Graph Theory

Self Reading

Abhilasha

2024

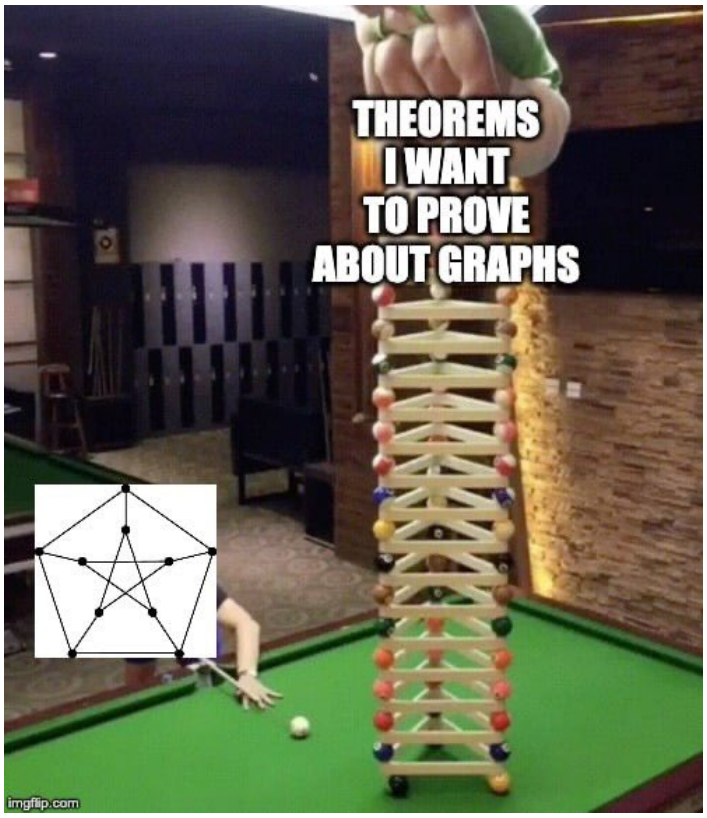

Figure 1

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 a set of vertices V and a set of edges E.
When we talk about the 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 the 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 starting from any vertex, you have a path to reach every other vertex.
2. Simple Graph: A graph in which any pair of vertices has 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.
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 connecting two vertices from the same bipartition set.
   - 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)
   - **Complete bipartite graphs**: bipartite graphs in which a partition's vertices are connected to all the vertices of the other partition.
6. A cyclic graph contains at least one cycle.
7. **Hamiltonian Cycles**: A cycle that visits **each vertex** in a graph exactly once.

1.5 Representing Graphs

Graphs can be represented in multiple ways depending on what information we need 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 Some Other Concepts (IMPORTANT)

1. Perfect Matching: A subset of edges that covers all vertices. Eg: Placing CCTV cameras in a network of streets to cover all streets at a given time
2. Multigraph: A graph in which we can have multiple edges between any pair of vertices.
3. Handshake Lemma: It states that the number of odd degreed vertices in a graph must be even (This is because the sum of degrees of all the vertices is twice the number of edges in the graph, beyond that use the magnificent tissue in your skull).
4. Spanning tree: A subgraph that happens to be a tree, in the minimum spanning tree we only want to include the least weighted edges
5. Circuit vs cycle: Both circuits and cycles must start and end at the same vertex but in a cycle, we cannot have repeating edges, we cannot repeat edges in either
6. **Eulerian tour**: Traverse **all edges** exactly once and return to the starting vertex
7. Tree: A connected graph with no cycles. A forest is a disjoint union of trees

1.7 Algorithms

1.7.1 Breadth-First Search (BFS)

BFS is a **traversal algorithm** that starts from a given node, explores all nodes at a specific distance from that node and moves 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);
            }
        }
    }
}
```

1.7.2 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 a 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);
}
```

1.7.3 Kruskal's Algorithm

This algorithm is used to make minimum spanning trees of a given graph. MSTs are useful in finding the least weighted paths in a graph from a given node to another graph. It starts by ordering all the edges in the order of increasing weights and then keeps adding edges starting from the least weighted edge which doesn't create a cycle and keeps adding edges like this. In case some edge creates a cycle it is not added. It gives the optimal solution but the solution thus achieved can be not unique.

This algorithm is a greedy algorithm and it's complexity of $O(|E|\ log|V|)$

```
        #include <bits/stdc++.h>
using namespace std;

class DSU {
    int* parent;
    int* rank;

public:
    DSU(int n)
    {
        parent = new int[n];
        rank = new int[n];

        for (int i = 0; i < n; i++) {
            parent[i] = -1;
            rank[i] = 1;
```

```
        }
    }

    int find(int i)
    {
        if (parent[i] == -1)
            return i;

        return parent[i] = find(parent[i]);
    }

    void unite(int x, int y)
    {
        int s1 = find(x);
        int s2 = find(y);

        if (s1 != s2) {
            if (rank[s1] < rank[s2]) {
                parent[s1] = s2;
            }
            else if (rank[s1] > rank[s2]) {
                parent[s2] = s1;
            }
            else {
                parent[s2] = s1;
                rank[s1] += 1;
            }
        }
    }
};

class Graph
{
    vector<vector<int> > edgelist;
    int V;

public:
    Graph(int V) { this->V = V; }


    void addEdge(int x, int y, int w)
    {
        edgelist.push_back({ w, x, y });
    }

    void kruskals_mst()
    {
```

```
        sort(edgelist.begin(), edgelist.end());

        DSU s(V);
        int ans = 0;
        cout << "Following are the edges in the MST"<< endl;
        for (auto edge : edgelist) {
            int w = edge[0];
            int x = edge[1];
            int y = edge[2];

            // Take this edge in MST if it does
            // not forms a cycle
            if (s.find(x) != s.find(y)) {
                s.unite(x, y);
                ans += w;
                cout << x << " -- " << y << " == " << w
                     << endl;
            }
        }
        cout << "Minimum Cost Spanning Tree: " << ans;
    }
};

int main()
{
    Graph g(4);
    g.addEdge(0, 1, 10);
    g.addEdge(1, 3, 15);
    g.addEdge(2, 3, 4);
    g.addEdge(2, 0, 6);
    g.addEdge(0, 3, 5);

    g.kruskals_mst();

    return 0;
}
```

1.8 Advantages and Disadvantages of Graphs

1. Advantages

   - Efficient representation of complex relationships.
   - Traversal algorithms are very useful in implementation of other algorithms 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.

1.9 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 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 neighbours.
3. After visiting all neighbours, 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)

But here is a much much better algorithm:

Kahn's Algorithm

Go through all the nodes and create an array containing the indegrees of all the nodes (indegree is the number of incoming edges).

Starting from 0, queue the elements with a specific indegree (not all at once indegree=n_0, then process and queue again). On queuing, store these elements in some container and then move on to the successive value of the current indegree and keep repeating the process until we can't go on.

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

– Task Scheduling and Dependency Resolution: Topological sorting is widely used in project management to schedule tasks that depend on one another. For example, for a given set of tasks where some must be performed before others, topological sorting helps determine a valid sequence of task execution.

- 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. Modelling these dependencies as a graph, the build system uses topological sorting to determine the correct order to compile and link the files.

  Package Managers: Similar to build systems, package managers such as `apt` in Linux or `npm` in 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 those requiring them.

2 Connectivity in Graphs

Seems easy, right? You're in for a ~~realisation~~ treat! Now onto serious stuff.

Connectivity is the stepping stone to reachability as we use graphs to represent much messier/-more complex ideas/concepts. In graphs, it indicates a path's presence between two parts (components/vertices) of the graph.

2.1 Some Terms

- Cut Edges: edges in a graph, whose deletion will cause the count of SCCs to increase. These are sort of 'bridges' between the two connected components. An edge cut should not be a part of any cycles.
- Cover: A graph G's cover is a family f of subgraphs of G.

  $$\bigcup_{F \in f} E(F) = E(G)$$

  A graph with no cut edges has a cyclic cover; all F are cyclic. A uniform cover covers each edge of the graph k times. A 1 cover is essentially a decomposition of the graph.

  The cycle double cover conjecture says that every bridgeless graph contains every edge twice. The proof of this continues to be an open problem but we can restrict ourselves to connected graphs to prove this.
- Cut Vertices: A vertex on removing which the number of connected components increases.
- A connected graph (of order ≥ 3) has no cut vertices iff $\forall$ u, v ϵ V (G), $\exists$ 2 internally disjoint uv paths in G. A connected graph of order greater than or equal to 3 if and only if for any pair of vertices, there are two disjoint paths uv.
- A vertex is a separating vertex if the graph can be divided into two graphs G_a and G_b either of whose edge sets aren't empty such that the intersection of the vertices set of both the graphs is the vertex itself and that of the edge sets is a null set.
- Non-separable graphs are graphs that have no separating vertices.
  - All non-trivial non-separable graphs have no cycles.

  - An acyclic graph is non-separable iff its underlying simple graph is non-separable. This implies that multiple edges don't play any role in separability.

- A connected graph G is non-separable iff any 2 edges of the graph participate in a common cycle.
- A **block** is a maximal non-separable subgraph of a graph.
  - Any 2 blocks can have at most 2 vertices in common and it one such vertex exists then it is a separating vertex.
  - Each of the graph's cycles is contained in a block of the graph.
  - Blocks of the graph comprise the decomposition of the graph.
  - Internal vertices of a block are the non-seperating vertices of the block.

2.2 Decomposition of Graphs

This is exactly what it sounds like, we break down the graph into smaller parts. There are multiple ways to do so, we could decompose the graph into constituent cycles, blocks, paths, star graphs, Hamiltonian cycles or acyclic paths. The list goes on, but we will discuss only a few here.

- **Block Decomposition**: Decomposing the graph into blocks. Let's look at this graph as an example:

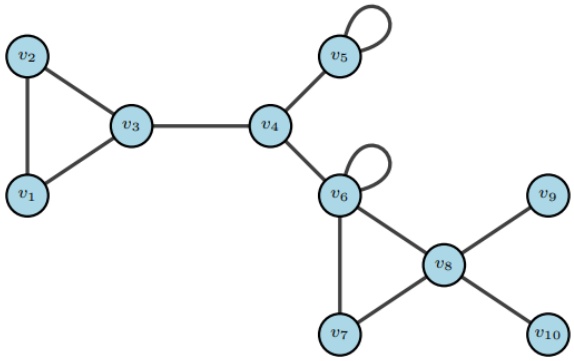

Figure 2: Block decomposition example

We can see that the given graph has 9 blocks [[v1,v2,v3],[v3,v4],[v4,v5],[v6,v7,v8],[v4,v6], [v8,v10],[v8,v9],[v6],[v5]]. Please note that there can be multiple possible decompositions. This is just the one that came to me.

In the shown example, we can observe how the loops are considered blocks of the graph. The separating vertices in this example are v3, v4, v5, v6, v8. Notice how they are the single common vertices between blocks (as we discussed above in the definition of blocks, if any groups have a common vertex, first of all, they can have at most one, which must be the separating vertex). From this realisation, we will now construct a bipartite graph that has the set of separating vertices as one of its partitions. For vertices of the other partition, we create new vertices from the existing blocks and delete all the internal vertices of the blocks from the original graph and declare the set of the newly formed vertices the other partition of the previously discussed bipartite graph. This is the **Block Tree**.

  - If the underlying graph is separable, blocks corresponding to the leaves of the block tree are called the **End Blocks**, that is, they have exactly one separating vertex between them.

- **Hamiltonian Decomposition**:

  - Quick back story: Kotzig conjecture states that the cartesian product of any two cycles can be decomposed into two Hamiltonian cycles. The cartesian product of two graphs is a graph whose vertex set is the cartesian product of the vertex sets of the 2 earlier graphs. Each term in this product now represents a new vertex This is better illustrated in the figure below

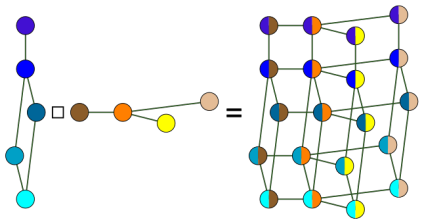

Figure 3: Cartesian Product of Graphs

  - A graph is considered decomposable into Hamiltonian cycles if its edge set can be partitioned into Hamiltonian cycles. We show that the cartesian product of any three cycles can be decomposed into three Hamiltonian cycles, thus the earlier stated conjecture.
    In the case of undirected graphs, the decomposition can also be considered a 2-factorization of the graph such that each factor is connected. (What is the k-factorization of graphs? A factor of a graph G is a spanning subgraph, i.e., a subgraph with the same vertex set as G. A k-factor of a graph is a spanning k-regular subgraph, and a k-factorization partitions the edges of the graph into disjoint k-factors. A 1-factor is a perfect matching, and a 1-factorization of a k-regular graph is a proper edge colouring with k colours. A 2 factor particularly is a collection of cycles that span the entire vertex set of the original graph).

3 Planarity

Planarity deals with how a graph can be drawn in the plane without edge crossings. A graph is **planar** if it can be drawn in the plane such that **if any two of its edges meet**, they do so **only at their endpoints**. There are a few definitions that we must be aware of,

- A graph is planar if it can be drawn in a plane such that any two of its edges if they meet, meet only at their endpoints.

- A curve is a continuous set of points that look like a straight line between two *related points*. A closed curve is a curve that encloses some area. A simple curve does not intersect with itself.

- A topological space E is said to be arcwise connected if, for every pair of points p and q of E there is a path in E joining p and q

- Any simple closed curve C in the plane partitions the rest of the plane into 2 disjoint arcwise connected open sets. This is the **Jordan Curve Theorem**

3.1 Some terms

- **Faces**: It is a region in the plane that is bounded by edges and vertices, and is not divided into smaller areas. More formally, a plane is partitioned into multiple arcwise connected open sets by a planar graph. These open sets are the faces and the boundary of these open sets is the boundary of the faces. In all non-separable graphs except K_1 and K_2, each face is bound by a cycle.
- **Dual of a graph**: A dual of a graph is constructed by creating vertices corresponding to individual faces of the graph; an edge between any of these vertices indicates that the underlying faces were separated by an edge in the original graph.

4 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 the brute force algorithm goes through all possible sequences of visiting cities to determine the most cost-efficient one, the algorithm's complexity is $O(n!)$. This is the worst possible scenario we can have and the time increases very quickly even when going from 4 or 5 nodes to 8 nodes.

As for how 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 the 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 that guarantees that the path it finds will be at most a factor of 1.5 off the length of the actual optimal path. It uses 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 the 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 an even degree. Now we create an 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 decide, we will choose to go to the city that incurs the least cost (that is, it is the closest city to 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;
}
```

5 Some random ideas that might be useful

1. **Subdivision of Edges**: Transforming an edge into a path of length 2 by creating a frivolous vertex between the existing vertices and making it a sort of buffer vertex between the 2 original vertices.