Graph Theory

Self Reading

Abhilasha

2024

Contents

1 Introduction to Graphs

1.1 Definition

A graph is a data structure consisting of a finite set of vertices (also called nodes) and edges. Graphs are used to represent networks of communication, data organization, computational devices, and many other systems where pairwise connections are required.
To talk about graphs we define graph G, with set of vertices V and set of edges E.
When we talk about subgraph of a graph we are talking about a graph represented by the subsets V1 of V and E1 of E such that each edge of G1 has same end vertices as in G.

1.2 Useful terms

- Walk: A valid sequence of edges to traverse the graph, it starts at a vertex and may or may not end at the same vertex
- Path: A walk in which no vertex and no edge is repeated
- Cycle: A path that starts and ends at the same vertex
- Degree of a vertex: the number of edges emanating from a vertex
- Component: A connected sub-graph of an undirected graph that is not part of any larger connected sub-graph

1.3 Directed and Weighted Graphs

Graphs can be classified into various types:

- **Directed Graph (Digraph)**:edges have directions.
- **Undirected Graph**:edges do not have directions.
- **Weighted Graph**:each edge has a weight or cost.
- **Unweighted Graph**:edges have no associated weights.

1.4 Types of Graphs (Based on structural properties)

1. Connected Graph: A graph where from any vertex, you have a path to reach every other vertex in the graph.
2. Simple Graph: A graph any pair of vertices have at most one edge between them.
3. Complete Graph: A simple graph in which the degree of each vertex is n-1 where n is the number of distinct nodes in the graph.
4. Regular Graph: All vertices in the graph have the same degree, that is, they have the same number of edges emanating from them.

5. Bipartite Graphs: A bipartite graph G is a graph whose vertex set V can be partitioned into two nonempty subsets A and B (i.e., A ∪ B=V and A ∩ B=∅) such that each edge of G has at most one endpoint in A and one endpoint in B. The partition V=A ∪ B is called a bipartition of G.
   - All subgraphs of a bipartite graph must be bipartite. We can have disconnected bipartite components or isolated vertices in a bipartite graph.
   - A bipartite graph cannot have any odd length cycles as this would require two vertices from the same bipartition set to be connected.
   - To colour a bipartite graph such that no two adjacent vertices have the same colour, we need only 2 colours (in minimal case, if you want you can choose more)
6. A cyclic graph is one which contains at least one cycle.

1.5 Representing Graphs

Graphs can be represented in multiple ways depending on what information we need them to store.

1. Adjacency Matrix An adjacency matrix is a 2D array of size $V \times V$ where V is the number of vertices in the graph. A value of 1 (or the edge weight) in cell (i, j) indicates an edge from vertex i to vertex j.

```
#include <iostream>
using namespace std;

const int V = 4;

void printMatrix(int graph[V][V]) {
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            cout << graph[i][j] << " ";
        }
        cout << endl;
    }
}

int main() {
    int graph[V][V] = {
        {0, 1, 0, 1},
        {1, 0, 1, 1},
        {0, 1, 0, 0},
        {1, 1, 0, 0}
    };
    printMatrix(graph);
    return 0;
}
```

2. Adjacency List An adjacency list is an array of linked lists. The array size is equal to the number of vertices. Each vertex has a list of vertices to which it's connected.

```
#include <iostream>
#include <list>
using namespace std;

class Graph {
    int V;
    list<int>* adj;

public:
    Graph(int V);
    void addEdge(int v, int w);
    void printGraph();
};

Graph::Graph(int V) {
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w) {
    adj[v].push_back(w);
    adj[w].push_back(v);
}

void Graph::printGraph() {
    for (int v = 0; v < V; ++v) {
        cout << "Adjacency list of vertex " << v << "\n";
        for (auto x : adj[v]) cout << "-> " << x;
        cout << endl;
    }
}

int main() {
    Graph g(4);
    g.addEdge(0, 1);
    g.addEdge(0, 3);
    g.addEdge(1, 2);
    g.addEdge(2, 3);

    g.printGraph();
    return 0;
}
```

1.6 Algorithms

1.6.1 Breadth-First Search (BFS)

BFS is a **traversal algorithm** that starts from a given node, explores all node at a specific distance from that node and move ahead (in trees it is like visiting all nodes at a level and then moving on).

```
#include <iostream>
#include <list>
#include <queue>
using namespace std;

class Graph {
    int V;
    list<int>* adj;

public:
    Graph(int V);
    void addEdge(int v, int w);
    void BFS(int s);
};

Graph::Graph(int V) {
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w) {
    adj[v].push_back(w);
}

void Graph::BFS(int s) {
    bool* visited = new bool[V];
    for (int i = 0; i < V; i++) visited[i] = false;

    queue<int> q;
    visited[s] = true;
    q.push(s);

    while (!q.empty()) {
        s = q.front();
        cout << s << " ";
        q.pop();

        for (auto i : adj[s]) {
            if (!visited[i]) {
                visited[i] = true;
                q.push(i);
```

```
            }
        }
    }
}
```

1.6.2 Depth-First Search (DFS)

DFS is another traversal algorithm that explores as far as possible along each branch before backtracking (In trees, it is like going as deep as you can along a branch before backtracking to the last point of difference. It is the same as pre-order walk of trees).

```
#include <iostream>
#include <list>
using namespace std;

class Graph {
    int V;
    list<int>* adj;
    void DFSUtil(int v, bool visited[]);

public:
    Graph(int V);
    void addEdge(int v, int w);
    void DFS(int v);
};

Graph::Graph(int V) {
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w) {
    adj[v].push_back(w);
}

void Graph::DFSUtil(int v, bool visited[]) {
    visited[v] = true;
    cout << v << " ";

    for (auto i : adj[v])
        if (!visited[i]) DFSUtil(i, visited);
}

void Graph::DFS(int v) {
    bool* visited = new bool[V];
    for (int i = 0; i < V; i++) visited[i] = false;

    DFSUtil(v, visited);
```

```
}
```

1.7 Advantages and Disadvantages of Graphs

1. Advantages
   - Efficient representation of complex relationships.
   - Traversal algorithms are very useful in implementation of other algos such as Dijkstra, kahn, etc.
   - Very useful to model real world systems like social networks, circuits, road or rail networks, server networks, etc.
2. Disadvantages
   - Can only represent relations, not individual properties
   - Complexity in implementing graph algorithms due to recursive exploration.
   - Representation of graphs is very costly, adjacency matrices take up a lot of space and have high time complexity processing algorithms.

1.8 Some Other Concepts (IMPORTANT)

1. Perfect Matching: A subset of edges that covers every vertex of the graph. Eg: Placing cctv cameras in a network of streets to cover all streets at a given time
2. Multigraph: A grah in which we can have multiple edges between any pair of vertices.

1.9 Topological Sorting

In graph theory, a **topological sort** or topological ordering of a directed acyclic graph (DAG) is a linear ordering of its vertices such that for every directed edge $u \to v$, vertex u comes before v in the ordering. A topological sort is only possible if the graph has no directed cycles, meaning it is a DAG. This sorting is particularly useful in scenarios where a series of tasks must be completed in a specific order due to dependencies.

Formally, given a directed acyclic graph $G = (V, E)$, where V is the set of vertices and E is the set of edges, topological sorting arranges the vertices into a sequence such that for each edge $(u, v) \in E$, u appears before v in the ordering.

The algorithm for topological sorting can be implemented using **Kahn's algorithm** or **Depth-First Search (DFS)**.

The algorithm for topological sorting A basic approach to perform topological sorting is using DFS:

1. Start with any unvisited node and perform a DFS traversal.
2. When visiting a node, recursively visit all its unvisited neighbors.
3. After visiting all neighbors, add the node to the topological order.

4. Once all nodes have been processed, reverse the order and voila, you have **a** topological sort. ('**a**' made bold because there might be multiple possible topological sortings for a given directed edge graph)

Following are some of the real life applications of topological sorting:

- Task Scheduling and Dependency Resolution: Topological sorting is widely used in project management for scheduling tasks that depend on one another. For example, for a given set of tasks where some tasks must be performed before others, topological sorting helps in determining a valid sequence of execution of tasks.
- Build Systems: Modern build systems such as `Make` or `CMake` use topological sorting to resolve dependencies between files. When building large software projects, source files and libraries often depend on one another. By modeling these dependencies as a graph, the build system uses topological sorting to determine the correct order in which to compile and link the files.

  Package Managers: Similar to build systems, package managers such as `apt` in Linux or `npm` for JavaScript, use topological sorting to install (or more aptly import/ load maybe) software packages in the correct order. Packages often rely on other packages (dependencies) to function. A DAG can represent these dependencies, and a topological sort ensures that dependent packages are installed before the packages that require them.

2 The Travelling Salesman Problem

The Traveling Salesman Problem (TSP) problem is simple to state but is very difficult to work on (computationally): Given a list of cities and the distances between each pair of cities, the goal is to find the shortest possible route that visits each city exactly once and returns to the starting city.
Despite its simple formulation, TSP is NP-hard which means that we know no way (algorithm) to solve this problem in polynomial time(complexity) as of now (fingers crossed). Makes me feel grateful for google maps more than anything else. Also, we can see that since in the brute force algorithm we must go through all possible sequences of visiting cities to determine the most cost efficient one, the complexity of the algorithm is $O(n!)$. This is the worst possible scenario we can have and the time increases where quickly even when going from 4 or 5 nodes to 8 nodes.

As for how do we frame this problem using graphs, the cities will be represented by vertices and the distances between them as the weight of the edges connecting the corresponding vertices. Our task is essentially to find a hamiltonian circuit with the least path weight/cost to the salesman.

In the interest of our poor salesman we come up with a compromise to solve the problem. Since brute force algorithm takes too much time, we choose a sub optimal but still acceptably good path through the cities. Christofides algorithm is one such algorithm; it is an approximate algorithm which guarantees that the path it finds with be at most a factor of 1.5 off the length of the actual optimal path. It makes use of the triangle inequality (the sum of the lengths of any two sides will always be greater than the individual length of the third side). It first constructs minimum spanning tree T of the given graph G. Then it makes a set of all odd-degreed vertices, let's call it O. From **handshake lemma** we know that the cardinality of O must be even. Then it finds a minimum-weight perfect matching M in the subgraph induced in G by O after which it combines

the edges of M and T to form a connected multigraph H in which each vertex has even degree. Now we form a Eulerian path in H and make the circuit into a Hamiltonian circuit by skipping repeated vertices.

We also have the greedy approach where at every point we have to make a decision, we will choose to go to the city which incurs the least cost (that is, it is the closest city from the current position). The code for the brute force method has been given in the repo.

```
    #include <bits/stdc++.h>
using namespace std;
#include<time.h>
#include<vector>
#define V 4 //number of cities / vertices

int TSP(int graph[][V], int s)
{
    vector<int> cities;

    for (int i = 0; i < V; i++)
        if (i != s)
            cities.push_back(i);

    // minimum weight Hamiltonian Cycle.
    int min_path = INT_MAX;
    do
    {
        int current_pathweight = 0;
        int k = s;

        for (int i = 0; i < cities.size(); i++) {
            current_pathweight += graph[k][cities[i]];
            k = cities[i];
        }
        current_pathweight += graph[k][s];

        min_path = min(min_path, current_pathweight);

    }
    while(next_permutation(cities.begin(), cities.end()));

    return min_path;
}

// Driver Code
int main()
{
    // clock_t start, end;
    // double timeTaken;
```

```
    int graph[4][4] = { { 0, 10, 15, 20 },
                        { 10, 0, 35, 25 },
                        { 15, 35, 0, 30 },
                        { 20, 25, 30, 0 } };

    // int graph[5][5] = { { 0, 10, 15, 12, 17 },
    //                     { 10, 0, 35, 25, 23 },
    //                     { 15, 35, 0, 30, 15 },
    //                     { 12, 25, 30, 0, 36 },
    //                     { 17, 23, 15, 36, 0 } };
    int s = 0;
    // start = clock();
    cout << TSP(graph, s) << endl;

    // end = clock();
    // timeTaken = ((double) (end - start)) / CLOCKS_PER_SEC;

    // std::cout<<end<<endl;
    // std::cout<<start<<endl;
    // std::cout<<"Time taken: "<<timeTaken<<" seconds"<<std::endl;
    return 0;
}
```