ΟΙΚΟΝΟΜΙΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ATHENS UNIVERSITY OF ECONOMICS AND BUSINESS

M.Sc. Program in Computer Science
Department of Informatics

Design and Analysis of Algorithms

Graph Algorithms

Vangelis Markakis

markakis@gmail.com

Graphs

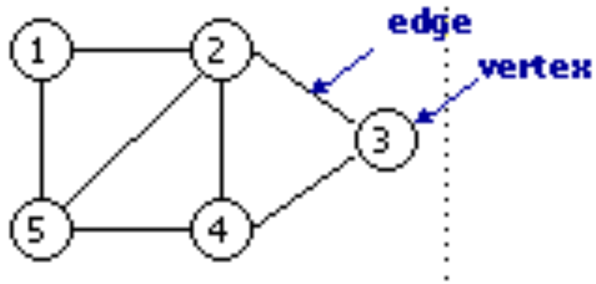
- $G = (V, E)$
- $V = \{1,2,\ldots,n\}$: set of nodes/vertices, $|V| = n$
- $E \subseteq V \times V = \{(u,v) \mid u,v \in V\}$: set of edges/arcs, $|E| = m$

- undirected graphs $(u,v) \equiv (v,u)$
  - $\Gamma(u) = \{v \mid (u,v) \in E\}$: neighborhood of u
  - $d(u) = |\Gamma(u)|$: degree of u

- directed graphs $(u,v) \neq (v,u)$
  - $\Gamma^+(u) = \{v \mid (u,v) \in E\}$: out-neighborhood of u
  - $\Gamma^-(u) = \{v \mid (v,u) \in E\}$: in-neighborhood of u
  - $d^+(u) = |\Gamma^+(u)|$: out-degree of u
  - $d^-(u) = |\Gamma^-(u)|$: in-degree of u

Graph representation

- n = # vertices
- m = #edges

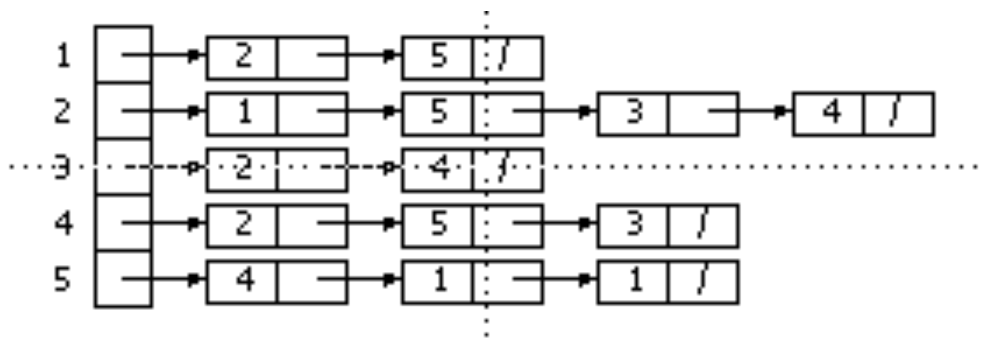


Adjacency matrix

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	0	0

space $O(n^2)$

Adjacency list



space $O(n+m)$

Dense graphs: m is $O(n^2)$

Graph Traversal

- Suppose we want to visit all the vertices and edges/arcs of a graph
- E.g., we may want to process all nodes in a specific order
- Many applications require such a traversal of the graph
- Can we do this efficiently?

Part 1: DFS and its applications

Depth-First Search (DFS)

The graph may be disconnected

Hence, we can start by an auxiliary method that answers the question: Given a node u, find out which nodes are reachable from node u

```
explore(u)  //for undirected graphs
{  previsit(u);//optional
   visited(u):=true;
   for each v ∈ Γ(u) do //in alphabetical order, Γ+(u) for
   digraphs
       if not visited(v)
             then explore(v);
   postvisit(u);//optional
}
```

- Previsit and postvisit are optional
- In case we want to process the node the first time we discover it, or the last time we are at it
- Will see applications soon

Depth-First Search (DFS)

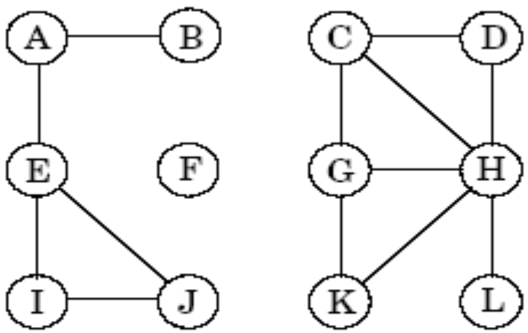
To explore all the nodes of a graph G:

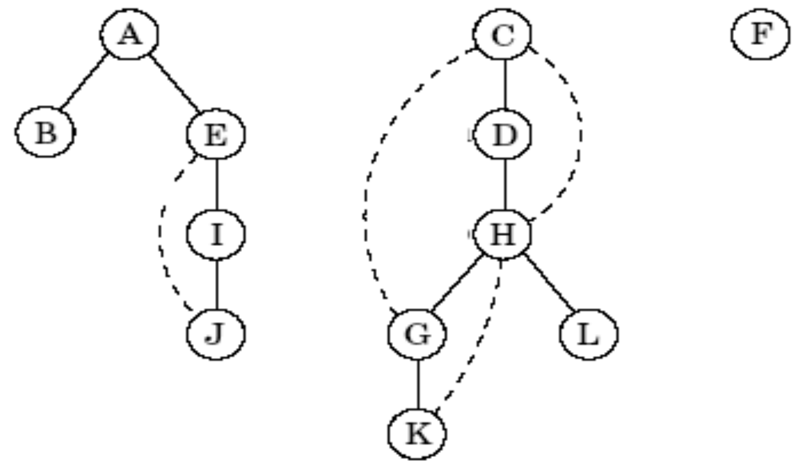
```
DFS (G)
{  for all u ∈ V do visited(u):= false;
   for all u ∈ V do
       if not visited(u) then  explore(u)}
```

- Complexity: O(n+m)
  - Assuming previsit and postvisit take O(1)
  - We only look at each edge (u, v) 2 times, once when we examine u and once when we examine v

Depth-First Search (DFS)

Example





Visiting order: A B E I J - C D H G K L - F

DFS tree T = (V, A) for an undirected graph, $A \subseteq E$

The graph G is decomposed in

- tree edges $(u,v) \in A$
- backward edges $(u,v) \notin A$

DFS - Simple applications

Let G be an undirected graph

- What is the visiting order of the nodes under DFS ?
- Can we decide if G is connected?
  - i.e., there is a path that connects every pair of nodes
- Can we find the # of connected components?
- Who is the parent of each node in the DFS tree?

To answer these, we will exploit the previsit and postvisit operations

DFS - Simple applications

Parameters

clock: integer counter (defining the visiting order)

pre(u): the visiting order of u

cc: counts the number of connected components

ccnum(u): the connected component that u belongs to

parent(u): the parent of u in the DFS tree

DFS - Simple applications

```
explore(u);
{  previsit(u);
   visited(u):=true;
   for each v ∈ Γ(u)  do
        if not visited(v)then
          {explore(v); parent(v):=u; }
   postvit(u); }
```

→

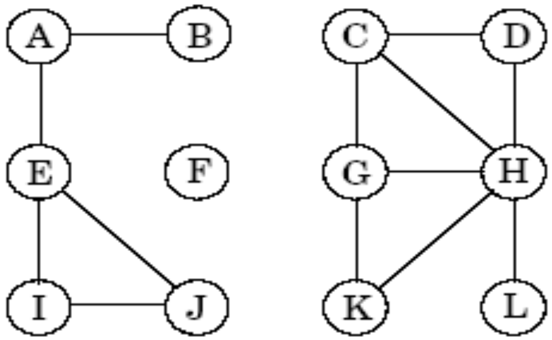
```
previsit(u);
{  ccnum(u)=cc;
   pre(u):= clock;
   clock:= clock+1   }
```

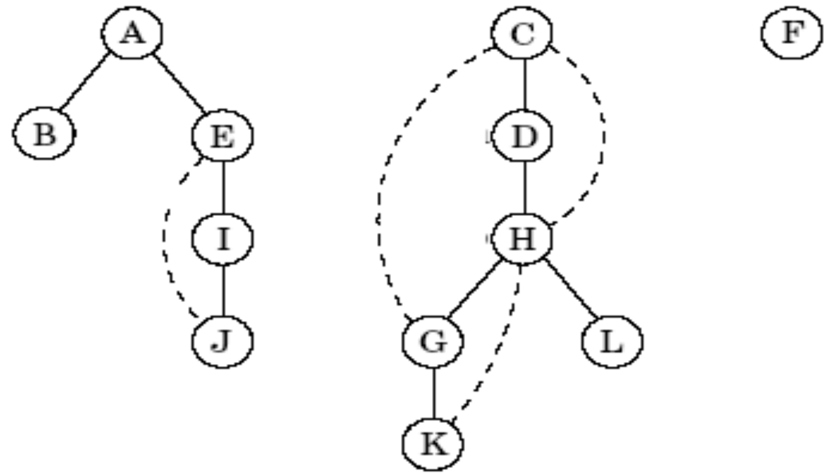
→ empty

```
DFS (G);
{  clock:=1;   cc:=0;
   for all u ∈ V do
        {visited(u):= false; parent(u):=null}
   for all u ∈ V do
        if not visited(u)  then
              { cc:=cc+1; explore(u);}  }
```

Complexity
O(n+m)

DFS - Simple applications





u	A	B	C	D	E	F	G	H	I	J	K	L
pre(u)	1	2	6	7	3	12	9	8	4	5	10	11
ccnum(u)	1	1	2	2	1	3	2	2	1	1	2	2
parent(u)	null	A	null	C	A	null	H	D	E	I	G	H

DFS - More applications

- In other applications we will need to work harder
- More information from the graph traversal
- For this, we will need to exploit the postvisit procedure too

Parameters

clock: integer counter (increased by previsit and postvisit)

pre(u): the first time we visit u (determined by previsit)

post(u): the last time we deal with u (determined by postvisit)

DFS orderings

```
explore(u);
{  previsit(u);
   visited(u):=true;
   for each v ∈ Γ(u) do
    if not visited(v) then explore(v);
   postvit(u); }
```

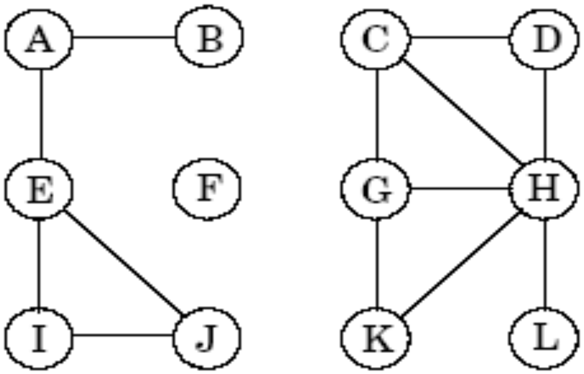
```
previsit(u);
{  pre(u):= clock;
   clock:=clock+1   }
```

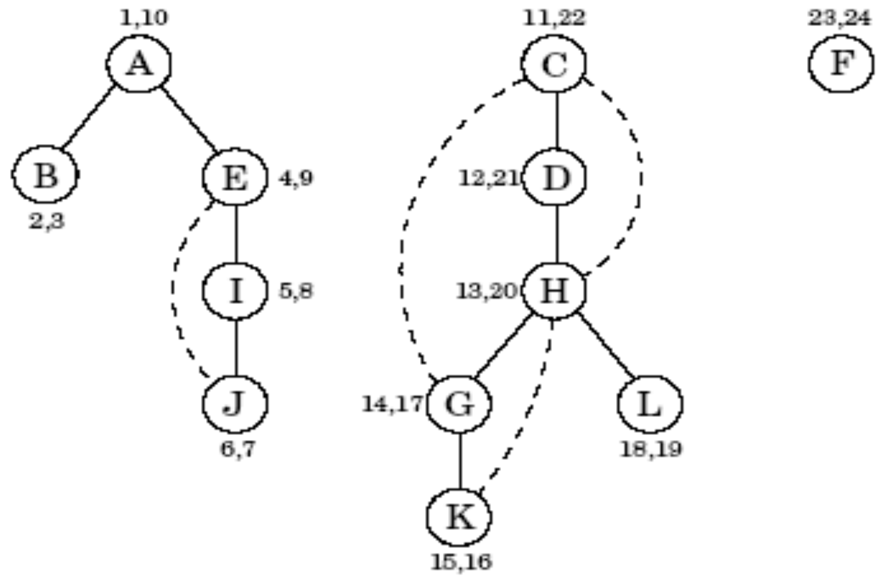
```
postvisit(u);
{ post(u):= clock;
   clock:=clock+1   }
```

```
DFS (G);
{ clock:=1;
  for all u ∈ V do
       visited(u):= false;
   for all u ∈ V do
       if not visited(u) then explore(u); }
```

Complexity: again O(n+m)

DFS orderings – undirected graphs

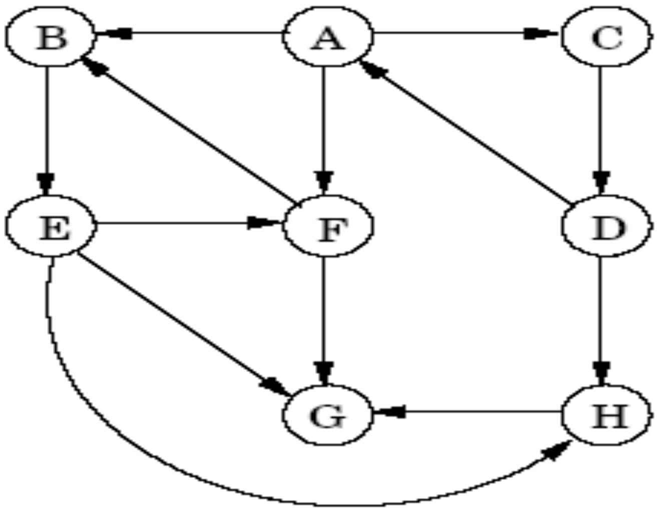


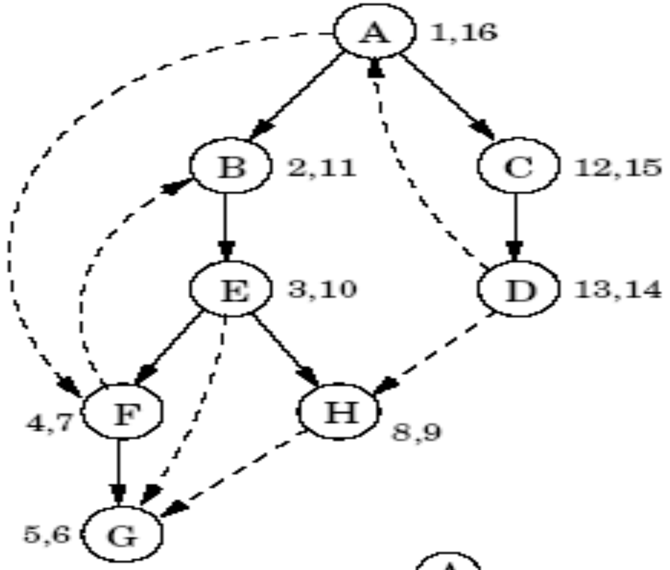


Claim: For any pair of nodes u and v the intervals [pre(u), post(u)] and [pre(v), post(v)] are either disjoint or one is contained in the other

Due to how recursion works (i.e., Last-In First-Out operation of recursion stack)

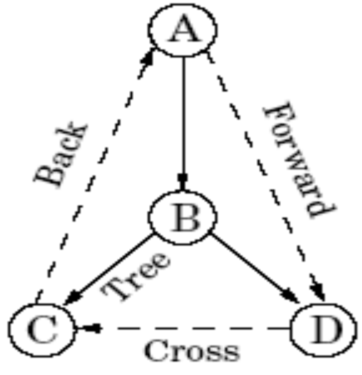
DFS orderings – directed graphs



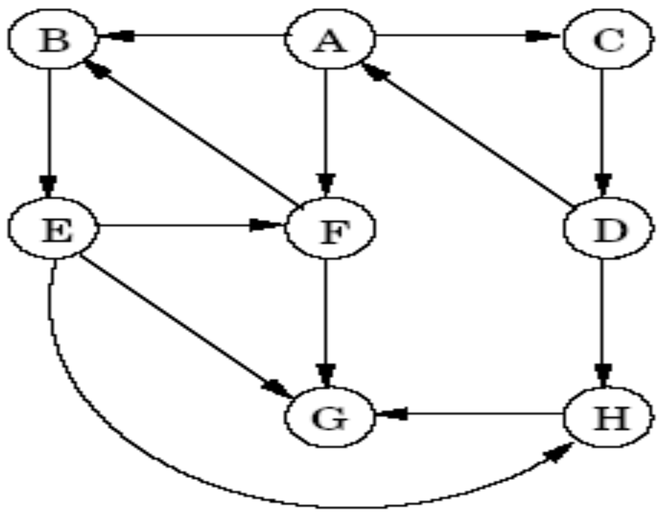


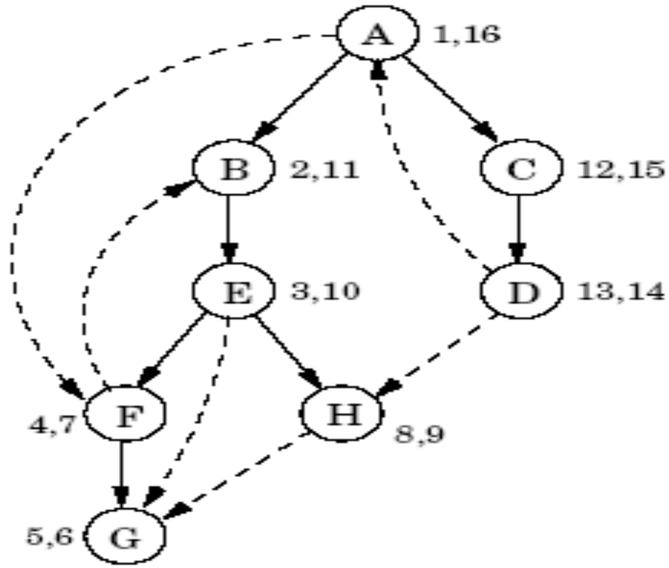
Four types of (directed) edges:

- Tree edges
- Forward edges: to a non-child descendant
- Backward edges: to an ancestor (cycle)
- Cross edges: to neither an ancestor nor a descendant



DFS orderings – directed graphs





Relations between the pre and post parameters for each edge type:

pre/post *ordering for* (u, v)	*Edge type*
$[_u \quad [_v \quad]_v \quad]_u$	Tree/forward
$[_v \quad [_u \quad]_u \quad]_v$	Back → **post(u)<post(v)**
$[_v \quad]_v \quad [_u \quad]_u$	Cross

Directed Acyclic Graphs (DAGs)

(directed) cycles in a directed graph:

paths of the form $v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots v_k \rightarrow v_0$

DAG: A directed graph G = (V, E) with no directed cycles

= a partial order on V

Q: Given a directed graph, is it acyclic ?

A: It is iff DFS(G) does not produce any back edge

- Many applications of DAGs in scheduling problems or in modeling hierarchies, dependencies, etc
- E.g., suppose we have to schedule a set of jobs with constraints of the form "job i cannot start before job j finishes"
- Modeling the dependencies can be done with a DAG
- Clearly there should be no cycles among the dependencies

Topological sorting of DAGs

Suppose we have a DAG modeling sheduling dependencies

Q: In what order should we execute the jobs?

Topological sorting = an ordering of the vertices such that for every edge (u, v), u is before v in the ordering

- not necessarily unique

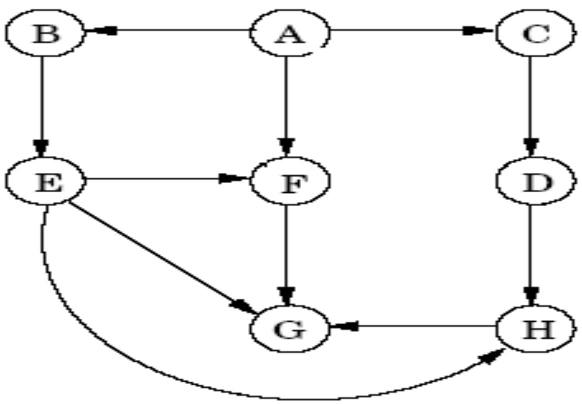
Q: Find a topological sorting of a DAG

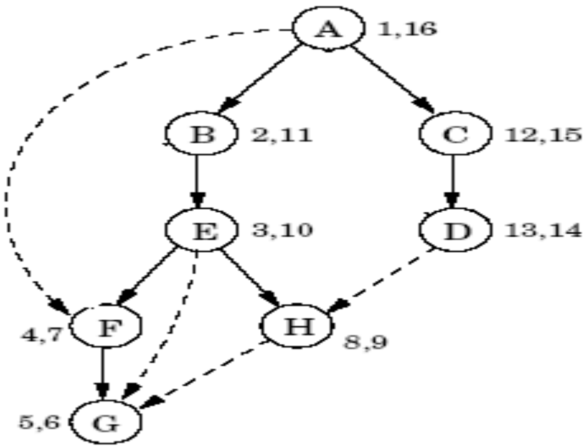
A: use DFS and output nodes in decreasing order of post(u)

Claim: For every edge (u, v) in a DAG, post(u) > post(v)

- Because there are no backward edges

Topological sorting of DAGs





Visiting order: A B E F G H C D
Topological order: A C D B E H F G

Every DAG has such a topological sorting

Topological sorting of DAGs

More properties of DAGs:

- Every DAG has at least one source node (a node with no incoming edges)
  - The node in the beginning of the topological sorting has to be a source
- Every DAG has at least one sink node (a node with no outgoing edges)
  - The last node in the topological sorting has to be a sink

A different approach to produce a topological sorting:

- Find a source node
- Print it and delete it from the graph (remaining graph is still a DAG)
- Continue in the same manner with the remaining vertices

Connectivity in Directed Graphs

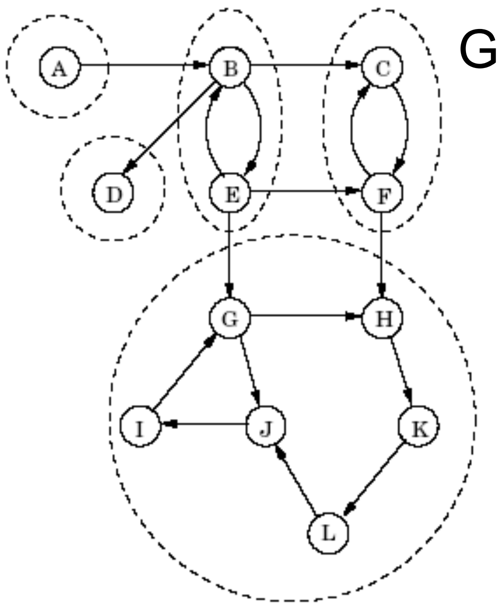
What does connectivity mean in a digraph G?

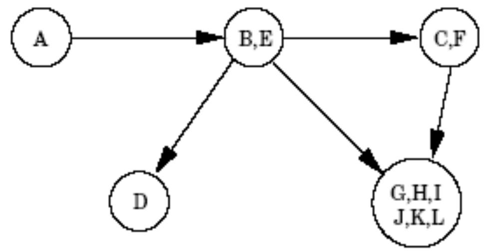
Two nodes u, v of G are connected iff

there is a path from u to v and a path from v to u

- A directed graph is strongly connected iff every pair of nodes is connected
- If a graph is not strongly connected, then a maximal subset of nodes S, such that any 2 nodes u, v from S are connected, is called a strongly connected component (SCC) of G
- A directed graph is weakly connected iff for every pair of nodes u, v, either there is a path from u to v or there is a path from v to u
- We can similarly define weakly connected components
- We are mostly interested in identifying the strongly connected components of a graph

Strongly Connected Components





How can we find the SCCs of a directed graph G?

Let's understand first the structure of the SCCs in a graph

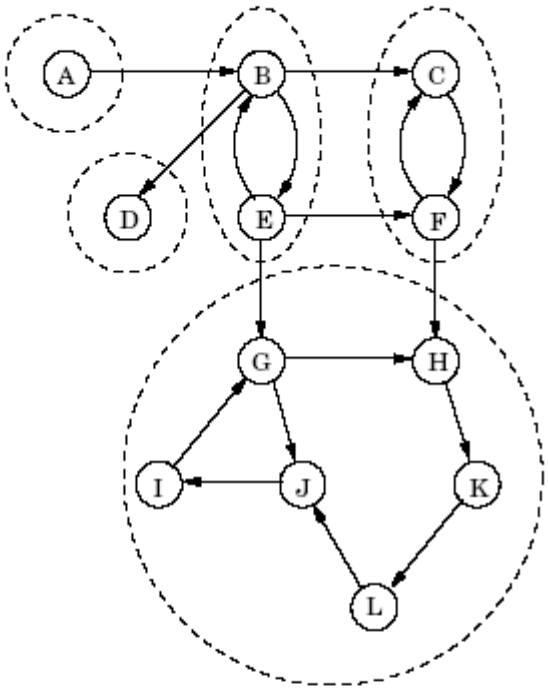
- Suppose we take each SCC and shrink it into one node
- This creates a meta-graph H, where:
  - The vertices of H are the SCCs
  - There is an edge from a SCC C to a SCC C' if there exists an edge (u, v) in G, such that u ∈ C and v ∈ C' (i.e., if there is a way to go from C to C')

Strongly Connected Components

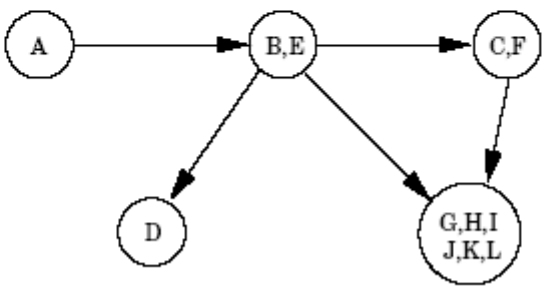
Claim: The graph of the SCCs is a DAG

Hence:

- H has at least one source node and at least one sink node
- Let S = source SCC
- Let T = sink SCC



G



Strongly Connected Components

Idea: **`explore(u)`** for some $u \in T$, will visit all nodes of T and no other node

- This way we can identify the sink SCC
- No such guarantees if we run explore(u) for some node u not in T

Q1: How can we find such a node $u \in T$, without knowing T in advance ?

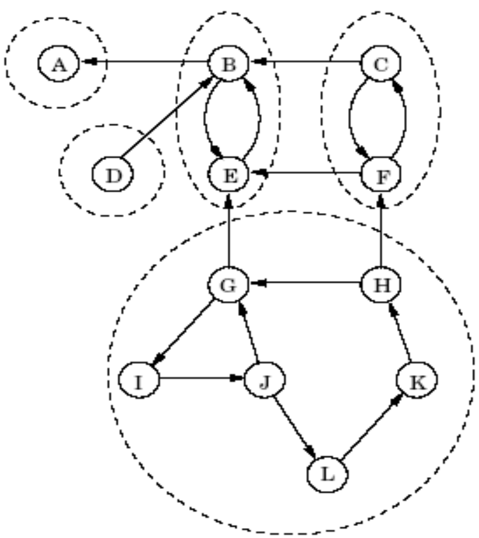
Q2: If we succeed in Q1, we have only identified one SCC. How can we find the other SCCs ?

There seems to be no direct way to locate a node in T...

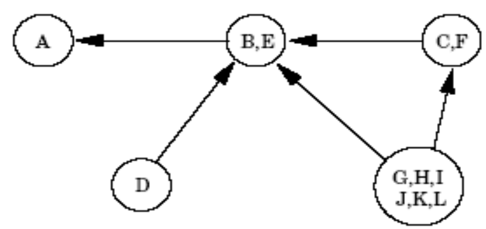
Strongly Connected Components

Q1: How can we find such a $u \in T$?

- Property 1: The node of G with the highest post number in DFS(G) is in a source SCC (why?)
- But, we need a node in a sink SCC of G…
- Idea: Work on the reverse graph of G
- G^R = same vertices as in G, but with all edges reversed



G^R



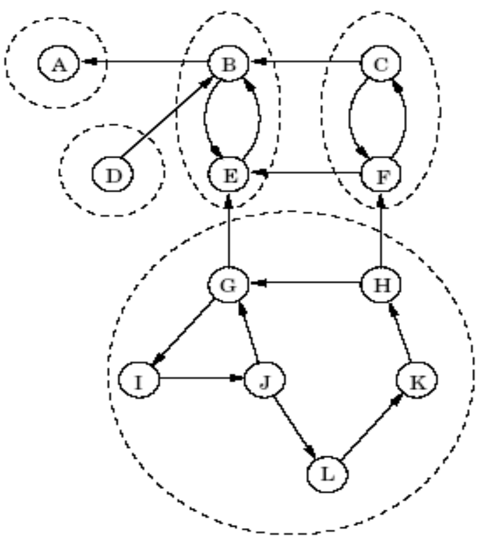
Strongly Connected Components

Q1: How can we find such a u ∈ T ?

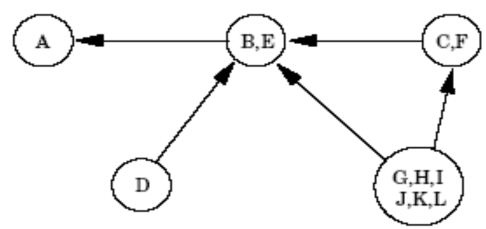
Property 2: G^R has the same SCCs with G (why?)

Hence:

Run DFS on G^R: The node u with highest post(u) lies in some source SCC of G^R ➔ u lies in a sink SCC of G !



G^R



Strongly Connected Components

Q2: How can we continue to the next SCC ?

Property 3: If for SCCs C and C', there is an edge from C to C', then the highest post(u), $u \in C$ is bigger than the highest post(v), $v \in C'$

- This suggests we can keep using the post(u) ordering of the DFS on G^R
- After we delete a sink SCC from G: the node in G\T of the highest post number according to DFS(G^R) belongs again to a source SCC of G^R → a sink SCC in G\T

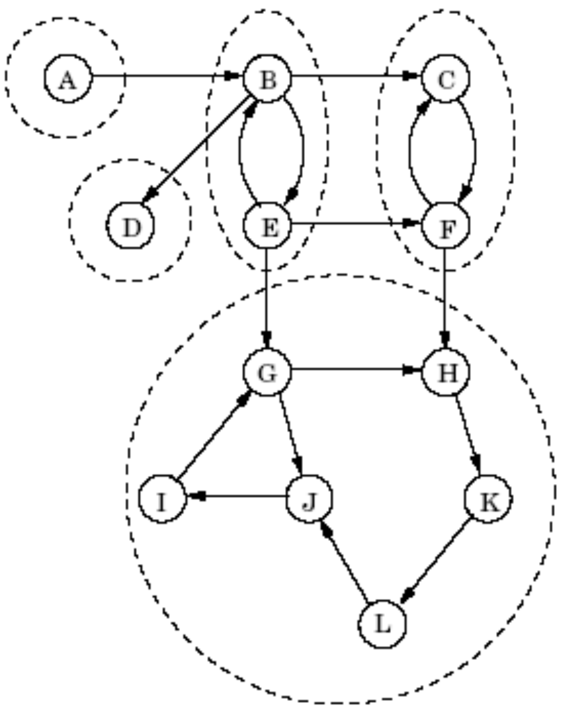
Strongly Connected Components

Algorithm for finding all SCCs

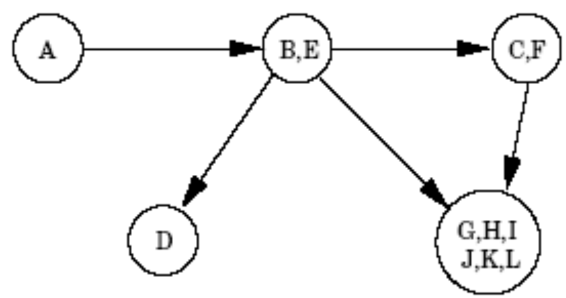
Run DFS on G^R to obtain post(u) for every $u \in V$;

Run on G:

- the algorithm we saw for finding connected components for undirected graphs (but using $\Gamma^+(u)$ instead of $\Gamma(u)$),
- processing the vertices of G in decreasing order of post(u)



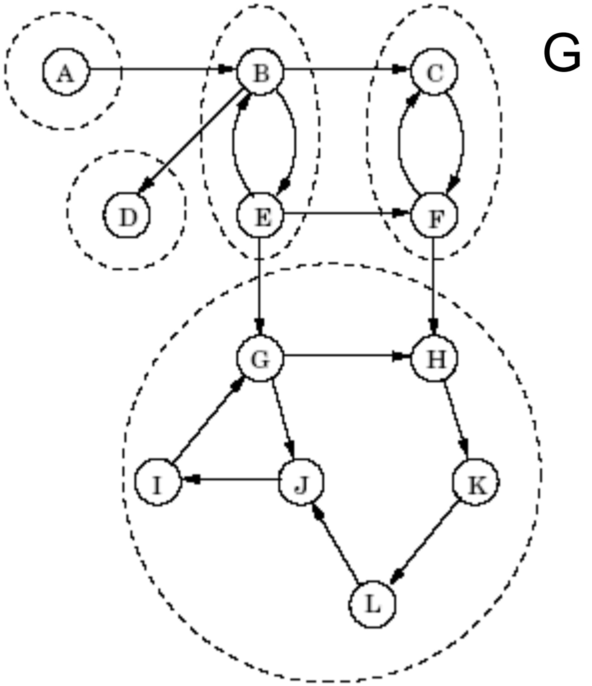
G



Strongly Connected Components

```
explore(u);
{  ccnum(u)=cc;
   visited(u):=true;
   for each v ∈ Γ+(u)  do
       if not visited(v)then explore(v)}

DFS (G);
{   cc:=0;
   for all u ∈ V do visited(u):= false;
   for all u ∈ V
       in decreasing order of post(u)  do
               if not visited(u)  then
                       {   cc:=cc+1;
                           explore(u)}  }
```



SCCs: {G,H,I,J,K,L} {D} {C,F} {B,E} {A}

Part 2: BFS and shortest path problems

Graph Traversals

- DFS is a particular way to perform a traversal of the nodes
- Suitable for solving problems related to connectivity
- Many other applications exploit different visiting orders of the nodes
- E.g., for shortest path computations, we need a "breadth-first" approach

Breadth-First Search (BFS)

- BFS starts from a node u and explores the graph level by level
- We first visit all the neighbors of u (at distance 1 from u)
- We then visit nodes at distance 2
- And so on and so forth
- How can we implement a level-by-level traversal?
  - Using a FIFO queue

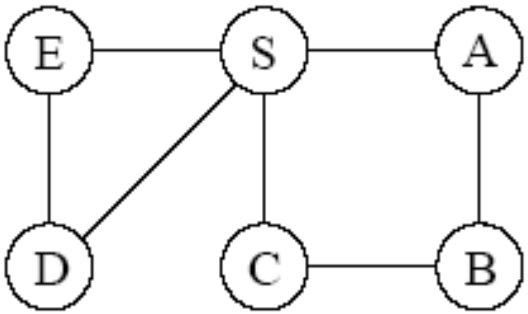
Breadth-First Search (BFS)

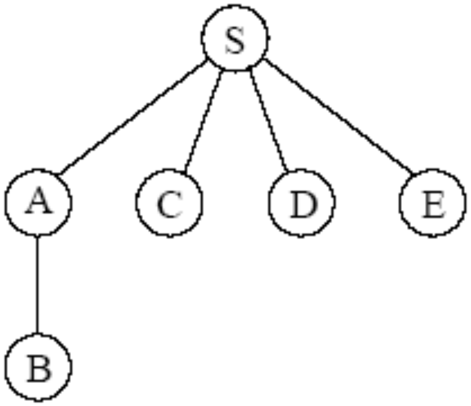
Instead of a stack (LIFO for DFS) it uses a queue Q (FIFO for BFS)

```
explore(u);
{  ENQUEUE(u); inq[u]:=true;
   while Q is non-empty do
   {    DEQUEUE  (u);
        visit(u);
        for each v ∈ Γ(u)  do
                if not inq(v)  then
                   { ENQUEUE (v); inq(v):=true    } } }
BFS (G);
{  clock:=1 ;
   for all u ∈ V do inq(u):= false;
   for all u ∈ V do if not inq(u) do explore(u)}
```

```
visit(u);
  { order(u):=clock
    clock:= clock+1 }
```

BFS - example





What is the complexity of BFS(G) ?

O(n+m):

- same as DFS(G)
- Again, we consider each edge 2 times

BFS – Unweighted shortest paths

- Suppose we want to compute the shortest path from a given node s to any node u ∈ V
- BFS is designed to do exactly this
- If we run explore(s), we will first visit all nodes that are at a distance of 1 from s, then all nodes at a distance 2,...
- We also need to keep track of the shortest paths
- We can simply store the parent of each node in the BFS tree

BFS – Unweighted shortest paths

(Unweighted) shortest paths

I: A graph G = (V, E), and a designated vertex s
Q: The shortest paths from s to all nodes

d(u) = length of shortest path to u

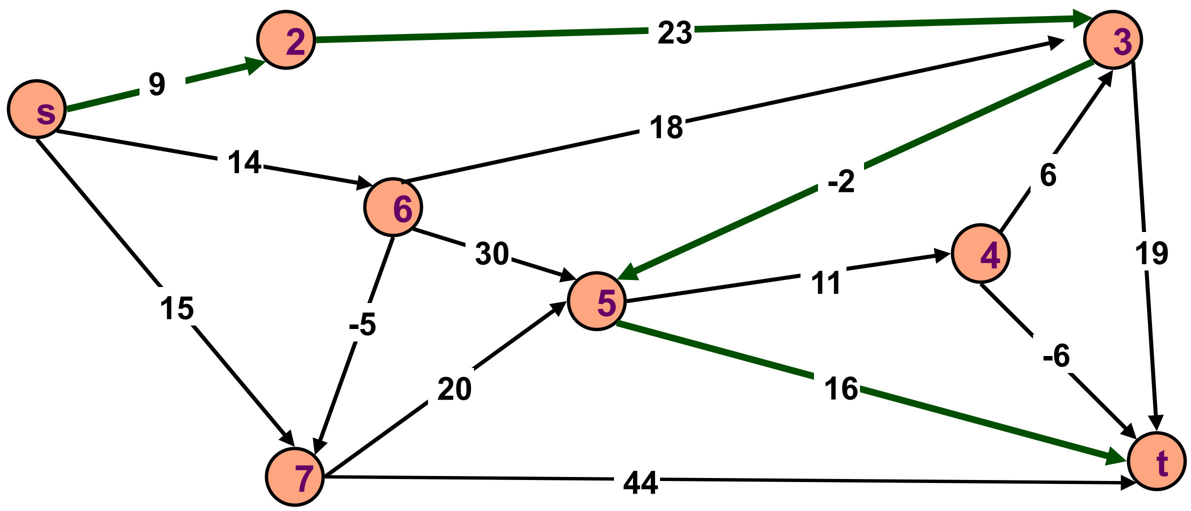
pred(u) = gives the predecessor of u in the shortest path

```
Algorithm Unweighted-Shortest-paths(G, s)
{ for all u ∈ V do
      {d(u):= ∞; pred(u):= null }
   d(s):=0;
   ENQUEUE(Q,s);
   while Q is non-empty do
        {      u:= DEQUEUE(Q);
               for each v ∈ Γ(u) do
                     if d(v) = ∞ then
                        { ENQUEUE(Q,v);
                          d(v):= d(u)+1;
                          pred(v):= u }   }   }
```

Weighted graphs

Directed weighted graph: G = (V, E, w)

- Edge cost w(e), e ∈ E
- For nodes s, t ∈ V, directed simple s-t path: p = {s 2 3 5 t}
- Cost of path p = {s 2 3 5 t} = w(p) = 9 + 23 - 2 + 16 = 44
- Goal: Find the shortest s-t path



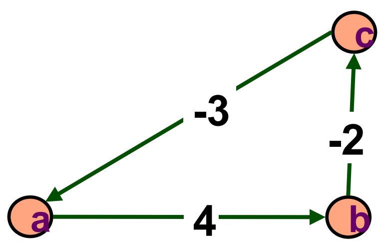
Shortest Paths Problem

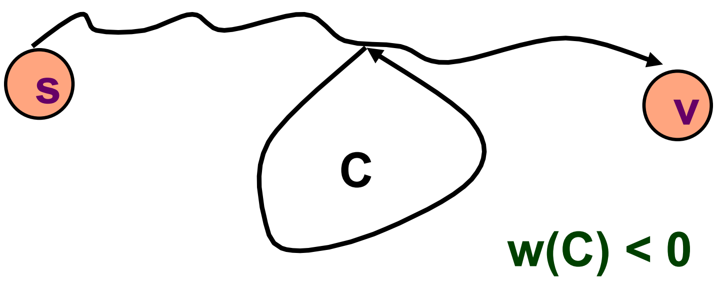
Single-source shortest paths

I: A weighted graph G = (V, E, w), and a designated vertex s
Q: The shortest paths from s to all nodes (the paths and their lengths)

Observation:

- If some path from s to v contains a negative cost cycle, then there is no shortest path





- In all other cases, there exists a shortest s-v path that is simple

(we can remove cycles without increasing the cost)

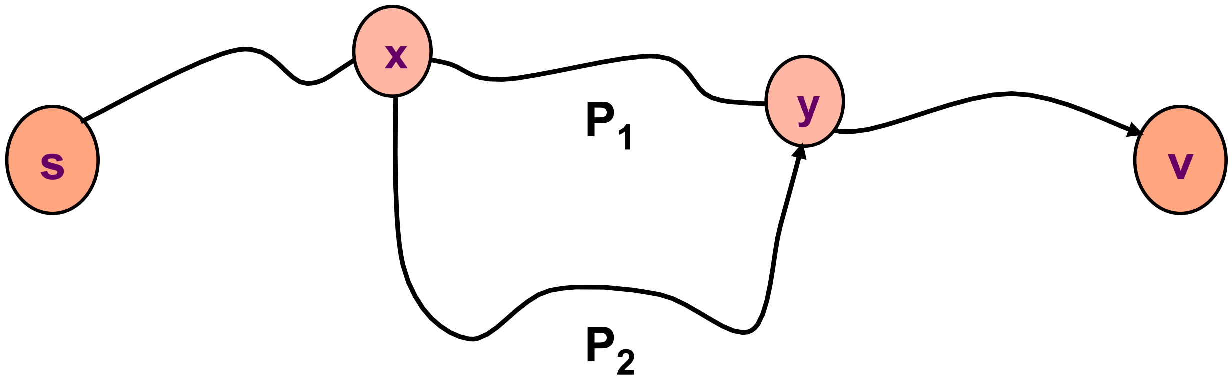
Assumption: the graph does not contain cycles of negative cost

Shortest Paths Property

- We will try to design a greedy algorithm
- For this we need to relate optimal solutions to subproblems with the optimal solution to the initial problem

- Claim: All sub-paths of shortest paths are shortest paths.
  - Let P_1 be an x-y sub-path of a shortest s-v path P.
  - Let P_2 be any x-y path.
  - $w(P_1) \leq w(P_2)$, otherwise P is not a shortest s-v path.

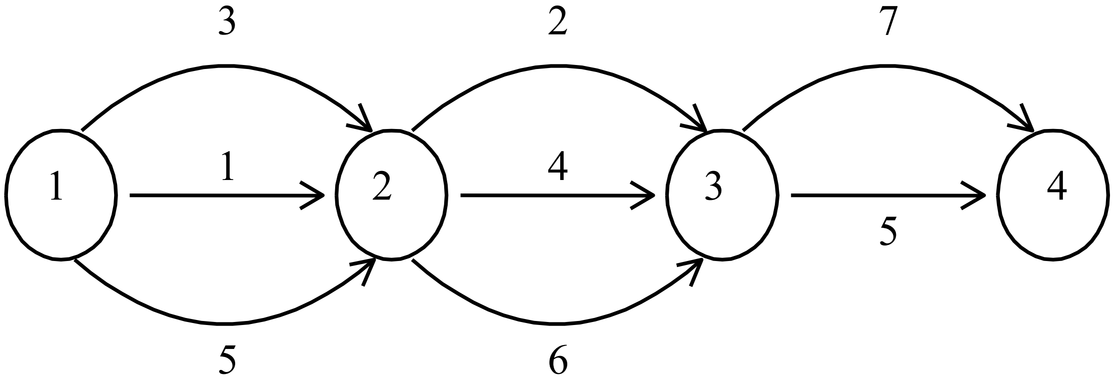
Optimal substructure property



Shortest Paths Problem

Special case: Single path multigraph

Find a shortest path from 1 to n



Apply the greedy method

1-n shortest path: 1 + 2 + 5 = 8

Shortest Paths Problem

- For general graphs, we will try to follow a similar approach
- The algorithm will work in rounds:
  - In each round, we will compute the shortest path to one new vertex
  - In round 1, we will find the shortest path to the vertex that has the smallest distance from s (i.e., pick the cheapest edge from s)
  - In round 2, we will find the shortest path to the vertex that has the 2^{nd} smallest distance from s
  - And so on...
- Hence, the algorithm keeps solving optimally larger and larger subproblems consisting of the vertices we have processed so far.

Shortest Paths Problem

In a few more details:

- We maintain a candidate shortest distance for each vertex
- Initially all set to infinity
- Suppose we have already found the shortest paths to a set S of vertices
- The next round will identify the next shortest distance from s to some vertex
- There are 2 cases for this:
  - Either this is the length of some edge (s, v)
  - Or the shortest path will have to go through one of the vertices in S
- Once we identify the next shortest distance, say from s to a node u, we check if going through u creates a shorter path for the rest of the nodes in V\S.

Shortest Paths Problem

- Summing up:
- We need to maintain in some data structure, the currently estimated shortest distances to all unprocessed nodes
- The minimum of these is a correct estimate (from the updates in the previous rounds)
- We then need to extract this minimum and update the current estimates for the remaining nodes (if necessary)
- But how can we extract efficiently the minimum in every round?
- The “right” data structure is a priority queue
  - Recall in the unweighted version, the right data structure is a FIFO queue

Priority Queues

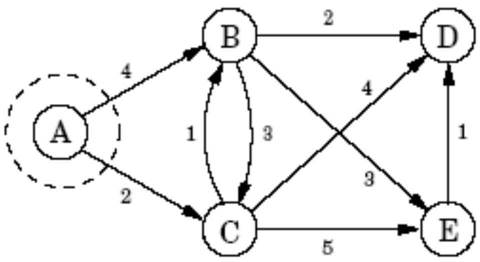
- A collection of elements with a key
- For our problem:
  - elements = nodes
  - key = distance from s

Operation	Description
`min(Q)`	returns (a pointer to) the element of Q with minimum key
`insert(Q,x)`	adds element x to the queue Q
`delete_min(Q)`	deletes the element of minimum key from Q
`union(Q’,Q’’)`	Combines queues Q’ and Q’’ into one
`decrease_key(Q,x)`	Updates the queue with a decreased key of element x
`Delete(Q,x)`	deletes element x from Q

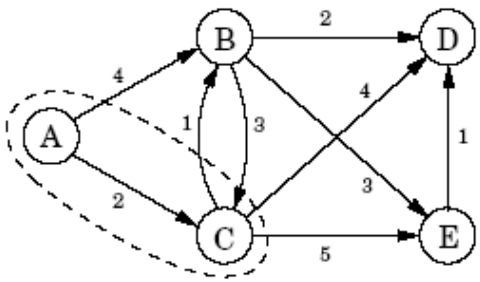
Dijkstra's Algorithm

```
Algorithm Dijkstra(G, s)
for each v ∈ V {d(v) = ∞;  pred(v) = null }
d(s) = 0;
Q = empty;
for each v ∈ V insert(Q,v)//using d(v) as the key
while Q is non-empty do
 { u = delete_min(Q)  //selects vertex with min. distance
   for each v ∈ Γ+(u) do
      if d(v) > d(u) + w(u,v) then
           { d(v) = d(u) + w(u,v);
             pred(v) = u;
             decrease_key(Q,v)}
}
```

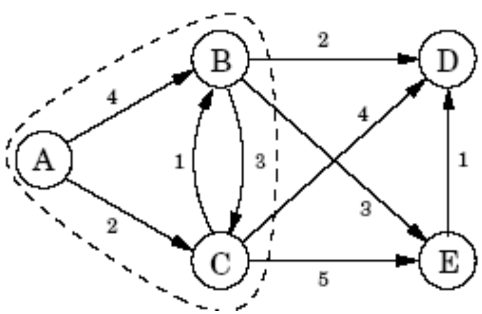
Dijkstra's Algorithm - Example



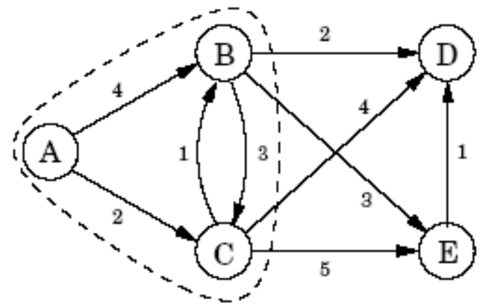
A: 0	D: ∞
B: 4	E: ∞
C: 2	



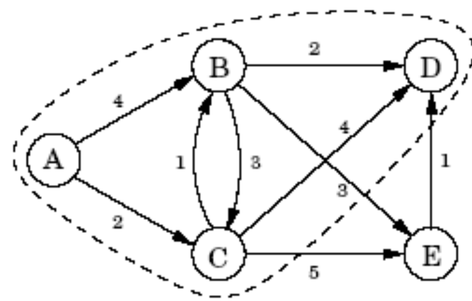
A: 0	D: 6
B: 3	E: 7
C: 2	



A: 0	D: 5
B: 3	E: 6
C: 2	



A: 0	D: 5
B: 3	E: 6
C: 2	



A: 0	D: 5
B: 3	E: 6
C: 2	

.

Dijkstra's Algorithm: Correctness

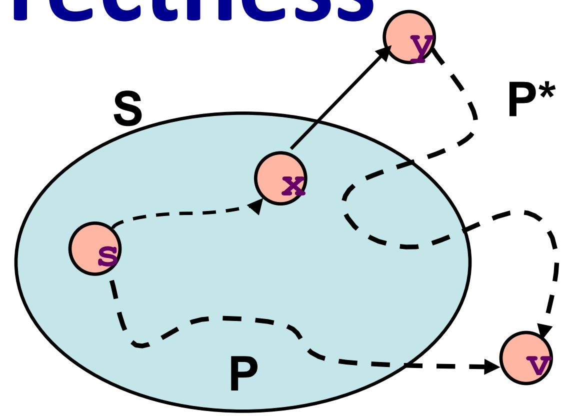
Theorem:

Upon termination, d(v) is the distance of the shortest s-v path for every $v \in V$, and the actual shortest path is obtained by following the value of pred(.) starting from v

Proof:

- Can be done by induction or by contradiction
- Let S = nodes that have already been extracted from the priority queue as the algorithm runs
- Initially $S = \emptyset$, eventually $S = V$
- We use induction on n to prove: For i=1,...,n, at the end of the i-th execution of the while loop, d(v) = shortest s-v path for every $v \in S$
  - Base case: end of step 1: S = {s}, d(s) = 0, trivially true
  - Hypothesis: Our claim holds up to step i
  - Induction step: Look at the end of step i+1

Dijkstra's Algorithm: Correctness



- Induction step: Let us look at the end of the (i+1)-th iteration
- Let v be the vertex Dijkstra's algorithm adds to S at this iteration
- Assume that the constructed path P is not an s-v shortest path and let P* be a shortest s-v path : w(P*) < w(P) = d(v)

Claim: P* must use an edge that leaves S, say (x, y)

Otherwise, by induction hypothesis, and by optimal substructure, P is optimal

It also holds that d(y) ≥ d(v), since the algorithm selected v and not y at this iteration

We have: d(y) = shortest s-y path ← Because when x is processed d(y) is updated

≤ shortest s-v path ← Assumes non-negative weights

< d(v), a contradiction

ATTENTION: Dijkstra’s algorithm works only for non-negative weights!

Priority Queues – Summary

Choices for implementing a priority queue: Binary heap (more standard), binomial heap, Fibonacci heap

Operation	Binary Worst	Binomial Worst	Binomial Amortized	Fibonacci Amortized
`min(Q)`	O(1)	O(1)	O(1)	O(1)
`insert(Q,x)`	O(logn)	O(logn)	O(1)	O(1)
`delete_min(Q)`	O(logn)	O(logn)	O(logn)	O(logn)
`union(Q',Q'')`	O(n)	O(logn)	O(logn)	O(1)
`decrease_key(Q,x)`	O(logn)	O(logn)	O(logn)	O(1)
`delete(Q,x)`	O(logn)	O(logn)	O(logn)	O(logn)

Dijkstra's Algorithm: Complexity

	# of operations	Queue implementation		
		Binary	Binomial*	Fibonacci*
Insert:	n	O(logn)	O(1)	O(1)
delete-min:	n	O(logn)	O(logn)	O(logn)
decrease-key:	m	O(logn)	O(logn)	O(1)

Binary heap: n logn + n logn + m logn ~ O(m logn)

Binomial heap: n · O(1) + n logn + m logn ~ O(m logn)

Fibonacci heap: n · O(1) + n logn + m · O(1) ~ O(m + n logn)

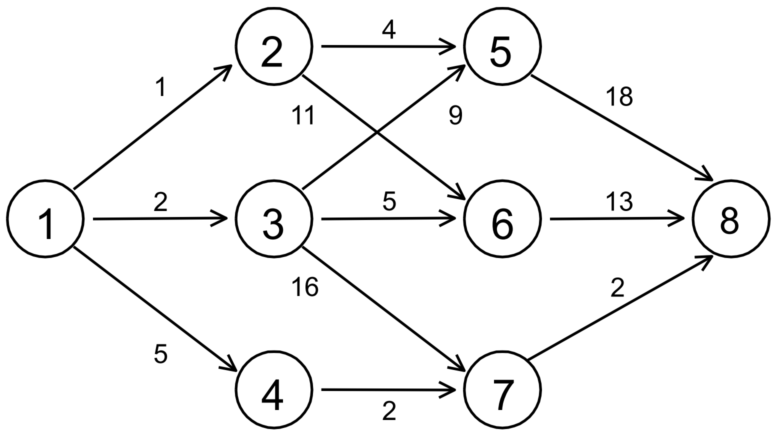
---

* amortized

Shortest paths in multistage graphs

Special case: Multistage graphs

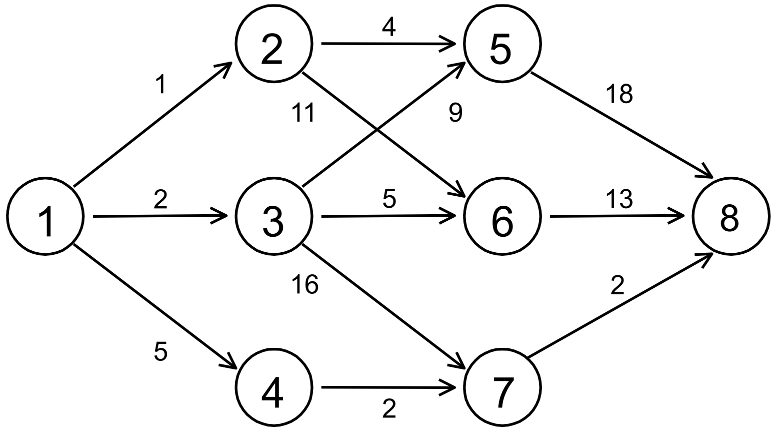
Multi-stage graph $G=(V_1,V_2,\ldots.V_k, E)$
$V_1 = \{s\}$, $V_k = \{t\}$
$E =\{(u,v) \mid u \in V_i, v \in V_{i+1}\ )$



Numbering of nodes level by level

Is the problem easier for this class?

Shortest paths in multistage graphs



The shortest path to u has to pass through a node in $\Gamma^-(u)$

d(1) = 0
d(2) = 1, d(3) = 2, d(4) = 5

d(5) = min{4+d(2), 9+d(3)} = min {4+1, 9+2} = 5

d(6) = min{11+d(2), 5+d(3)} = min{11+1, 5+2} = 7

d(7) = min{16+d(3), 2+d(4)} = min{16+2, 2+5} = 7

d(8) = min{18+d(5), 13+d(6), 2+d(7)} = min{18+5, 13+7, 2+7} = 9

Recurrences useful for a dynamic programming approach!

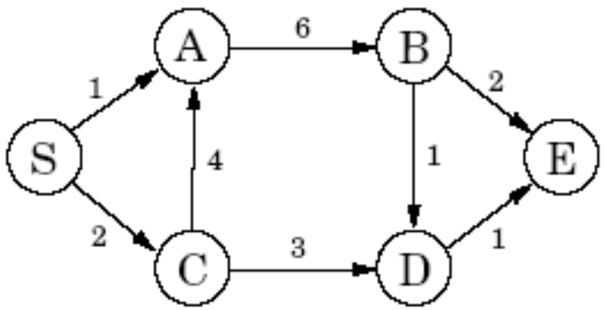
Shortest paths in multistage graphs

$$d(u) = \min_{v \in \Gamma^-(u)} \{w(v,u) + d(v)\},\ d(1) = 0$$

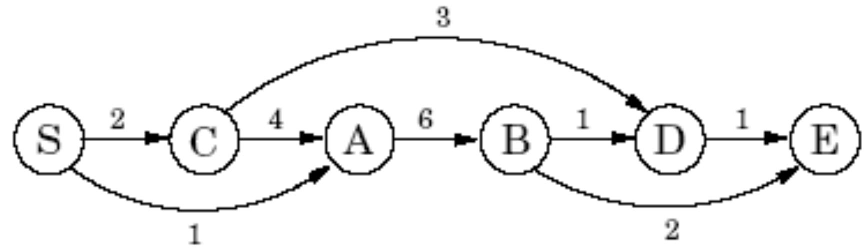
```
for each u ∈ V do { d(u}:= ∞; pred(u):= null }
d(1):=0;
for u:=2 to n do
   {for each v∈Γ⁻(u) do
      if w(v,u)+d(v) < d(u) then
         { d(u):=w(v,u)+d(v);
           pred(u):=v; }   }
```

- Complexity?
  - O(n+m): like the analysis of DFS/BFS
- Negative weights?
  - No problem!

Shortest paths in DAGs



Dag G=(V,E)



Topological sorting of G

- We know that there are no negative weight cycles in such graphs
- Multi-stage graphs are a special case of DAGs
- Same dynamic programming approach can be applied for DAGs after we find a topological sorting of the vertices

$$d(u) = \min_{v \in \Gamma^{-}(u)} \{w(v,u) + d(v)\}, \; d(s) = 0$$

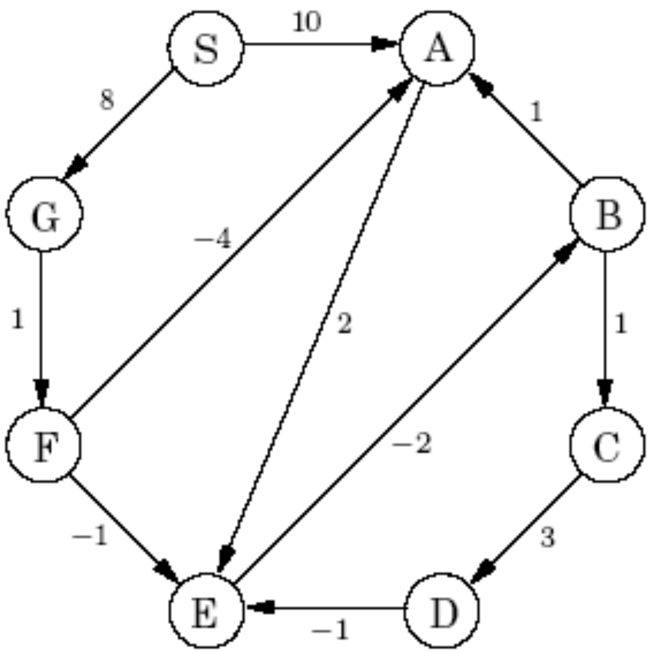
Shortest paths in DAGs

$$d(u) = \min_{v \in \Gamma^-(u)} \{w(v,u) + d(v)\},\ d(s) = 0$$

```
for each u ∈ V do { d(u}:= ∞; pred(u):= null }
d(s):=0;
Find a topological sorting of G
for each u ∈ V-{s} in topological order do
    {for each v ∈ Γ⁻(u) do
        if w(v,u)+d(v) < d(u) then
            { d(u):= w(v,u)+d(v) ;
              pred(u):= v; }    }
```

- Complexity? O(n+m)
- Negative weights? No problem!
- Same arguments as for multi-stage graphs

Shortest paths with negative weights

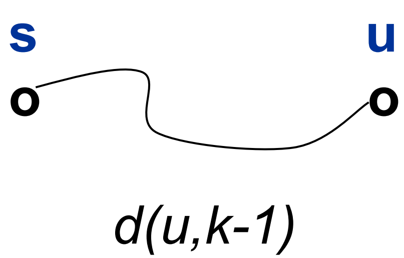


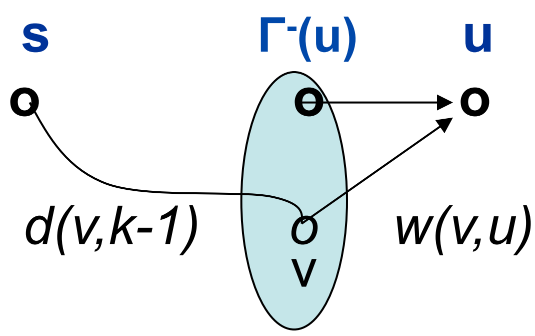
- Consider now a general weighted directed graph G = (V, E, w) possibly with negative weights
- Dijkstra does not work
- Can anything else work?

Shortest paths with negative weights

We will resort again to dynamic programming

d(u,k): shortest path from node s to node u **using at most k edges**

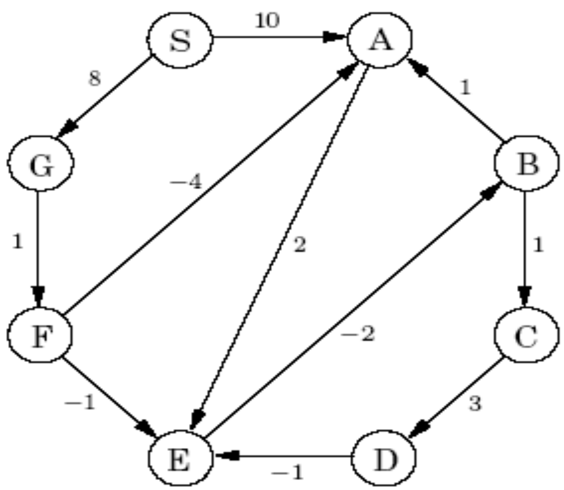




$$d(u,k) = \min\{d(u,k-1), \min_{v \in \Gamma^-(u)} \{w(v,u) + d(v,k-1)\}\}, \quad d(s,0) = 0$$

Note: Any path has length at most |V|-1 = n-1
Hence: Suffices to do n-1 times the updates that Dijkstra does!

Shortest paths with negative weights

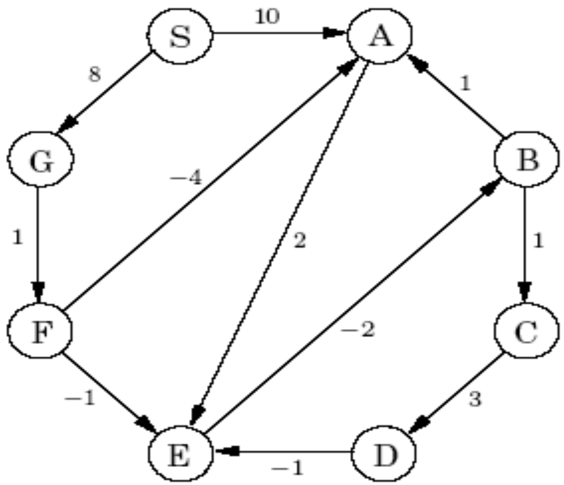


	Iteration k							
Node	0	1	2	3	4	5	6	7
S								
A								
B								
C								
D								
E								
F								
G								

Idea of the algorithm:

- Use d(u) as the current estimate, initially set to ∞
- Think of it as gradually filling up a n x (n-1) array
- Iteration 1: shortest paths of length 1, i.e., only for vertices that s directly connects to
- ...
- Iteration k: update d(u) to be equal to the shortest s-u path with at most k edges
- Continue until iteration n-1

Shortest paths with negative weights



	Iteration k							
Node	0	1	2	3	4	5	6	7
S								
A								
B								
C								
D								
E								
F								
G								

Correctness:

- Before the beginning of iteration k, we have already updated d(u) for every u ∈ V, to equal the shortest s-u path with at most k-1 edges
- Hence, we can correctly update d(u) using our recurrence

Update operation:

$$d(u) = \min\{d(u), \min_{v \in \Gamma^-(u)} \{w(v,u) + d(v)\}\}, \quad d(s) = 0$$

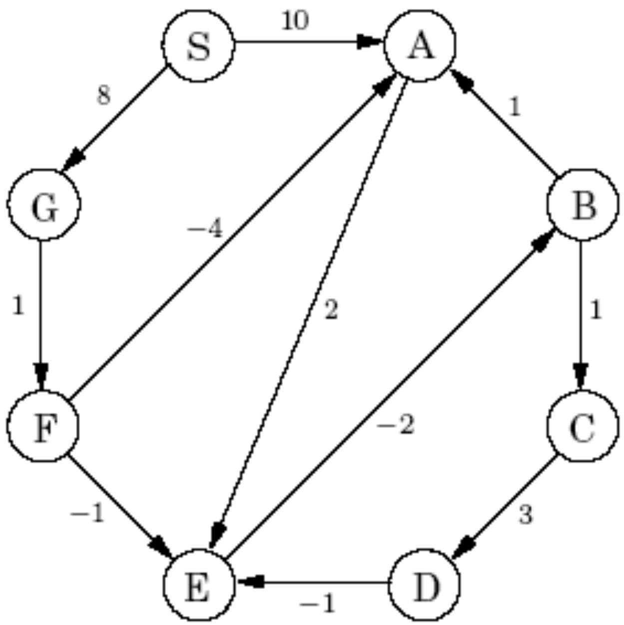
Shortest paths with negative weights

Algorithm Bellman- Ford(G, s)

```
for each u ∈ V do { d(u}:= ∞; pred(u)=null }
d(s):=0;
for k:=1 to n-1 do
   for each u ∈ V-{s} do
      {for each v ∈ Γ⁻(u) do
         if w(v,u) + d(v) < d(u) then
            { d(u):= w(v,u)+d(v) ;
              pred(u):= v; }    }
```

Complexity: O(n·m) (why?)

Shortest paths with negative weights



	Iteration							
Node	0	1	2	3	4	5	6	7
S	0	0	0	0	0	0	0	0
A	∞	10	10	5	5	5	5	5
B	∞	∞	∞	10	6	5	5	5
C	∞	∞	∞	∞	11	7	6	6
D	∞	∞	∞	∞	∞	14	10	9
E	∞	∞	12	8	7	7	7	7
F	∞	∞	9	9	9	9	9	9
G	∞	8	8	8	8	8	8	8

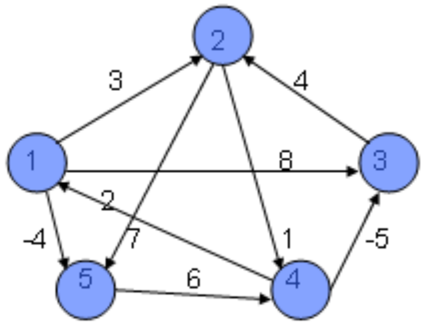
- Speeding up convergence: stop if in a round no update occurred
- Detecting negative cycles: allow one more iteration

All pairs shortest paths

<u>Single-source shortest paths</u>

I: A weighted graph G = (V, E, w)

Q: A shortest path for every pair of nodes



- Run Bellman-Ford n times $O(n^2m)$
- Can we do better?

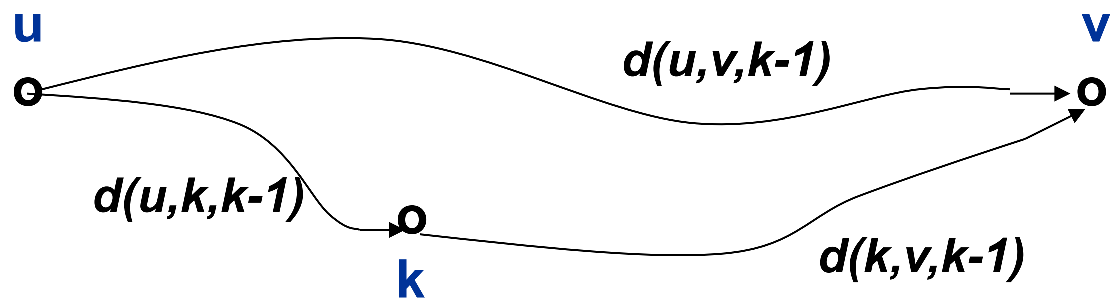
First, we extend the weight function to all pairs:

$$w(u,v) = \begin{cases} 0 & \text{if } u = v \\ w(e) & \text{if } u \neq v \text{ and } e = (u,v) \in E \\ \infty & \text{if } u \neq v \text{ and } e = (u,v) \notin E \end{cases}$$

All pairs shortest paths

- We will again use a dynamic programming approach
- but a different one from before
- Suppose we name the vertices as 1, 2, ..., n

d(u, v, k):= shortest path from node u to node v using <u>only</u> nodes {1,2,..., k} as intermediates



$$d(u,v,0) = w(u,v)$$

$$d(u,v,k) = \min\{d(u,v,k-1), d(u,k,k-1) + d(k,v,k-1)\}$$

All pairs shortest paths

$$d(u,v,0) = w(u,v)$$

$$d(u,v,k) = \min\{d(u,v,k-1), d(u,k,k-1) + d(k,v,k-1)\}$$

We will gradually find *d(u,v,0), d(u,v,1), d(u,v,2), …, d(u,v,n)*

Idea of the algorithm:

- Use d(u, v) as the current estimate, initially set to w(u, v)
- Think of it as filling up a n x n array for each k
- Iteration 1: update d(u, v) to equal the shortest path length using node 1 as intermediate
- ...
- Iteration k: update d(u, v) to equal the shortest u-v path with {1, 2, ...,k} as intermediates
- Continue until iteration n

$$d(u,v) = \min\{d(u,v), d(u,k) + d(k,v)\}$$

All pairs shortest paths

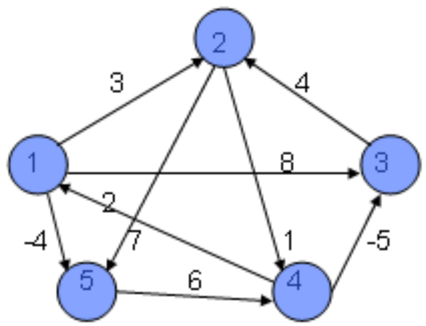
Algorithm Floyd-Warshall(G)

```
for u:=1 to n do
    for v:=1 to n do
       { d(u,v):= w(u,v);
         pred(u,v):= null  }
for k:=1 to n do
    for u:=1 to n do
        for v:=1 to n do
            if d(u,v) + d(k,v) < d(u,v) then
                  {  d(u,v):= d(u,k) + d(k,v);
                     pred(u,v):= k }
```

`pred(u,v)`: to be used for extracting the u-v shortest path

Complexity: $O(n^3)$

All pairs shortest paths



$d^{(0)}=$

0	3	8	∞	-4
∞	0	∞	1	7
∞	4	0	∞	∞
2	∞	-5	0	∞
∞	∞	∞	6	0

$d^{(1)}=$

0	3	8	∞	-4
∞	0	∞	1	7
∞	4	0	∞	∞
2	5	-5	0	-2
∞	∞	∞	6	0

$d^{(2)}=$

0	3	8	4	-4
∞	0	∞	1	7
∞	4	0	5	11
2	5	-5	0	-2
∞	∞	∞	6	0

$d^{(3)}=$

0	3	8	4	-4
∞	0	∞	1	7
∞	4	0	5	11
2	-1	-5	0	-2
∞	∞	∞	6	0

$d^{(4)}=$

0	3	-1	4	-4
3	0	-4	1	-1
7	4	0	5	3
2	-1	-5	0	-2
8	5	1	6	0

All pairs shortest paths

Extract the u-v shortest path

pred(u,v)=k : the u-v shortest path passes through node k

Consult pred(u,v):

- If pred(u,v)= null then the u-v shortest path is the edge (u,v)
- Otherwise compute recursively the shortest paths:
    - from u to pred(u,v), and
    - from pred(u,v) to v

All pairs shortest paths

Extract the u-v shortest path

```
Path (u,v);
{ if pred(u,v)=null then output (u,v)
     else {Path(u,pred(u,v)), Path(pred(u,v),v) } }
```

Find the shortest path from vertex 2 to vertex 3.

2..3	Path(2, 3)	$pred[2,3] = 4$	
2..4..3	Path(2, 4)	$pred[2,4] = 5$	
2..5..4..3	Path(2, 5)	$pred[2,5] = nil$	$Output$(2,5)
25..4..3	Path(5, 4)	$pred[5,4] = nil$	$Output$(5,4)
254..3	Path(4, 3)	$pred[4,3] = 6$	
254..6..3	Path(4, 6)	$pred[4,6] = nil$	$Output$(4,6)
2546..3	Path(6, 3)	$pred[6,3] = nil$	$Output$(6,3)
25463			