A graph consists of a set of vertices $V$ and a set of edges $E = \{(v_1, v_2) | v_1, v_2 \in V\}$.

If $(v_1, v_2)$ are ordered, we have a directed graph, or digraph.

If $(v_1, v_2)$ are unordered, we have a undirected graph (or simply graph).

In a directed graph, edge $(u, v)$ is different from edge $(v, u)$. For a directed edge $(u, v)$, $u$ is called the source and $v$ the destination.

In an undirected graph, $(u, v)$ and $(v, u)$ represent the same edge.
**Applications**

Graphs are used to represent and investigate the relationships (edges) among entries (vertices).

**Examples:**
Let vertex set $V_1$ contain all the states of USA.

- Graph $G_1 = (V_1, E_1)$ where
  $$E_1 = \{(v_1, v_2) \mid \text{states } v_1 \text{ and } v_2 \text{ are geometrically adjacent}\}$$

- Graph $G_2 = (V_1, E_2)$ where
  $$E_1 = \{(v_1, v_2) \mid \text{there is a direct Northwest flight from state } v_1 \text{ to state } v_2\}$$
Let vertex set $V_2$ be all the 1999 CS courses at GMU.

- Graph $G_3 = (V_2, E_3)$ where
  
  \[ E_3 = \{(v_1, v_2) \mid \text{course } v_1 \text{ is a prerequisite of course } v_2\}\]

- Graph $G_4 = (V_2, E_4)$ where
  
  \[ E_4 = \{(v_1, v_2) \mid \text{courses } v_1 \text{ and } v_2 \text{ are taught by the same instructor}\}\]

Let vertex set $V_3$ be all the computers connected to the Internet.

- The topology of the Internet can be defined as a graph $G_5 = (V_3, E_5)$, where
  
  \[ E_5 = \{(v_1, v_2) \mid \text{machines } v_1 \text{ and } v_2 \text{ are connected directly via some communication link}\}\]

---

**One More Application**

Let us consider a little game.

- To start the game, you place three coins on the table in a line as (H, T, H).

- You may flip the middle coin whenever you want to.

- You may flip one of the end coins only if the other two coins are the same as each other.

- The goal of the game is to reach the configuration (T, H, T).
We could use a graph to help figure out a solution.

- Vertices: all possible coin configurations, that is, all combinations of H/T of three coins.
- Edges: two configurations $u$ and $v$ are connected if we can go from configuration $u$ to configuration $v$ with a legal flip.

```
(H, H, H)
(T, H, H)   (H, T, H)   (H, H, T)
(T, T, H)   (T, H, T)   (H, T, T)
(T, T, T)
```

**Terminology**

- A vertex is also called a **node**, and an edge is also called a **link**.
- Two nodes $u$ and $v$ are **adjacent** if $e = (u, v)$ is an edge in the graph.
  We also say $u$ and $v$ are connected by edge $e$.
- A **path** a sequence of vertices

$$v_0, v_1, v_2, \ldots, v_k$$

such that $(v_i, v_{i+1})$, $0 \leq i < k$, is an edge in $G$.
- A **cycle** is a path that starts and ends at the same vertex.
- A graph $G$ is **connected** if starting with any vertex $v$ you can reach all the other vertices in $G$. 
Representations of Graphs

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

- Adjacency Table:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>E</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- Adjacency List:

A = 0
B = 1
C = 2
D = 3
E = 4
• **Edge Sets**: use a `Set` data structure to store the neighboring vertices of each vertex.
  Assume that we have implemented a template `Set` class, which internally uses a B-tree to store the objects of a set. To represent a graph with 10 vertices:
  ```
  Set<int> neighbors[10]
  ```
  (set `neighbors[i]` contains all neighbors of vertex `i`).

---

**Which Representation is Best?**

• Adjacent table
  🔴 Space consumption: $O(N^2)$, where $N$ is the number of vertices.
  🔴 $O(1)$ time to add/remove edges or to check if two vertices are connected.
  🔴 $O(N)$ time to find out all the neighbors of a given node.
  🔴 Easy to implement
  🔴 Bad for **sparse** graphs where there is a large number of nodes but each node has only a small number neighbors.
• Adjacent list
  ☞ Space consumption depends on the “density” of the graph.
    General cases ?
    Worst case ?
  ☞ Time to add/remove edges or to check if two vertices are connected:
    General cases ?
    Worst case ?
  ☞ Time to find out all the neighbors of a given node:
    General cases ?
    Worst case ?
  ☞ Good for sparse graphs.

• Edge sets
  ☞ Space consumption depends on the “density” of the graph.
    General cases ?
    Worst case ?
  ☞ Time to add/remove edges or to check if two vertices are connected:
    General cases ?
    Worst case ?
  ☞ Time to find out all the neighbors of a given node:
    General cases ?
    Worst case ?
  ☞ Performance is stable across wildly different graphs.
  ☞ Difficult to implement due to the involvement of advanced tree structures.
Implementing Digraphs Using Adjacency List

```c++
struct ListNode {int vertex_id, ListNode* next};
typedef ListNode* ListNode_ptr;

class Digraph {
protected:
    ListNode **neighbors_list; // a ListNode* array whose
    // size determined by max_size
    string* labels; // a string array whose size determined by
    // max_size
    int vertex_count;
    int max_size; // max_size not const; it can vary
    // from graph to graph

public:
    Digraph (int n) {
        vertex_count=0;
        max_size=n;
        labels=new string[max_size];
        neighbors_list = new ListNode_ptr[max_size];
    }
    ~Digraph();
    void add_vertex (const string& label);
    void add_edge (int u, int v);
    void remove_edge (int u, int v);
    bool is_connected (int u, int v); // return true if
    // edge (u,v) exists.
    string& operator[] (int i) {return labels[i];}
};
```
void Digraph::add_vertex (const string& label) {
    int new_vertex_number = vertex_count++;
    neighbors_list[new_vertex_number] = NULL;
    labels[new_vertex_number] = label;
}

void Digraph::add_edge (int u, int v) {
    if (is_connected (u,v)) return;

    ListNode* p = new ListNode;
    p->vertex_id = v;
    p->next = neighbors_list[u];
    neighbors_list[u] = p;
}

bool Digraph::is_connected (int u, int v) {
    ListNode *p;
    for (p=neighbors_list[u]; p!=NULL; p=p->next)
        if (p->vertex_id == v) return true;
    return false;
}

bool Digraph::remove_edge (int u, int v) {
    ListNode **p;
    for (p=&neighbors_list[u]; *p!=NULL; p=&(*p)->next)
        if ((*p)->vertex_id == v) {
            *p = (*p)->next;
            return true;
        }
    return false;
}
To build the graph shown in page 1,

```cpp
Digraph g(20); // max node number = 20
g.add_vertex ('A'); // vertex A numbered 0
g.add_vertex ('B'); // vertex A numbered 1
g.add_vertex ('C'); // vertex A numbered 2
g.add_vertex ('D'); // vertex A numbered 3
g.add_vertex ('E'); // vertex A numbered 4
g.add_edge (0, 2); // A -> C
g.add_edge (0, 3); // A -> D
g.add_edge (0, 4); // A -> E
g.add_edge (1, 4); // B -> E
g.add_edge (2, 0); // C -> A
g.add_edge (2, 1); // C -> B
... and other edges ...
```

To change the name of node A to X, `g[0] = 'X';`

---

**Implementing Undirected Graphs**

```cpp
class Graph : public Digraph {
public:
    Graph (int n) : Digraph(n) {} // constructor
    void add_edge (int u, int v);
    void remove_edge (int u, int v);
};
```
void Graph::add_edge (int u, int v) {
}

void Graph::remove_edge (int u, int v) {
}

But, do we have access to private resources of a base class?
Solution?

---

**Graph Traversals**

In what order should we visit vertices in a graph?

- By the order of vertex numbers?

- Depth-first search (DFS)
  - Start with a given vertex, keep moving forward until you cannot, and then backtrack.
  - When there are multiple out-going edges, choose any.
  - Due to the above random selection, the depth-first search of a graph does not define a unique node sequence.
- Breadth-first search (BFS)
  - A starting vertex is given.
  - Visit all your neighbors before visiting neighbors of neighbors.
  - The order in which the neighbors are visited does not matter.
  - Again, the breadth-first search of a graph does not define a unique node sequence.

Note: DFS and BFS do not guarantee to visit all the vertices of the given graph.

Example

Starting vertex = 40

“One” DFS order:

“One” BFS order:

How about when starting vertex = 25?
Example

```
Starting vertex = 0
“One” DFS order:
“One” BFS order:
```

Implementing DFS

First, we add a new public method to class `Digraph`.

```cpp
void Digraph::dfs (int start)
{
    bool* marked = new bool [vertex_count];
    for (int i=0; i<size(); i++)
        marked[i] = false;

    recursive_dfs (start, marked);
}
```

The `recursive_dfs()` method is private.
void Diraph::recursive_dfs (int start, bool* marked)
{
    marked[start] = true;
    cout << start;

    ListNode* p
    for (p=neighbors_list[start]; p!=NULL; p=p->next)
        if (!marked[p->vertex_id])
            recursive_dfs (p->vertex_id, marked);
}

Implementing BFS

The BFS can be implemented by a single, public method.

void Digraph::bfs (int start)
{
    bool* marked = new bool [vertex_count];
    for (int i=0; i<vertex_count; i++)
        marked[i] = false;

    marked[start] = true;
    cout << start;
    Queue<int> q;
    q.insert (start);
do {
    int v = q.remove();
    ListNode* p;
    for (p=neighbors_list[v]; p!=NULL; p=p->next)
        if (!marked[p->vertex_id]) {
            marked[p->vertex_id] = true;
            cout << p->vertex_id;
            q.insert(p->vertex_id);
        }
} while (!q.is_empty());

### Implementing Digraphs Using Adjacency Table

```cpp
template <int MAX_SIZE>
class Digraph {
private:
    bool edges[MAX_SIZE][MAX_SIZE];
    string labels[MAX_SIZE];
    int vertex_count;
public:
    static const int max_size = MAX_SIZE;
    Digraph () {vertex_count = 0;} ~Digraph() {}
    void read (); void write ();
    void add_vertex (const T& label);
    void add_edge (int u, int v) {edges[u][v] = true;}
    void remove_edge (int u, int v) {edges[u][v] = false;}
    bool is_connected (int u, int v) {return edges[u][v];}
    T& operator[] (int i) {return labels[i];}
};
```
Implementing DFS in Adjacency Table

```cpp
template <int MAX_SIZE>
void Diraph<MAX_SIZE>::recursive_dfs (int start, bool* marked)
{
    marked[start] = true;
    cout << start;
}
```