

# Breadth First Search

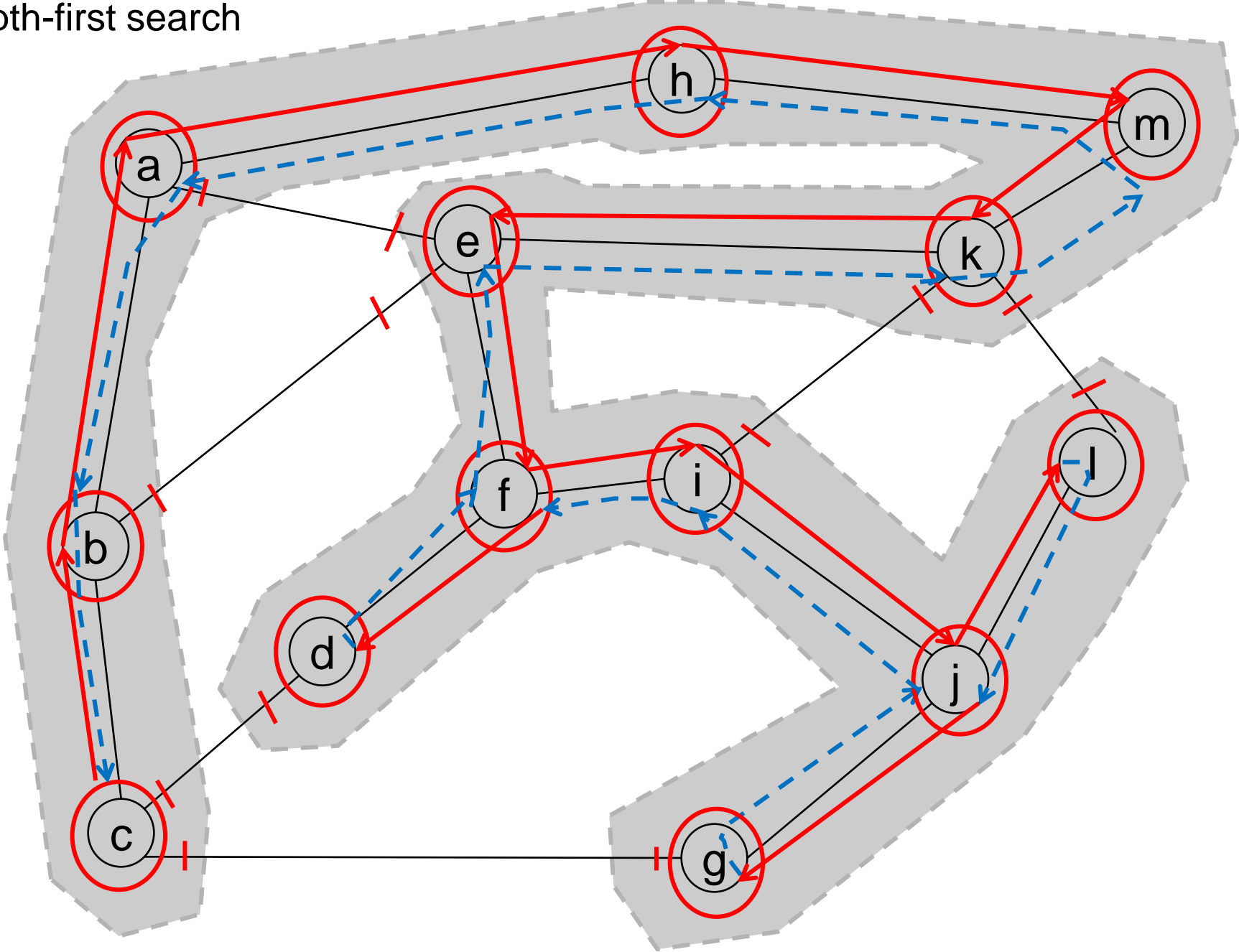
## Directed Graphs

## Transitive Closure



For a simple undirected graph with  $n$  vertices, what is the maximum number of edges?

Depth-first search

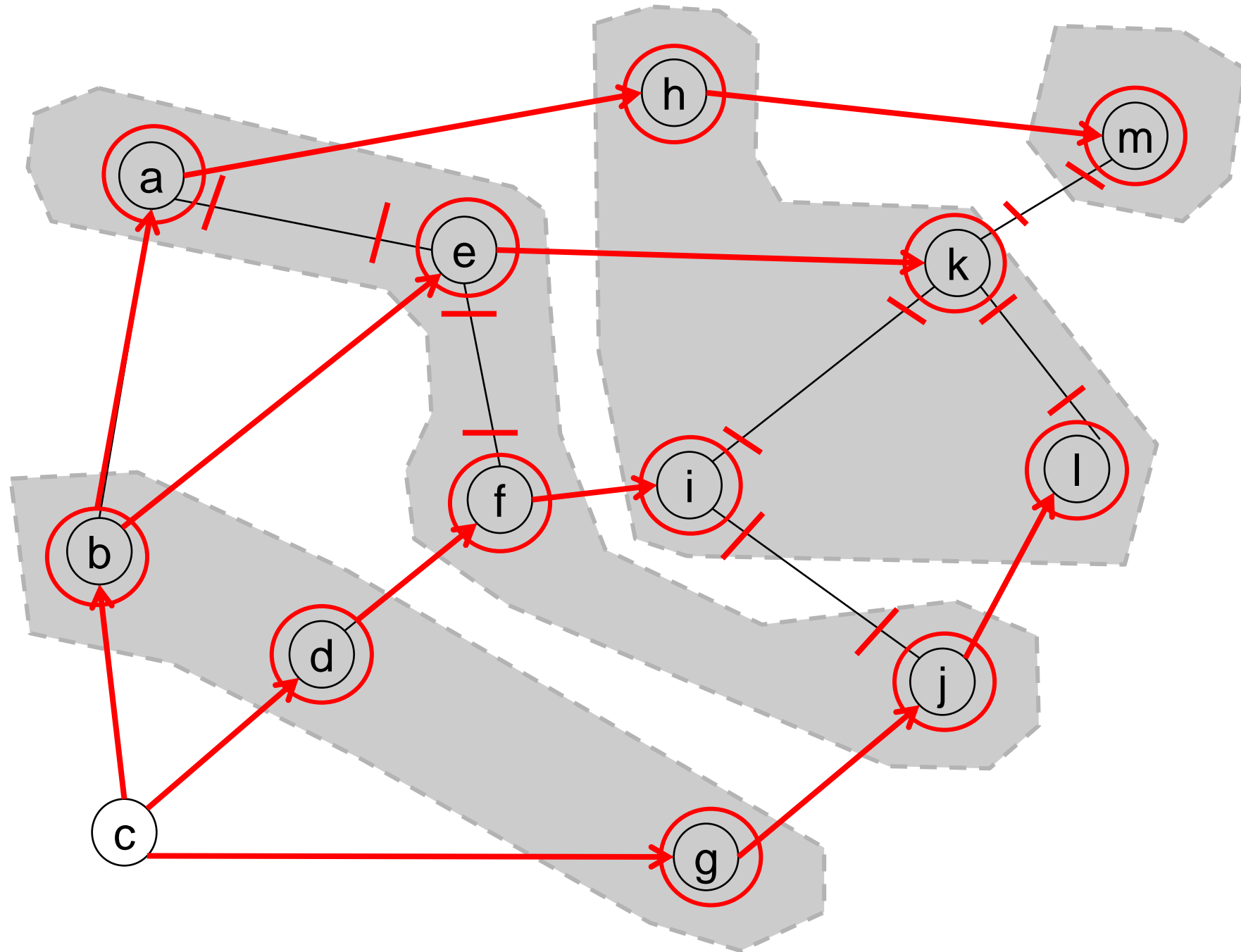


# Breadth-first Traversal

Depth-first search was easy to implement, and is fast.

But some vertices which are very close to the start vertex are not discovered until late in the traversal.

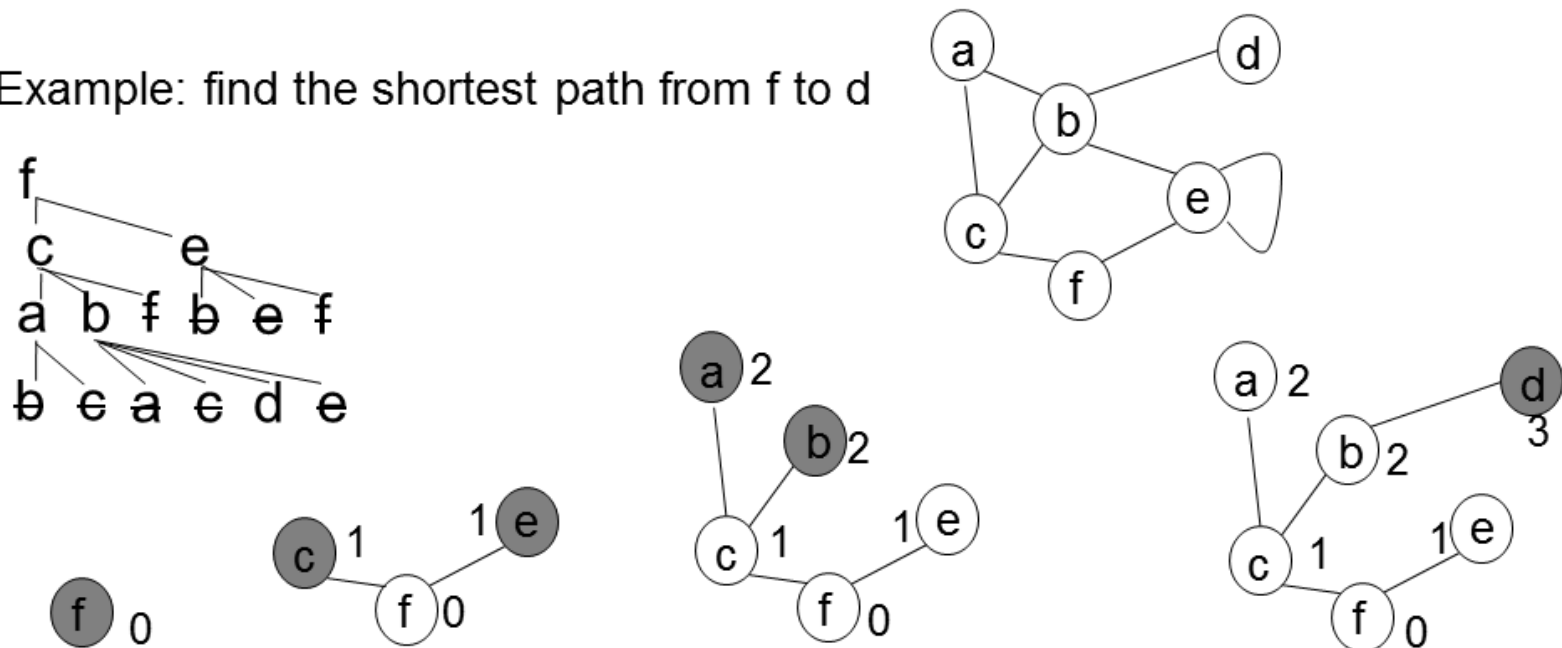
Breadth-first traversal visits all vertices that are 1 hop away from the start before those that are 2 hops away, etc.



# Finding the shortest path (informal)

We start at the start vertex. We look at all vertices that are adjacent that we haven't seen before. If one of them is the end vertex, we stop; else we mark as being 1 hop away. We then take each 1-hop vertex in turn, and look at all vertices that are adjacent to them (ignoring any we have seen before). If one of them is the end vertex, we stop; else we mark as 2 hops. We then take each 2-hop vertex in turn, ... and so on. Either we find the end vertex via the shortest path, or the start vertex cannot be connected to the end vertex.

Example: find the shortest path from f to d



```
def breadthfirstsearch(self, v):  
    marked = {v:None}  
    level = [v]  
    while len(level) > 0:  
        nextlevel = []  
        for w in level:  
            for e in self.get_edges(w):  
                x = e.opposite(w)  
                if x not in marked:  
                    marked[x] = e  
                    nextlevel.append(x)  
        level = nextlevel  
    return marked
```

# Breadth First Search: properties

1. Breadth-first search computes, for each vertex, the path with the fewest number of edges from the start vertex
2. Breadth-first search has worst-case time complexity  $O(n+m)$

Proof:

(by a similar argument to that for Depth-first search – each vertex is expanded at most once, and each edge is examined at most twice)



# Directed Graphs

The ADT for Directed graphs is essentially the same as for undirected, except we treat the order of the vertices in an Edge as significant,

Edge

get\_start()

get\_end()

and we require the following methods for the Graph:

in_degree(x):	return the in-degree of vertex x
out_degree(x):	return the out-degree of vertex x
get_in_edges(x):	return a list of all edges pointing in to x
get_out_edges(x):	return a list of all edges pointing away from x

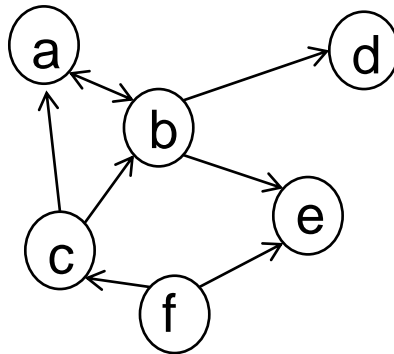
For a simple *directed* graph with  $n$  vertices, what is the maximum number of edges?

# Directed graph: traversals

The basic algorithms for depth-first and breadth-first can be easily adapted to find all vertices reachable from a start vertex.

But note that a path from X to Y does not imply that there is a path from Y to X

The definition of a connected component no longer applies.



Notation: if  $R$  is a relation  $R \subseteq A \times A$ , we will say  $R^{(2)} = R \circ R$ ,  $R^{(3)} = R \circ R^{(2)}$ , etc and so  $R^{(n)} = R \circ R^{(n-1)}$ .

Let  $A$  be a set s.t.  $|A| = n$ , and let  $R$  be a relation  $R \subseteq A \times A$   
 The **transitive closure** of  $R$  is  $R \cup R^{(2)} \cup \dots \cup R^{(n-1)}$

Example:  $A = \{1,2,3,4\}$

$R = \{(1,2), (2,4)\}$

transitive closure of  $R = \{(1,2), (2,4), (1,4)\}$

$R = \{(1,2), (2,4)\}$

$R^{(2)} = R \circ R = \{(1,4)\}$

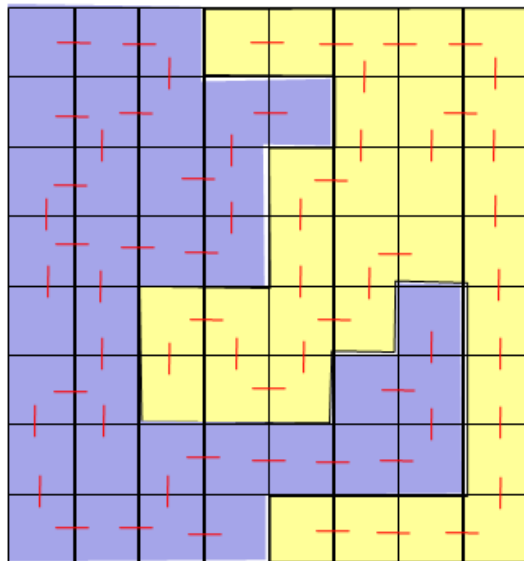
$R^{(3)} = R \circ R^{(2)} = \{\}$

i.e. add in to  $R$  every pair  $(x,y)$   
 needed to make  $R$  transitive

transitive closure of  $R = R \cup R^{(2)} \cup R^{(3)} = \{(1,2), (2,4), (1,4)\}$

Example: consider the problem of finding a path for a robot from one area of a factory to another.

Or analysing secure zones in an airport – which areas are reachable from which other areas, without going through security control?



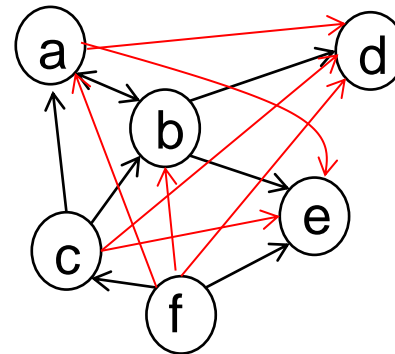
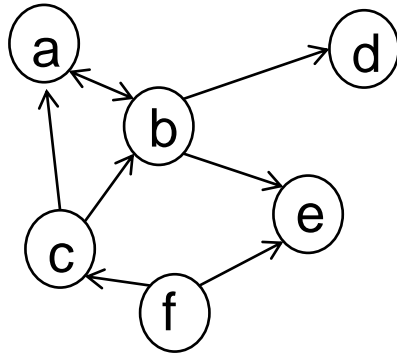
The red lines indicate doors that open between zones.

The transitive closure is the set of rooms that can be reached from each starting point.

# Transitive Closure of a graph

Given a graph  $G = (V, E)$ , the transitive closure  $G^* = (V, t(E))$

where an edge  $(x, y) \in t(E)$  if and only if there is a path from  $x$  to  $y$  in  $G$



What is the transitive closure of this graph?

# Computing the transitive closure

Version 1 (pseudocode)

Input: graph  $G = (V, E)$

Output: graph  $G^* = (V, t(E))$

```
G* = G.copy()
```

```
for each vertex v in G
```

```
    tree = depthfirstsearch(v)
```

```
    for each x in tree
```

```
        add edge v-x into G*
```

Analysis: a single iteration round the for loop has one run of depthfirstsearch ( $O(n+m)$ ), and we may add up to  $n$  edges into  $G^*$ , so  $O(n+m)$  for one iteration. There are  $n$  vertices in  $G$ , so  $O(n(n+m))$

# Transitive closure (v2)

To simplify notation, if  $G = (V, E)$  and  $e \in E$ , we say  $e \in G$

For a given graph  $G$ ,  
assign a unique number from 1 to  $n$  to each vertex, and define:

$$G^{(0)} = G$$

$$G^{(i)} = G^{(i-1)} + \{(x, y) : (x, v_i) \in G^{(i-1)} \text{ and } (v_i, y) \in G^{(i-1)}\}, \text{ for } i > 0$$

$G^{(i)}$  contains as edges all pairs  $(x, y)$  where there is a path with from  $x$  to  $y$  in  $G$  containing only intermediate vertices in  $\{v_1, v_2, \dots, v_i\}$ .

For a graph with  $n$  vertices, the transitive closure  $G^* = G^{(n)}$ .



# The Floyd Warshall Algorithm to compute transitive closure

FloydWarshall

Input: A directed graph  $G = (V, E)$ , with  $|V| = n$

Output:  $G^*$

$G^{(0)} = G$

for  $k$  from 1 to  $n$

$G^{(k)} = G^{(k-1)}$

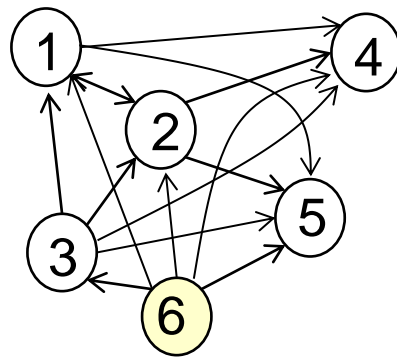
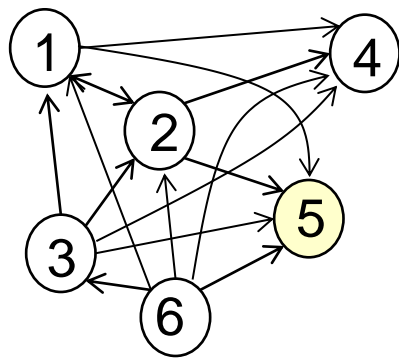
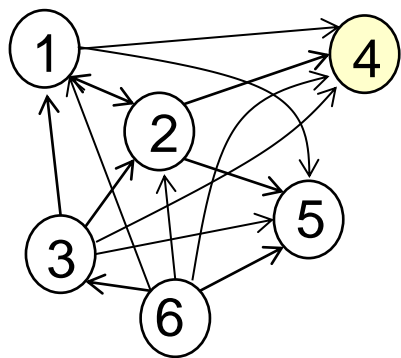
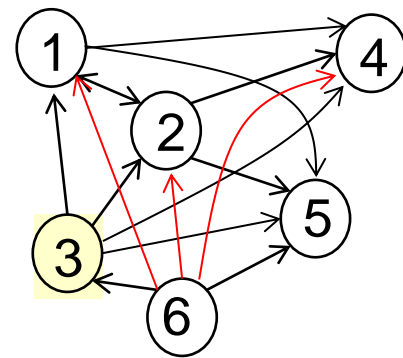
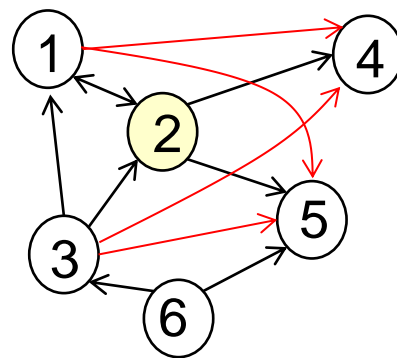
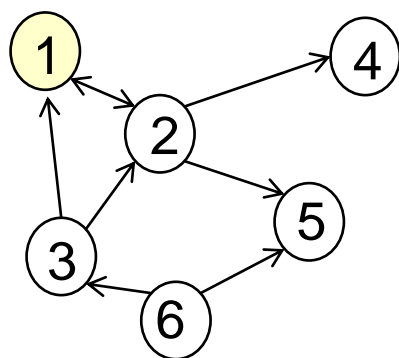
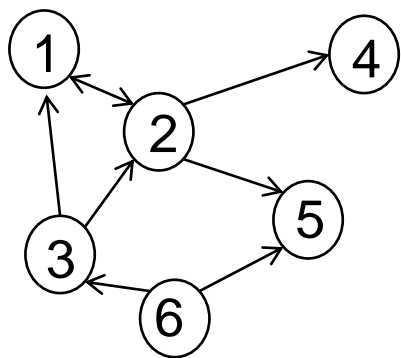
        for all pairs  $(v_i, v_j)$  s.t.  $i \neq j, i \neq k, j \neq k$

            if  $(v_i, v_k) \in G^{(k-1)}$  and  $(v_k, v_j) \in G^{(k-1)}$

                add  $(v_i, v_j)$  to  $G^{(k)}$

return  $G^{(n)}$

Analysis: each inner for loop considers  $(n-1)*(n-2)$  pairs, checks if they are in a graph, and maybe adds them to a graph. We do the outer loop  $n$  times. So, if we can check and add edges in  $O(1)$ , FW runs in worst case time  $O(n^3)$



The first transitive closure algorithm has time complexity  $O(n(n+m))$ .  
Floyd Warshall has time complexity  $O(n^3)$

For dense graphs, the number of edges is  $O(n^2)$ , and so the two algorithms have the same complexity. In these cases, Floyd Warshall is often faster (the lower order terms are better)

For sparse graphs, the number of edges is  $O(n)$ , and so Floyd Warshall has a higher time complexity.

Also, Floyd Warshall performs faster when the graph is represented as an adjacency matrix, while transitive closure v1 performs better when the graph is an adjacency list or adjacency map.

```
def floydwarshall(self):
    gstar = copy.deepcopy(self)
    vs = gstar.vertices()
    n = len(vs)
    for k in range(n):
        for i in range(n):
            if (i != k and
                gstar.get_edge(vs[i],vs[k]) != None):
                for j in range(n):
                    if (i != j and k != j
                        and gstar.get_edge(vs[k],vs[j]) is not None):
                        if gstar.get_edge(vs[i],vs[j]) == None:
                            gstar.add_edge(vs[i],vs[j],1)
    return gstar
```

# Next lecture

Directed Acyclic Graphs