Algorithms and Complexity

Graphs: Representations and Exploration

Julian Mestre

School of Information Technologies
The University of Sydney

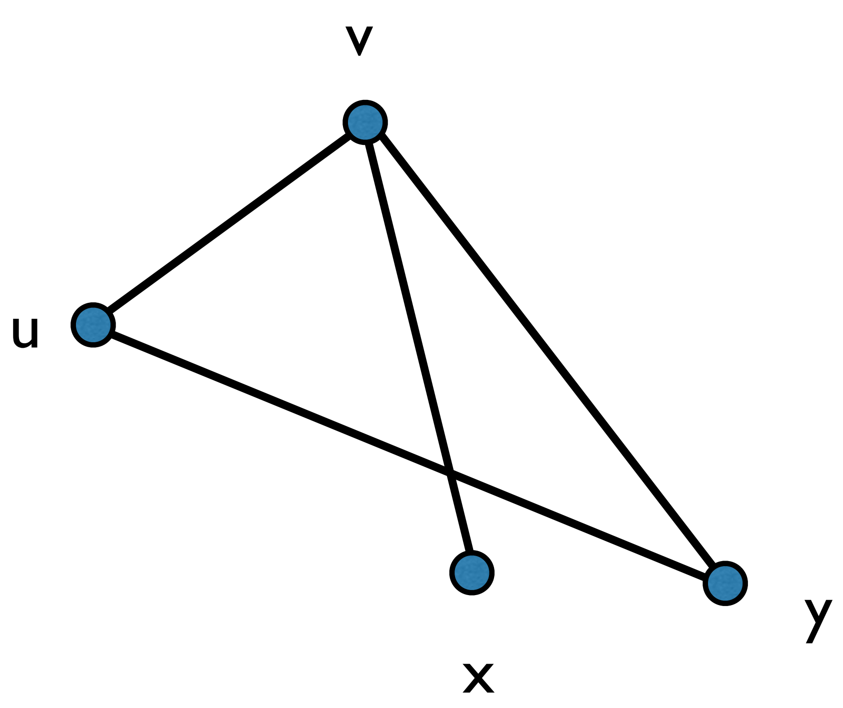
Undirected graphs

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



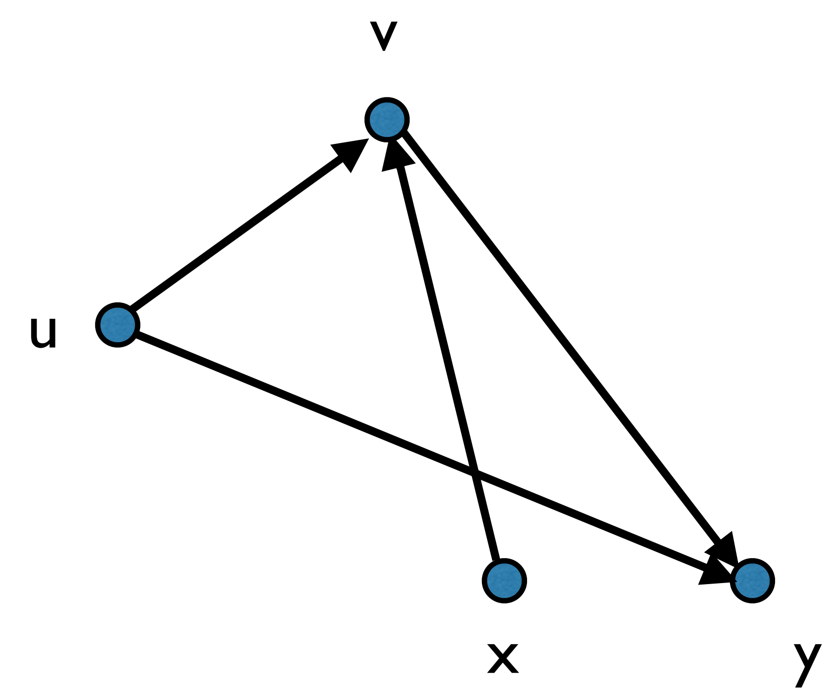
Directed graphs

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

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



THE UNIVERSITY OF SYDNEY

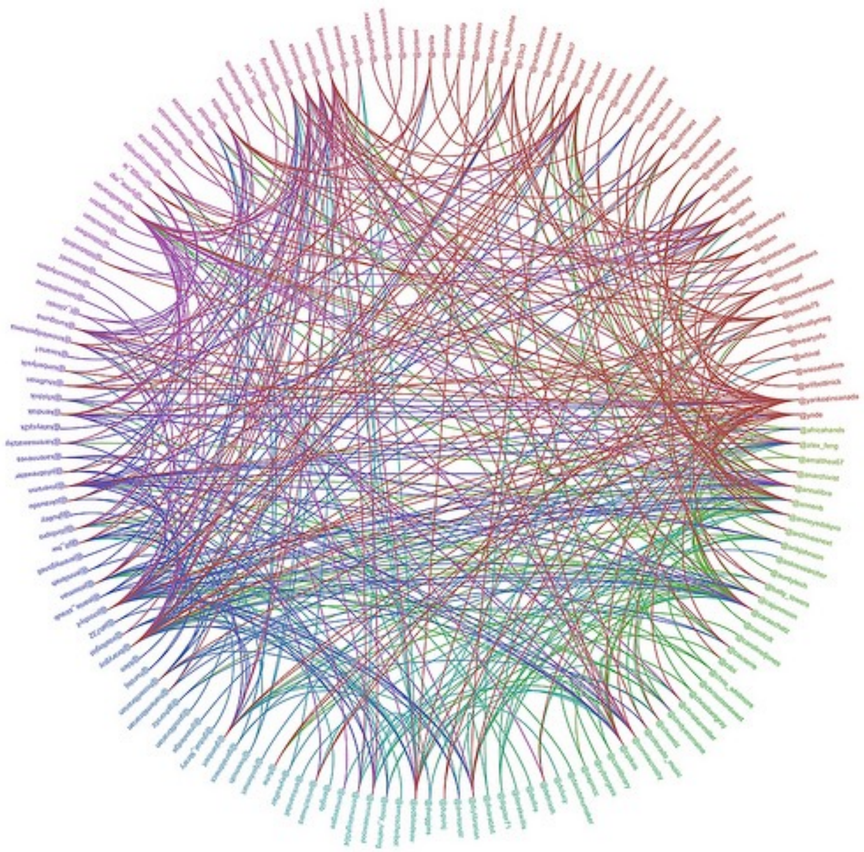
Graphs as a modeling tool

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 $v_1, v_2, ..., v_k$ is a path if (v_i, v_{i+1}) is an edge in E for all $i = 1, \ldots, k-1$
- *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

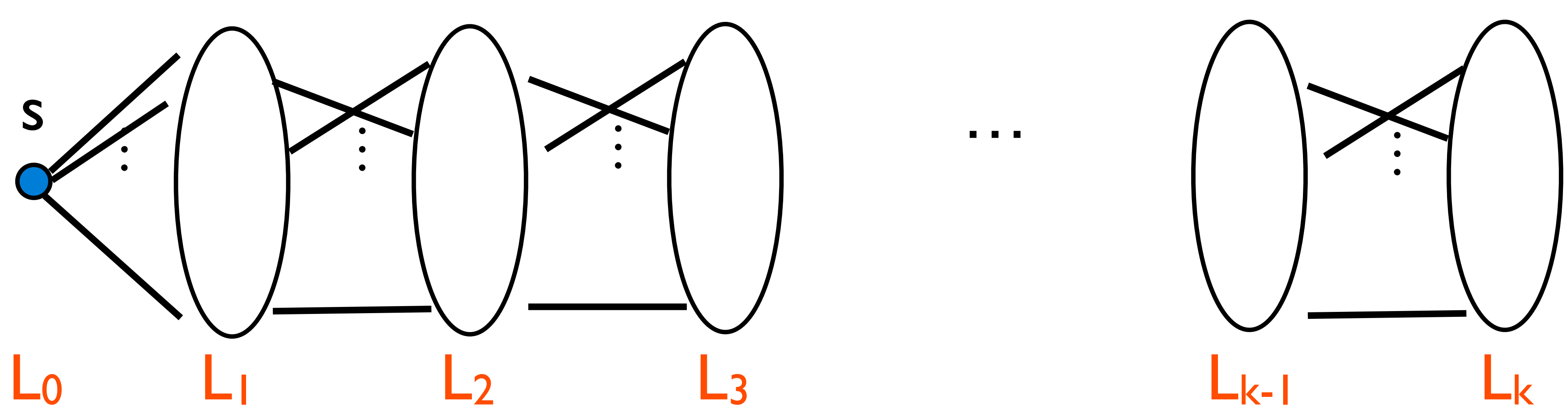
We say that G is a *tree* if:

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

Breadth 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, but not closer
- ⋮
- L_k = vertices that are k hops away from s, but not closer



Breadth first search

```
def BFS(G,s):

  layers = []                        // layer is an empty list
  current_layer = [s]                // current_layer is a list holding s
  next_layer = []
  for v in G:                        // v in G iterates over vertices
    seen[v] = false                  // seen is an associative map
  seen[s] = true
  while “current_layer not empty” :
    layers.append(current_layer)
    for u in current_layer:
        for v in G[u]:               // G[u] returns the neighbors of u
            if not seen[v]:
               next_layer.append(v)
               seen[v] = true
    current_layer = next_layer
    next_layer = []

  return layers
```

Properties of BFS

<u>Obs.</u>: Let G be a graph and s be a vertex in G. Suppose `BFS(G,s)` returns layers $L_0, L_1, \ldots, L_k$, then:

- 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

<u>Obs.</u>: Edges across layers must connect adjacent layers.

Complexity analysis

We would like to bound the running time of `BFS(G,s)` in terms of the size of G.

We need to decide the implementation details of the data structures used by the algorithm:

- list: `layers`, `current_layer`, `next_layer`
- associative map: `seen`
- graph: `G`

Lists

We need an implementation that supports

- iteration over the element in the list
- append at the end

If we use a linked list, we can

- iterate over the elements in $O(n)$ time, where n is the size
- append at the end in $O(1)$ time

If we use an array, we can

- iterate over the elements in $O(n)$ time, where n is the size
- append at the end in $O(1)$ time
- but we end up wasting a lot of space

Associative map

We need an iteration that supports

- setting/changing key-value pair
- lookup

If we use a Hash table, depending on the implementation

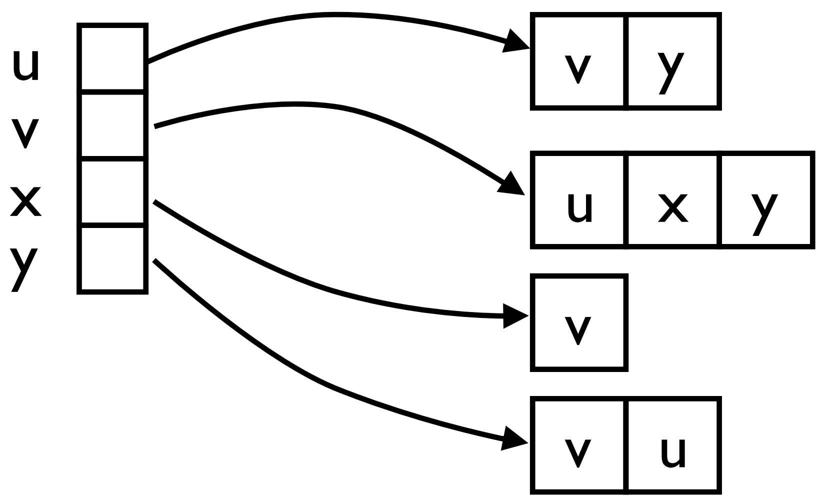
- lookup and setting/changing typically take **expected** $O(1)$ time
- building or growing the table can take more than linear time

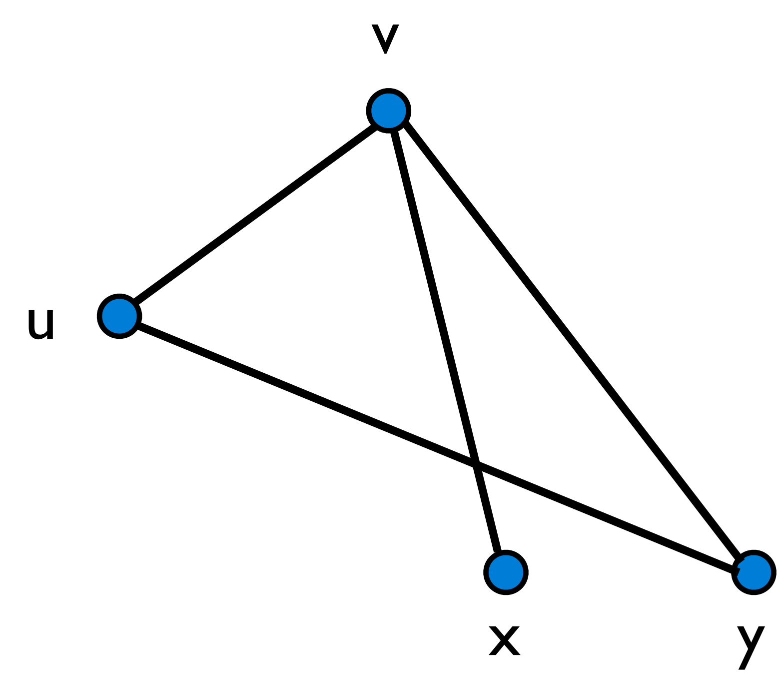
If the keys are $1, 2, \ldots, n$, then we can use array

- lookups and setting/changing in $O(1)$ time
- building the table in $O(1)$ time (depends on how memory is allocated)
- uses $\Omega(n)$ space, which is not an issue here

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 = []
  for v in G:
    seen[v] = false
  seen[s] = true
  while "current_layer not empty" :
    layers.append(current_layer)
    for u in current_layer:
        for v in G[u]:
            if not seen[v]:
               next_layer.append(v)
               seen[v] = true
    current_layer = next_layer
    next_layer = []

  return layers
```

this loop 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:

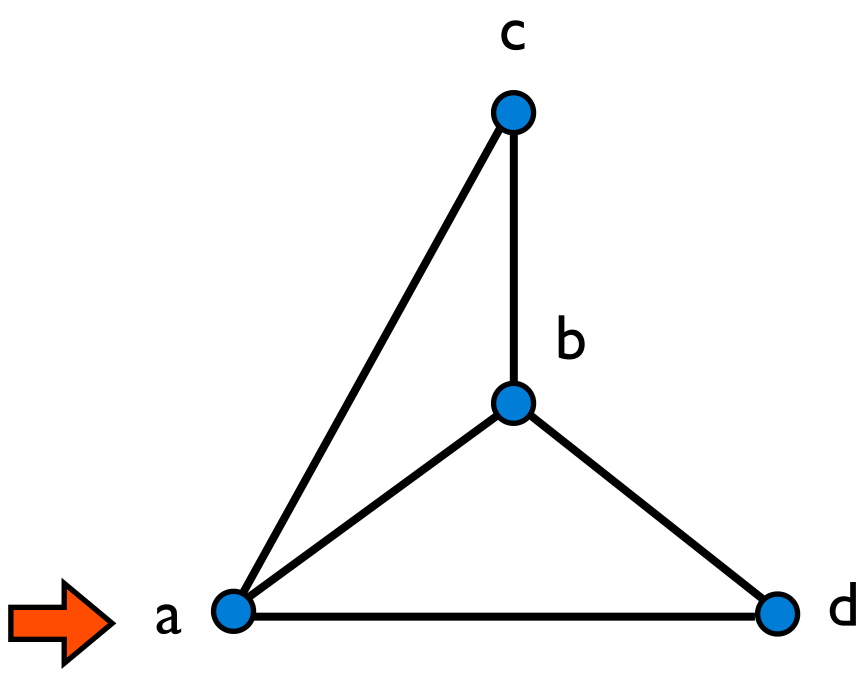
- discrete object encoding a relation between vertices
- representations: adjacency lists, adjacency matrix
- time complexity of basic primitives depends on representation

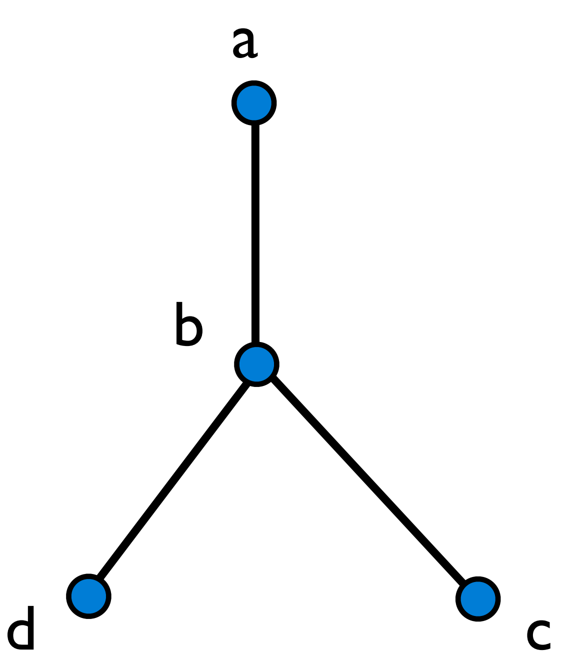
Breadth first search (BFS):

- 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 $O(n+m)$ 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".





DFS tree

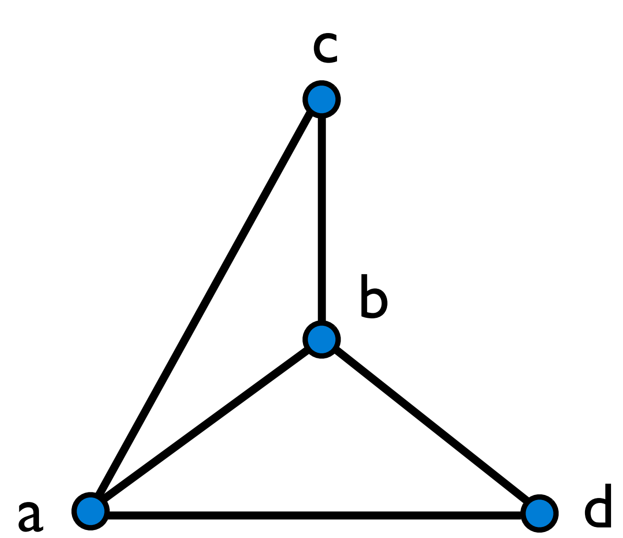
Depth first search

```
def DFS(G):

  for u in G
    visited[u] = false
    parent[u] = None
  time = 0
  for u in G:
    if not visited[u]:
      DFS_visit(u)
  return parent
```

```
def DFS_visit(u):

  visited[u] = true
  time = time + 1
  discovery[u] = time
  for v in G[u]:
    if not visited[v]:
      parent[v] = u
      DFS_visit(v)
  time = time + 1
  finish[u] = time
```



Time complexity of DFS

```
def DFS(G):
  for u in G
    visited[u] = false
    parent[u] = None
  time = 0
  for u in G:
    if not visited[u]:
      DFS_visit(u)
  return parent
```

```
def DFS_visit(u):
  visited[u] = true
  time = time + 1
  discovery[u] = time
  for v in G[u]:
    if not visited[v]:
      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

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

<u>Obs.</u>: 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.

<u>Obs.</u>: Each discovery and finish 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 G, run DFS to check if the new graph is still connected

A better algorithms runs in $O(nm)$:

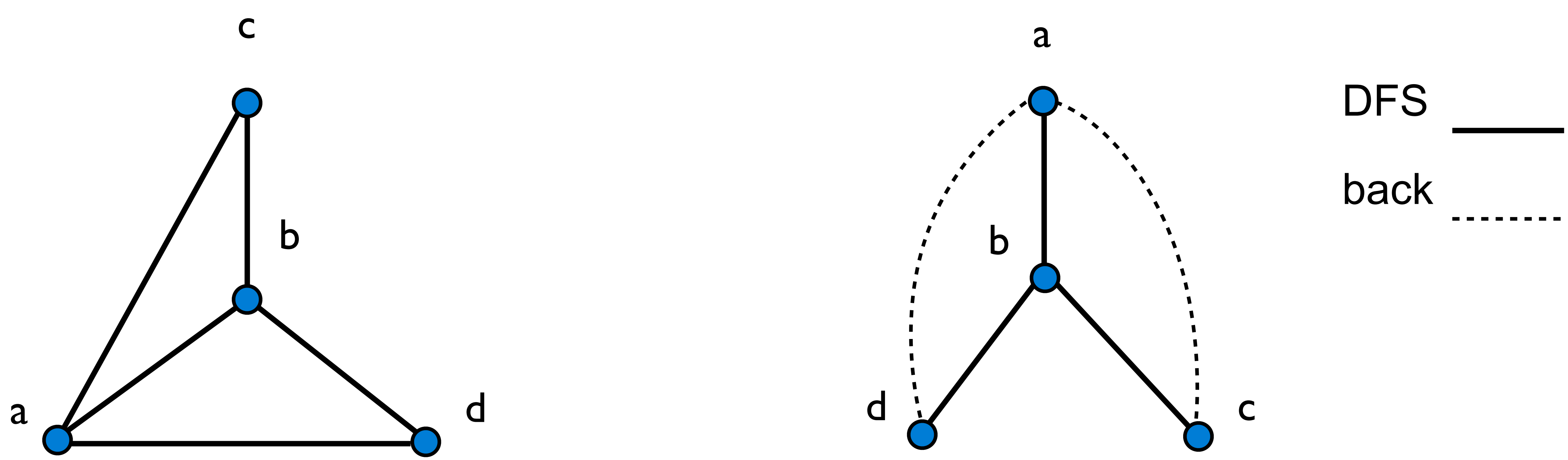
- run DFS in G
- for each DFS-tree edge (u,v) remove it from G, run DFS to check if the new graph is still connected

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

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

<u>Obs.</u>: 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

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

<u>Obs.</u>: If u is not the root of the DFS, u is a cut vertex it has a child v and no vertex in T_v (subtree rooted at v) can "jump over" u

THE UNIVERSITY OF SYDNEY

Cut vertices

Let u be an internal vertex in the DFS tree.

Def.: up[u] = min discovery[v] where $v \in N(u)$

Def.: down-&-up[u] = min up[v] where v is in T_u

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

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

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

Recap: DFS

Another graph exploration strategy that follows edges to new nodes until "stuck", then backtracks

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 $O(n+m)$ time using adjacency lists representation

This week

Quiz 1

- during your tutorial session

Tutorial Sheet 2:

- will be posted on Monday 3 August

Assignment 1:

- due on Monday 3 August

Assignment 2:

- out on Tuesday 4 August, due on Monday 10 August