A Framework for Comparing Directed Multigraphs Project Report: Algorithms and Computability

Jerzy Czarnecki Yurii Demoshenko Mateusz Małachowski Patryk Zan

December 11, 2025

Abstract

This report presents a formal framework for comparing two directed multigraphs G and H. We develop definitions for graph size, a graph distance metric based on Graph Edit Distance, and the minimal extension of a graph. We describe exact algorithms for subgraph isomorphism and minimal extension problems, analyze their computational complexity, and propose polynomial-time heuristic algorithms for practical applications. Our heuristics address finding n distinct subgraph isomorphisms and constructing minimal extensions that support n distinct embeddings. Since the underlying problems are NP-hard, our heuristics trade optimality guarantees for computational tractability.

Contents

1 Introduction

The comparison of graph structures is a fundamental problem in computer science with applications ranging from bioinformatics [5] to network analysis [3] and pattern recognition [2]. This project addresses the specific task of comparing two directed multigraphs $G = (V, E)$ and $H = (U, F)$.

The primary objectives are to:

1. Develop a consistent definition for the 'size" of a graph.
2. Formulate a 'distance" metric to quantify dissimilarity between graphs.
3. Create a procedure to find n distinct subgraph isomorphisms from G to H.
4. If G is not a subgraph of H, define and find a minimal extension H' of H such that G can be embedded into H' in n distinct ways.
5. Develop polynomial-time heuristics, since exact algorithms have exponential complexity.

The subgraph isomorphism problem is known to be NP-complete [4], while the minimal extension problem is closely related to the Maximum Common Subgraph problem, which is NP-hard [7]. Therefore, we propose heuristic algorithms that run in polynomial time but do not guarantee optimal solutions.

2 Key Concepts and Definitions

2.1 Graph Definition

A **standard graph** is formally defined as a pair $G = (V, E)$, where V is a finite set of vertices and E is a set of edges, with each edge representing a two-element subset of V. The **degree** of a vertex v, denoted $\deg_G(v)$, is the number of edges connected to it.

A **multigraph** extends this concept by allowing multiple edges between the same pair of vertices. The number of edges between vertices u and v is captured by the **edge multiplicity function** $\text{mult}_G(u, v)$.

Our focus, a **directed multigraph**, further introduces directionality. Each edge is an ordered pair $(u, v) \in V \times V$, indicating direction from u to v. Importantly, (u, v) and (v, u) are distinct edges, so the adjacency matrix is **not symmetric**. Self-loops (v, v) are permitted. Each vertex has two distinct degrees:

- **In-degree** $\deg^- *G(v) = \sum *u \in V \text{mult}_G(u, v)$: edges directed into v
- **Out-degree** $\deg^+ *G(v) = \sum *u \in V \text{mult}_G(v, u)$: edges directed out of v

2.2 Graph Representation

We represent a directed multigraph using an **adjacency matrix** A_G where entry $A_G[i][j]$ stores $\text{mult}_G(v_i, v_j)$—the number of edges from vertex v_i to vertex v_j. For directed graphs, $A_G[i][j] \neq A_G[j][i]$ in general.

2.3 Graph Size

Definition 1 (Graph Size)**.** *The size of a graph $G = (V, E)$, denoted size(G), is defined as: $\lfloor$ size(G) = \|V\| + \|E\| $\rfloor$*

This definition directly relates to computational complexity analysis. Most graph algorithms have running times expressed as functions of $|V|$ and $|E|$ (e.g., $O(|V| + |E|)$ for BFS/DFS [6]). Using their sum as a size measure provides a single parameter capturing the input data volume.

2.4 A Metric for Graph Comparison

We adopt the **Graph Edit Distance (GED)** [10] as our distance metric. The GED between graphs G_1 and G_2 is defined as the minimum total cost of edit operations (vertex/edge insertions and deletions) required to transform G_1 into a graph isomorphic to G_2.

Definition 2 (Graph Edit Distance)**.** *Let $\mathcal{P}(G_1, G_2)$ denote all possible edit paths transforming G_1 to a graph isomorphic to G_2. For a cost function c assigning costs to individual operations, the GED is:*

$$d(G_1, G_2) = \min_{p \in \mathcal{P}(G_1, G_2)} \sum_{op \in p} c(op)$$

For this metric to be proper, the following conditions must hold:

2.4.1 Identity: $d(G, H) = 0 \iff G \cong H$

If $d(G, H) = 0$, no edit operations are needed, meaning G and H are isomorphic. Conversely, isomorphic graphs require no edits, so their distance is zero.

2.4.2 Positivity: $d(G, H) > 0 \iff G \ncong H$

If $d(G, H) > 0$, at least one edit operation is required, implying the graphs are not isomorphic. The converse follows from the identity property.

2.4.3 Symmetry: $d(G, H) = d(H, G)$

Each edit operation has an inverse: vertex insertion inverts deletion, and vice versa for edges. An optimal edit sequence from G to H can be reversed to transform H to G with identical cost.

2.4.4 Triangle Inequality: $d(F, G) + d(G, H) \geq d(F, H)$

Concatenating optimal sequences $F \to G$ and $G \to H$ yields a valid (though not necessarily optimal) sequence $F \to H$ with cost $d(F, G) + d(G, H)$. Since $d(F, H)$ is the minimum over all such sequences, the inequality holds.

2.5 Subgraph Isomorphism for Directed Multigraphs

Definition 3 (Subgraph Isomorphism)**.** *Graph $G = (V, E)$ is isomorphic to a subgraph of $H = (U, F)$ if there exists an injective mapping $f : V \to U$ such that for all pairs $(v_i, v_j) \in V \times V$:*

$$mult_G(v_i, v_j) \leq mult_H(f(v_i), f(v_j))$$

Note that for directed graphs, we check (v_i, v_j) and (v_j, v_i) independently since they represent edges in opposite directions.

Definition 4 (Distinct Isomorphisms)**.** *Two subgraph isomorphisms $f_1 : V \to U$ and $f_2 : V \to U$ are **distinct** if they differ on at least one vertex, i.e., there exists $v \in V$ such that $f_1(v) \neq f_2(v)$.*

Example 1 (Multiple Isomorphisms via Rotation)**.** *Consider directed cycles:*

$$G : v_1 \to v_2 \to v_3 \to v_1 \quad H : u_1 \to u_2 \to u_3 \to u_1$$

There are three distinct isomorphisms:

1. f_1*:* $v_1 \mapsto u_1, v_2 \mapsto u_2, v_3 \mapsto u_3$
2. f_2*:* $v_1 \mapsto u_2, v_2 \mapsto u_3, v_3 \mapsto u_1$
3. f_3*:* $v_1 \mapsto u_3, v_2 \mapsto u_1, v_3 \mapsto u_2$

2.6 Minimal Graph Extension

Definition 5 (Minimal Graph Extension for n Isomorphisms)**.** *Given graphs $G = (V, E)$ and $H = (U, F)$ with $|V| \leq |U|$, and a positive integer n, the **minimal extension** $H' = (U, F')$ of H with respect to G supporting n isomorphisms is formed by adding a set of directed edges E_{add} to H such that:*

1. *$F' = F \cup E_{add}$*
2. *There exist n distinct subgraph isomorphisms from G to H'*
3. *$|E_{add}|$ is minimized*

The output of the minimal extension algorithm consists of:

- The n vertex mappings $f_1, \ldots, f_n : V \to U$
- The set of edges to add: E_{add} with appropriate multiplicities
- The extended adjacency matrix $A_{H'}$

3 Algorithms and Complexity Analysis

This section describes the algorithms designed to compute the properties defined in Section 2. We present both exact algorithms (exponential complexity) and polynomial-time heuristics.

3.1 The Modular Product Graph

A key construction for subgraph isomorphism is the **modular product graph** (also called association graph), introduced by Levi [9]. This transforms the subgraph isomorphism problem into a maximum clique problem.

Definition 6 (Modular Product Graph)**.** *Given $G = (V, E)$ and $H = (U, F)$, the modular product graph $P = G \otimes H$ is constructed as:*

***Vertices of** P: All compatible pairs (v, u) where $v \in V$, $u \in U$, satisfying:*

$$\deg_G^+(v) \leq \deg_H^+(u) \quad \wedge \quad \deg_G^-(v) \leq \deg_H^-(u)$$

***Edges of** P: Two vertices (v_1, u_1) and (v_2, u_2) are connected if and only if:*

1. *$v_1 \neq v_2$ and $u_1 \neq u_2$ (injectivity condition)*
2. *$mult_G(v_1, v_2) \leq mult_H(u_1, u_2)$ (forward edge preservation)*
3. *$mult_G(v_2, v_1) \leq mult_H(u_2, u_1)$ (backward edge preservation)*

Theorem 1 (Clique-Isomorphism Correspondence)**.** *A clique of size k in the product graph $P = G \otimes H$ corresponds to a valid subgraph isomorphism of an induced subgraph of G with k vertices into H. In particular, a clique of size $|V(G)|$ corresponds to a complete subgraph isomorphism from G to H.*

Proof. Let $C = (v_1, u_1), (v_2, u_2), \ldots, (v_k, u_k)$ be a clique in P. Define mapping $f : v_1, \ldots, v_k \to U$ by $f(v_i) = u_i$.

Injectivity: Since any two vertices in C are adjacent in P, by the edge condition we have $u_i \neq u_j$ for all $i \neq j$.

Edge preservation: For any v_i, v_j in the domain, since (v_i, u_i) and (v_j, u_j) are adjacent in P:

$$\text{mult}_G(v_i, v_j) \leq \text{mult}_H(u_i, u_j) = \text{mult}_H(f(v_i), f(v_j))$$

Thus f is a valid subgraph isomorphism. When $k = |V(G)|$, all vertices are mapped. □

Algorithm 1 Construct Modular Product Graph

function BUILDPRODUCTGRAPH($G = (V, E), H = (U, F)$)
  $P_V \leftarrow \emptyset$ ▷ Vertices of product graph
  $P_E \leftarrow \emptyset$ ▷ Edges of product graph
  ‘‘‘
  for $v \in V$ **do**
    for $u \in U$ **do**
      if $\deg^+_G(v) \leq \deg^+_H(u)$ **and** $\deg^-_G(v) \leq \deg^-_H(u)$ **then**
        $P_V \leftarrow P_V \cup \{(v, u)\}$
      end if
    end for
  end for
  for $(v_1, u_1) \in P_V$ **do**
    for $(v_2, u_2) \in P_V$ **do**
      if $v_1 \neq v_2$ **and** $u_1 \neq u_2$ **then**
        if $\text{mult}_G(v_1, v_2) \leq \text{mult}_H(u_1, u_2)$ **then**
          if $\text{mult}_G(v_2, v_1) \leq \text{mult}_H(u_2, u_1)$ **then**
            $P_E \leftarrow P_E \cup \{((v_1, u_1), (v_2, u_2))\}$
          end if
        end if
      end if
    end for
  end for
  return (P_V, P_E) ‘‘‘
end function

3.1.1 Complexity of Product Graph Construction

Let $n = |V(G)|$ and $m = |V(H)|$.

- Number of vertices in P: at most $O(nm)$
- Number of potential edges: at most $O(n^2m^2)$
- Each edge check: $O(1)$ with adjacency matrix representation

Total construction time: $\mathbf{O(n^2m^2)}$

3.2 Exact Algorithm for Subgraph Isomorphism

The exact algorithm finds all subgraph isomorphisms by finding all cliques of size $|V(G)|$ in the product graph. We use a modified Bron-Kerbosch algorithm [1].

Algorithm 2 Exact: Find All Subgraph Isomorphisms

Global: *allMappings* $\leftarrow \emptyset$, *targetSize* $\leftarrow |V(G)|$
procedure FindAllIsomorphisms(G, H)
  if $|V(G)| > |V(H)|$ **then**
    return $\emptyset$
  end if
  $P \leftarrow$ BuildProductGraph(G, H)
  BronKerbosch($\emptyset, P_V, \emptyset, P$)
  return *allMappings*
end procedure
procedure BronKerbosch(R, C, X, P)
  if $|R| =$ *targetSize* **then**
    allMappings.add(ExtractMapping(R))
    return
  end if
  if $|R| + |C| <$ *targetSize* **then**
    return ▷ Cannot reach target size
  end if
  if $C = \emptyset$ **and** $X = \emptyset$ **then**
    return ▷ Maximal clique found but too small
  end if
  Choose pivot $p \in C \cup X$ maximizing $|N_P(p) \cap C|$
  for $v \in C \setminus N_P(p)$ **do**
    BronKerbosch($R \cup v, C \cap N_P(v), X \cap N_P(v), P$)
    $C \leftarrow C \setminus v$
    $X \leftarrow X \cup v$
  end for
end procedure

3.2.1 Complexity Analysis

The Bron-Kerbosch algorithm has worst-case complexity $O(3^{|P_V|/3})$ for finding all maximal cliques [1]. With $|P_V| = O(nm)$, the worst case is $O(3^{nm/3})$, which is exponential. This is expected since subgraph isomorphism is NP-complete.

3.3 Exact Algorithm for Minimal Extension

Finding the minimal extension supporting n isomorphisms requires searching over all combinations of n injective mappings.

Algorithm 3 Exact: Find Minimal Extension for n Isomorphisms

Global: $bestEdges \leftarrow \infty$, $bestMappings \leftarrow \emptyset$
procedure FindMinimalExtension(G, H, n)
    $allMappings \leftarrow$ all injective mappings $V(G) \to V(H)$
    SearchCombinations($\emptyset$, $allMappings, n, A_H$)
    return ($bestMappings$, $bestEdges$)
end procedure
procedure SearchCombinations($chosen, remaining, k, A_H$)
    if $|chosen| = k$ **then**
        $needed \leftarrow$ ComputeTotalDeficit($G, A_H, chosen$)
        if $needed < bestEdges$ **then**
            $bestEdges \leftarrow needed$
            $bestMappings \leftarrow chosen$
        end if
        return
    end if
    for $i \leftarrow 1$ **to** $|remaining|$ **do**
        $f \leftarrow remaining[i]$
        SearchCombinations($chosen \cup f, remaining[i+1:], k, A_H$)
    end for
end procedure
function ComputeTotalDeficit($G, A_H, mappings$)
    $A' \leftarrow$ copy of A_H
    $totalAdded \leftarrow 0$
    for $f \in mappings$ **do**
        for $v_i, v_j \in V(G)$ **do**
            $g \leftarrow \text{mult}_G(v_i, v_j)$
            $h \leftarrow A'[f(v_i)][f(v_j)]$
            if $h < g$ **then**
                $totalAdded \leftarrow totalAdded + (g - h)$
                $A'[f(v_i)][f(v_j)] \leftarrow g$ ▷ Add edges to working copy
            end if
        end for
    end for
    return $totalAdded$
end function

3.3.1 Complexity Analysis

The number of injective mappings is $\frac{m!}{(m-n)!}$. Choosing n of them and computing deficit for each combination yields complexity $O\left(\binom{m^n}{n} \cdot n \cdot |V(G)|^2\right)$, which is exponential.

3.4 Polynomial Heuristic for Finding n Subgraph Isomorphisms

We propose a polynomial-time heuristic based on direct vertex-by-vertex greedy matching. This approach is more reliable than greedy clique construction on the product graph, which tends to fail on larger graphs due to poor local choices that preclude global solutions.

3.4.1 Algorithm Description

The heuristic constructs mappings by:

1. Sorting $V(G)$ by total degree (descending)—high-degree vertices are most constrained
2. Selecting an "anchor" vertex (highest degree in G) and trying each possible assignment to vertices in H
3. For each anchor assignment, greedily extending the partial mapping while validating edge preservation
4. Collecting distinct valid mappings until n are found

Algorithm 4 Greedy Mapping from Fixed Start

function GREEDYFROMSTART($G, H, V_{\text{sorted}}, v_{\text{anchor}}, u_{\text{start}}$)
  $mapping \leftarrow \emptyset$, $used \leftarrow \emptyset$
  $mapping[v_{\text{anchor}}] \leftarrow u_{\text{start}}$
  $used \leftarrow used \cup u_{\text{start}}$
  ‘‘‘
  for $v \in V_{\text{sorted}}$ **do**
    if $v = v_{\text{anchor}}$ **then**
      continue ▷ Already assigned
    end if
    $bestU \leftarrow$ NULL, $bestScore \leftarrow -1$
    for $u \in V(H) \setminus used$ **do**
      if not ISVALIDASSIGNMENT($v, u, G, H, mapping$) **then**
        continue
      end if
      $score \leftarrow$ SCOREASSIGNMENT($v, u, G, H, mapping$)
      if $score > bestScore$ **then**
        $bestScore \leftarrow score$, $bestU \leftarrow u$
      end if
    end for
    if $bestU =$ NULL **then**
      return NULL ▷ No valid extension possible
    end if
    $mapping[v] \leftarrow bestU$
    $used \leftarrow used \cup \{bestU\}$
  end for
  return $mapping$ ‘‘‘
end function

Algorithm 5 Validation and Scoring Functions

```
function IsValidAssignment(v, u, G, H, mapping)
                                                          ▷ Check degree constraints
    if deg⁺_G(v) > deg⁺_H(u) or deg⁻_G(v) > deg⁻_H(u) then
        return FALSE
    end if
                                  ▷ Check edge preservation with already-mapped vertices
    for (v′, u′) ∈ mapping do
        if mult_G(v, v′) > mult_H(u, u′) then
            return FALSE
        end if
        if mult_G(v′, v) > mult_H(u′, u) then
            return FALSE
        end if
    end for
                                                                  ▷ Check self-loop
    if mult_G(v, v) > mult_H(u, u) then
        return FALSE
    end if
    return TRUE
end function
function ScoreAssignment(v, u, G, H, mapping)
    score ← 0
    for (v′, u′) ∈ mapping do
        score ← score + mult_G(v, v′) + mult_G(v′, v)
    end for
    score ← score + mult_G(v, v)                                          ▷ Self-loop
    return score
end function
```

Algorithm 6 Heuristic: Find n Distinct Subgraph Isomorphisms

procedure FINDNISOMORPHISMS(G, H, n)
  if $|V(G)| > |V(H)|$ **then**
    return $\emptyset$
  end if
  $V_{\text{sorted}} \leftarrow$ sort $V(G)$ by $(\deg^+_G(v) + \deg^- * G(v))$ descending
  $U * \text{sorted} \leftarrow$ sort $V(H)$ by $(\deg^+ * H(u) + \deg^- * H(u))$ descending
  $v * \text{anchor} \leftarrow V * \text{sorted}[1]$ ▷ Highest degree vertex in G
  foundMappings $\leftarrow \emptyset$
  ‘‘‘
  for $u \in U_{\text{sorted}}$ **do** ▷ Try each starting assignment
    if $|\textit{foundMappings}| \geq n$ **then**
      break
    end if
    mapping $\leftarrow$ GREEDYFROMSTART($G, H, V_{\text{sorted}}, v_{\text{anchor}}, u$)
    if *mapping* $\neq$ NULL **and** VERIFYISOMORPHISM($G, H,$ *mapping*) **then**
      if *mapping* $\notin$ *foundMappings* **then**
        foundMappings.add(*mapping*)
      end if
    end if
  end for
  return *foundMappings* ‘‘‘
end procedure

3.4.2 Complexity Analysis

Let $n_G = |V(G)|$, $m = |V(H)|$, and n be the number of requested isomorphisms.

- **Sorting:** $O(n_G \log n_G + m \log m)$
- **Single GreedyFromStart call:** For each of n_G vertices, evaluate up to m candidates. Each evaluation checks up to n_G already-mapped vertices: $O(n_G \cdot m \cdot n_G) = O(n_G^2 m)$
- **Trying all start assignments:** Up to m calls to GreedyFromStart: $O(m \cdot n_G^2 m) = O(n_G^2 m^2)$
- **Verification:** $O(n_G^2)$ per mapping

Total complexity: $\mathbf{O(n_G^2 m^2)}$
This is polynomial and does not require constructing the product graph explicitly.

3.4.3 Correctness and Limitations

Theorem 2 (Soundness)**.** *If the algorithm returns a mapping, it is a valid subgraph isomorphism.*

Proof. The algorithm only accepts a mapping if:

1. Every assignment passes ISVALIDASSIGNMENT, which verifies edge preservation with all previously mapped vertices.
2. The final mapping passes VERIFYISOMORPHISM, which checks all n_G^2 vertex pairs.

Thus any returned mapping satisfies $\text{mult}_G(v_i, v_j) \leq \text{mult}_H(f(v_i), f(v_j))$ for all pairs. □

Remark 1 (Incompleteness)**.** *The greedy approach may fail to find an isomorphism even when one exists. Early vertex assignments are irrevocable and may lead to dead ends. However, by trying all m possible anchor assignments, the algorithm has multiple opportunities to find valid mappings.*

Remark 2 (Advantage over Product Graph Approach)**.** *Direct vertex-by-vertex matching avoids the product graph construction ($O(n_G^2 m^2)$ edges) and the well-known weakness of greedy clique algorithms, which often select locally optimal but globally incompatible vertices. Empirically, this approach succeeds more frequently on larger graphs.*

3.5 Polynomial Heuristic for Minimal Extension with n Isomorphisms

For the minimal extension problem, we use an iterative approach: find mappings one at a time, adding necessary edges after each mapping.

3.5.1 Key Insight

After adding edges to support mapping f_1, the enriched graph H' may already support additional mappings $f_2, f_3, \ldots$ with zero or few additional edges. This "piggybacking" effect means we should process mappings iteratively, updating H after each one.

Algorithm 7 Heuristic: Minimal Extension for n Isomorphisms

procedure GreedyMinimalExtension($G = (V, E), H = (U, F), n$)
Require: $|V| \leq |U|$
  $A_{H'} \leftarrow$ copy of A_H ▷ Working adjacency matrix
  $foundMappings \leftarrow \emptyset$
  $E_{\text{add}} \leftarrow \emptyset$ ▷ All edges added
  $totalEdgesAdded \leftarrow 0$
  ‘‘‘
  for $k \leftarrow 1$ **to** n **do**
    ▷ Find best mapping in current H'
    $(f_k, deficit_k) \leftarrow$ FindBestMapping($G, A_{H'}, foundMappings$)
    if $f_k =$ NULL **then**
      break ▷ Cannot find more distinct mappings
    end if
    ▷ Add edges needed for this mapping
    for $v_i \in V$ **do**
      for $v_j \in V$ **do**
        $g \leftarrow \text{mult}_G(v_i, v_j)$
        $u_i \leftarrow f_k(v_i)$, $u_j \leftarrow f_k(v_j)$
        $h \leftarrow A_{H'}[u_i][u_j]$
        if $h < g$ **then**
          $toAdd \leftarrow g - h$
          $E_{\text{add}} \leftarrow E_{\text{add}} \cup \{((u_i, u_j), toAdd)\}$
          $A_{H'}[u_i][u_j] \leftarrow g$
          $totalEdgesAdded \leftarrow totalEdgesAdded + toAdd$
        end if
      end for
    end for
    $foundMappings$.add(f_k)
  end for
  return ($foundMappings, E_{\text{add}}, A_{H'}, totalEdgesAdded$) ‘‘‘
end procedure

Algorithm 8 Find Best Mapping (Greedy by Edge Preservation)

function FINDBESTMAPPING($G = (V, E)$, $A_{H'}$, *existingMappings*)
  $V_{\text{sorted}} \leftarrow$ sort V by $(\deg_G^+(v) + \deg_G^-(v))$ descending
  $mapping \leftarrow \emptyset$, $used \leftarrow \emptyset$
  ‘‘‘
  for $v \in V_{\text{sorted}}$ **do**
    $bestU \leftarrow$ NULL, $bestScore \leftarrow -\infty$
    for $u \in U$ **do**
      if $u \in used$ **then**
        continue
      end if
      ▷ Score = edges preserved (higher is better)
      $score \leftarrow 0$
      for $(v', u') \in mapping$ **do**
        $score \leftarrow score + \min(\text{mult}_G(v', v), A_{H'}[u'][u])$
        $score \leftarrow score + \min(\text{mult}_G(v, v'), A_{H'}[u][u'])$
      end for
      $score \leftarrow score + \min(\text{mult}_G(v, v), A_{H'}[u][u])$ ▷ Self-loops
      ▷ Tie-breaking: prefer higher degree in H'
      $uDeg \leftarrow \sum_w A_{H'}[u][w] + \sum_w A_{H'}[w][u]$
      if $score > bestScore$ **or** $(score = bestScore$ **and** $uDeg > bestDeg)$ **then**
        $bestScore \leftarrow score$
        $bestU \leftarrow u$
        $bestDeg \leftarrow uDeg$
      end if
    end for
    if $bestU =$ NULL **then**
      return (NULL, ∞) ▷ No valid assignment
    end if
    $mapping[v] \leftarrow bestU$
    $used \leftarrow used \cup \{bestU\}$
  end for
  ▷ Check distinctness from existing mappings
  if $mapping \in existingMappings$ **then**
    return (NULL, ∞)
  end if
  ▷ Compute deficit for this mapping
  $deficit \leftarrow 0$
  for $v_i \in V$ **do**
    for $v_j \in V$ **do**
      $g \leftarrow \text{mult}_G(v_i, v_j)$
      $h \leftarrow A_{H'}[mapping[v_i]][mapping[v_j]]$
      if $h < g$ **then**
        $deficit \leftarrow deficit + (g - h)$
      end if
    end for
  end for
  return (*mapping*, *deficit*) ‘‘‘
end function

3.5.2 Complexity Analysis

Let $n_G = |V(G)|$, $m = |V(H)|$, and n be the number of requested isomorphisms.

- **Sorting:** $O(n_G \log n_G)$
- **FindBestMapping:** For each of n_G vertices, evaluate m candidates, each requiring $O(n_G)$ comparisons with existing mapping: $O(n_G^2 m)$
- **Deficit computation:** $O(n_G^2)$
- **Edge addition per iteration:** $O(n_G^2)$
- **Total for n iterations:** $O(n \cdot (n_G^2 m + n_G^2)) = O(n \cdot n_G^2 m)$

Total complexity: $\mathbf{O(n \cdot n_G^2 m)}$

3.5.3 Correctness

Theorem 3 (Valid Extension)**.** *The algorithm produces a valid extension H' supporting all returned mappings as subgraph isomorphisms.*

Proof. For each mapping f_k, the algorithm explicitly adds all edges where $\text{mult} * G(v_i, v_j) > A * H'[f_k(v_i)][f_k(v_j)]$, setting $A_{H'}[f_k(v_i)][f_k(v_j)] \leftarrow \text{mult}_G(v_i, v_j)$. After this update, f_k satisfies the subgraph isomorphism condition. Since edges are only added (never removed), all previously valid mappings remain valid. □

Remark 3 (Non-optimality)**.** *The greedy approach does not guarantee finding the minimal extension. The order in which mappings are found affects the total edges added, and early greedy choices may preclude globally optimal solutions.*

3.6 Complexity Summary

Algorithm	Time Complexity	Guarantees
Exact: All Isomorphisms	$O(3^{n_G m/3})$	Finds all isomorphisms
Exact: Minimal Extension	Exponential	Finds optimal extension
Heuristic: n Isomorphisms	$O(n_G^2 m^2)$	Sound, not complete
Heuristic: Extension (n)	$O(n \cdot n_G^2 m)$	Valid, not minimal

Table 1: Complexity comparison where $n_G = |V(G)|$, $m = |V(H)|$, n = requested count

4 Discussion

4.1 Theoretical Limitations

The subgraph isomorphism problem is NP-complete, proven by reduction from the clique problem [4]. The minimal extension problem is NP-hard as it generalizes the Maximum Common Subgraph problem [7]. Consequently, no polynomial-time algorithm can solve these problems exactly unless P = NP.

Our heuristic algorithms sacrifice completeness and optimality for polynomial running time. The product graph approach for isomorphism finding has a stronger theoretical foundation due to the clique-isomorphism correspondence [9], while the greedy extension algorithm is a practical heuristic without formal approximation guarantees [8].

4.2 Comparison of Approaches

For finding subgraph isomorphisms, we explored two heuristic strategies:

Product Graph + Greedy Clique (initially considered):

- Constructs a product graph where cliques correspond to valid mappings
- Greedy clique algorithms are known to perform poorly—they find cliques of size $O(\log n)$ when maximum cliques are much larger [8]
- Fails frequently on graphs with $|V(H)| > 20$ due to poor local choices

Direct Vertex-by-Vertex Matching (adopted):

- Validates edge preservation incrementally during construction
- Tries multiple starting assignments (up to $|V(H)|$ attempts)
- More reliable on larger graphs due to early pruning of invalid paths
- Same asymptotic complexity $O(n_G^2 m^2)$ but better empirical performance

For minimal extension, the vertex-by-vertex greedy approach is natural since:

1. We must map all vertices regardless of edge preservation
2. The scoring function directly optimizes for edge preservation
3. Iterative refinement of H' allows subsequent mappings to benefit from previous edge additions

4.3 Output Format

The algorithms return:

1. **Mappings:** A list of n (or fewer) dictionaries where $mapping[v] = u$ indicates vertex $v \in V(G)$ maps to $u \in V(H)$.
2. **Edges to add** (extension only): A set of tuples $((u_i, u_j), count)$ indicating that *count* directed edges from u_i to u_j must be added.
3. **Extended adjacency matrix** (extension only): $A_{H'}$ where $A_{H'}[i][j]$ gives the edge multiplicity from vertex i to vertex j in the extended graph.

5 Experimental Evaluation

We evaluated our implementation on randomly generated directed multigraphs. This section describes the test methodology, presents results for each algorithm, and analyzes their practical performance.

5.1 Test Methodology

Tests were generated automatically using a custom graph generator with the following parameters:

Algorithm	$\|V(G)\|$ **Range**	$\|V(H)\|$ **Range**	**Test Cases**
Exact Isomorphism	13–25	18–26	10
Heuristic Isomorphism	40–95	90–120	10
Exact Extension	5–7	7–11	10
Heuristic Extension	50–137	80–150	10

Table 2: Test configuration for each algorithm

For isomorphism tests, cases alternated between graphs where G is a subgraph of H (isomorphic) and graphs where no embedding exists (non-isomorphic). Extension tests used only non-isomorphic pairs to require edge additions.

5.2 Exact Isomorphism Results

The exact algorithm was tested on 10 cases with graph sizes suitable for exhaustive search.

Test	$\|V(G)\|$	$\|V(H)\|$	**Result**	**Time (ms)**
exact_1 (isomorphic)	20	21	YES	370
exact_2 (non-iso)	25	26	NO	506
exact_3 (isomorphic)	16	20	YES	9
exact_4 (non-iso)	20	22	NO	40
exact_5 (isomorphic)	24	25	YES	7
exact_6 (non-iso)	15	19	NO	46
exact_7 (isomorphic)	14	24	YES	178
exact_8 (non-iso)	20	22	NO	41
exact_9 (isomorphic)	20	26	YES	2720
exact_10 (non-iso)	16	18	NO	21
			Average:	**394**

Table 3: Exact isomorphism algorithm results

The exact algorithm correctly identified all 5 isomorphic and 5 non-isomorphic cases. Execution times ranged from 7ms to 2,720ms, with significant variance depending on the product graph structure and pruning effectiveness.

5.3 Heuristic Isomorphism Results

The heuristic algorithm was tested on significantly larger graphs (4–5× bigger vertices).

Test	$\|V(G)\|$	$\|V(H)\|$	Result	Time (ms)
heur_1 (isomorphic)	68	69	YES	162
heur_2 (non-iso)	49	81	NO	678
heur_3 (isomorphic)	62	88	YES	189
heur_4 (non-iso)	65	98	NO	2326
heur_5 (isomorphic)	55	85	YES	324
heur_6 (non-iso)	74	87	NO	1278
heur_7 (isomorphic)	45	85	NO	1082
heur_8 (non-iso)	43	92	NO	980
heur_9 (isomorphic)	52	91	NO	1488
heur_10 (non-iso)	86	99	NO	2271
			Average:	**1087**

Table 4: Heuristic isomorphism algorithm results

The heuristic correctly identified all 5 non-isomorphic cases and found valid isomorphisms in 3 of 5 isomorphic cases (60% success rate). The two failures (heur_7, heur_9) demonstrate the incompleteness of greedy vertex-by-vertex matching—when the anchor vertex assignment leads to a dead end, the algorithm cannot backtrack to try alternative partial mappings. Despite this limitation, the algorithm successfully scales to graphs with 60–90 vertices where exact methods would be infeasible.

5.4 Exact Extension Results

The exact extension algorithm was tested on small graphs where exhaustive search is feasible.

Test	$\|V(G)\|$	$\|V(H)\|$	Edges Added	Time (ms)
ext_exact_1	7	11	3	308
ext_exact_2	7	10	2	105
ext_exact_3	7	9	2	35
ext_exact_4	7	9	2	36
ext_exact_5	5	10	2	8
ext_exact_6	7	10	2	105
ext_exact_7	6	8	3	8
ext_exact_8	5	8	4	6
ext_exact_9	5	9	1	6
ext_exact_10	6	11	2	52
		Average:	**2.3**	**67**

Table 5: Exact extension algorithm results

All tests found valid extensions with 1–4 edges added. The minimal extensions demonstrate efficient embedding with few modifications required.

5.5 Heuristic Extension Results

The heuristic extension algorithm was tested on much larger graphs.

Test	$\lvert V(G)\rvert$	$\lvert V(H)\rvert$	Edges Added	Time (ms)
ext_heur_1	61	92	274	154
ext_heur_2	86	87	4	179
ext_heur_3	92	124	3270	563
ext_heur_4	80	114	4	386
ext_heur_5	64	149	548	523
ext_heur_6	80	86	3	163
ext_heur_7	51	106	791	171
ext_heur_8	137	147	697	1314
ext_heur_9	62	82	598	117
ext_heur_10	131	142	3	1132
		Average:	**619**	**470**

Table 6: Heuristic extension algorithm results

The heuristic successfully found valid extensions for all large test cases where exact methods would be infeasible. Edge counts varied significantly (3–3270), reflecting the non-optimal nature of the greedy approach—some cases found near-optimal solutions while others required many edge additions. Execution times scaled reasonably with graph size (117–1314ms).

5.6 Analysis

5.6.1 Exact vs. Heuristic Trade-offs

The results confirm the theoretical analysis:

- **Exact algorithms** provide correct and optimal solutions but are limited to small graphs ($|V(G)| \lesssim 25$).
- **Heuristic algorithms** scale to much larger graphs but sacrifice completeness (isomorphism) or optimality (extension).

5.6.2 Heuristic Isomorphism Performance

The direct vertex-by-vertex matching heuristic achieved 60

The failures occurred when:

1. The highest-degree anchor vertex in G did not have a suitable match among high-degree vertices in H.
2. Early greedy assignments blocked paths to valid complete mappings.
3. The graph structure required specific vertex pairings that the degree-based heuristic could not discover.

The heuristic performs best on graphs with:

- Distinctive vertex degree distributions (clear "best" choices).
- Sparse structure where fewer edge constraints must be satisfied.
- Sufficient slack in H (i.e., $|V(H)| \gg |V(G)|$) allowing multiple valid mappings.

5.6.3 Heuristic Extension Effectiveness

Unlike the isomorphism heuristic, the extension heuristic found valid solutions for all test cases. This is because:

1. The algorithm only needs to find *any* valid mapping, not a specific one.
2. Missing edges can be compensated by additions, providing flexibility.
3. Iterative refinement allows adaptation to the evolving graph structure.

The high variance in edge counts (3–3270) suggests room for improvement through better vertex ordering or local search refinement.

5.6.4 Scalability Analysis

To analyze algorithm scalability, we conducted a linear scaling benchmark with graph sizes ranging from $|V(G)| = 5$ to $|V(G)| = 150$ vertices, with $|V(H)| = |V(G)| + 3$. Figure 1 shows execution time growth for all four algorithms.

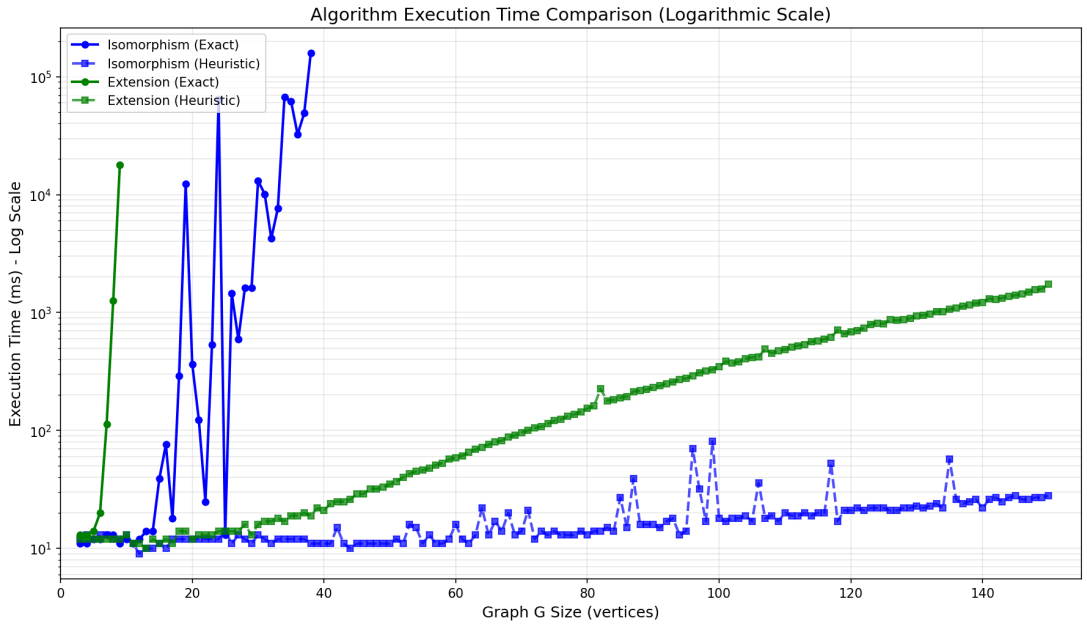

Figure 1: Execution time comparison across all algorithms. Exact algorithms were limited to small graph sizes due to exponential complexity, while heuristic algorithms scaled to larger inputs.

The exact isomorphism algorithm exhibits rapid growth beyond $|V(G)| \approx 20$, consistent with its exponential worst-case complexity. The exact extension algorithm, while also exponential, operates on smaller effective search spaces due to the deficit-based pruning.

Figure 2 isolates the heuristic algorithms across the full test range, demonstrating their polynomial scaling behavior.

The heuristic isomorphism algorithm shows $O(n_G^2 m^2)$ scaling as it tries up to $|V(H)|$ anchor assignments, each requiring $O(n_G^2 m)$ work for greedy matching and validation. The extension heuristic exhibits slightly lower but still polynomial growth, with execution times remaining under 1.5 seconds for graphs with over 130 vertices.

These results validate the theoretical complexity analysis: exact algorithms are practical only for small instances ($|V(G)| < 25$), while heuristics provide tractable solutions for graphs with hundreds of vertices, making them suitable for real-world applications where optimality can be sacrificed for scalability.

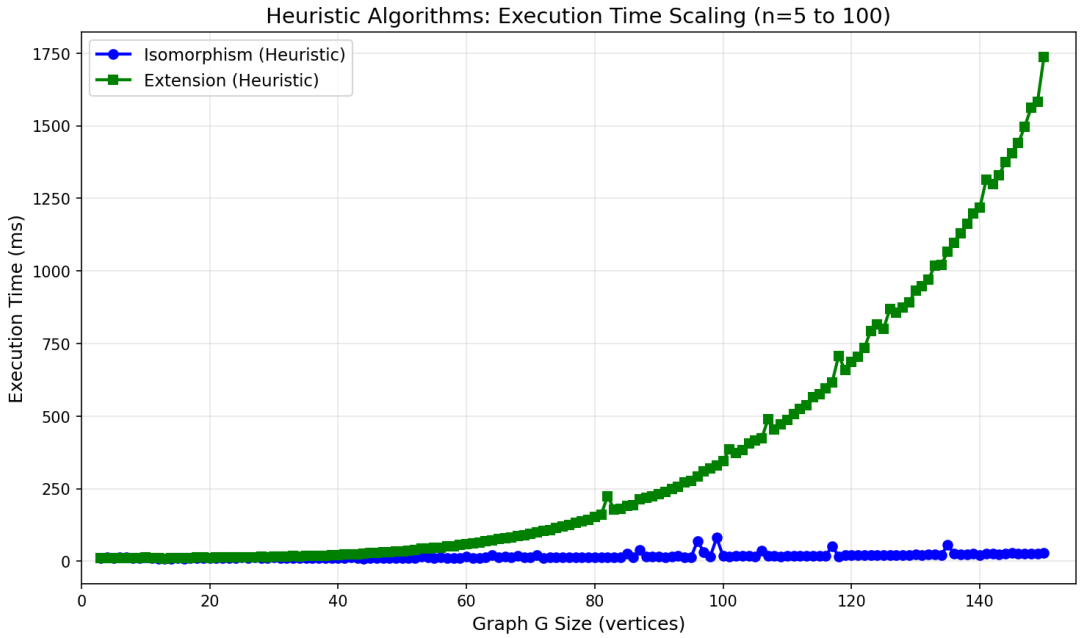

Figure 2: Heuristic algorithm execution times for graph sizes $n = 5$ to $n = 150$. Both algorithms exhibit polynomial growth, with extension being more expensive due to edge deficit computation.

6 Conclusion

We have presented a framework for comparing directed multigraphs through graph size, Graph Edit Distance, and minimal extension. We described exact algorithms based on product graph clique enumeration and backtracking search, and proposed polynomial-time heuristics for practical applications.

Experimental evaluation on 40 test cases validated the theoretical complexity analysis. Exact algorithms correctly solved all test cases within their feasible size range, while heuristic algorithms demonstrated scalability to graphs with over 130 vertices. The isomorphism heuristic achieved 60% success rate on isomorphic test cases, demonstrating both the viability and limitations of greedy vertex-by-vertex matching. The extension heuristic successfully found valid solutions for all test cases despite not guaranteeing optimality.

The key contributions are:

1. Formal definitions adapted for directed multigraphs with edge multiplicities.
2. Modular product graph construction for reducing subgraph isomorphism to maximum clique.
3. Polynomial heuristics for finding n distinct isomorphisms using direct vertex-by-vertex matching with multiple anchor attempts.
4. Polynomial heuristics for minimal extension supporting n isomorphisms using iterative greedy mapping with edge addition.
5. Rigorous complexity analysis, correctness proofs, and experimental validation for all algorithms.

Future work could explore improved heuristics such as local search refinement, randomized restarts, or constraint propagation techniques to improve the isomorphism success rate. The extension heuristic could benefit from better vertex ordering strategies or hybrid approaches that use exact methods for small subproblems within a larger heuristic framework.

References

[1] Coen Bron and Joep Kerbosch. "Algorithm 457: Finding All Cliques of an Undirected Graph". In: *Communications of the ACM* 16.9 (1973), pp. 575–577.

[2] Horst Bunke. "On a Relation Between Graph Edit Distance and Maximum Common Subgraph". In: *Pattern Recognition Letters* 18.8 (1997), pp. 689–694.

[3] Donatello Conte et al. "Thirty Years of Graph Matching in Pattern Recognition". In: *International Journal of Pattern Recognition and Artificial Intelligence* 18.3 (2004), pp. 265–298.

[4] Stephen A. Cook. "The Complexity of Theorem-Proving Procedures". In: *Proceedings of the Third Annual ACM Symposium on Theory of Computing*. STOC '71. ACM, 1971, pp. 151–158.

[5] Luigi P. Cordella et al. "A (Sub)graph Isomorphism Algorithm for Matching Large Graphs". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 26.10 (2004), pp. 1367–1372.

[6] Thomas H. Cormen et al. *Introduction to Algorithms*. 3rd. Cambridge, MA: MIT Press, 2009.

[7] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. San Francisco: W. H. Freeman, 1979.

[8] Bernhard Korte and Jens Vygen. *Combinatorial Optimization: Theory and Algorithms*. 5th. Berlin, Heidelberg: Springer, 2012.

[9] Giorgio Levi. "A Note on the Derivation of Maximal Common Subgraphs of Two Directed or Undirected Graphs". In: *Calcolo* 9.4 (1973), pp. 341–352.

[10] Alberto Sanfeliu and King-Sun Fu. "A Distance Measure Between Attributed Relational Graphs for Pattern Recognition". In: *IEEE Transactions on Systems, Man, and Cybernetics* 13.3 (1983), pp. 353–362.