COMP90038 Algorithms and Complexity

Tutorial 4 Graphs and Graph Traversal

Tutor: Chuang(Frank) Wang

THE UNIVERSITY OF MELBOURNE

1. Review

Graphs

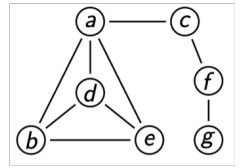
Graphs

Types

- Undirected
- Directed
- Unweighted
- Weighted
- Cyclic
- Acyclic Graph

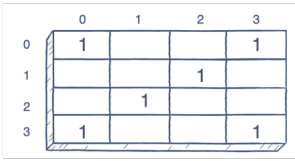
What

1. A data structure that consists of a finite set of nodes (or vertices) and a set of edges connecting them.
2. A pair (x,y) is referred to as an edge, which communicates that the x vertex connects to the y vertex.
3. Mathematic notation:
   - G = 〈 V , E 〉
   - V : Set of nodes or vertices
   - E: Set of edges (a binary relation on V)



Graph Representation

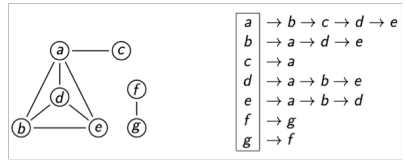
Adjacency Matrix



An Adjacency Matrix is a 2D array of size V x V where V is the number of nodes in a graph.

A slot matrix[i][j] = 1 indicates that there is an edge from node i to node j.

Adjacency List



(Assuming lists are kept in sorted order.)

Depth First Search (DFS)

Ideas:

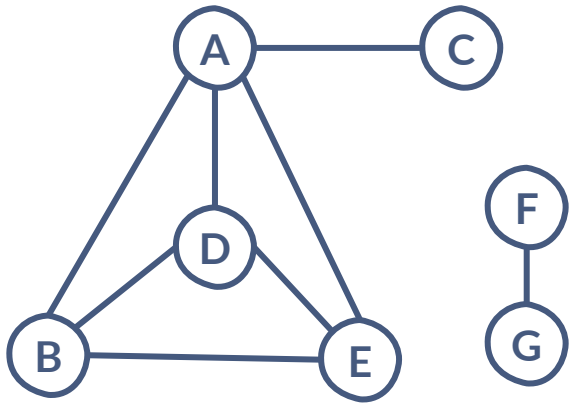
- To plunge depth first into a graph while there is any edge it can take next.
- It backtracks and continues if it cannot go any further.

When you are moving forward and there are no more nodes along the current path, you move back on the same path to find nodes to traverse.

```
function DFS(⟨V, E⟩)
    mark each node in V with 0
    count ← 0
    for each v in V do
        if v is marked with 0 then
            DfsExplore(v)

function DfsExplore(v)
    count ← count + 1
    mark v with count
    for each edge (v, w) do                ▷ w is v's neighbour
        if w is marked with 0 then
            DfsExplore(w)
```

DFS(cont.)



A → B → C → D → E

B → A → D → E

C → A

D → A → B → E

E → A → B → D

F → G

G → F

Traversal stack

- to track the nodes that have been visited already
- to remember the next node to traverse, when a dead end occurs in any iteration(backtrack).

Steps

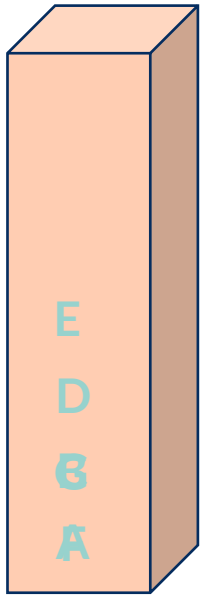
Step1:

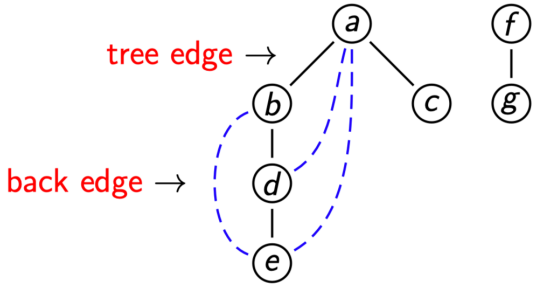
- Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.
- If no adjacent vertex is found, pop up a vertex from the stack.

Step2:

- Repeat step1 until all nodes have been visited & the stack is empty.

Traversal stack





$e_{4,1}$

$d_{3,2}$

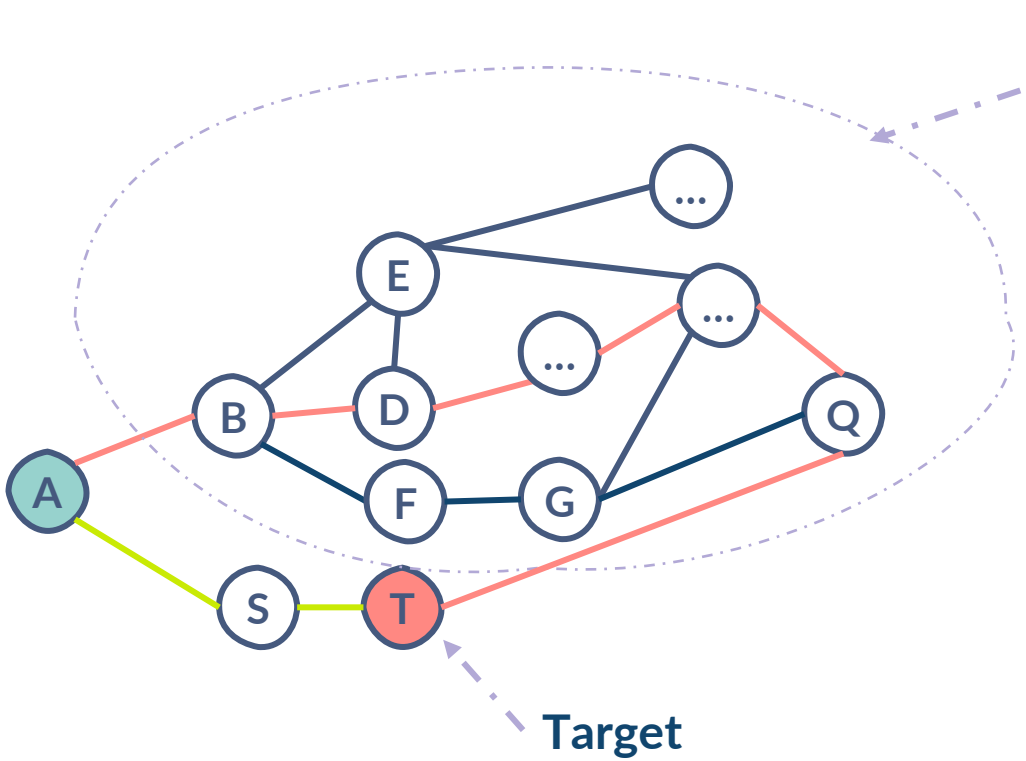
$b_{2,3}$ $c_{5,4}$ $g_{7,6}$

$a_{1,5}$ $f_{6,7}$

The first subscripts give the order in which nodes are pushed, the second the order in which they are popped off the stack.

DFS(cont.)

Downside of DFS



DFS cannot guarantee the shortest path to the target on unweighted graphs!

Red path cost: 6+
Green path cost: 2

Breadth First Search (BFS)

Ideas:

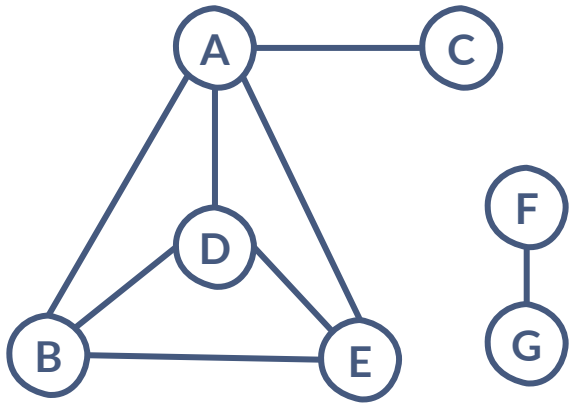
- BFS starts at a start node of a graph and explores the neighbour nodes first, before moving to the next level neighbours

Queue(FIFO):

- The BFS algorithm uses a queue data structure to track. which node to visit next.
- Upon reaching a new node, the algorithm adds it to the queue to visit it later.

```
function BFS(⟨V, E⟩)
    mark each node in V with 0
    count ← 0, init(queue)                    ▷ create an empty queue
    for each v in V do
        if v is marked with 0 then
            count ← count + 1
            mark v with count
            inject(queue, v)                  ▷ queue containing just v
            while queue is non-empty do
                u ← eject(queue)              ▷ dequeues u
                for each edge (u, w) do       ▷ w is u's neighbour
                    if w is marked with 0 then
                        count ← count + 1
                        mark w with count
                        inject(queue, w)      ▷ enqueues w
```

Breadth First Search (BFS)



Steps

1) The queue is initialized with the traversal's starting vertex, which is marked as visited.
2) Traverse
   - On each iteration, the algorithm identifies all unvisited vertices that are adjacent to the front vertex, marks them as visited, and adds them to the queue.
   - After that, the front vertex is removed from the queue.
3) Repeat step2 until all nodes have been visited & the queue is empty.

A → B → C → D → E

B → A → D → E

C → A

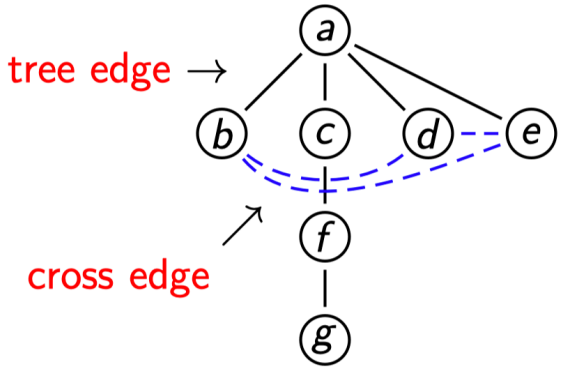
D → A → B → E

E → A → B → D

F → G

G → F

A B C D E F G



2. Tutorial Questions

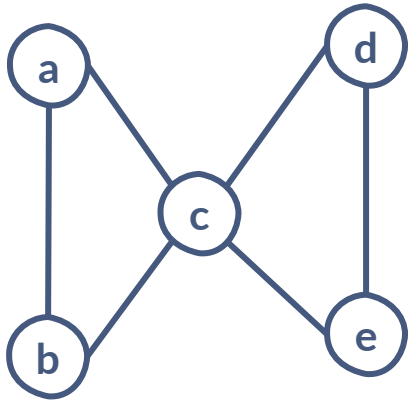
Question 27

27. Draw the undirected graph whose adjacency matrix is

	a	b	c	d	e
a	0	1	1	0	0
b	1	0	1	0	0
c	1	1	0	1	1
d	0	0	1	0	1
e	0	0	1	1	0

Starting at node a, traverse the graph by depth-first search, resolving ties by taking nodes in alphabetical order.

Solution:



a → b → c

b → a → c

c → a → b → d → e

d → c → e

e → c → d

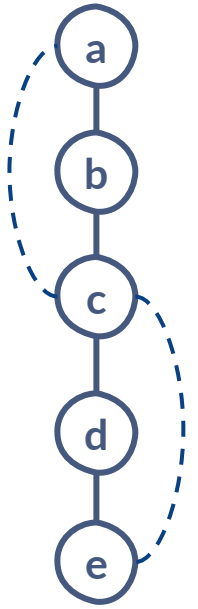
e5 1

d4 2

c3 3

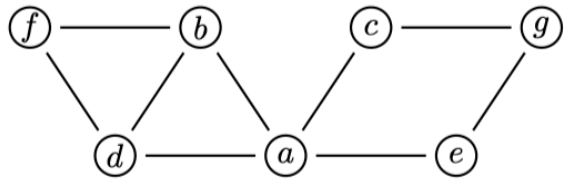
b2 4

a1 5



Question 28

28. Consider this graph:



(a) Write down the adjacency matrix representation for this graph, as well as the adjacency list representation (assume nodes are kept in alphabetical order in the lists).

(b) Starting at node a, traverse the graph by depth-first search, resolving ties by taking nodes in alphabetical order. Along the way, construct the depth-first search tree. Give the order in which nodes are pushed onto to traversal stack, and the order in which they are popped off.

(c) Traverse the graph by breadth-first search instead. Along the way, construct the breadth-first search tree.

Solution:

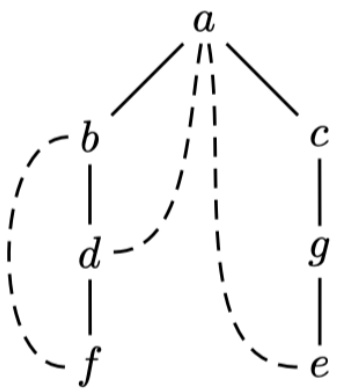
(a) Graph representation

	a	b	c	d	e	f	g
a	0	1	1	1	1	0	0
b	1	0	0	1	0	1	0
c	1	0	0	0	0	0	1
d	1	1	0	0	0	1	0
e	1	0	0	0	0	0	1
f	0	1	0	1	0	0	0
g	0	0	1	0	1	0	0

a	$\to b \to c \to d \to e$
b	$\to a \to d \to f$
c	$\to a \to g$
d	$\to a \to b \to f$
e	$\to a \to g$
f	$\to b \to d$
g	$\to c \to e$

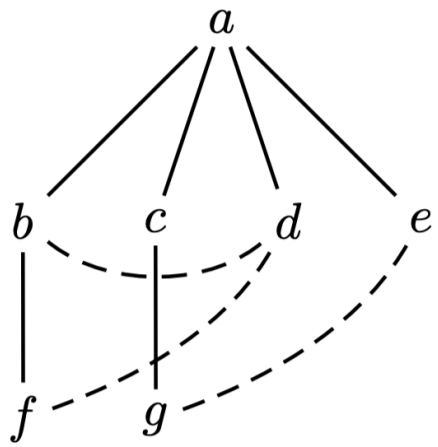
(b) DFS

f4	1	e7	4
d3	2	g6	5
b2	3	c5	6
a1	7		



(c) BFS

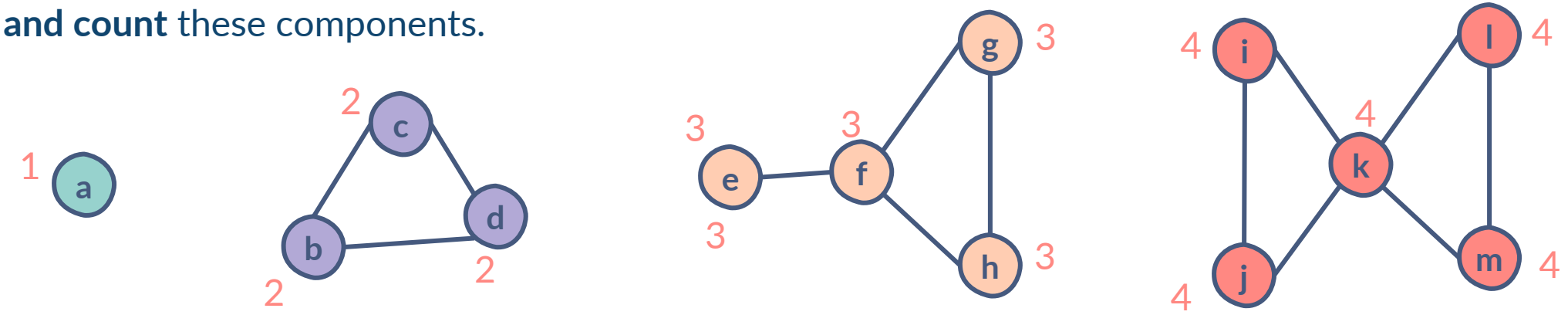
a b c d e f g



Question 29

29. Explain how one can use depth-first search to identify the connected components of an undirected graph. Hint: Number the components from 1 and mark each node with its component number.

Solution: Connected components: Sometimes a graph is split into multiple components. It's useful to **identify and count** these components.



Idea – Use a DFS to identify components

- Assign a unique integer to each group to tell that they are different.
- Mark all nodes in V with 0.
- Start a DFS at every node(except if it's already been visited) and mark all **reachable** nodes as being part of the same component and so on for other components

mark each node in V with 0 (indicates not yet visited)
component $\leftarrow 1$
for each v in V **do**
  if v is marked with 0 **then**
    DFS(v)
    component $\leftarrow$ *component* $+ 1$

function DFS(v)
  mark v with *component*
  for each vertex w adjacent to v **do**
    if w is marked with 0 **then**
      DFS(w)

Question 30

30. The function CYCLIC is intended to check whether a given undirected graph is cyclic.

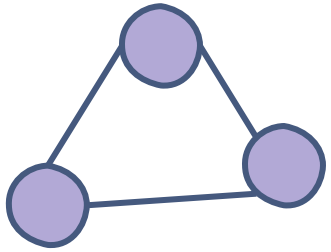
```
function CYCLIC(⟨V, E⟩)
    mark each node in V with 0
    count ← 0
    for each v in V do
        if v is marked with 0 then
            cyclic ← HASCYCLES(v)
            if cyclic then
                return True
    return False

function HASCYCLES(v)
    count ← count + 1
    mark v with count
    for each edge (v, w) do                    ▷ w is v's neighbour
        if w's mark is greater than 0 then
            return True                        ▷ w has been visited before
        if HASCYCLES(w) then
            return True                        ▷ a cycle can be reached from w
    return False
```

Show, through a worked example, that the algorithm is incorrect. A later exercise will ask you to develop a correct algorithm for this problem.

Solution:

Acyclic



Cyclic

$$V = \{a, b\}$$
$$E = \{(a, b), (b, a)\}$$

Question 23

23. Consider the *subset-sum problem*: Given a set S of positive integers, and a positive integer t, find a subset $S' \subseteq S$ such that $\sum S' = t$, or determine that there is no such subset. Design an exhaustive-search algorithm to solve this problem. Assuming that addition is a constant-time operation, what is the complexity of your algorithm?

Solution:

size: 2^n

```
function SubsetSum(S, t)
    for each S' in PowerSet(S) do
        if ∑S' = t then
            return True
    return False
```

$O(n)$

$O(2^n * n)$
because for each subset, we need O(n) to solve $\sum S'$

```
function PowerSet(S)
    if s = ∅ then
        return {∅}
    else
        x ← some element of S
        S' ← S \ {x}
        P ← PowerSet(S')
        return P ∪ {s ∪ {x} | s ∈ P}
```

$O(2^n)$

$S = \{a, b\}$

$x = a$

$S' = \{b\}$

$x = b$

$S' = \emptyset$

$return\ \{\emptyset\}$

$P = \{\emptyset\}$

$return\ \{\emptyset, \{b\}\}$

$P = \{\emptyset, \{b\}\}$

$return\ P \cup \{s \cup \{a\} | s \in P\}$

$\{\emptyset, \{b\}, \{a\}, \{a, b\}\}$

Question 24

24. Consider the *partition problem*: Given n positive integers, partition them into two disjoint subsets so that the sum of one subset is the same as the sum of the other, or determine that no such partition exists. Designing an exhaustive-search algorithm to solve this problem seems somewhat harder than doing the same for the subset-sum problem. Show, however, that there is a simple way of exploiting your algorithm for the subset-sum problem (that is, try to *reduce* the partition problem to the subset-sum problem).

Solution:

```
function HasPartition(S)
    sum ← ∑ S
    if sum is odd then
        return False
    return SubsetSum(S, sum/2)
```

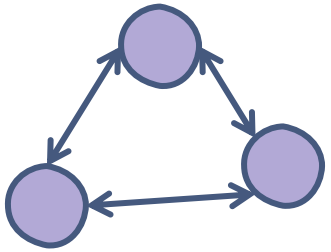
```
function SubsetSum(S, t)
    for each S' in Powerset(S) do
        if ∑ S' = t then
            return True
    return False
```

```
function Powerset(S)
    if s = ∅ then
        return {∅}
    else
        x ← some element of S
        S' ← S \ {x}
        P ← Powerset(S')
        return P ∪ {s ∪ {x} | s ∈ P}
```

Question 25

25. Consider the *clique problem*: Given a graph G and a positive integer k, determine whether the graph contains a *clique* of size k, that is, G has a complete sub-graph with k nodes. Design an exhaustive-search algorithm to solve this problem.

What is a clique?



Clique of size 3

- In a clique of size k, the out-degree for each of k nodes is k-1.
- Out-degree: number of edges going from a node.
- In-degree: number of edges going to a node.

Answer: Assume $G = (V, E)$ where V is the set of nodes, and E is the set of edges.

```
function HasClique((V, E), k)
    for each U ⊆ V with |U| = k do
        if IsClique(U, E) then
            return True
    return False

function IsClique(U, E)
    for each u ∈ U do
        for each v ∈ U do
            if u ≠ v and (u, v) ∉ E then
                return False
    return True
```

THE UNIVERSITY OF MELBOURNE

Next Week:

Decrease and conquer

Thank you !