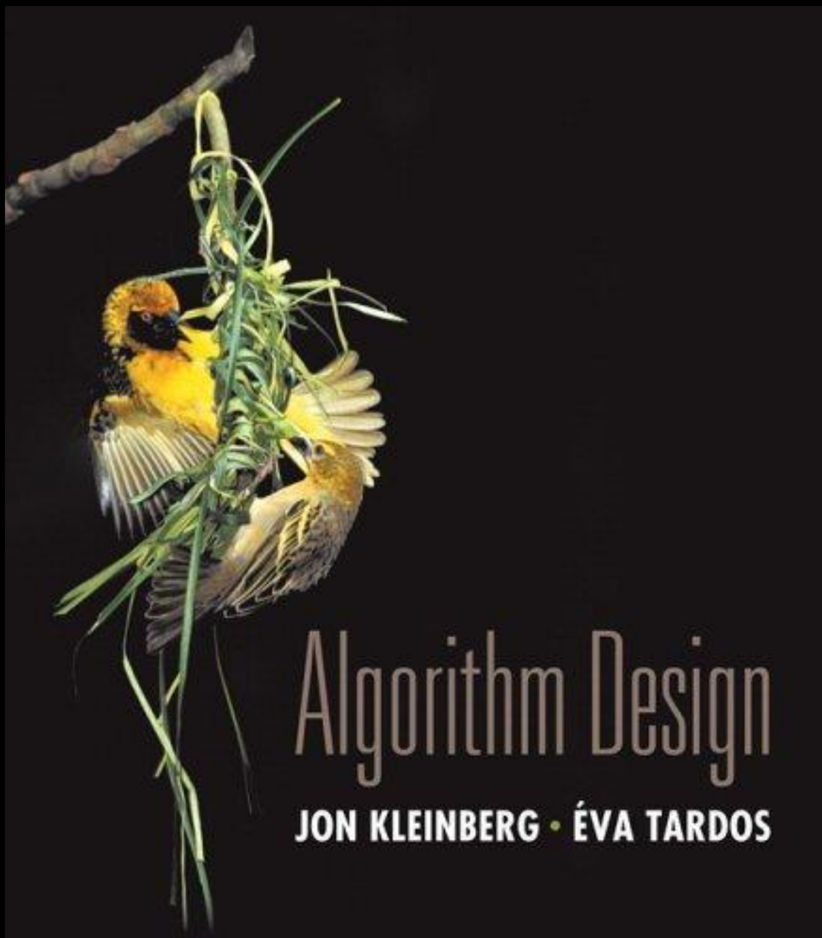
Chapter 4.1
Análisis y Diseño de Algoritmos (Algorítmica III)
-Graphs-

Profesores:
Herminio Paucar.
Luis Guerra.



Chapter 3

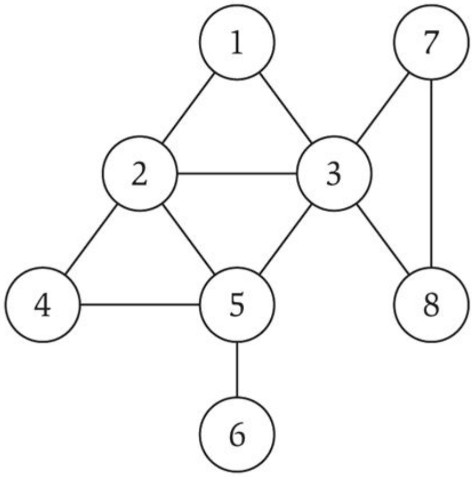
Graphs

Basic Definitions and Applications

Undirected Graphs

Undirected graph. G = (V, E)

- V = nodes.
- 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 = { 1-2, 1-3, 2-3, 2-4, 2-5, 3-5, 3-7, 3-8, 4-5, 5-6 }

n = 8

m = 11

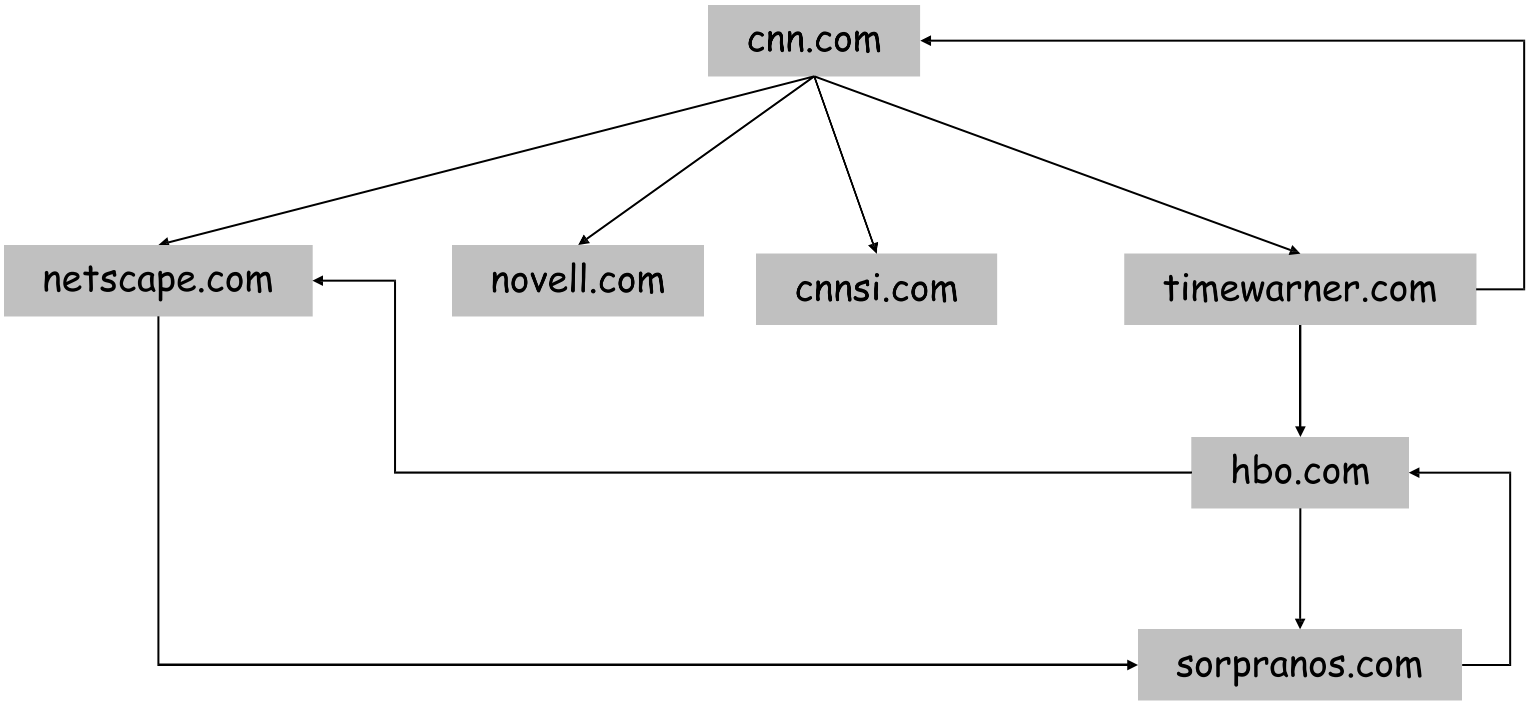
Some Graph Applications

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.

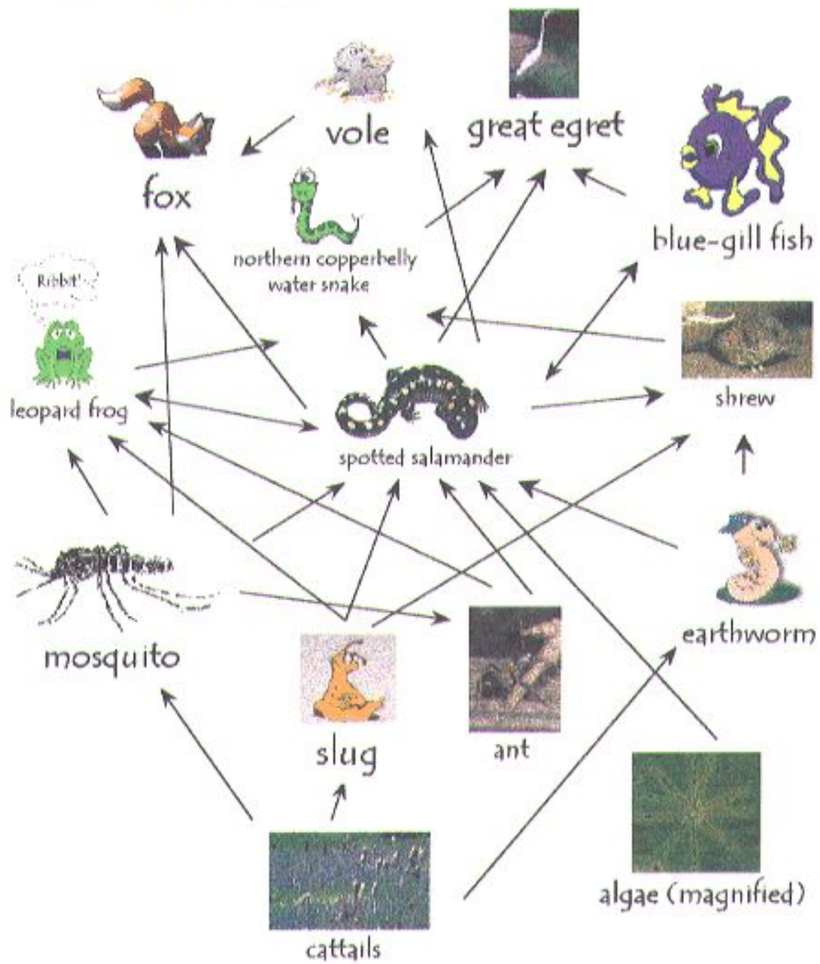
- Node: people.
- Edge: relationship between two people.

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

Ecological Food Web

Food web graph.

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

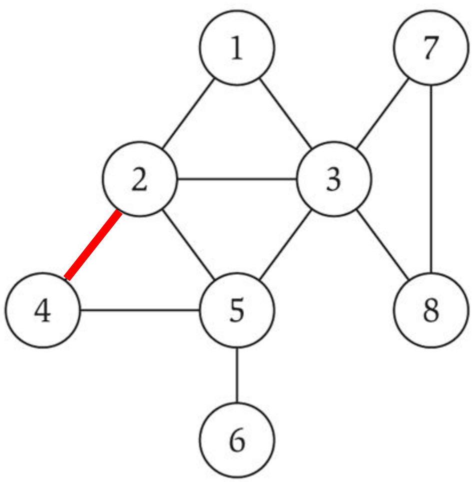


Reference: http://www.twingroves.district96.k12.il.us/Wetlands/Salamander/SalGraphics/salfoodweb.giff

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

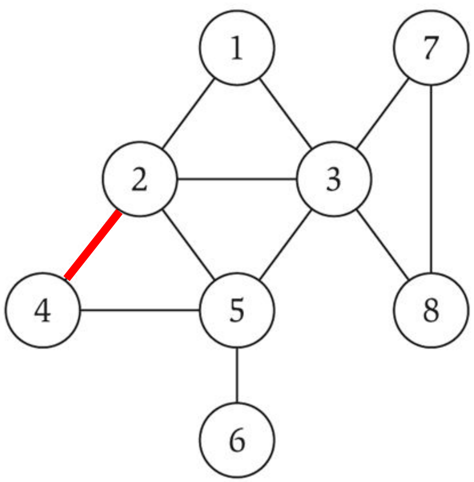
Drawback: independent of number of edges

- In line graph (n vertices and n-1 edges) adjacency matrix is full of 0's

Facebook

- 750M nodes
- 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

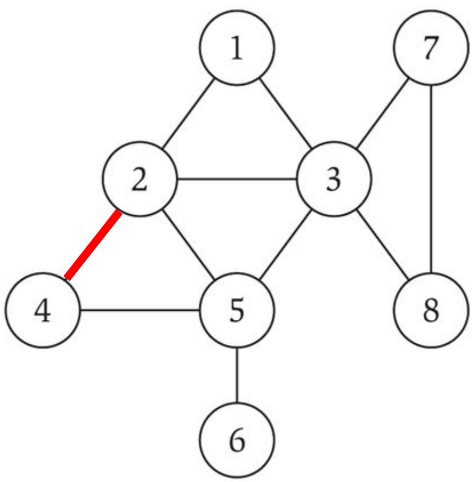
Graph Representation: Adjacency List

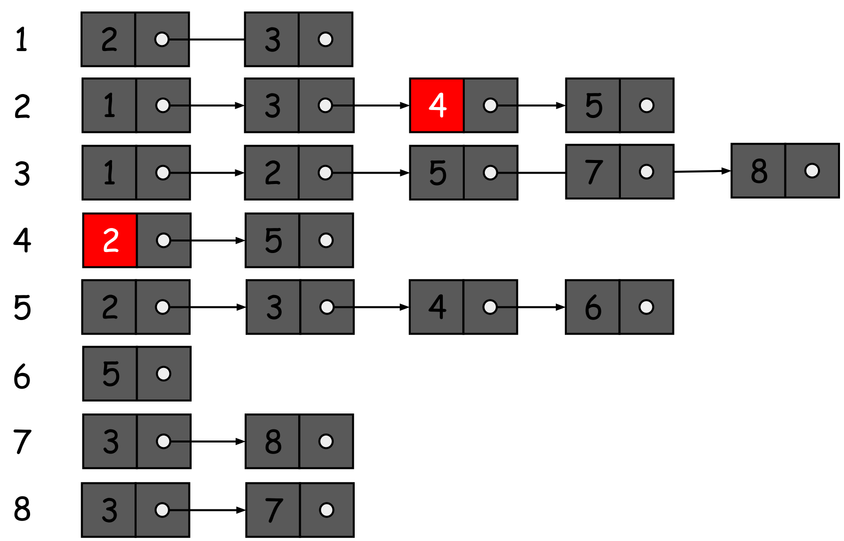
Adjacency list. Node indexed array of lists.

- Two representations of each edge.

degree = number of neighbors of u

- 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.





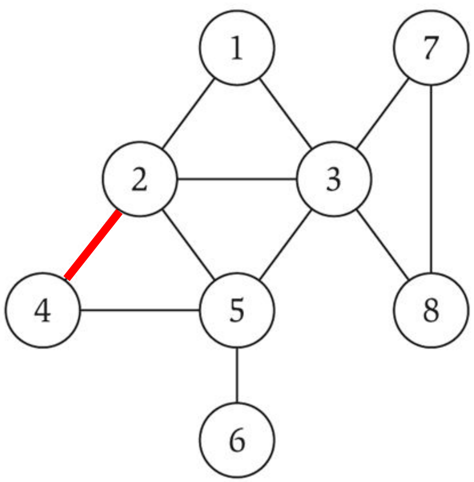
Graph Representation: Adjacency List

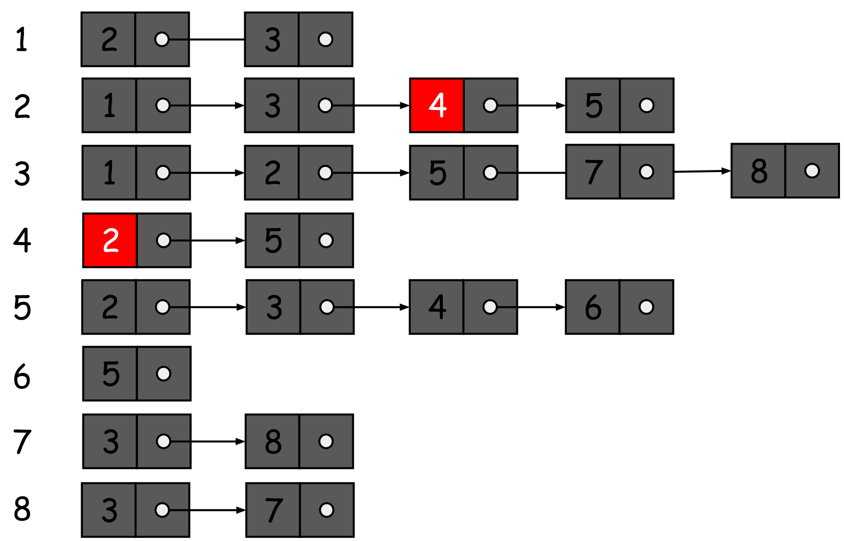
Advantage: sensitive to the number of edges

Facebook

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

➔ **100 Gigabytes** to store approximately 50 Billion edges;





Graph Traversal

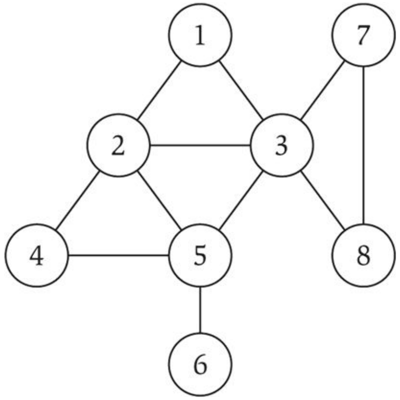
Connectivity

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.

- Reachable states
- Fastest route
- Minimum number of connections to reach a person on LinkedIn
- Fewest number of hops in a communication network.

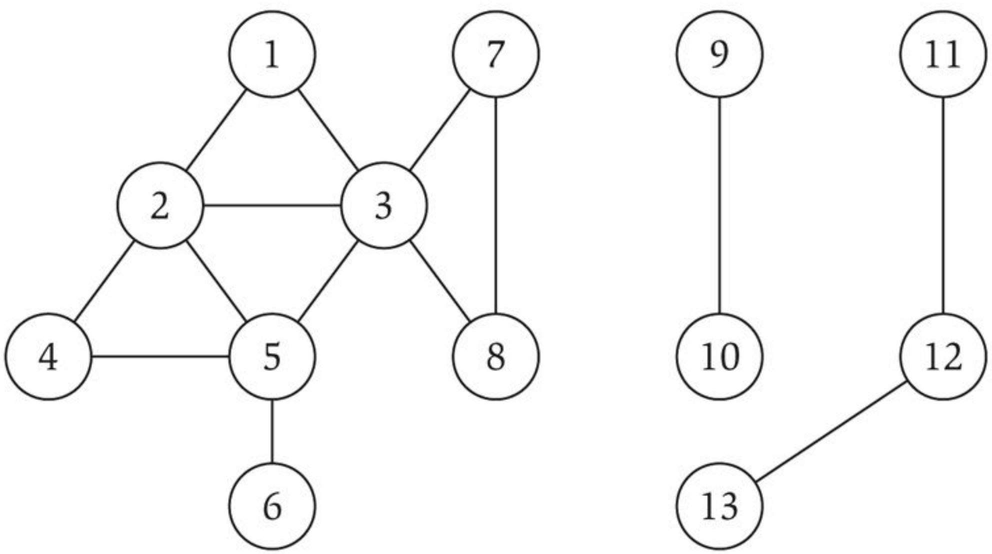


Paths and Connectivity

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

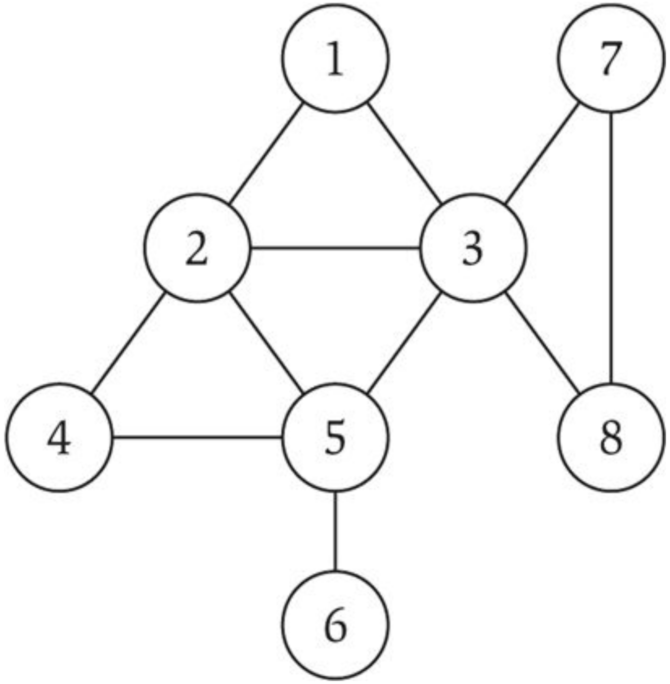
Def. A path is simple if all nodes are distinct.

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



Distance

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



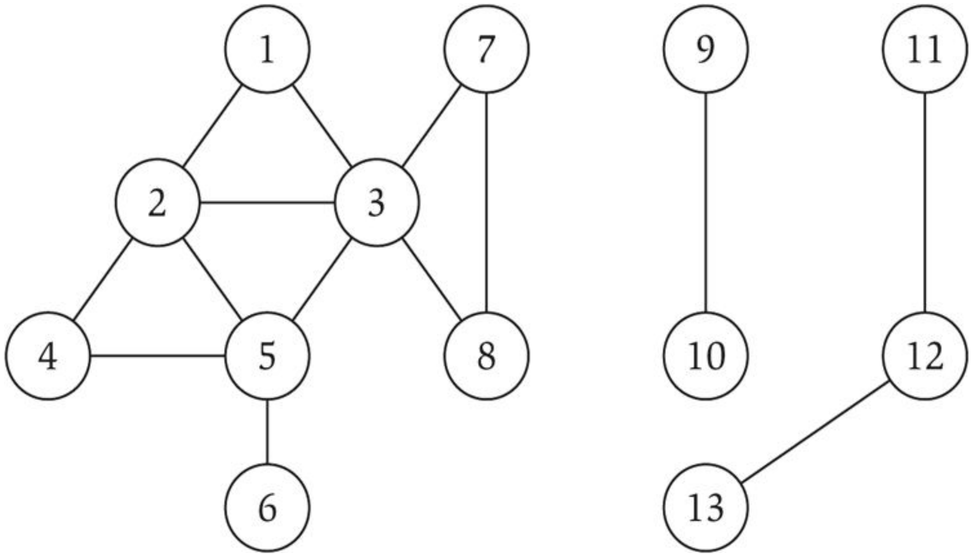
Distance(1,4) = 2
Distance(6,3) = 2
Distance(7,8) = 1

Connected Component

Def (informal): Connected Component: The connected "blocks" that compose the graph

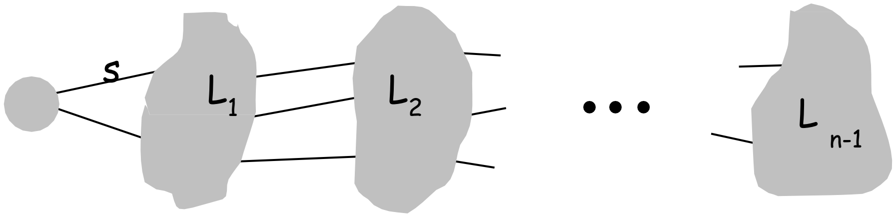
Def: Connected set: S is a connected set if v is reachable from u and u is reachable from v for every u,v in S

Def (formal): S is a **connected component** if is a connected set and for every u in V–S, S ∪ {u} is not connected



Breadth First Search

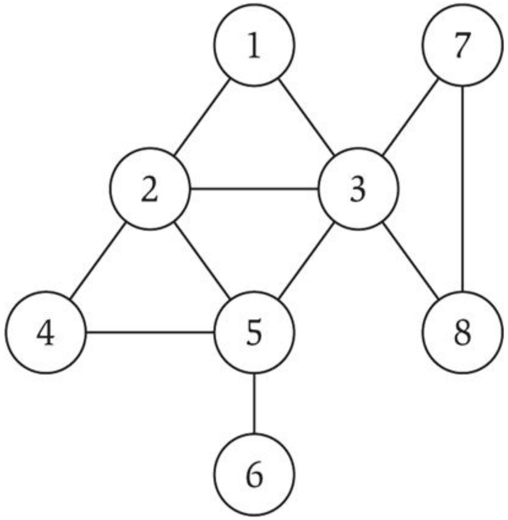
BFS intuition. Explore outward from ***s*** in all possible directions, adding nodes one "layer" at a time.



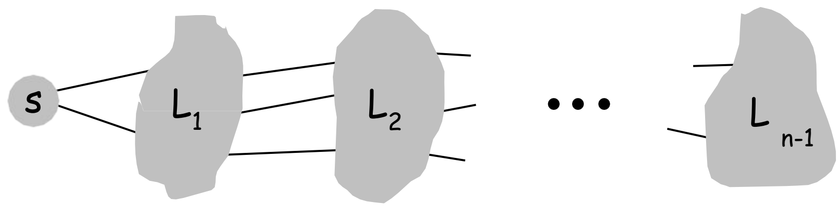
Algorithm BFS(G, s).

- 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.

Ex: Run BFS(G,1) on this graph

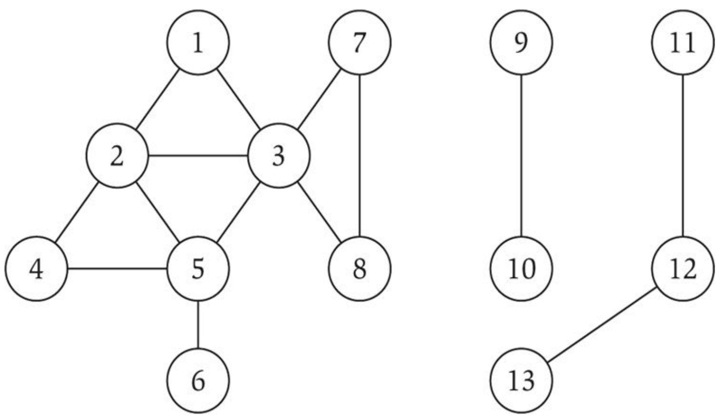


Breadth First Search



Q: **What is the distance of a node in L_i from s?**

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



Q: If G is the graph on the right, which nodes does BFS(G,1) visit?

A: Nodes 1,2,...,8

Q: How can we use BFS(G,s) to visit all nodes in the graph?

A: For each node **s** in G

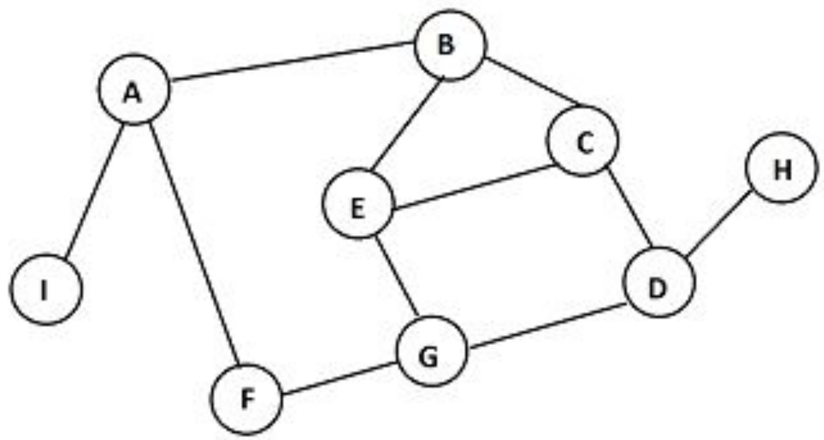
If **s** has not been visited, do BFS(G,s)

End for

Breadth First Search: Implementation

breadthFirstSearch(G, s)

```
foreach v ∈ V do
    dist[v] ← ∞; pred[v] ← -1
    color(v) ← White
dist(s) ← 0; color[s] ← Gray
Q ← empty Queue; enqueue (Q, s)
while (Q is not empty) do
    u ← dequeue (Q)
    foreach v ← Adj(u) do
        if (color[v] is White) then
            dist[v] ← dist[u] + 1
            pred[v] ← u
            color[v] ← Gray
            enqueue (Q, v)
    color(u) ← Black
```

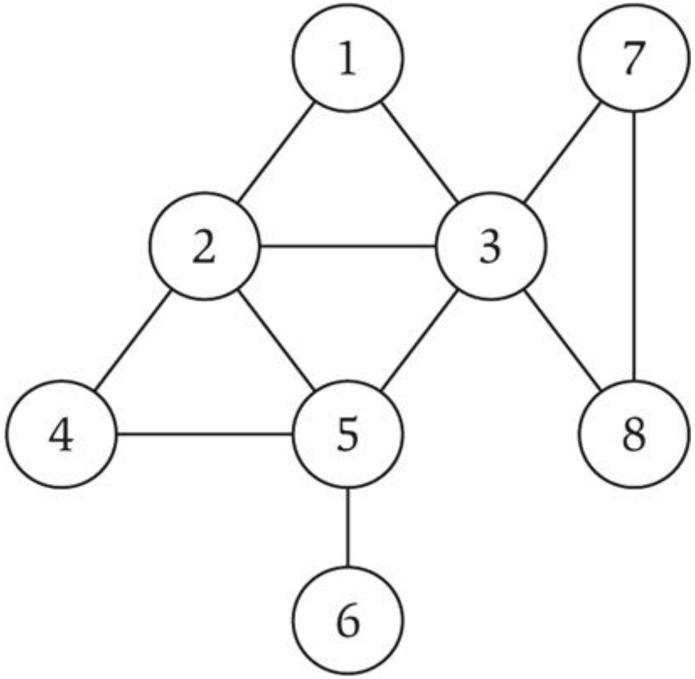


Breadth First Search: Analysis

BFS can “touch” a node many times

- In graph below, BFS(G,1) touches node 3 when looking at neighbors of 1, neighbors of 2, neighbors of 5…

But only touches each edge twice (once in each direction)



Breadth First Search: Analysis

Analysis $O(n^2)$:

- Initialization part costs in total O(n)
- The cost of each execution of the blue block is at most O(n)
- All red for's together are just sweeping through all layers => one iteration per node
- Total cost for all executions of blue block is $O(n^2)$
- Total: $O(n^2)$

```
BFS(G, s)
    foreach v ∈ V do
        dist[v] ← ∞; pred[v] ← -1
        color(v) ← White
    dist(s) ← 0; color[s] ← Gray
    Q ← empty Queue; enqueue (Q, s)
    while (Q is not empty) do
        u ← dequeue (Q)
        foreach v ← Adj(u) do
            if (color[v] is White) then
                dist[v] ← dist[u] + 1
                pred[v] ← u
                color[v] ← Gray
                enqueue (Q, v)
        color(u) ← Black
```

```
BFS(G)      //does BFS visiting everyone
    Mark all nodes as unvisited
    for every vertex s of G not explored yet
            do BFS(G,s)
```

Breadth First Search: Analysis

Analysis O(n + m):

- Initialization part costs in total O(n)
- The cost of each execution of the blue block is at most O(deg(u) + 1)
- All red for's together are just sweeping through all layers => one iteration per node
- Total cost for all executions of blue block is ~**$\Sigma_{u \in V}$ (degree(u) + 1) = 2m + n**
- Total: O(n + m)

```
BFS(G, s)
    foreach v ∈ V do
        dist[v] ← ∞; pred[v] ← -1
        color(v) ← White
    dist(s) ← 0; color[s] ← Gray
    Q ← empty Queue; enqueue (Q, s)
    while (Q is not empty) do
        u ← dequeue (Q)
        foreach v ← Adj(u) do
            if (color[v] is White) then
                dist[v] ← dist[u] + 1
                pred[v] ← u
                color[v] ← Gray
                enqueue (Q, v)
        color(u) ← Black
```

```
BFS(G)    //does BFS visiting everyone
    Mark all nodes as unvisited
    for every vertex s of G not explored yet
        do BFS(G,s)
```

Breadth First Search: Applications

Application 1: Finding if there is a path from node s to node t

- Just run BFS(G, s); if there is path from s to t, this BFS visits t, otherwise it does not

Application 2: Length of the shortest path from s to t

- It's the level dist[.] computed by BFS(G, s) to which t belongs (if there is a path from s to t)

Application: Connected Component

Aplication: Return all the connected components

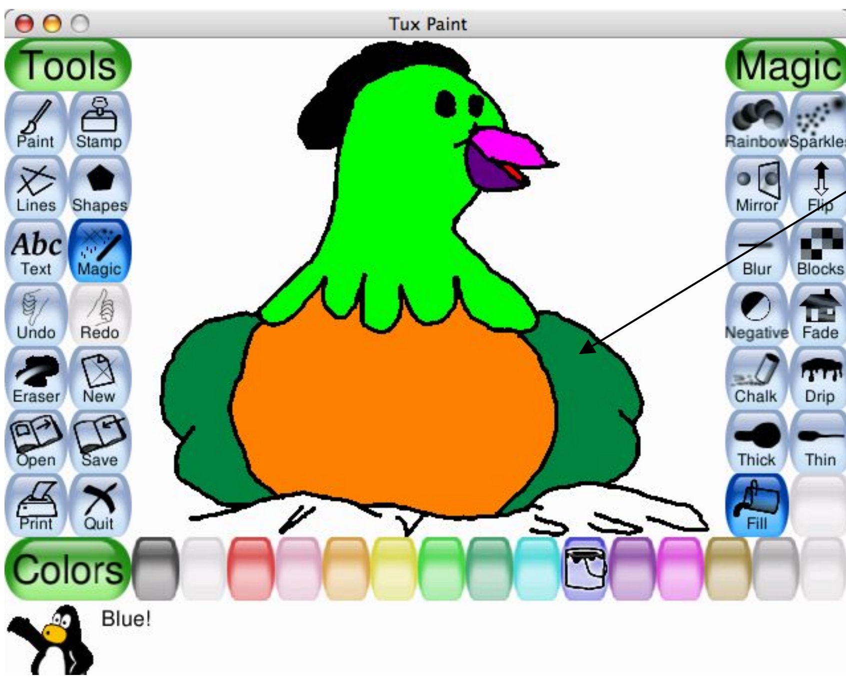
Since BFS(G,s) visits exactly the nodes in the connected component containing s, we can just return a Queue of nodes L to get the connected comp. of s

```
BFS(G, s)
    foreach v ∈ V do
        dist[v] ← ∞; pred[v] ← -1
        color(v) ← White
    dist(s) ← 0; color[s] ← Gray
    Q ← empty Queue; enqueue (Q, s)
    L ← empty Queue;
    while (Q is not empty) do
        u ← dequeue (Q)
        enqueue(L, u)
        foreach v ← Adj(u) do
            if (color[v] is White) then
                dist[v] ← dist[u] + 1
                pred[v] ← u
                color[v] ← Gray
                enqueue (Q, v)
        color(u) ← Black
```

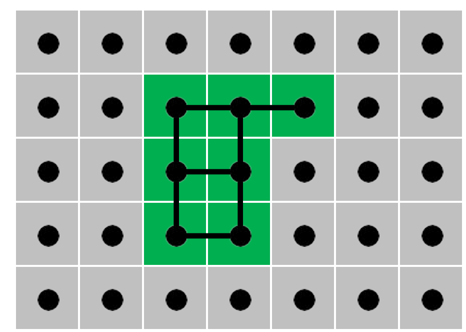
Application: Flood Fill

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

- **Node:** pixel.
- **Edge:** connects two neighboring pixels with same color
- **Blob:** connected component of green pixels



recolor this green blob to blue

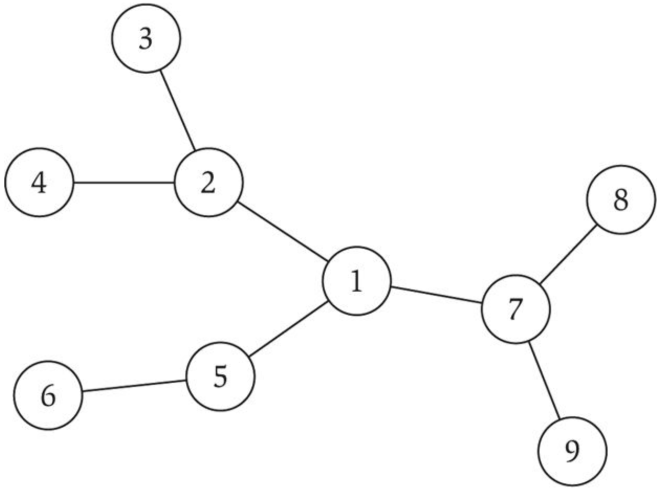


Trees

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.

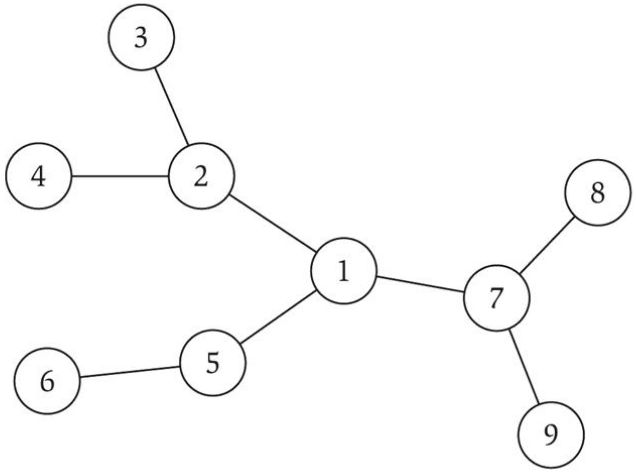
- G is connected.
- G does not contain a cycle.
- G has n-1 edges.



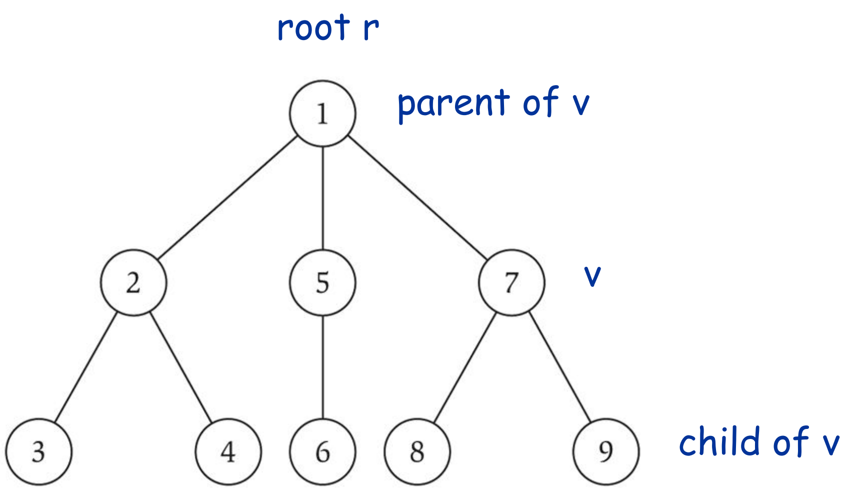
Rooted Trees

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

Importance. Models hierarchical structure.



a tree



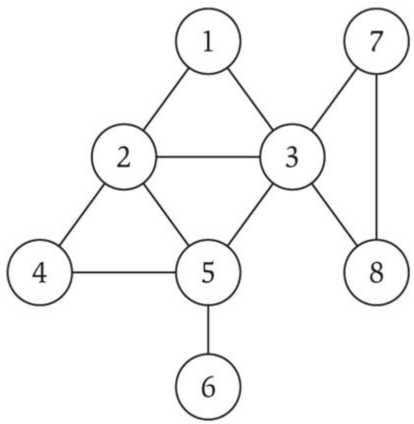
the same tree, rooted at 1

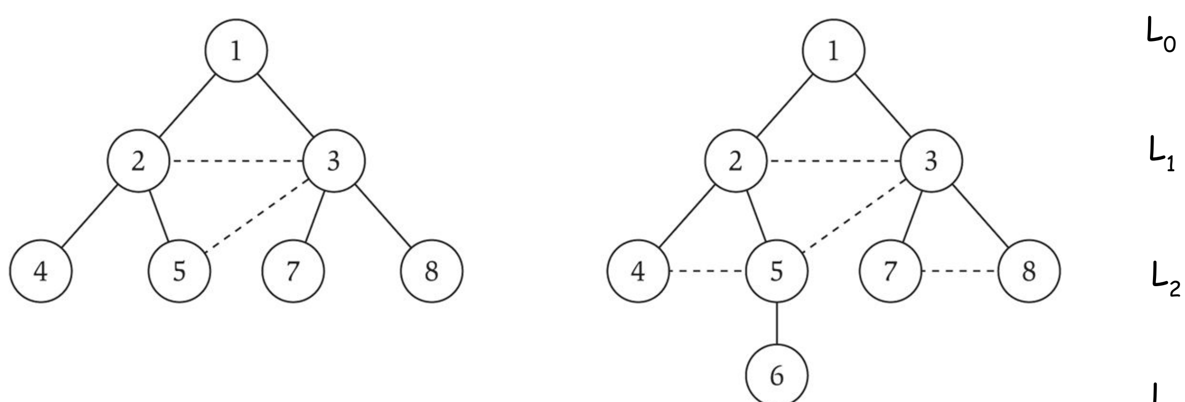
Breadth First Search: BFS tree

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 starting 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 (i.e., u is in the green for)

Ex: BFS(G,1)





Breadth First Search: BFS tree

Our BFS algorithm (implicitly) finds a BFS tree: the variable **parent[v]** indicates the **parent** of node **v** in the **BFS** tree

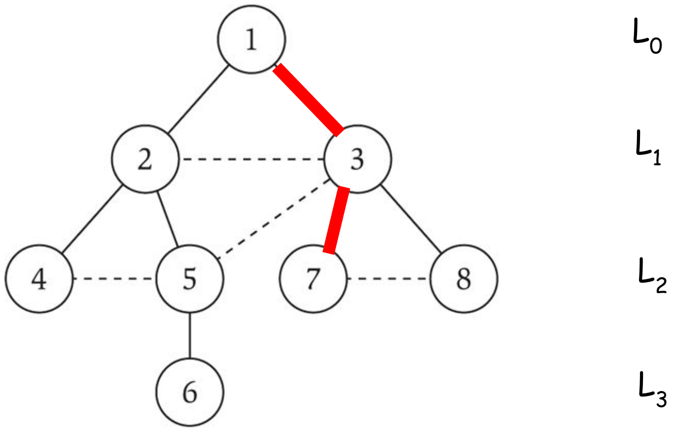
Observation: For the same graph **there can be different BFS trees**. The BFS tree topology depends on the starting point of the BFS and the order in which we scan the nodes at the same level

Breadth First Search: BFS tree

Q: How do we get the shortest path from s to t using BFS(G,s) ?

A: Run BFS(G,s) and follow the path in the BFS tree from ***s*** to ***t***
(or better, start at t and follow to its parent, and then its parent,... until reach s, getting the reverse shortest path from s to t)

Shortest path from 1 to 7



Breadth First Search

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

Proof: Cannot be that level(y) > level(x) + 1: when exploring x, either:

- y has been visited by someone at level <= level(x), so y it put at level <= level(x) + 1
- y has not been visited yet, so x himself add y to level(x) + 1

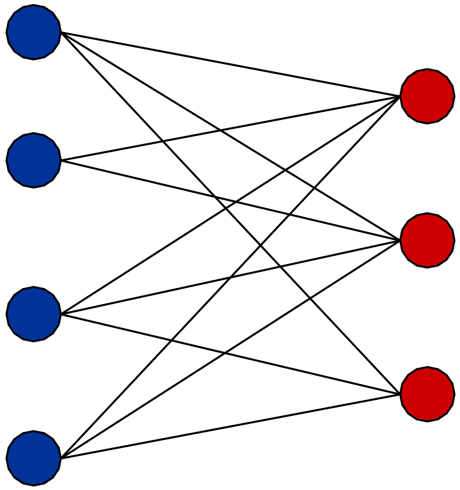
Another application: Testing Bipartiteness

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.

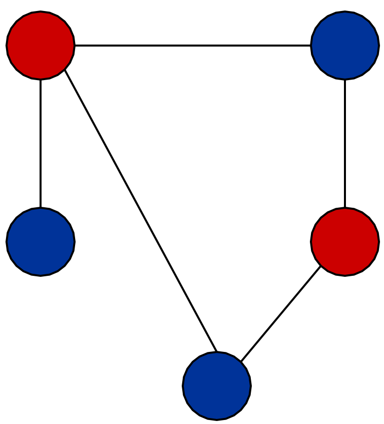


a bipartite graph

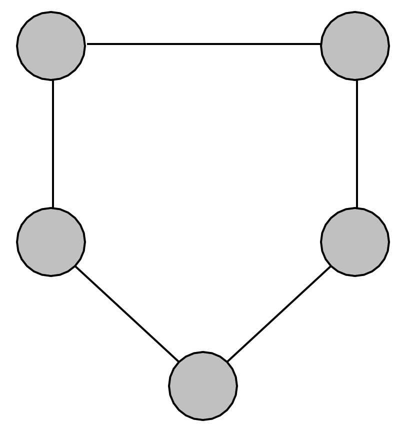
An Obstruction to Bipartiteness

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)



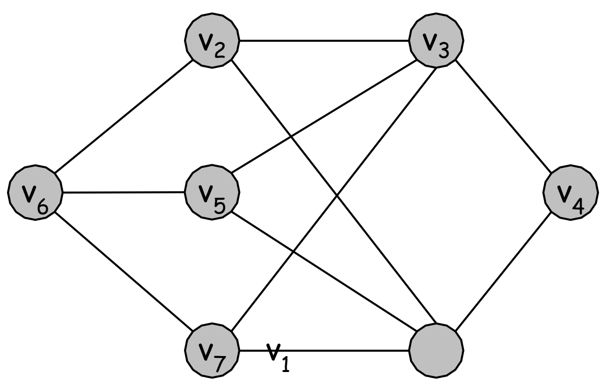
not bipartite
(not 2-colorable)

Testing Bipartiteness

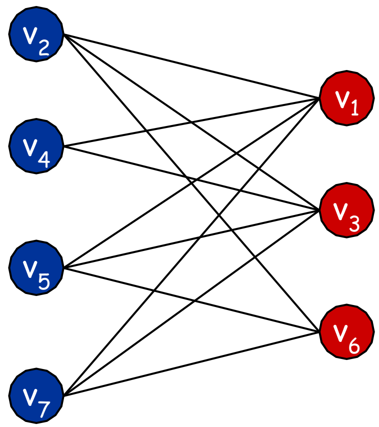
Testing bipartiteness. Given a graph G, is it bipartite?

- Many graph problems become:
  - easier if the underlying graph is bipartite (matching)
  - tractable if the underlying graph is bipartite (independent set)

- So if we detect our graph is bipartite, we may be able to use better algorithms



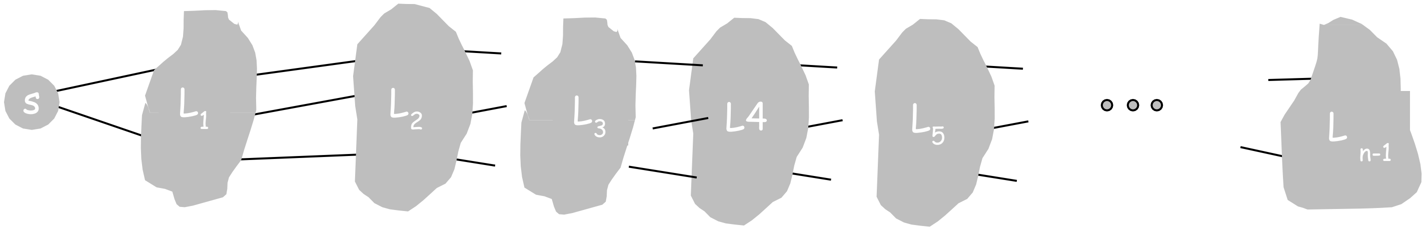
a bipartite graph G



another drawing of G

Testing Bipartiteness

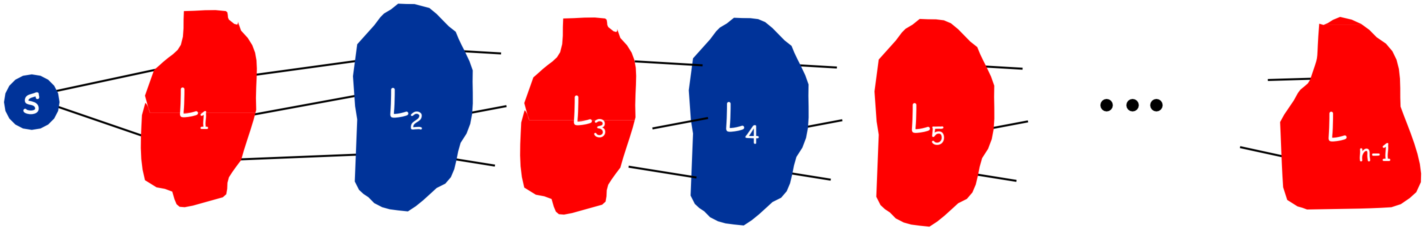
Q: Can we use BFS to test if a graph is bipartite/try to color it?



Testing Bipartiteness

Q: Can we use BFS to test if a graph is bipartite/try to color it?

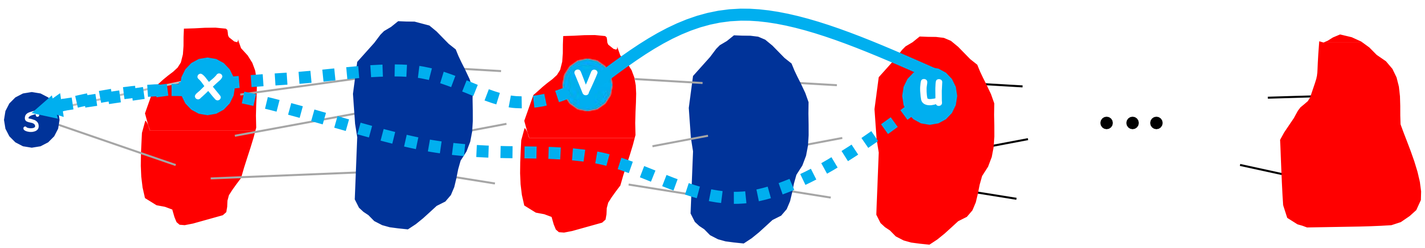
Idea: Color the levels of a BFS(G,s) tree with alternate colors



If there are no edges of G between blue/blue or red/red: done, bipartite

Testing Bipartiteness

Q: Can we use BFS to test if a graph is bipartite/try to color it?
Idea: Color the levels of a BFS(G,s) tree with alternate colors



If there are no edges of G between blue/blue or red/red: done, **bipartite**
If there is an edge of G betweeen **blue/blue** or **red/red**: **odd cycle**

- Suppose this edge is between nodes u and v
- Walk back from u and from v in the BFS tree; at some point you reach a common node x (it can be the root s)
- The cycle u—x—v–u is odd:
  - Since u and v have the same color, the length of segments u—x and x—v have the same parity (either both odd or both even)
- So ̀in this case the graph is not bipartite

Bipartite Graphs

We have just proved the following

Lemma. Let G be a connected graph, and let L_0, ..., L_k be the layers produced by BFS(G,s). If we color the layers alternately blue and red, exactly one of the following holds:

(i) There is no blue/blue or red/red edge, and so G is bipartite
(ii) There is a blue/blue or red/red edge, and G contains an odd-length cycle (and hence is not bipartite).

So the only way we cannot color the graph is if it has an odd cycle

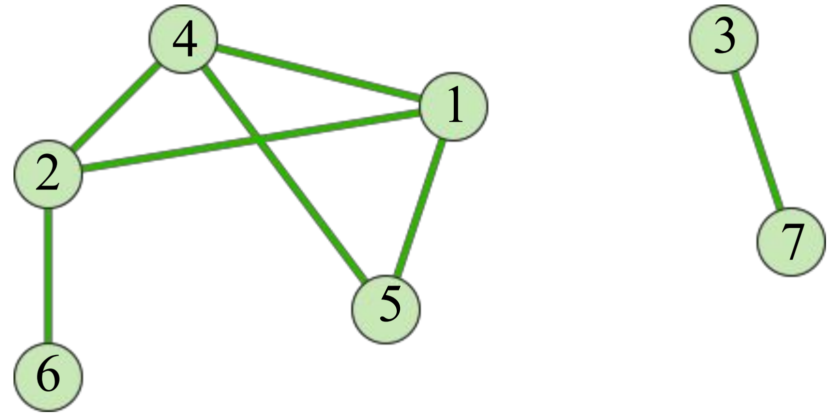
Corollary. (Konig 1916) A graph G is bipartite if and only if it contains no odd length cycle.

Depth first search

Depth First Search (DFS)

DFS(*G*)
For *v* in *G*
    If *v* not visited **then**
        DFS-Visit(*G*, v)

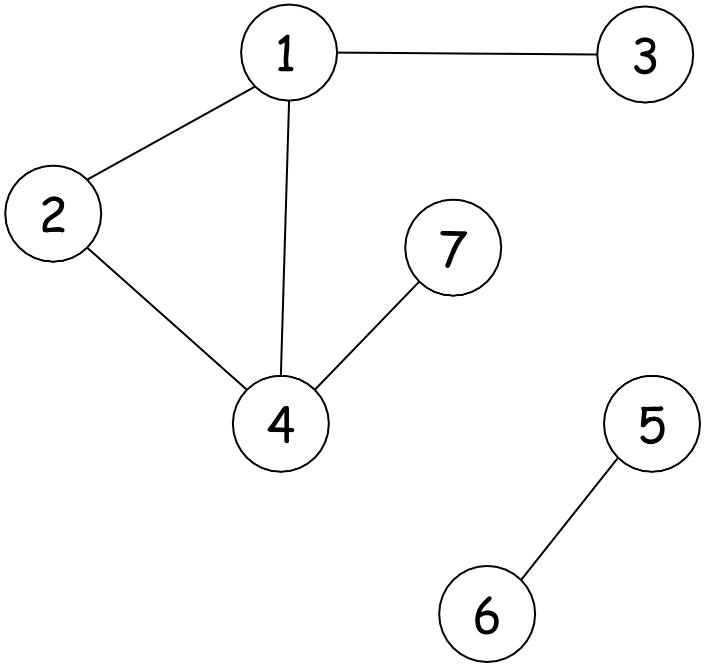
DFS-Visit(*G*, u)
Mark *u* as visited
For *v* in Adj(u)
    If *v* not visited **then**
        Insert edge (u, v) in DFS tree
        DFS-Visit(*G*, v)



DEMO DFS

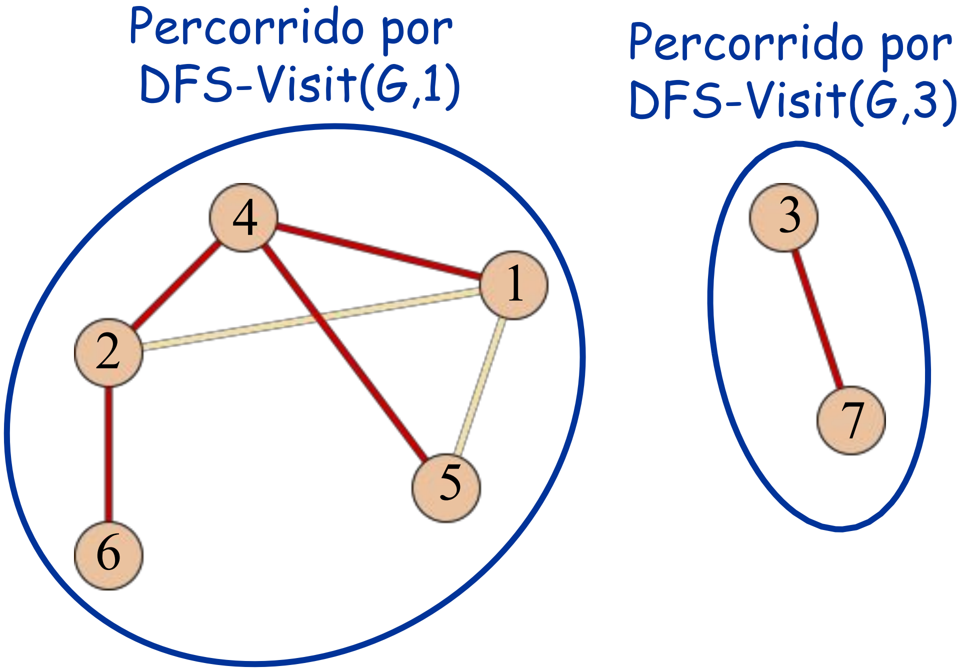
Depth First Search (DFS)

Exercise: Run DFS for the following graph



Depth First Search

Assim como na busca em largura, DFS–Visit(G, u) visita apenas o componente conexo contendo o no de inicio u



Depth First Search: Analysis

DFS-Visit(G,u) tem complexidade

O(#nos no comp. conexo de u + #arestas no comp. conexo de u)

Justificativa:

- O número de blocos na arvore de recursao é exatamente o #nos no comp. conexo de u, pois cada no e visitado uma unica vez
- O custo de cada bloco da arvore de recursao (sem contar as chamadas recursivas) é ~(1 + número de vizinhos do nó associado):
  - Checa pra cada vizinho se ja foi visitado
- Somando o custo de todos os blocos, temos

  ~ #nos no comp. conex. de u + $_{v : v \text{ em comp conexo}}\ \deg(v)$

  ~ #nos no comp. conex. de u + 2#arestas no comp conexo

Depth First Search: Analysis

A busca completa DFS(G) tem complexidade O(n + m)

Justificativa: Lança uma busca por componente conexo. Somando o custo de cada uma dessas buscas, obtemos o resultado:

custo = O((#nos no comp. conexo + #arestas no comp. conexo))
comp conexo
= O(#nos grafo + #arestas grafo)

DFS(*G*)

For *v* in *G*
  If *v* not visited **then**
    DFS-Visit(*G*, u)

DFS-Visit(*G*, u)

Mark *u* as visited
For *v* in Adj(u)
  If *v* not visited **then**
    ……

Resumo: DFS(G) tem complexidade O(n + m)

Depth First Search

Just like for BFS, we have a DFS tree

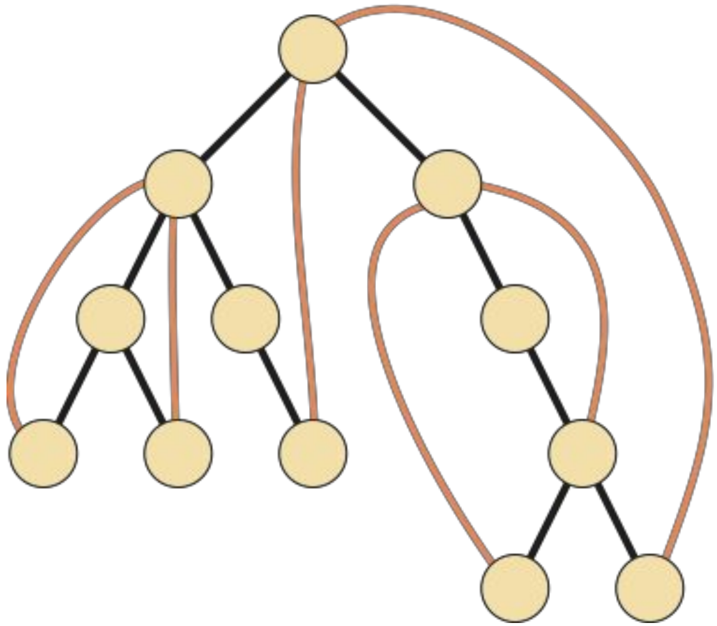
Definition A DFS tree of G = (V, E), is the tree induced by a DFS search on G.

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

Exactly the recursion tree of the algorithm

Properties of DFS

Theorem: Consider a graph G and let T be a DFS tree. Then for any edge vw of G, if v is visited before w then v is an ancestor of w in T

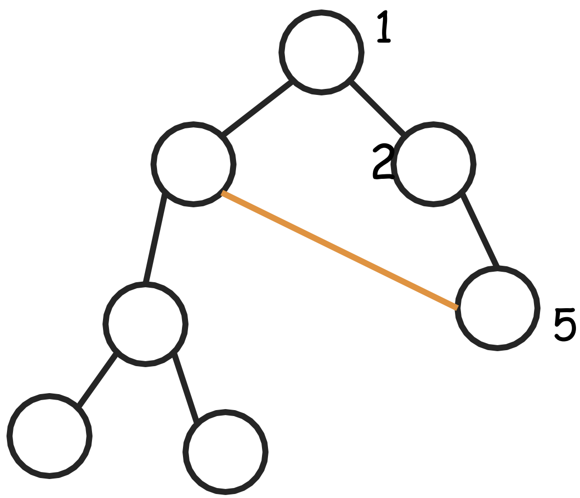


Edges in black: DFS tree
Edges in orange: other graph edges

Properties of DFS

Theorem: Consider a graph G and let T be a DFS tree. Then for any edge vw of G, if v is visited before w then v is an ancestor of w in T

Ex: We cannot "crossing edges" like in the following situation (numbers indicate order in which nodes are visited)



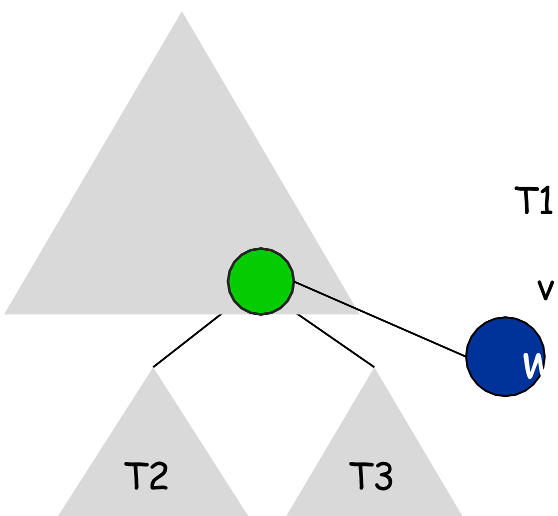
Properties of DFS

Theorem: Consider a graph G and let T be a DFS tree. Then for any edge vw of G, if **v** is visited before **w** then **v** is an ancestor of **w** in T

Proof: Consider the exploration of **v**

- Before stareted exploring **v**, did not visit **w** (so w not in T1)
- Then explored some neighbors of **v** (visiting T2 and T3)
- Now **v** tries to explore neighbor **w**
    - If **w** has not been explored, then **v** is the parent of **w**
    - If **w** has been explored, it must be in T2 or T3, **v** is an ancestor of **w**

(recall **w** not in T1)



Properties of DFS

Theorem: Consider a graph G and let T be a DFS tree. Then for any edge vw of G, if v is visited before w then v is an ancestor of w in T

[Write this on the board, we'll use in the next application]

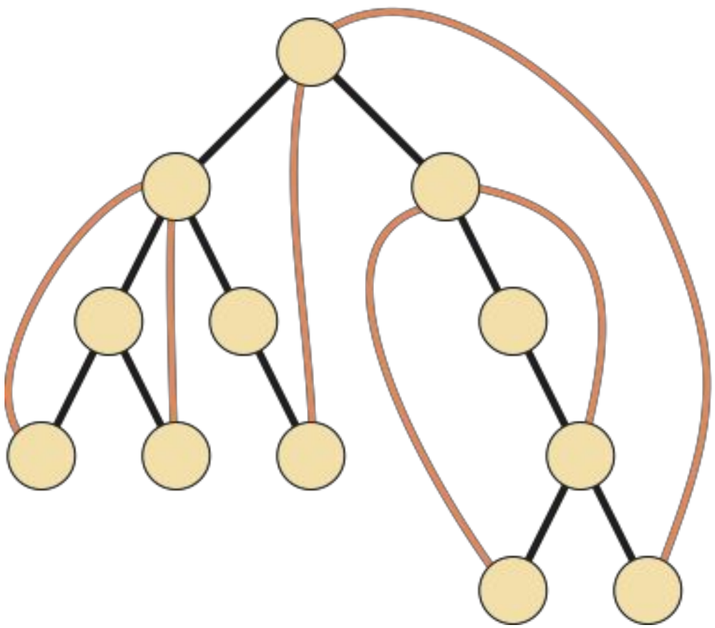
Obs: This is not true for BFS

Exercise: Construct a graph that shows this

Application of DFS: Finding cycles

Q: How can we use DFS to find a cycle in the graph?

A: If tries to revisit nodes in DFS => cycle (only exclude case where trying to revisit parent)

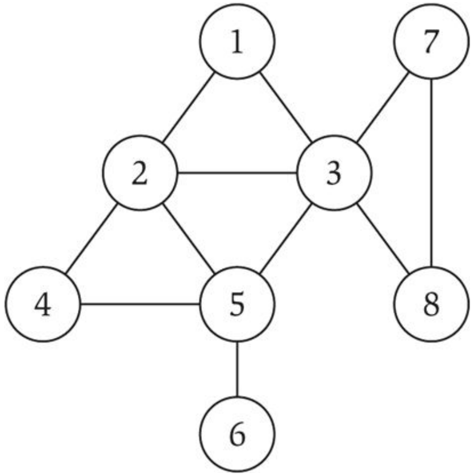


```
DFS(G)
    Para todo v em G
        Se v não visitado então
            DFS-Visit(G, v)

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)
        Senao
            Se w<>pai(v)
                Return Existe Ciclo
        Fim Se
    Fim Para
```

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

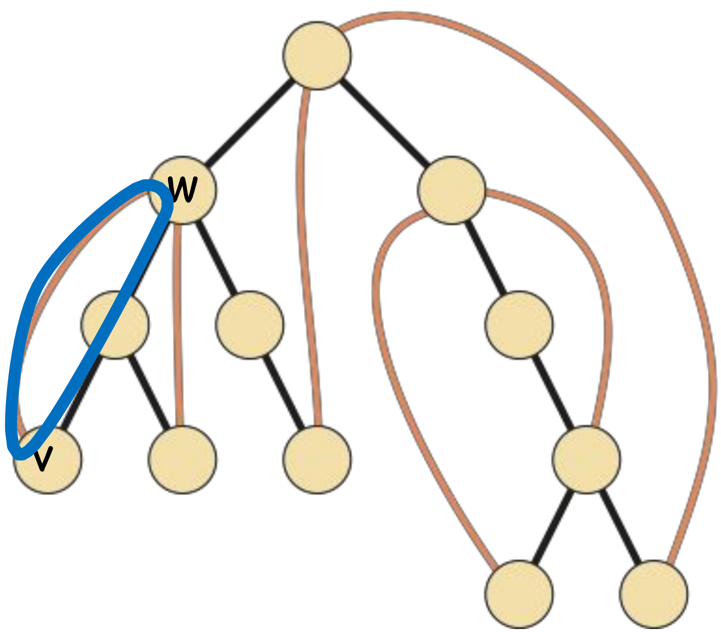
Application of DFS: Finding cycles

Need to show it actually works

Claim 1: If returned "Existe ciclo", then there is a cycle in the graph

Proof: If **w** was already visited and is a neighbor of **v**, then an ancestor of n DFS tree

If **w** is not the parent of v in the tree, have cycle w ---- v - w




```
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)
        Senao
            Se w<>pai(v)
                Return Existe Ciclo
        Fim Se Fim Para
```

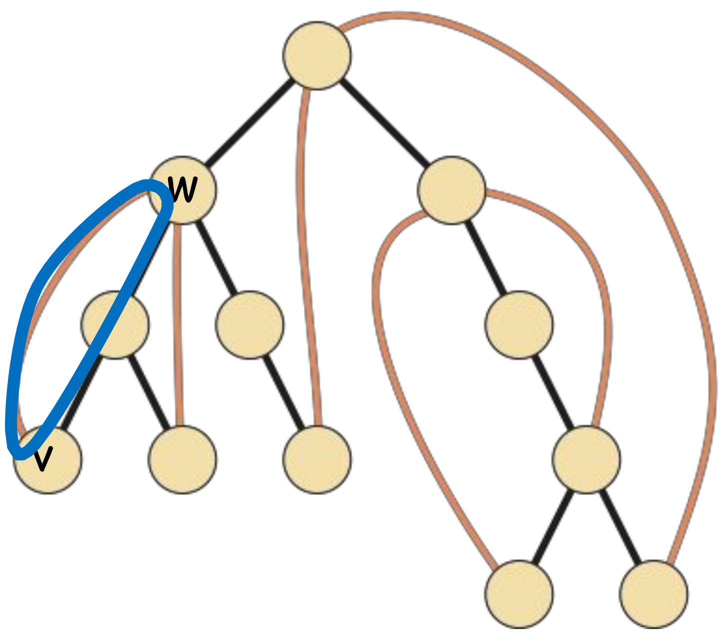
Application of DFS: Finding cycles

Need to show it actually works

Claim 1: If returned "Existe ciclo", then there is a cycle in the graph

Proof: If **w** was already visited and is a neighbor of **v**, then **w** is an ancestor
of **v** in DFS tree

If w is not the parent of v in the tree,




```
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)
        Senao
            Se w<>pai(v)
                Return Existe Ciclo
        Fim Se
    Fim Para
```

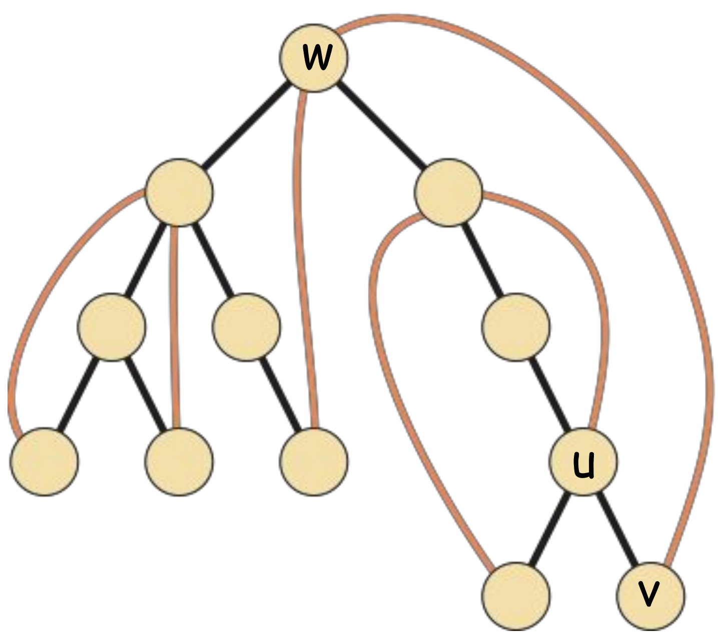
Application of DFS: Finding cycles

Claim 2: If there is cycle in the graph, algo returns "Existe

ciclo" **Proof:** Let **v** be the last vertex of the cycle visited by the

DFS So both neighbors of **v** in the cycle are of **v**

At least one of them is not the parent of **v** => DFS returns "Existe ciclo"



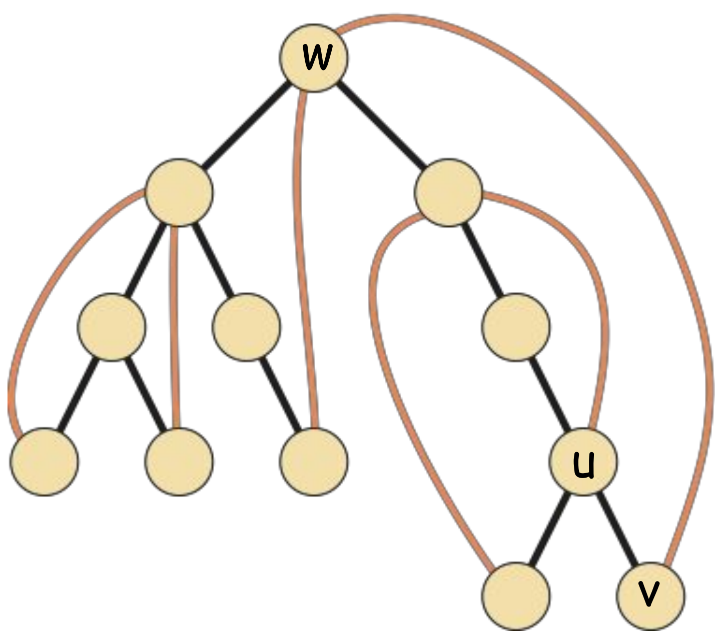

```
DFS(G)
    Para todo v em G
        Se v não visitado então
            DFS-Visit(G, v)

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)
        Senao
            Se w<>pai(v)
                Return Existe Ciclo
        Fim Se
    Fim Para
```

Application of DFS: Finding cycles

Claim 2: If there is cycle in the graph, algo returns "Existe ciclo" **Proof:** Let **v** be the last vertex of the cycle visited by the DFS So both neighbors of **v** in the cycle are ancestors of **v**

At least one of them is not the parent of **v** => DFS returns "Existe ciclo"




```
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)
        Senao
            Se w<>pai(v)
                Return Existe Ciclo
        Fim Se Fim Para
```

Exercise
s

Exercise 1:

(1 pts) Dado um grafo **direcionado** e uma de suas arestas (u, v), encontre o menor **ciclo** no grafo contendo essa aresta, onde o tamanho do ciclo é o número de arestas que possui. Seu algoritmo deve ter complexidade $O(n + m)$

Exercise 2: Can we use **BFS** to detect cycles in undirected graphs? How?

Exercícios de Implementação

Exercicio 1: 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

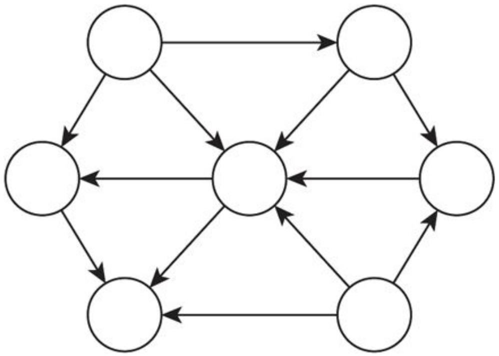
(ou seja, voce esta reconhecendo os componentes conexos do grafo)

Exercicio 2: 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.

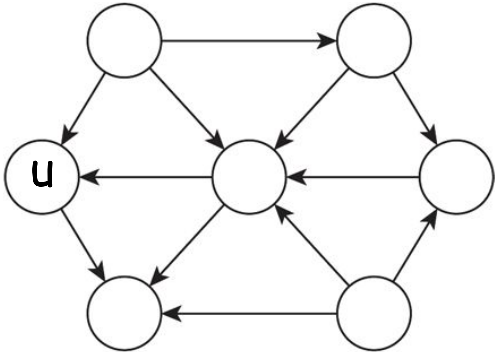


Ex. Web graph - hyperlink points from one web page to another.
- Directedness of graph is crucial.
- Modern web search engines exploit hyperlink structure to rank web pages by importance.

Directed Graphs

- The in-degree $d^-(u)$ of a vertex u is the number of edges that arrive at u

- The out-degree $d^+(u)$ of a vertex u is the number of edges that leave u



$d^-(u)=2$

$d^+(u)=1$

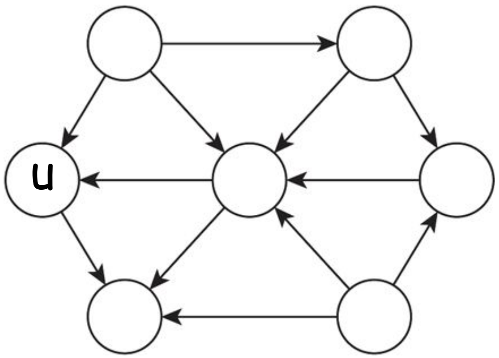
Important property:

sum of indegrees	sum of outdegre

Directed Graphs

- The in-degree $d^-(u)$ of a vertex u is the number of edges that arrive at u

- The out-degree $d^+(u)$ of a vertex u is the number of edges that leave u



$d^-(u)=2$

$d^+(u)=1$

Important property:

sum of indegrees = sum of outdegre = m

Representation via Adjacency List

Undirected Graph

Directed Graph

Graph Search

Directed reachability. Given a node s, find all nodes reachable from s. (need to use arcs in the right direction)

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 and DFS extend naturally to directed graphs.

Exercise: Check that you know how to do BFS and DFS in directed graphs!

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

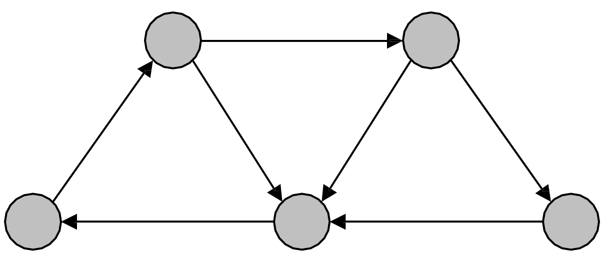
Strong Connectivity

Def. A graph is strongly connected if for every pair of nodes **u**,**v** there is a path from **u to v** and from **v to u**

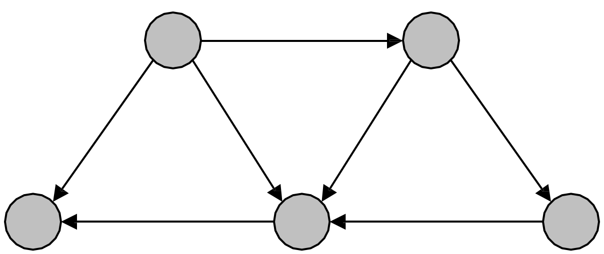
How to decide whether a given graph is strongly connected?

Q: Applications?

A: Road/bus connectivity: no one gets stuck
User interface: make sure user can navigate to/from everywhere



strongly connected



not strongly connected

Strong Connectivity

Q: Give a simple algorithm to decide where a graph is strongly connected or not

Algorithm 1

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

Analysis:

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

Strong Connectivity

Q: Can we do better?

A: Can use 1 search to check if everyone is reachable from u

Algorithm 2

SC ← true

For all u in V

Run DFS(u)

If the search does not visit all nodes SC

← False

End If

End Return SC

Analysis:

O(n (m+n))

Strong Connectivity

Q: Even better??

Lemma. Consider a node s. G is strongly connected ⇔ every node is reachable from s, and s is reachable from every node.

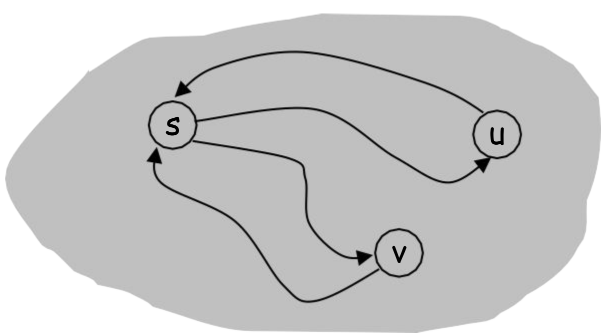
Pf. ⇒ Follows from definition.

Pf. ⇐ Can go from any node u to v (in both directions):

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



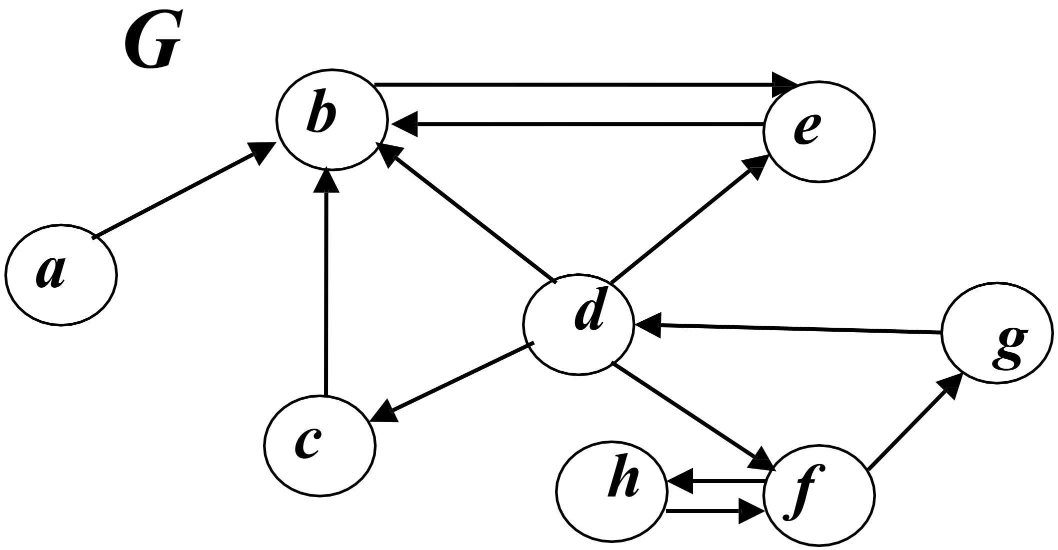
Strong Connectivity

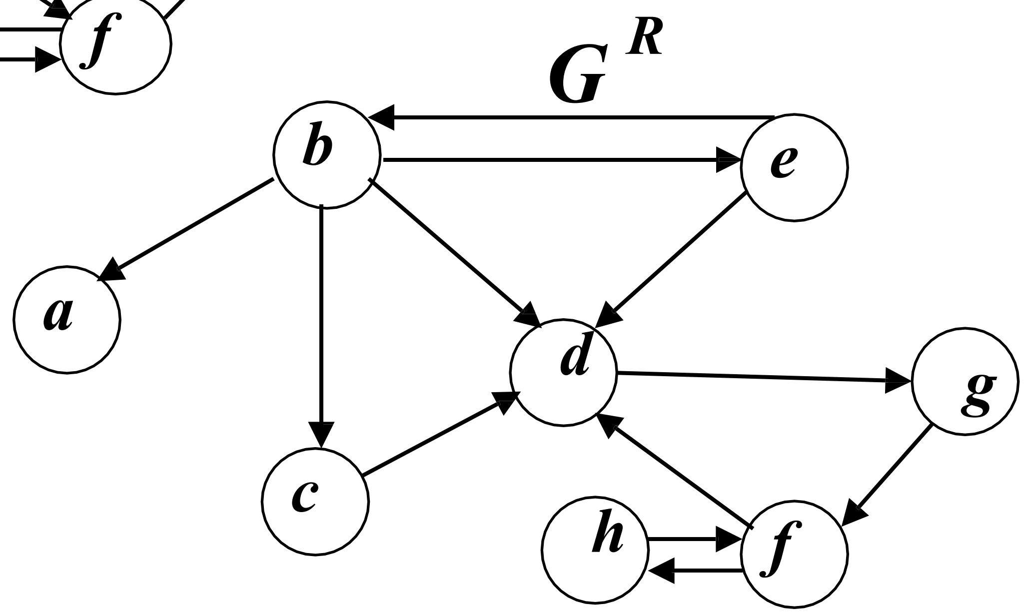
Def. The reverse graph of a graph G is obtained by reversing the directions of all the edges

Observation: The reverse graph of a graph G can be constructed in O(m+n) time

Strong Connectivity

Example:

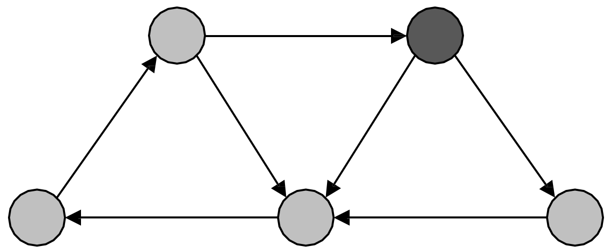




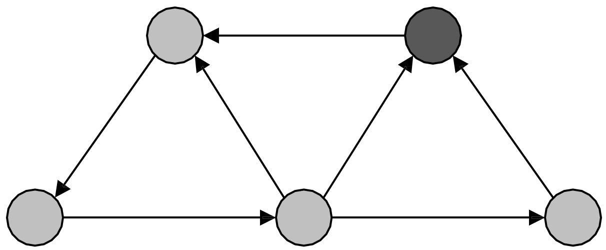
Strong Connectivity: Algorithm

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

- Pick any node s.
- (s reaches everyone?) Run BFS/DFS from **s** in G.
- (everyone reaches s?) Run BFS/DFS from **s** in reverse graph G^R.
- Return true iff all nodes reached in both BFS/DFS executions.
- Correctness follows immediately from previous lemma. ▪



Graph G



Reverse graph G^R

Using graphs to model state space

Problema

- Seja um grafo G=(V,E) com n vértices representando a planta de um edifício. Inicialmente temos dois robos localizados em dois vértices *a* e *b*, que devem alcançar os vértices *c* e *d* respectivamente. Queremos manter sempre uma distancia de seguranca r entre eles.

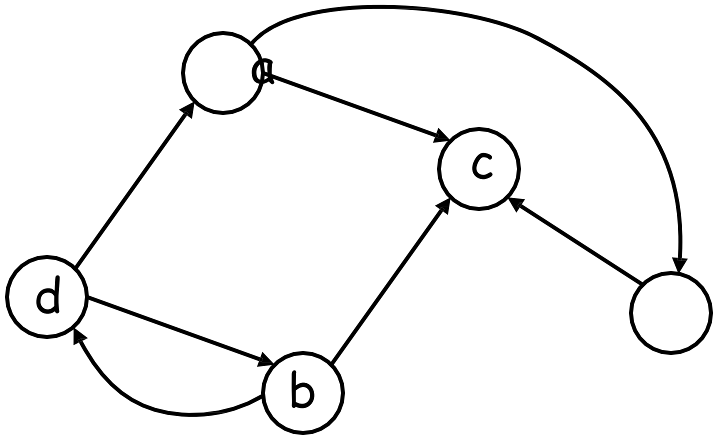
- No passo i+1 um dos dois robos deve caminhar para um vértice adjacente ao vértice que ele se encontra no momento i. Exiba um algoritmo polinomial para resolver o seguinte problema:

- **Entrada:** Grafo G=(V,E) , quatro vértices: *a*,*b*,*c* e *d* e um inteiro r.

- **Saída: SIM** se é possível os robos partirem dos vértices *a* e *b* e chegarem em *c* e *d*, respectivamente, sem que em nenhum momento eles estejam a distância menor do que r. **NÃO**, caso contrário.

Example graph

r = 2



Solução

Seja H=(V',E') um grafo representando as configurações possíveis (posições dos robos) do problema. Cada nó de H corresponde a um par ordenado de vértices do grafo original G cuja distância é menor ou igual a r. Logo existem no máximo $|V|^2$ vértices em H.

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. Mais formalmente, se uv é uma aresta de E', com u=(u1,u2) e v=(v1,v2), então uma das alternativas é válida

(i) u1=v1 e (u2,v2) pertence a E

(ii) u2=v2 e (u1,v1) pertence a E

O problema, portanto, consiste em decidir se existe um camìnho entre o nó x=(a,b) e o nó y=(c,d) em H.

Solução

Para construir o grafo H basta realizar **n** BFS's no grafo G, cada uma delas partindo de um vértice diferente. Ao realizar uma BFS a partir de um nó s obtemos o conjunto de todos os vértices que estão a distância maior ou igual a r de s. A obtenção do conjunto V' tem custo O(n(m+n)) e a do conjunto de arestas E' tem custo $O(n^3)$.

Decidir se existe um camìnho entre o nó x=(a,b) e o nó y=(c,d) em H tem complexidade O(|V'|+|E'|). Como |V'| tem $O(n^2)$ vértices e |E'| tem $O(n^3)$ arestas, o algoritmo executa em $O(n^3)$. Note que |E'| é $O(n^3)$ porque cada vértice de H tem no máximo 2(n-1) vizinhos

BFS/DFS exercises

Exercises:

1. Suppose your graph is an undirected tree. If run BFS starting from the root of the tree, in which order are the nodes explored? What about in DFS?

2. Using the BFS/DFS tree, show that every connected undirected graph has a node that can be removed keeping the graph still connected [show example]

3. Suppose your undirected graph has a value $x(v)$ for each node. Modify DFS to compute

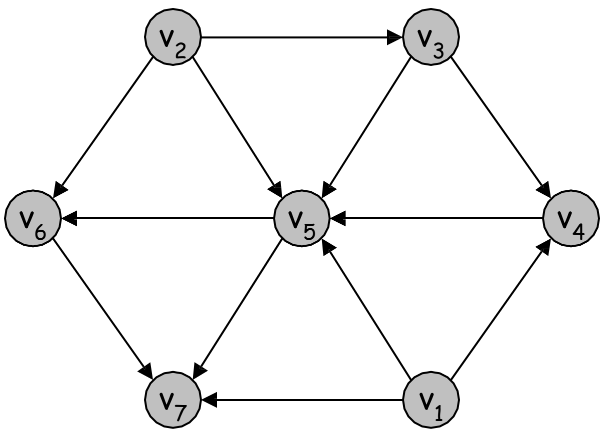
   $z(v)$=sum of values of all descendants of v in the DFS tree,

   for all nodes. The algorithm should still run in $O(n + m)$

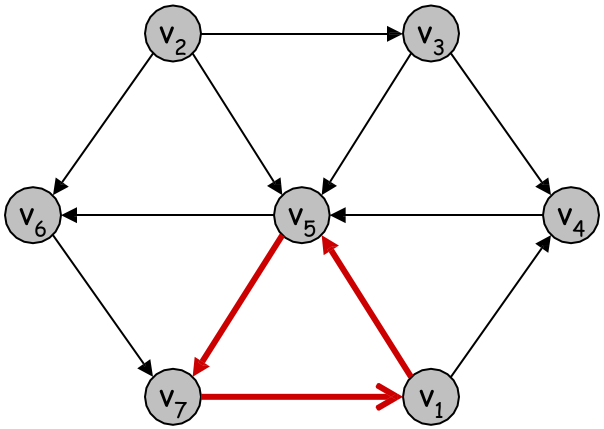
3.6 DAGs and Topological Ordering

Directed Acyclic Graphs

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



a DAG



Not a DAG

Precedence Constraints

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

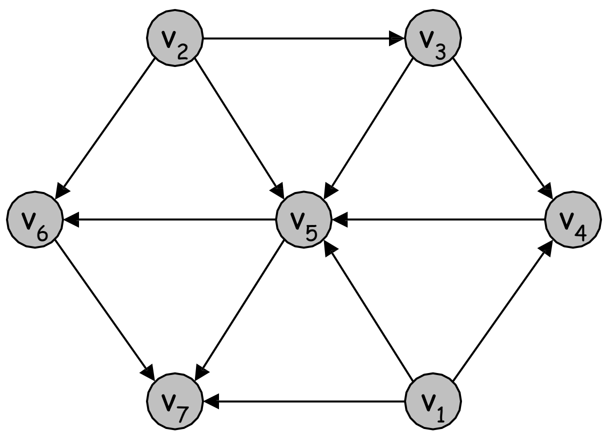
Applications.

- Course prerequisite graph: course v_i must be taken before v_j.
- Compilation: module v_i must be compiled before v_j.

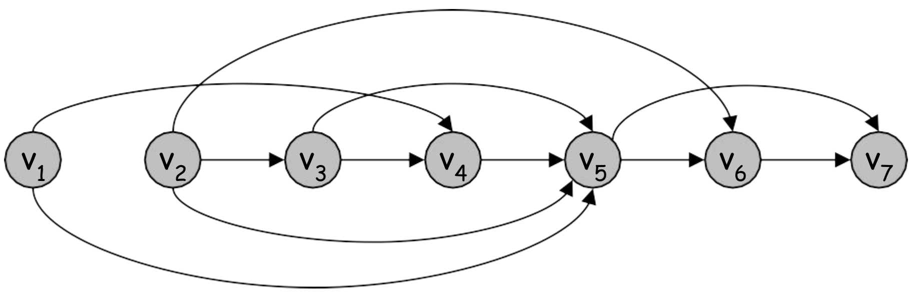
Q: What is a feasible sequence of courses?
What is a feasible order to compile the jobs?

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 $i < j$.



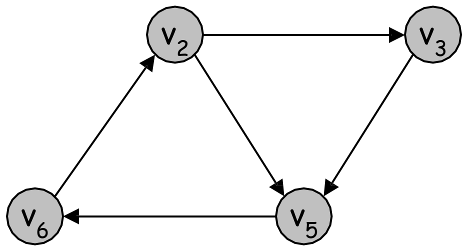
G



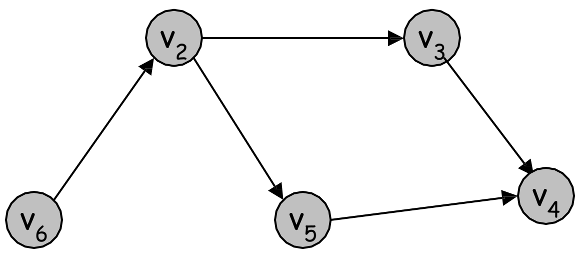
a topological ordering for G

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 $i < j$.



Has no topological order



Topological orders:

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

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

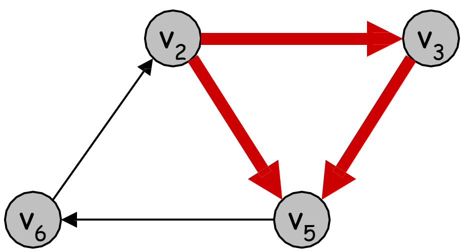
What is the relation between DAG's and topological orderings?

Directed Acyclic Graphs

Obs: Directed cycle does not have a topological order

Since we cannot topologically order a directed cycle, we cannot do it for any graph containing a directed cycle

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



Has no topological order

Q. Does every DAG have a topological ordering?

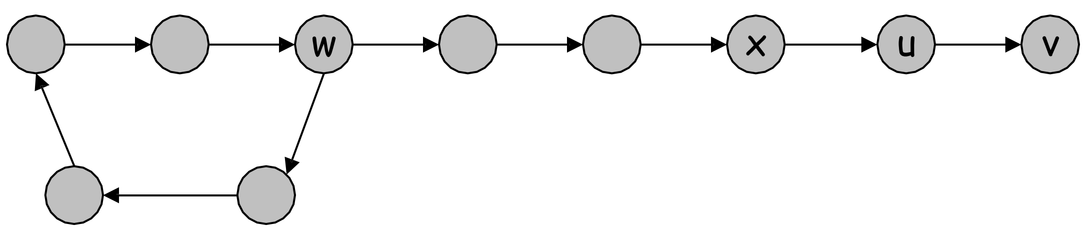
Q. If so, how do we compute one?

Directed Acyclic Graphs

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

Pf. (by contradiction)

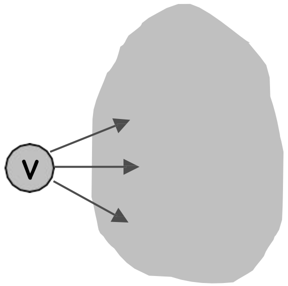
- Suppose that G is a DAG and every node has at least one incoming edge. Let's see what happens.
- Pick any node v, and begin following edges backward from v. Since v has at least one incoming edge (u, v) we can walk backward to u.
- Then, since u has at least one incoming edge (x, u), we can walk backward 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.

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

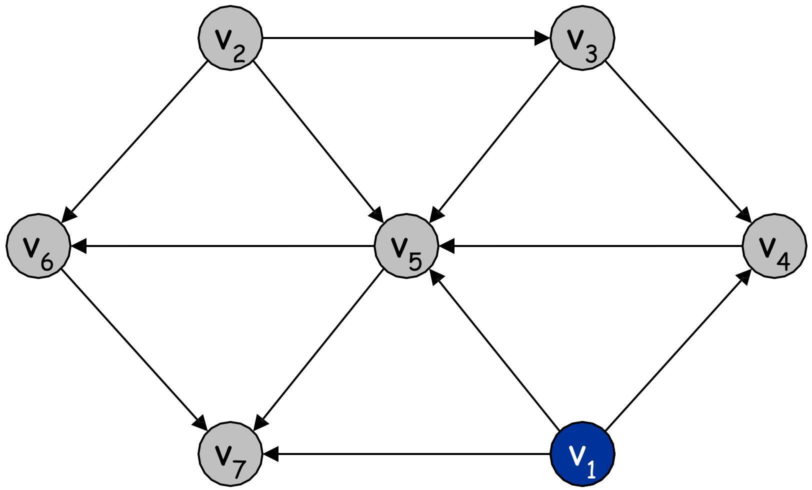


Proof that it works: (by induction on n)

- 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.

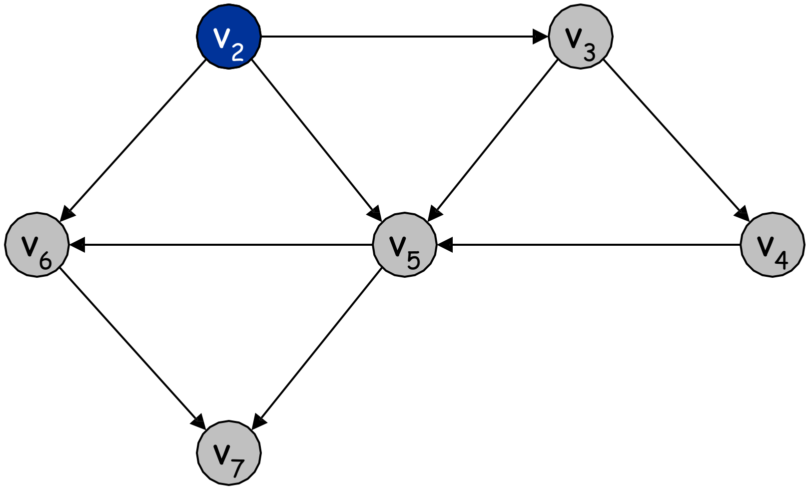
▪

Topological Ordering Algorithm: Example



Topological order:

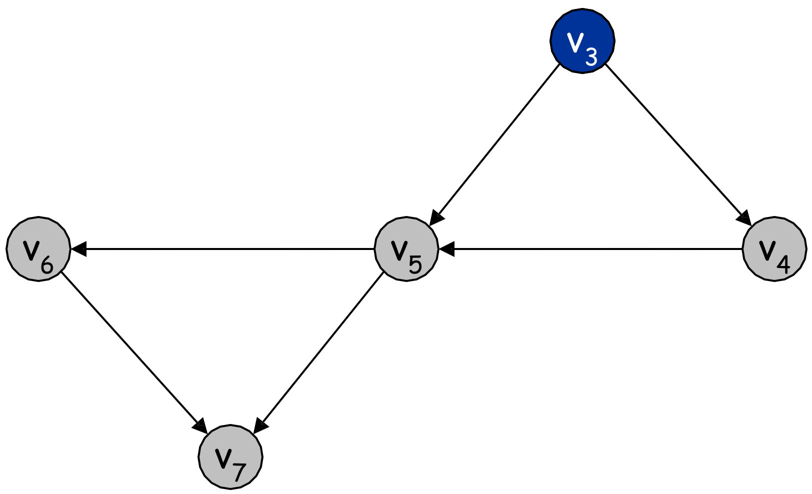
Topological Ordering Algorithm: Example



Topological order:

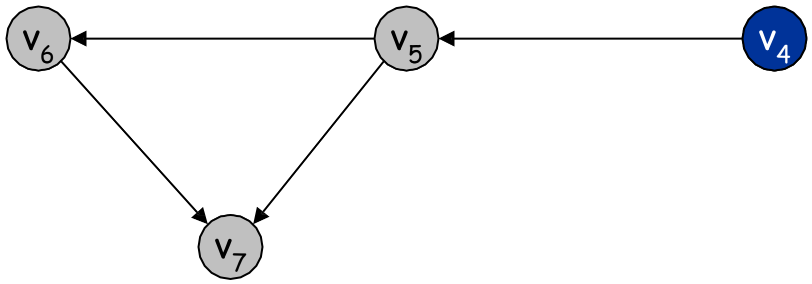
v_1

Topological Ordering Algorithm: Example



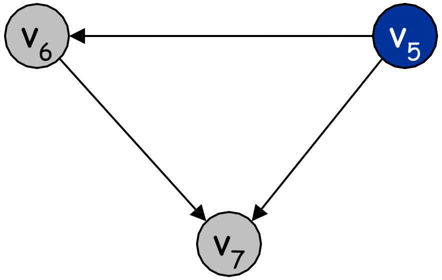
Topological order: v_1, v_2

Topological Ordering Algorithm: Example



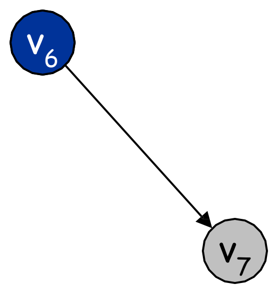
Topological order: v_1, v_2, v_3

Topological Ordering Algorithm: Example



Topological order: v_1, v_2, v_3, v_4

Topological Ordering Algorithm: Example

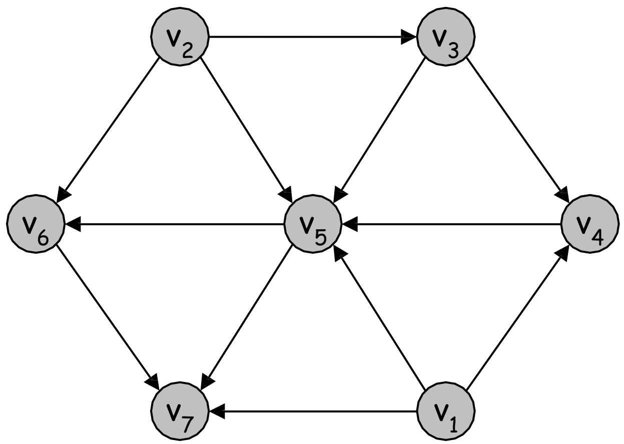


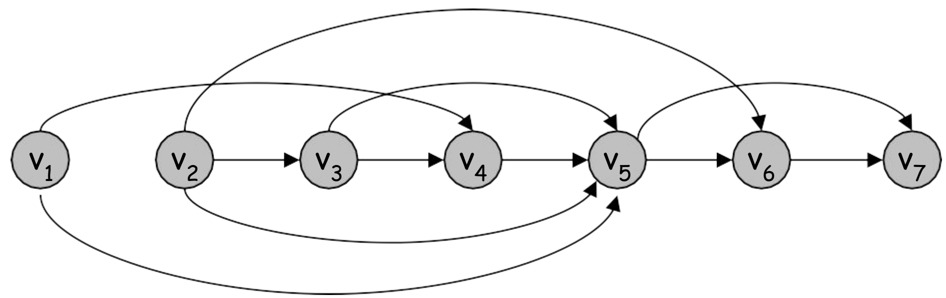
Topological order: v_1, v_2, v_3, v_4, v_5

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

Q: How to implement this algorithm with fast running time?

Implementation idea: keep a vector `count` that stores for each node v the number of remaining edges that are incident in v

Implementation 1:

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

`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 initialize `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. We can implement the algorithm to find 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 `count[w]` hits 0
  - this is O(1) per edge ▪

Detecting if a directed graph is DAG

Q: How can we detect if a directed graph G has a directed cycle or not?

A: Try to run topological ordering algorithm on G. Works $\Leftrightarrow$ G does not have cycle

- G does not have cycle => works
- G does have a cycle => cannot work, since G does not have top. order

Q: Where does algorithm does not work if graph has cycle?

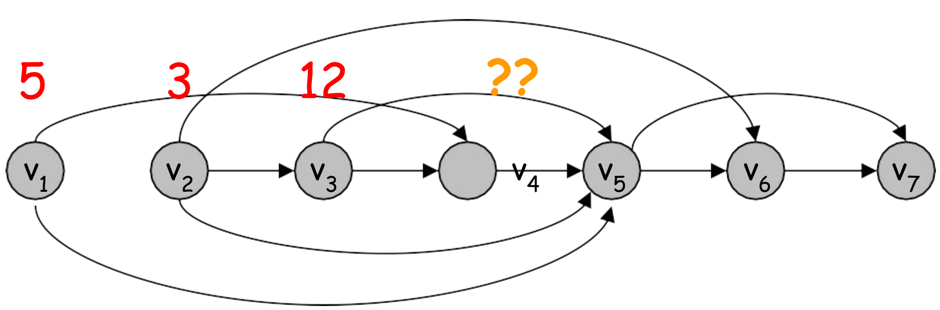
A: At some point it will not find a node with in-degree 0

Applications of topological order

Topological order can be used to solve many problem in DAGs very efficiently, especially when we have to assign numbers (or other information) to each node

The high-level idea is that it allows one to compute things "inductively" by traversing the nodes according to the topological order (or in reverse order)

Information of all anti-neighbors (or neighbors) have been computed, can use it to help



Applications of topological order

Problem 1: Suppose you have a DAG where each node has a price p(v). Let cost(u) be the smallest price of all nodes reachable from u. Use topological order to compute cost(u) for all nodes in the graph in O(n + m)

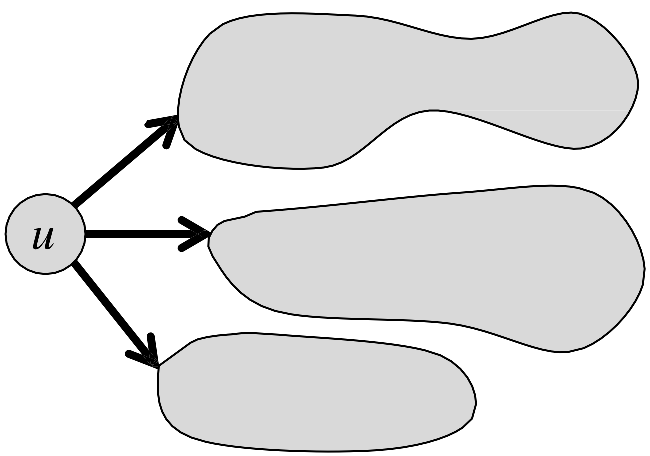
[give concrete example on the board]

Solution:

1) Do topological sort of the graph
2) Scan nodes in reverse order of the topological sorting
3) At node u, compute

Information about cost of neighbors is available when we need it

$$cost(u) = \min\left\{ p(u), \min_{v \text{ neighbor of } u} \{cost(v)\} \right\}$$



All nodes reachable from u can be reached from one of its neighbors (or is u itself) => pick the best option

Applications of topological order

Problem 2: Given a list of courses a student needs to take and the prerequisites between then, *give an algorithm* that finds the minimum number of semesters needed for the student to finish all the courses

[give concrete example on the board]

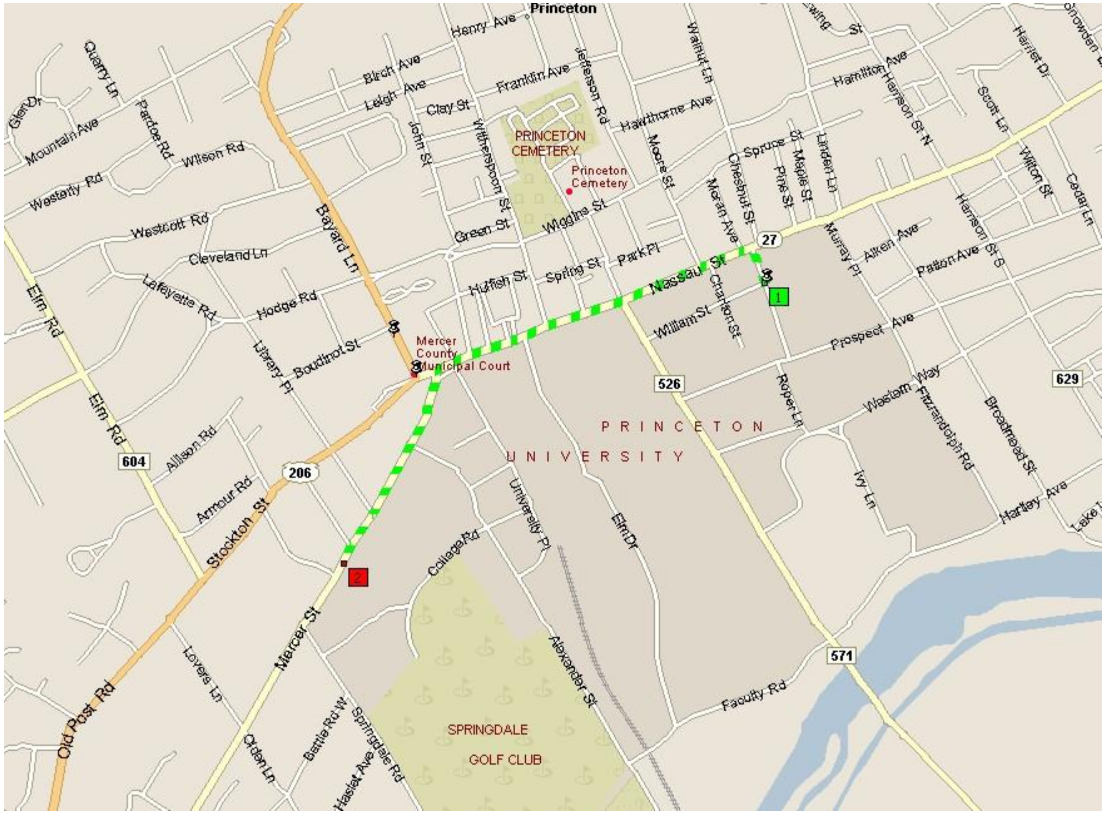
Possible solution: Compute for each node u the first semester f(u) that we can do that course:

1) Do topological sort of the graph
2) Scan nodes in the order of the topological sorting
3) At node u, compute

$$f(u) = 1 + \min_{v \text{ antineighbor of } u} \{f(v)\}$$

Time complexity is O(n + m) (construct reverse graph to obtain anti-nneighbors)

4.4 Weighted Shortest Paths in a Graph



shortest path from Princeton CS department to Einstein's house

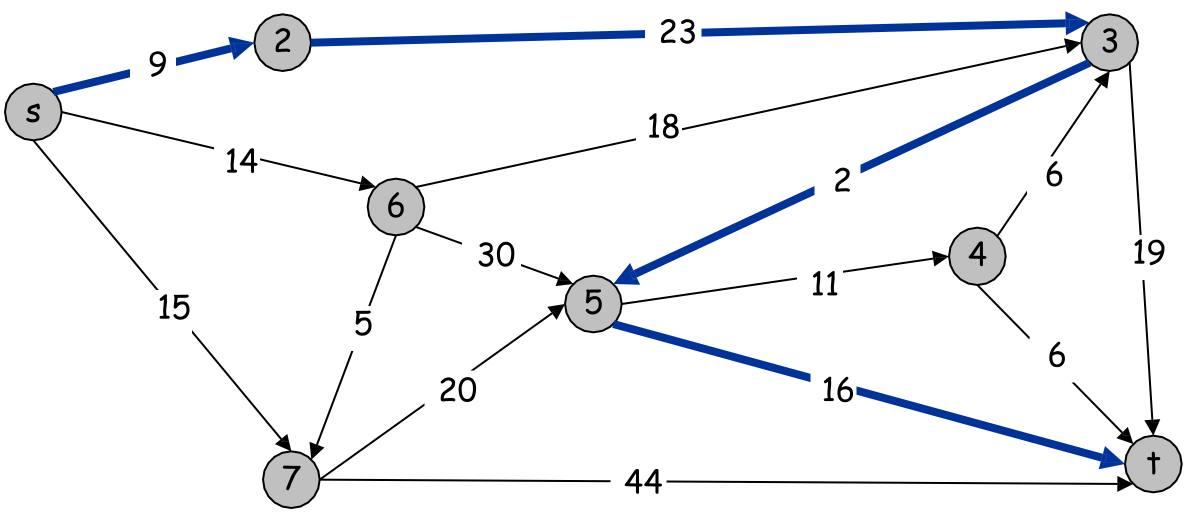
Shortest Path Problem

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.

Length of path = sum of lengths in path

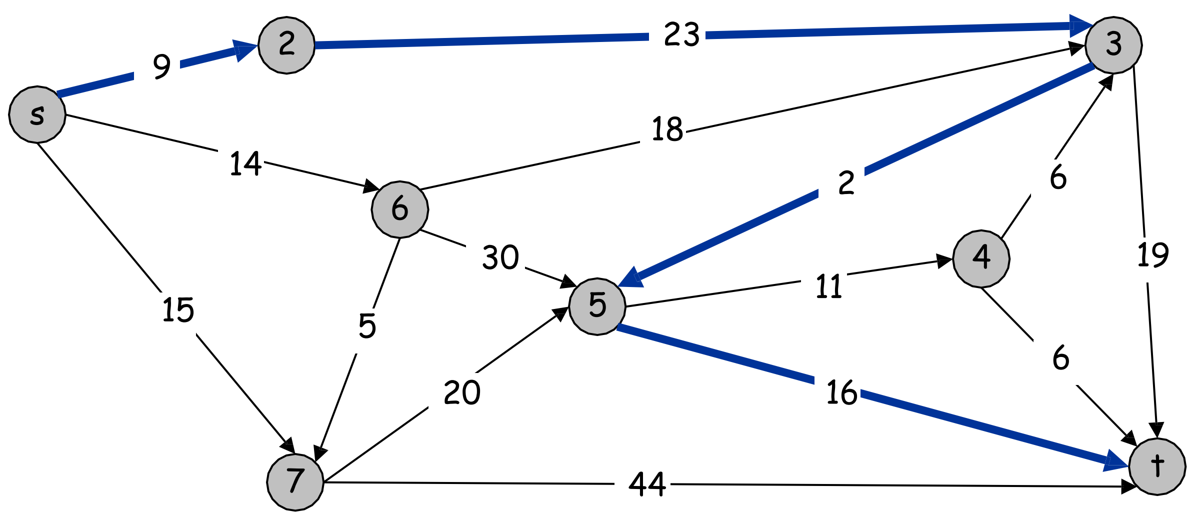


Length of path s-2-3-5-t
= 9 + 23 + 2 + 16
= 50.

Shortest Path Problem

Q: Does BFS give shortest path now that we have different lengths?

A: No



Shortest Path Problem

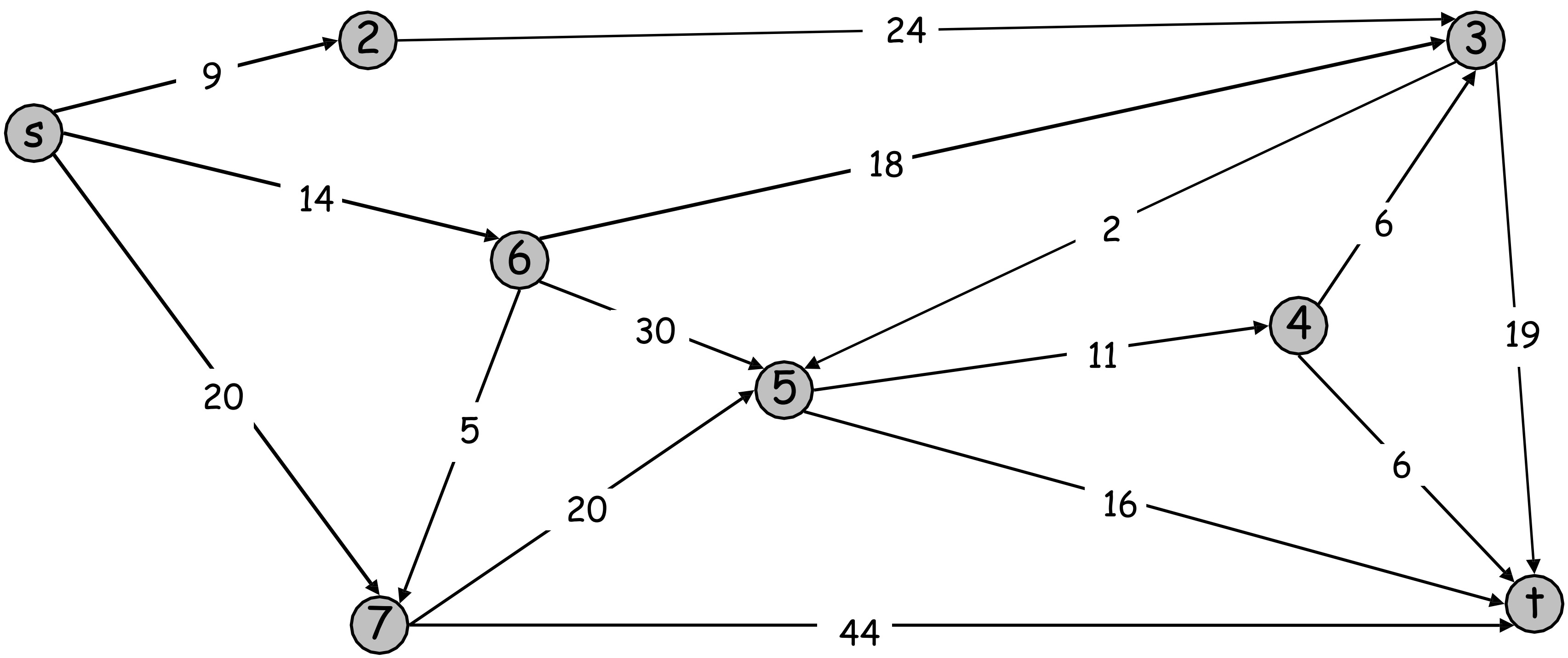
Q: Suppose all lengths are integers. Can we use BFS on a modified graph to find shortest path?

A: Replace each arc of length x by a path with x-1 intermediate nodes, run BFS in the new graph.

Dijkstra's Algorithm

Approach
- Find the node closest to **s**, then the second closest, then the third closest, and so on ..., computing their distances from **s** (similar to BFS)

Find closest node to **s**, second closest, etc.



Dijkstra's Algorithm

Define

$\pi(u)$ = smallest distance from start node s to u

using only visited nodes (and u)

We will start by only computing the distance d(u) from s to u

SlowDijkstra Algorithm

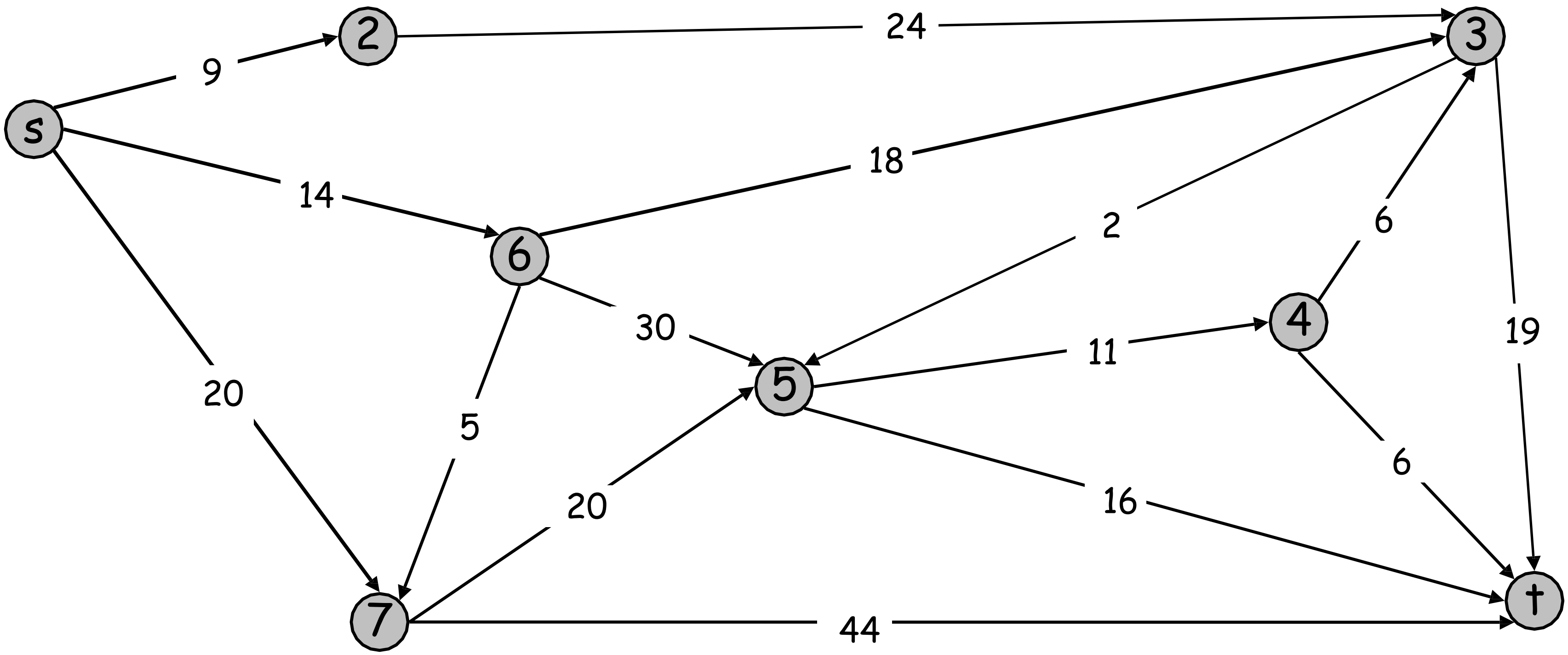
pi(s)=0, pi(u) = infinity for all other u, visited = {}

For i=1 to n-1

- Pick unvisited node u with smallest $\pi(u)$
- Add u to visited
- Set d(u) = π(u)
- (Update π) For each unvisited neighbor v of u, set

$$\pi(v) \leftarrow \min\{\pi(v), d(u) + c_{uv}\}$$

Dijkstra's Algorithm



Dijkstra's Algorithm

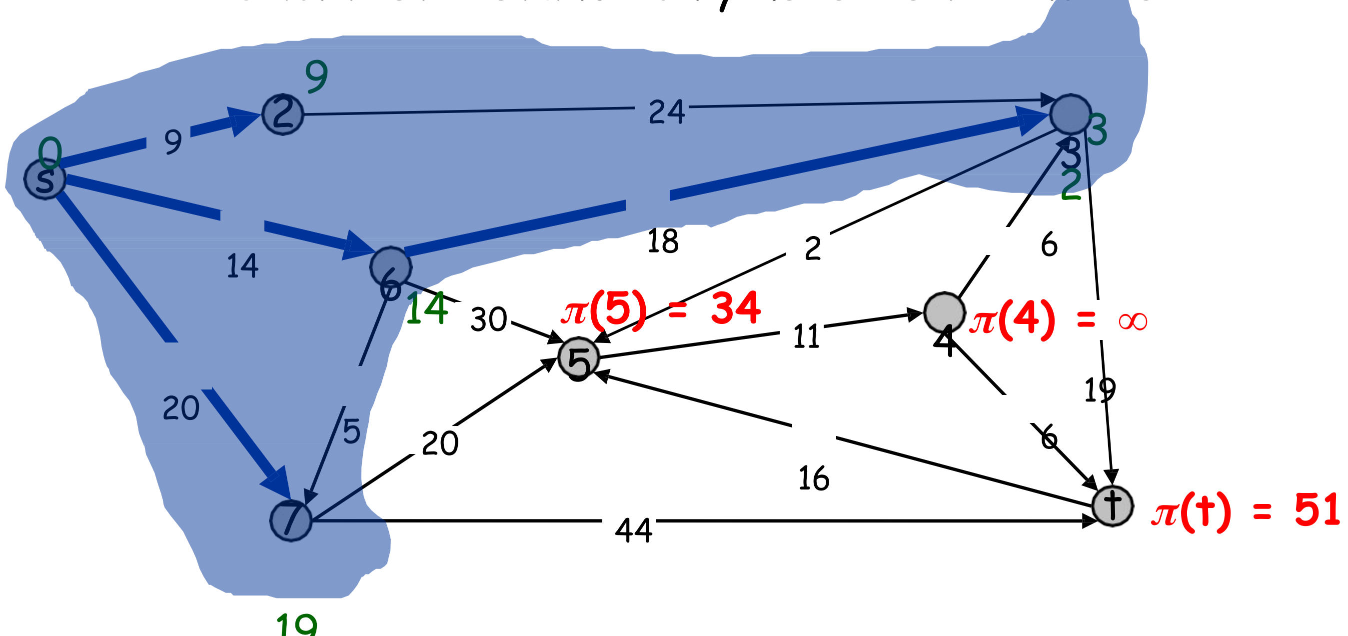
Correctness: This algorithm computes distances from s correctly

Consider an iteration of the algorithm. Suppose pi is correct

Obs 1: Let u be unvisited with smallest pi(u). Then pi(u) is the shortest distance from s to u

Why? Using any other unvisited node w is worse, since has to pay at least $\pi(w) + c_{wu} \geq \pi(u)$

=> Shortest distance only uses visited nodes



Dijkstra's Algorithm

Correctness: This algorithm computes distances from s correctly

Consider an iteration of the algorithm. Suppose pi is correct

Obs 2: π at the end of the iteration is also correct

Why? Shortest distance from s to v with visited nodes (that now include u) either

- **does not use u:** so equals pi(v) of the previous iteration
- **or uses u**: so costs going until u and then u->v: $d(u) + c_{uv}$

Algorithm picks the shortest option

Dijkstra's Algorithm

SlowDijkstra Algorithm

pi(s)=0, pi(u) = infinity for all other u, visited = {} For i=1 to n-1

- Pick unvisited node u with smallest $\pi(u)$
- Add u to visited
- Set d(u) = π(u)
- (Update π) For each unvisited neighbor v of u, set

$$\pi(v) \leftarrow \min\{\pi(v), d(u) + c_{uv}\}$$

Q: Time-complexity of SlowDijkstra?

A: Each iteration takes at most O(n) [picking smallest + updating pi]

=> Total: O(n^2)

Dijkstra's Algorithm

Dijkstra Algorithm

pi(s)=0, pi(u) = infinity for all other u, visited = {}, MakeHeap For

i=1 to n-1

- Pick unvisited node u with smallest $\pi(u)$
- Add u to visited
- Set d(u) = π(u)
- (Update π) For each unvisited neighbor v of u, set

$$\pi(v) \leftarrow \min\{\pi(v), d(u) + c_{uv}\}$$

Q: Complexity of Dijkstra if we keep pi in a heap? A: Initialization: O(n) to make heap

Each iteration:

- O(log n) for finding and removing from heap node with smallest $\pi(u)$
- O(out-deg(u) * log(n)) for updating pi's
- O(1) for all else

Total (including initialization): $O((n+m)\log n)$

Dijkstra's Algorithm: Getting the Path

Q: How to get shortest path from s to t, not just distance?

A: Similar to BFS:

- Keep track of who caused last update to node u

Dijkstra Algorithm

pi(s)=0, pi(u) = infinity for all other u, visited = {}, MakeHeap

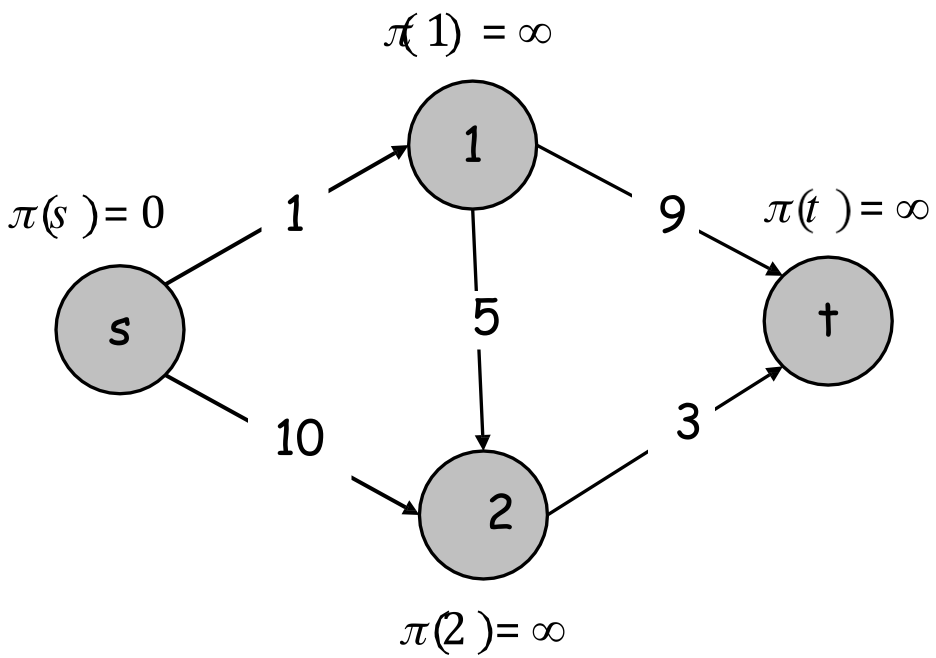
For i=1 to n-1

- Pick unvisited node u with smallest $\pi(u)$
- Add u to visited
- Set d(u) = π(u)
- (Update π) For each unvisited neighbor v of u
  - set $\pi(v) \leftarrow \min\{\pi(v), d(u) + c_{uv}\}$
  - if updated $\pi(v)$, set parent(v) = u

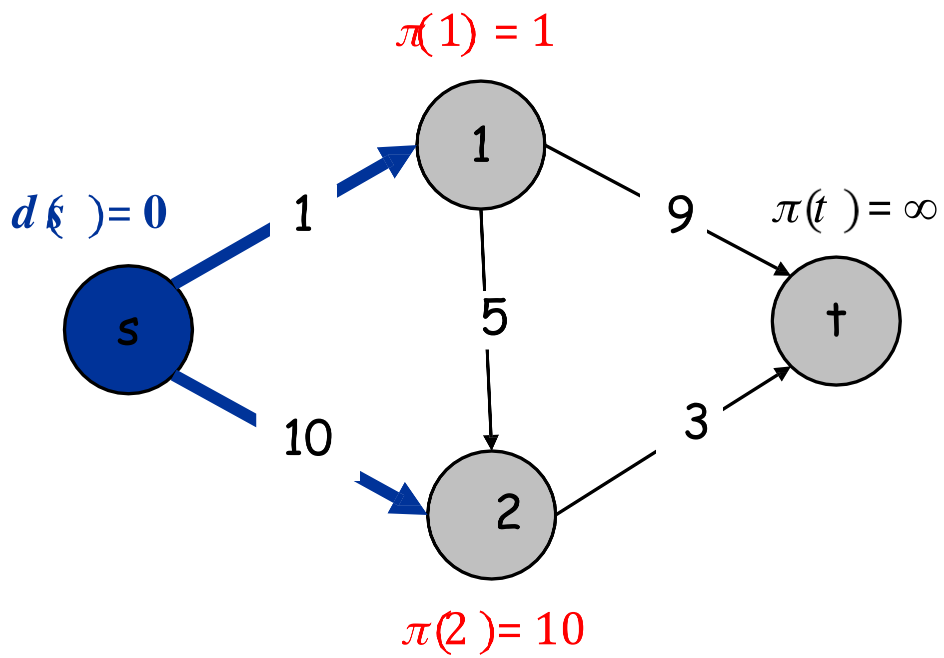
- Traverse starting from t, follow its parent, and its parent, etc.

Dijkstra's Algorithm: Getting the Path

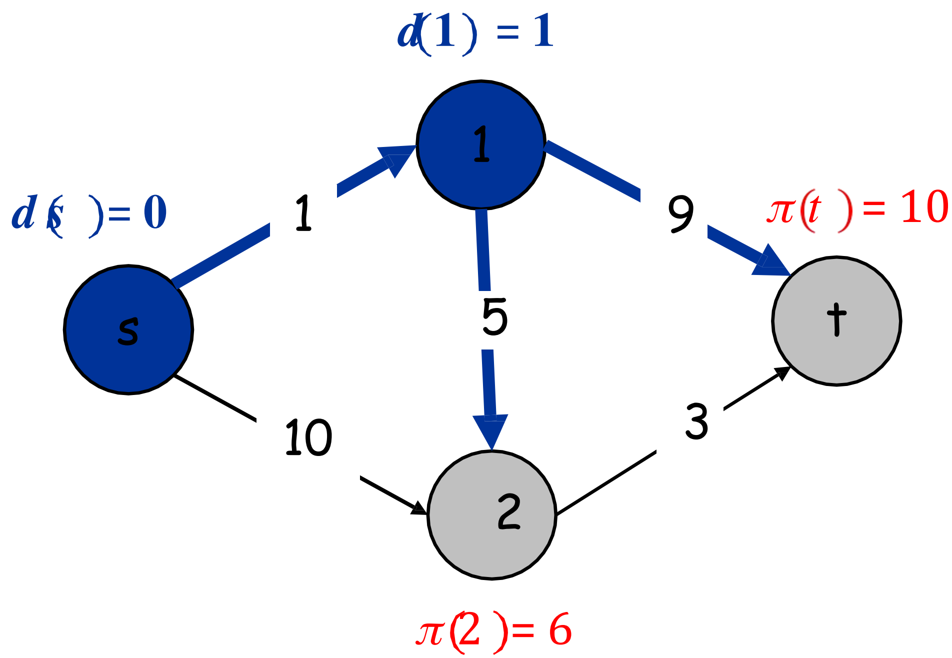
("Parent" will be indicated by blue edges)



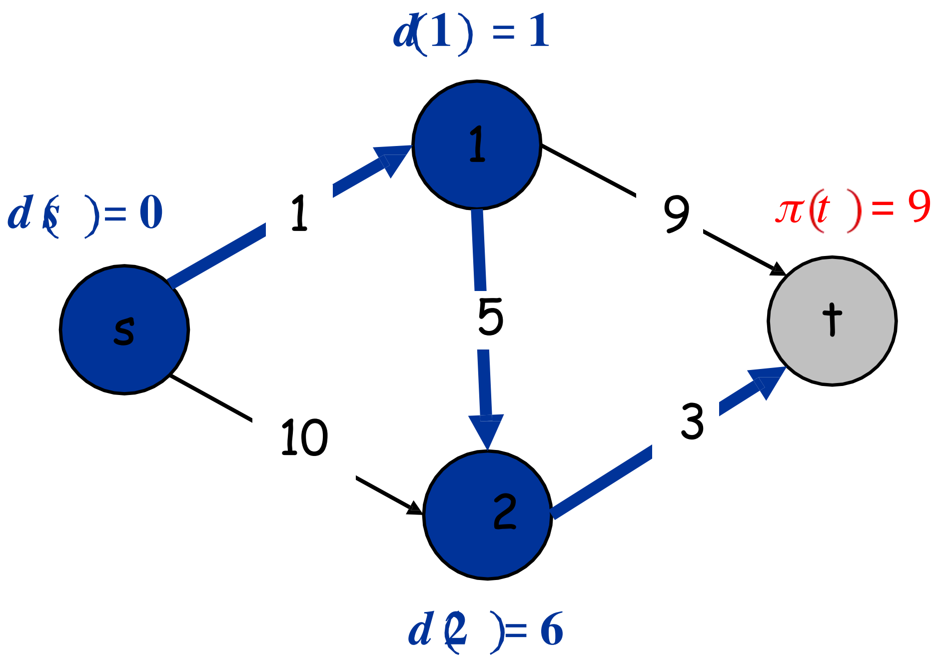
Dijkstra's Algorithm: Getting the Path



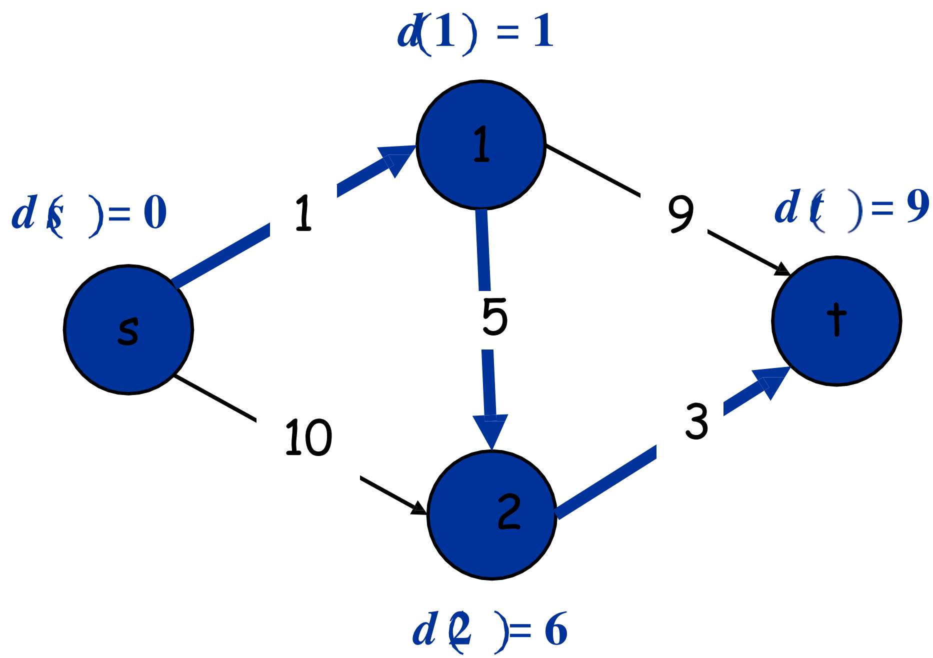
Dijkstra's Algorithm: Getting the Path



Dijkstra's Algorithm: Getting the Path



Dijkstra's Algorithm: Getting the Path



Exercises: Weighted Shortest Paths

Exercise 1: Run Dijkstra's algorithm on the following graph, starting from node **s**

Exercise 2: Can we run Dijkstra's algorithm on undirected graphs? How?

Exercise 3: Show that Dijkstra's algorithm may not return the correct distance if there are negative lengths (construct a graph)

Exercise 4: Consider a slightly different problem: You are given a directed graph and costs on the **nodes**. You want to find the shortest cost path from s to t, where the cost of a path is the sum of the costs of the nodes in the path.

Find an algorithm to solve this problem. (Hint: run Dijkstra on a modified graph)