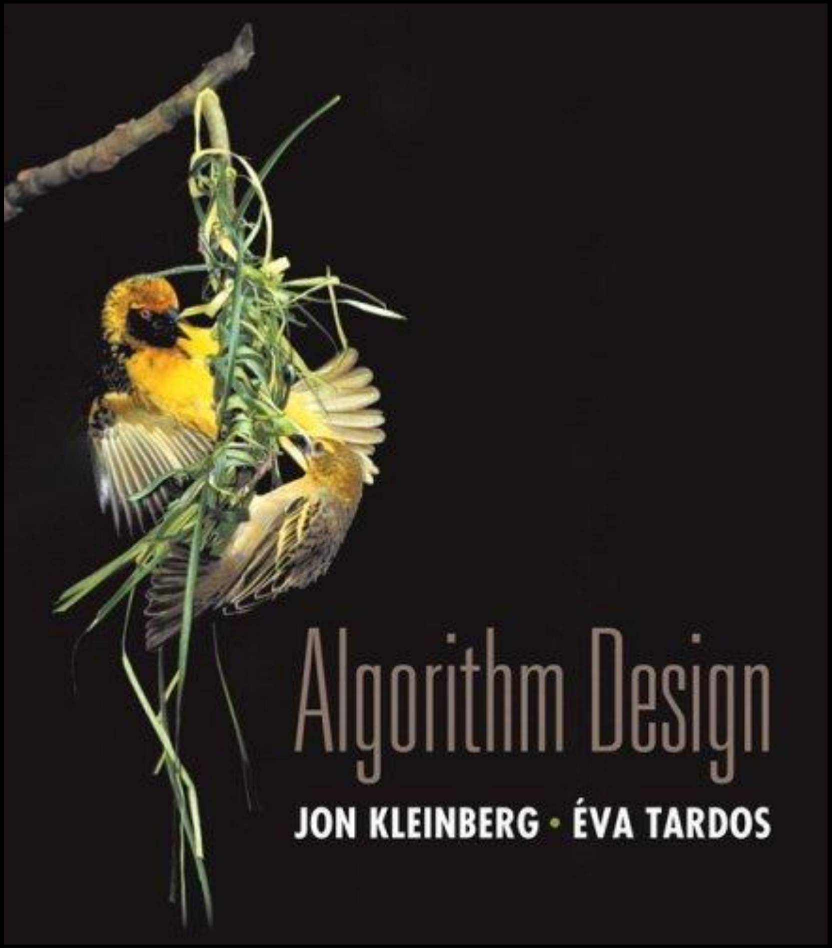


Chapter 3

Graphs

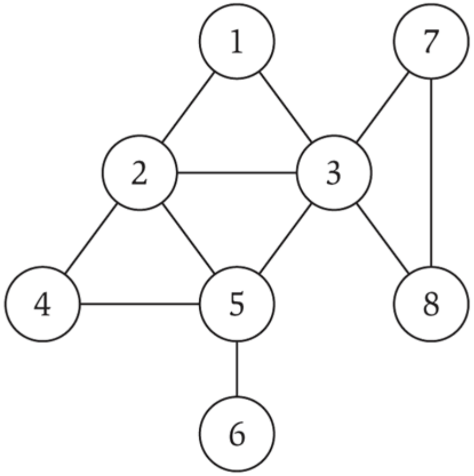
Slides by Kevin Wayne.

3.1 Basic Definitions and Applications

Undirected Graphs

Undirected graph. $G = (V, E)$

- V = nodes (non-empty)
- E = edges between pairs of nodes.
- Captures pairwise relationship between objects.
- Graph size parameters: $n = |V|$, $m = |E|$.



$V = \{ 1, 2, 3, 4, 5, 6, 7, 8 \}$

$E = \{ 1\text{-}2, 1\text{-}3, 2\text{-}3, 2\text{-}4, 2\text{-}5, 3\text{-}5, 3\text{-}7, 3\text{-}8, 4\text{-}5, 5\text{-}6 \}$

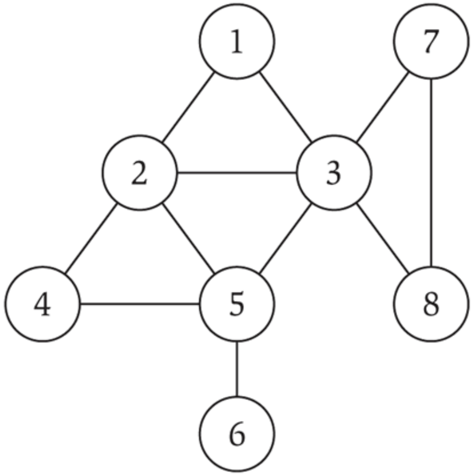
$n = 8$

$m = 11$

Undirected Graphs

Undirected graph. G = (V, E)

- u and v are adjacent (neighbors) in G iff there is an edge between u and v in G
- The degree d(u) of a vertex u is the number of neighbors of u



1 and 3 are adjacent

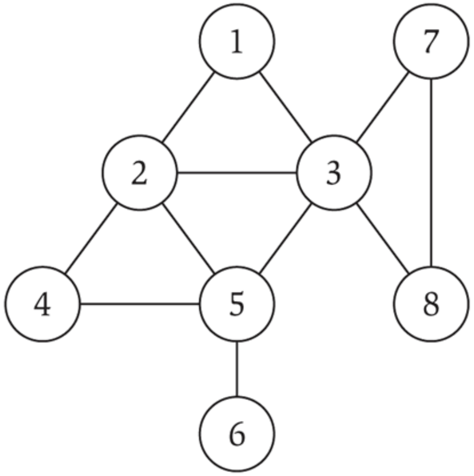
2 and 8 are not adjacent

d(3)=5

d(4)=2

Undirected Graphs

Important Property: For every graph G, the sum of degrees of G equals twice the number of edges.



m=11

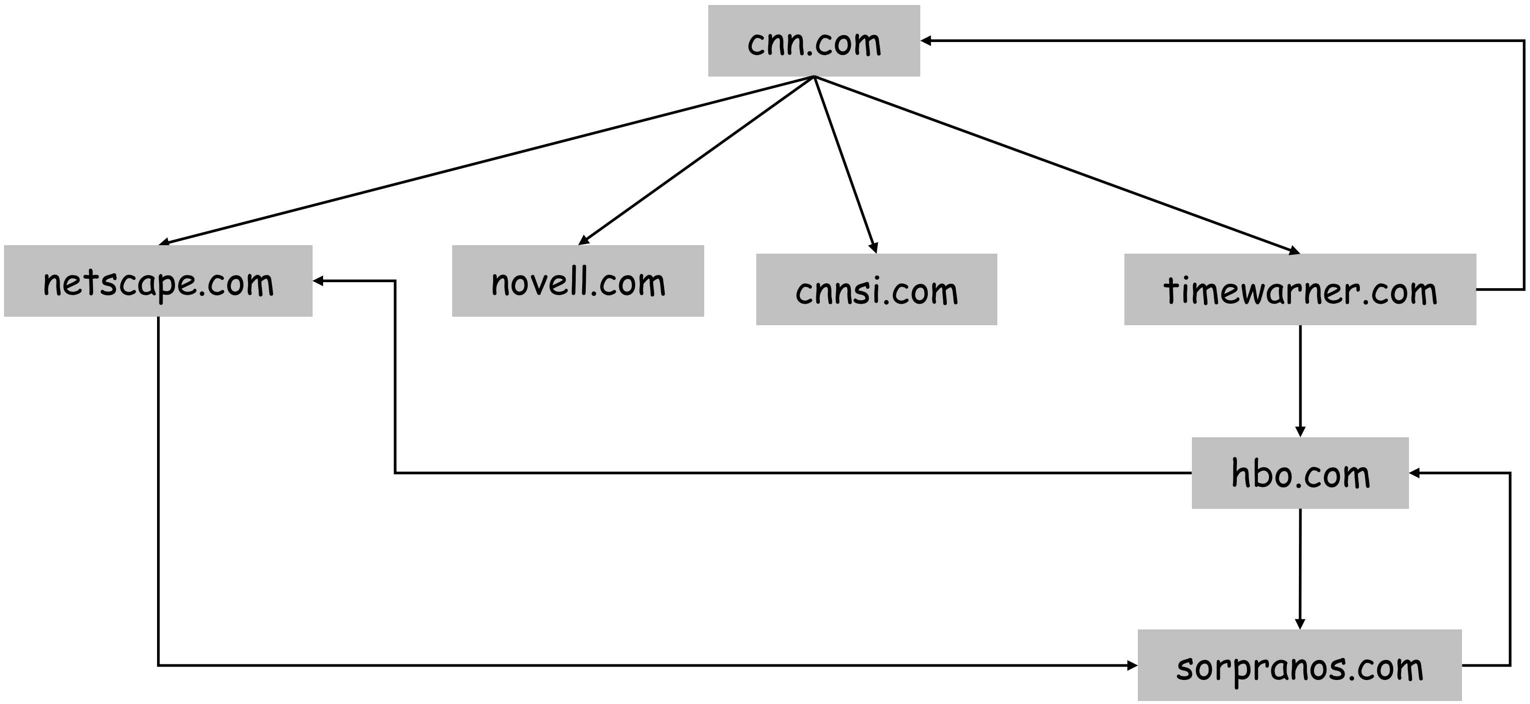
Some Graph Applications

Graph	*Nodes*	*Edges*
transportation	street intersections	highways
communication	computers	fiber optic cables
World Wide Web	web pages	hyperlinks
social	people	relationships
food web	species	predator-prey
software systems	functions	function calls
scheduling	tasks	precedence constraints
circuits	gates	wires

World Wide Web

Web graph.

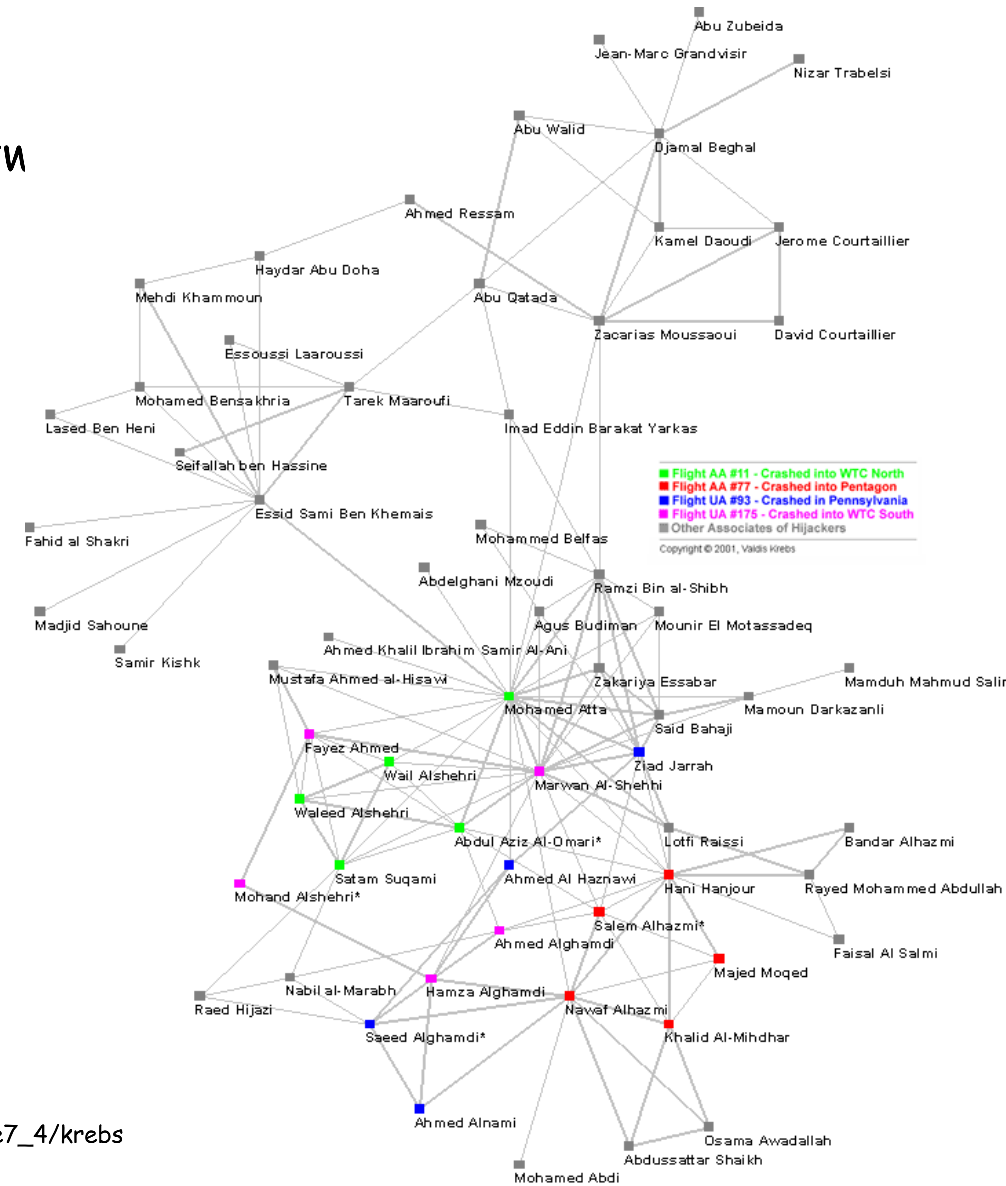
- Node: web page.
- Edge: hyperlink from one page to another.



9-11 Terrorist Network

Social network graph.

- Node: people.
- Edge: relationship between tw

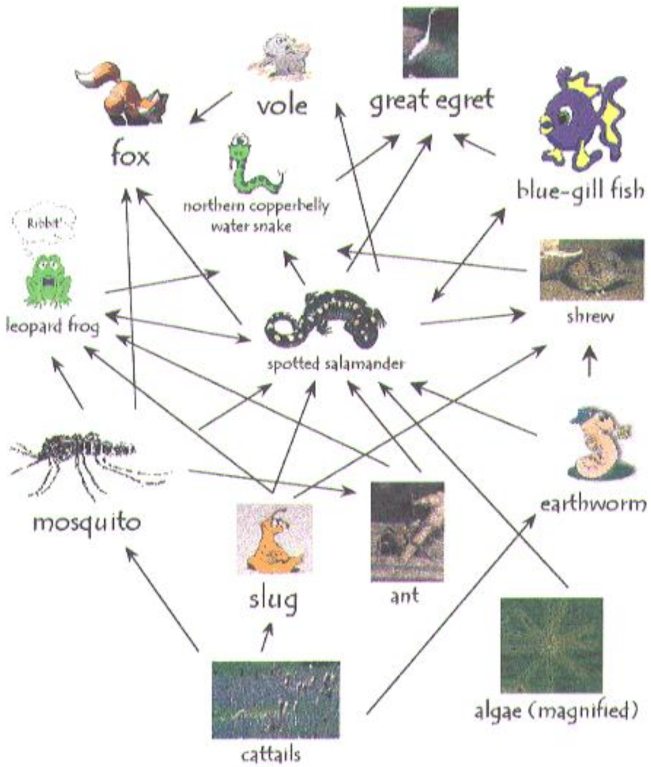


Reference: Valdis Krebs, http://www.firstmonday.org/issues/issue7_4/krebs

Ecological Food Web

Food web graph.

- Node = species.
- Edge = from prey to predator.

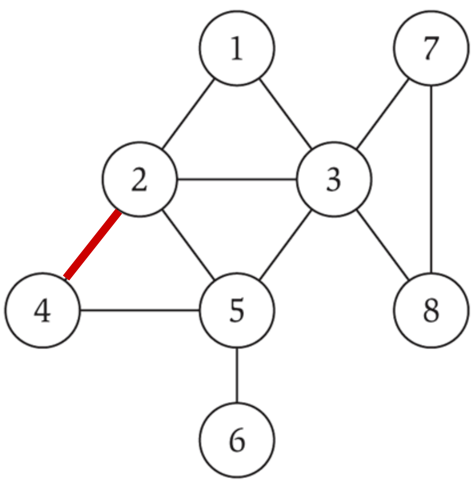


Reference: http://www.twingroves.district96.k12.il.us/Wetlands/Salamander/SalGraphics/salfoodweb.giff

Graph Representation: Adjacency Matrix

Adjacency matrix. n-by-n matrix with A_{uv} = 1 if (u, v) is an edge.

- Two representations of each edge.
- Space proportional to n^2.
- Checking if (u, v) is an edge takes $\Theta(1)$ time.
- Identifying all edges takes $\Theta(n^2)$ time.



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

Graph Representation: Adjacency Matrix

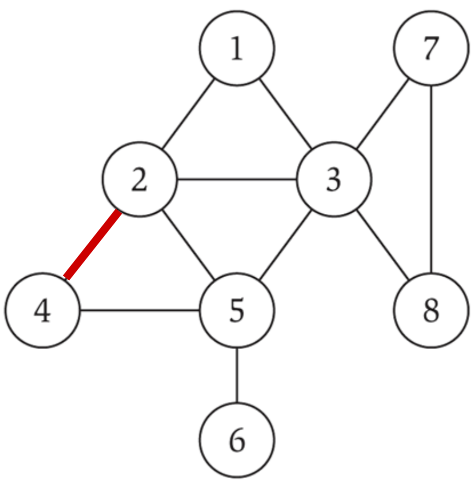
Line Graph: n vertices and n-1 edges.

- Adjacency matrix is full of 0's

Facebook

- 750M vertices
- Assumption: each person has 130 friends in average

➔ 550 Petabytes to store approximately 50 Billion edges;



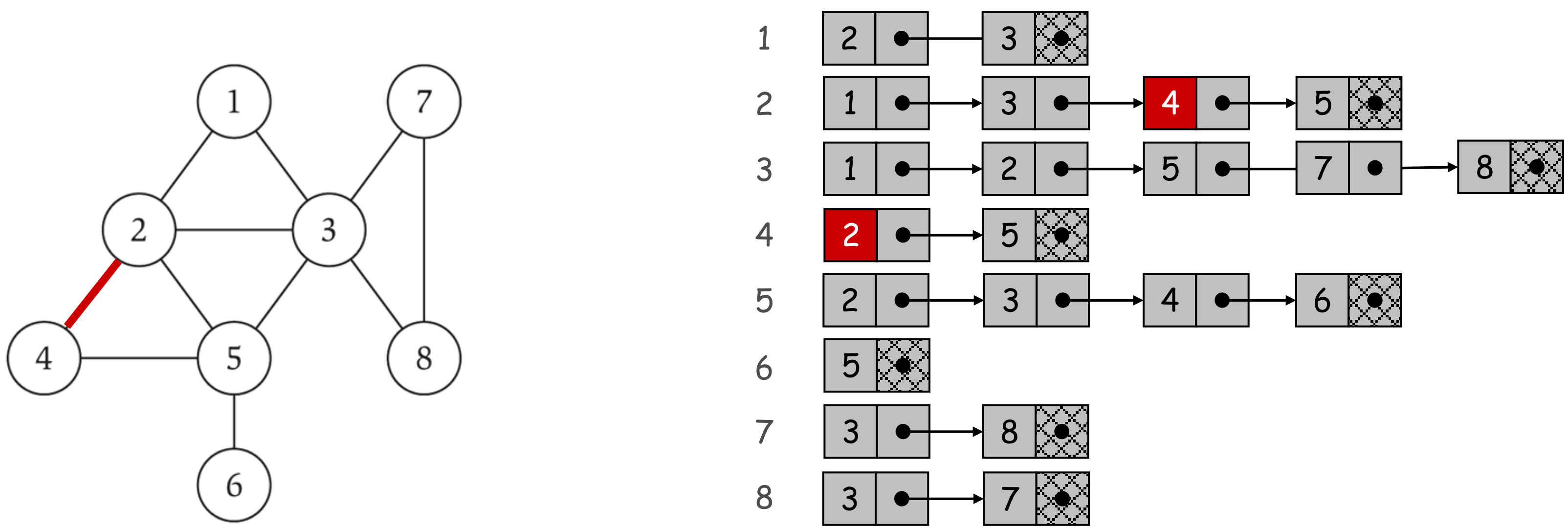
	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	0	0
2	1	0	1	1	1	0	0	0
3	1	1	0	0	1	0	1	1
4	0	1	0	0	1	0	0	0
5	0	1	1	1	0	1	0	0
6	0	0	0	0	1	0	0	0
7	0	0	1	0	0	0	0	1
8	0	0	1	0	0	0	1	0

Graph Representation: Adjacency List

Adjacency list. Node indexed array of lists.

- Two representations of each edge.
- Space proportional to m + n.
- Checking if (u, v) is an edge takes O(deg(u)) time.
- Identifying all edges takes Θ(m + n) time.

degree = number of neighbors of u

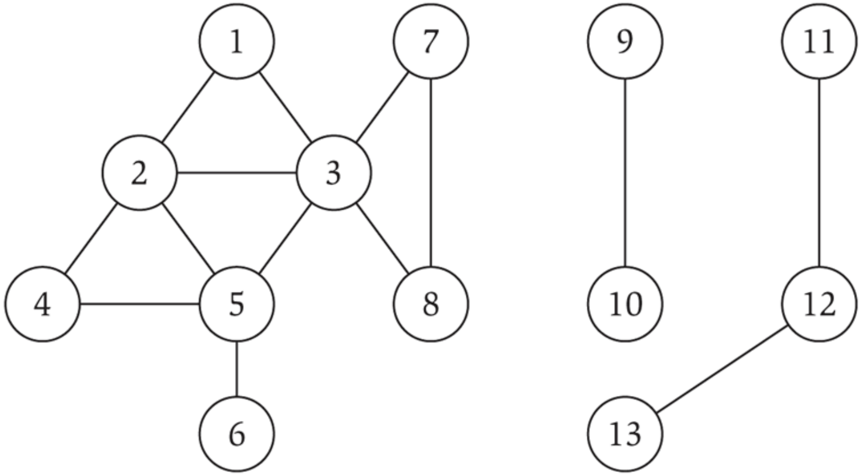


Paths and Connectivity

Def. A path in an undirected graph $G = (V, E)$ is a sequence P of nodes $v_1, v_2, \ldots, v_{k-1}, v_k$ with the property that each consecutive pair v_i, v_{i+1} is joined by an edge in E.

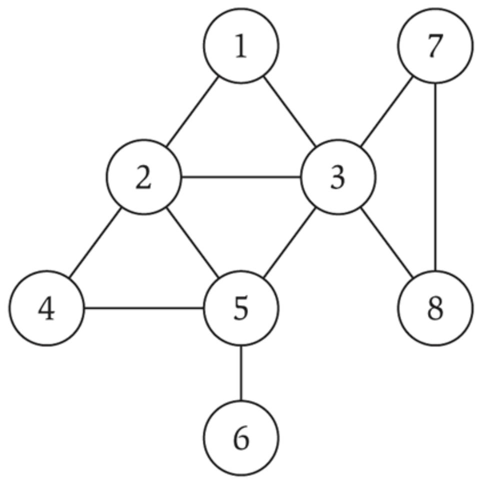
Def. A path is simple if all nodes are distinct.

Def. An undirected graph is connected if for every pair of nodes u and v, there is a path between u and v.



Cycles

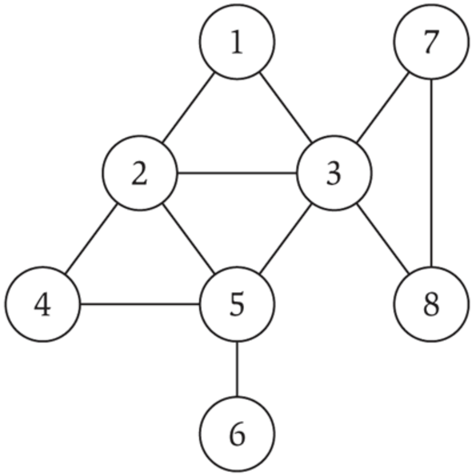
Def. A cycle is a path $v_1, v_2, \ldots, v_{k-1}, v_k$ in which $v_1 = v_k$, $k > 3$, and the first k-1 nodes are all distinct.



cycle C = 1-2-4-5-3-1

Distance

Def. The distance between vertexes s and t in a graph G is the number of edges of the shortest path connecting s to t in G.



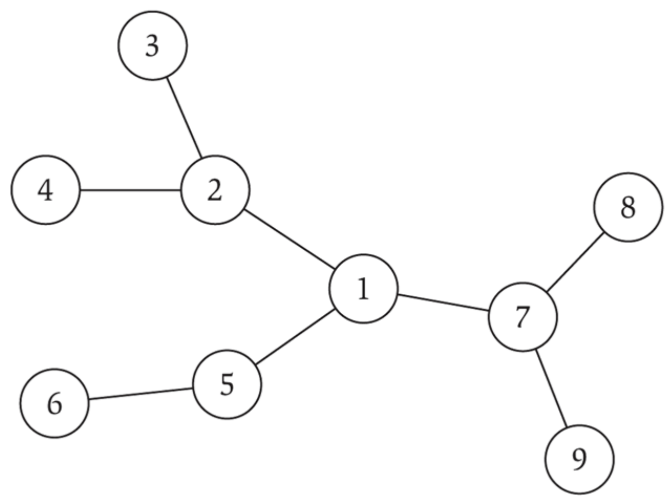
Distance(1,4) =2
Distance(6,3)= 2
Distance(7,8) =1

Trees

Def. An undirected graph is a tree if it is connected and does not contain a cycle.

Theorem. Let G be an undirected graph on n nodes. Any two of the following statements imply the third.

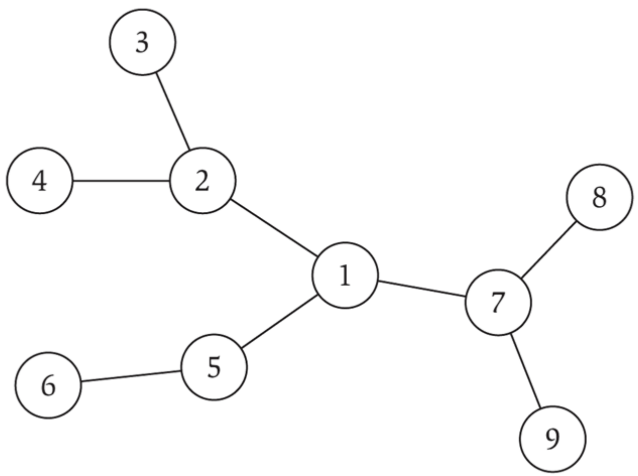
- G is connected.
- G does not contain a cycle.
- G has n-1 edges.



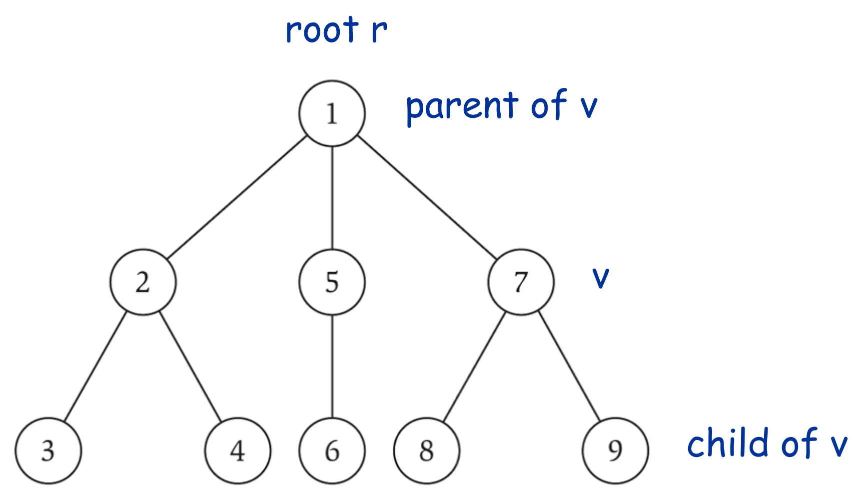
Rooted Trees

Rooted tree. Given a tree T, choose a root node r and orient each edge away from r.

Importance. Models hierarchical structure.



a tree



the same tree, rooted at 1

3.2 Graph Traversal

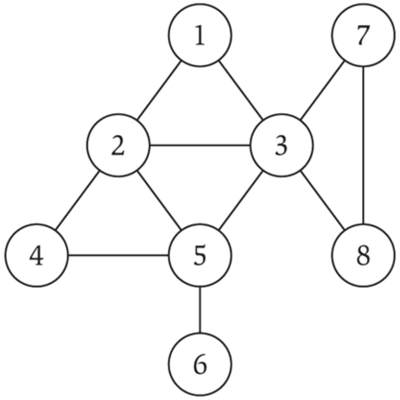
Connectivity

s-t connectivity problem. Given two node s and t, is there a path between s and t?

s-t shortest path problem. Given two node s and t, what is the length of the shortest path between s and t?

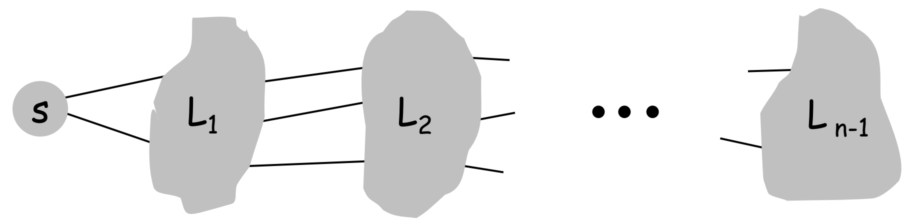
Applications.

- Maze traversal.
- Kevin Bacon number.
- Fewest number of hops in a communication network.



Breadth First Search

BFS intuition. Explore outward from s in all possible directions, adding nodes one "layer" at a time.



BFS algorithm.

- $L_0 = \{ s \}$.
- L_1 = all neighbors of L_0.
- L_2 = all nodes that do not belong to L_0 or L_1, and that have an edge to a node in L_1.
- L_{i+1} = all nodes that do not belong to an earlier layer, and that have an edge to a node in L_i.

Theorem. For each i, L_i consists of all nodes at distance exactly i from s. There is a path from s to t iff t appears in some layer.

Breadth First Search

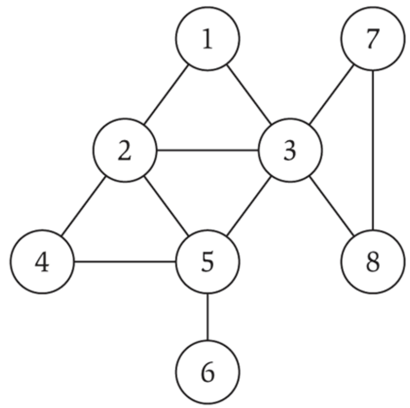
Definition A BFS tree of G = (V, E), is the tree induced by a BFS search on G.

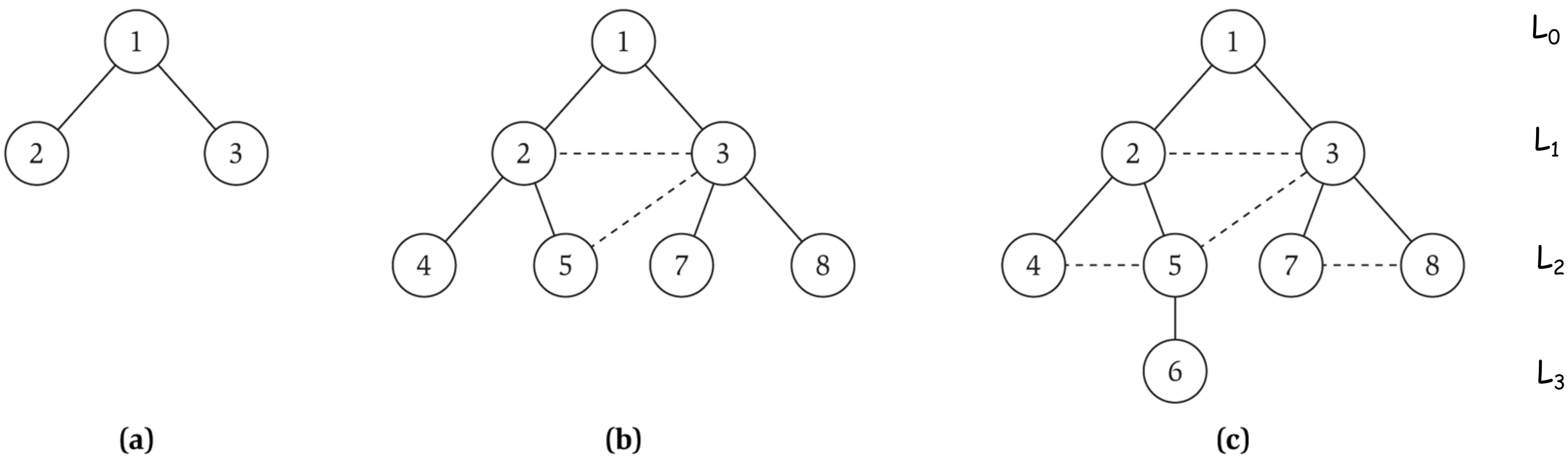
- The root of the tree is the start point of the BFS
- A node u is a parent of v if v is first visited when the BFS traverses the neighbors of u

Observation For the same graph there can be different BFS trees. The BFS tree topology depends on the start point of the BFS and the rule employed to break ties (how the data structure is traversed)

Breadth First Search

Property. Let T be a BFS tree of G = (V, E), and let (x, y) be an edge of G. Then the level of x and y differ by at most 1.





BFS(G)

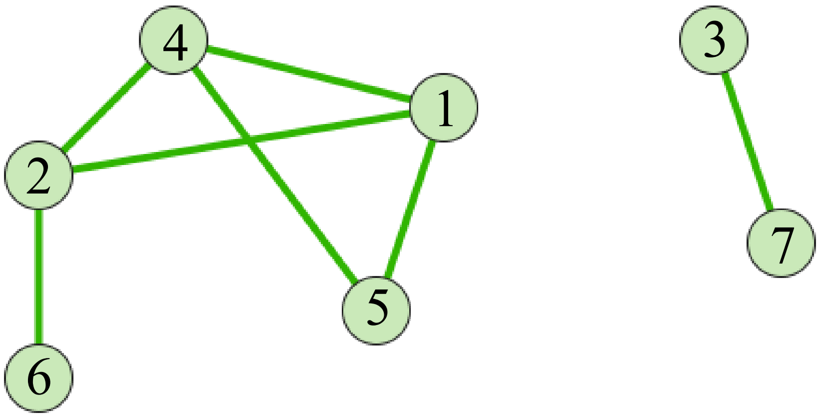
```
for every vertex s of G  not explored yet
        do Enqueue(S,s)
        mark vertex s as visited
        while S is not empty do
                u ← Dequeue(S);
                For each  v in Adj[u] then
                        if v is unexplored then
                                mark edge (v,u) as tree edge
                                mark vertex v as visited
                                Enqueue(S,v)
```

Notação

- Adj [u] : lista dos vértices adjacentes a u em alguma ordem
- Dequeue(S): Remove o primeiro elemento da fila S
- Enqueue (S,v) : Adiciona o nó v na fila S

Exemplo

S

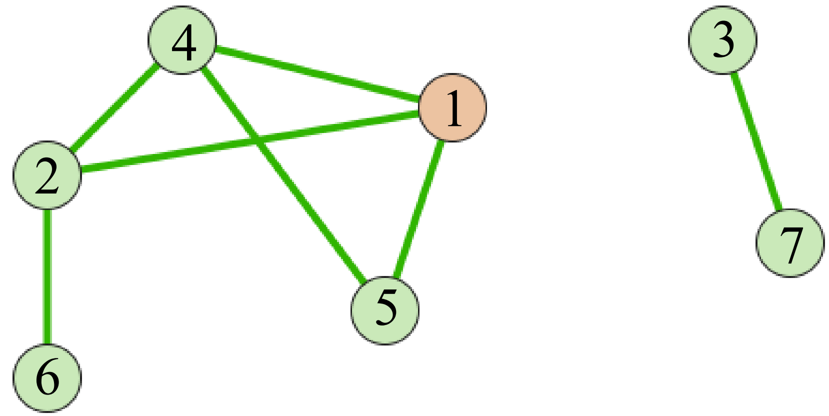


BFS(G)

```
for every vertex s of G  not explored yet
        do Enqueue(S,s);
        mark vertex s as visited
        while S is not empty do
                u ← Dequeue(S);
                For each  v in Adj[u] then
                        if v is unexplored then
                                mark edge (v,u) as tree edge
                                mark vertex v as visited
                                Enqueue(S,v)
```

Exemplo

S | 1

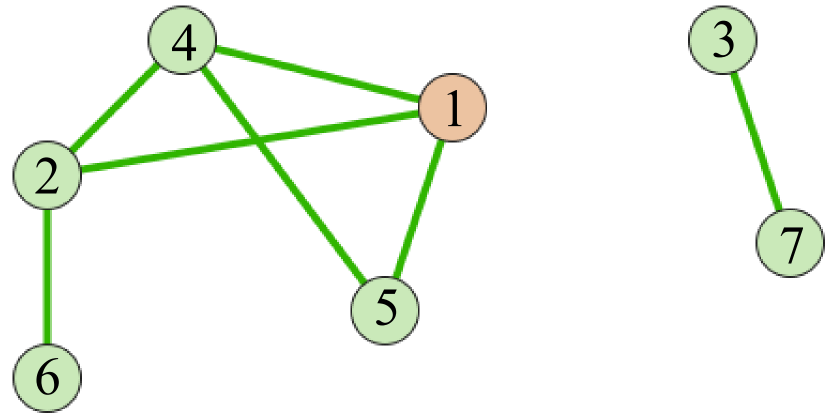


BFS(G)

```
for every vertex s of G  not explored yet
      do Enqueue(S,s);
      mark vertex s as visited
      while S is not empty do
            u ← Dequeue(S);
            For each  v in Adj[u] then
                  if v is unexplored then
                        mark edge (v,u) as tree edge
                        mark vertex v as visited
                        Enqueue(S,v)
```

Exemplo

S

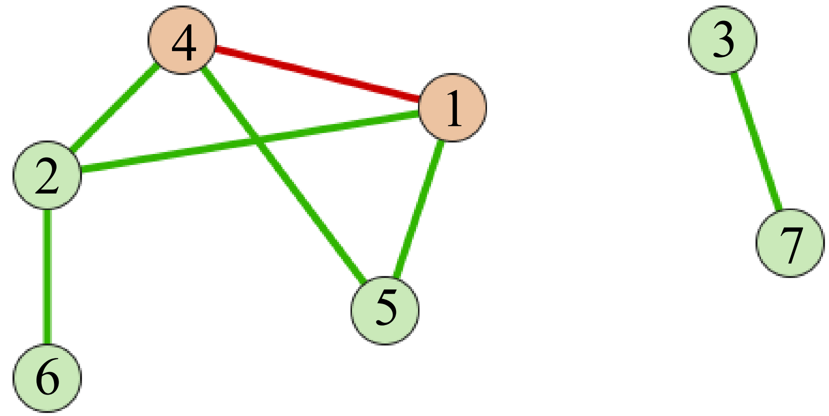


BFS(G)

```
for every vertex s of G  not explored yet
        do Enqueue(S,s);
        mark vertex s as visited
        while S is not empty do
                u ← Dequeue(S);
                For each  v in Adj[u] then
                        if v is unexplored then
                                mark edge (v,u) as tree edge
                                mark vertex v as visited
                                Enqueue(S,v)
```

Exemplo

S 4

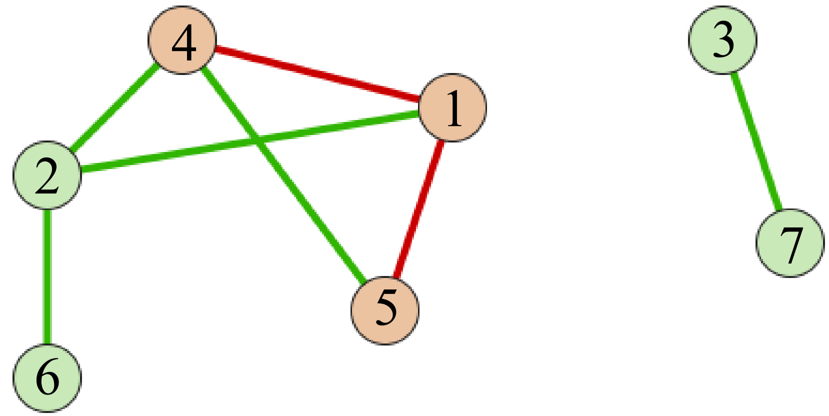


BFS(G)

```
for every vertex s of G  not explored yet
      do Enqueue(S,s);
      mark vertex s as visited
      while S is not empty do
            u ← Dequeue(S);
            For each  v in Adj[u] then
                  if v is unexplored then
                        mark edge (v,u) as tree edge
                        mark vertex v as visited
                        Enqueue(S,v)
```

Exemplo

S 4 5

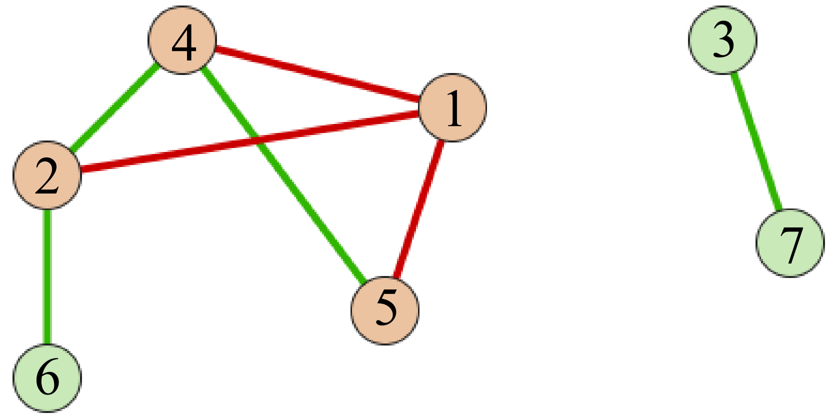


BFS(G)

```
for every vertex s of G  not explored yet
        do Enqueue(S,s);
        mark vertex s as visited
        while S is not empty do
                u ← Dequeue(S);
                For each  v in Adj[u] then
                        if v is unexplored then
                                mark edge (v,u) as tree edge
                                mark vertex v as visited
                                Enqueue(S,v)
```

Exemplo

S 4 5 2

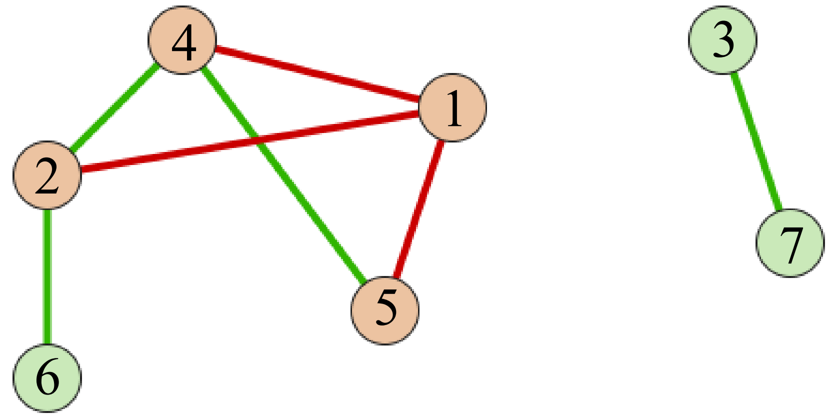


BFS(G)

```
for every vertex s of G  not explored yet
        do Enqueue(S,s);
        mark vertex s as visited
        while S is not empty do
                u ← Dequeue(S);
                For each  v in Adj[u] then
                        if v is unexplored then
                                mark edge (v,u) as tree edge
                                mark vertex v as visited
                                Enqueue(S,v)
```

Exemplo

S 5 2



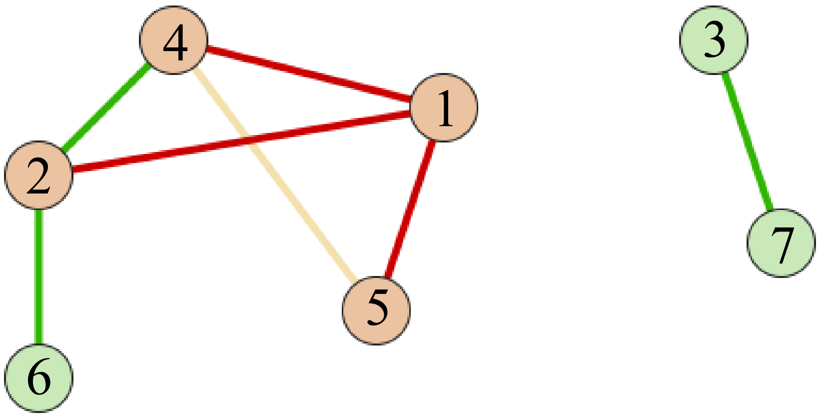
BFS(G)

```
for every vertex s of G  not explored yet
    do Enqueue(S,s);
    mark vertex s as visited
    while S is not empty do
        u ← Dequeue(S);
        For each  v in Adj[u] then
            if v is unexplored then
                mark edge (v,u) as tree edge
                mark vertex v as visited
                Enqueue(S,v)
```

Exemplo

S

5 2

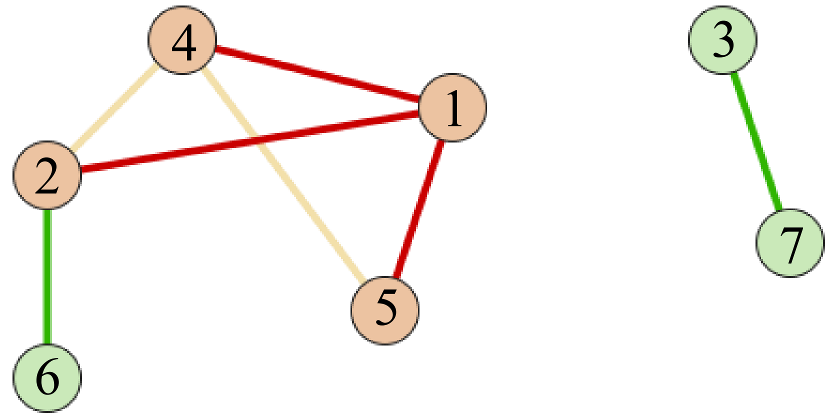


BFS(G)

```
for every vertex s of G  not explored yet
      do Enqueue(S,s);
      mark vertex s as visited
      while S is not empty do
            u ← Dequeue(S);
            For each  v in Adj[u] then
                  if v is unexplored then
                        mark edge (v,u) as tree edge
                        mark vertex v as visited
                        Enqueue(S,v)
```

Exemplo

S 5 2

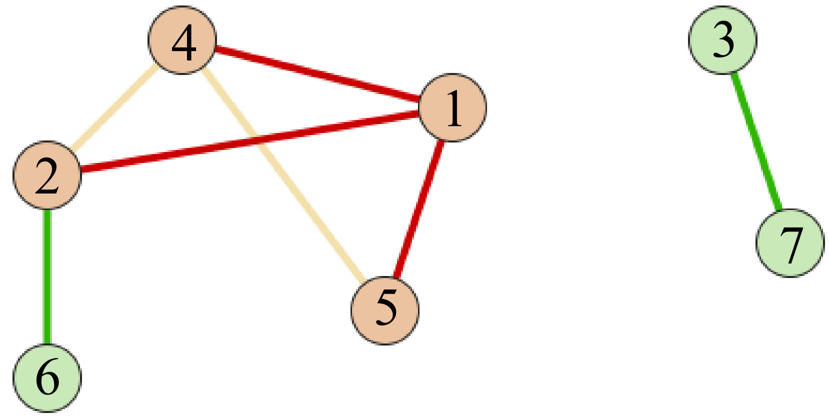


BFS(G)

```
for every vertex s of G  not explored yet
      do Enqueue(S,s);
      mark vertex s as visited
      while S is not empty do
            u ← Dequeue(S);
            For each  v in Adj[u] then
                  if v is unexplored then
                        mark edge (v,u) as tree edge
                        mark vertex v as visited
                        Enqueue(S,v)
```

Exemplo

S 2

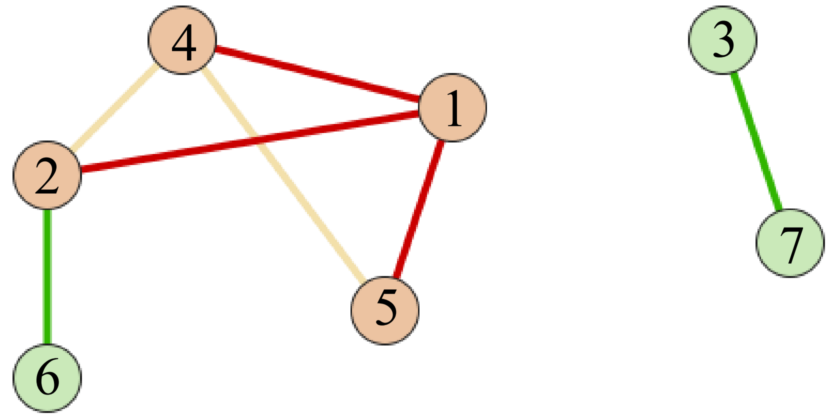


BFS(G)

```
for every vertex s of G  not explored yet
    do Enqueue(S,s);
    mark vertex s as visited
    while S is not empty do
        u ← Dequeue(S);
        For each  v in Adj[u] then
            if v is unexplored then
                mark edge (v,u) as tree edge
                mark vertex v as visited
                Enqueue(S,v)
```

Exemplo

S

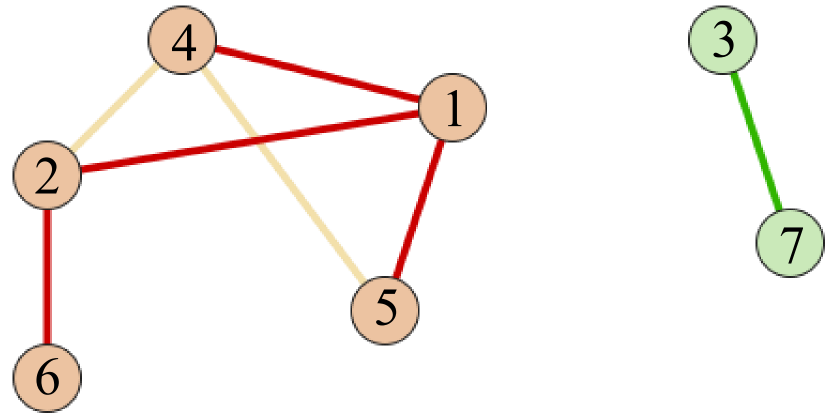


BFS(G)

```
for every vertex s of G  not explored yet
    do Enqueue(S,s);
    mark vertex s as visited
    while S is not empty do
        u ← Dequeue(S);
        For each  v in Adj[u] then
            if v is unexplored then
                mark edge (v,u) as tree edge
                mark vertex v as visited
                Enqueue(S,v)
```

Exemplo

S | 6

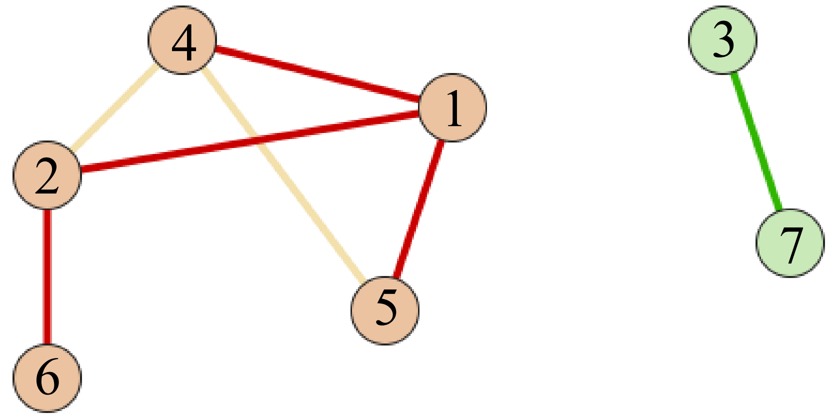


BFS(G)

```
for every vertex s of G  not explored yet
      do Enqueue(S,s);
      mark vertex s as visited
      while S is not empty do
            u ← Dequeue(S);
            For each  v in Adj[u] then
                  if v is unexplored then
                        mark edge (v,u) as tree edge
                        mark vertex v as visited
                        Enqueue(S,v)
```

Exemplo

S

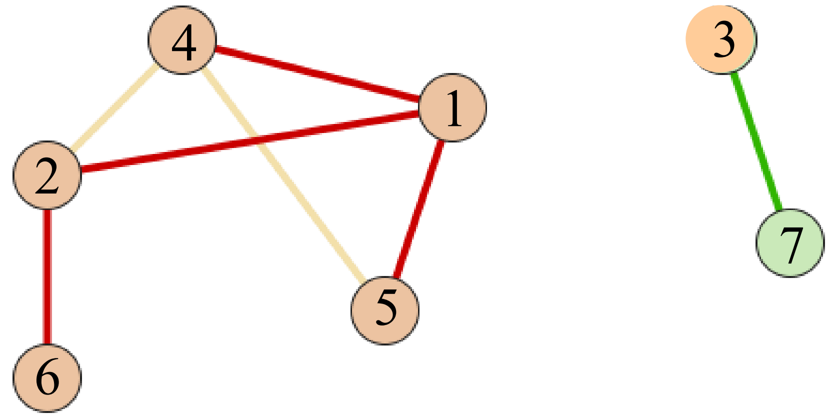


BFS(G)

```
for every vertex s of G  not explored yet
      do Enqueue(S,s);
      mark vertex s as visited
      while S is not empty do
            u ← Dequeue(S);
            For each  v in Adj[u] then
                  if v is unexplored then
                        mark edge (v,u) as tree edge
                        mark vertex v as visited
                        Enqueue(S,v)
```

Exemplo

S 3

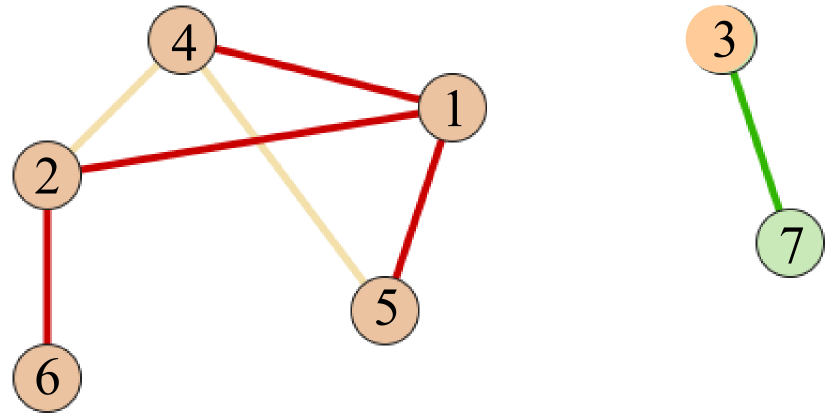


BFS(G)

```
for every vertex s of G  not explored yet
      do Enqueue(S,s);
      mark vertex s as visited
      while S is not empty do
            u ← Dequeue(S);
            For each  v in Adj[u] then
                  if v is unexplored then
                        mark edge (v,u) as tree edge
                        mark vertex v as visited
                        Enqueue(S,v)
```

Exemplo

S

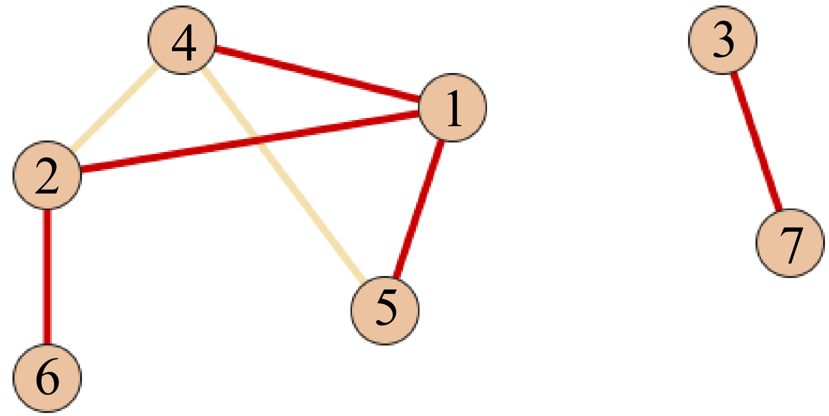


BFS(G)

```
for every vertex s of G  not explored yet
      do Enqueue(S,s);
      mark vertex s as visited
      while S is not empty do
            u ← Dequeue(S);
            For each  v in Adj[u] then
                  if v is unexplored then
                        mark edge (v,u) as tree edge
                        mark vertex v as visited
                        Enqueue(S,v)
```

Exemplo

S 7

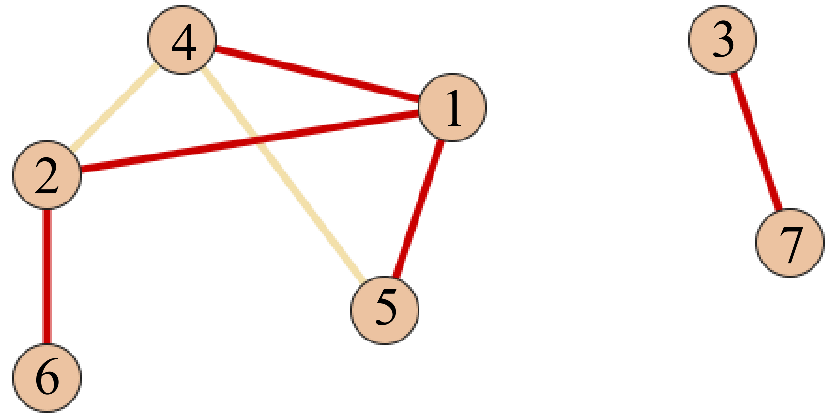


BFS(G)

```
for every vertex s of G  not explored yet
    do Enqueue(S,s);
    mark vertex s as visited
    while S is not empty do
        u ← Dequeue(S);
        For each  v in Adj[u] then
            if v is unexplored then
                mark edge (v,u) as tree edge
                mark vertex v as visited
                Enqueue(S,v)
```

Exemplo

S



BFS(G)

```
for every vertex s of G  not explored yet
        do Enqueue(S,s);
        mark vertex s as visited
        while S is not empty do
                u ← Dequeue(S);
                For each  v in Adj[u] then
                        if v is unexplored then
                                mark edge (v,u) as tree edge
                                mark vertex v as visited
                                Enqueue(S,v)
```

Breadth First Search: Analysis

Análise $O(n^2)$:

- (i) Cada vértice entra na fila S no máximo uma vez ➔ Loop **While** é executado no máximo n vezes.
- (ii) Cada vértice tem no máximo n-1 vizinhos ➔ **For** é executado no máximo n vezes
- (i) e (ii) implicam em $O(n^2)$

Análise $O(m + n)$

each edge (u, v) is counted exactly twice in sum: once in deg(u) and once in deg(v)

- (i) custo dentre do **While** $O(n)$
- (ii) Custo dentro do **For** para vértice u é degree(u). Somando para todos os vertices temos $\Sigma_{u \in V}$ degree(u) = 2m
- (i) e (ii) implicam em $O(n+m)$

Breadth First Search: Analysis

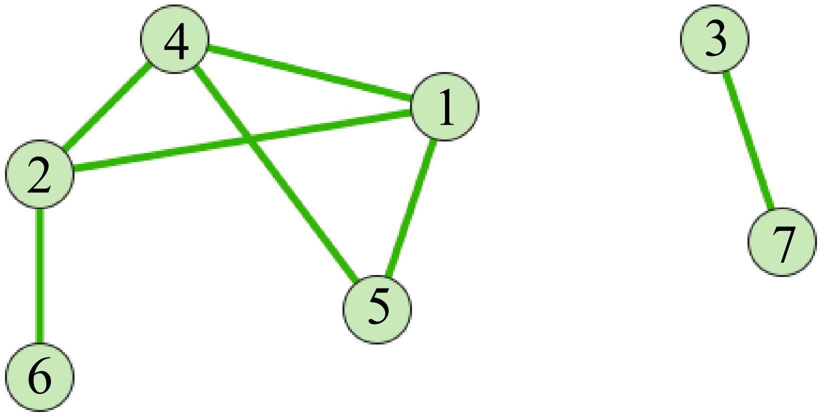
Theorem. The above implementation of BFS runs in $O(m + n)$ time if the graph is given by its adjacency representation.

Pf.

- when we consider node u, there are deg(u) incident edges (u, v)
- total time processing edges is $\Sigma_{u \in V} \deg(u) = 2m$ ▪

each edge (u, v) is counted exactly twice in sum: once in deg(u) and once in deg(v)

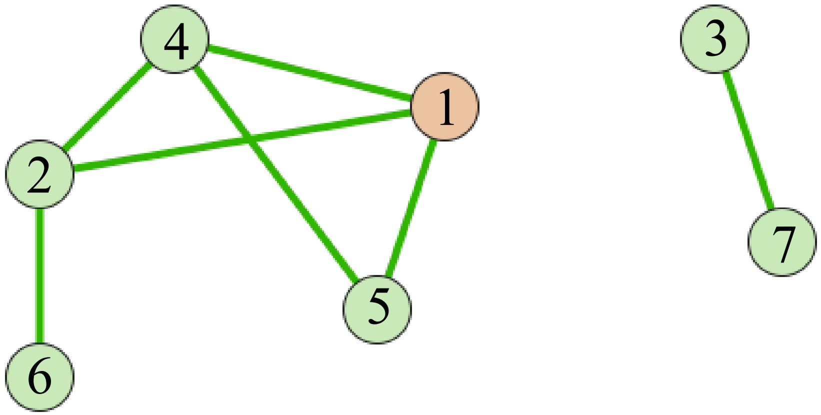
Busca em Profundidade




```
DFS(G)
    Para todo v em G
        Se v não visitado então
            DFS-Visit(G, v)

DFS-Visit(G, v)
    Marque v como visitado
    Para todo w em Adj(v)
        Se w não visitado então
            Insira aresta (v, w) na árvore
            DFS-Visit(G, w)
```

Busca em Profundidade : Exemplo



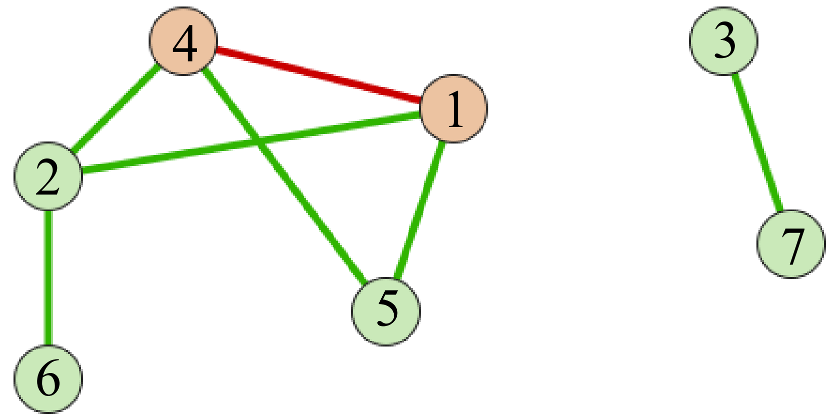
DFS(*G*)

```
Para todo v em G
    Se v não visitado então
        DFS-Visit(G, v)
```

DFS-Visit(*G*, *v*)

```
Marque v como visitado
Para todo w em Adj(v)
    Se w não visitado então
        Insira aresta (v, w) na árvore
        DFS-Visit(G, w)
```

Busca em Profundidade : Exemplo



DFS(*G*)

Para todo *v* em *G*

  Se *v* não visitado **então**

    DFS-Visit(*G*, *v*)

DFS-Visit(*G*, *v*)

Marque *v* como visitado

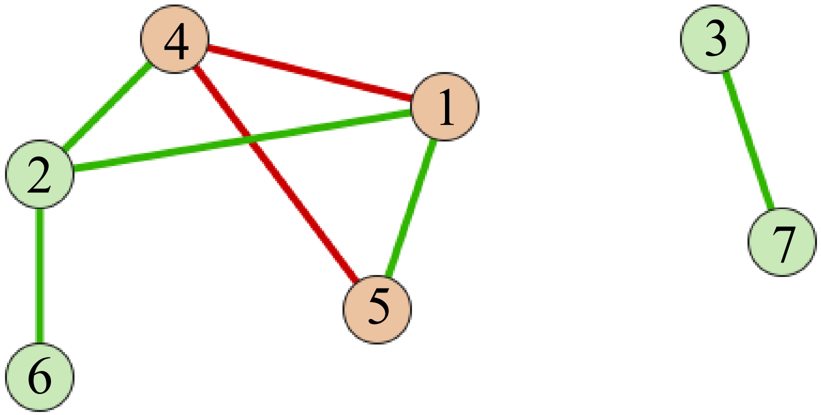
Para todo *w* em Adj(*v*)

  Se *w* não visitado **então**

    Insira aresta (*v*, *w*) na árvore

    DFS-Visit(*G*, *w*)

Busca em Profundidade : Exemplo



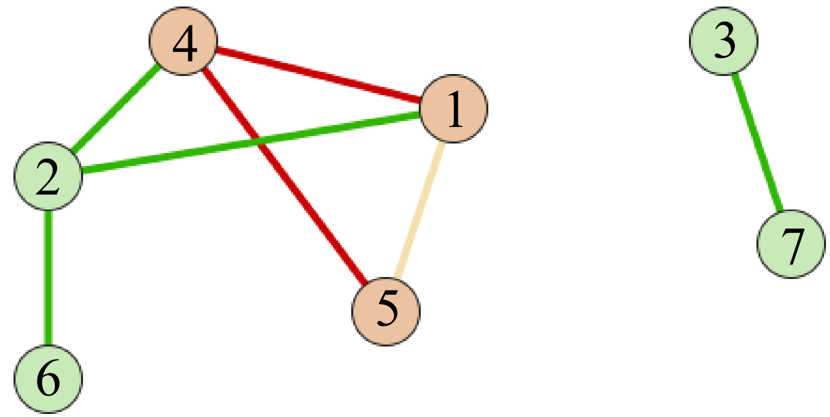
DFS(*G*)

Para todo *v* em *G*
    Se *v* não visitado **então**
        DFS-Visit(*G*, *v*)

DFS-Visit(*G*, *v*)

Marque *v* como visitado
Para todo *w* em Adj(*v*)
    Se *w* não visitado **então**
        Insira aresta (*v*, *w*) na árvore
        DFS-Visit(*G*, *w*)

Busca em Profundidade : Exemplo



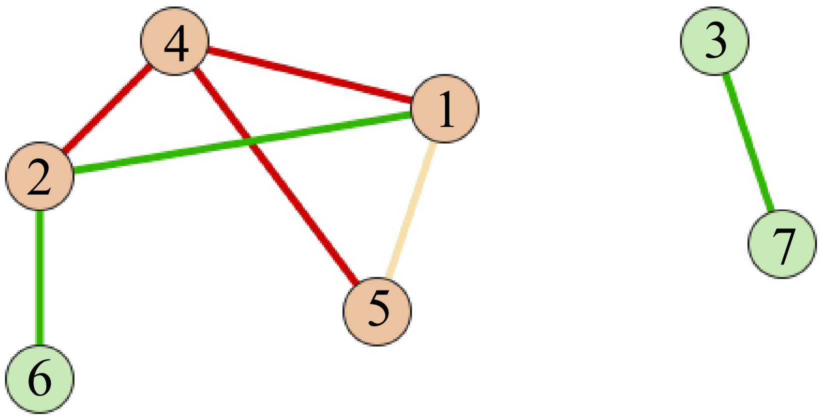
DFS(*G*)

```
    Para todo v em G
        Se v não visitado então
            DFS-Visit(G, v)
```

DFS-Visit(*G*, *v*)

```
    Marque v como visitado
    Para todo w em Adj(v)
        Se w não visitado então
            Insira aresta (v, w) na árvore
            DFS-Visit(G, w)
```

Busca em Profundidade : Exemplo



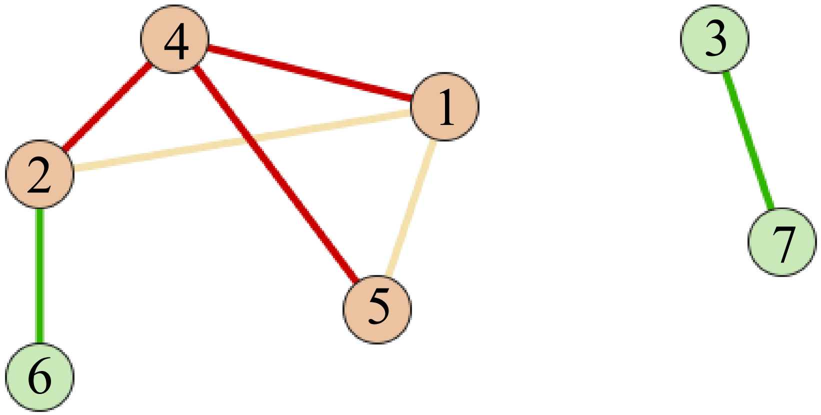
DFS(*G*)

```
Para todo v em G
    Se v não visitado então
        DFS-Visit(G, v)
```

DFS-Visit(*G*, *v*)

```
Marque v como visitado
Para todo w em Adj(v)
    Se w não visitado então
        Insira aresta (v, w) na árvore
        DFS-Visit(G, w)
```

Busca em Profundidade : Exemplo



DFS(*G*)

    Para todo *v* em *G*

        Se *v* não visitado **então**

            DFS-Visit(*G*, *v*)

DFS-Visit(*G*, *v*)

    Marque *v* como visitado

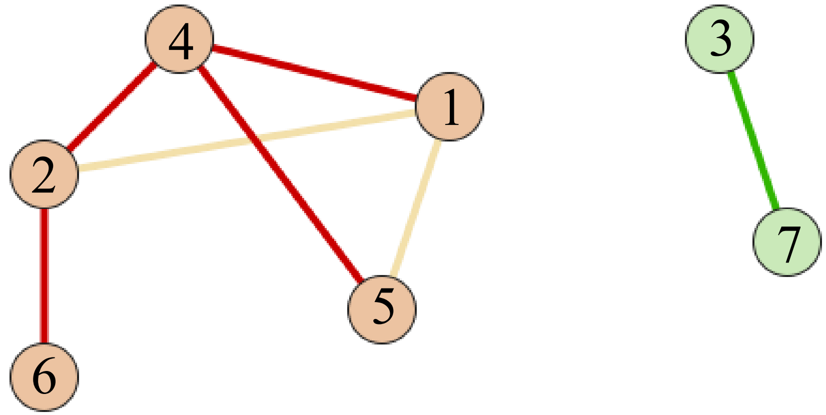
    Para todo *w* em Adj(*v*)

        Se *w* não visitado **então**

            Insira aresta (*v*, *w*) na árvore

            DFS-Visit(*G*, *w*)

Busca em Profundidade : Exemplo



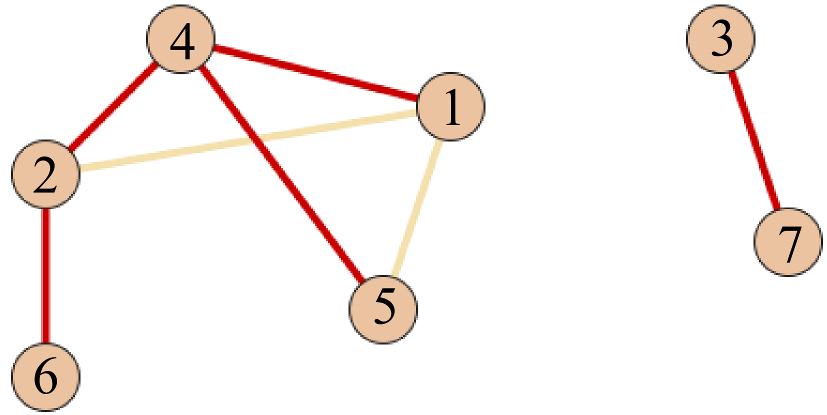
DFS(*G*)

Para todo *v* em *G*
  Se *v* não visitado **então**
    DFS-Visit(*G*, *v*)

DFS-Visit(*G*, *v*)

Marque *v* como visitado
Para todo *w* em Adj(*v*)
  Se *w* não visitado **então**
    Insira aresta (*v*, *w*) na árvore
    DFS-Visit(*G*, *w*)

Busca em Profundidade : Exemplo



DFS(*G*)

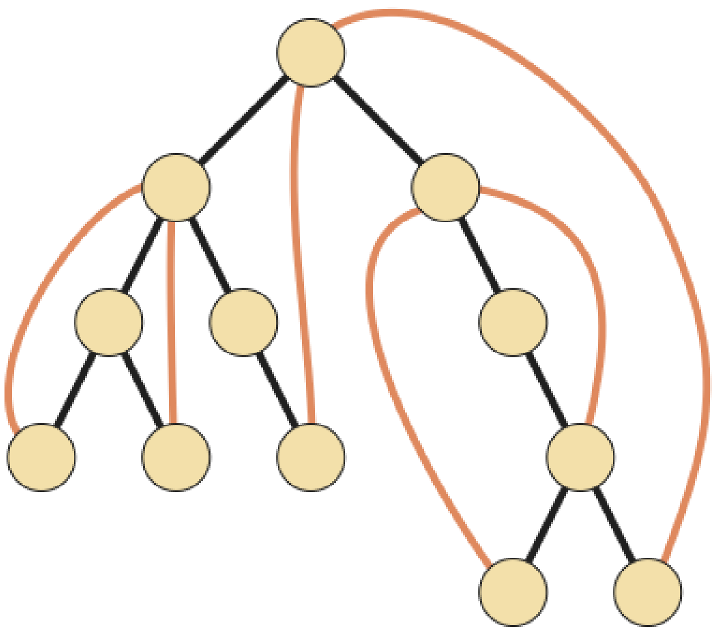
```
Para todo v em G
    Se v não visitado então
        DFS-Visit(G, v)
```

DFS-Visit(*G*, *v*)

```
Marque v como visitado
Para todo w em Adj(v)
    Se w não visitado então
        Insira aresta (v, w) na árvore
        DFS-Visit(G, w)
```

Propriedades da Busca de Profundidade

Teorema: *Seja T a árvore produzida por uma DFS em G e seja vw uma aresta de G. Se v é visitado antes de w então v é ancestral de w em T.*



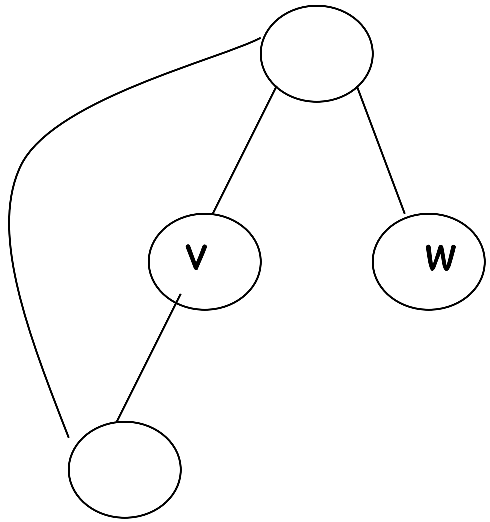
Arestas em preto: arvore DFS
Arestas em preto e laranja: arestas do grafo

Propriedades da Busca de Profundidade

Nota: Se *T é uma árvore produzida por uma DFS em G, existe um caminho entre v e w* em *G e v é visitado antes de w então* ***não*** *necessariamente v é ancestral de w em T.*

Propriedades da Busca de Profundidade

Nota: Se *T é uma árvore produzida por uma DFS em G, existe um caminho entre v e w* em *G e v é visitado antes de w então* **não** *necessariamente v é ancestral de w em T.*



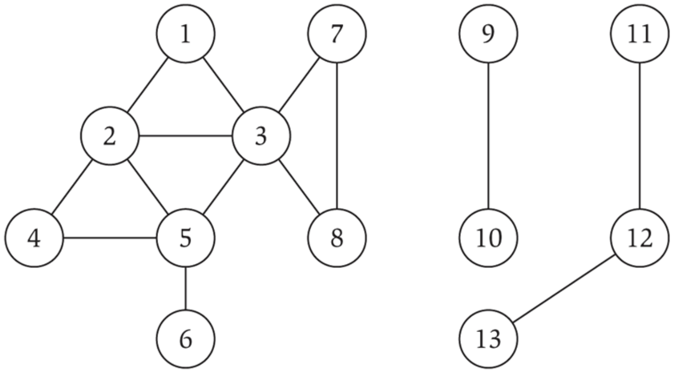
Connected Component

Def Connected set. S is a connected set if and only if

- v is reachable from u and u is reachable from v for every u,v in S

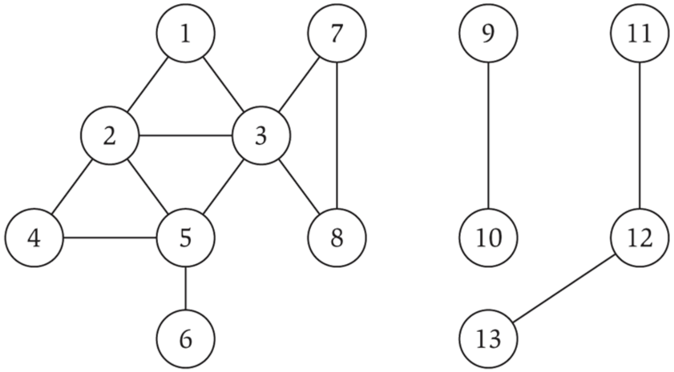
Def Connected Component. S is a connected component if and only if

- v is a connected set
- for every u in V-S, S ∪ {u} is not connected



Connected Component

Connected components.



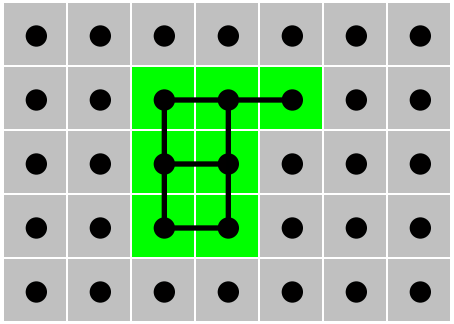
Connected component containing node 1 = { 1, 2, 3, 4, 5, 6, 7, 8 }.

Flood Fill

Flood fill. Given lime green pixel in an image, change color of entire blob of neighboring lime pixels to blue.

- Node: pixel.
- Edge: two neighboring lime pixels.
- Blob: connected component of lime pixels.

recolor lime green blob to blue

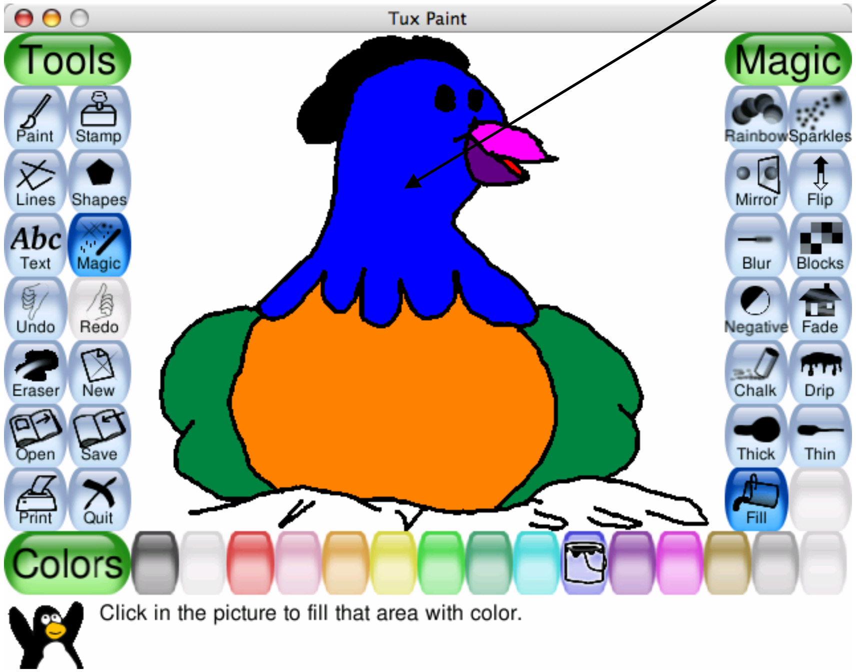


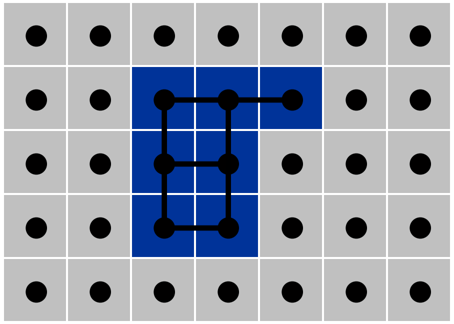
Flood Fill

Flood fill. Given lime green pixel in an image, change color of entire blob of neighboring lime pixels to blue.

- Node: pixel.
- Edge: two neighboring lime pixels.
- Blob: connected component of lime pixels.

recolor lime green blob to blue



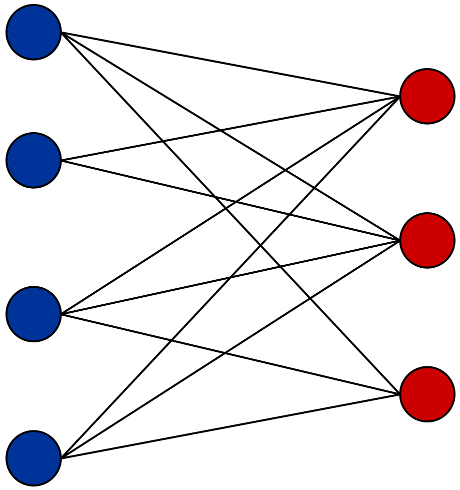


3.4 Testing Bipartiteness

Bipartite Graphs

Def. An undirected graph $G = (V, E)$ is bipartite if the nodes can be colored red or blue such that every edge has one red and one blue end.

Applications.
- Stable marriage: men = red, women = blue.
- Scheduling: machines = red, jobs = blue.

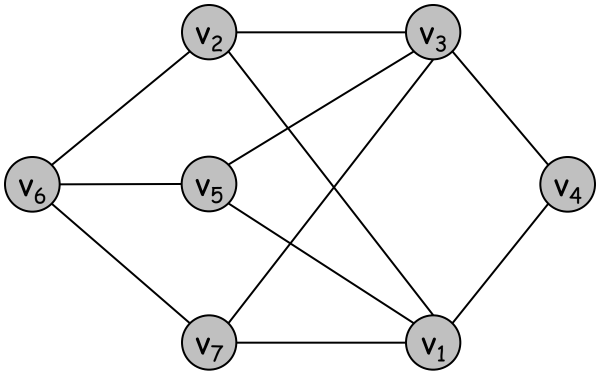


a bipartite graph

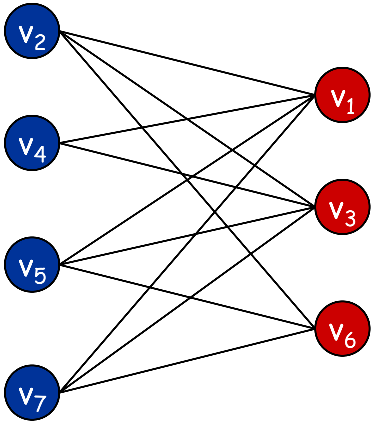
Testing Bipartiteness

Testing bipartiteness. Given a graph G, is it bipartite?

- Many graph problems become:
  - easier if the underlying graph is bipartite (matching)
  - tractable if the underlying graph is bipartite (independent set)
- Before attempting to design an algorithm, we need to understand structure of bipartite graphs.



a bipartite graph G

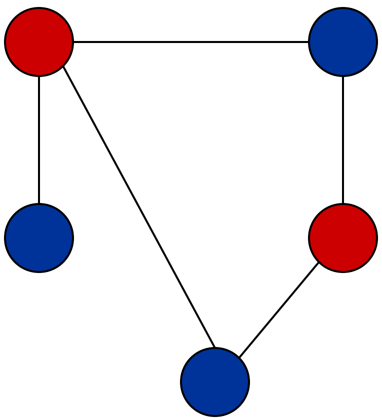


another drawing of G

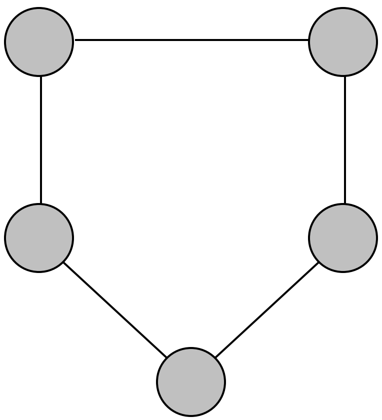
An Obstruction to Bipartiteness

Lemma. If a graph G is bipartite, it cannot contain an odd length cycle.

Pf. Not possible to 2-color the odd cycle, let alone G.



bipartite
(2-colorable)



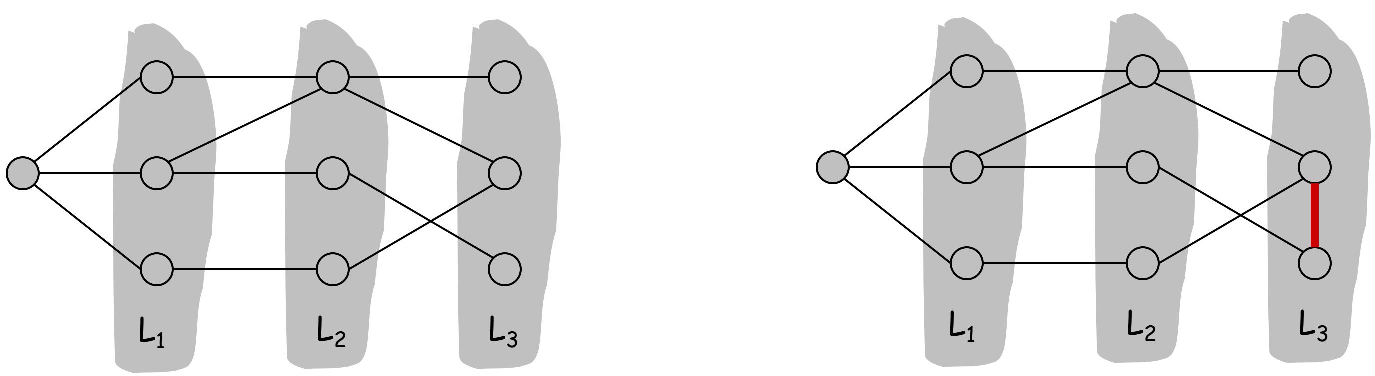
not bipartite
(not 2-colorable)

Bipartite Graphs

Lemma. Let G be a connected graph, and let $L_0, \ldots, L_k$ be the layers produced by BFS starting at node s. Exactly one of the following holds.

(i) No edge of G joins two nodes of the same layer, and G is bipartite.

(ii) An edge of G joins two nodes of the same layer, and G contains an odd-length cycle (and hence is not bipartite).



Case (i)   Case (ii)

Bipartite Graphs

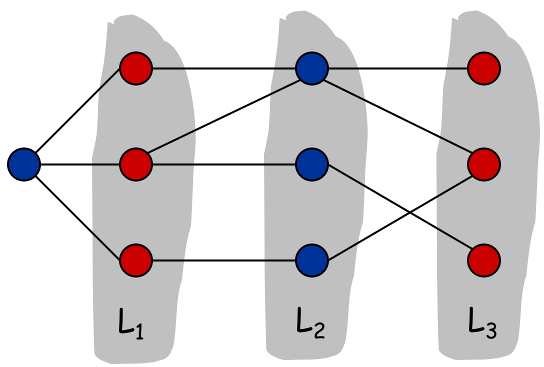
Lemma. Let G be a connected graph, and let $L_0, \ldots, L_k$ be the layers produced by BFS starting at node s. Exactly one of the following holds.

(i) No edge of G joins two nodes of the same layer, and G is bipartite.

(ii) An edge of G joins two nodes of the same layer, and G contains an odd-length cycle (and hence is not bipartite).

Pf. (i)

- Suppose no edge joins two nodes in the same layer.
- By previous lemma, this implies all edges join nodes on consecutive levels.
- Bipartition: red = nodes on odd levels, blue = nodes on even levels.



Case (i)

Bipartite Graphs

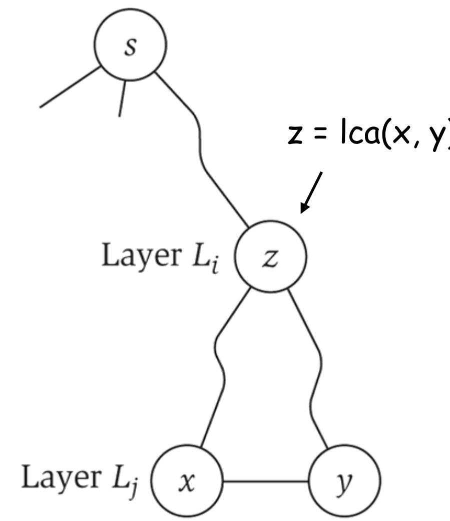
Lemma. Let G be a connected graph, and let $L_0, \ldots, L_k$ be the layers produced by BFS starting at node s. Exactly one of the following holds.

(i) No edge of G joins two nodes of the same layer, and G is bipartite.

(ii) An edge of G joins two nodes of the same layer, and G contains an odd-length cycle (and hence is not bipartite).

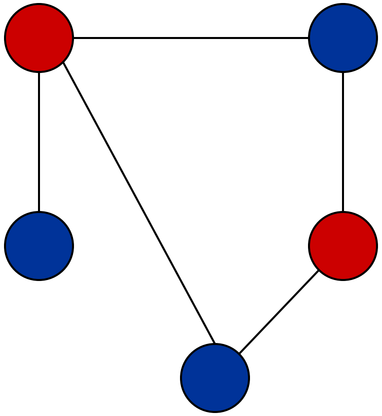
Pf. (ii)

- Suppose (x, y) is an edge with x, y in same level L_j.
- Let $z = \text{lca}(x, y)$ = lowest common ancestor.
- Let L_i be level containing z.
- Consider cycle that takes edge from x to y, then path from y to z, then path from z to x.
- Its length is $\underbrace{1}_{(x,\, y)} + \underbrace{(j-i)}_{\text{path from } y \text{ to } z} + \underbrace{(j-i)}_{\text{path from } z \text{ to } x}$, which is odd. ▪

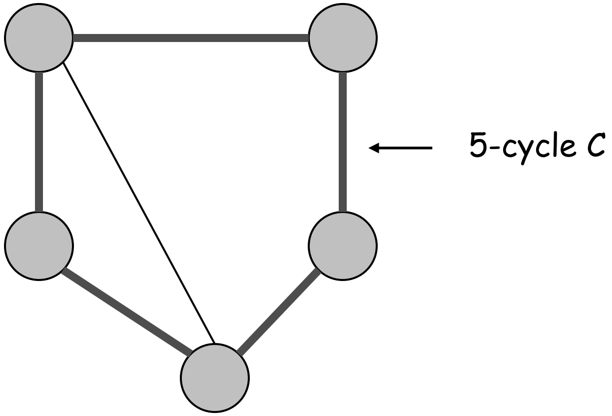


Obstruction to Bipartiteness

Corollary. A graph G is bipartite iff it contains no odd length cycle.



bipartite
(2-colorable)



not bipartite
(not 2-colorable)

Exercício de Impleementação

1. Modifique o código da BFS para que esta preencha um vetor **d** com n entradas aonde **d**(u) é a distância da origem **s** até o vértice u.

2. Modifique o algoritmo de busca em profundidade para que ele atribua números inteiros aos vértices do grafo de modo que

(i) Vértices de uma mesma componente recebam o mesmo número
(ii) Vértices de componentes diferentes recebam números diferentes

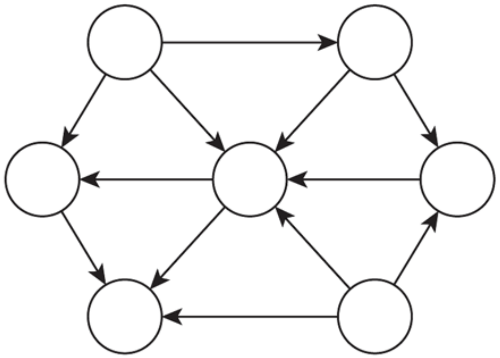
3. Modifique o código da BFS para que ela identifique se um grafo é bipartido ou não.

3.5 Connectivity in Directed Graphs

Directed Graphs

Directed graph. $G = (V, E)$

- Edge (u, v) goes from node u to node v.



Ex. Web graph - hyperlink points from one web page to another.

- Directedness of graph is crucial.
- Modern web search engines exploit hyperlink structure to rank web pages by importance.

Graph Search

Directed reachability. Given a node s, find all nodes reachable from s.

Directed s-t shortest path problem. Given two node s and t, what is the length of the shortest path between s and t?

Graph search. BFS extends naturally to directed graphs.

Web crawler. Start from web page s. Find all web pages linked from s, either directly or indirectly.

Strong Connectivity

Def. Node u and v are mutually reachable if there is a path from u to v and also a path from v to u.

Def. A graph is strongly connected if every pair of nodes is mutually reachable.

How to decide wether a given graph is strongly connected?

Strong Connectivity

Algorithm 1

```
SC ← true
For all u,v  in V
        If DFS(u) does not reach v
                SC ← False
        End If
End
```

Analysis $O(n^2 (m+n))$

Strong Connectivity

Obs. When we execute a DFS (BFS) starting at u we find all nodes reachable from u

Algorithm 2

```
SC ← true
For all u in V
      If DFS(u) does not reach |V| nodes
            SC ← False
      End If
End
```

Analysis O(n (m+n))

Strong Connectivity

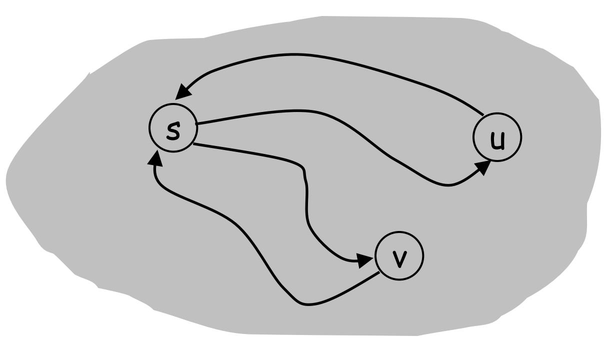
Def. Node u and v are mutually reachable if there is a path from u to v and also a path from v to u.

Def. A graph is strongly connected if every pair of nodes is mutually reachable.

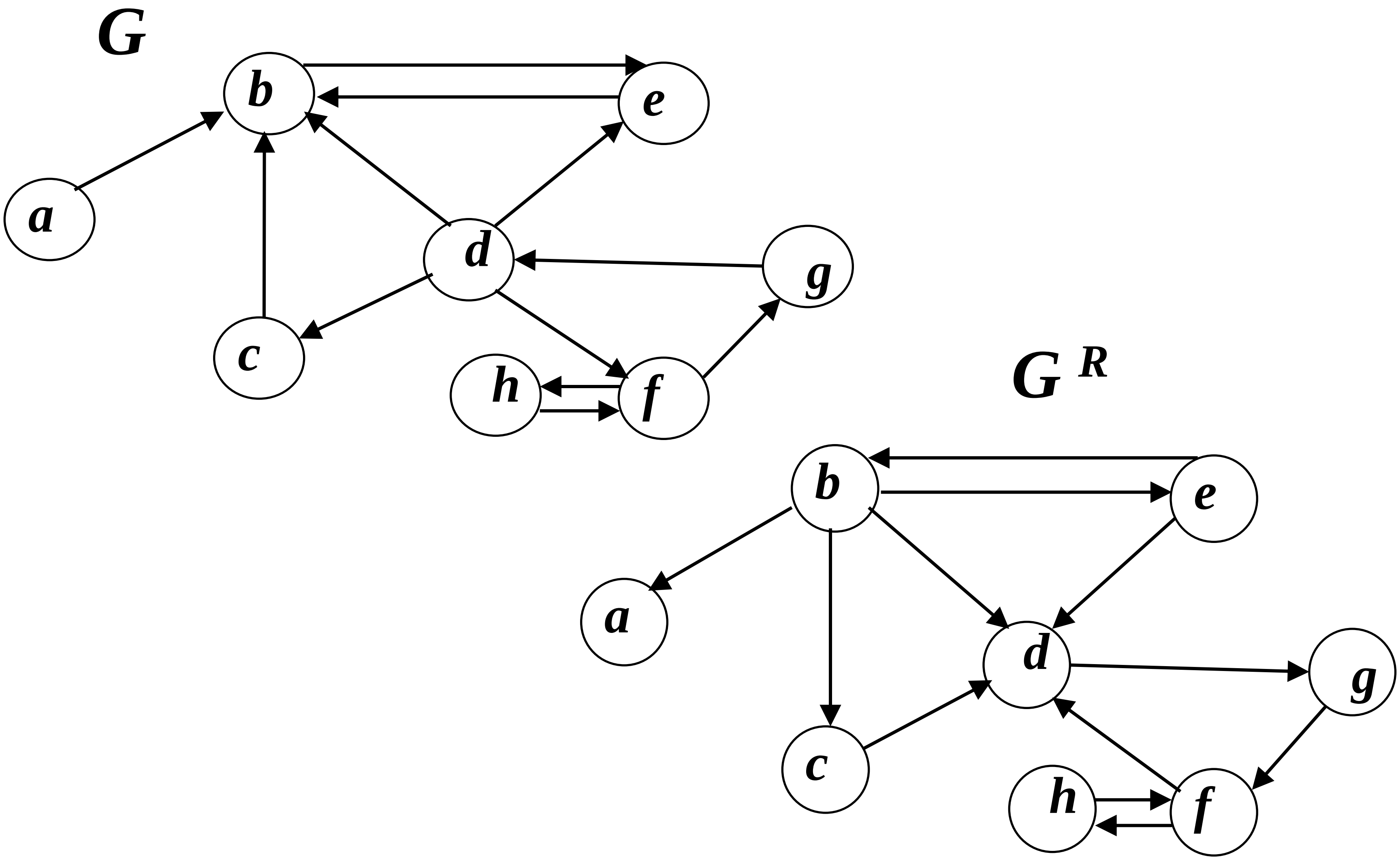
Lemma. Let s be any node. G is strongly connected iff every node is reachable from s, and s is reachable from every node.

Pf. $\Rightarrow$ Follows from definition.
Pf. $\Leftarrow$ Path from u to v: concatenate u-s path with s-v path.
Path from v to u: concatenate v-s path with s-u path. ▪

ok if paths overlap



Example: reverse graph G^R

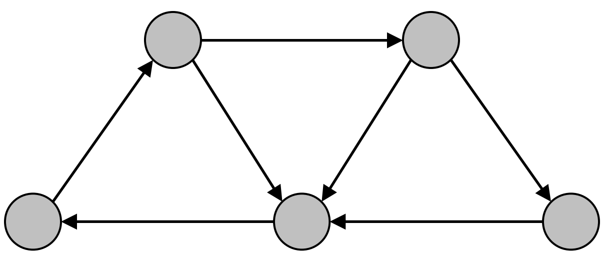


Strong Connectivity: Algorithm

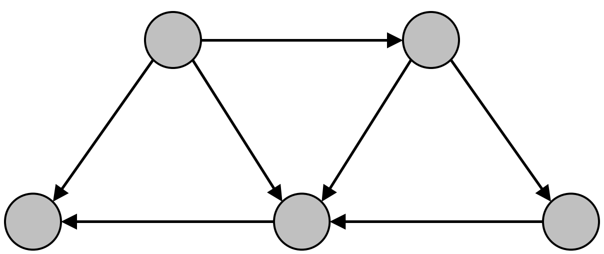
Theorem. Can determine if G is strongly connected in $O(m + n)$ time.
Pf.

- Pick any node s.
- Run BFS from s in G.
- Run BFS from s in G^{rev}.
- Return true iff all nodes reached in both BFS executions.
- Correctness follows immediately from previous lemma. ▪

reverse orientation of every edge in G



strongly connected



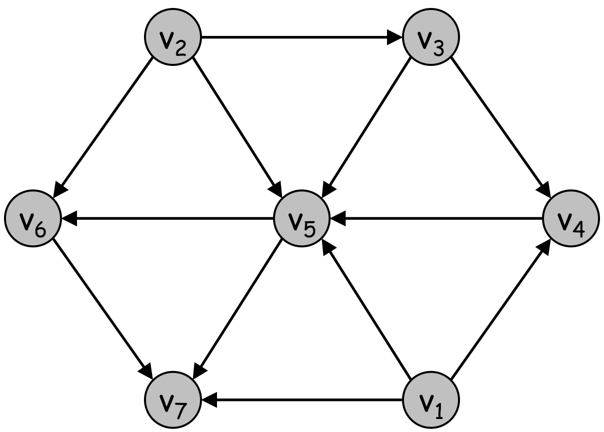
not strongly connected

3.6 DAGs and Topological Ordering

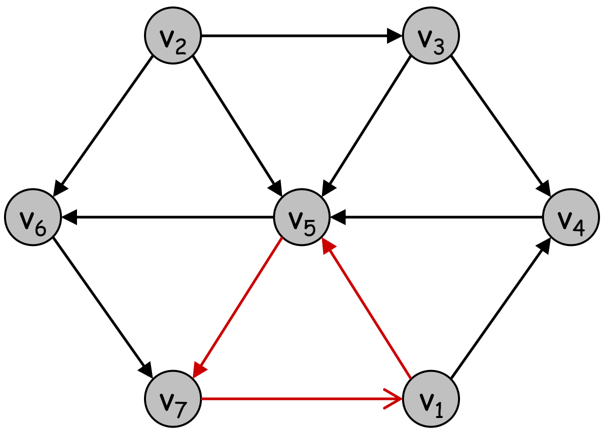
Directed Acyclic Graphs

Def. An DAG is a directed graph that contains no directed cycles.

Ex. Precedence constraints: edge (v_i, v_j) means v_i must precede v_j.



a DAG



Not a DAG

Precedence Constraints

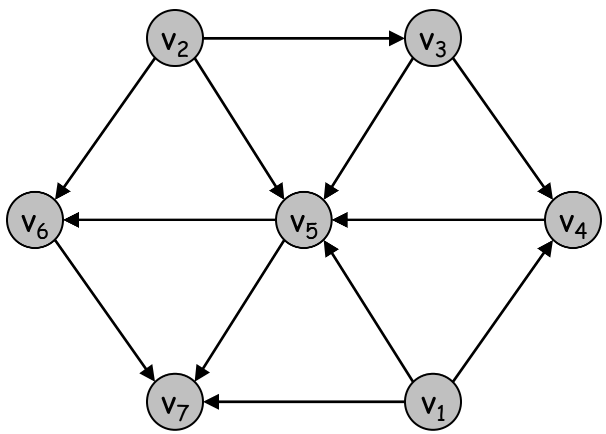
Precedence constraints. Edge (v_i, v_j) means task v_i must occur before v_j.

Applications.

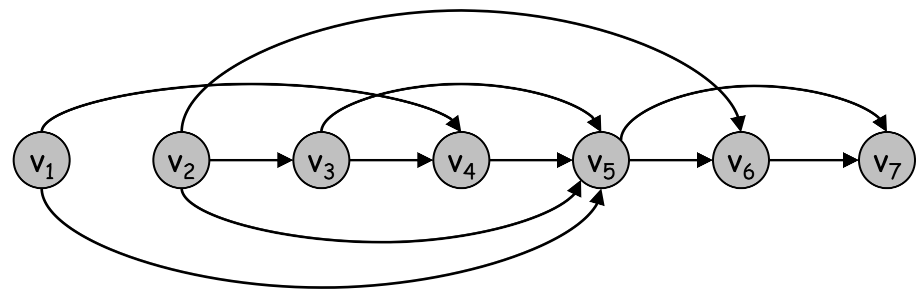
- Course prerequisite graph: course v_i must be taken before v_j.
- Compilation: module v_i must be compiled before v_j. Pipeline of computing jobs: output of job v_i needed to determine input of job v_j.

Directed Acyclic Graphs

Def. A topological order of a directed graph $G = (V, E)$ is an ordering of its nodes as $v_1, v_2, \ldots, v_n$ so that for every edge (v_i, v_j) we have $i < j$.



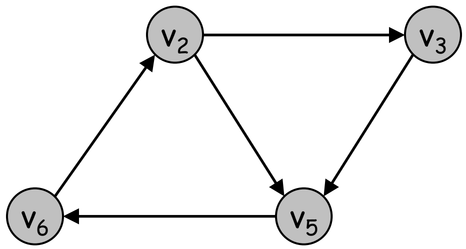
G



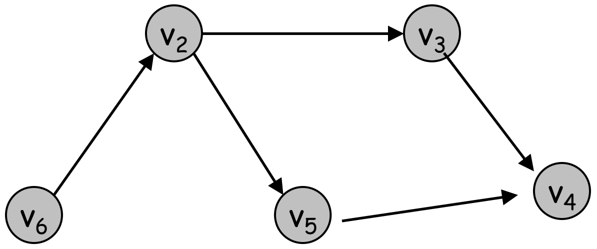
a topological ordering for G

Directed Acyclic Graphs

Def. A topological order of a directed graph $G = (V, E)$ is an ordering of its nodes as $v_1, v_2, \ldots, v_n$ so that for every edge (v_i, v_j) we have $i < j$.



No topological order



Topological orders:

v6->v2->v3->v5->v4

v6->v2->v5->v3->v4

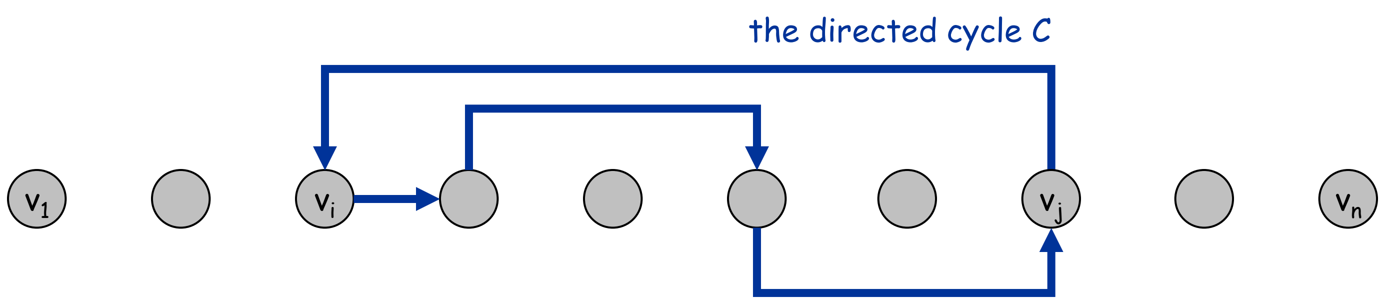
What is the relation between DAG's and topological orderings?

Directed Acyclic Graphs

Lemma. If G has a topological order, then G is a DAG.

Pf. (by contradiction)

- Suppose that G has a topological order $v_1, \ldots, v_n$ and that G also has a directed cycle C. Let's see what happens.
- Let v_i be the lowest-indexed node in C, and let v_j be the node just before v_i; thus (v_j, v_i) is an edge.
- By our choice of i, we have $i < j$.
- On the other hand, since (v_j, v_i) is an edge and $v_1, \ldots, v_n$ is a topological order, we must have $j < i$, a contradiction. ▪



the supposed topological order: $v_1, \ldots, v_n$

Directed Acyclic Graphs

Lemma. If G has a topological order, then G is a DAG.

Q. Does every DAG have a topological ordering?

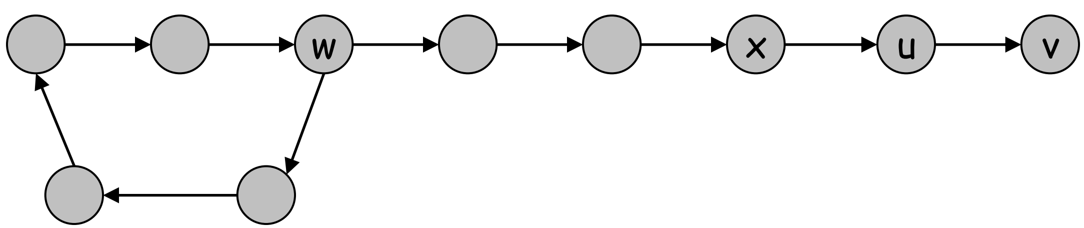
Q. If so, how do we compute one?

Directed Acyclic Graphs

Lemma. If G is a DAG, then G has a node with no incoming edges.

Pf. (by contradiction)

- Suppose that G is a DAG and every node has at least one incoming edge. Let's see what happens.
- Pick any node v, and begin following edges backward from v. Since v has at least one incoming edge (u, v) we can walk backward to u.
- Then, since u has at least one incoming edge (x, u), we can walk backward to x.
- Repeat until we visit a node, say w, twice.
- Let C denote the sequence of nodes encountered between successive visits to w. C is a cycle. ▪



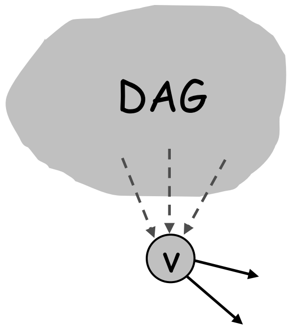
Directed Acyclic Graphs

Lemma. If G is a DAG, then G has a topological ordering.

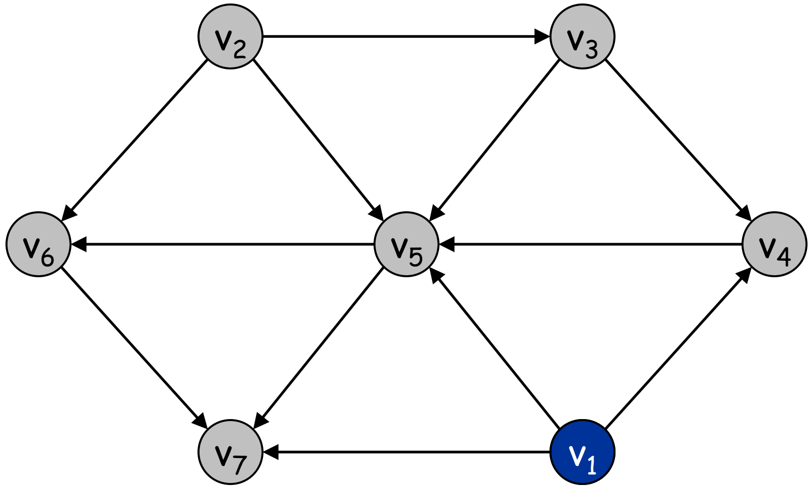
Pf. (by induction on n)

- Base case: true if n = 1.
- Given DAG on n > 1 nodes, find a node v with no incoming edges.
- G - { v } is a DAG, since deleting v cannot create cycles.
- By inductive hypothesis, G - { v } has a topological ordering.
- Place v first in topological ordering; then append nodes of G - { v }
- in topological order. This is valid since v has no incoming edges. ▪

```
To compute a topological ordering of G:
Find a node v with no incoming edges and order it first
Delete v from G
Recursively compute a topological ordering of G−{v}
  and append this order after v
```

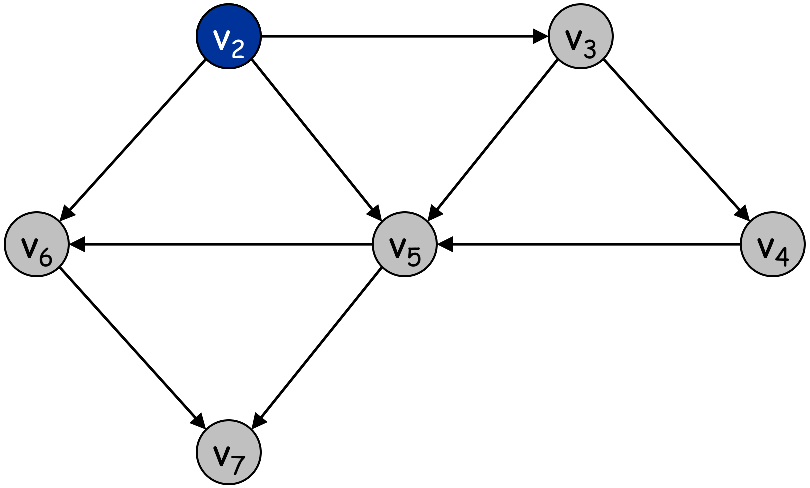


Topological Ordering Algorithm: Example



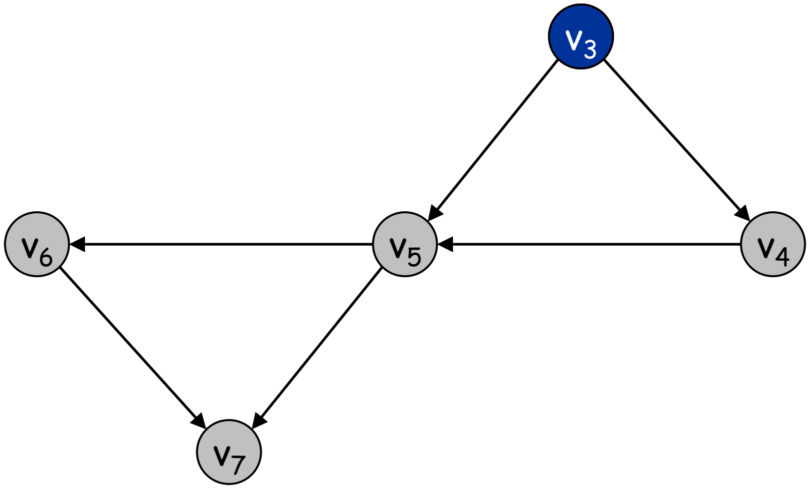
Topological order:

Topological Ordering Algorithm: Example



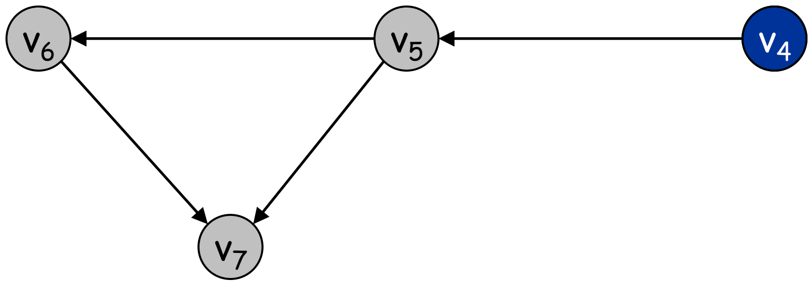
Topological order: v_1

Topological Ordering Algorithm: Example



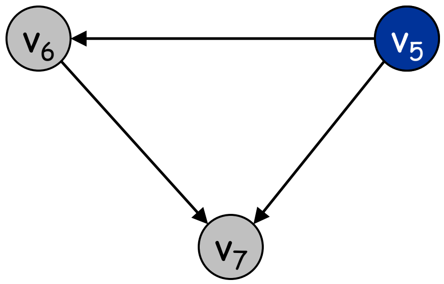
Topological order: v_1, v_2

Topological Ordering Algorithm: Example



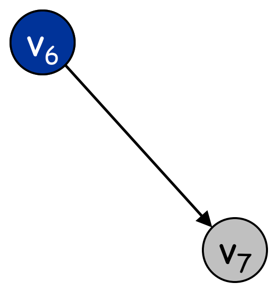
Topological order: v_1, v_2, v_3

Topological Ordering Algorithm: Example



Topological order: v_1, v_2, v_3, v_4

Topological Ordering Algorithm: Example

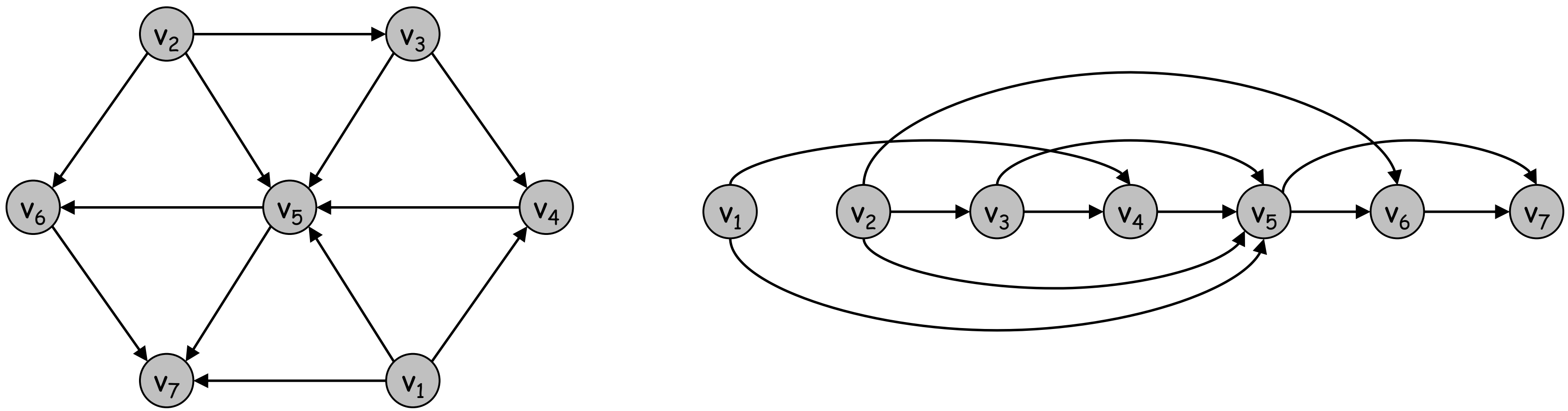


Topological order: v_1, v_2, v_3, v_4, v_5

Topological Ordering Algorithm: Example

Topological order: v_1, v_2, v_3, v_4, v_5, v_6

Topological Ordering Algorithm: Example



Topological order: $v_1, v_2, v_3, v_4, v_5, v_6, v_7$.

Topological Sorting Algorithm: Running Time

Algorithm 1

Modify the DFS (BFS) to fill a vector `count` that stores, for each node v, the number of remaining edges that are incident in v

```
i←0
While i< n
    v ←node with minimum degree in G
    i++
    If v has degree larger than 0
        Return G is not a DAG
    End If
    Add v to the topological order
    Remove v from G
    Update the vector count for the nodes adjacent to v
End
```

Topological Sorting Algorithm: Running Time

Analysis : `count` stored as a vector

- $O(n+m)$ to compute the `count`
- The loop executes at most n times
  - $O(n)$ to find the node v with minimum degree
  - $O(1)$ to remove v
  - $O(d(u))$ to update the neighbors of v
- ➔ $O(n^2 + m)$

Analysis : `count` stored as a heap

- $O(n+m)$ to compute the vector `count`
- The loop executes at most n times
  - $O(1)$ to find the node v with minimum degree
  - $O(\log n)$ to remove v
  - $O(d(u) \log n)$ to update the neighbors of v
- ➔ $O(n \log n + m \log n)$

Topological Sorting Algorithm: Running Time

Theorem. Algorithm finds a topological order in $O(m + n)$ time.

Pf.

- Maintain the following information:
  - `count[w]` = remaining number of incoming edges
  - S = set of remaining nodes with no incoming edges
- Initialization: $O(m + n)$ via single scan through graph.
- Update: to delete v
  - remove v from S
  - decrement `count[w]` for all edges from v to w, and add w to S if c `count[w]` hits 0
  - this is $O(1)$ per edge ▪

Exercicio Resolvido

Problema

Seja um grafo G=(V,E) representando a planta de um edifício. Inicialmente temos dois robos localizados em dois vértices de V, a e b, que devem alcançar os vértices c e d de V.

A cada passo um dos robos deve caminhar para um vertice adjacente ao vértice que ele se encontra no momento.

Exiba um algoritmo polinomial que decida se é possível, ou não, os robos partirem de a e b e chegarem em c e d, respectivamente, sem que em nenhum momento eles estejam a distância menor do que r, onde r é um inteiro dado.

Exercicio Resolvido

Solução

Seja H um grafo representando as configurações possíveis (posições dos robos) do problema. Cada nó de H corresponde a um par ordenado de vértices de V a distância menor ou igual a r.
Um par de nós u e v de H tem uma aresta se e somente em um passo é possível alcançar a configuração v a partir da configuração u em um único passo, ou seja, se u=(u1,u2) e v=(v1,v2) então uma das alternativas é válida
(i) u1=v1 e (u2,v2) pertence a E
(i) u2=v2 e (u1,v1) pertence a E

O problema portanto consiste em decidir se existe um camìnho entre o nó (a,b) e o nó (c,d) em H. Isso requer tempo linear no tamanho de H. Como H tem $O(n^2)$ vértices e $O(n^3)$ arestas, o algoritmo executa em $O(n^3)$.

Strongly connected components

Strongly connected components

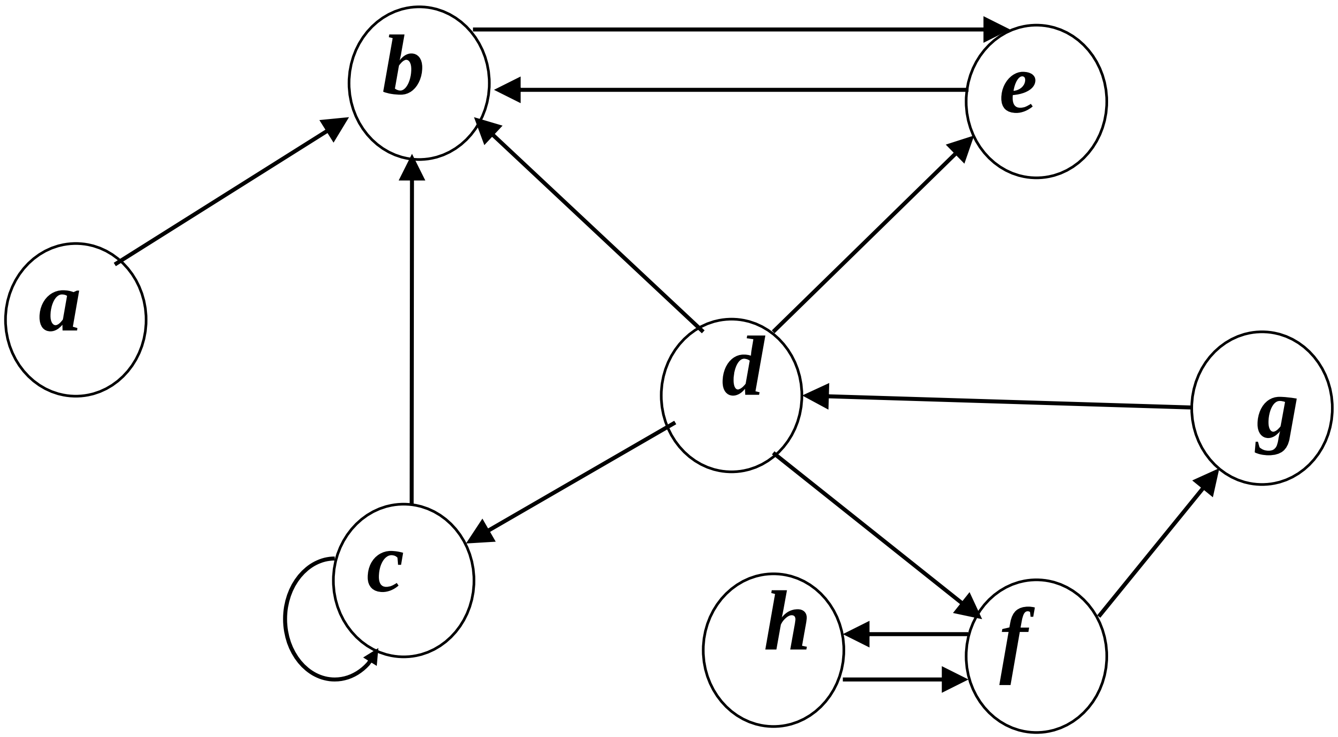
Definition: a strongly connected components C of a directed graph $G = (V,E)$ is a maximal sub-graphs (no common vertices or edges) such that for any two vertices u and v in C_i, there is a path from u to v and from v to u.

Equivalence classes of the binary *path*(u,v) relation, denoted by $u \sim v$.

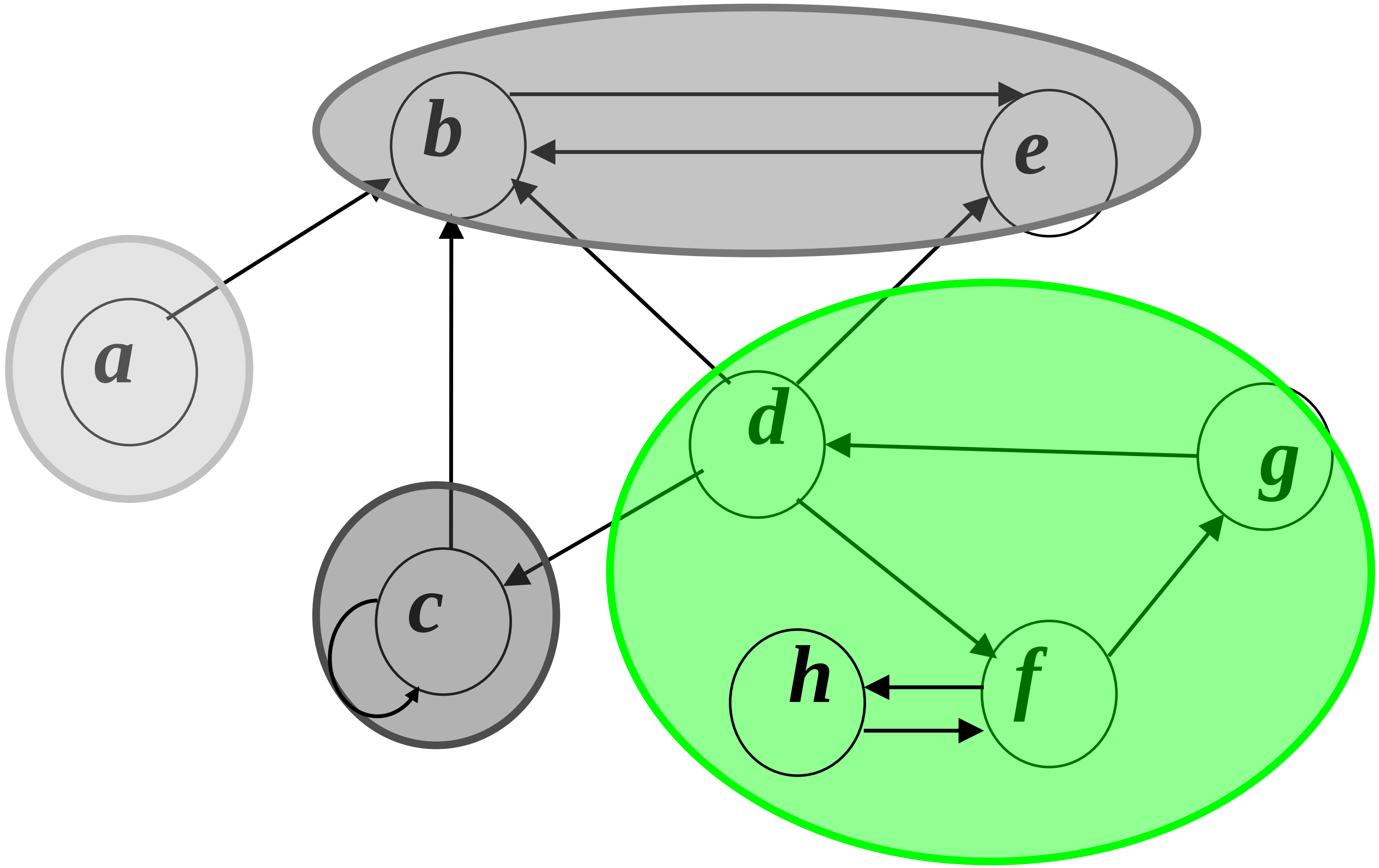
Applications: networking, communications.

Problem: compute the strongly connected components of G in linear time $\Theta(|V|+|E|)$.

Example: strongly connected components



Example: strongly connected components



Strongly connected components graph

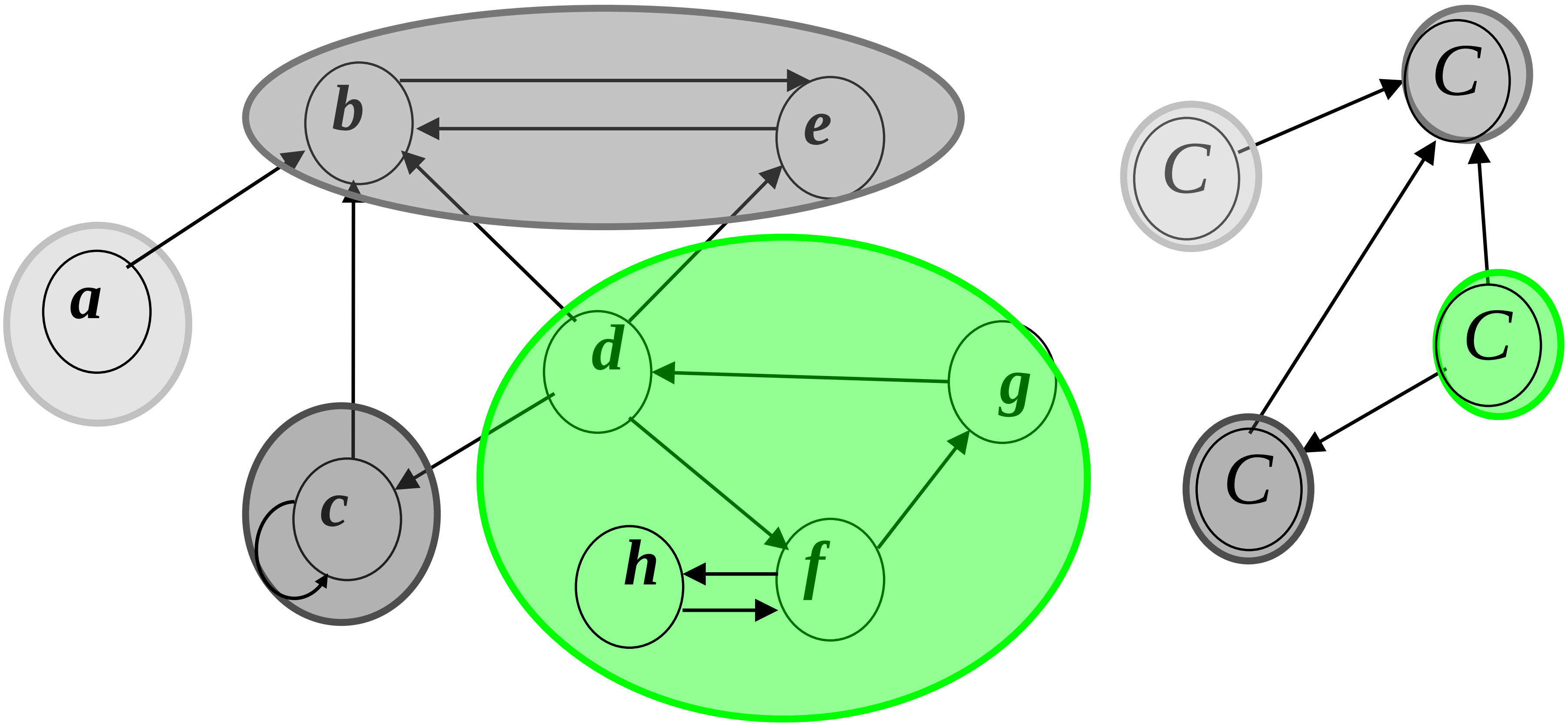
Definition: *the SCC graph* $G^{\sim} = (V^{\sim}, E^{\sim})$ of the graph $G = (V,E)$ is as follows:

- $V^{\sim} = \{C_1, \ldots, C_k\}$. Each SCC is a vertex.
- $E^{\sim} = \{(C_i, C_j) \mid i \neq j$ and there is $(x,y) \in E$, where $x \in C_i$ and $y \in C_j\}$. A directed edge between components corresponds to a directed edge between them from any of their vertices.

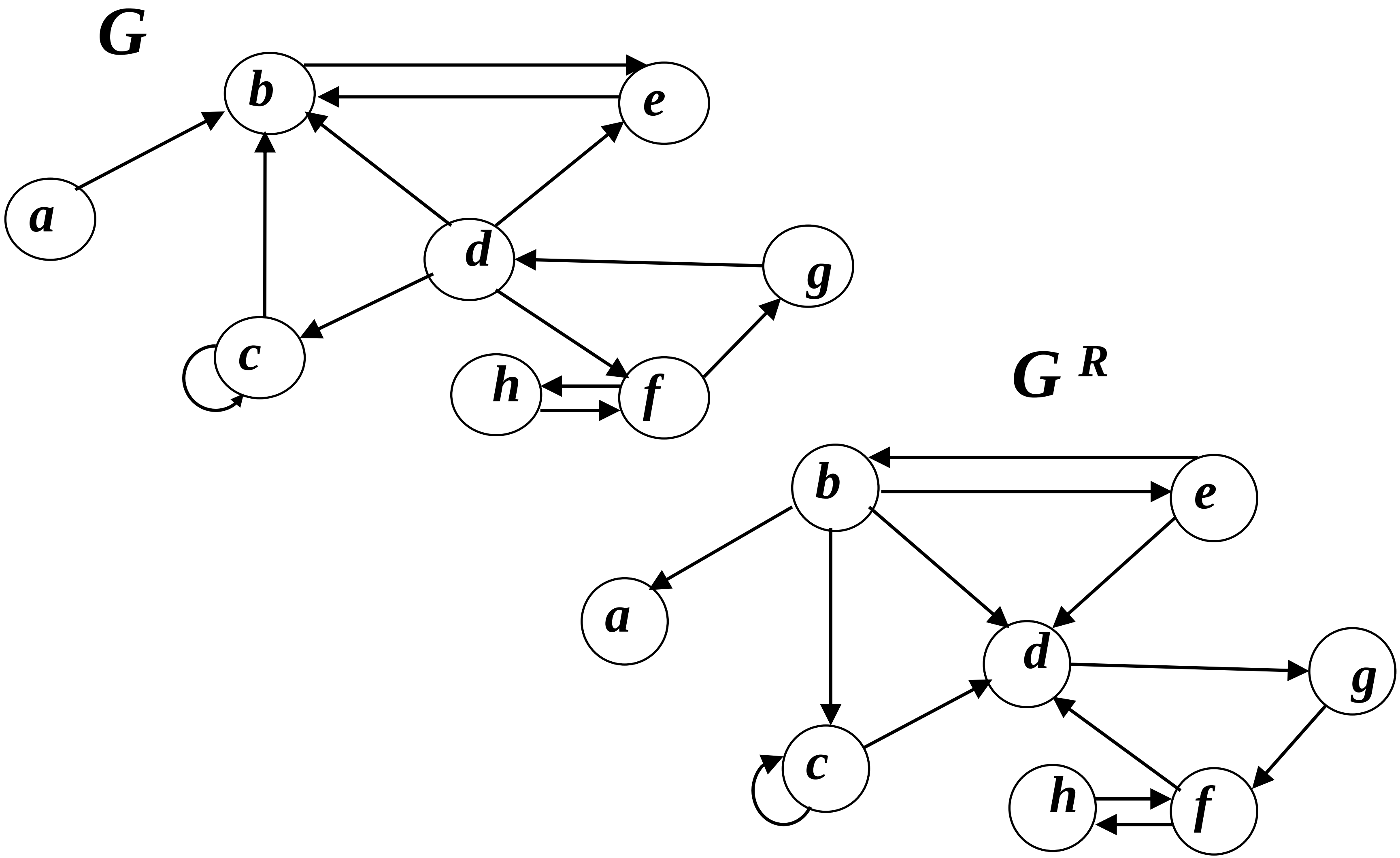
$G^{\sim}$ is a directed acyclic graph (no directed cycles)!

Definition: the reverse *graph* $G^R = (V, E^R)$ of the graph $G = (V,E)$ is G with its edge directions reversed: $E^R = \{(u,v) \mid (v,u) \in E\}$.

Example: SCC graph



Example: reverse graph G^R



SCC algorithm: Approach

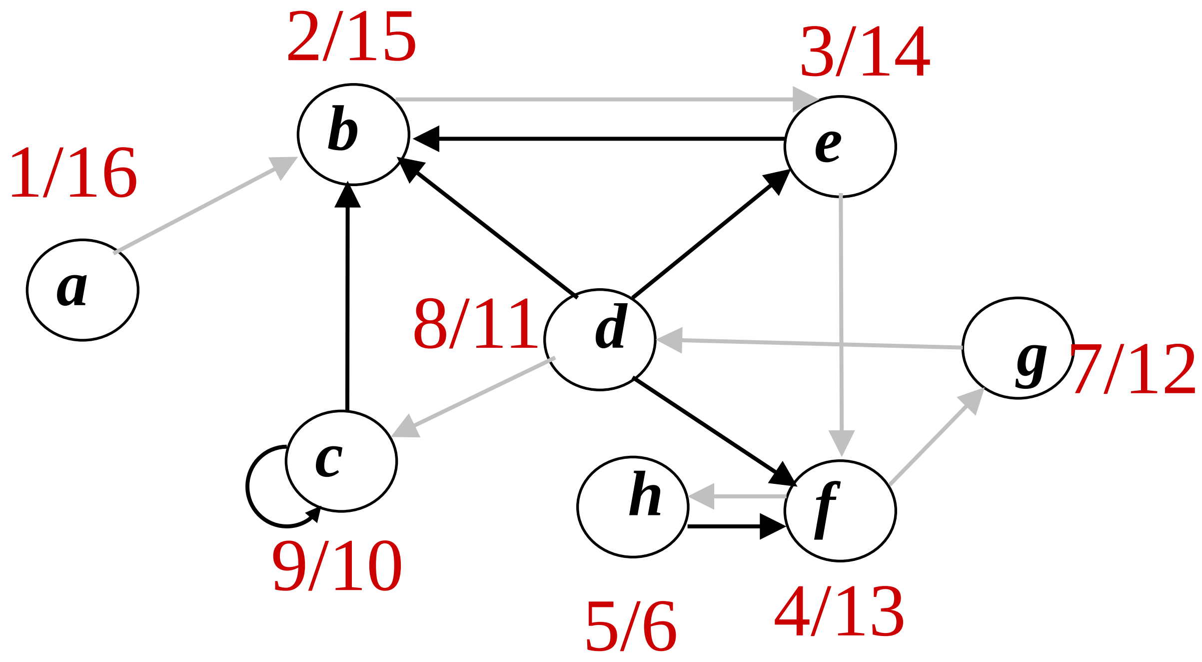
H← G~
While H is not empty
    v ← node of G that lies in a sink node of H   (*)
    C ← connected component retuned by DFS(v)
    H ← H - C
End

- *A sink node is a node with outdegree 0*
- *If we manage to execute (*) we are done*

DFS numbering

Useful information

- pre(v): "time" when a node is visited in a DFS
- post(v): "time" when DFS(v) finishes



DFS numbering

```
DFS(G)
0.    time ← 1
1     Para todo v em G
2         Se v não visitado então
3              DFS-Visit(G, v)

DFS-Visit(G, v)
1     Marque v como visitado
1.5   pre(v)← time; time++
2     Para todo w em Adj(v)
3         Se w não visitado então
4              Insira aresta (v, w) na árvore
5              DFS-Visit(G, w)
6     Post(v)← time; time++
```

DFS numbering

Property If C an D are strongly connected components and there is an edge from a node in C to a node in D, then the highest post() in C is bigger than the highest post() number in D

*Proof. Let **c** be the first node visited in C and **d** be the first node visited in D.*

*Case i) **c** is visited before **d**.*

- *DFS(**c**) visit all nodes in D (they are reachable from **c** due to the existing edge)*
- *Thus, post(**c**) > post(x) for every x in D*

DFS numbering

Property If C an D are strongly connected components and there is an edge from a node in C to a node in D, then the highest post() in C is bigger than the highest post() number in D

*Proof. Let **c** be the first node visited in C and **d** be the first node visited in D*

*Case 2) **d** is visited before **c**.*

- *DFS(**d**) visit all nodes in D because all of them are reachable from **d***
- *DFS(**d**) does not visit nodes from C since they are not reachable from **d**.*
- *Thus , post(x) <post(y) for every pair of nodes x,y, with x in D and y in C*

DFS numbering

Corollary. The node of G with highest post number lies in a source node *in G~*

Observation 1. *The strongly connected components are the same in G and G^R*

Observation 2. *If a node lies in a source node of G~ then it lies in a sink node in (G^R)~*

SCC algorithm

Idea: compute the SCC graph $G^{\sim} = (V^{\sim}, E^{\sim})$ with two DFS, one for G and one for its reverse G^R, visiting the vertices in reverse order.

SCC(G)

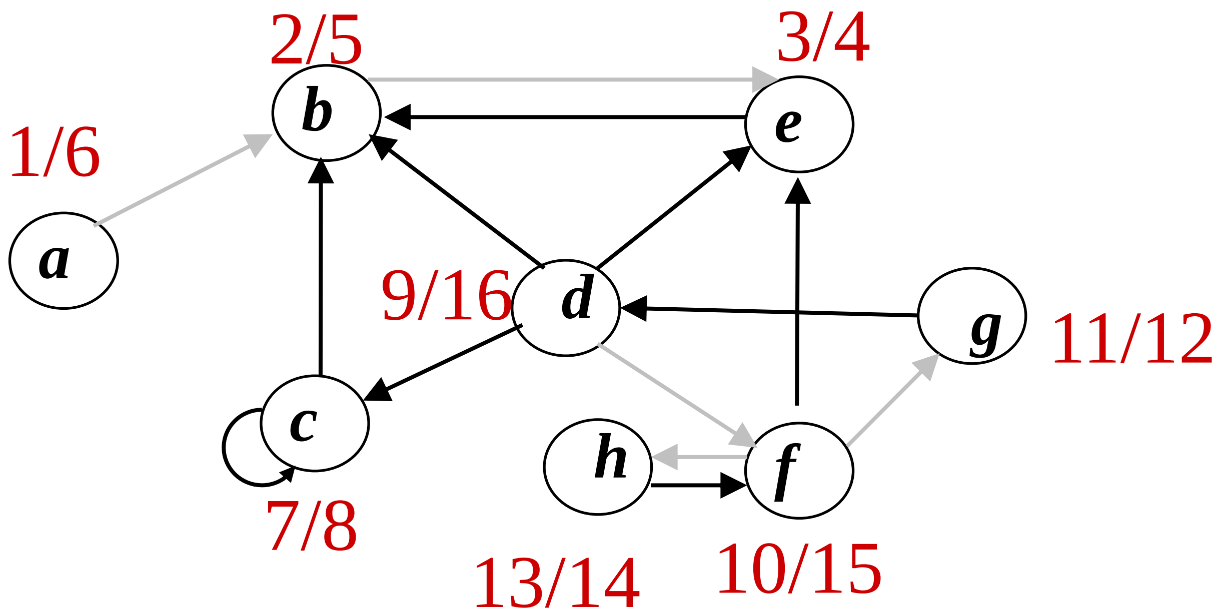
1. DFS(G) to compute $post[v]$, $\forall v \in V$

2. Compute G^R

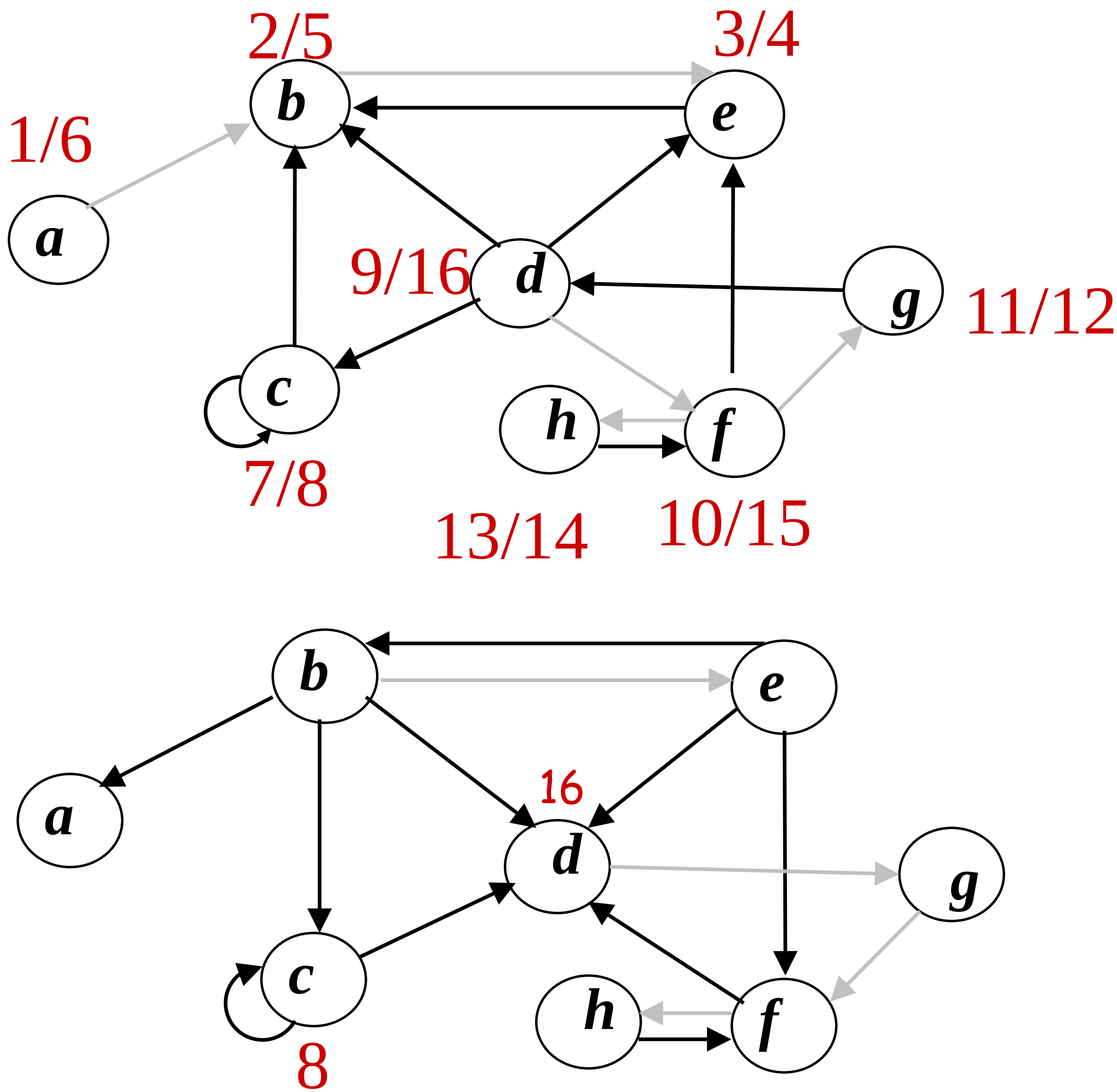
3. DFS(G^R) in the order of decreasing post[v]

4. Output the vertices of each tree in the DFS forest as a separate SCC.

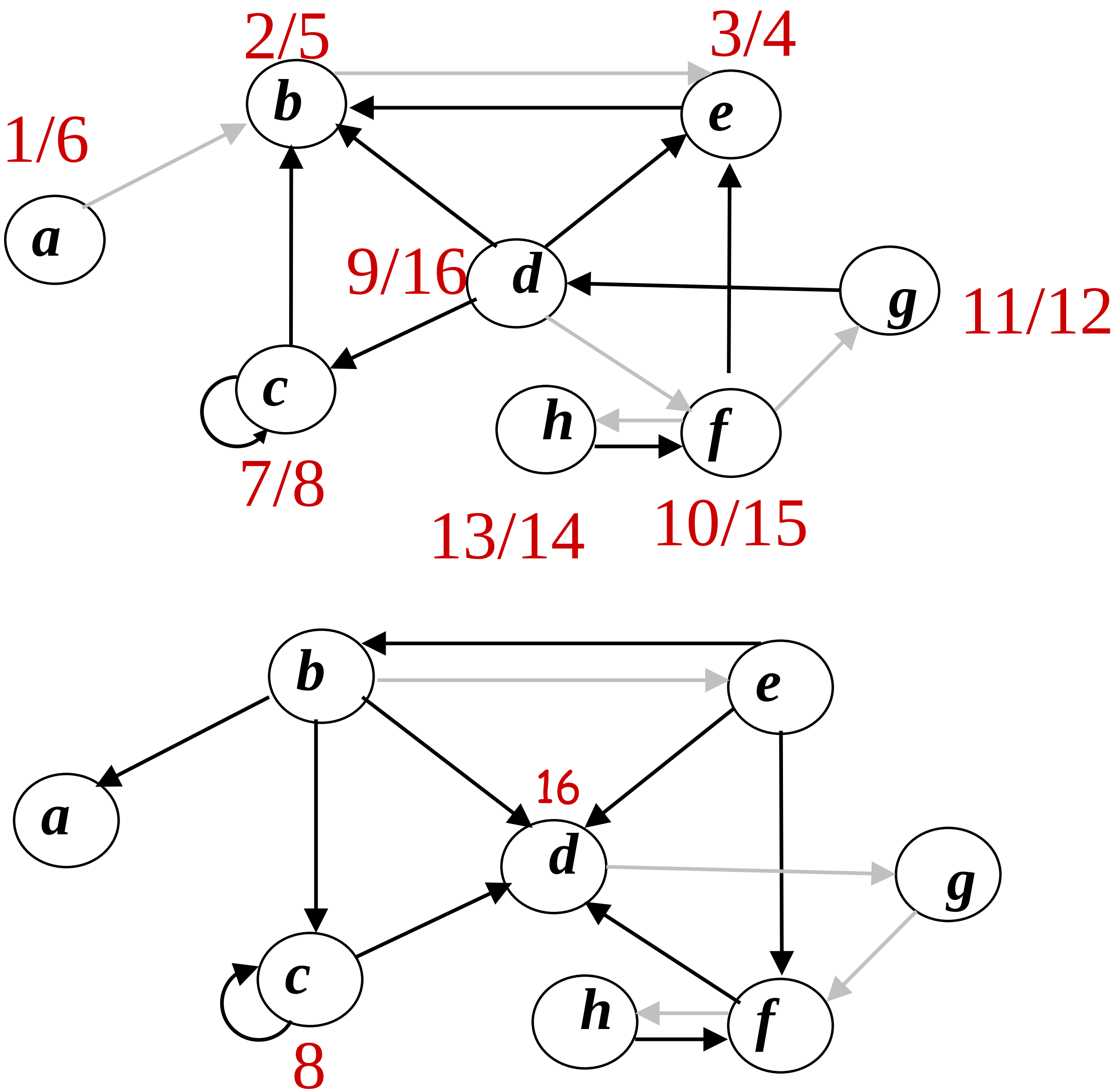
Example: computing SCC

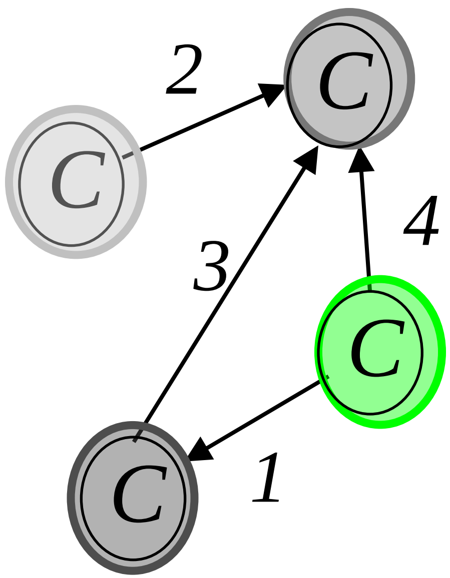


Example: computing SCC

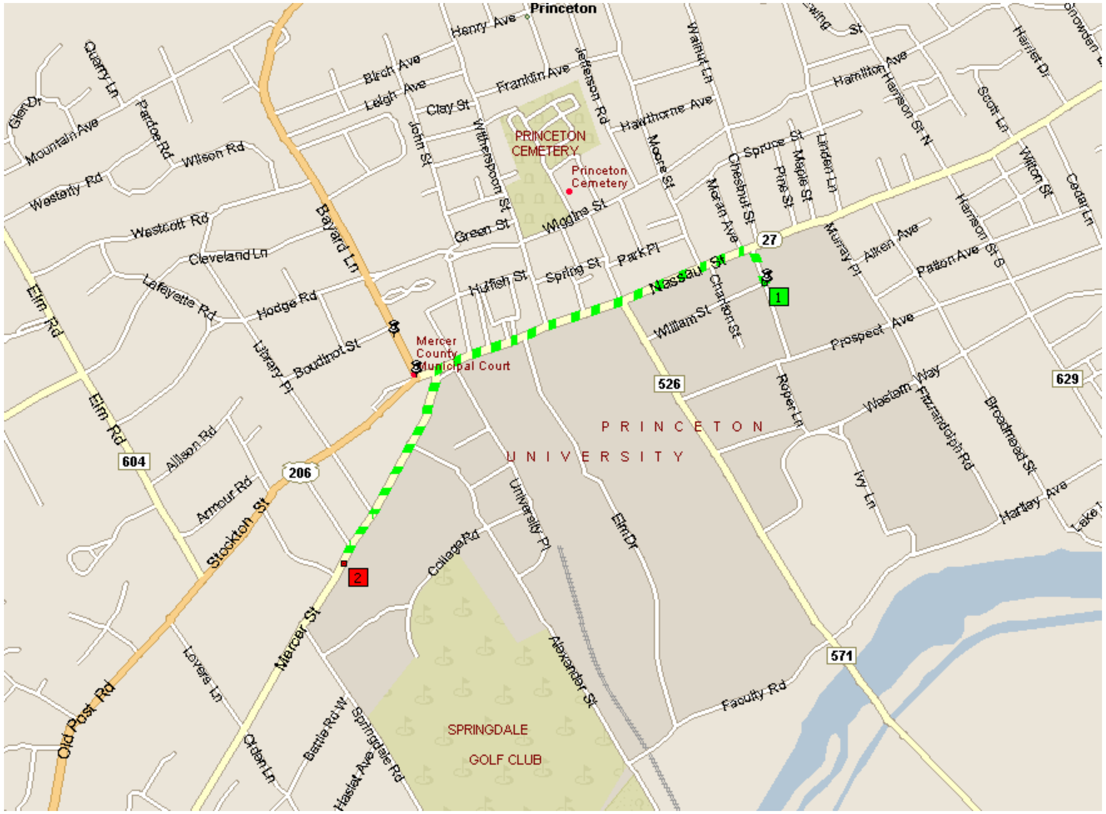


Example: computing SCC





4.4 Shortest Paths in a Graph



shortest path from Princeton CS department to Einstein's house

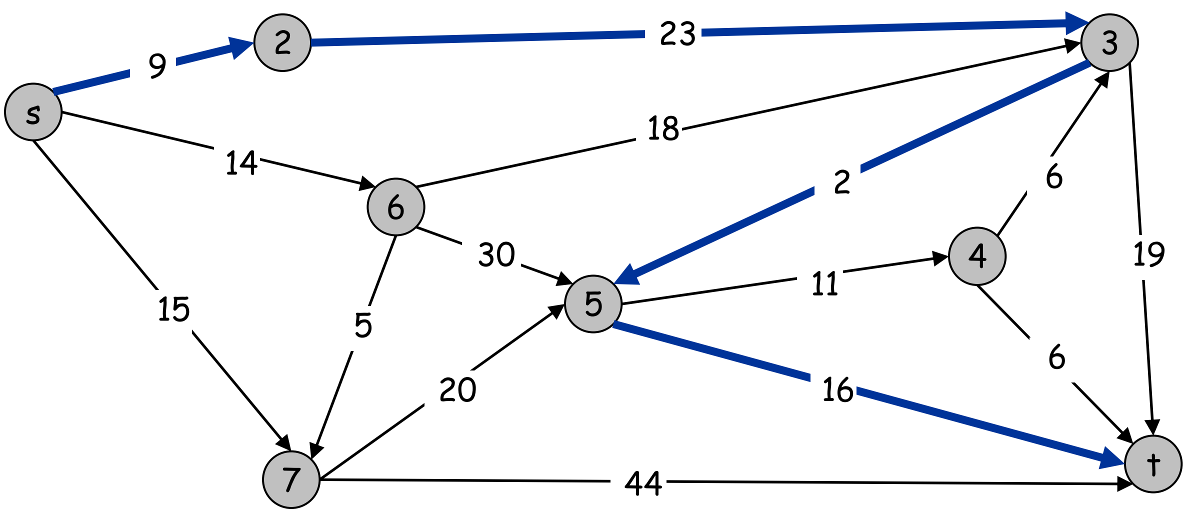
Shortest Path Problem

Shortest path network.

- Directed graph $G = (V, E)$.
- Source s, destination t.
- Length c_e = length of edge e. (non-negative numbers)

Shortest path problem: find shortest directed path from s to t.

cost of path = sum of edge costs in path

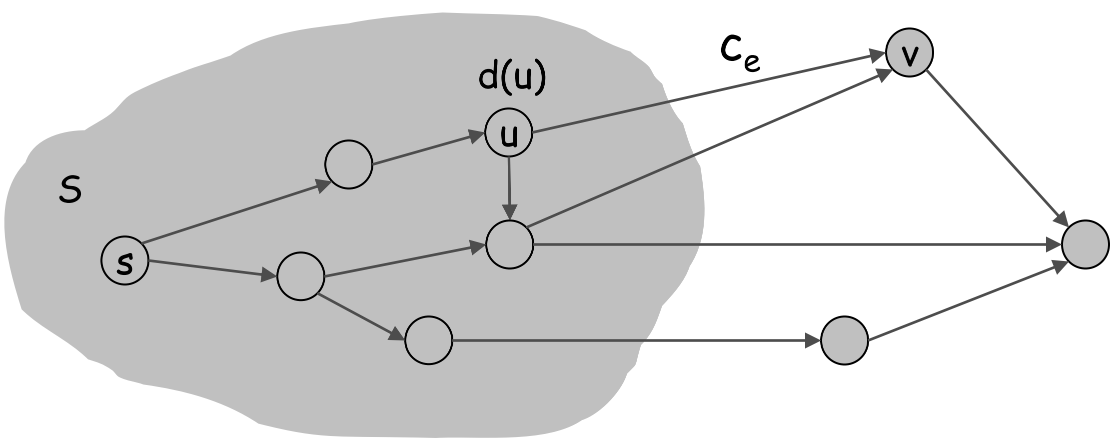


Cost of path s-2-3-5-t
= 9 + 23 + 2 + 16
= 50.

Dijkstra's Algorithm

Approach

- Find the node closest to the s, then the second closest, then the third closest, and so on ...
- Key observation:
  - the shortest path from s to the k-th closest node can be decomposed as the shortest path from s to the i-th closest node (for some i<k) and an edge from the i-th closest node to the k-th closest node.



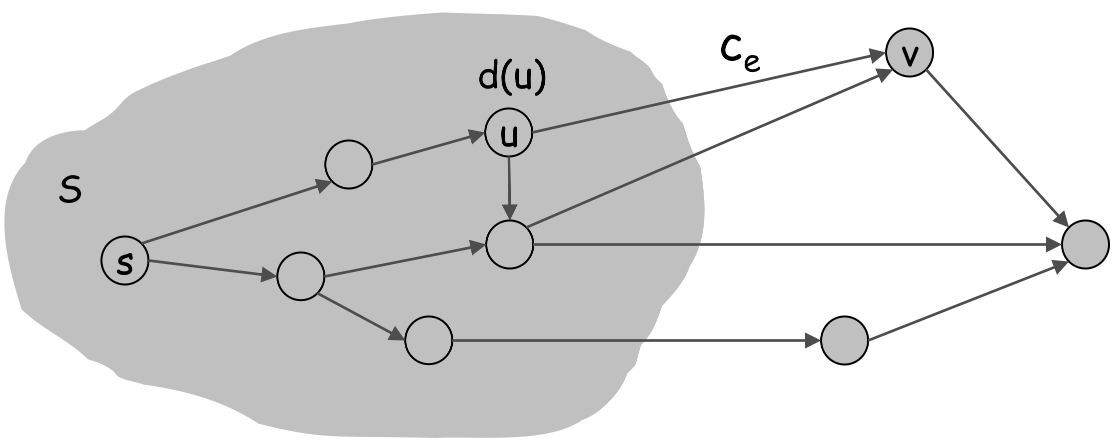
Dijkstra's algorithm.

- Maintain a set of explored nodes S for which we have determined the shortest path distance d(u) from s to u.
- Initialize S = { s }, d(s) = 0.
- Repeatedly choose unexplored node v which minimizes

$$\pi(v) = \min_{e = (u,v) : u \in S} d(u) + c_e ,$$

add v to S, and set d(v) = π(v).

shortest path to some u in explored part, followed by a single edge (u, v)



Dijkstra's Algorithm

Dijkstra's algorithm.

- Maintain a set of explored nodes S for which we have determined the shortest path distance d(u) from s to u.
- Initialize S = { s }, d(s) = 0.
- Repeatedly choose unexplored node v which minimizes

$$\pi(v) = \min_{e = (u,v) : u \in S} d(u) + c_e ,$$

shortest path to some u in explored part, followed by a single edge (u, v)

add v to S, and set d(v) = π(v).

Complexity (Naïve Implementation)

- n loops, one for each node
- (n+m) to find the the node with minimum π
- ➔ O(n(n+m)) time

Dijkstra's Algorithm: Implementation

For each unexplored node, explicitly maintain $\pi(v) = \min_{e=(u,v):u\in S} d(u) + c_e$.

- Next node to explore = node with minimum π(v).
- When exploring v, for each incident edge e = (v, w), update

$$\pi(w) = \min \{ \pi(w), \pi(v) + c_e \}.$$

Efficient implementation. Maintain a priority queue of unexplored nodes, prioritized by π(v).'

PQ Operation	Dijkstra	Array	Binary heap
Insert	n	n	log n
ExtractMin	n	n	log n
ChangeKey	m	1	log n
IsEmpty	n	1	1
Total		n^2	m log n

† Individual ops are amortized bounds

Dijkstra's Algorithm

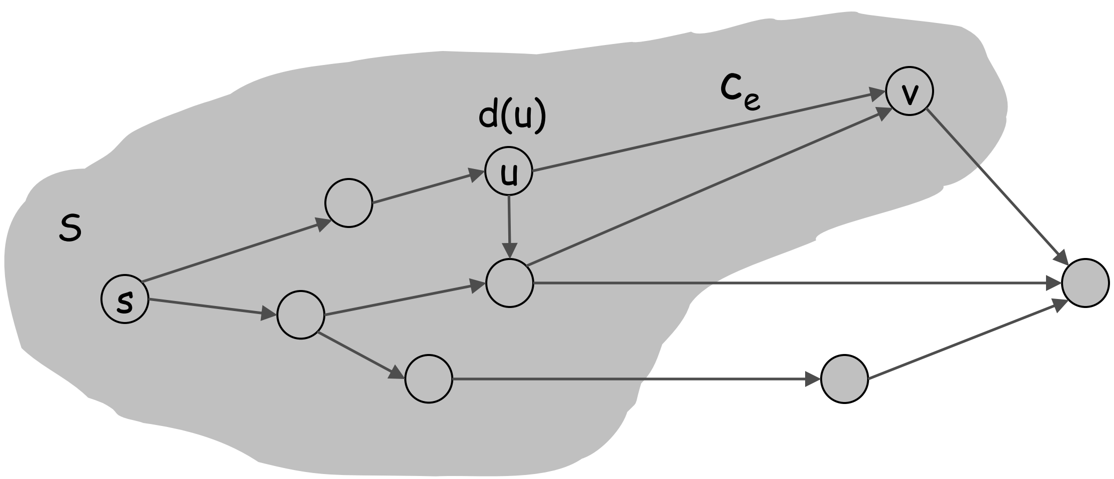
Dijkstra's algorithm.

- Maintain a set of explored nodes S for which we have determined the shortest path distance d(u) from s to u.
- Initialize S = { s }, d(s) = 0.
- Repeatedly choose unexplored node v which minimizes

$$\pi(v) = \min_{e = (u,v) : u \in S} d(u) + c_e ,$$

shortest path to some u in explored part, followed by a single edge (u, v)

add v to S, and set d(v) = π(v).



Dijkstra's Algorithm: Proof of Correctness

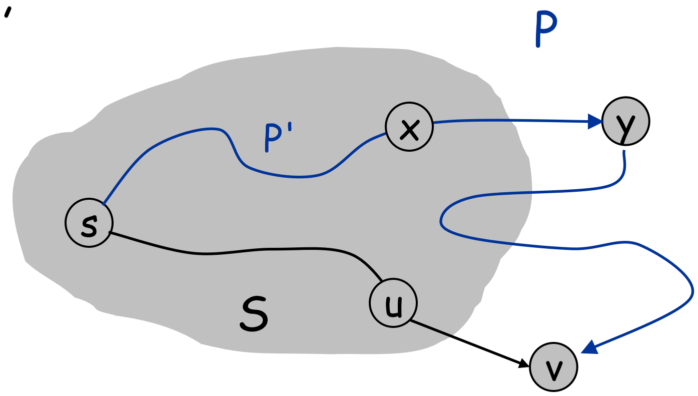
Invariant. For each node $u \in S$, $d(u)$ is the length of the shortest s-u path.

Pf. (by induction on $|S|$)

Base case: $|S| = 1$ is trivial.

Inductive hypothesis: Assume true for $|S| = k \geq 1$.

- Let v be next node added to S, and let u-v be the chosen edge.
- The shortest s-u path plus (u, v) is an s-v path of length $\pi(v)$.
- Consider any s-v path P. We'll see that it's no shorter than $\pi(v)$.
- Let x-y be the first edge in P that leaves S, and let P' be the subpath to x.
- P is already too long as soon as it leaves S.



$$\underbrace{c(P) \geq}_{\text{nonnegative weights}} c(P') + \underbrace{c(x,y) \geq}_{\text{inductive hypothesis}} d(x) + \underbrace{c(x,y)}_{\text{defn of } \pi(y)} \geq \underbrace{\pi(y)}_{\text{Dijkstra chose v instead of y}} \geq \pi(v)$$