Algorithms and Complexity

Graphs: Representations and Exploration

Julian Mestre

School of Information Technologies
The University of Sydney

THE UNIVERSITY OF SYDNEY

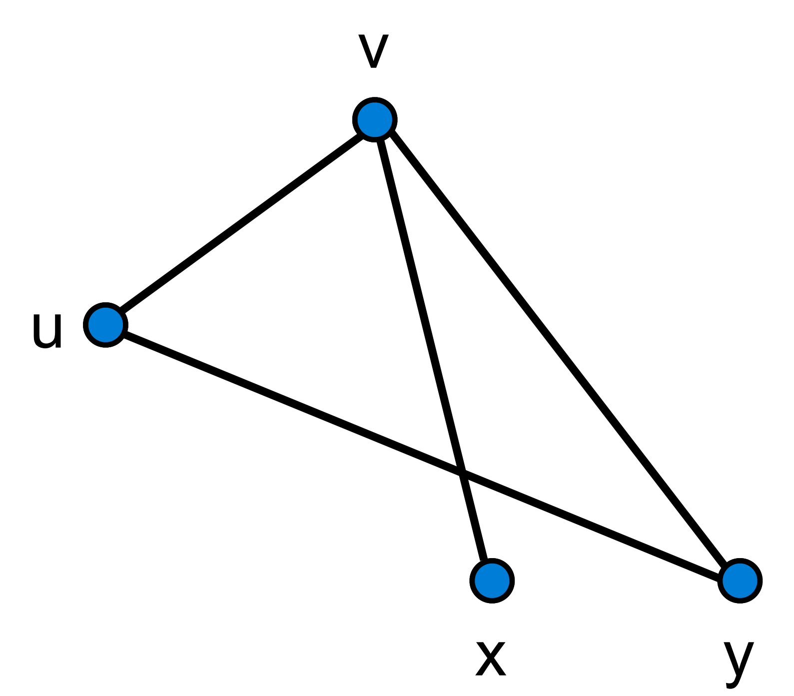
Undirected graphs

Let G=(V,E) be an undirected graph:

- V = set of vertices (a.k.a. nodes)
- E = set of edges

Some notation

- deg(u) = # edges incident to u
- deg(G) = max u deg(u)
- N(u) = neighborhood of u
- n = |V|
- m = |E|



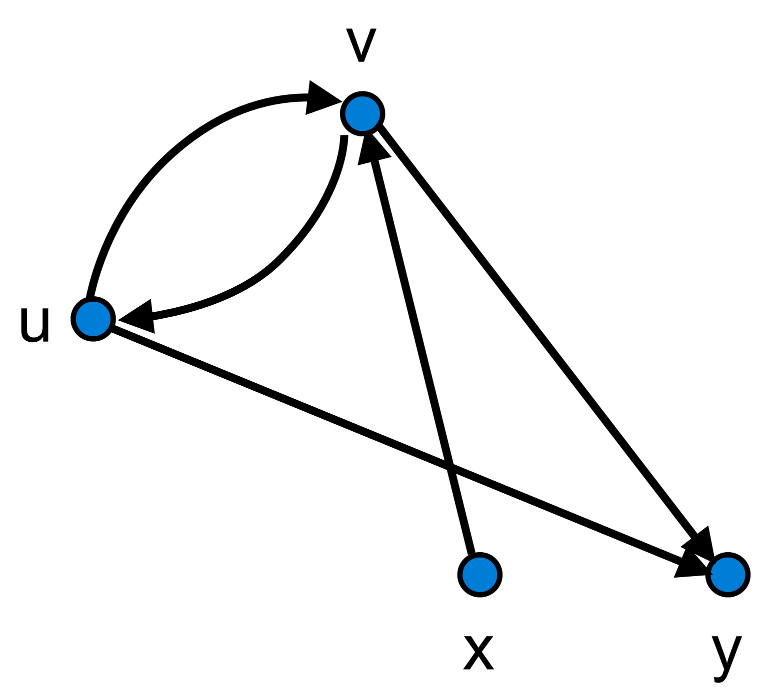
Directed graphs

Let G=(V,E) be a directed graph:

- V = set of vertices (a.k.a. nodes)
- E = set of directed edges (a.k.a. arcs)

Some notation

- out-deg(u) = # arcs out of u
- in-deg(u) = # arcs into u
- $N^{out}(u)$ = out neighborhood of u
- $N^{in}(u)$ = in neighborhood of u



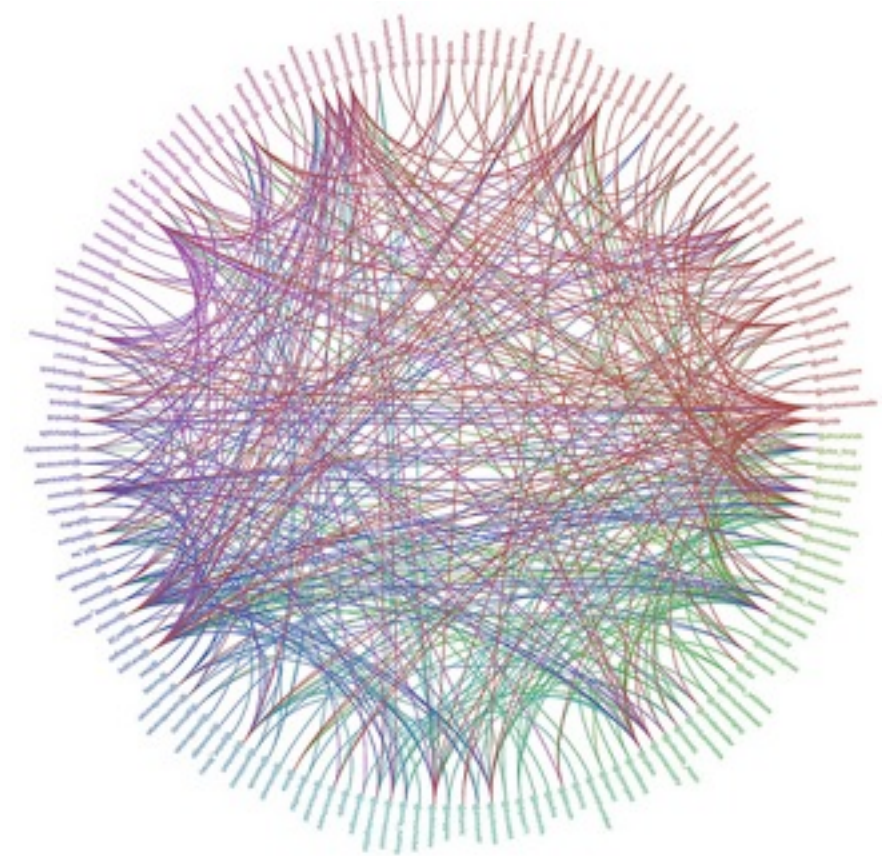
Graphs as a modeling tool

Can model many relations among elements in a set:

- Social network
- Internet topology
- Protein-protein interaction

Can help formulate problems:

- What's the distance between two nodes?
- What's a central node?
- How well connected the network is?
- What's a critical node?



Graph connectivity

Let $G=(V,E)$ be an undirected graph:

- Sequence $v_1, v_2, \ldots, v_k$ is a *path* if (v_i, v_{i+1}) is an edge in E for all $i=1, \ldots, k-1$
- The *length* of the path is the # of edges in it
- A path is *simple* if no repeated vertices
- A *cycle* is a path $v_1, v_2, \ldots, v_k$ where $v_1 = v_k$
- A cycle $v_1, v_2, \ldots, v_k$ is *simple* if $v_1, v_2, \ldots, v_{k-1}$ is a simple path
- G is connected if every every vertex can reach every other vertex

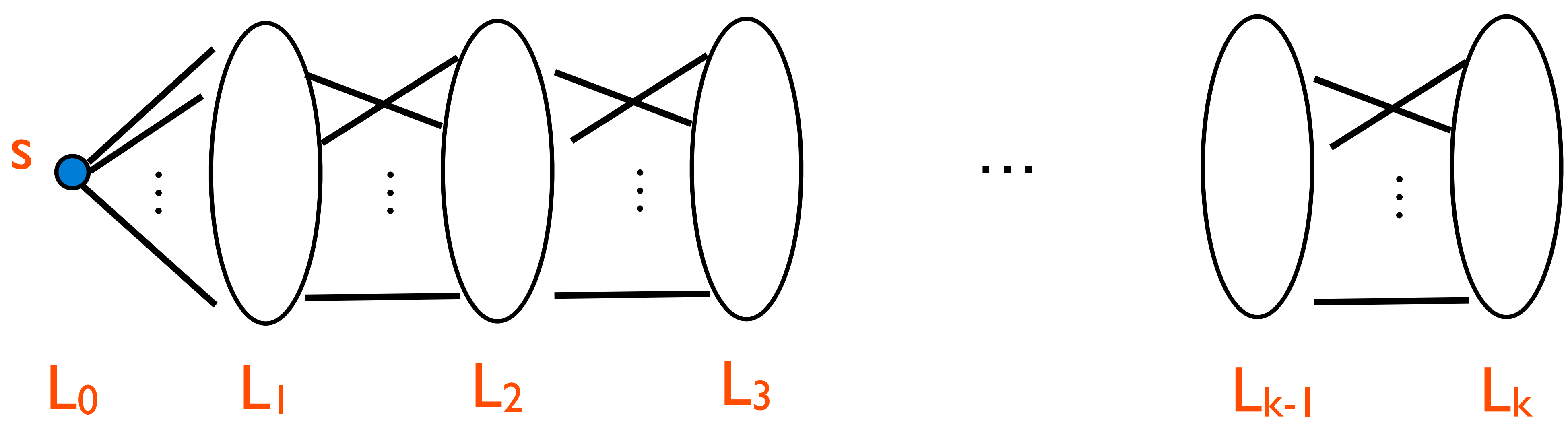
We say that G is *tree* if:

- G is connected and doesn't have a cycle, or equivalently
- G is connected and $|E| = |V|-1$

Breath first search

Exploring a graph from a starting vertex s, layer by layer:

- L_0 = {s}
- L_1 = vertices that are one hop away from s
- L_2 = vertices that are two hops away from s
- ⋮
- L_k = vertices that are k hops away from s



Breath first search

```
def BFS(G,s):

  layers = []
  current_layer = [s]
  next_layer = []
  “mark every vertex except s as not seen”
  while “current_layer not empty” :
    layers.append(current_layer)
    for u in current_layer:
        for v in “neighborhood of u”:
           if “haven’t seen v yet”:
              next_layer.append(v)
              “mark v as seen”
    current_layer = next_layer
    next_layer = []

  return layers
```

Properties of BFS

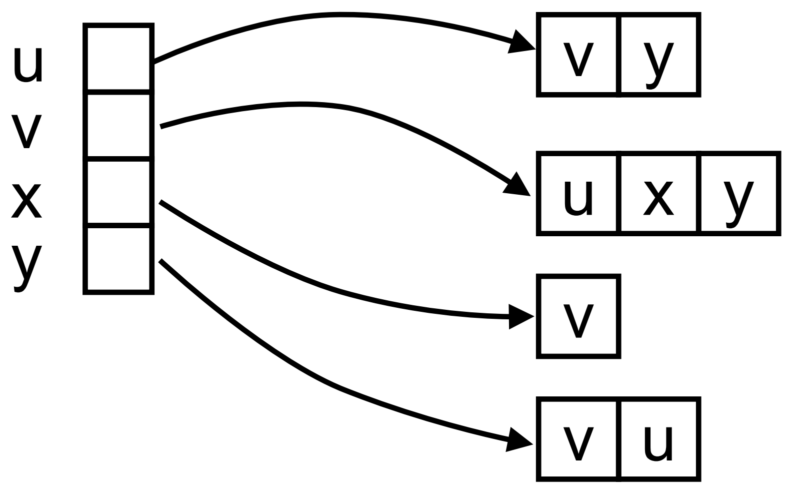
Let G be a graph and s a vertex in G. Suppose we run BFS(G,s) and that it returns layers $L_0, L_1, \ldots, L_k$, then for u:

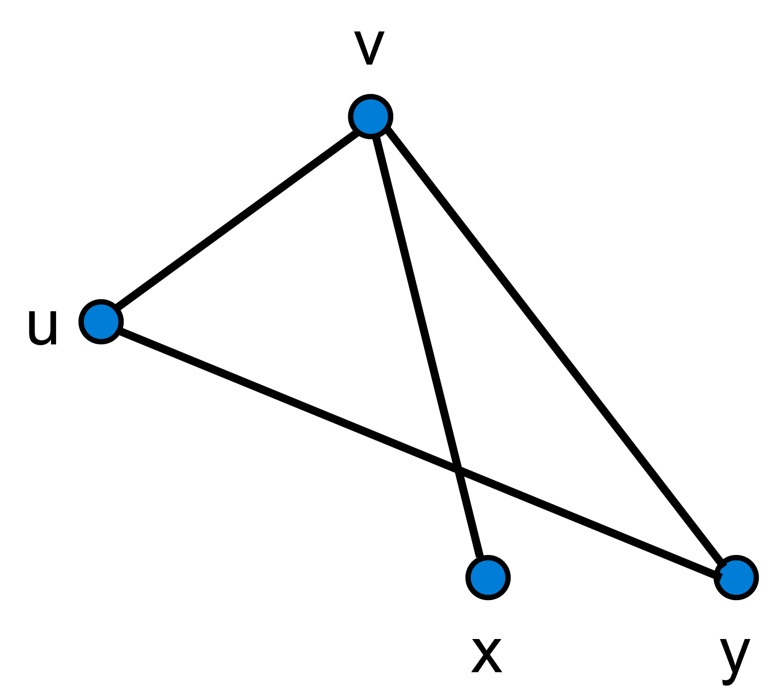
- If u belongs to some layer L_i, then there is a path from s to u
- If there is a path from s to u, then u belongs to some L_i
- In fact, u belongs to L_i if and only if the shortest s-u path has i edges

Edges across layers must connect adjacent layers.

Graph representation

Adjacency lists





Adjacency matrix

	u	v	x	y
u	0	1	0	1
v	1	0	1	1
x	0	1	0	0
y	1	1	0	0

Complexity depends on representation

Scan neighborhood of vertex u

- Adj. list : $\Theta(|N(u)|)$
- Adj. matrix : $\Theta(n)$

Check if u and v are adjacent:

- Adj. list : $\Theta(\min\{|N(u)|, |N(v)|\})$
- Adj. matrix : $\Theta(1)$

Space:

- Adj. list : $\Theta(|V|+|E|)$
- Adj. matrix : $\Theta(|V|^2)$

Time complexity of BFS

```
def BFS(G,s):

  layers = []
  current_layer = [s]
  next_layer = []
  “mark every vertex except s as not seen”
  while current_layer “not empty”:
    layers.append(current_layer)
    for u in current_layer:
        for v in “neighborhood of u”:
           if “haven’t seen v yet”:
              next_layer.append(v)
              “mark v as seen”
    current_layer = next_layer
    next_layer = []

  return layers
```

This takes $O(|V|)$ time

This loop takes $O(|N(u)|)$ time*

Adding up over all u, we get $O(\sum_u |N(u)|) = O(|E|)$

Recap: graphs and BFS

Graph:

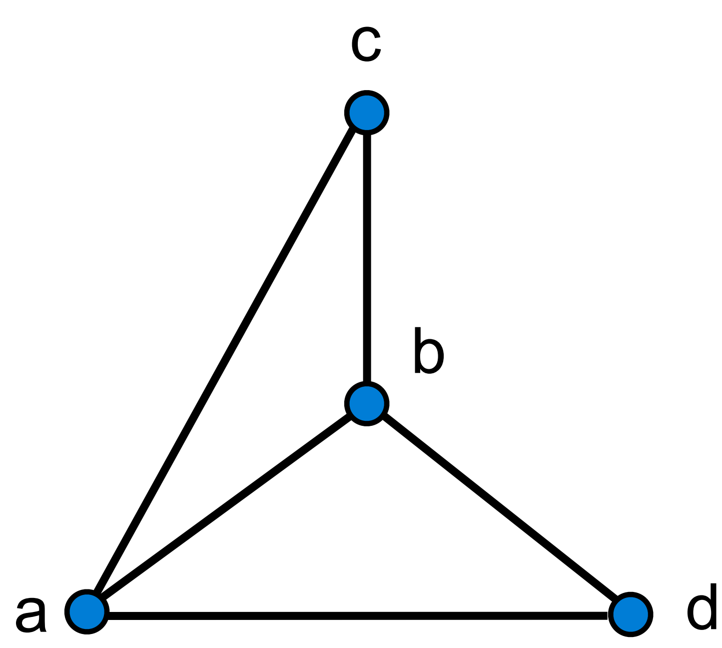
- discrete object encoding a relation between vertices
- representations: adjacency lists, adjacency matrix
- time complexity of basic primitives depends on representation

Breath first search (BFS):

- a graph exploration strategy
- starting from a vertex s, visit vertices reachable from s, layer by layer
- L_i holds vertices at distance i from s
- Runs in $O(m+n)$ time if the graph is represented with adjacency lists

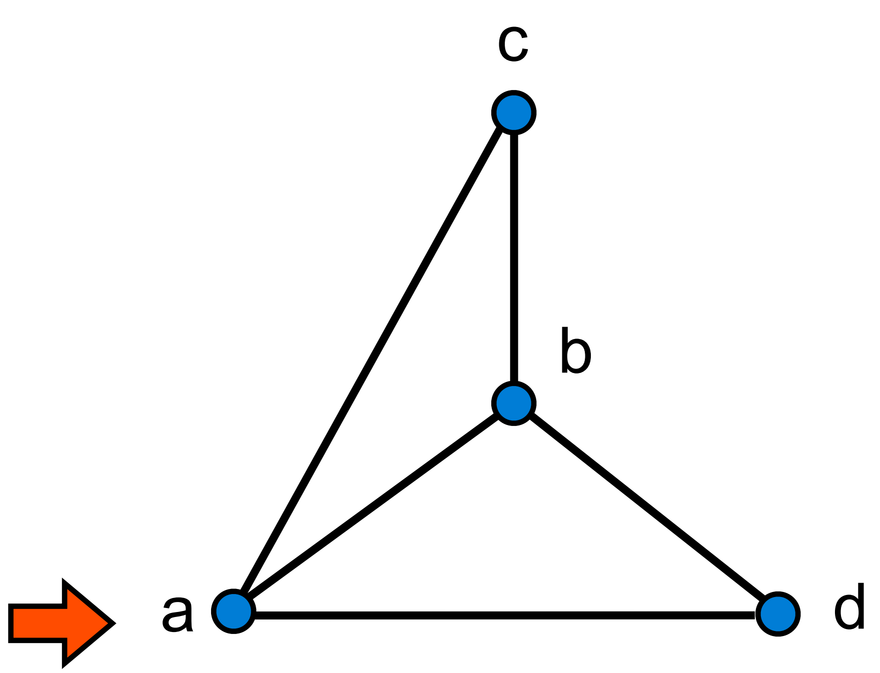
Depth first search

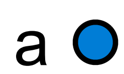
Pick a starting vertex, follow outgoing edges that lead to new vertices, and backtrack whenever "stuck".



Depth first search

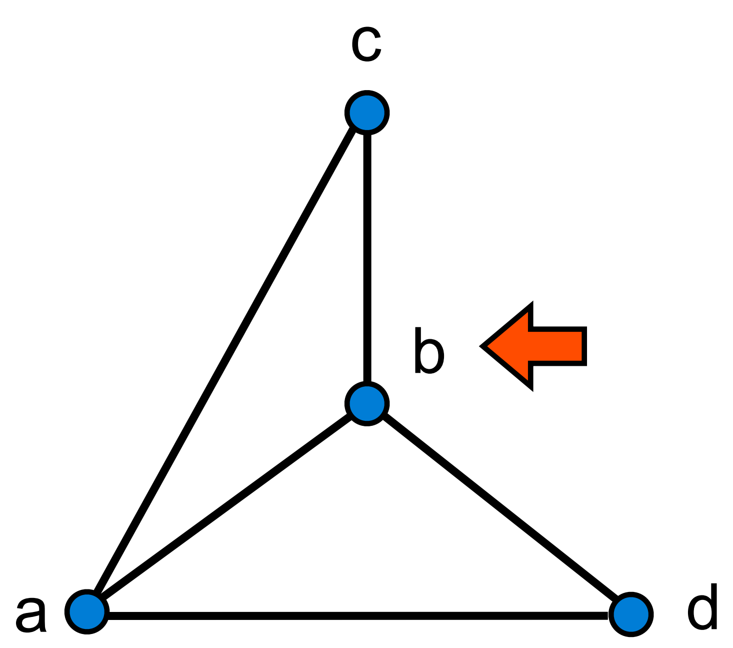
Pick a starting vertex, follow outgoing edges that lead to new vertices, and backtrack whenever "stuck".

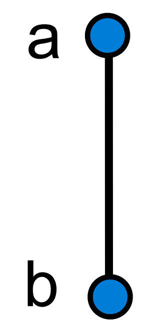




Depth first search

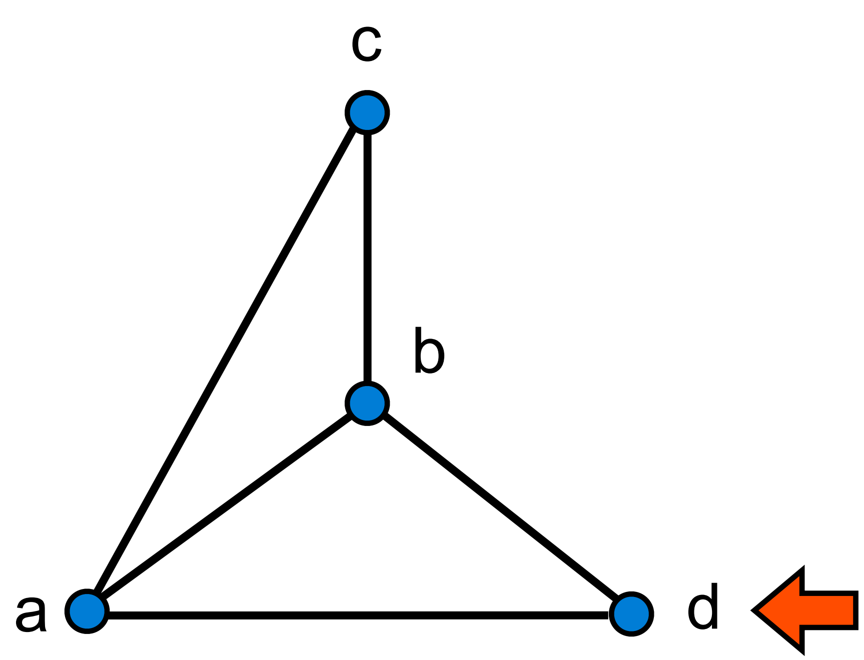
Pick a starting vertex, follow outgoing edges that lead to new vertices, and backtrack whenever "stuck".

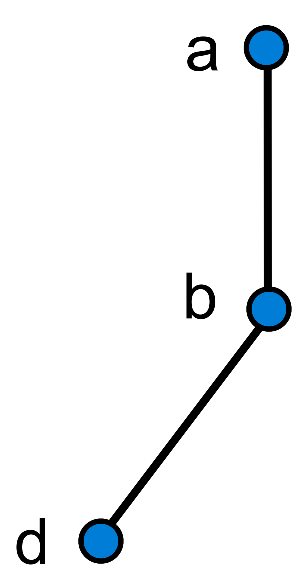




Depth first search

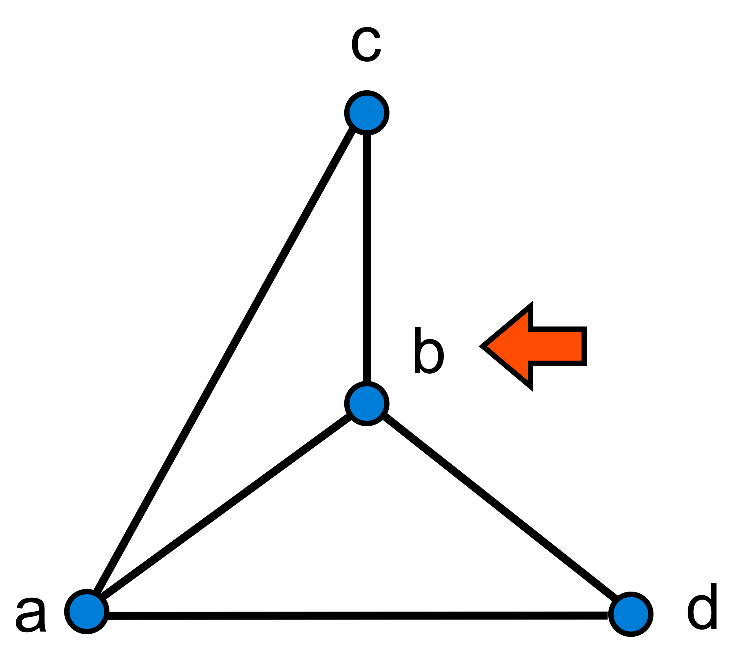
Pick a starting vertex, follow outgoing edges that lead to new vertices, and backtrack whenever "stuck".

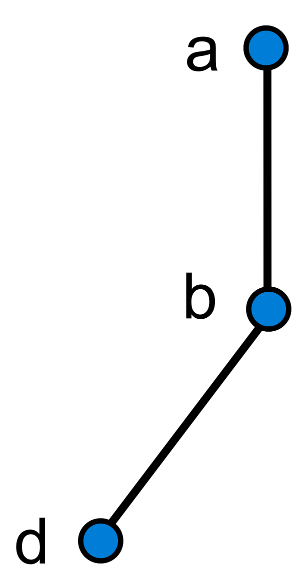




Depth first search

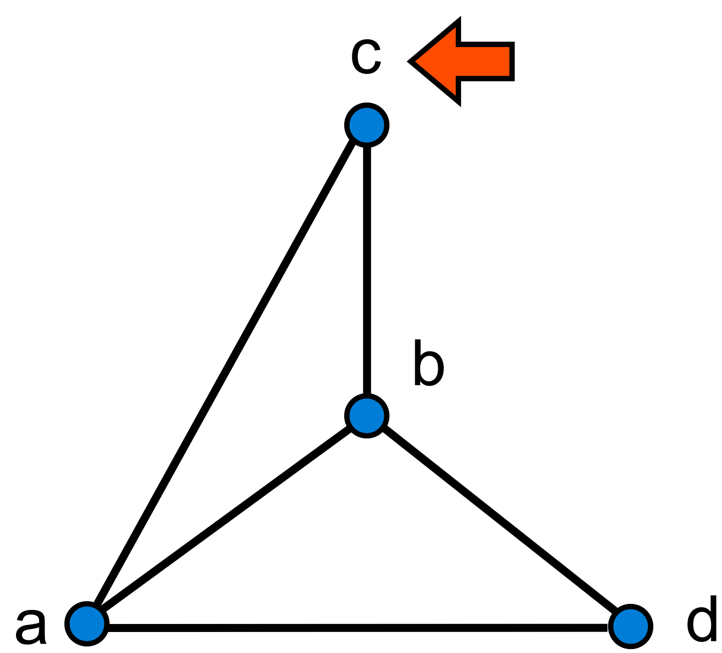
Pick a starting vertex, follow outgoing edges that lead to new vertices, and backtrack whenever "stuck".

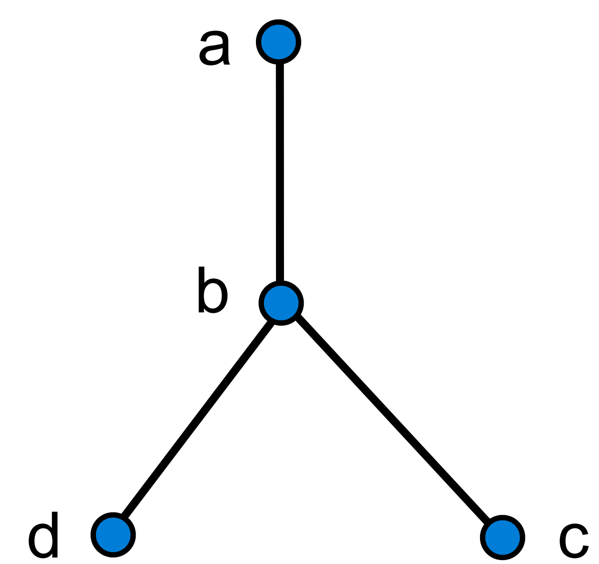




Depth first search

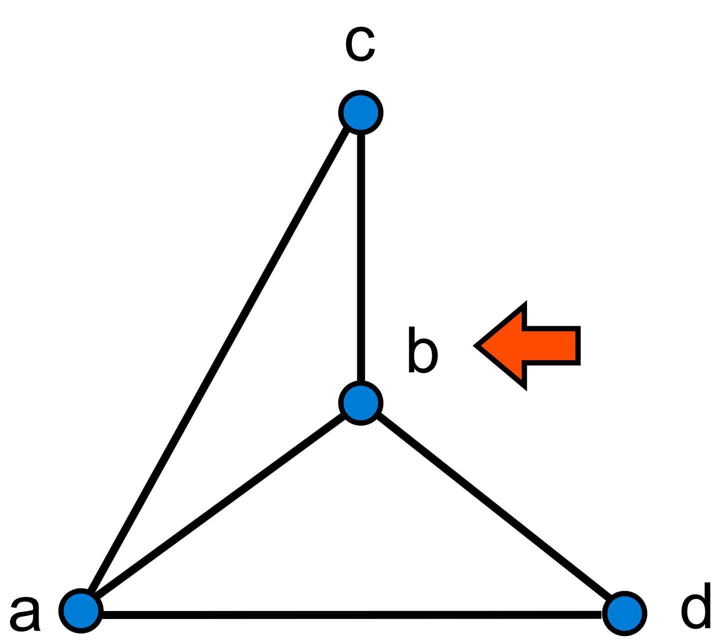
Pick a starting vertex, follow outgoing edges that lead to new vertices, and backtrack whenever "stuck".

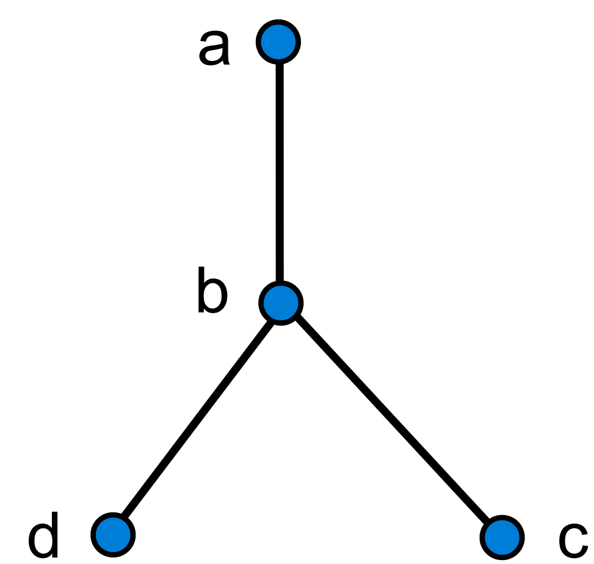




Depth first search

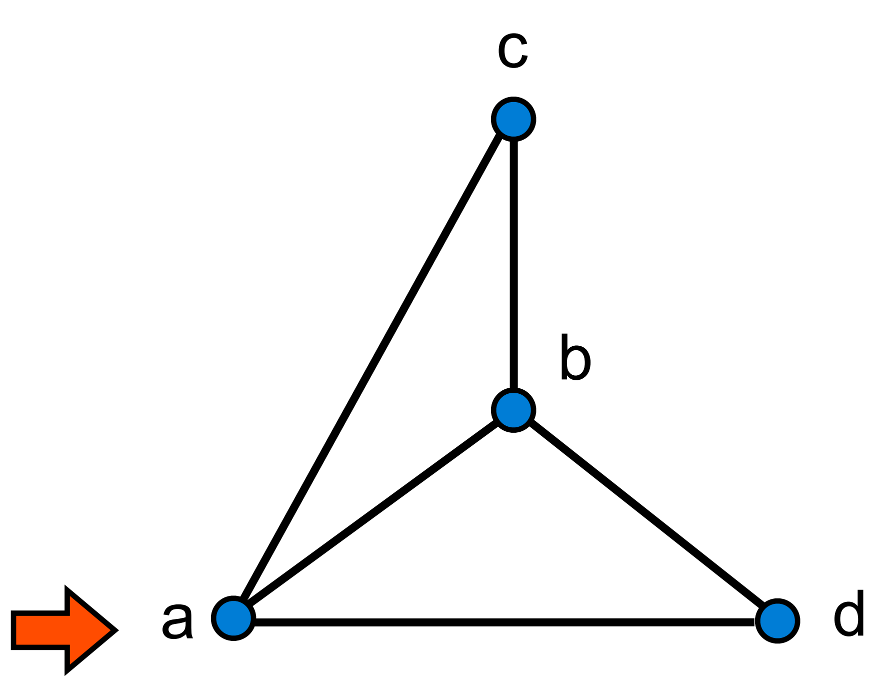
Pick a starting vertex, follow outgoing edges that lead to new vertices, and backtrack whenever "stuck".

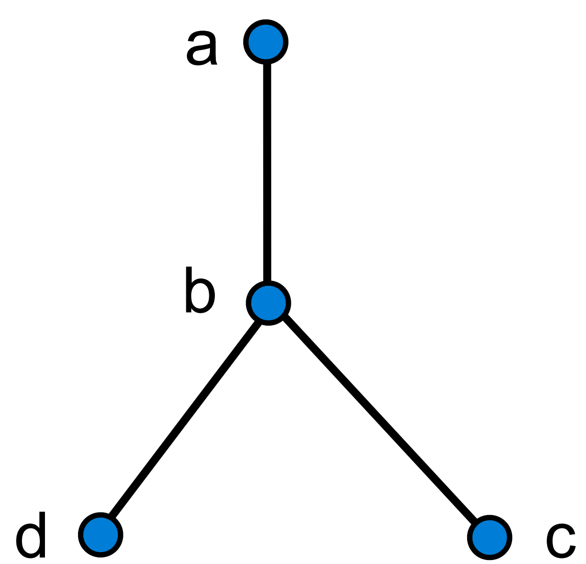




Depth first search

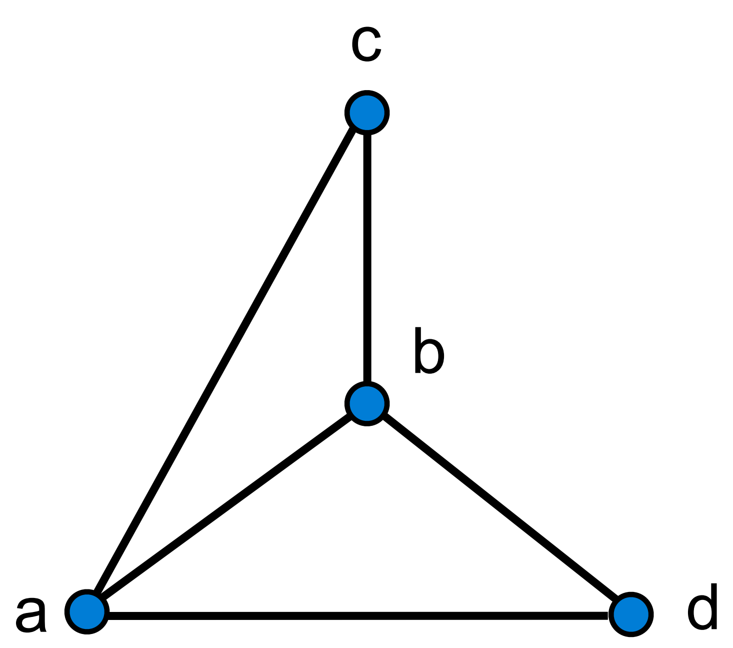
Pick a starting vertex, follow outgoing edges that lead to new vertices, and backtrack whenever "stuck".

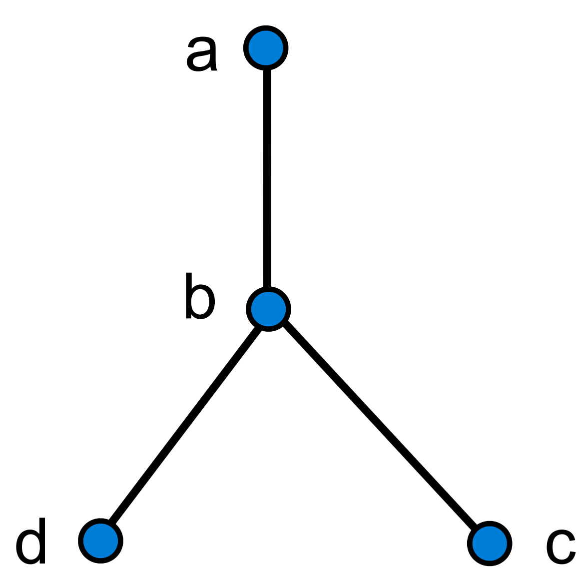




Depth first search

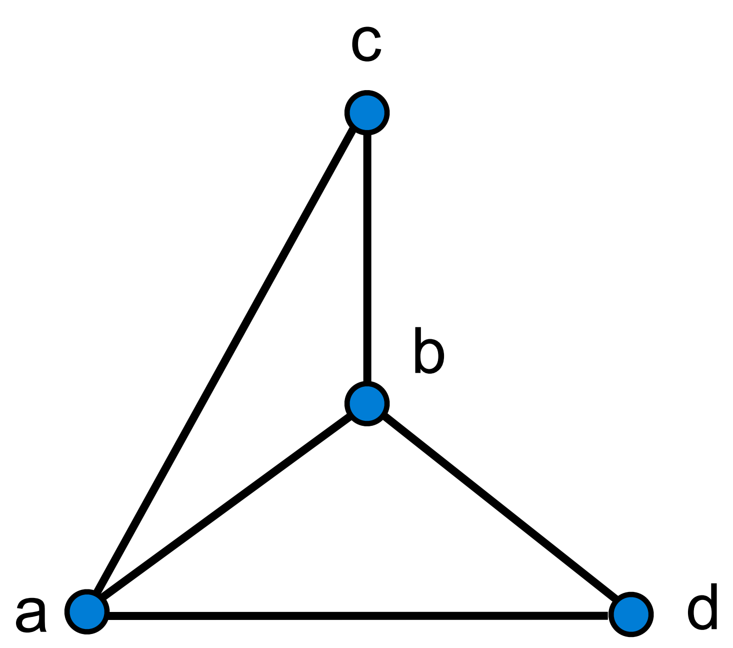
Pick a starting vertex, follow outgoing edges that lead to new vertices, and backtrack whenever "stuck".

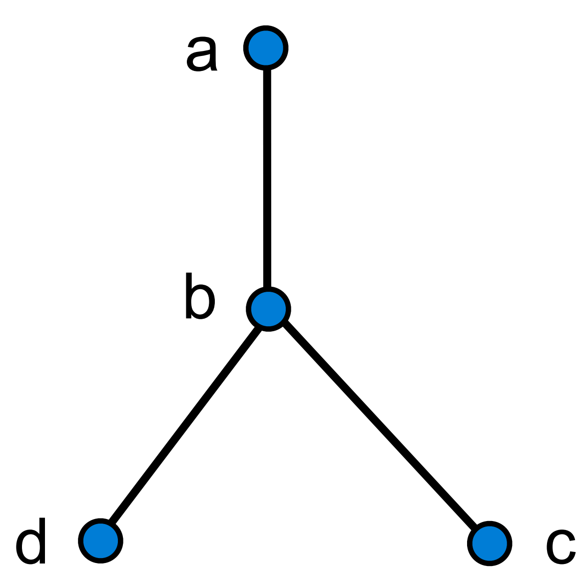




Depth first search

Pick a starting vertex, follow outgoing edges that lead to new vertices, and backtrack whenever "stuck".



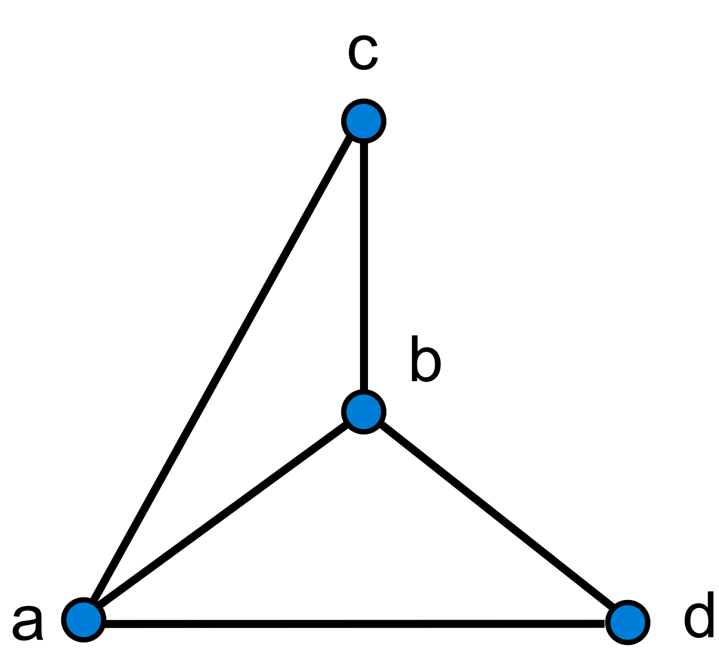


DFS tree

Depth first search

```
def DFS(G):
  "mark vertices as unvisited"
  "set vertices' parent as None"
  time = 0
  for u in "vertices of G":
    if "haven't seen u yet":
      DFS_visit(u)
  return parent
```

```
def DFS_visit(u):
  "mark u as visited"
  time = time + 1
  discovery[u] = time
    for v in "neighborhood of u":
    if "haven't seen v yet":
      parent[v] = u
      DFS_visit(v)
  time = time + 1
  finish[u] = time
```



Time complexity of DFS

```
def DFS(G):
  "mark vertices as unvisited"
  "set vertices' parent as None"
  time = 0
  for u in "vertices of G":
    if "haven't seen u yet":
      DFS_visit(u)
  return parent
```

```
def DFS_visit(u):
  "mark u as visited"
  time = time + 1
  discovery[u] = time
    for v in "neighborhood of u":
    if "haven't seen v yet":
      parent[v] = u
      DFS_visit(v)
  time = time + 1
  finish[u] = time
```

Ignoring work done inside function calls, it runs in O(n) time

Ignoring work done inside recursive calls, it runs in O(|N(u)|) time

⇒ O(m) time overall here

Properties of DFS

The subset of edges {(u, parent[u]): u in V} forms a collection of trees (a.k.a. forest)

An undirected graph is connected if and only if we have a single tree in the DFS forest. In fact, each tree corresponds to a connected component of the graph.

Each discovery and finishing time is a unique number in [1,2n]

Cut edges

Def.: In a connected graph $G=(V,E)$, we say that $(u,v) \in E$ is a *cut edge* if $(V, E-(u,v))$ is not connected.

Trivial algorithm runs in $O(m^2)$: For each edge $(u,v) \in E$

- Remove (u,v) from G
- Run DFS to check if the new graph is still connected

A better algorithms runs in $O(nm)$: For each edge in DFS tree

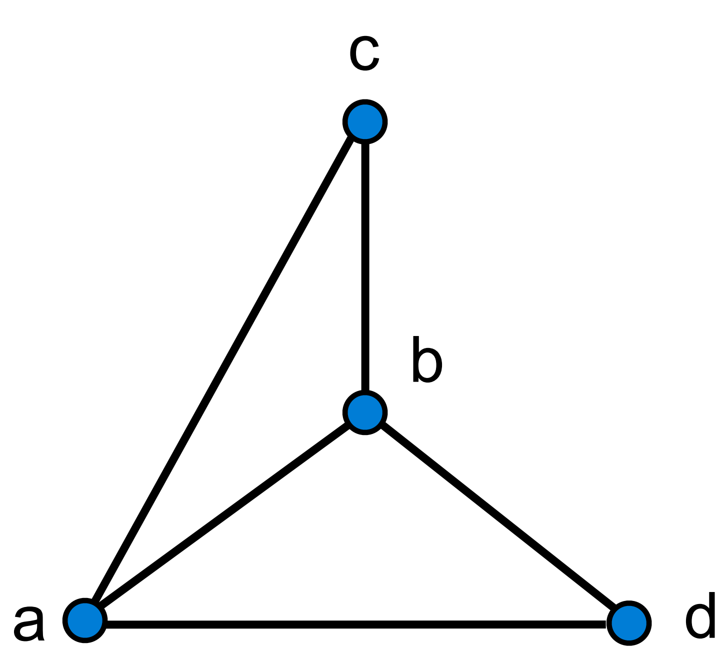
- Remove (u,v) from G
- Run DFS to check if the new graph is still connected

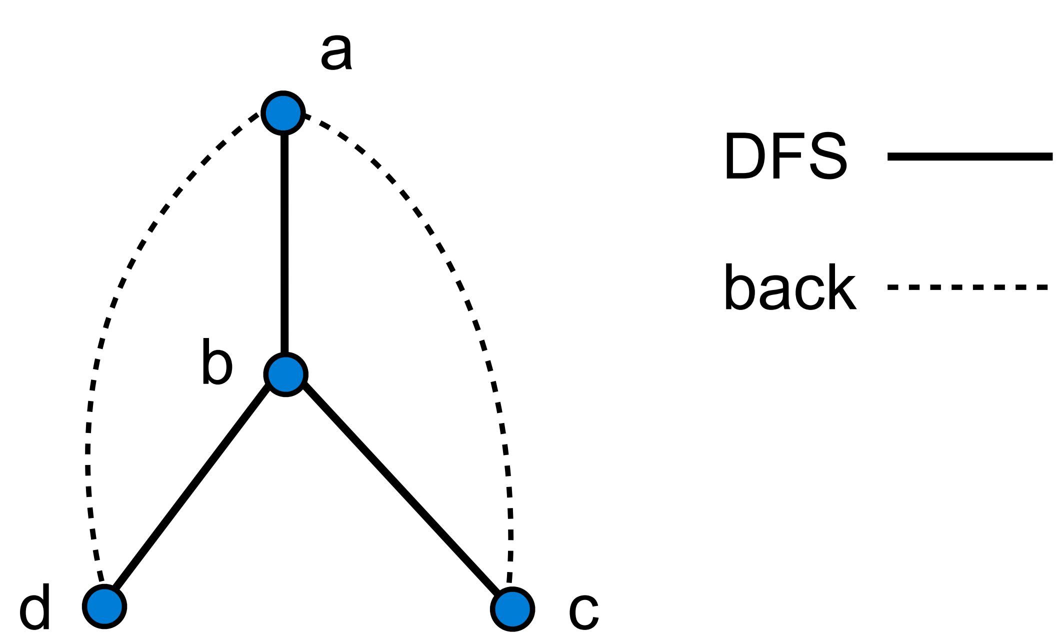
Back edges in DFS forest

Def.: If parent[u] ≠ None then we call (u, parent(u)) a *tree edge*

Def.: We say that a non-tree edge (u,v) is a *back edge* if u is a descendant of v in the DFS forest, or vice-versa

Obs.: In the DFS forest every non-tree edge is a back edge





Cut vertex

Def.: In a connected graph $G=(V,E)$, we say that $u \in V$ is a *cut vertex* if $G-u$ is not connected.

Obs.: If u is the root of the DFS tree, then u is a cut vertex if and only if it has two or more children

Obs.: If u is a leaf of the DFS tree, then u is not a cut vertex

Obs.: If u is not the root of the DFS, u is a cut vertex it has a child v and no vertex in v's subtree can "jump over" u

Cut vertices

Let u be an internal vertex in the DFS tree.

v is allowed to be u

Def.: up[u] = min discovery[v] where v ∈ N(u)

Def.: down-&-up[u] = min up[v] where v is a descendent of u

An internal vertex u is a cut vertex if and only if it has a child v:

down-&-up[v] = discovery[u]

Only need to show how to compute down-&-up in O(m) time

Thm. Given a connect graph, there is an O(m) time algorithm for computing its cut vertices

Recap: DFS

Another graph exploration strategy: Follow edges to new nodes until "stuck", then backtrack

Vertices are assigned a discovery and a finishing time

In undirected graph, each edge can be a tree edge or a back edge

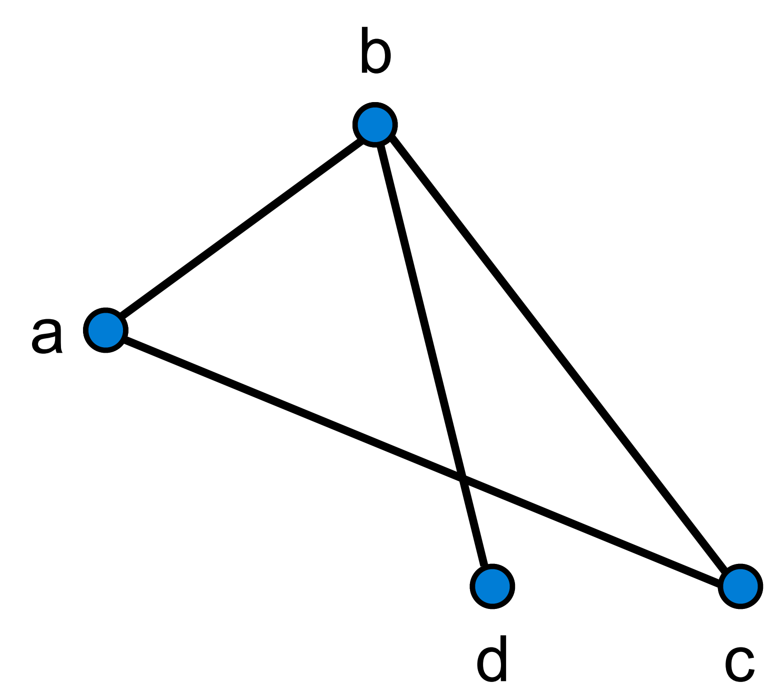
DFS is useful for solving other graph problems, like cut edges and cut vertices

Runs in O(m+n) time if graph is represented with adjacency lists

Simple Python graph representation

Let G=(V, E) be a graph on n vertices, then:

- Vertices can be any non-mutable object: e.g., a string, or a number
- G is a dictionary (dict)
- G[u] = adjacency list of vertex u



Example:

- ```
  G = { "a": ["b", "c"],
        "b" : ["a", "d", "c"],
        "c" : ["a", "b"],
        "d" : ["b"] }
  ```
- To iterate over vertices in G use → `for u in G:`
- To iterate over neighbors of "a" use → `for u in G["a"]:`
  ```

This week

Quiz 2

- 15 minutes long
- during your tutorial session

Problem set 2:

- will be posted later today (Monday 6 August)
- make sure you work on it before the tutorial

Assignment 1:

- Is due tomorrow (Tuesday 7 August)

Assignment 2:

- Out tomorrow, due next Tuesday