CSE 101, Winter 2018

Discussion Section
Week 1

January 8 - January 15

Important

- Annotations were added (post-lecture) to the tablet slides, to fill in a few gaps (Lecture 1)
- Look through **Additional Resources** (more practice problems and reading material) at the end of the website:

  http://vlsicad.ucsd.edu/courses/cse101-w18/

- Closed-form - is a solution to a recurrence relation

Review of Graphs

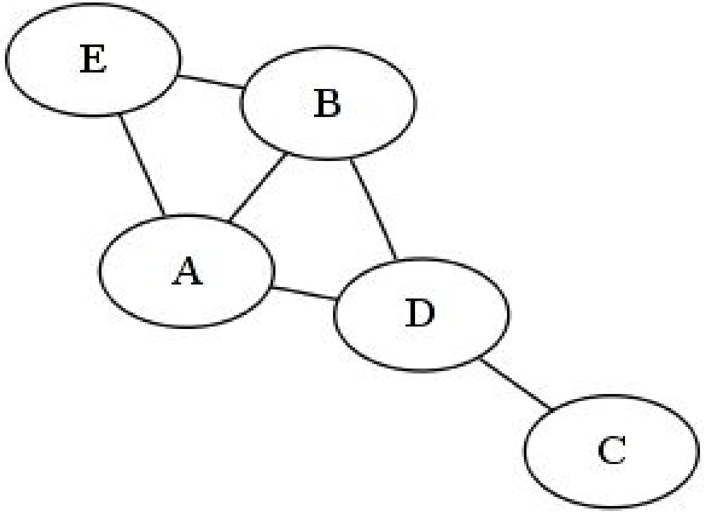
Graph is a pair of finite sets $G = (V, E)$, where:

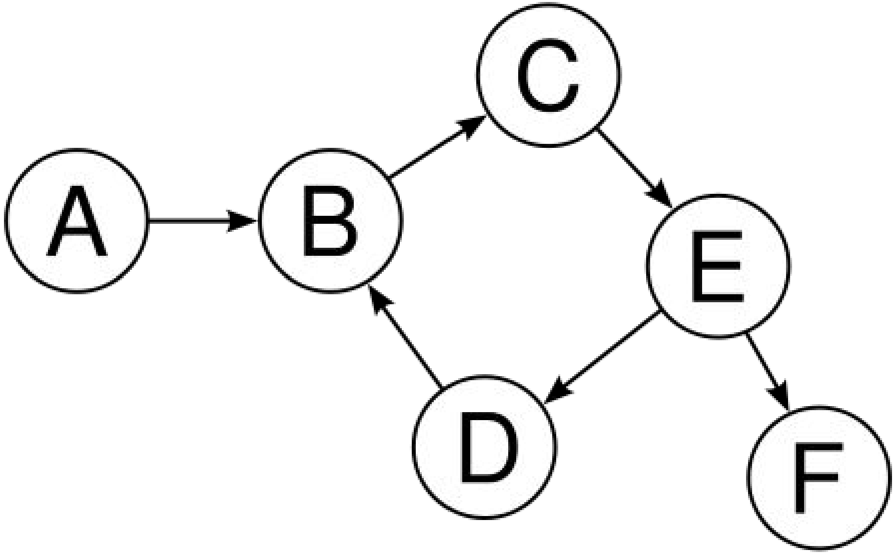
V is a set of vertices, $V = \{v_1, v_2, \dots ,v_n\}$

E is a set of edges, $E = \{(v, u) \mid v, u \text{ belong to } V\}$

If edges (v, u) are unordered, then graph G is called an **undirected graph**, otherwise G is called a **directed graph**.

Review of Graphs





Undirected Graph
V = {A, B, C, D, E}
E = {(A,B), (A,D), (A,E), (B,D), (B, E), (C, D)}

Directed Graph
V = {A, B, C, D, E, F}
E = {(A,B), (B, C), (C, E), (E,D), (D, B), (E, F)}

Review of Graphs

The number of vertices in a graph G = (V, E) is usually denoted by n, i.e. |V| = n.

The number of edges in a graph G = (V, E) is usually denoted by m, i.e. |E| = m

The vertices and edges of a graph may be labelled in different ways:

- Vertices with numbers: V = {1, 2, … , n}
- Vertices with letters: V = {a, b, … } or V = {A, B, … }
- Edges with letters: E = {e_1, e_2, … , e_m}, where e_i is a pair of vertices (v_i, u_j).
- General labelling: V = {v_1, v_2, … , v_n}, E = {(v_i, v_j) | v_i, v_j are both in V}

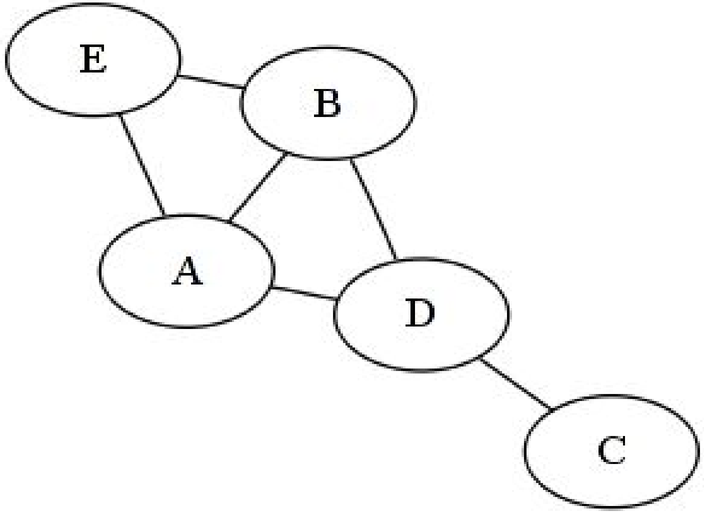
Review of Graphs

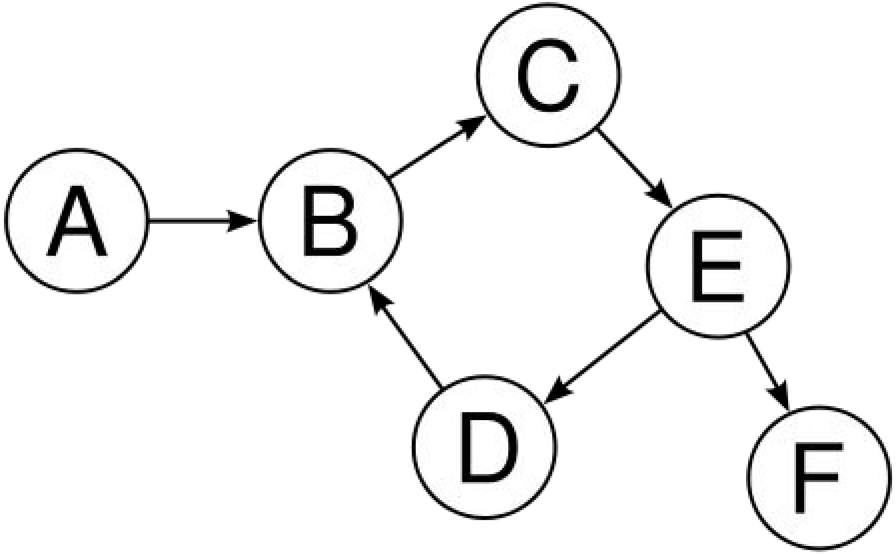
If e = (u, v) is an edge of graph G = (u, v), i.e. (u, v) belongs to E, then:

- Vertices u and v are called **endpoints** of the edge e = (u, v)
- Vertices u and v are **adjacent** (in other words, any two vertices connected with an edge are adjacent)
- Vertices u, v and edge e = (u, v) are **incident**
- If u = v, then edge e = (u, u) is called a self-loop
- Undirected graphs without self-loops are called **simple graphs** (we will assume simple graphs when we say graphs in the future)

If two edges e_1 and e_2 share endpoints, then these edges are also adjacent.

Review of Graphs





Endpoints of e=(A, B): A and B
A, B are adjacent
A, B are incident with e=(A, B)
There are not self-loops

A is incident to e=(A, B), e=(A, B) is incident from A
B is incident from e=(A, B), e=(A, B) is incident to B
(A, B) and (B, D) are adjacent
(B,C) and (C, E) are adjacent

Review of Graphs

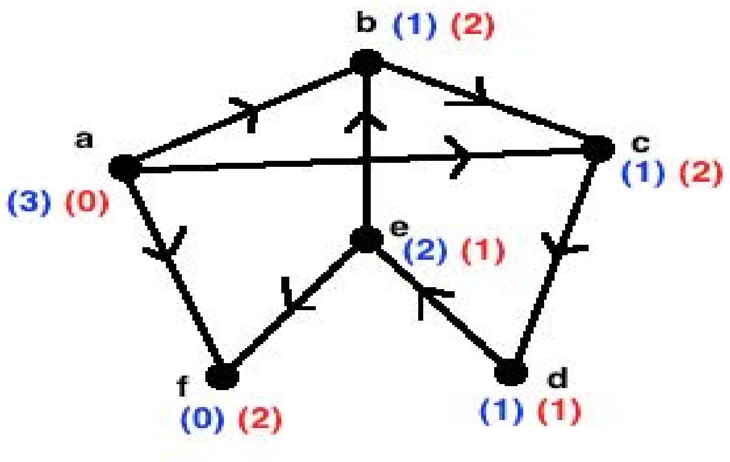
If G is an undirected graph (all pair of edges (u, v) are unordered). Then:

- The number of edges incident to vertex v, where v belongs to V, is called a **degree** of vertex v and denoted as deg(v). Self-loops are counted twice.

If G is a directed graph (all pair of edges (u, v) are ordered). Then:

- The number of all edges (v, x) is an **outdegree** of vertex v and the number of all edges (x, v) is an **indegree** of vertex v (outdegree - the number of edges that “point out” from a vertex, indegree - the number of edges that “point” to a vertex)

Review of Graphs



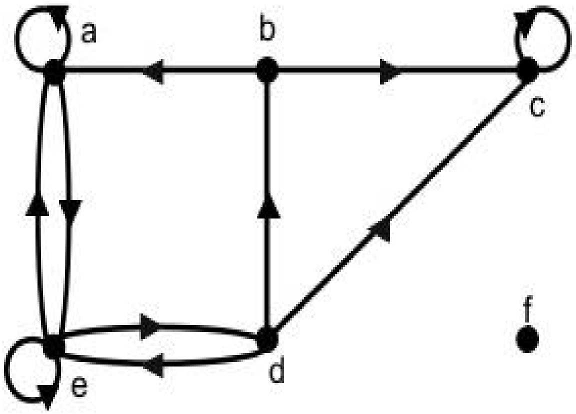
- Outdegree shown in blue.
- Indegree Shown in red.

Question: What would be the degree of each node if this was an undirected graph (i.e. the vertices and the edges remain the same, only the arrows are removed)?

Ans: Sum of indegree and outdegree in the original graph.

Review of Graphs

Q: What are the indegree and outdegree of each vertex?

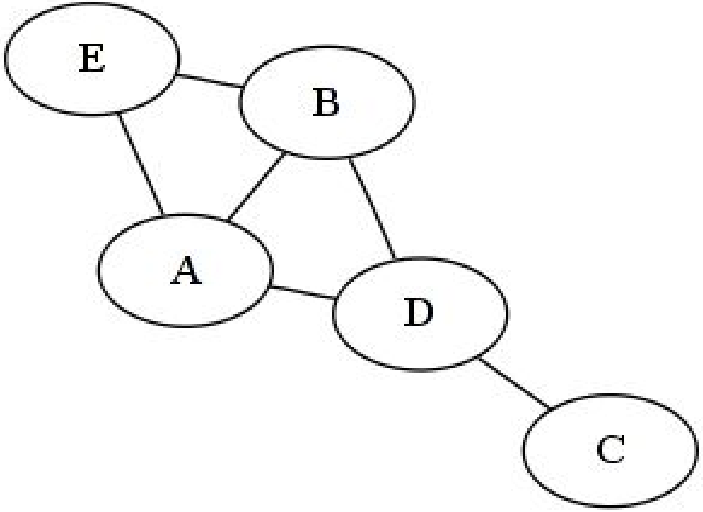


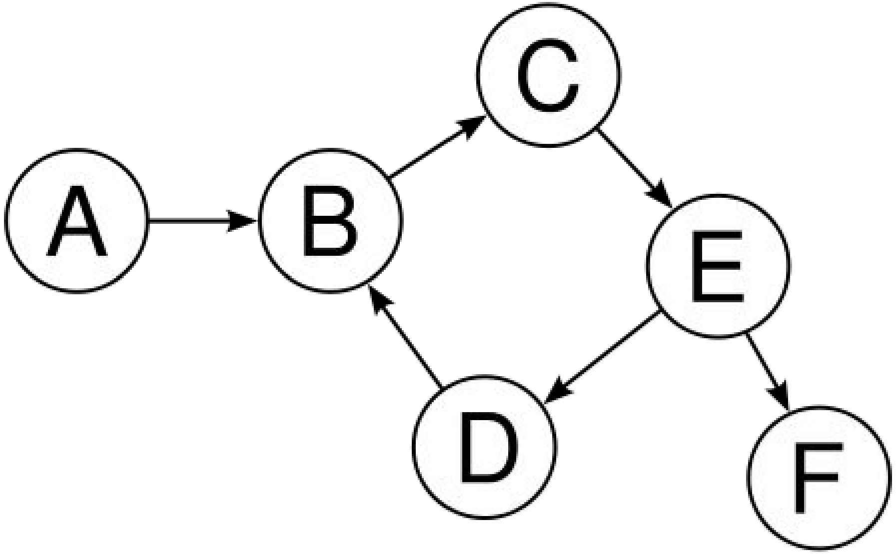
Vertex	Indegree	Outdegree
a	3	2
b	1	2
c	3	1
d	1	3
e	3	3
f	0	0

Review of Graphs

- A sequence of vertices $\{v_1, v_2, \dots, v_k\}$ is called a **path**, if all pairs (v_i, v_{i+1}) belong to E for all i: 1 <= i <= k - 1.
- A path is called a **simple path** if all vertices in this path are unique.
- Vertex v is said to be reachable from vertex u if there is a path from vertex v to vertex u.
- A **cycle** is a path starts and ends at the same vertex.
- The length of a path is the number of edges in this path.
- A graph without cycles called an **acyclic graph**

Review of Graphs





Path: (A, E, B, A, D, C)
Simple path: (A, E, B, D, C)

Path: (A, B, C, E, D, B, C)
Simple path: (A, B, C, E, D)

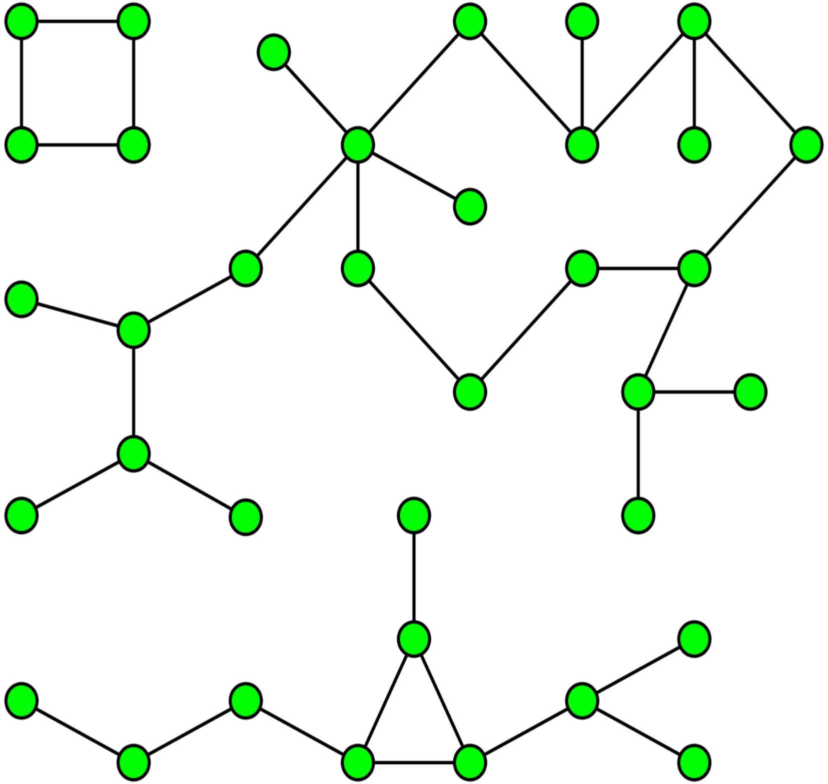
Review of Graphs

An undirected graph G = (V, E) is called **connected** if there is a path between any two vertices in this graph.

A maximal subset of nodes S in an undirected graph G = (V, E) is called a **connected component** if a subgraph of graph G that consists of vertices from S and all edges that have both endpoints in S is connected.

- If graph G is connected then V = S (S needs to be maximal)
- Two distinct connected components of a graph can’t intersect and there can’t be edges that have endpoints in different connected components.
- If graph G is not connected then V can be represented as a union of several connected components.

Review of Graphs



Q: How many connected components are there in this graph?

Ans: 3

Review of Graphs

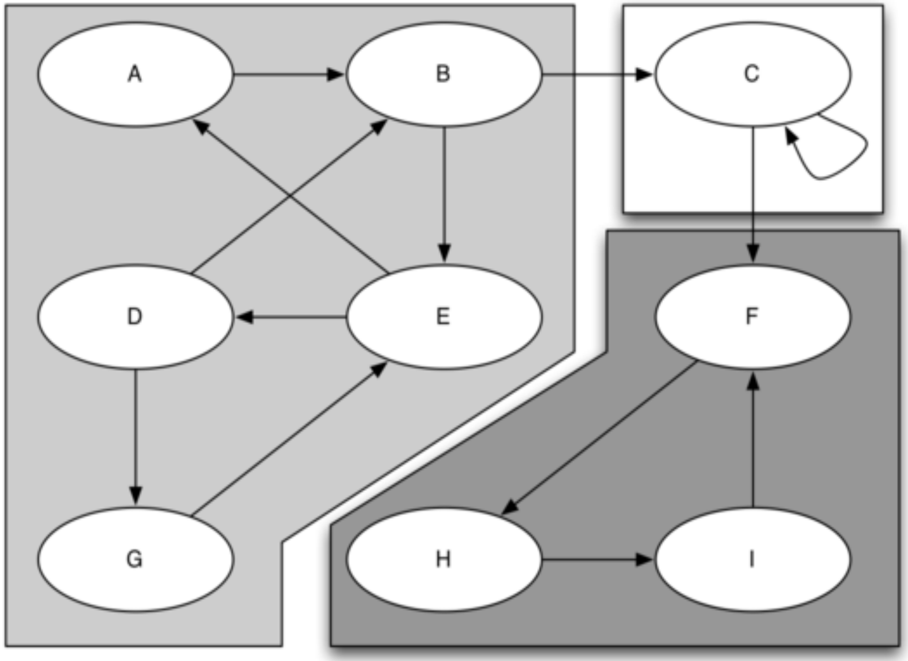
A directed graph is called **strongly connected** if for any pair of vertices (u, v) there is a path from vertex u to vertex v and there is a path from vertex v to vertex u.

A maximal subset of nodes S in a directed graph G = (V, E) is called a **strongly connected component** if a subgraph of graph G that consists of vertices from S and all edges that have both endpoints in S is strongly connected.

- If graph G is strongly connected then V = S (S needs to be maximal)
- Two distinct SCC can't intersect, but there can be edges that have endpoints in different SCC.
- If graph G is not connected then V can be represented as a union of several SCC.

*SCC: Strongly Connected Components

Review of Graphs



SCC:

1. A, B, E, D, G
2. C
3. F, I, H

Review of Asymptotic Notation

- Asymptotic Analysis:
    - Value or a curve a function f(n) approaches or becomes almost equal to when value of n is sufficiently large.
    - How fast does a function grow?
    - Describes long-term behaviour of functions.

- Application in Algorithms:
    - Analysing runtimes of the algorithms.
    - Interested in how time taken grows as the size of input grows.
    - Use asymptotic analysis to compare two algorithms.

Review of Asymptotic notation

We are given two functions $f: N \rightarrow R$, $g: N \rightarrow R$:

- We say that $f = O(g(n))$ if there exists constants $c > 0$ and $n_0 > 0$, such that $f(n) <= c * g(n)$ for all $n > n_0$
- We say that $f = \Omega(g(n))$ if there exists constants $c > 0$ and $n_0 > 0$, such that $f(n) >= c * g(n)$ for all $n > n_0$

- We say that $f = \Theta(g(n))$ if $f = O(g(n))$ and $f = \Omega(g(n))$

Note that: $f = O(g(n))$ if and only if $g = \Omega(f(n))$

Review of Asymptotic notation

Let $\lim_{n \to \infty} f(n) / g(n) = C$

- If $C < \infty$, then $f(n) = O(g(n))$
- If $C > 0$, then $f(n) = \Omega(g(n))$
- If $0 < C < \infty$, then $f(n) = \Theta(g(n))$

Example 2.1

```
for(int i = 0; i < n; i++)
    for(int k = n - 1; k >= 0; k--)
        cout << "CSE 101\n";
```

What is the time complexity of the algorithm?

Example 2.1

```
for(int i = 0; i < n; i++)
    for(int k = n - 1; k >= 0; k--)
        cout << “CSE 101\n”;
```

There are n iterations in the outer loop and there are n iterations in the inner loop.

Thus, the time complexity is $\Theta(n^2)$

Example 2.2

```
void f(n) {
    if(n == 0)    return;
    f(n - 1);
    f(n - 1);
}
```

What is the time complexity of the algorithm?

Example 2.2

```
void f(n) {
    if(n == 0)    return;
    f(n - 1);
    f(n - 1);
}
```

Let $T(n)$ be the number of operations.

Then

- $T(0) = c$
- $T(n) = 2 * T(n - 1),\ n > 0$

$T(n) = 2 * T(n - 1) = 2 * 2 * T(n - 2) = \dots = 2^k * T(n - k) = \dots = 2^n * T(0) = 2^n$

Thus $T(n) = \Theta(2^n)$

Example 2.3

```
for(int l = 0, r = 0, sum = 0; r < n; r++) {
    sum += a[r];
    while(sum > k && l <= r) {
        sum -= a[l];
        l++;
    }
}
// k is given
```

What is the time complexity of the algorithm?

Example 2.3

```
for(int l = 0, r = 0, sum = 0; r < n; r++) {
    sum += a[r];
    while(sum > k && l <= r) {
        sum -= a[l];
        l++;
    }
}
// k is given
```

Let $T(n)$ be the number of operations.

Let m_i be the number of steps the inner loop makes when $r = i$. At each step of the inner loop variable l increases by 1 and l can't be larger than r, which can't be larger than n.

Then:

$T(n) = \Theta(m_1) + \Theta(m_2) + \ldots + \Theta(m_n) + \Theta(n) = \Theta(m_1 + m_2 + \ldots + m_n) + \Theta(n) = \Theta(n) + \Theta(n) = \Theta(n)$

Example 2.4

```
for(int i = 1; i <= n; i++)
    for(int k = i; k <= n; k += i)
        cout << "CSE 101\n";
```

What is the time complexity of the algorithm?

Example 2.4

```
for(int i = 1; i <= n; i++)
    for(int k = i; k <= n; k += i)
        cout << "CSE 101\n";
```

Let $T(n)$ be the number of operations.

There are n steps in the outer loop.

There are n / i steps in the inner loop.

Then:

$T(n) = n + n / 2 + n / 3 + \ldots + n / (n-1) + n / n = n * (1 + 1 / 2 + 1 / 3 + \ldots + 1 / (n-1) + 1 / n) = \quad = n * \Theta(\ln(n))$

$T(n) = \Theta(n\log(n))$