iJournals: International Journal of Software & Hardware Research in Engineering
ISSN:2347-4890, ijournals.in/ijshre
Volume 12 Issue 8, August 2024

Analysis of Graph Isomorphism with Examples and Implementation of VF2++ Algorithm

Author: Esthelle Chung

Chadwick International School

echung2025@chawickschool.org

DOI: 10.26821/IJSHRE.12.8.2024.120801

ABSTRACT

Discrete mathematics, with graph theory as a key component, explores relationships among countable, distinct objects. Graphs, comprising vertices and edges, are essential for analyzing connectivity and processing discrete data across various fields like computer science, network theory, transportation, and social sciences. In practical applications such as electrical circuits, organic chemistry, and transportation networks, graph theory facilitates analysis and optimization. For instance, in electrical circuits, nodes and edges represent components and connections, while in organic chemistry, molecules are graphically depicted for structural analysis. Similarly, in transportation networks like airports, graph theory aids in optimizing flight schedules to minimize delays and congestion. In this research, basic terms and the notations are described, and important background observations and the definitions are followed. Detail of the isomorphism and bipartite is covered. Also, computational algorithm of VF2++ is introduced. Finally, the paper concludes with the future research works with MATLAB.

Keywords: graph theory, graph isomorphism, VF2++

1. INTRODUCTION

Discrete mathematics is one significant mathematic field that studies relationships between countable, distinct, and separate objects [1]. Of its key components, graph theory plays a critical role. Graph theory is employed to abstract and analyze relationships and connectivity between different objects, making it a suitable tool for processing discrete data and events. Graphs consist of vertices and edges that possess discrete properties, thus being a fundamental concept in discrete mathematics. Graph theory is applied in various fields, including computer science, network theory, transportation, and logistics, as well as data analysis in the social sciences: all of which contribute to researching and comprehending diverse aspects of modern mathematics in the real world.

Graph theory plays a significant role in various modern infrastructures or sciences, including electrical circuits, organic chemistry, and transportation networks such as airports. In electrical circuits, graph theory can be employed to represent circuit components as nodes, or vertices, and electrical connections as edges, which allows for the creation of circuit diagrams to analyze and manipulate in intended directions. In organic chemistry, molecules can be represented as graphs or networks, atoms being nodes and chemical bonds being edges [2]. This representation can simplify and visualize the analysis of molecular structures. In airports, flights are represented as nodes, and connections between flights as edges, in which graph theory can model and optimize flight schedules to reduce delays or congestion.

Isomorphism in graph plays important role in several fields [3]. One significant application of graph comparison is in identifying duplicate web pages, such as those associated with illicit activities like human trafficking. With the vastness of the web, manual monitoring is impractical, so algorithms can create text graphs of pages and compare them for similarities, detecting near- identical matches even if minor changes occur. Another use case is community detection on the web, where algorithms analyze interconnected networks to identify clusters like friend groups. By comparing subgraphs of larger networks to

smaller ones representing specific communities, algorithms can discern potential matches, aiding in identifying cohesive groups within the broader network.

The VF2++ algorithm, building upon the VF2 approach, incorporates new cutting rules and optimizes node access order to enhance efficiency. Unlike its predecessor, VF2++, is non-recursive, leading to savings in both time and space [4]. Optimal node ordering is determined by considering node degree and label rarity, prioritizing nodes with higher matching likelihood. This strategic ordering allows for the examination of more promising branches early in the process, while rules based on node labels aid in pruning unproductive branches, further streamlining the algorithm.

Basic terms and the notations are described in chapter II. Important background observations and the definitions are followed in chapter III. In chapter Iv, detail of the isomorphism and bipartite is covered. In chapter V, computational algorithm of VF2++ is introduced. Finally, the paper concludes with the future research works in the last chapter.

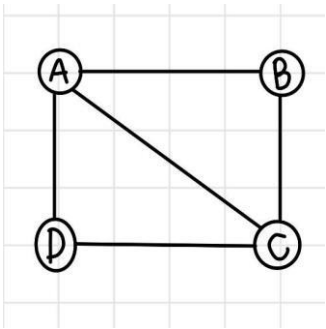
2. TERMS AND NOTATIONS

Table 1. List of Definitions

Term	Definition
Isomorphism	Structural similarity or equivalence between two mathematical figures (i.e. graphs, groups, functions), where the two have different representations but similar properties
Adjacency Matrix	square matrix representing adjacent vertices in a graph • If edge exists, 1 • If no edge, 0
Incidence Matrix	matrix representing relationship between vertices and edges • rows: vertices • column: edges • entry: whether vertex is incident to edge
Subgraph	graph formed by selecting a subset of vertices and subset of edges from a larger graph set
Degree	describes number of edges connected to vertex 1. vertex degree (node degree): number of edges incident to particular vertex 2. in-degree: number of edges directed towards vertex 3. out-degree: number of edges originating from vertex
Path	describes sequence of vertices within a graph, used to traverse + find routes within the graph
Cycle	closed path within a graph that starts and ends at same vertex, without repeating any vertexes or edges within the route
Walk	sequence of vertices and edges that enable a move from one vertex to another

Considering **Fig. 1** below, each term in **Table 1** is described with examples.

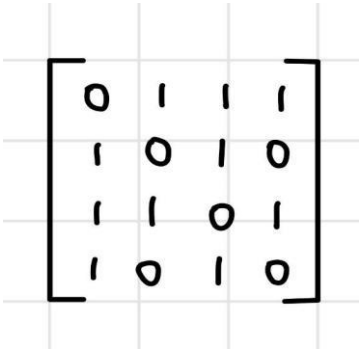
Fig 1: Example Graph of A-B-C-D



2.1.1 Adjacency Matrix

The adjacency matrix of **Fig. 1** is:

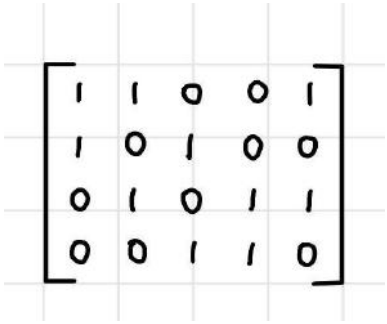
Fig 2: Adjacency Matrix of Figure 1



2.1.2 Incidence Matrix

The incidence matrix of **Fig. 1** is:

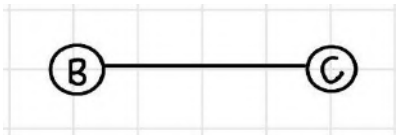
Fig 3: Incidence Matrix of Figure 1



2.1.3 Subgraph

The subgraph of **Fig. 1** is:

Fig 4: Subgraph of Figure 1



2.1.4 Degree

The degree of **Fig. 1** is:

- A: 3
- B: 2
- C: 3
- D: 2

2.1.5 Path

The path of **Fig. 1** is:

- A→ D: A-B-C-D

2.1.6 Cycle

The cycle of **Fig. 1** is:

- A, B, C, D: A-B-C-D-A

2.1.7 Walk

The walk of **Fig. 1** is:

- B → C: B-C

3. BACKGROUND KNOWLEDGE

3.1 Theorem 1. Walk Theorem

Let A^n be an adjacency matrix A with m nodes of graph G. Then, elements (i,j) of A^n represent the number of walks with distance n from node v_i to node v_j.

3.1.1 Proof

i) Let n=1.

If selecting (i,j) from A^1, output is either 1 or 0; 1 represents the number of direct edges connecting v_i and v_j, which is equal to walks with distance 1. Thus, the proposition holds true.

ii) Assume that proposition is true for some positive integer k, i.e., elements (i,j) of A^{k-1} represent the number of walks with distance k-1 from node v_i to node v_j.

iii) n=k.

Consider $A^k = A^{k-1} \cdot A$

Fig 5: Matrix Calculation of $A^{k-1} \cdot A$

$A^{k-1} \cdot A$	=	$a^{k-1}_{1,1}$, $a^{k-1}_{1,2}$, ..., $a^{k-1}_{1,n}$ $a^{k-1}_{2,1}$, $a^{k-1}_{2,2}$, ..., $a^{k-1}_{2,n}$ ... $a^{k-1}_{n,1}$, $a^{k-1}_{n,2}$, ..., $a^{k-1}_{n,n}$	$a^{1}_{1,1}, a^{1}_{1},2, ..., a^{1}_{1,n}$ $a^{1}_{2,1}, a^{1}_{2,2}, ..., a^{1}_{2,n}$ ... $a^{1}_{n,1}, a^{1}_{n,2}, ..., a^{1}_{n,n}$

Each entry (i,j) of A^k is calculated as the sum of products of the ith row of matrix A^{k-1} and the jth column of matrix A, which represents the number of walks from v_i to v_j with a distance of k+1 by taking a k-length walk from v_i to some node v_p and taking a single step from v_p to v_j. Therefore, using the inductive hypothesis that (i,j) of A^{k-1} represents the number of walks of distance k-1 from v_i to v_j, we can conclude that (i,j) of A^k represents the number of walks of distance k from v_i to v_j.

Thus, the theorem holds true. ■

3.1.1.1 Example 2

Let A^1 = adjacency matrix from Figure 2. If k=3, A^3 is:

Fig 6: Adjacency Matrix of A^3

4	5	5	5
5	2	5	2
5	5	4	5
5	2	5	2

Assume we pick element (1,4) from A^3. By the Walk Theorem, (1,4), or 5, represents the number of walks of distance 3 from v_1 to v_4.

In Figure 1, v_1 = A and v_4 = D. There are 5 total walks of distance 3 from A to D.

- A-B-C-D
- A-D-C-D
- A-B-A-D
- A-C-A-D
- A-D-A-D

3.2 Theorem 2. Handshaking Theorem

$$\sum d(v) = 2\,|E|$$

3.2.1 Proof

Consider each row in adjacency graph as $s1, s2, \ldots, s|v|$. Then, $\sum d(v) = \sum si$. Because the adjacency graph represents the connection between each node, $\sum si = |E|$.

Thus, $\sum d(v) = 2\,|E|$ holds true. ■

3.2.1.1 Example 3

Consider **Fig. 1**. Its degrees have been counted above as following:

Degree:

- A: 3
- B: 2
- C: 3
- D: 2

First, the sum of all the degrees of all the vertices is 3+2+3+2=10.

The number of edges in the graph is 5.

According to the Handshake Theorem, the sum of all the degrees of all the vertices should be twice the number of edges in the graph. 10=2*5; thus, the theorem holds true.

3.3 Definition 1. Regular Graph

A regular graph is a graph where all vertices have the same degree, and an equal number of edges connected to it. For example, in <*figure 1*>, if B-D is connected, it becomes a 3-regular graph.

- Complete Graph K^n = (n-1)-regular graph. For example, k^3: 2-regular graph

3.3.1.1 Example 4

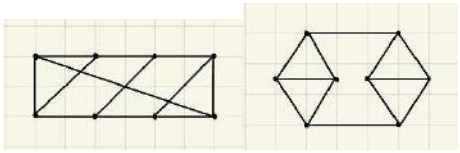


Fig 7: Two 3-Regular Graphs with 8 Vertices

Observation 1. E = vC2 is a conditional for G to be a complete graph.

Proof.

If K^n is a complete graph, the number of edges E_n is equal to v_nC2, where v_n represents the number of vertices.

i) Base case (n=1)

If n=1, there is 1 vertex and no edges. (1 2) = 0; thus, the proposition holds true.

ii) Inductive hypothesis

If n=k, assume the number of edges $E_k = v_kC2$. The number of edges for K^n = n(n-1)2 or nC2. Thus, for K^k, E_k = kC2 = v_kC2.

iii) Inductive step

Assume n=k+1.

The number of edges in K^{k+1} is equal to the number of edges in K^k plus the number of new edges formed by the (k+1) th vertex. Thus, $E_{k+1} = E_k + k = kC2 + k$.

kC2 + k = k(k-1)2+k=k(k-1) + 2k2=k2-k+2k2=k2+k2=k(k+1)2 = (k+1) C2.

Thus, E_{k+1} = (k+1) C2 = $v_{k+1}C2$.

By induction, the statement holds true for all positive integers n.

4. BIPARTITE AND ISOMORPHISM

4.1 Injective Function

If for a function f: X -> Y from set X (domain) to set Y (range), $f(x_1) = f(x_1)$ for elements x_1 and x_2 of set X, function f is an injective function.

4.1.1 Example 5

$f(x) = x^3$ for domain and range all real numbers.

If $f(a) = a^3 = b^3 = f(b)$, $a^3-b^3 = (a-b)(a^2+ab+b^2) = 0$

Then, a=b or a = -b-3b22. Since a and b are all real numbers, a=b. Thus, injective function.

4.2 Surjective Function

If for a function f: X -> Y from set X (domain) to set Y (range), y = f(x) for element y of set Y, function f is a surjective function.

4.2.1 Example 6

$f(x) = x^3$ for domain and range all real numbers.

For a real number y, f(3y) = y. Thus, surjective function.

4.3 Bijective Function

If for a function f: X -> Y from set X (domain) to set Y (range), f(x) is both injective and surjective, function f is a bijective function.

4.3.1 Example 7

$f(x) = x^3$ for domain and range all real numbers.

From the two proofs above, f(x) is both injective and surjective. Thus, bijective function.

4.4 Inverse Function

If function f: X -> Y from set X (domain) to set Y (range) is a bijective function, there exists a unique bijective function g: Y -> X such that: f(g(y)) = y for all elements y of set Y.

Function g is the inverse function of f, also written as f^{-1}.

4.5 Isomorphism

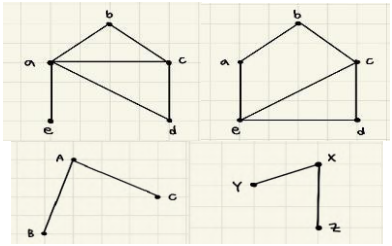
For vector space V and W, if there exists a bijective linear transformation between V and W, V is isomorphic to W. The bijective linear transformation between V and W is isomorphism.

Observation 2. If, for G_1 (V_1, E_1) and G_2 (V_2, E_2), there exists a one-to-one function f: $v(G_1)$ -> $v(G_2)$, G_1 and G_2 are isomorphic.

Proof. If G and H are isomorphic, v(G) = v(H) and E(G) = E(H).

4.6 Bipartite Graph

Let graph G = (V, E). G is a bipartite graph if and only if all its edges connect a vertex from set V_1 and V_2. Thus, there are no adjacent vertices in V_1 and V_2; if



two vertices are adjacent, one is in V_1 and one is in V_2.

Fig 8: Examples of Bipartite Graphs

Assume G is a bipartite graph with two sets of vertices: V_1 and V_2. where the number of vertices in both V_1 and V_2 are v_1 and v_2 . The number of edges in G is, e and the total number of vertices is v.

By definition, a bipartite graph connects vertices from V_1 and V_2; no edges exist within V_1 or V_2. Therefore, the maximum number of edges in G is achieved when each vertex in V_1 is connected to all vertices in V_2, $v \leq v1 \cdot v2$.

Based on arithmetic-geometric mean, $v1 \cdot v2 \leq (\frac{a+b}{4})^2$.

5. VF2++ Algorithm

5.1 VF Algorithm

Here are the steps of VF algorithm:

1) Set mapping m (empty mapping)
2) Compute the set of candidate pairs for the extension of the current mapping m.
3) Evaluate the pairs, Consists or Cut.
4) If the Consists function returns true, the whole process is applied to extension recursively.
5) Or, return false of the whole process.

5.2 VF2++ Algorithm

VF2++ improves upon VF2 by efficiently determining a matching order that considers the graph's structure and node labeling, leading to immediate pruning of unfruitful branches in the search space [5]. Additionally, it introduces more efficient cutting rules, further reducing the chance of straying off course.

Specialized versions for induced subgraph and graph isomorphism problems have also been developed. The goal is to recognize inconsistencies or prune infeasible branches at higher levels, delving deeper only when necessary.

5.3 Implementations

1) Computes a BFS: in ascending order of rareness.
2) Appends the node of the current level in descending order.

main

```
def main(self, f1, f2):

    self.__origin = GraphSet(f1)
    self.__sub = GraphSet(f2)


    subLen = len(self.__sub.graphSet())
    gLen = len(self.__origin.graphSet())

    for i in range(subLen):
        for j in range(gLen):
            result = {}
            result = self.dfsMatch(i, j, result)
            if len(result) == len(self.__sub.curVSet(i)):
                print "Match! %s %d-th graph isomorphism %s %d-th graph!" %(f2, i, f1,
                print result
                print "~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~"
            else:
                print "Mismatch! %s %d-th graph isomerism %s %d-th graph!" %(f2, i, f1
                print "~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~"
```

bfs

```
def dfsMatch(self, i, j, result):
    #print "in dfsMatch() result: ", result
    if not isinstance(result, dict):
        print "Class VF dfsMatch() arguments type error! result expected dict!"

    curMap = Map(result)


    if curMap.isCovered(self.__sub.curVSet(i)):
        print "yes!"
        return result


    subNeighbor = curMap.neighbor(i, self.__sub, 0, True)
    gNeighbor = curMap.neighbor(j, self.__origin, 1, True)

    if not (subNeighbor and gNeighbor):
        print "Class VF dfsMatch(), subNeighbor or gNeighbor is empty!"
        exit()

    subNNeighbor = curMap.neighbor(i, self.__sub, 0, False)
    gNNeighbor = curMap.neighbor(j, self.__origin, 1, False)
    #print "in dfsMatch() subNNeighbor: ", subNNeighbor
    #print "in dfsMatch() gNNeighbor: ", gNNeighbor

    #notice, choose one vertex in subgraphNeighbor is ok
    while(len(subNNeighbor) > 1):
        subNNeighbor.pop()

    pairs = self.candidate(subNNeighbor, gNNeighbor)
    if not pairs:
        return result

    for pair in pairs:
        v1, v2 = pair.strip().split(":")
        if(self.isMeetRules(int(v1), int(v2), i, j, result, curMap.subMap(), curMap.gM
            result[int(v1)] = int(v2)
            self.dfsMatch(i, j, result)
            #notice, it's important to return result when len(result) == len(self.__su
            #otherwise it will continue to pop
            if len(result) == len(self.__sub.curVSet(i)):
                return result
            result.pop(int(v1))
    return result
```

6. Conclusion and Future Works

For future works, I am planning to evaluate several more algorithms on isomorphism in the aspect of time complexity and the efficiency.

7. REFERENCES

[1]. “Discrete mathematics is a branch of mathematics that deals with countable”, University of Moratuwa

[2]. Fang, X., Liu, L., Lei, J. et al. “Geometry-enhanced molecular representation learning for property prediction.” Nat Mach Intell 4, 127–134 (2022).

[3]. Sathya Selvarajan, “Graph Isomorphism in Computer Science”, 2023

[4]. Isomorphism#. Isomorphism - NetworkX 3.3 documentation. (n.d.).

[5]. Jüttner, Madarasi, “VF2++—An improved subgraph isomorphism algorithm”, 2018, Pages 69- 81