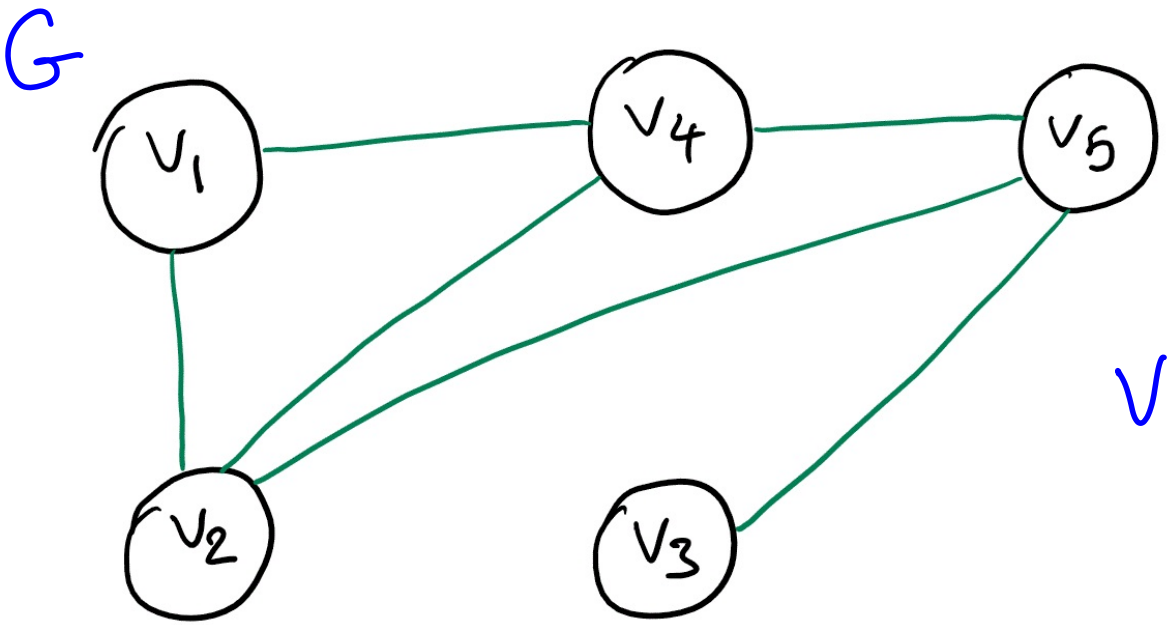
CSC263H
Data Structures and Analysis

Prof. Bahar Aameri & Prof. Marsha Chechik

Winter 2024 – Week 8

Graph ADT

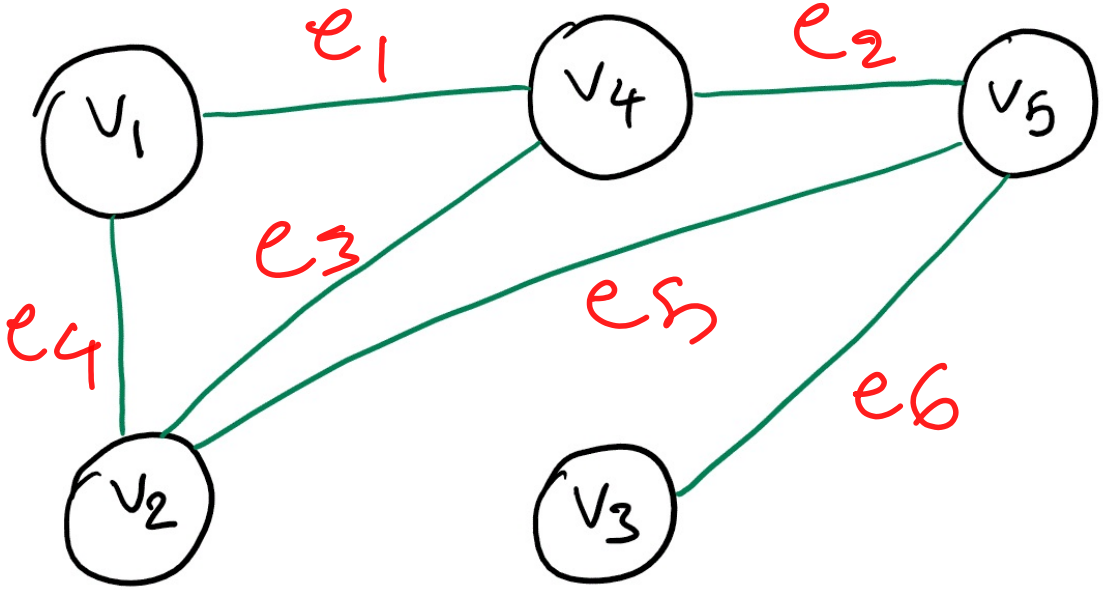
A graph is a tuple of two sets $G = \langle V, E \rangle$, where V is a set of **vertices**, and E is a set of **edges** between those vertices.
E itself is a set of tuples (v_i, v_j), where $v_i, v_j \in V$.



$G = \langle V, E \rangle$

$V = \{v_1, v_2, v_3, v_4, v_5\}$

$E = \{(v_1, v_2), (v_1, v_4), (v_4, v_5), \ldots\}$



$e_1 = (v_1, v_4)$

$e_4 = (v_1, v_2)$

$e_4 \in E$

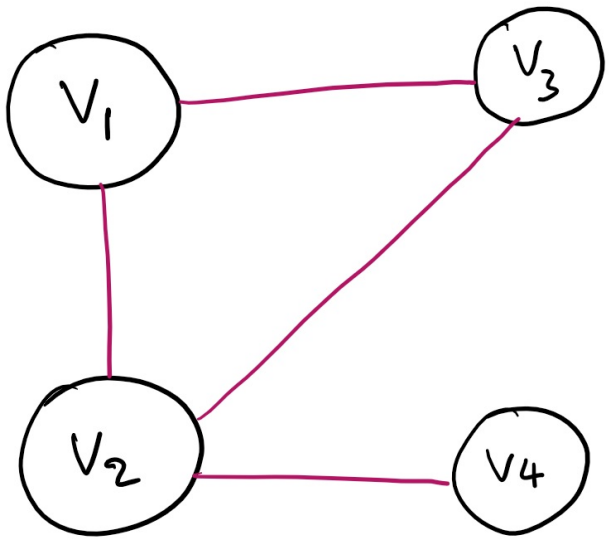
Applications of Graphs

- Network Traffic flow
- Social Webs
- Task Scheduling (Operating Systems)
- Maps and GPS
- VLSI and Chip Design

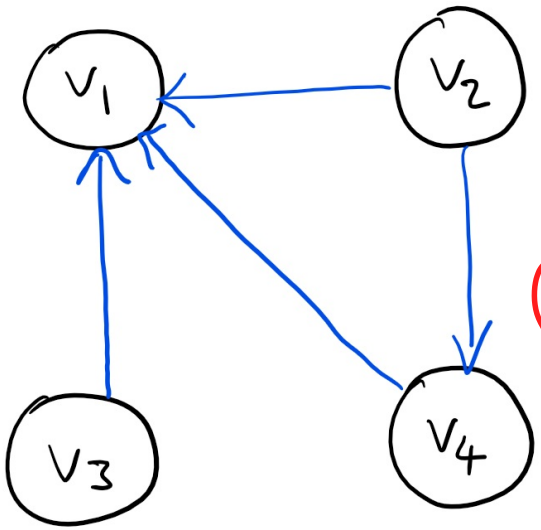
Graph ADT: Basic Definitions

A directed graph is a graph in which edges have orientations.
That is, the order of tuples matter: (v_i, v_j) and (v_j, v_i) represent **different** edges.

An undirected graph is a graph in which edges have no orientation. That is, the order of tuples does **not** matter: (v_i, v_j) is the same as (v_j, v_i).



$(v_i, v_j) = (v_j, v_i)$



$(v_4, v_1) \in E$

$(v_1, v_4) \notin E$

$(v_i, v_j) \neq (v_j, v_i)$

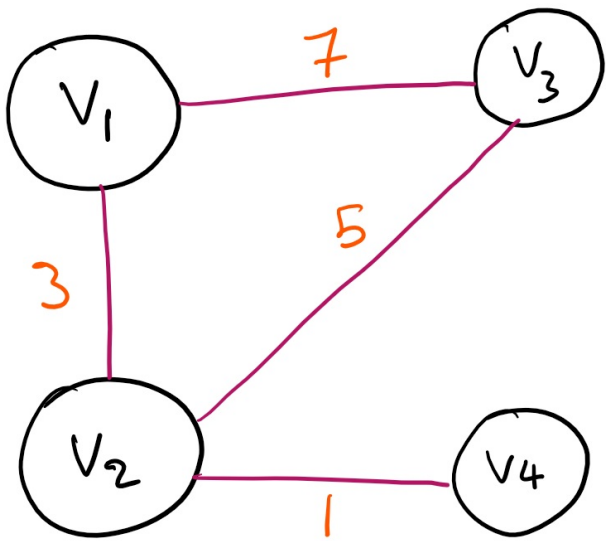
- Facebook friendship can be represented by undirected graphs:
  - Each person (account) is represented by a vertex;
  - If u and v are friends iff there exists an edge between u and v.
- Instagram followers can be represented by directed graphs.
  - Each person (account) is represented by a vertex;
  - If u follows v iff there exists an edge from u to v.

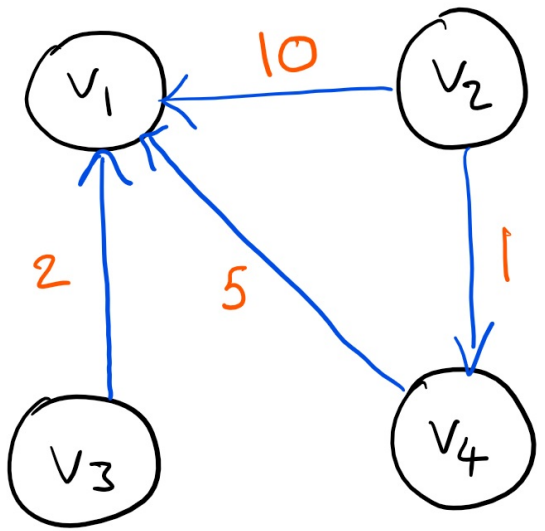
Note: Usually, if neither type of graph is specified, our default will be an undirected graph.

Graph ADT: Basic Definitions

A weighted graph is a graph in which a weight (which is a number) is assigned to each edge.

Note: Weights of edges might represent costs, lengths or capacities, depending on the problem at hand.





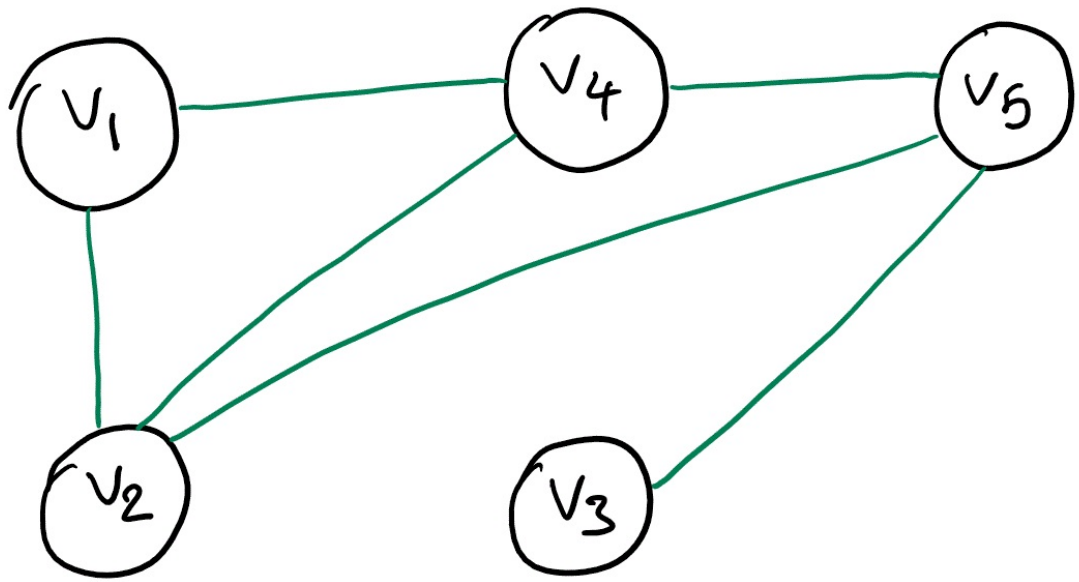
- Network traffic flow can be represented by weighted directed graphs:
  - Each hub is represented by a vertex;
  - Incoming traffic from a hub u to a hub v is represented by an incoming edge from u to v;
  - Outgoing traffic from a hub v to a hub z is represented by an outgoing edge from v to z;
  - Traffic flow is represented by the weight of each edge.

- Maps can be represented by weighted graphs:
  - Each address point is represented by a vertex;
  - Roads between address points are represented by edges;
  - The distance between two points is represented by the weight of the edge between the points.

Note: Our default is that a graph is NOT weighted, unless otherwise is specified.

Graph ADT: Basic Definitions

Two vertices are adjacent if there is an edge between them.
In this case, we also say that the two vertices are neighbours.



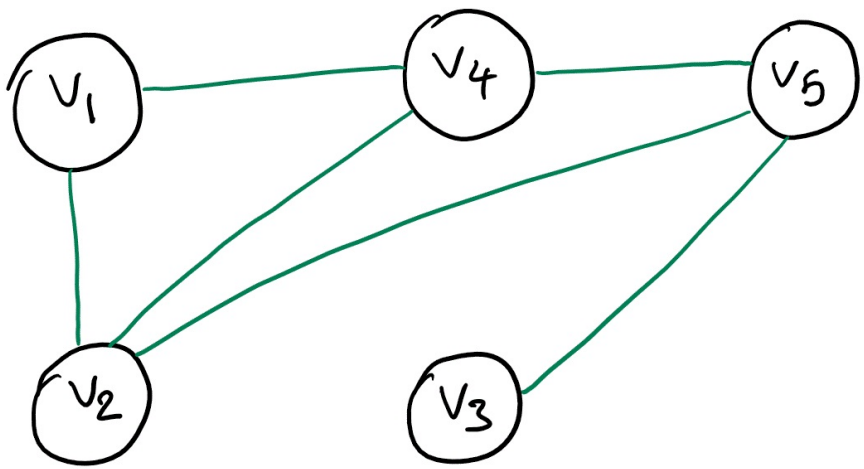
E.g.:

v_1 and v_4 are adjacent

Graph ADT: Basic Definitions

A path between vertices u and w is a sequence of distinct edges $(v_0, v_1), (v_1, v_2), ..., (v_{k-1}, v_k)$, where $u = v_0$ and $w = v_k$, and for all tuples (v_i, v_j) in the sequence, v_i and v_j are distinct.
The length of the path is the number of edges in the path.

Start at vertex u, take an edge to get to a new vertex v_1, take another edge to get to another new vertex v_2, and keep following edges until arriving at v_k.



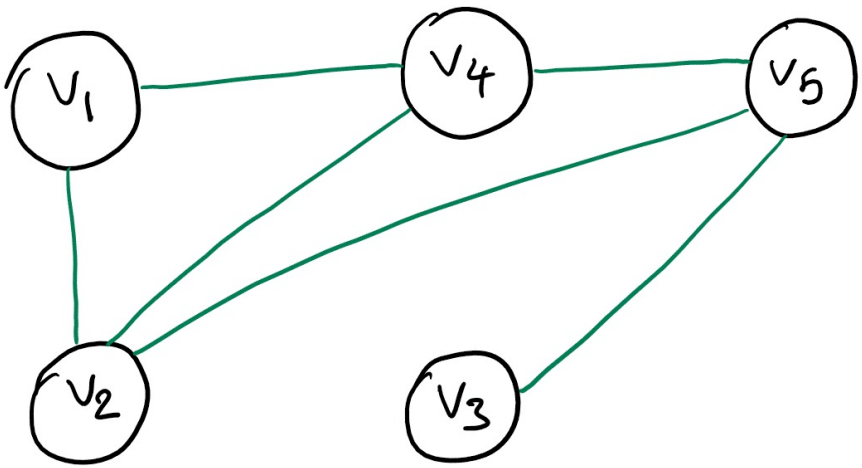
Path 1: (v_1, v_4)
length: 1

Path 2: $(v_1, v_2)(v_2, v_4)$
length: 2

Path 3: $(v_1, v_2)(v_2, v_5)(v_5, v_4)$
length: 3

Graph ADT: Basic Definitions

The distance between two vertices is the length of the **shortest path** between the two vertices. The distance between a **vertex and itself** is always 0.



distance of v_1 and v_4: 1

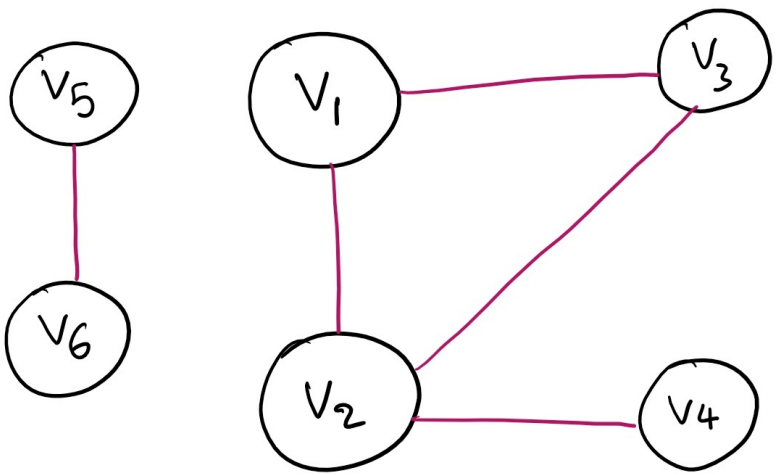
" " v_1 and v_5: 2

" " v_1 and v_3: 3

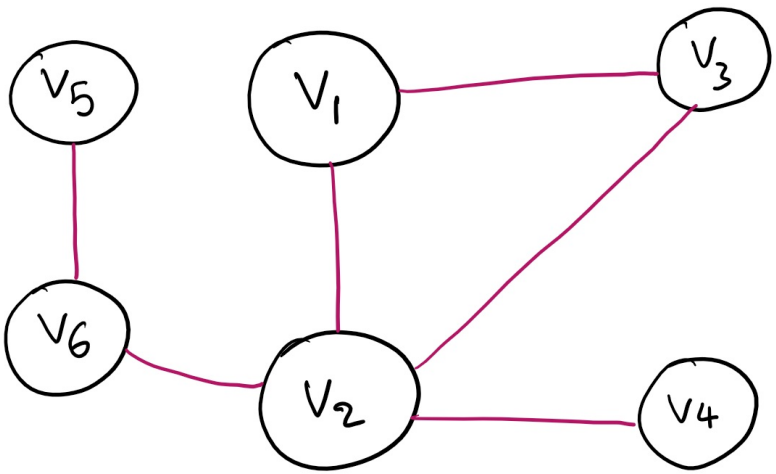
Graph ADT: Basic Definitions

A graph is connected if there is a **path** between **every pair** of vertices in the graph. Otherwise, the graph is disconnected.

G_1 ✗



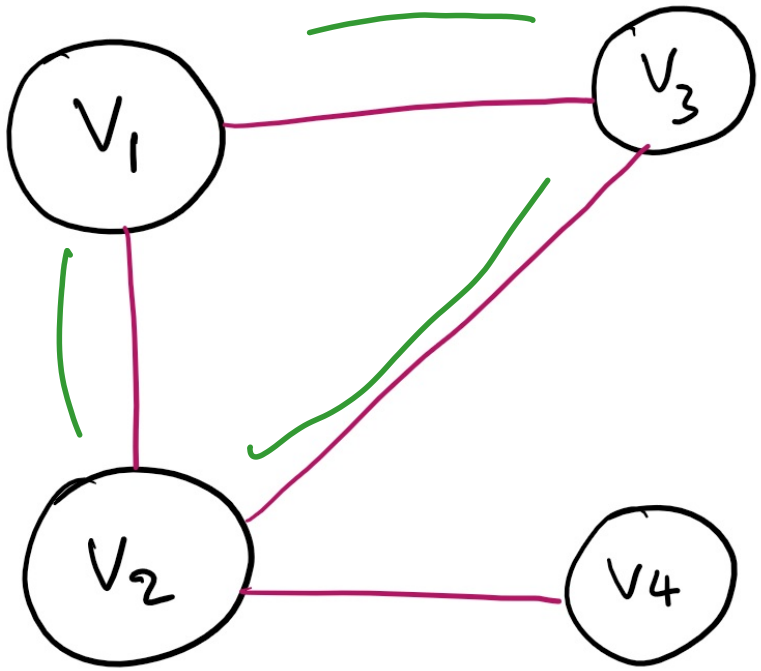
G_2 ✓



Minimum number of edges in an undirected **connected** graph with n vertices: $n-1$

Graph ADT: Basic Definitions

In a graph, a cycle is path from an vertex to **itself**.

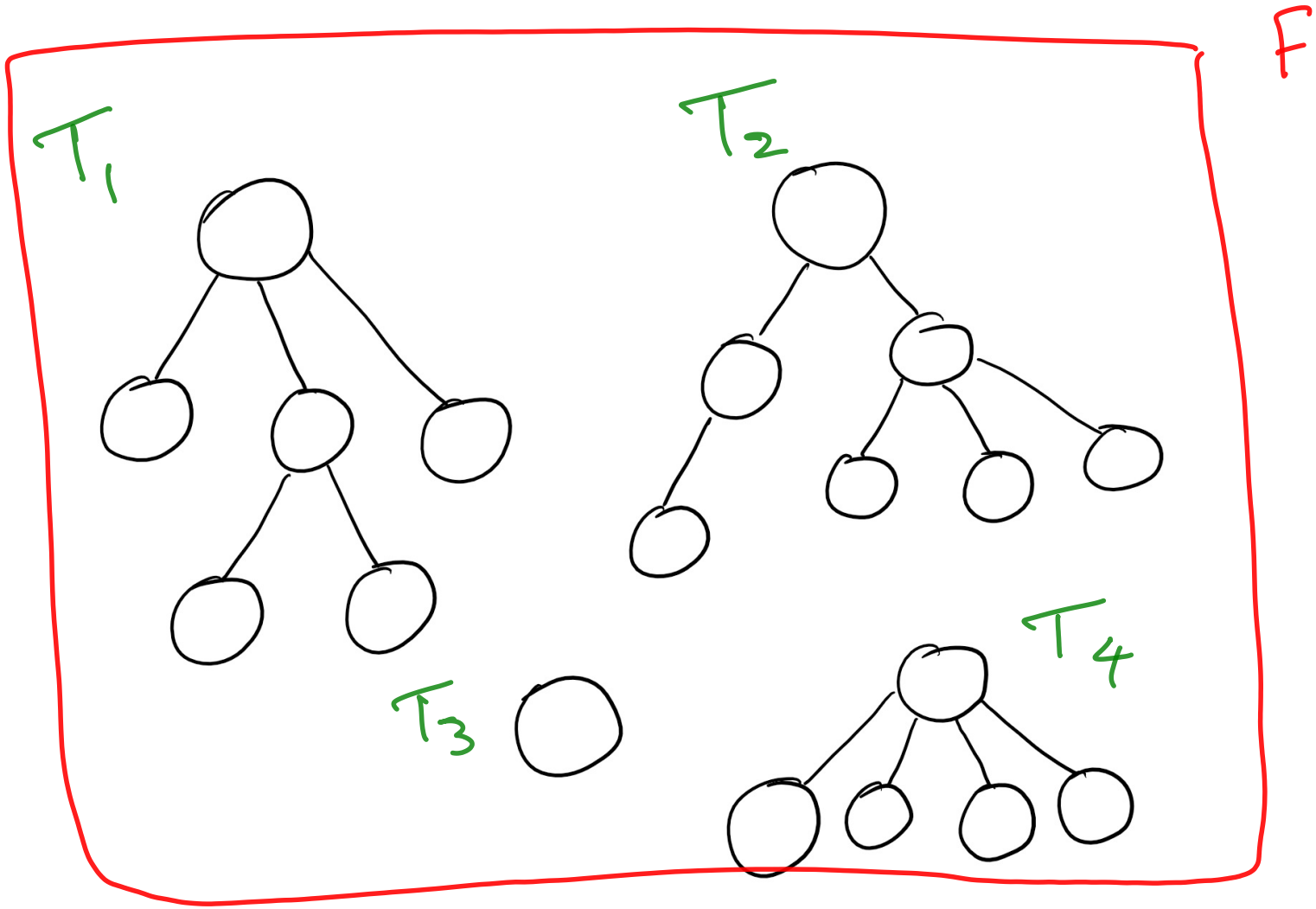


Assumption:

Length of a cycle is at least 3

Graph ADT: Basic Definitions

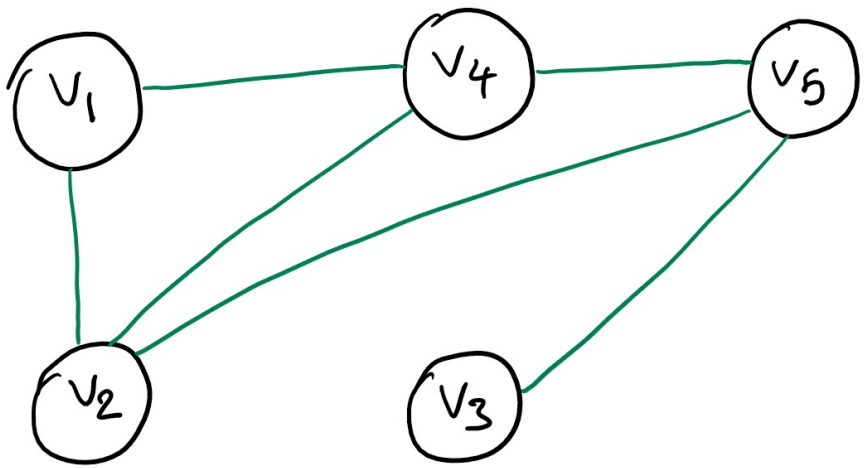
A tree is **connected** graph and contains **no cycle**.
A forest is a **collection** of **disjoint trees**.



Data Structures for Graphs: Adjacency Matrix

Let $V = \{v_1, v_2, ..., v_n\}$ ($|V| = n$).
Let A be a $n \times n$ matrix.

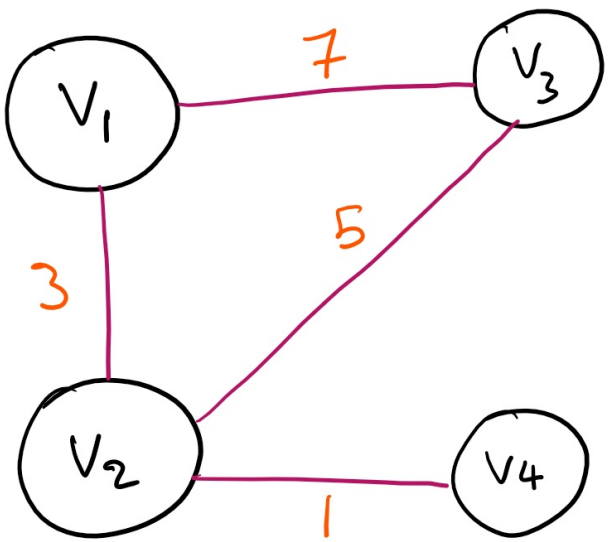
$$A[i,j] = \begin{cases} 1 & \text{if } (v_i, v_j) \in E \\ 0 & \text{otherwise} \end{cases}$$

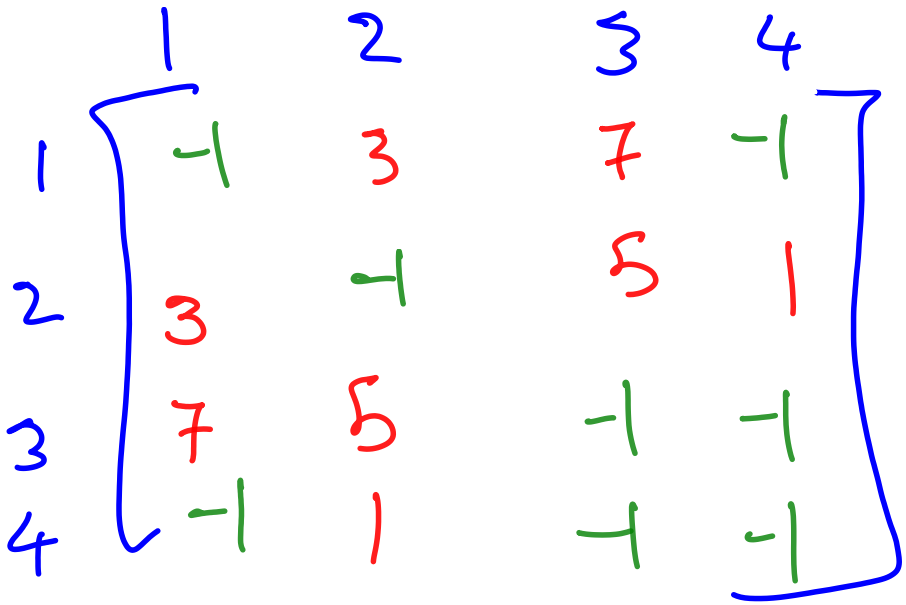


	1	2	3	4	5
1	0	1	0	1	0
2	1	0	0	1	1
3	0	0	0	0	1
4	1	1	0	0	1
5	0	1	1	1	0

For an undirected graph, adjacency matrix is **symmetric**: $A[i,j] = A[j,i]$.

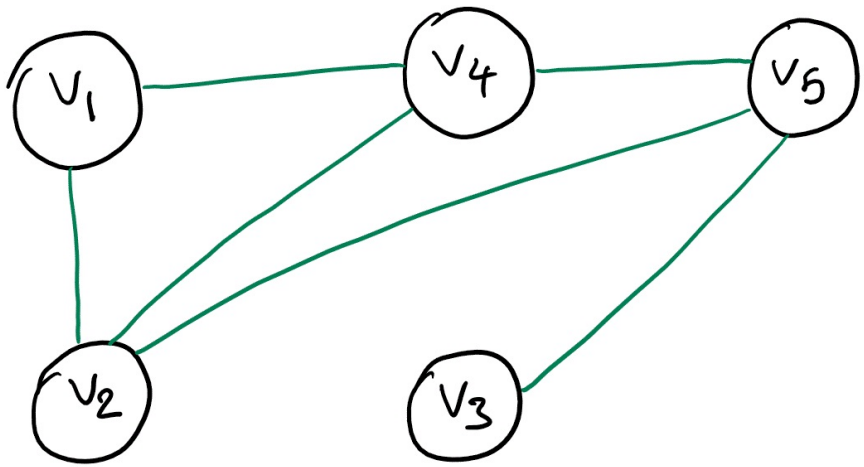
For a **weighted** graph:
If $(v_i, v_j) \in E$, store $w(v_i, v_j)$ in $A[i, j]$ ($w(v_i, v_j)$ denotes the weight of the edge (v_i, v_j));
If $(v_i, v_j) \notin E$, store -1/0/∞ (depending on application).

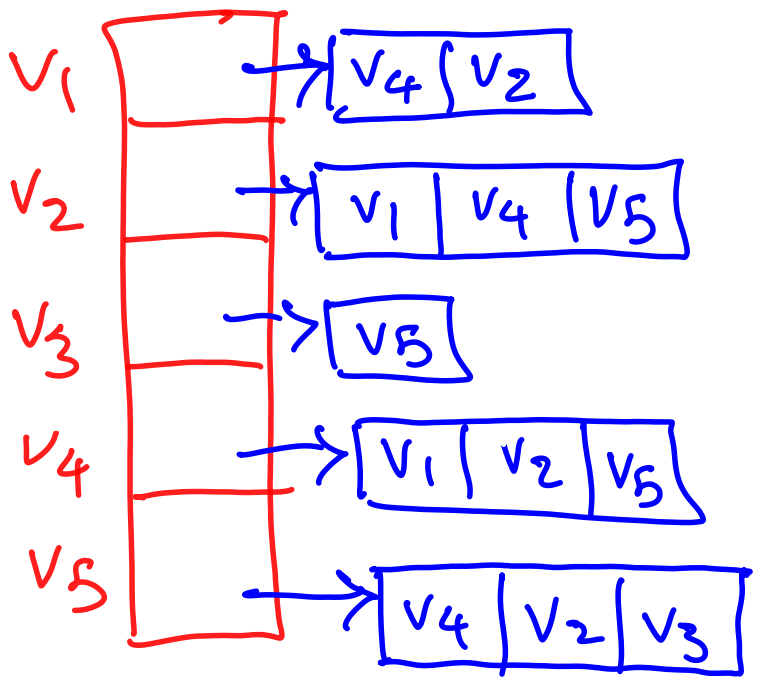




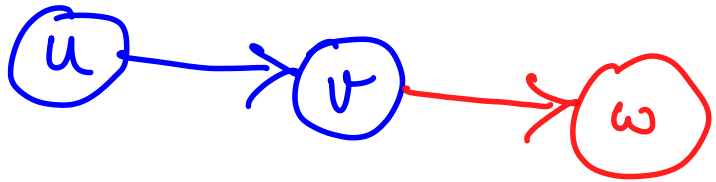
Data Structures for Graphs: Adjacency Lists

Let $V = \{v_1, v_2, ..., v_n\}$ ($|V| = n$).
Let L be a list of size n.
Each position $L[i]$ corresponds with a vertex v_i and stores a list A_i of all vertices that have an edge from v_i:
A_i includes v_j iff $(v_i, v_j) \in E$





For an undirected graph, edge (u, v) is stored **twice** (u in sub-list corresponding with v and v in sub-list corresponding with u).



- Add/Remove a vertex
- Add/Remove an edge.
- Edge Query: given vertices u, v, does G contain edge (u, v) or (v, u)?
- Neighbourhood (undirected graph): given vertex v, return set of vertices u such that $(v, u) \in E$.
- In-neighbourhood / out-neighbourhood (directed graph): given vertex v, return set of vertices u such that $(u, v)/(v, u) \in E$.
- Degree/ In-degree/ Out-degree:
  size of neighbourhood / in-neighbourhood / out-neighbourhood.
- Traversal: visit each vertex of a graph to perform some task.

Adjacency Matrix vs Adjacency Lists

$|V| = n$ $|E| = m$

	Adjacency Matrix	Adjacency Lists
Space Complexity	$\Theta(n^2)$	$\Theta(n+m)$
Add/Remove a Vertex	$\Theta(n)$	$\Theta(1)$ → add $\Theta(m)$ → remove
Edge Query	$\Theta(1)$	$O(n)$ → more precise bound: $O(\min(n, m))$
Neighbourhood	$\Theta(n)$	$O(n)$

For an arbitrary undirected graph

Choose the more appropriate implementation depending on the problem.

Graph Traversal

Graph Traversal:
Visit **all** the vertices in a graph to perform some task (task depends on the application).
The visits can follow **different orders**.

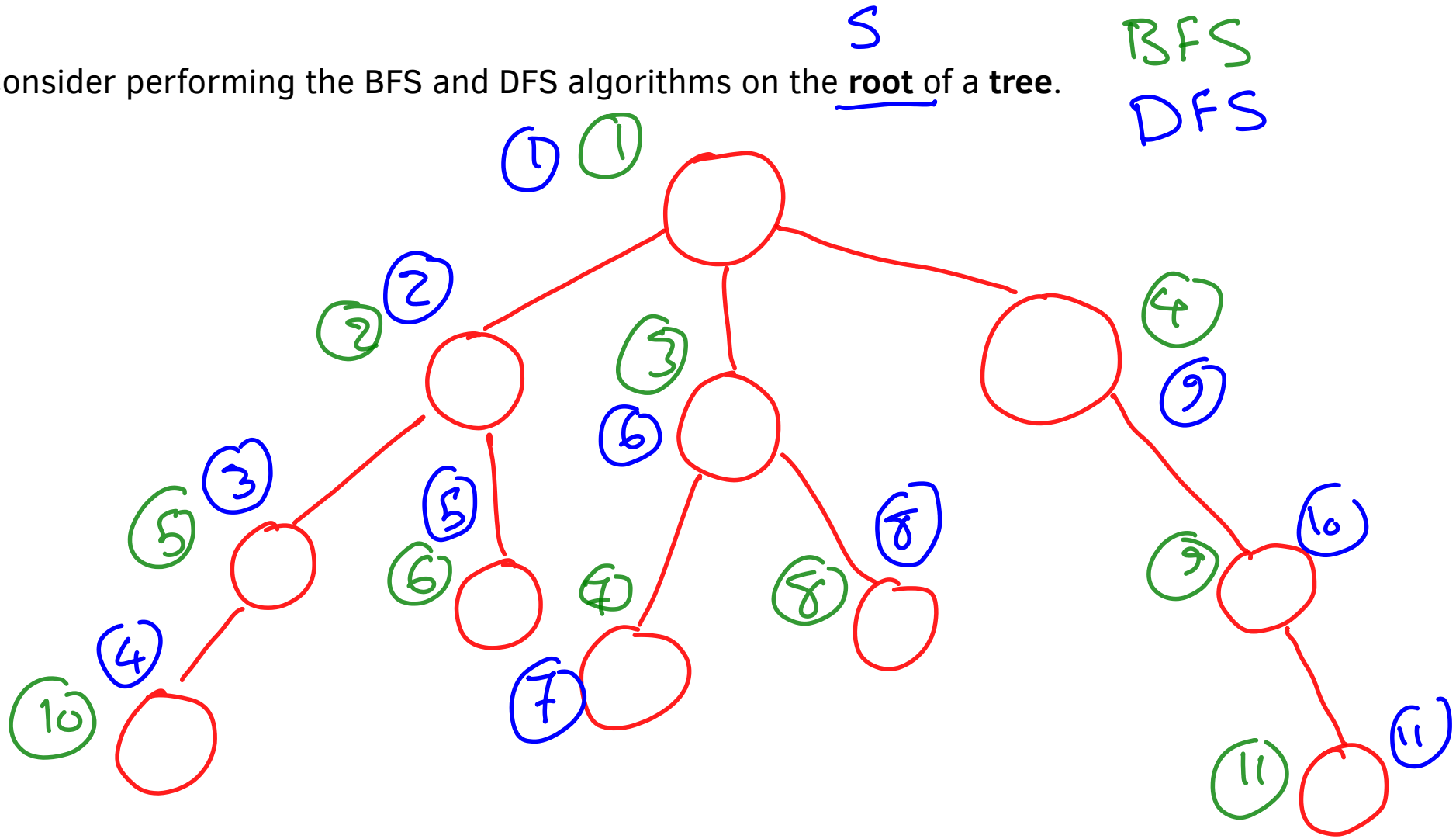
Graph Traversal

Graph Traversal Algorithms: Starting from a given vertex $s \in V$, visit every vertex in the graph that is reachable from s.

- **Breadth-First Search (BFS)**:
  - **Intuition:** Starting from s, first visit **all** (unvisited) neighbours of s, then visit **all** (unvisited) neighbours of all neighbours of s, then visit **all** (unvisited) neighbours of all neighbours of all neighbours s, and so on, until all reachable vertices are visited.
- **Depth-First Search (DFS)**:
  - **Intuition:** Traverse the depth of any particular path before exploring its breadth. Starting from s, first visit **a neighbour** n_1 of s, then visit **an unvisited neighbour** n_2 of n_1, then visit **an unvisited neighbour** n_3 of n_2, and so on. If visiting a neighbour n_k of n_{k-1} and all neighbours of n_k are already visited, try another neighbour of n_{k-1}.
    Stop when all vertices reachable from s are visited.

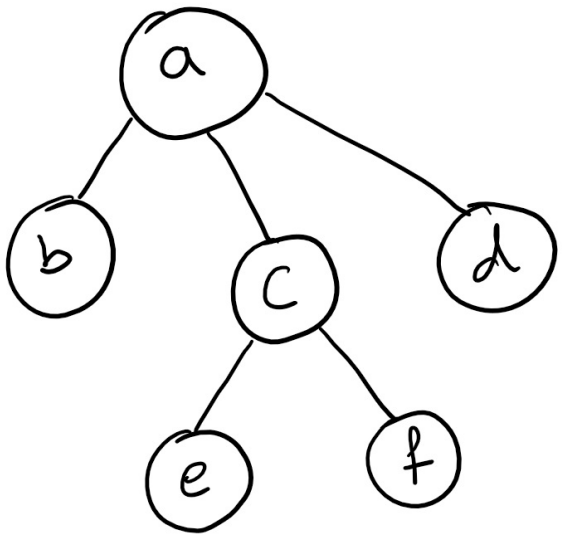
BFS vs DFS: A Familiar Example

Consider performing the BFS and DFS algorithms on the **root** of a **tree**.



BFS in a tree (starting from root) is a level-by-level traversal.
DFS visits the child vertices before visiting the sibling vertices.

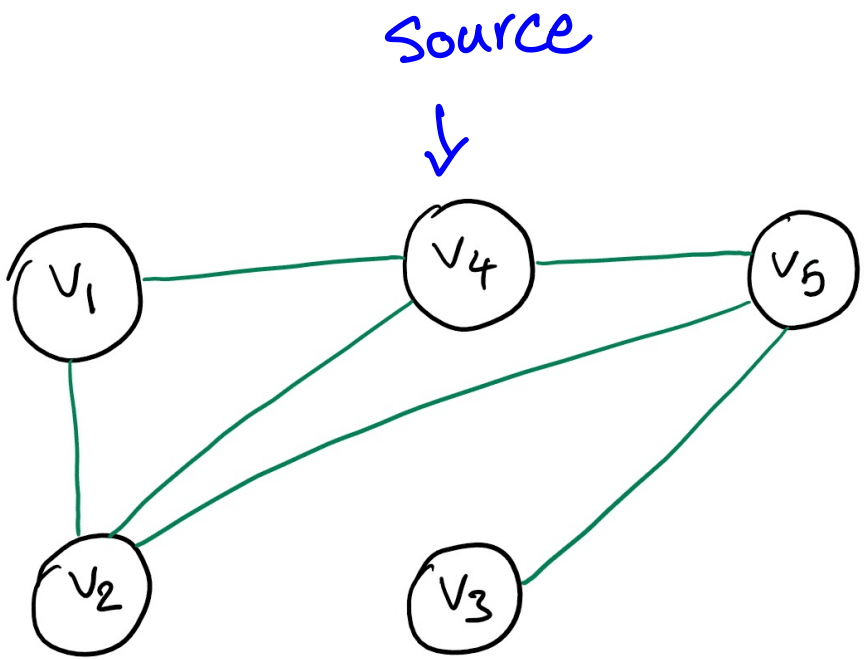
```
NotYetBFS(T, root):
1       Q = ∅
2       Enqueue(Q, root)
3       While Q not empty:
4           u = Dequeue(Q)
5           print u
6           for each child c of u:
7               Enqueue(Q, v)
```

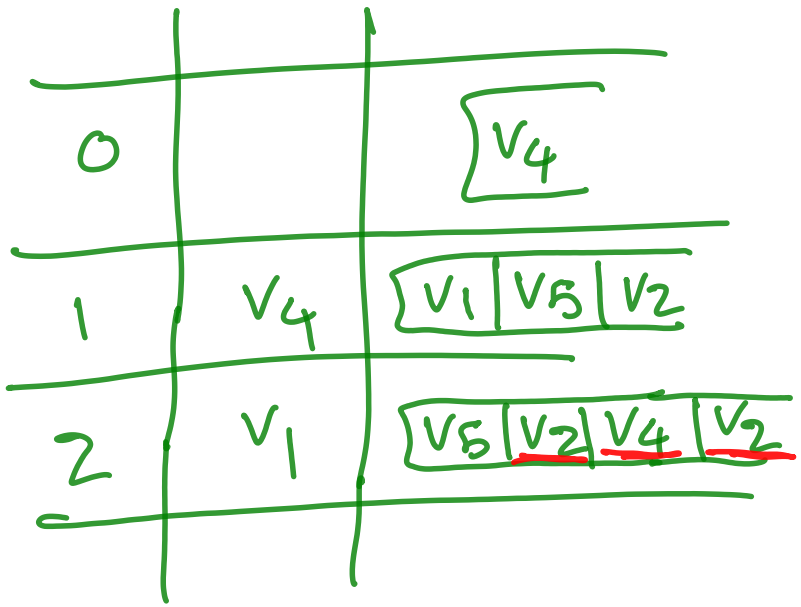


Iteration	Output	Queue
0	–	a
1	a	b c d
2	b	c d
3	c	d e f
4	d	e f
5	e	f
6	f	

BFS Implementation

What happens if we run NotYetBFS on a non-tree graph?
BFS will try to visit some of the vertices **twice**.





Note: We refer to the **starting vertex** of a BFS call as the source vertex.

How avoid visiting a vertex **twice**?
Remember the visited vertices by labelling them using colours.

- White: **Unvisited** (undiscovered) vertices.
  Have **not** been enqueued.

- Gray: **Encountered** (discovered) vertices.
  Have been **enqueued**.

- Black: **Explored** vertices.
  Have been **denqueued** and all of their **neighbours** are **enqueued**.

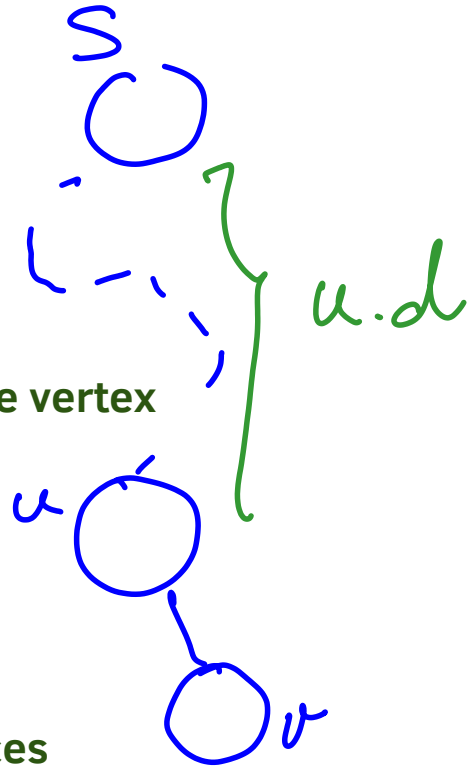
- Initially all vertices are **white**.
- Change a vertex's color to **gray** the first time visiting (enqueueing) it.
- Change a vertex's color to **black** when all its neighbours have been encountered (enqueued).
- Avoid visiting (enqueueing) **gray** or **black** vertices.
- In the end, all vertices that are reachable from the source are **black**.

Other useful values to remember during the traversal:

- The vertex from which v is encountered. (stored in $v.p$)
- The distance value: the distance from v to the source vertex of the BFS. (stored in $v.d$)

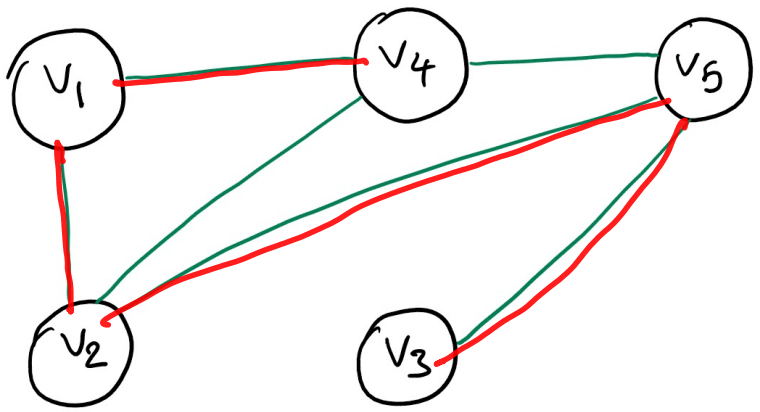
BFS(G, s) :

1. **for** each vertex $v \in V - \{s\}$: **# Initialize vertices**
2. $v.colour =$ White
3. $v.d = \infty$
4. $v.p =$ Nil
5. $Q = \varnothing$
6. $s.colour =$ Gray **# start BFS by encountering the source vertex**
7. $s.d = 0$ **# distance from s to s is 0**
8. Enqueue(Q, s)
9. **While** Q not empty:
10. $u =$ Dequeue(Q)
11. **for** each $v \in G.adj[u]$:
12. **if** $v.colour ==$ White **# only visit unvisited vertices**
13. $v.colour =$ Gray
14. $v.d = u.d + 1$ **# v is "1-level" farther from s than u**
15. $v.p = u$ **# v is introduced as u's neighbour**
16. Enqueue(Q, v)
17. $u.colour =$ Black **# u is explored as all its neighbours have been encountered**



The blue lines are the same as NotYetBFS.

Source : v_1



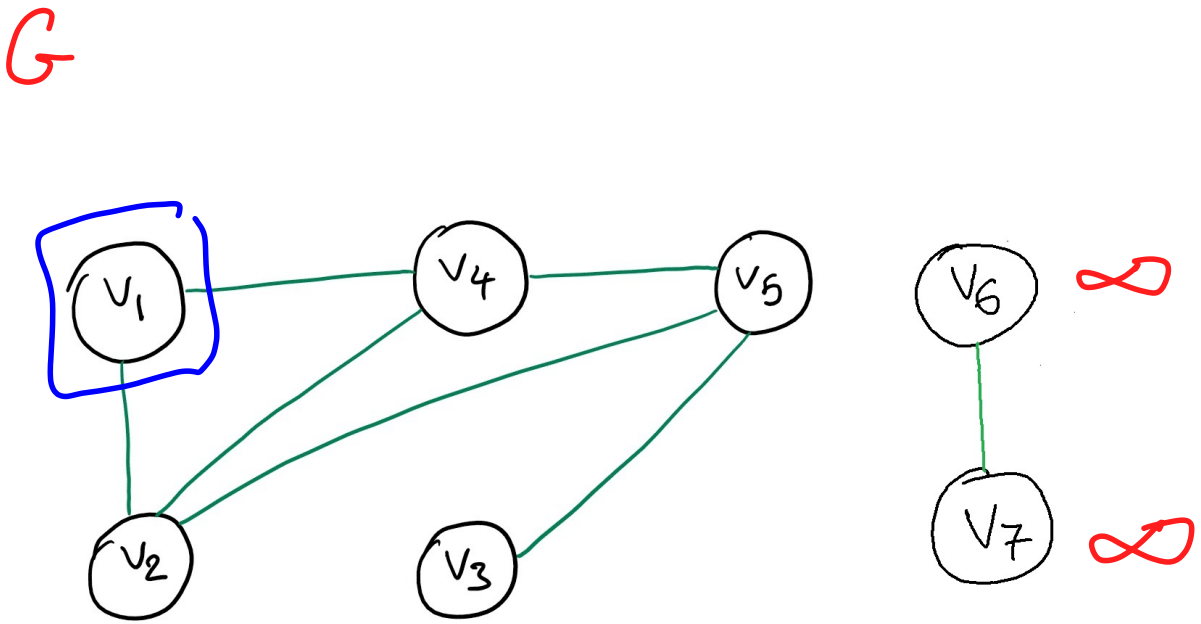
Vertex	*colour*	d	p	
v_1	~~G~~ B	0	Nil	
v_2	~~W~~ ~~G~~ B	~~∞~~ 1	~~Nil~~ v_1	v_1
v_3	~~W~~ ~~G~~ B	~~∞~~ 3	~~Nil~~ v_5	v_5
v_4	~~W~~ ~~G~~ B	~~∞~~ 1	~~Nil~~ v_1	v_1
v_5	~~W~~ ~~G~~ B	~~∞~~ 2	~~Nil~~ v_2	v_2

Iteration	Queue	u
0	v_1	
1	v_2 | v_4	v_1
2	v_4 | v_5	v_2
3	v_5	v_4
4	v_3	v_5
5		v_3
6		

BFS Tree

The set of all edges between vertices and their **introducer** form a tree, call a BFS tree

What if G is disconnected?



The **infinite distance** values of v_6 and v_7 (i.e., $v_6.d$ and $v_7.d$) indicate that they are unreachable from the source.

BFS Worst-case Running Time

The total amount of work (use **adjacency list**): $n = |V|$ $m = |E|$

1. When visiting each vertex: n
   Enqueue, Dequeue, assign values to $v.colour$, $v.d$, $v.p$, etc. $\Theta(1)$

2. At each vertex, check all its neighbours (i.e., all its incident edges).
   Each edge is checked at most twice (by the two end vertices)

- Total for 1: $\Theta(n)$
- Total for 2: $\Theta(m)$

Total running time: $\Theta(n + m)$

Exercise: What is the BFS Worst-case running time when using an adjacency **matrix**? $\Theta(n^2)$

BFS Properties

- BFS can be performed on both **directed and undirected** graphs.

- What do we get after performing a BFS:

  1. We get to visit every vertex which is **reachable** from the **source** s.

  2. We generate a **BFS tree**.
     Each edge in a BFS tree is called a **BFS tree edge**.
     The BFS tree connects **all** vertices, **if** the graph is **connected**.
     That is, if the graph is **not** connected, then for **every** BFS tree of the graph, **some** vertices of the graph are **not included** in the tree.

  3. For every vertex v, $v.d$ stores the minimum distance (or shortest-path distance) between the BFS **source** and v.
     To get the shortest-path itself, follow $.p$ attribute (i.e., the red edges in the example) from v to s backward.
     (Proof: Theorem 22.5 in CLRS)

BFS Applications

- **Traversing** graphs (Based on Property 1).
- Identifying **connectedness** (Based on Property 2).
- Finding **shortest-path** between two points (Based on Property 3).

After Lecture

- Relevant sections and exercises from Chapter 6 of the course notes.

- After-lecture Readings: CLRS Sections 22.1, 22.2