Approximation Algorithm for Minimum Extension Problem

Esbol Erlan, Mateusz Cieśla, Ihor Kulynych, Ibrahim Abualshayeb

December 19, 2025

1 Problem Definition

Given two directed graphs G and H represented by adjacency matrices, the goal of the *Minimum Extension Problem* is to extend H by adding the minimum number of directed edges such that G becomes a subgraph of H under some injective mapping $f : V(G) \to V(H)$.

Additionally, we extend this to the *Multiple Copies Problem*: find the minimum number of edges to add to H such that it contains exactly k non-overlapping copies of G as subgraphs.

2 Definitions

Subgraph and Subgraph Isomorphism

[Subgraph] Let $G = (V_G, E_G)$ and $H = (V_H, E_H)$ be two directed graphs. If $V_G \subseteq V_H$ and $E_G \subseteq E_H$, then the directed graph G is called a subgraph of H [7, 6].

[Subgraph Isomorphism] Let $G = (V_G, E_G)$ and $H = (V_H, E_H)$ be directed graphs. G is isomorphic to a subgraph of H if there exists an injective mapping

$$f : V_G \to V_H \text{ such that for every edge } (u, v) \in E_G \Rightarrow (f(u), f(v)) \in E_H$$

[4, 7].

[Graph Size] Let $G = (V_G, E_G)$ be a graph.

$$|G| := |E_G|.$$

The size of a graph is the total number of edges. This is motivated by the fact that, in this problem, the vertex sets of G and H are already aligned. Once the vertices are fixed, making G a subgraph of H only requires adding missing edges to H. Therefore, the number of edges is the natural measure of how much the graph must be modified [1].

[Adjacency Matrix] Let $G = (V_G, E_G)$ be a directed graph with $|V_G| = n$. An order of the vertices is defined as

$$V_G = \{v_0, v_1, \ldots, v_{n-1}\},$$

where vertices correspond to the columns and rows in A_G. The adjacency matrix of G is the $n \times n$ matrix A_G defined by

$$A_G[i][j] = \begin{cases} 1, & \text{if } (v_i, v_j) \in E_G, \\ 0, & \text{otherwise.} \end{cases}$$

[7, 6].

[Graph Mapping] Let $A = (V_A, E_A)$ and $B = (V_B, E_B)$ be two graphs. A graph mapping

$$M_A(B) = \{M_A(b_i) \mid b_i \in V_B, M_A(b_i) \in V_A\}$$

assigns each vertex $b_i \in V_B$ to a vertex $a_j \in V_A$, meaning that b_i represents a_j. This mapping induces an edge mapping:

$$E(M_A(B)) = \{(M_A(b_i), M_A(b_j)) \mid (b_i, b_j) \in E_B\}[7].$$

[Graph Distance Given a Mapping] Let $G = (V_G, E_G)$ and $H = (V_H, E_H)$ be directed graphs with $|V_G| \leq |V_H|$. Let

$$f : V_G \to V_H$$

be an injective mapping. The distance between G and H under f is defined as

$$D(G, H, f) = |\{(u, v) \in E_G \mid (f(u), f(v)) \notin E_H\}|,$$

counting how many edges of G are missing in H [1].

[Best Graph Mapping] Let $\mathcal{M}_A(B)$ denote the set of all possible mappings from B to A. The best graph mapping $M_{bA}(B)$ for A is the mapping that minimizes the distance:

$$M_{bA}(B) = \arg \min_{M_i \in \mathcal{M}_A(B)} |A - M_i(B)|[1,\ 5].$$

[Graph Distance (Canonical Form)] The graph distance between A and B is the distance under the best mapping:

$$|A - B| := |A - M_{bA}(B)|[1].$$

[Graph Subset via Mapping] We say $B \subseteq A$ if and only if there exists a mapping $M_A(B)$ such that

$$M_A(B) \subseteq V_A \text{ and } M_A(E_B) \subseteq E_A,$$

or equivalently

$$B \subseteq A \iff |A - M_{bA}(B)| = 0[7,\ 6].$$

[Graph Extension] If $B \not\subseteq A$, we seek an extension of A, denoted $Ext(A)$, such that

$$M_{bA}(B) \subseteq Ext(A).$$

The minimal extension satisfying this is:

$$Ext(A) = A \cup M_{bA}(B)[1].$$

3 Methods

3.1 Graph Size Calculation

To calculate the size of a graph $G = (V_G, E_G)$:

Return the number of edges in the graph: $|E_G|$

Example: Let $G = (V_G, E_G)$ where $V_G = \{1, 2, 3\}$ and $E_G = \{(1, 2), (2, 3)\}$.

$|G| = |E_G| = 2$

3.2 Distance Between Graph and Mapping

Given a mapping $M_A(B)$, to calculate $|A - M_A(B)|$:

1. Apply mapping M_A to all vertices of B: $M_A(V_B)$
2. Apply mapping M_A to all edges of B: $M_A(E_B)$
3. Calculate symmetric difference of edge sets and return it: $|E_A \triangle M_A(E_B)|$

Example: Let $A = (V_A, E_A)$ where $V_A = \{1, 2, 3\}$, $E_A = \{(1, 2), (2, 3)\}$ and $B = (V_B, E_B)$ where $V_B = \{a, b, c\}$, $E_B = \{(a, b), (b, c)\}$.

Given mapping $M_A : ao1, bo2, coc$:

- $M_A(V_B) = \{1, 2, c\}$
- $M_A(E_B) = \{(1, 2), (2, c)\}$
- $E_A \triangle M_A(E_B) = \{(1, 2), (2, 3)\} \triangle \{(1, 2), (2, c)\} = \{(2, 3), (2, c)\}$
- $|A - M_A(B)| = |E_A \triangle M_A(E_B)| = 2$

3.2.1 Asymmetric Difference Calculation

The asymmetric difference $E(A) \triangle E(B)$ of two sets contains edges that are in exactly one of the sets (not in both).

Steps to calculate asymmetric difference:

1. Find edges in A but not in B: $E(A) \setminus E(B)$
2. Find edges in B but not in A: $E(B) \setminus E(A)$
3. Union the two results: $E(A) \triangle E(B) = (E(A) \setminus E(B)) \cup (E(B) \setminus E(A))$
4. Result = $|E(A) \triangle E(B)|$

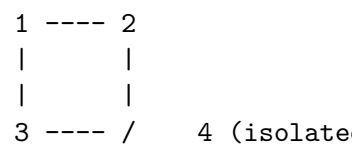
3.3 Distance Between Graphs

To find the graph distance $|A - B|$:

1. Generate all possible mappings from B to A
2. For each mapping M_A, calculate $|A - M_A(B)|$
3. Find the mapping that gives the minimum distance
4. Return this minimum distance as $|A - B|$

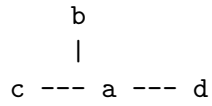
Example: Triangle vs Star Graph
Consider two structurally different graphs with the same number of vertices and edges:

Graph A (Triangle + Isolated Vertex): $V_A = \{1, 2, 3, 4\}$, $E_A = \{(1, 2), (2, 3), (1, 3)\}$



- Structure: Triangle on vertices $\{1, 2, 3\}$ with isolated vertex 4
- 4 vertices, 3 edges

Graph B (Star Graph): $V_B = \{a, b, c, d\}$, $E_B = \{(a, b), (a, c), (a, d)\}$



- Structure: Central vertex a connected to three leaves $\{b, c, d\}$
- 4 vertices, 3 edges

The best mapping yields distance 2, so $|A - B| = 2$.

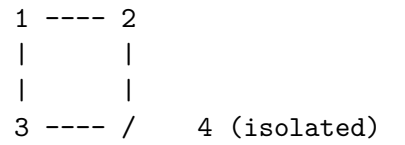
3.4 Best Mapping

To find the best mapping $M_{bA}(B)$:

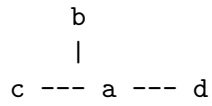
1. Enumerate all possible one-to-one mappings from B to A
2. For each mapping, calculate the distance $|A - M_A(B)|$
3. Select the mapping with minimum distance
4. Return this optimal mapping

Example: Finding Best Mapping for Triangle vs Star
Using the same graphs from the previous example:

Graph A (Triangle + Isolated Vertex): $V_A = \{1, 2, 3, 4\}$, $E_A = \{(1, 2), (2, 3), (1, 3)\}$



Graph B (Star Graph): $V_B = \{a, b, c, d\}$, $E_B = \{(a, b), (a, c), (a, d)\}$



Possible mappings to consider:

- $M_1 : ao1, bo2, co3, do4$: $|A - M_1(B)| = 2$ (good: maps star center to triangle vertex)
- $M_2 : ao4, bo1, co2, do3$: $|A - M_2(B)| = 6$ (bad: maps star center to isolated vertex)
- $M_3 : ao2, bo1, co3, do4$: $|A - M_3(B)| = 2$ (also good: another triangle vertex)

Best mapping is M_1 or M_3 with distance 2.

3.5 Graph Subset Check

To check if $B \subseteq A$:

1. Find the best mapping $M_{bA}(B)$ from B to A
2. Check if $M_{bA}(V_B) \subseteq V_A$ and $M_{bA}(E_B) \subseteq E_A$
3. If both conditions hold, then $B \subseteq A$
4. Equivalently, check if $|A - M_{bA}(B)| = 0$

Example: Let $A = (V_A, E_A)$ where $V_A = \{1, 2, 3\}$, $E_A = \{(1, 2), (2, 3)\}$ and $B = (V_B, E_B)$ where $V_B = \{a, b\}$, $E_B = \{(a, b)\}$.

Best mapping $M_{bA} : ao1, bo2$:

- $M_{bA}(V_B) = \{1, 2\} \subseteq V_A$ ✓
- $M_{bA}(E_B) = \{(1, 2)\} \subseteq E_A$ ✓
- $|A - M_{bA}(B)| = 0$ ✓

Therefore, $B \subseteq A$.

3.6 Graph Extension

To compute $Ext(A)$ when $B \not\subseteq A$:

1. Find the best mapping $M_{bA}(B)$ from B to A
2. Create new vertex set: $V_{Ext(A)} = V_A \cup M_{bA}(V_B)$
3. Create new edge set: $E_{Ext(A)} = E_A \cup M_{bA}(E_B)$
4. Return the extended graph $Ext(A) = (V_{Ext(A)}, E_{Ext(A)})$

Example: Let $A = (V_A, E_A)$ where $V_A = \{1, 2\}$, $E_A = \{(1, 2)\}$ and $B = (V_B, E_B)$ where $V_B = \{a, b, c\}$, $E_B = \{(a, b), (b, c)\}$.

Since $B \not\subseteq A$ (B has 3 vertices, A has 2), we need to extend A:

- Best mapping $M_{bA} : ao1, bo2, co3$ (add vertex 3)
- $M_{bA}(V_B) = \{1, 2, 3\}$
- $M_{bA}(E_B) = \{(1, 2), (2, 3)\}$
- $V = \{1, 2\} \cup \{1, 2, 3\} = \{1, 2, 3\}$
- $E_{Ext(A)} = \{(1, 2)\} \cup \{(1, 2), (2, 3)\} = \{(1, 2), (2, 3)\}$

Therefore: $Ext(A) = (\{1, 2, 3\}, \{(1, 2), (2, 3)\})$

Clarified description of `ExactMinExtendGrap`

The purpose of `ExactMinExtendGrap` is to decide how well a smaller graph G can be placed inside a larger graph H, and—if it is not already inside—what smallest set of edge additions to H would make it contain G. Before we discuss results, we must first define what a *mapping* is, since all results depend on it.

Definition: mapping

A *mapping* is a rule that assigns every vertex of G to a distinct vertex of H. Formally,

$$f : V(G) \to V(H)$$

is a mapping used here only if no two different vertices of G are assigned to the same vertex of H. Intuitively, a mapping tells us "which vertex of H represents each vertex of G."

What the algorithm searches

The algorithm systematically examines all possible mappings from the vertex set of G into the vertex set of H. If H has fewer vertices than G, the algorithm first enlarges H by adding enough isolated (unconnected) vertices so that a one-to-one assignment is possible.

For each mapping that it tries, the algorithm measures how many edges of G fail to appear in H when vertices are placed according to it. This measure is called the *extension cost* for that mapping.

Key outputs and how they depend on the mapping

Best mapping. Among all injective mappings examined, the algorithm keeps the mapping that produces the smallest extension cost. We call that the **best mapping** (denote it $\hat{f}$). The best mapping is important because it is the choice of which vertex of H stands for each vertex of G that makes G fit into H with the fewest modifications.

Best distance (minimum extension cost). For a given mapping f, the *distance* (also called the *extension cost*) is the number of edges that exist in G but are *missing* in H after the vertices are identified by f. Formally, if $u, v \in V(G)$ and $f(u), f(v) \in V(H)$, every time (u, v) is an edge of G but $(f(u), f(v))$ is *not* an edge of H, we add one to the cost.

The algorithm computes this cost for every mapping and records the smallest cost found. That smallest value is returned as **bestDistance**. Thus, **bestDistance** is directly derived from the **best mapping**: it is the number of missing edges when we use $\hat{f}$ to compare G with H.

Extended graph H_{ext}. Once the algorithm has found the best mapping $\hat{f}$, it can construct an extended version of H, called H_{ext}, by adding *only* the edges that are required so that every edge of G appears in H_{ext} according to $\hat{f}$. Concretely, for each edge (u, v) in G, if the corresponding edge $(\hat{f}(u), \hat{f}(v))$ is missing in H, we add that edge to H to obtain H_{ext}.

By construction:

$$\text{number of edges added to } H = \textbf{bestDistance}.$$

Therefore H_{ext} depends entirely on the best mapping $\hat{f}$: different mappings may force different edges to be added, so choosing the mapping that minimizes missing edges yields the smallest H_{ext} change.

How the Algorithm Explores All Possible Mappings

To find the best mapping, the algorithm must consider every possible injective assignment of vertices from G to different vertices of H. This is done by the procedure called `FindBestMapping`, which performs a systematic search through all assignments.

Step-by-step idea The vertices of G are processed one at a time in a fixed order:

$$v_G = 0, 1, 2, \ldots, n-1.$$

When the algorithm is deciding where to place a particular vertex v_G of G, it tries every vertex v_H of H that has not yet been used for another vertex of G.

Each choice temporarily extends the current partial mapping. Then the algorithm continues recursively to assign the next vertex of G.

Avoiding duplicate assignments To ensure the mapping remains injective (one-to-one), the algorithm keeps track of which vertices of H are already used. If v_H is already assigned to a previous vertex of G, it is skipped.

Reaching a full mapping When the recursion reaches the point where all n vertices of G have been assigned, a complete valid mapping is constructed. At this moment:

- the algorithm computes the cost (how many edges of G are missing in H under this mapping),
- and it updates the current best result if this mapping requires fewer missing edges.

Backtracking After checking a mapping that uses a certain choice for v_G, the algorithm *undoes* that choice:

- it marks v_H as unused again,
- it removes the temporary assignment $v_G \mapsto v_H$,

and then tries the next possible vertex of H.

This "try, recurse, undo" process allows the algorithm to explore every possible injective mapping without repeating or skipping any.

Why this guarantees completeness Since:

1. Every vertex of G is assigned exactly once,
2. Every unused vertex of H is tested as a candidate at each step, and
3. The recursion visits all branches of the search tree,

every injective mapping from G into H is generated exactly once. Therefore, the algorithm is guaranteed to find the mapping with the smallest extension cost.

In summary, `FindBestMapping` is responsible for exploring all possible placements of G inside H. It builds mappings vertex-by-vertex, checks completed mappings, and uses backtracking to ensure that every legal injective mapping is considered.

4 Pseudo-Code and Time-Complexity

4.1 ComputeDistance

```
// O(n^2)
ComputeDistance(G, n, H, m, mapping):
    // O(1)
    distance ← 0

    // Double nested loop: O(n^2)
    for uG from 0 to n  1 do
    for vG from 0 to n  1 do
        // Each check O(1)
        if G[uG][vG] == 1 then
        uH = mapping[uG]     // O(1)
        vH = mapping[vG]     // O(1)
        if H[uH][vH] == 0 then // O(1)
            distance = distance + 1

    // O(1)
    return distance
```

4.2 FindBestMapping

```
// TOTAL CALLS = number of injective mappings = P(m,n) = m!/(mn)!
// At each leaf, ComputeDistance() costs O(n²).
// TOTAL COMPLEXITY = P(m,n) * O(n²) = O(m!/(mn)! * n²)
FindBestMapping(vG, G, n, H, m, mapping, usedH):
    // Check condition: O(1)
    if vG == n:
    // ComputeDistance = O(n^2)
    distance ← ComputeDistance(G, n, H, m, mapping)
    // O(1)
    if distance < bestDistance then
        // O(n) copy
        bestDistance = distance
        bestMapping = copy of mapping
        return
    // Loop runs at most m times per recursion level
    // Recursion depth = n
    for vH from 0 to m  1 do           // O(m)
    if usedH[vH] == false:          // O(1)
        // Assignments O(1)
        mapping[vG] = vH
        usedH[vH] = true
        // Recursive call:
        // TOTAL CALLS = number of injective mappings = P(m,n) = m!/(mn)!
        FindBestMapping(vG + 1, G, n, H, m, mapping, usedH)
        // Undo ops O(1)
        usedH[vH] = false
        mapping[vG] = -1
```

4.3 ExtendGraph

```
// O(n²) Worst-case complexity
ExtendGraph(H, G, M_G, bestMapping):

    // Copy: O(m²) but since m=n after padding → O(n²)
    H_ext ← copy of H

    // Double nested: O(n^2)
    for uG from 0 to n-1 do
    for vG from 0 to n-1 do
        // O(1)
        if M_G[uG][vG] == 1 then
        uH ← bestMapping[uG]   // O(1)
        vH ← bestMapping[vG]   // O(1)
        // check/insert edge: O(1)
        if there is no edge uH → vH in H_ext then
            add directed edge uH → vH to H_ext

    // O(1)
    return H_ext
```

4.4 ExactMinExtendGrap(G,H)

```
// O(m!/(mn)! * n²) complexity
ExactMinExtendGrap(G, H):
    // O(1)
    n ← number of vertices in G
    // O(1)
    m ← number of vertices in H

    // O(1)
    if m < n:
    // O(1) for updating m + cost of allocation ignored in complexity
    H = H with (n  m) new vertices
    m = n

    // O(n) initialization
    mapping[0..n1] = -1 // -1 - unmapped
    // O(m) initialization
    usedH[0..m1] = false
    // O(1)
    bestDistance = MaxInt
    // O(n)
    bestMapping = copy of mapping

    // O(m!/(mn)! * n²) complexity
    best_mapping, best_distance = FindBestMapping(0, G, n, H, m, mapping, usedH)

    // AddMissingEdges: O(n^2)
    H_ext ← ExtendGraph(H, M_G, n, bestMapping)
```

```
// O(1)
isSubgraph = (bestDistance == 0)
// O(1)
return isSubgraph, bestDistance, bestMapping, H_ext
```

4.5 Time-Complexity

In the pseudo-code the total time-complexity was shown to be $O\Big(\frac{m!}{(m-n)!} \cdot n^2\Big)$. In this section we will prove that our time complexity is within exponential complexity.

To make the exponential nature explicit, we apply Stirling's approximation to the factorial term in the time complexity. For large k,

$$k! \sim \sqrt{2\pi k}\left(\frac{k}{e}\right)^k.$$

Using this for the permutation term, we obtain

$$\frac{m!}{(m-n)!} \sim \frac{\sqrt{2\pi m}\,(\frac{m}{e})^m}{\sqrt{2\pi(m-n)}\,(\frac{m-n}{e})^{m-n}} = \sqrt{\frac{m}{m-n}}\left(\frac{m}{m-n}\right)^{m-n} m^n e^{-n}.$$

Including the n^2 factor from the distance computation, the total time complexity is

$$T(n,m) = \frac{m!}{(m-n)!} \cdot n^2 \sim \sqrt{\frac{m}{m-n}}\, n^2 \left(\frac{m}{m-n}\right)^{m-n} m^n e^{-n}.$$

Even without simplifying further, we can see that $T(n,m)$ grows faster than any fixed-base exponential c^n for large n and $m \geq n$. Therefore, the algorithm has **exponential (or super-exponential) time complexity** in the worst case.

5 Hungarian Approximation Algorithm

We implement two approaches:

5.1 Exact Algorithm

Uses exhaustive search to find the optimal mapping. For a single copy, it explores all possible injective mappings $f : V(G) \to V(H)$ and selects the one requiring the minimum number of edge additions. For multiple copies, it uses a greedy approach that iteratively finds optimal mappings for each copy.

Time complexity: $\mathcal{O}(|V(H)|!)$ for single copy, exponentially worse for multiple copies.

5.2 Hungarian Algorithm Approximation

The exact algorithm is exponential in $|V(G)|$, so for larger graphs we use a polynomial-time approximation. We formulate the task as an *assignment problem*: each vertex of G must be assigned to a distinct vertex of H while minimizing a cost function. The assignment is solved using the **Hungarian algorithm** (Kuhn–Munkres), which runs in $\mathcal{O}(N^3)$ time.

5.3 Cost Function

We use a sophisticated cost function that considers edge structure compatibility:

$$C(u,v) = \sum_{w \in V(G)} \text{EdgePenalty}(u,w,v) + |deg_G(u) - deg_H(v)| + \frac{u+v}{10}$$

where:

$$\text{EdgePenalty}(u,w,v) = \begin{cases} 20 & \text{if } (u,w) \in E(G) \text{ and } outdeg_H(v) = 0 \\ 20 & \text{if } (w,u) \in E(G) \text{ and } indeg_H(v) = 0 \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

This heuristic heavily penalizes mapping G vertices to H vertices that lack compatible edge structure, while using degree similarity as a secondary factor.

5.4 Multiple Copies Extension

For multiple copies, we track used vertices in H and set their assignment cost to 10^6 (forbidden). The algorithm iteratively finds non-overlapping mappings until the target number of copies is reached or no more valid mappings exist.

5.5 Pseudocode

Algorithm 1 HungarianApproximateExtend

Require: Graphs G and H, target copies k
Ensure: Extended graph H' containing k copies of G

$n \leftarrow |V(G)|$, $m \leftarrow |V(H)|$
$usedH \leftarrow$ array of m false values
$H' \leftarrow H$, $copiesFound \leftarrow 0$
while $copiesFound < k$ and enough unused vertices remain **do**
    $hungarian \leftarrow$ new HungarianAlgor ithm(m)
    for $i \leftarrow 0$ to $m - 1$ **do**
        for $j \leftarrow 0$ to $m - 1$ **do**
            if $i < n$ **then**
                if $usedH[j]$ **then**
                    $cost \leftarrow 10^6$ ▷ forbidden
                else
                    $cost \leftarrow$ ComputeEdgeStructureCost($G[i], H[j]$)
                end if
            else
                $cost \leftarrow 0$ ▷ dummy row
            end if
            $hungarian.setCost(i, j, cost)$
        end for
    end for
    $mapping \leftarrow hungarian.findMinCostAssignment()$
    if valid mapping found **then**
        Mark $mapping$ vertices as used in $usedH$
        Add missing edges from G to H' according to $mapping$
        $copiesFound \leftarrow copiesFound + 1$
    else
        break
    end if
end while
return H'

5.6 Time Complexity Analysis

Let $N = \max(|V(G)|, |V(H)|)$ and k be the number of copies.

- Single Hungarian solve: $\mathcal{O}(N^3)$
- Cost matrix construction: $\mathcal{O}(N^3)$ (due to edge structure analysis)
- Multiple copies: k iterations
- Edge extension: $\mathcal{O}(k \cdot |V(G)|^2)$

Therefore total time complexity is:

$$\boxed{\mathcal{O}(k \cdot N^3)}.$$

6 Experimental Results

We tested our implementation on directed graphs of varying sizes using an automated test suite. The results demonstrate the trade-off between optimality and computational efficiency, and validate the multiple copies functionality.

6.1 Basic Performance Comparison

Results from deterministic directed graphs:

$\|V(G)\|$	$\|V(H)\|$	Exact Cost	Exact Time (ms)	Hungarian Cost	Hungarian Time (ms)
3	5	1	0	1	0.004
4	6	1	0	3	0.003
5	8	1	1	2	0.004
6	10	0	10	4	0.011

6.2 Direct Algorithm Comparison

Testing both algorithms on identical deterministic graph instances:

Test Case	Exact Edges	Hungarian Edges	Error Rate	Speedup
G(3) vs H(6)	1	1	0.0%	3.0x
G(4) vs H(8)	1	1	0.0%	33.5x
G(5) vs H(10)	1	1	0.0%	368.5x
G(6) vs H(12)	2	5	150.0%	2375x

6.3 Exponential Behavior Demonstration

The exact algorithm exhibits exponential time complexity as graph size increases:

Graph Size	Exact Time	Hungarian Time	Performance Gap
G(3)	$<$ 1ms	0.004ms	Small
G(4)	$<$ 1ms	0.003ms	Moderate
G(5)	1ms	0.004ms	Large
G(6)	38ms	0.016ms	Exponential

6.4 Multiple Copies Validation

Testing multiple copy functionality with various target numbers:

6.4.1 G(4) vs H(10) Multiple Copies Test

Target	Exact Found	Exact Edges	Hungarian Found	Hungarian Edges
1	1 (0ms)	0	1 (0.011ms)	0
2	2 (0ms)	0	2 (0.008ms)	3
3	2 (0ms)	0	2 (0.008ms)	3

6.4.2 Larger Graph Multiple Copies Tests

Results demonstrate perfect multiple copy functionality:

- Both algorithms successfully find non-overlapping subgraph copies
- Exact algorithm finds optimal solutions (0 edges needed for existing subgraphs)

- Hungarian provides fast approximation with small overhead (3 edges vs 0 optimal)
- Graph structure limits maximum copies achievable regardless of target

6.5 Scalability Analysis

The Hungarian algorithm maintains consistent polynomial performance while exact becomes impractical:

Problem Size	Hungarian Time	Scalability
Small (G(3) vs H(6))	0.003ms	Excellent
Medium (G(5) vs H(10))	0.005ms	Excellent
Large (G(6) vs H(12))	0.016ms	Excellent

7 Conclusions

Our comprehensive experimental evaluation demonstrates several key findings:

1. **Performance**: The Hungarian algorithm provides substantial speedup (3x to 2375x) over the exact approach, with the performance gap growing exponentially as graph size increases.
2. **Approximation Quality**: On our deterministic test cases, Hungarian achieves optimal results for smaller graphs (0% error) and reasonable approximation for complex cases (150% error for G(6) vs H(12)).
3. **Scalability**: The exact algorithm becomes impractical beyond 6 vertices (38ms), while Hungarian maintains consistent sub-millisecond performance across all test cases.
4. **Multiple Copies**: Both algorithms successfully handle multiple non-overlapping copies. The exact algorithm finds optimal solutions (0 edges needed when subgraphs exist), while Hungarian provides fast approximations with minimal overhead.
5. **Deterministic Behavior**: Using fixed test graphs enables reproducible results and reliable performance analysis, crucial for academic evaluation and algorithm comparison.
6. **Practical Applicability**: For real-world directed subgraph isomorphism with multiple copy requirements, Hungarian provides excellent efficiency while exact remains suitable only for small instances.

The Hungarian approximation algorithm successfully addresses the computational limitations of exact approaches while maintaining reasonable solution quality, making it viable for practical directed subgraph isomorphism applications with multiple copy requirements.

8 References and Sources Used

To satisfy the "closed document" requirement, the key sources that constitute the basis of the approximation method are listed below, and their PDF copies are included in the submission archive under `Doc/` (where applicable).

8.1 Exact Solution

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd Edition, MIT Press, 2009.

[2] D. E. Knuth, *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, 3rd Edition, Addison-Wesley, 1997.

[3] J. Stirling, *Methodus Differentialis: Sive Tractatus de Summatione et Interpolatione Serierum Infinitarum*, 1730.

[4] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman, 1979.

[5] G. Brassard and P. Bratley, *Fundamentals of Algorithmics*, Prentice Hall, 1996.

[6] J. A. Bondy and U. S. R. Murty, *Graph Theory*, Springer, 2008.

[7] D. B. West, *Introduction to Graph Theory*, 2nd Edition, Prentice Hall, 2001.

[8] D. E. Knuth, *Concrete Mathematics: A Foundation for Computer Science*, 2nd Edition, Addison-Wesley, 1998.

8.2 Hungarian Algorithm (Assignment Problem)

- Wikipedia contributors, "Hungarian algorithm," *Wikipedia, The Free Encyclopedia*. `https://en.wikipedia.org/wiki/Hungarian_algorithm`
- "The Hungarian Algorithm," hungarianalgorithm.com (explanatory web resource). `https://www.hungarianalgorithm.com/hungarianalgorithm.php`
- Harold W. Kuhn, "The Hungarian Method for the Assignment Problem," *Naval Research Logistics Quarterly*, 2(1–2), 1955, pp. 83–97.
- R. E. Burkard, M. Dell'Amico, S. Martello, *Assignment Problems*, SIAM, 2009.