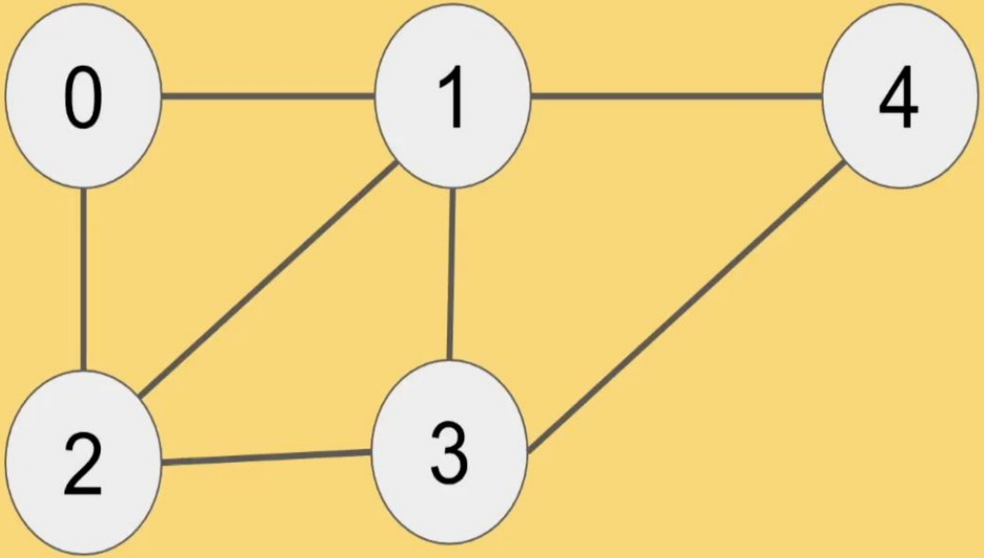
Graph Representation and Algorithms

Andy Bakir
CS 131

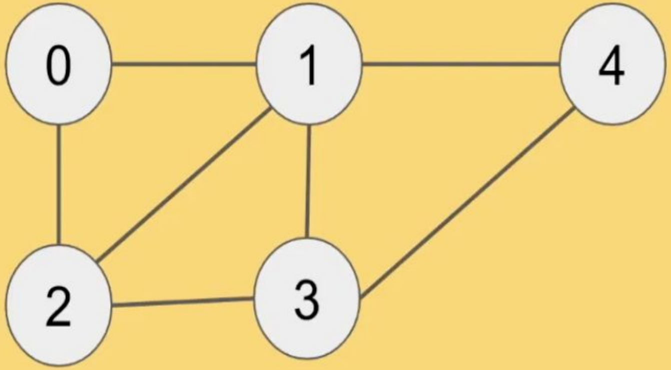


How to **represent** this in **C++**?

Algorithms: DFS, BFS, Shortest Path, ...

Outline

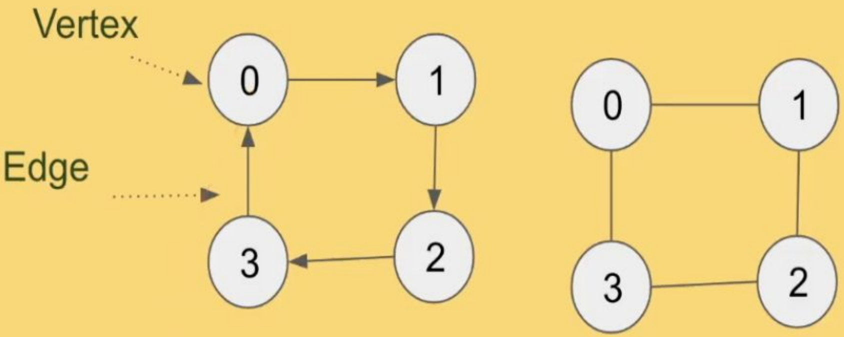
- What is a **Graph**?
- Graph **Representation in C++ using STL**
  - What kind of of **data structure** can should use?
- Implement an **Algorithm**



What is a Graph

Graph Definition

- **V**: Set of **vertices** (AKA Nodes)
- **E**: Set of edges
- **G=**(V, E)
    - **Undirected Graph:**
        - E: set of unordered Pairs of vertices
    - **Directed Graph:**
        - E: set of ordered **Pairs** of vertices
- What data structure to use?
    - It really comes down to representing **sets** and **pairs!**



V={0, 1, 2, 3}

E={(0,1), (1,2), (2,3), (3,0)}

If **e** = **(a,b)** ∊ **E**, then **a** and **b** are **adjacent** (AKA **neighbors**)

History: The Seven Bridges of Königsberg

Walk through the city and cross each of those bridges **exactly once.**

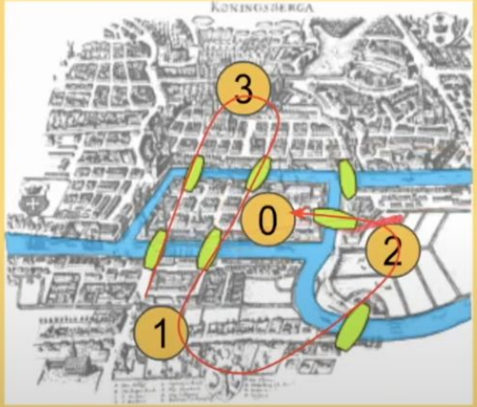
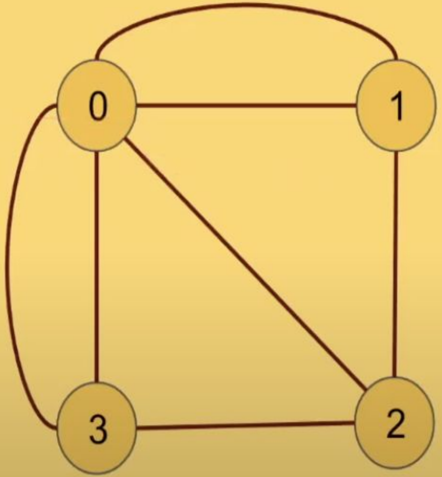
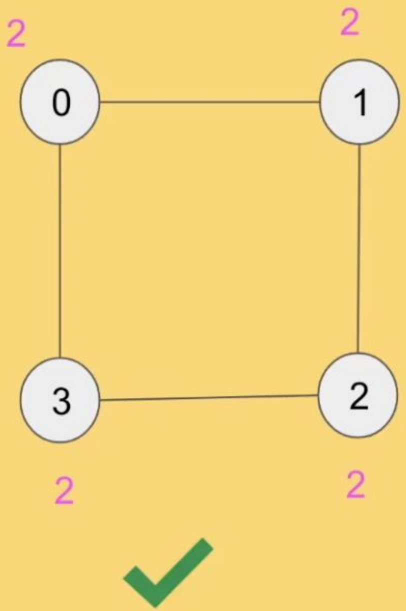


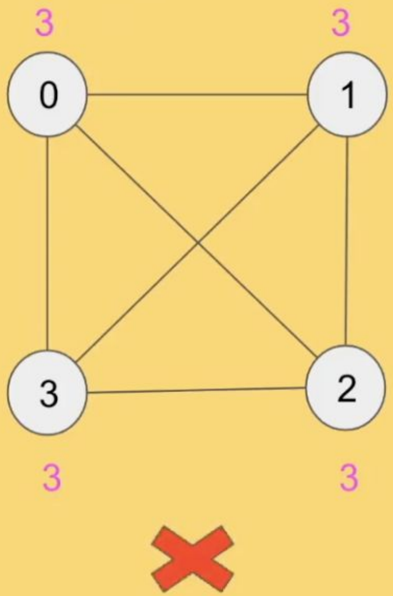
Image: Wikipedia



Euler's Observation

- Degree: Number of edges that touch a vertex
- Exactly **zero** or **two** vertices can have an **odd** degree

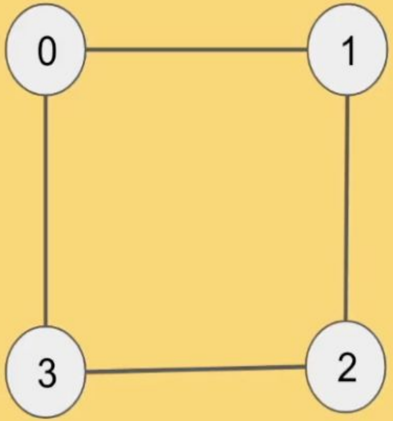




Graph Representation in C++

Graph Representation

- **Direct Translation of the Definition:**
  - **V: Set/List/Vector** of vertices
  - **E: Set/List/Vector** of pairs
- **Adjacency List**
  - **V: Set/List/Vector** of vertices
  - **E: Set/Map/List/Vector** of all adjacent vertices
- **Adjacency matrix**

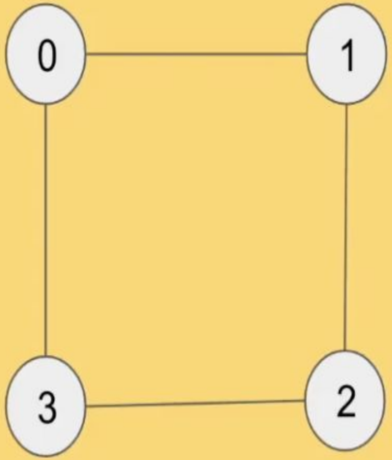


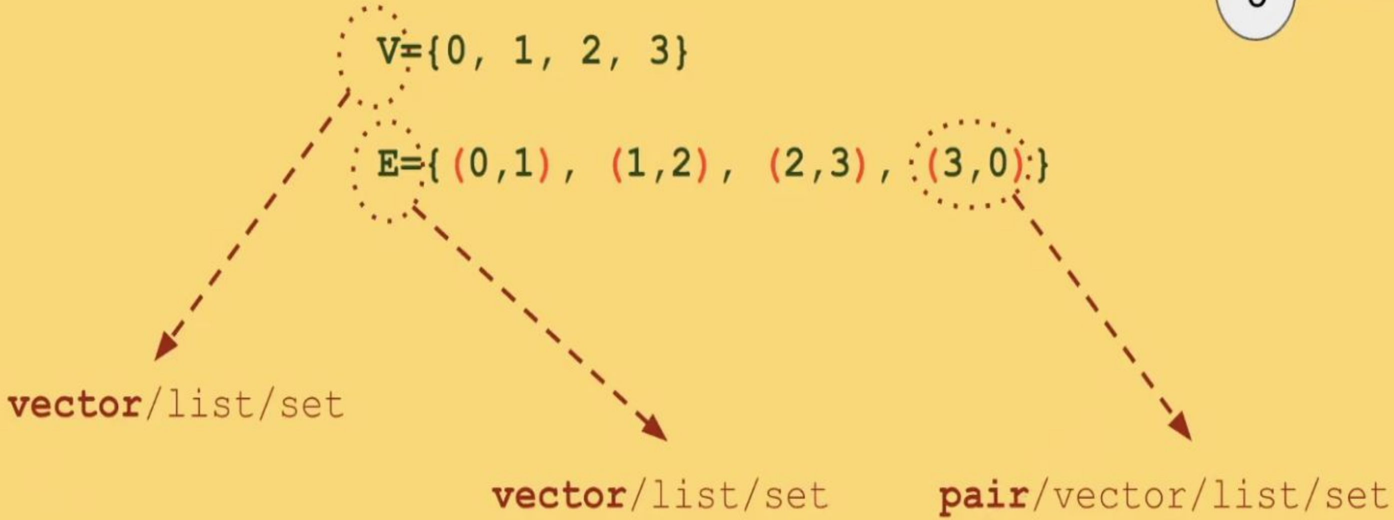
What you need to know about each method?

1. How to initialize a graph
2. How to run simple algorithms

Direct Translation of the Definition:

- **V: Set/List/Vector** of vertices
- **E: Set/List/Vector** of pairs





```
V={0, 1, 2, 3}

E={(0,1), (1,2), (2,3), (3,0)}
```

vector/list/set

vector/list/set

pair/vector/list/set

V: Vector of Vertices, E: Vector of Pairs

```
class Graph {
public:
 Graph(std::vector<int> &v, std::vector<std::pair<int, int>> &e)
     : v_(v), e_(e) {}
 bool IsEulerWalkable();
 std::vector<int> v_;
 std::vector<std::pair<int, int>> e_;
};
```

```
V={0, 1, 2, 3}

E={(0,1), (1,2), (2,3), (3,0)}
```

```
int main() {
 std::vector<int> v = {0, 1, 2, 3};
 std::vector<std::pair<int, int>> e = {{0, 1}, {1, 2}, {2, 3}, {3, 0}};
 Graph g(v, e);
 std::cout << g.IsEulerWalkable() << std::endl;
}
```

1. How to initialize?

V: Vector of Vertices, E: Vector of Pairs

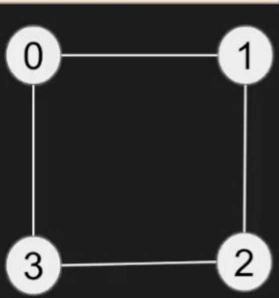
```cpp
bool Graph::IsEulerWalkable() {
 std::vector<int> degrees(v_.size());
 for (auto e : e_) {
   degrees[e.first]++;
   degrees[e.second]++;
 }
 int countOdds = 0;
 for (auto d : degrees) {
   if (d % 2 == 1) {
     countOdds++;
   }
 }
 return (countOdds == 0 || countOdds == 2);
}
```

2. How to run algorithms?

Is the count of odd-degree vertices 0 or 2?

```cpp
std::vector<int> v_;
std::vector<std::pair<int, int>> e_;
v_ = {0, 1, 2, 3}
e_ = {(0,1), (1,2), (2,3), (3,0)}
```

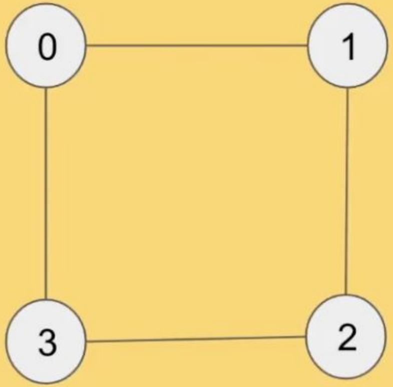
2	2	2	2

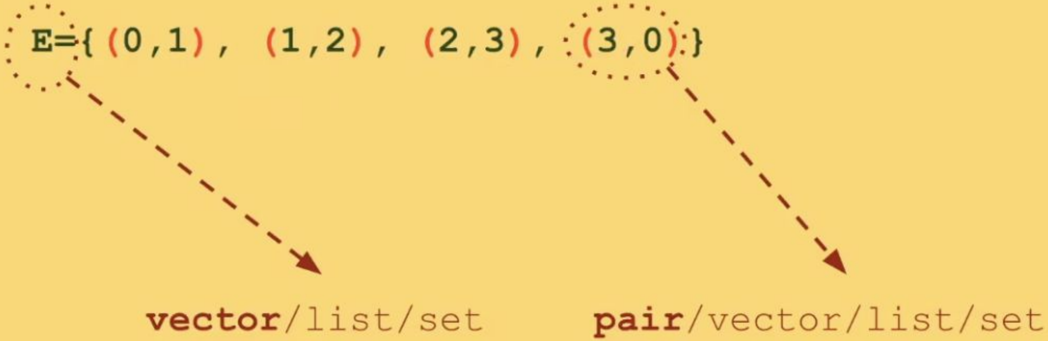


Direct Translation of the Definition:

- ~~V: Set/List/Vector of vertices~~
- **E: Set/List/Vector** of pairs

We might drop V!





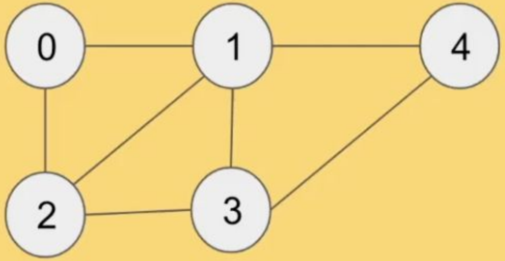
`E={(0,1), (1,2), (2,3), (3,0)}`

vector/list/set

pair/vector/list/set

Adjacency List

Vertex Number	Adjacents
0	{1, 2}
1	{0, 2, 3, 4}
2	{0, 1, 3}
3	{1, 2, 4}



New class

```
Vertex:
    Vertex_number: 0
    Adjacents: {1, 2}
```

V: Vector of Vertices, E: Adjacency List (Set)

```
struct Vertex {
 Vertex(int v, std::set<int> &a) : vertex_number(v), adjacents(a) {}
 int vertex_number;
 std::set<int> adjacents;
};
class Graph {
public:
 Graph(std::vector<Vertex> &v) : v_(v) {}
 std::vector<Vertex> v_;
};
```

Vertex Number	Adjacents
0	{1, 2}
1	{0, 2, 3, 4}
2	{0, 1, 3}
3	{1, 2, 4}

```
int main() {
 Graph g({Vertex(0, {1, 2}), Vertex(1, {0, 2, 3, 4}), Vertex(2, {0, 1, 3}),
          Vertex(3, {1, 2, 4})});
 std::cout << g.IsEulerWalkable() << std::endl;
}
```

1. How to initialize?

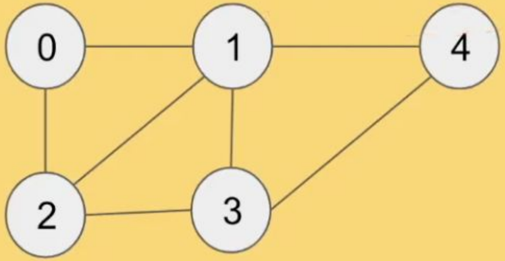
V: Vector of Vertices, E: Adjacency List (Set)

```
bool Graph::IsEulerWalkable() {
 std::vector<int> degrees(v_.size());

 for (auto v : v_) {
   degrees[v.vertex_number] = v.adjacents.size();
 }



 int countOdds = 0;

 for (auto d : degrees) {
   if (d % 2 == 1) {
     countOdds++;
   }
 }
 return (countOdds == 0 || countOdds == 2);
}
```

```
struct Vertex {
 int vertex_number;
 std::set<int> adjacents;
};
class Graph {
public:
 std::vector<Vertex> v_;
};
```

Vertex Number	Adjacents
0	{1, 2}
1	{0, 2, 3, 4}
2	{0, 1, 3}
3	{1, 2, 4}

Adjacency Matrix

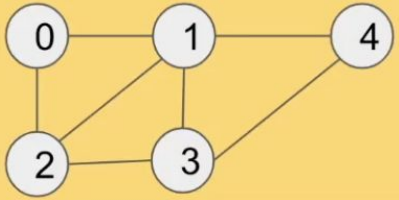


	0	**1**	**2**	**3**	**4**
0	0	1	1	0	0
1	1	0	1	1	1
2	1	1	0	1	0
3	0	1	1	0	1
4	0	1	0	1	0

- **Directed**:
  - M[i,j] = 1 If (i,j) ∊ E
  - M[i,j] = 0 otherwise
- For **undirected graphs**, edges are considered to be bidirectional
  - M[i,j] = M[j,i] = 1 If {i,j} ∊ E
  - M[i,j] = 0 otherwise

What is the memory cost?

Adjacency Matrix: Vector of Vector



```
class Graph {
public:
 Graph(std::vector<std::vector<int>> & adjacency) : adjacency_(adjacency) {}
 bool IsEulerWalkable();
 std::vector<std::vector<int>> adjacency_;
};
int main() {
 std::vector<std::vector<int>> adjacency = {{0, 1, 1, 0, 0},
                                            {1, 0, 1, 1, 1},
                                            {1, 1, 0, 1, 0},
                                            {0, 1, 1, 0, 1},
                                            {0, 1, 0, 1, 0}};
 Graph g(adjacency);
}
```

1. How to initialize?

Adjacency Matrix: Vector of Vector

```cpp
bool Graph::IsEulerWalkable() {
 std::vector<int> degrees(adjacency_.size());
 for (int i = 0; i < adjacency_.size(); i++) {
   for (int j = 0; j < adjacency_.size(); j++) {
     if (adjacency_[i][j] == 1) {
       degrees[i]++;
     }
   }
 }

 int countOdds = 0;

 for (auto d : degrees) {
   std::cout << "d: " << d << std::endl;
   if (d % 2 == 1) {
     countOdds++;
   }
 }
 return (countOdds == 0 || countOdds == 2);
}
```

```cpp
class Graph {
public:
 std::vector<std::vector<int>> adjacency_;
};
```

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

What Should We Take Away?

```
v_  = {0,  1,  2,  3}
e_  = {(0,1),  (1,2),  (2,3),  (3,0)}
```

Vertex Number	Adjacents
0	{1, 2}
1	{0, 2, 3, 4}
2	{0, 1, 3}
3	{1, 2, 4}

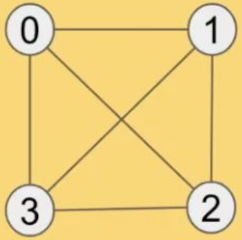
	0	**1**	**2**	**3**	**4**
0	0	1	1	0	0
1	1	0	1	1	1
2	1	1	0	1	0
3	0	1	1	0	1
4	0	1	0	1	0

	Direct Translation	**Adjacency List**	**Adjacency Matrix**
Iterating edges	O(m)	O(m+n)	O(n^2)
Finding adjacents of `i`	O(m)	O(1)	O(n)
Check if `i,j` are adjacent	O(m): vector, O(log m): set	O(n): vector, O(log n): set	O(1)
Degree of each vertex	O(m)	O(1)	O(n)
Memory size	O(m+n)	O(m+n)	O(n^2)

Comparison Of Data Structures

- Set
  - Automatically sorted
  - Insert/Delete/Find: **O(log n)**

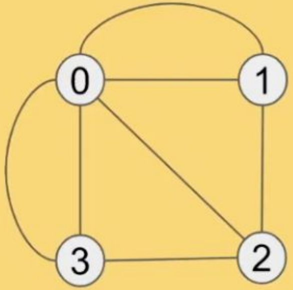
- Unordered Set
  - Not sorted
  - Insert/Delete/Find: **O(1), amortized**



Graph

- Vector
  - Not sorted
  - Insert/Delete/Search **O(n)**
  - Push_back **O(1), amortized**

- List
  - Not Sorted
  - Insert/Delete/Find: **O(n)**
  - Insert/Delete once found: **O(1)** (unlike vector)



Multigraph

Thanks Guys for watching