University of Science and Technology
CSAI 201 : Data Structures

Data Structures
Graphs

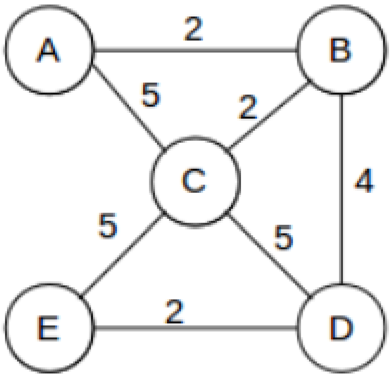
Objectives

After this lab, the student should be able to:

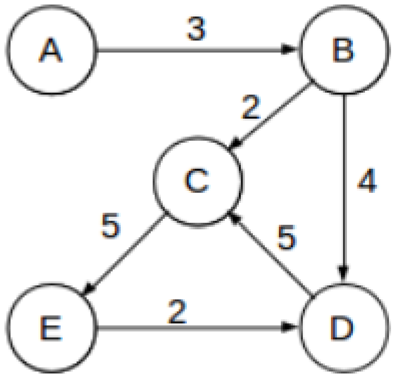
- Use class templates to implement the directed graph ADT using an adjacency matrix based implementation.
- Use class templates to implement the directed graph ADT using an adjacency list implementation.
- Implement graph traversal using BFS, DFS, and topological sort.

Part I – Graph ADT

A graph is a non-linear abstract data structure consisting of vertices and edges. The edges can be directed, or bi-directional. They can also have different weights depending on the problem at hand.

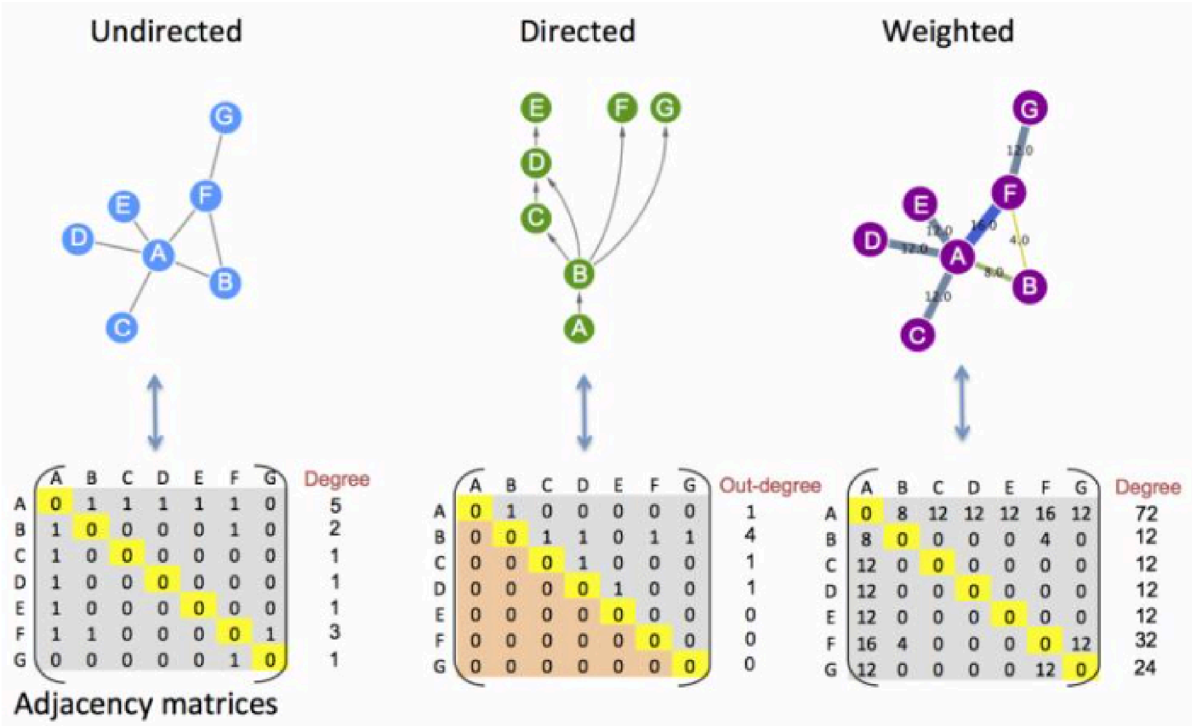


Undirected



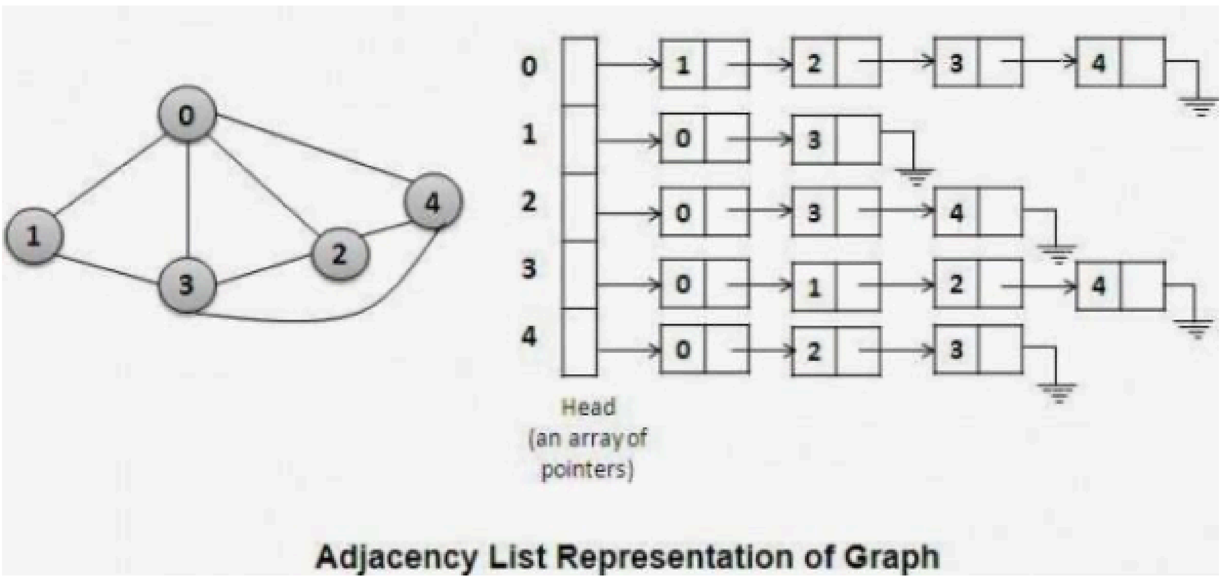
Directed

As discussed in the lecture and demonstrated in the tutorial, to store a graph, we can use an adjacency matrix.



Adjacency matrices have the advantage of O(1) access time since arrays allow random access. However, their major drawback is their $O(|V|2)$ space, which they take up even for sparse graphs. Another drawback of adjacency matrices is that arrays are not dynamic. Consequently, upon adding or removing a vertex, the matrix must be resized or copied, which is $O(|V|2)$ in the worst case.

Alternatively, we can use an adjacency list. Adjacency list save up space by only storing as many edges as needed ($O(|V| + |E|)$). They can also be edited dynamically since they use linked lists. Of course, this improvement comes at the expense of the lack of random access.



The Graph ADT interface is as follows:

```cpp
template<typename T>
class MatrixGraph : public GraphADT<T>
{
private:
    vector<T> vertices;
    vector<vector<int>> adjMatrix;

public:
    MatrixGraph();

    bool IsEmpty() const;
    bool AddVertex(const T& v);
    bool RemoveVertex(const T& v);
    bool AddEdge(const T& fromV, const T& toV, int weight = 1);
    void RemoveEdge(const T& fromV, const T& toV);
    void DisplayNeighbors(const T& fromV) const;
    bool EdgeExists(const T& fromV, const T& toV) const;
    bool IsAdjacent(const T& v1, const T& v2) const;
    void PrintGraph();
    void DFS(const T& startV) const;
    void BFS(const T& startV) const;
    ~MatrixGraph();
};
```

Part II – Graph Implementation

In this lab, you are given the matrix implementation of the graph ADT.

Code Examples

- Open "Graphs.sln," run project "*1-MatrixGraphs*", and see how the graph is implemented as a class *template* using a matrix representation:

- **Graph operations are implemented as class member functions**
- **This implementation uses vectors for easier handling**
- **Note the "Util.h" file, which includes useful utility functions for dealing with vectors**
- **This implementation uses an additional vector called vertices.**
    - **Can you figure out what it is for?**

Practice Exercises

Exercise 1

Create a new class *ListGraph* that implements the graph ADT using an adjacency list representation.

- **What is the most suitable underlying data structure?**
- **What member variables do you need?**
- **Do you need to add any new classes?**
- **How would you modify the implementations of the graph member functions?**

Exercise 2

Implement the member function MatrixGraph::DFS(). This function should print the graph nodes to the console in a DFS traversal fashion.

Exercise 3

Implement the member function MatrixGraph::BFS(). This function should print the graph nodes to the console in a BFS traversal fashion.

Exercise 4

Implement the two member functions ListGraph::DFS(), ListGraph::BFS(),