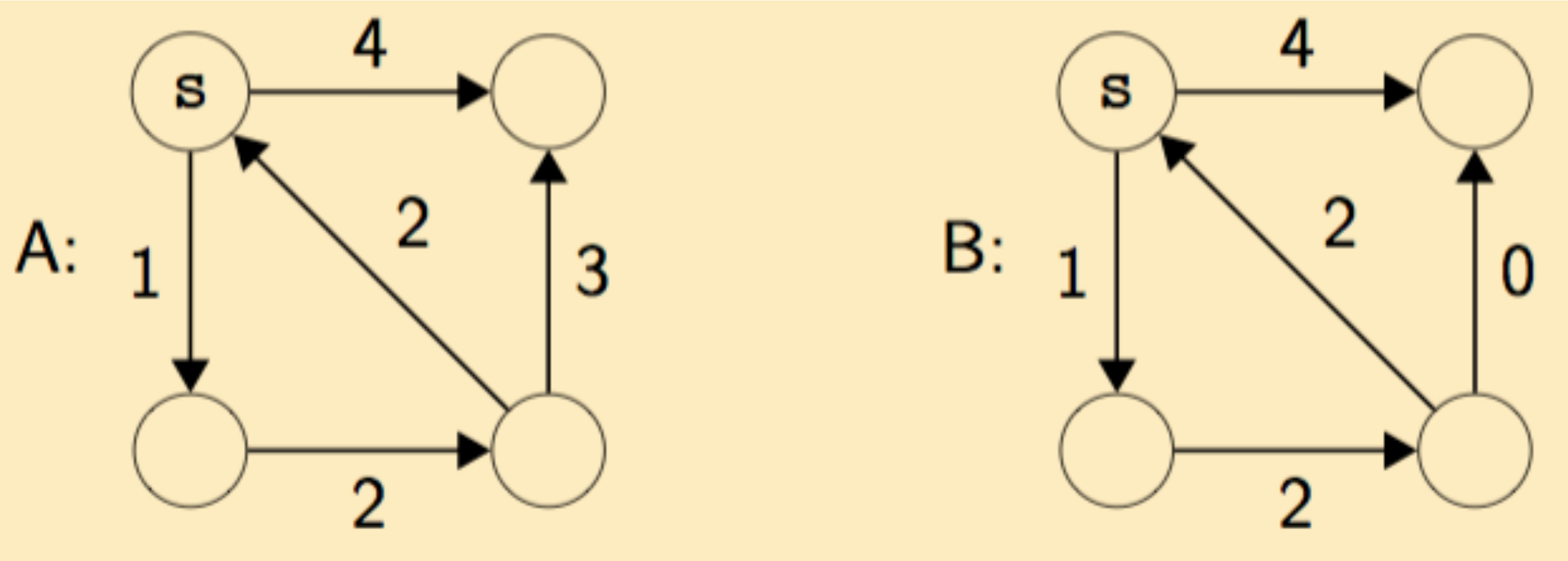
```
d = new array of size |V|, initialised to ∞
p = new array of size |V|, initialised to null
q = new empty queue
q.enqueue(s)
d[s] = 0
while q is non-empty
  do
     v = q.dequeue()
     for each direct successor v' of v
        do
          if d[v'] = ∞ then
                d[v'] = d[v] + 1
                p[v'] = v
                q.enqueue(v')
return (d, p)
```

Shortest Path based on BFS

d = new array of size |V|, initialised to ∞ (O(|V|))
p = new array of size |V|, initialised to null (O(|V|))
q = new empty queue
q.enqueue(s)
d[s] = 0
while q is non-empty (O(|V|))
  do
    v = q.dequeue()
    for each direct successor v' of v (O(|E|))
      do
        if d[v'] = ∞ **then**
            d[v'] = d[v] + 1
            p[v'] = v
            q.enqueue(v')
return (d, p)

Question – Part 1

Test out the algorithm on the following graphs.



Question – Part 2

What kind of graphs is the previous algorithm good for finding the shortest path:

A. No graphs
B. Undirected graphs
C. Unweighted graphs
D. Any graphs

govote.at
Code 637013

Dijkstra's Algorithm

d = new array of size |V|, initialised to ∞
p = new array of size |V|, initialised to null
k = new array of size |V|, initialised to false
d[s] = 0
repeat
  if no v' satisfies !k[v] && d[v'] < ∞ **then break**
  v = v' with smallest d[v'] that satisfies ! k[v']
  k[v] = true
  for each direct successor v' of v
    do
      if (! k[v']) and d[v'] > d[v] + c(v,v') **then**
        d[v'] = d[v] + c(v,v')
        p[v'] = v
return (d, p)

Edges are represented with adjacency lists

Dijkstra's Algorithm

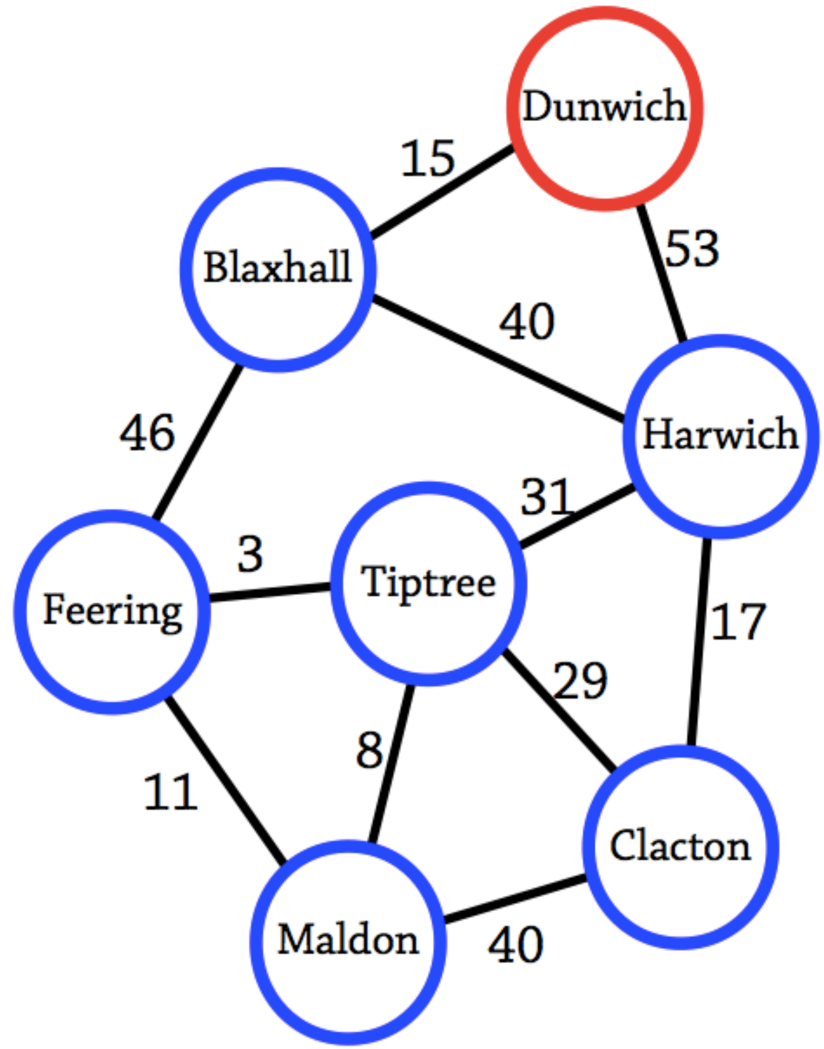
Complexity

$O(|V|^2 + |E|) = O(|V|^2)$

Motivation in the book

Dijkstra's Algorithm - Example

Demonstration on the board !

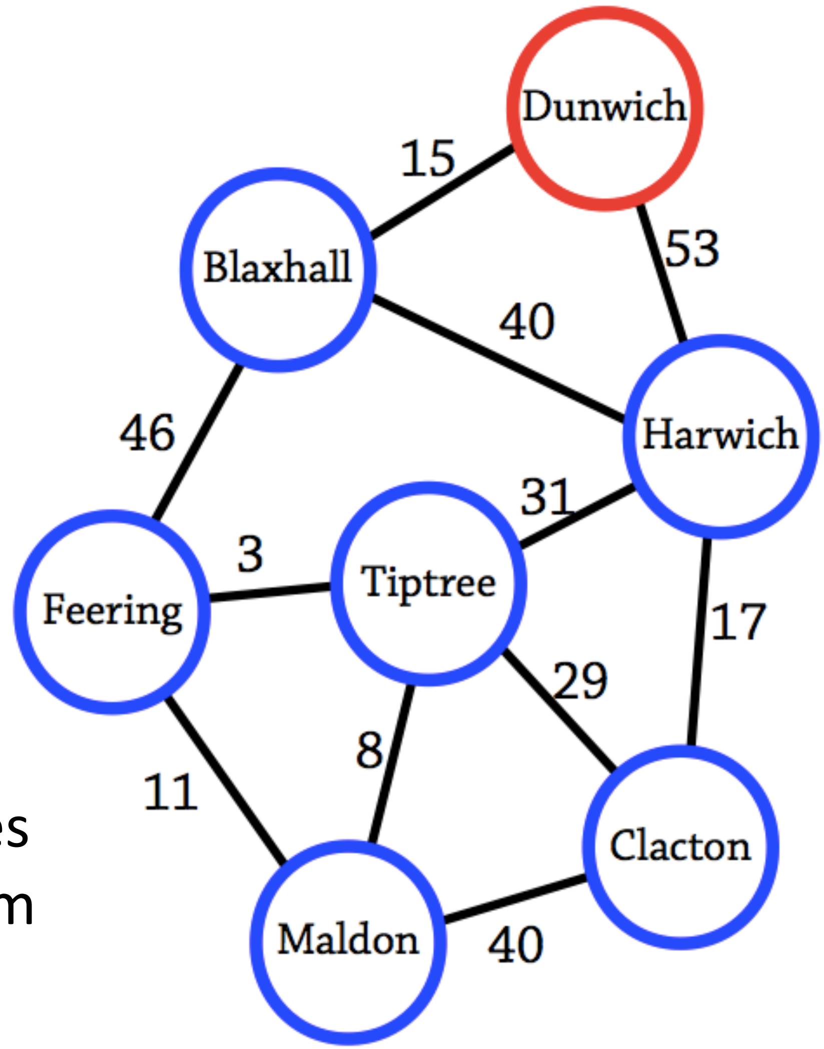


Dijkstra’s Algorithm - Example

S = {Dunwich → 0,
Blaxhall → 15,
Harwich → 53,
Feering → 61,
Tiptree → 64,
Clacton → 70,
Maldon → 72}

Finished ☺

Dijkstra's algorithm enumerates nodes in order of how far away they are from the start node



Dijkstra's Algorithm

Where can we optimize the algorithm ?

1. Can't
2. Choose a better starting point
3. Use an adjacency matrix instead
4. Use a better structure to compute the next node

govote.at
Code 308091

Dijkstra’s Algorithm – Take 2

```
d,p,k – as before
q = new empty priority queue
d[s] = 0
q.insert(s, 0)
while q is non-empty
   do
      v = q.delete-min()
      if ! k[v] then
            k[v] = true
            for each direct successor v' of v do
               if (not k[v']) and d[v'] > d[v] + c(v,v') then
                     d[v'] = d[v] + c(v,v')
                     p[v'] = v
                     q.insert(v', d[v'])
return (d, p)
```

Dijkstra's Algorithm – Take 2

Motivation in the book

Complexity

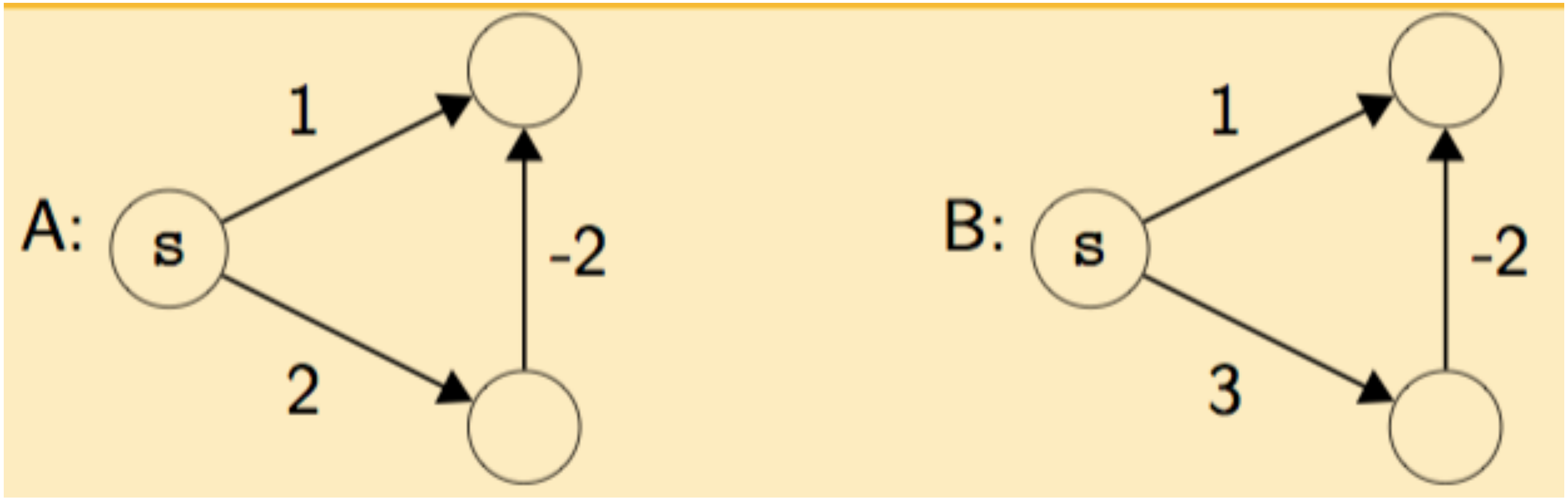
$O(|E| \log |V| + |V| \log |V|)$

$= O(|E|\log|V|)$

Because of **decreaseKey** and **findMin** operations of the priority queus

Dijkstra’s Algorithm

Let’s look at the following example.
What happens here if we apply Dijkstra’s algorithm ?



Dijkstra’s Algorithm

Dijkstra’s algorithm only works for positive weights !!

There is a more general algorithm for weighted graphs

- more complex ☹
- described in the book ☺

To Do

Read from the book:
+ 9.3, 9.6
+ **better** reference for graphs:
Wikipedia
Introduction to Algorithms by Cormen, Leiserson, Rivest and Stein

Fun with graphs:
+ famous graph problem: **The Seven Bridges of Königsberg**
http://www.mathsisfun.com/activity/seven-bridges-konigsberg.html

Implement:
+ graphs + the algorithms from today in your favourite programming language

Labs:
+ 26th Nov – deadline Lab 2
+ 27th Nov – final deadline Lab 1