# A Comprehensive Guide to Subgraph Isomorphism, Minimal Graph Extension, and Approximation Techniques

## Introduction: Framing the Computational Challenge

The task of comparing graph structures is a fundamental problem in computer science, with applications ranging from pattern recognition in social networks and computer vision to substructure searching in bioinformatics and chemistry.[1] The problem presented involves two distinct but related computational challenges concerning a pair of graphs, a smaller "pattern" graph $G_{small}$ and a larger "target" graph $G_{big}$. The first challenge is to determine if $G_{small}$ exists as a subgraph within $G_{big}$. The second, more complex challenge arises when no such subgraph exists: to find the minimal set of additions (vertices and edges) to $G_{big}$ that would create a new graph, $G'_{big}$, which does contain $G_{small}$ as a subgraph.

At the heart of this problem lies the Subgraph Isomorphism (SI) problem, a classic computational task that is known to be NP-complete.[5] Its NP-completeness is demonstrated through a straightforward reduction from other well-known NP-complete problems, such as the Clique problem or the Hamiltonian Cycle problem.[5] For instance, determining if a graph $G$ contains a clique of size $k$ is equivalent to asking if the complete graph $K_k$ is a subgraph of $G$.[5] This classification has profound implications: it means that no known algorithm can provide a correct solution for all possible inputs in a time that scales polynomially with the size of the graphs. Consequently, any exact algorithm that guarantees an optimal solution will, in the worst case, require a runtime that grows exponentially with the input size.[1]

This computational barrier directly necessitates the dual-algorithm approach requested:

1. An **exact algorithm** that provides a provably optimal solution but operates with exponential time complexity.
2. An **approximate algorithm** that sacrifices guaranteed optimality in favor of a practical, polynomial-time complexity, aiming for a "good" but not necessarily perfect solution.[9]

A critical clarification of the problem reveals that the task of finding a "minimal extension" is not best approached by directly considering what to add. Instead, it is more effectively solved by first determining what is already shared between the two graphs. The minimal set of components to add to $G_{big}$ to embed $G_{small}$ corresponds precisely to the parts of

$G_{small}$ that are *not* found within the largest possible common structure shared by both graphs. This largest shared structure is known as the **Maximum Common Subgraph (MCS)**. The process of taking the union of $G_{big}$ with the missing parts of $G_{small}$ results in the **Minimum Common Supergraph (MCSG)** of the two.[11] This establishes a fundamental duality: the problem of minimizing the extension is equivalent to the problem of maximizing the common subgraph. This perspective reframes the entire challenge, transforming it from an ambiguous construction task into a well-defined search for maximal commonality, thereby providing a clear and established algorithmic path forward.

This report is structured to address this multifaceted problem comprehensively. Part I will detail the exact, exponential-time solution, covering both subgraph isomorphism detection and the subsequent minimal extension via the Maximum Common Subgraph. Part II will explore approximate, polynomial-time solutions by presenting heuristic methods for the same tasks. Finally, Part III will serve as a practical guide to implementation, discussing graph data structures and surveying available software libraries and codebases.

---

# Part I: The Exact Algorithm — An Optimal Solution with Exponential Complexity

This section details the methodology for finding a provably optimal solution to the stated problem. The approach follows a two-stage logical process. First, it addresses the decision problem of whether an exact subgraph isomorphism already exists. If it does not, the second stage addresses the optimization problem of computing the minimal extension by leveraging the duality with the Maximum Common Subgraph problem.

## Section 1. Foundational Problem: Exact Subgraph Isomorphism Detection

Before attempting to extend a graph, it is necessary to first determine if an extension is needed at all. This requires solving the Subgraph Isomorphism (SI) problem.

### 1.1. Theoretical Underpinnings of the Subgraph Isomorphism Problem

The Subgraph Isomorphism problem exists in several variations, and a precise understanding of their definitions is crucial. Let $G_{small} = (V_s, E_s)$ be the pattern graph and $G_{big} = (V_b, E_b)$ be the target graph.

- **Graph Isomorphism:** $G_{small}$ and $G_{big}$ are isomorphic if there exists a bijective (one-to-one and onto) mapping $f: V_s \to V_b$ such that for any two vertices $u, v \in V_s$, the edge $(u, v) \in E_s$ if and only if the edge $(f(u), f(v)) \in E_b$. This implies the graphs are structurally identical.[6]

- **Subgraph Isomorphism (non-induced):** $G_{small}$ is isomorphic to a subgraph of $G_{big}$ if there exists an injective (one-to-one) mapping $f: V_s \to V_b$ such that for every edge $(u, v) \in E_s$, the edge $(f(u), f(v))$ exists in $E_b$.[3] This is the standard definition and the one relevant to the user's query. It allows the subgraph in $G_{big}$ to have *more* edges than are required by $G_{small}$.
- **Induced Subgraph Isomorphism:** This is a stricter condition requiring that for any two vertices $u, v \in V_s$, the edge $(u, v) \in E_s$ *if and only if* the edge $(f(u), f(v)) \in E_b$.[14] This forbids extra edges between mapped vertices in the target graph.

While the worst-case complexity of SI is exponential, this landscape is nuanced. If the pattern graph $G_{small}$ is of a fixed, constant size $k$, the problem becomes solvable in polynomial time, with a brute-force approach having a complexity of $O(|V_{big}|^k \cdot k^2)$.[5] Furthermore, if the target graph $G_{big}$ has special structural properties, such as being planar, and $G_{small}$ is fixed, the runtime can be reduced to linear time.[5] However, if both graphs are arbitrary inputs, the NP-complete nature dominates.

### 1.2. The Ullmann Algorithm: A Backtracking and Refinement Approach

Published in 1976, Ullmann's algorithm is a foundational backtracking procedure for solving the subgraph isomorphism problem.[5] Its strategy is to intelligently search the space of all possible vertex mappings.

The algorithm's core is a boolean matrix $M$ of size $|V_s| \times |V_b|$, where an entry $M_{ij} = 1$ indicates that vertex $v_i \in V_s$ is potentially mappable to vertex $v_j \in V_b$.[8] Initially, this matrix is populated based on a simple necessary condition, or graph invariant: a vertex $v_i$ can only map to $v_j$ if its degree is less than or equal to the degree of $v_j$, i.e., $deg(v_i) \le deg(v_j)$.[8] If vertex or edge labels are present, these must also match.[19]

The key to Ullmann's algorithm is its **refinement procedure**, which iteratively prunes the search space by enforcing neighborhood consistency.[1] For a potential mapping $M_{ij} = 1$ to remain valid, it must be true that for every neighbor $v_x$ of $v_i$ in $G_{small}$, there must exist at least one neighbor $v_y$ of $v_j$ in $G_{big}$ such that $M_{xy} = 1$. If this condition is not met for any neighbor of $v_i$, the mapping is impossible, and $M_{ij}$ is set to 0. This change can trigger a cascade of further invalidations, so the refinement process is repeated until no more entries in $M$ can be changed to 0.[8]

The algorithm then proceeds with a recursive backtracking search, attempting to find a valid assignment of one '1' per row and at most one '1' per column that satisfies the isomorphism condition. The efficiency of this search can be improved with heuristics, such as ordering the vertices of the pattern graph by decreasing degree to encourage earlier pruning.[8]

### 1.3. The VF2 Algorithm: A State-Space Search with Advanced Pruning

The VF2 algorithm, proposed by Cordella et al. in 2004, represents a significant improvement over Ullmann's method, particularly for large or sparse graphs.[5] Instead of using costly matrix manipulations, VF2 employs a depth-first, **state-space search** methodology.[26] In this framework, each state corresponds to a partial mapping of vertices from $G_{small}$ to $G_{big}$. The algorithm starts with an empty mapping and recursively extends it, one vertex pair at a time.
The efficiency of VF2 stems from a set of inexpensive **feasibility rules** that are checked at each step to prune entire branches of the search tree.[4] These rules check for both syntactic and semantic consistency. For a candidate pair $(u, v)$ to be added to the current partial mapping, the algorithm verifies:

1. **One-to-one mapping:** $v$ is not already mapped.
2. **Structural consistency:** The neighborhood structure of $u$ and $v$ is consistent with the vertices already in the partial mapping.
3. **Look-ahead rules:** It checks the number of neighbors of $u$ and $v$ that are *outside* the current partial mapping to determine if a future complete mapping is plausible. These are often referred to as 1-look-ahead and 2-look-ahead rules.[31]

A major advantage of VF2 is its memory efficiency. It has a linear space complexity of $O(|V_s|)$, a significant improvement over Ullmann's quadratic $O(|V_s| \cdot |V_b|)$ requirement, making it far more suitable for matching very large graphs.[24] The VF2 algorithm has been further refined in subsequent versions like VF2 Plus and VF2++, which introduce more sophisticated heuristics for node ordering and more powerful cutting rules to further accelerate the search.[27]

### 1.4. Comparative Analysis of Ullmann and VF2

While both algorithms solve the same NP-complete problem and thus share an exponential worst-case time complexity, their practical performance and resource usage differ significantly. The choice between them depends on the specific characteristics of the graphs and the implementation context.

| **Feature** | **Ullmann's Algorithm** | **VF2 Algorithm** |
|---|---|---|
| **Core Method** | Backtracking with matrix refinement | Depth-first state-space search |
| **Pruning Strategy** | Neighborhood compatibility check (refinement) | Syntactic/semantic feasibility rules (k-look-ahead) |
| **Worst-Case Time** | Exponential (e.g., $O( | V_s |
| **Space Complexity** | $O( | V_s |
| **Typical Use Case** | Foundational; still effective on smaller or dense graphs. | Generally faster and more memory-efficient; state-of-the-art for general-purpose exact matching, especially on large, |

| | | |
|---|---|---|
| | | sparse graphs. |

## Section 2. The Extension Problem: Finding the Minimal Common Supergraph

If the subgraph isomorphism check fails, the task becomes finding the minimal extension to $G_{big}$. As established, this is equivalent to finding the Minimum Common Supergraph (MCSG), which is constructed from the Maximum Common Subgraph (MCS).

### 2.1. Reframing Minimal Extension via the Maximum Common Subgraph (MCS)

The concept of a "minimal extension" is formalized by the **Minimum Common Supergraph (MCSG)**, defined as the smallest graph that contains both $G_{small}$ and $G_{big}$ as subgraphs.[11] The computation of the MCSG is directly linked to the **Maximum Common Subgraph (MCS)**.[11] The MCS is the largest graph that is isomorphic to a subgraph of both $G_{small}$ and $G_{big}$.
Once the MCS is identified, the MCSG can be constructed by taking the union of the vertices and edges of both graphs, where the components corresponding to the MCS are merged. The minimal number of edges to add to $G_{big}$ is therefore $|E_s| - |E_{mcs}|$, and the number of vertices is $|V_s| - |V_{mcs}|$. The extension itself consists precisely of those vertices and edges from $G_{small}$ that are not part of the identified MCS mapping.
Like subgraph isomorphism, the MCS problem has two main variants:

- **Maximum Common Induced Subgraph (MCIS):** Maximizes the number of vertices in the common subgraph, with the strict induced subgraph condition.[35]
- **Maximum Common Edge Subgraph (MCES):** Maximizes the number of edges in the common subgraph, using the non-induced subgraph condition.[35] Since the user's goal is to add edges to make $G_{small}$ a subgraph (non-induced), the MCES is the more relevant formulation.

### 2.2. The Modular Product Graph and Reduction to Maximum Clique

The standard exact algorithm for finding the MCIS involves a polynomial-time reduction to the **Maximum Clique problem**, another NP-hard problem.[36] This is achieved by constructing an auxiliary graph called the **modular product** or compatibility graph.
The construction of the modular product graph, $G_p$, from $G_{small}=(V_s, E_s)$ and $G_{big}=(V_b, E_b)$ proceeds as follows:

1. The vertex set of $G_p$ is the Cartesian product $V_p = V_s \times V_b$. Each vertex

   $(u, v) \in V_p$ represents a potential mapping of vertex $u \in V_s$ to vertex $v \in V_b$.
2. An edge is created between two vertices $(u_1, v_1)$ and $(u_2, v_2)$ in $G_p$ if and only if the structural relationship between $u_1$ and $u_2$ in $G_{small}$ is compatible with the relationship between $v_1$ and $v_2$ in $G_{big}$. For an MCIS, this compatibility means that an edge $(u_1, u_2)$ exists in $E_s$ *if and only if* an edge $(v_1, v_2)$ exists in $E_b$.[36]

The fundamental result of this construction is that a **maximum clique** in the modular product graph $G_p$ corresponds directly to a **maximum common induced subgraph** between $G_{small}$ and $G_{big}$. The vertices in the clique, such as $\{(u_1, v_1), (u_2, v_2), \dots\}$, define the exact vertex mapping of the MCIS. To find the MCES, this same technique can be applied not to the original graphs, but to their **line graphs**, where vertices represent edges and edges represent adjacency between the original edges.[36]

### 2.3. A Step-by-Step Guide to Constructing the Exact Minimal Extension

Combining these concepts yields a complete, exact algorithm for the user's problem:

1. **Input:** Graphs $G_{small}$ and $G_{big}$.
2. **Step 1: Check for Direct Subgraph Isomorphism.** Execute the VF2 algorithm to determine if $G_{small}$ is already a subgraph of $G_{big}$. If a mapping is found, the minimal extension is null, and the algorithm terminates.
3. **Step 2: Find the Maximum Common Subgraph.** If no isomorphism exists, proceed to find the MCS. For an MCES-based extension, construct the line graphs of $G_{small}$ and $G_{big}$. Then, build the modular product of these line graphs. Find the maximum clique in the modular product graph using an exact algorithm like the Bron-Kerbosch algorithm.[36] This clique identifies the edge mapping of the MCES. From this, the corresponding vertex mapping can be derived.
4. **Step 3: Identify Missing Components.** Compare the vertex and edge sets of $G_{small}$ with those included in the MCS found in the previous step. Compile a list of all vertices and edges from $G_{small}$ that are not part of the MCS.
5. **Step 4: Construct the Extended Graph.** Create a new graph, $G_{extended}$, as a copy of $G_{big}$. Add the missing vertices and edges identified in Step 3 to $G_{extended}$. Care must be taken to handle vertex identities correctly, ensuring that new vertices are distinct and edges connect to the appropriate vertices (some of which may already exist in $G_{big}$ as part of the MCS mapping).
6. **Output:** The graph $G_{extended}$, which now contains $G_{small}$ as a subgraph and represents the minimal such extension.

---

## Part II: The Approximate Algorithm — A Practical

# Solution with Polynomial Complexity

For many practical applications involving large graphs, the exponential runtime of exact algorithms is prohibitive. This section explores approximate methods that trade guaranteed optimality for polynomial-time performance, providing solutions that are "good enough" for the task at hand.

## Section 3. Principles of Algorithmic Approximation for NP-Hard Graph Problems

### 3.1. Understanding Heuristics vs. Guaranteed Approximation Ratios

When dealing with NP-hard problems, it is important to distinguish between two types of non-exact algorithms:

- **Heuristic Algorithms:** These employ problem-solving strategies based on rules of thumb, educated guesses, or simplifications to find a solution quickly.[21] Heuristics often perform well on typical real-world instances but offer no formal guarantee on how close their solution is to the true optimum. Their effectiveness is usually demonstrated empirically.
- **Approximation Algorithms:** These are more formally defined. An $\alpha$-approximation algorithm for a maximization problem is one that is proven to always find a solution that is at least $\alpha$ times the size of the optimal solution, where $\alpha < 1$ is the approximation ratio. For minimization, the solution is guaranteed to be no more than $\beta$ times the optimum.

For the Maximum Common Subgraph problem, it has been shown that achieving a strong approximation ratio is as difficult as approximating the Maximum Clique problem.[38] This means that no polynomial-time algorithm can provide a good constant-factor approximation unless P=NP. Therefore, for the user's purpose, a heuristic approach that works well in practice is the most viable path for a polynomial-time solution.

### 3.2. The Trade-off: Sacrificing Optimality for Tractability

The exponential complexity of exact algorithms arises from the need to explore a combinatorial search space that grows factorially with the number of vertices. Approximate algorithms circumvent this by making locally optimal or "greedy" choices that drastically reduce the search space.[37] This is the fundamental trade-off: by not exploring all possibilities, the algorithm runs much faster but risks missing the globally optimal solution. The goal is to design a heuristic that makes intelligent choices, leading to a high-quality solution most of the time.

## Section 4. Heuristic and Greedy Approaches to the Extension Problem

The duality between minimal extension and maximum commonality holds true in the approximate case as well. To find a *near-minimal* extension in polynomial time, one must first find an *approximate* Maximum Common Subgraph. The quality of the final extended graph is directly contingent on the quality of the MCS approximation. If the heuristic finds an MCS that is close in size to the true optimum, the resulting extension will also be close to the true minimum.

### 4.1. Approximating the Maximum Common Subgraph

Several heuristic strategies can be employed to find a large common subgraph in polynomial time.

#### 4.1.1. Greedy Construction and Local Search Heuristics

A straightforward and effective heuristic is a greedy, best-first search approach.[37] The algorithm can be outlined as follows:

1. **Initialization:** Start with an empty mapping for the common subgraph.
2. **Iterative Selection:** In each step, evaluate all possible pairs of unmapped vertices (one from $G_{small}$, one from $G_{big}$) based on a heuristic scoring function. This function might consider vertex degree, the number and frequency of vertex/edge labels, or the similarity of local neighborhood structures.
3. **Greedy Choice:** Select the highest-scoring pair that is compatible with the current partial mapping and add it to the solution.
4. **Termination:** Repeat until no more compatible pairs can be added.

This greedy solution can often be improved using a **local search** phase. Starting with the greedily constructed MCS, the algorithm can iteratively attempt to make small modifications—such as swapping one vertex mapping for another or replacing a set of mappings—if the change results in a larger common subgraph. This helps the algorithm escape from poor local optima reached during the initial greedy construction.

#### 4.1.2. Reformulation-Based Approaches

Another class of heuristics involves reformulating the problem. Just as the exact MCIS problem can be reduced to finding a maximum clique in the modular product, this relationship

can be exploited for approximation. Instead of solving Maximum Clique exactly, one can use a well-known polynomial-time approximation algorithm for it.[40]
Alternatively, the MCS problem can be related to the **Maximum Independent Set (MIS)** problem, which is equivalent to Maximum Clique on the complement graph. There are effective reduction heuristics for MIS that can be used to prune the graph size, enabling faster computation and yielding near-optimal solutions for the corresponding MCS problem.[41] Other advanced approaches include using genetic algorithms [33] or formulating the problem as a Quadratic Assignment Problem (QAP) [42], though these are typically more complex to implement.

### 4.2. Constructing a Near-Optimal Extension in Polynomial Time

Synthesizing these concepts leads to a complete polynomial-time algorithm for finding a good approximation of the minimal extension.

1. **Input:** Graphs $G_{small}$ and $G_{big}$.
2. **Step 1: Approximate the Maximum Common Subgraph.** Run a polynomial-time heuristic algorithm to find an approximate MCS. A greedy construction algorithm is a strong candidate for its simplicity and effectiveness.
3. **Step 2: Identify Missing Components.** Based on the mapping of the *approximate* MCS, identify the vertices and edges present in $G_{small}$ but not included in the common subgraph.
4. **Step 3: Construct the Extended Graph.** Create a copy of $G_{big}$ and add the missing components identified in the previous step.
5. **Output:** The near-minimal extended graph, $G_{extended}$.

---

# Part III: Implementation Guide and Resource Compendium

This section provides practical resources and guidance for implementing the algorithms discussed, from selecting appropriate data structures to leveraging existing high-performance libraries.

## Section 5. Practical Considerations for Graph Representation

### 5.1. Selecting the Right Data Structure: Adjacency Lists vs. Matrices

The choice of data structure for representing a graph is a critical performance consideration. The most common representations are the adjacency list and the adjacency matrix.[43]

- **Adjacency List:** Each vertex is associated with a list of its neighbors. This is the most memory-efficient representation for **sparse graphs** (where the number of edges $|E|$ is much less than $|V|^2$), which are common in real-world applications. Operations like iterating over a vertex's neighbors are very fast.
- **Adjacency Matrix:** A $|V| \times |V|$ matrix where the entry $(i, j)$ is 1 if an edge exists from vertex $i$ to vertex $j$, and 0 otherwise. This representation is better for **dense graphs** (where $|E|$ is close to $|V|^2$) and allows for constant-time edge existence checks. However, its $O(|V|^2)$ space complexity is prohibitive for large, sparse graphs.

For most general-purpose graph algorithm implementations, the adjacency list is the preferred choice due to its superior space efficiency and performance on the types of graphs commonly encountered in practice.[43]

### 5.2. Overview of Key Graph Libraries

Several mature libraries provide robust implementations of graph data structures and algorithms, which can significantly accelerate development. The most prominent for this task are NetworkX (Python), the Boost Graph Library (C++), and igraph (C, with Python and R wrappers).[43]

## Section 6. A Curated List of Libraries and Codebases

### 6.1. High-Performance C++ Implementations: Boost Graph Library and igraph

- **Boost Graph Library (BGL):** BGL is a powerful, generic C++ library known for its performance. It provides a highly configurable implementation of the VF2 algorithm.
  - **Functionality:** The function vf2_subgraph_iso is available for finding *induced* subgraph isomorphisms, while vf2_subgraph_mono handles the standard (non-induced) case, which is what the user's problem requires.[45] The library uses callbacks to report each found isomorphism.[45]
- **igraph:** This is a high-performance C library with excellent wrappers for Python and R.
  - **Functionality:** Like BGL, igraph's primary tool for this problem is its implementation of the VF2 algorithm, accessible via the igraph_subisomorphic_vf2 function.[47] It also provides implementations of other isomorphism algorithms like Bliss for more specialized tasks.[47]

## 6.2. Rapid Prototyping with Python: NetworkX

- **Functionality:** NetworkX is an excellent choice for rapid development and experimentation due to its user-friendly Python API. It provides the VF2 algorithm via the GraphMatcher and DiGraphMatcher classes.[29]
    - **Subgraph Isomorphism:** The subgraph_is_isomorphic() method returns a boolean indicating if an isomorphism exists, while subgraph_isomorphisms_iter() provides an iterator over all possible mappings.[29]
    - **Maximum Common Subgraph:** NetworkX does **not** provide a direct function for finding the Maximum Common Subgraph (MCES or MCIS). The ISMAGS class includes a largest_common_subgraph() method, but this is specifically for finding the largest common *induced* subgraph, which may not be suitable for the minimal extension problem as formulated.[51] An MCES algorithm would need to be sourced from another library or implemented manually.

## 6.3. Annotated Standalone GitHub Repositories for Core Algorithms

For developers seeking to understand the algorithms at a deeper level or implement a custom version, standalone codebases are invaluable.

- **Ullmann's Algorithm:**
    - **betterenvi/Ullman-Isomorphism (Python):** A clear Python implementation that is useful for educational purposes and for comparing against a custom implementation.[54]
    - **CDL Library (C++):** This computational chemistry library contains a generic C++ implementation of Ullmann's algorithm designed for use with BGL-like graph structures.[19]
- **VF2 Algorithm:**
    - **kpetridis24/vf2-pp (C++):** An implementation of the advanced VF2++ algorithm, providing a reference for a state-of-the-art, non-recursive solver.[55]
    - **OwenTrokeBillard/vf2 (Rust):** A modern implementation in Rust that clearly separates the logic for graph, subgraph, and induced subgraph isomorphism, serving as an excellent reference.[56]
- **Maximum Common Subgraph:**
    - **stefanoquer/Maximum-Common-Sugraph (C/C++):** A repository containing several versions of an MCS solver based on the McSplit algorithm, including sequential, multi-threaded, and GPU implementations. This is a powerful resource for the exact extension problem.[57]
    - **dilkas/maximum-common-subgraph (Python/ML):** This project focuses on algorithm selection for MCS, providing implementations of several core algorithms (McSplit, clique-based) and is a valuable source for the exact extension

problem.[58]

---

## Conclusion: Synthesizing a Solution Pathway

This report has dissected a complex graph problem into a series of well-defined algorithmic tasks. The central finding is that the user's "minimal extension" problem is an alternative formulation of the Minimum Common Supergraph problem, which is most effectively solved by first finding the Maximum Common Subgraph. Based on this, two complete solution pathways have been outlined: an exact method with exponential complexity, and an approximate method with polynomial complexity.
For practical implementation, the following strategy is recommended:

1. **Prototype in Python with NetworkX:** Begin by using the networkx.isomorphism.GraphMatcher class to quickly implement the initial subgraph isomorphism check. This will validate the first part of the exact algorithm's logic. For the extension component, since NetworkX lacks a direct MCES function, implement a simple greedy heuristic for the MCS approximation (as described in Part II). This will allow for the rapid development and testing of the complete logic for the polynomial-time version.
2. **Optimize in C++ with BGL or a Standalone Library:** For the high-performance, exact version, transition to C++. Use the boost::vf2_subgraph_mono function from the Boost Graph Library for the robust and efficient initial isomorphism check. For the minimal extension, the most challenging step is the exact MCS computation. While implementing the modular product and maximum clique reduction is possible, it is a complex task. A more pragmatic approach would be to integrate a high-quality, standalone MCS library from the provided GitHub resources, such as the McSplit implementation.[57]

While the underlying problems are computationally hard, the path to a solution is well-trodden. By leveraging the duality between minimal extension and maximum commonality, and by utilizing the powerful algorithms and libraries detailed in this report, the challenge becomes a tractable and well-defined engineering task.

### Cytowane prace

1. Improvements to Ullmann's Algorithm for the Subgraph Isomorphism Problem, otwierano: października 28, 2025, https://www.researchgate.net/publication/282393761_Improvements_to_Ullmann's_Algorithm_for_the_Subgraph_Isomorphism_Problem
2. Announcing vf2: A subgraph isomorphism algorithm in Rust #1 - GitHub, otwierano: października 28, 2025, https://github.com/OwenTrokeBillard/vf2/discussions/1
3. Structural Equivalence in Subgraph Matching - UCLA Mathematics, otwierano: października 28, 2025, https://ww3.math.ucla.edu/wp-content/uploads/2023/02/Cam23-07.pdf

4. VF2 Algorithm Overview and Implementation Examples - Deus Ex Machina, otwierano: października 28, 2025, https://deus-ex-machina-ism.com/?p=77202&lang=en
5. Subgraph isomorphism problem - Wikipedia, otwierano: października 28, 2025, https://en.wikipedia.org/wiki/Subgraph_isomorphism_problem
6. Proof that Subgraph Isomorphism problem is NP-Complete - GeeksforGeeks, otwierano: października 28, 2025, https://www.geeksforgeeks.org/dsa/proof-that-subgraph-isomorphism-problem-is-np-complete/
7. Subgraph isomorphism reduction from the Clique problem - Computer Science Stack Exchange, otwierano: października 28, 2025, https://cs.stackexchange.com/questions/64509/subgraph-isomorphism-reduction-from-the-clique-problem
8. Ullman's Subgraph Isomorphism Algorithm, otwierano: października 28, 2025, https://adriann.github.io/Ullman%20subgraph%20isomorphism.html
9. An Algorithm Using Length-R Paths to Approximate Subgraph ..., otwierano: października 28, 2025, https://digitalcommons.calpoly.edu/cgi/viewcontent.cgi?article=1000&context=eeng_fac
10. Approximate Graph Isomorphism? - informatik.hu-berlin.de, otwierano: października 28, 2025, https://www.informatik.hu-berlin.de/de/forschung/gebiete/algorithmenII/Publikationen/Papers/approx-gi.pdf
11. On the Minimum Common Supergraph of Two Graphs | Request PDF, otwierano: października 28, 2025, https://www.researchgate.net/publication/220261444_On_the_Minimum_Common_Supergraph_of_Two_Graphs
12. Implement maximum common subgraph · Issue #133 - GitHub, otwierano: października 28, 2025, https://github.com/gap-packages/Digraphs/issues/133
13. Survey of Graph Matching Algorithms - Vincent A. Cicirello, otwierano: października 28, 2025, https://www.cicirello.org/publications/survey-1999.pdf
14. 1 Subgraph Isomorphism - Stanford CS Theory, otwierano: października 28, 2025, http://theory.stanford.edu/~virgi/cs267/lecture1.pdf
15. Induced subgraph isomorphism problem - Wikipedia, otwierano: października 28, 2025, https://en.wikipedia.org/wiki/Induced_subgraph_isomorphism_problem
16. Subgraph isomorphism problem - Math Stack Exchange, otwierano: października 28, 2025, https://math.stackexchange.com/questions/1304064/subgraph-isomorphism-problem
17. Subgraph isomorphism in planar graphs - Computer Science Stack Exchange, otwierano: października 28, 2025, https://cs.stackexchange.com/questions/38482/subgraph-isomorphism-in-planar-graphs
18. [PDF] An Algorithm for Subgraph Isomorphism | Semantic Scholar, otwierano: października 28, 2025,

https://www.semanticscholar.org/paper/An-Algorithm-for-Subgraph-Isomorphism-Ullmann/065066a94860279587ecc7c7caaa65303008940f
19. Ullmann Algorithm for subgraph isomorphism - Chemical Descriptors Library, otwierano: października 28, 2025, https://cdelib.sourceforge.net/doc/ullmann.html
20. docs.chemaxon.com, otwierano: października 28, 2025, https://docs.chemaxon.com/display/docs/background_graph-matching.md#:~:text=Ullmann's%20algorithm%20builds%20a%20mapping,as%20the%20matrix%20is%20changed.
21. Subgraph Isomorphism - CSE - IIT Kanpur, otwierano: października 28, 2025, https://www.cse.iitk.ac.in/users/dsrkg/cs210old/cs245/html/seminar/isomorph.pdf
22. An Algorithm for Subgraph Isomorphism - SciSpace, otwierano: października 28, 2025, https://scispace.com/pdf/an-algorithm-for-subgraph-isomorphism-1t0qre6b4v.pdf
23. (PDF) Search Strategies for Subgraph Isomorphism Algorithms, otwierano: października 28, 2025, https://www.researchgate.net/publication/259801352_Search_Strategies_for_Subgraph_Isomorphism_Algorithms
24. MiviaLab/vf2lib: VF2 - Subgraph Isomorphism - GitHub, otwierano: października 28, 2025, https://github.com/MiviaLab/vf2lib
25. (PDF) A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs - ResearchGate, otwierano: października 28, 2025, https://www.researchgate.net/publication/3193784_A_SubGraph_Isomorphism_Algorithm_for_Matching_Large_Graphs
26. en.wikipedia.org, otwierano: października 28, 2025, https://en.wikipedia.org/wiki/Graph_isomorphism#:~:text=The%20vf2%20algorithm%20is%20a,graphs%20with%20thousands%20of%20nodes.
27. VF2++ — An Improved Subgraph Isomorphism Algorithm - EGRES, otwierano: października 28, 2025, https://egres.elte.hu/tr/egres-18-03.pdf
28. PERFORMANCE OF GENERAL GRAPH ISOMORPHISM ALGORITHMS - University of Houston, otwierano: października 28, 2025, https://uh.edu/nsm/_docs/cosc/technical-reports/2010/09_07.pdf
29. VF2 Algorithm — NetworkX 3.5 documentation, otwierano: października 28, 2025, https://networkx.org/documentation/stable/reference/algorithms/isomorphism.vf2.html
30. VF2 Algorithm — NetworkX 1.11 documentation, otwierano: października 28, 2025, https://networkx.org/documentation/networkx-1.11/reference/algorithms.isomorphism.vf2.html
31. Exploiting Vertex Relationships in Speeding up Subgraph Isomorphism over Large Graphs - VLDB Endowment, otwierano: października 28, 2025, http://www.vldb.org/pvldb/vol8/p617-ren.pdf
32. VF2++ - An improved subgraph isomorphism algorithm | Semantic Scholar, otwierano: października 28, 2025, https://www.semanticscholar.org/paper/VF2%2B%2B-An-improved-subgraph-isomorphism-algorithm-J%C3%BCttner-Madarasi/714b0795bb1b4c31b86eab917bc

66d99be6ede90
33. lkawka/mcs: Finding maximum common subgraph and minimum common supergraph of two graphs - GitHub, otwierano: października 28, 2025, https://github.com/lkawka/mcs
34. A graph distance metric combining maximum common subgraph and minimum common supergraph | Request PDF - ResearchGate, otwierano: października 28, 2025, https://www.researchgate.net/publication/223238850_A_graph_distance_metric_combining_maximum_common_subgraph_and_minimum_common_supergraph
35. The Maximum Common Subgraph Problem: A Parallel and Multi-Engine Approach - MDPI, otwierano: października 28, 2025, https://www.mdpi.com/2079-3197/8/2/48
36. Maximum Common Subgraph Isomorphism Algorithms - White Rose Research Online, otwierano: października 28, 2025, https://eprints.whiterose.ac.uk/id/eprint/102232/3/MCS_review_final.pdf
37. Greedy algorithm - Wikipedia, otwierano: października 28, 2025, https://en.wikipedia.org/wiki/Greedy_algorithm
38. On the Approximability of the Maximum Common Subgraph Problem. - ResearchGate, otwierano: października 28, 2025, https://www.researchgate.net/publication/220995032_On_the_Approximability_of_the_Maximum_Common_Subgraph_Problem
39. (PDF) An Approximate Maximum Common Subgraph Algorithm for ..., otwierano: października 28, 2025, https://www.researchgate.net/publication/220880221_An_Approximate_Maximum_Common_Subgraph_Algorithm_for_Large_Digital_Circuits
40. max_clique — NetworkX 3.5 documentation, otwierano: października 28, 2025, https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.approximation.clique.max_clique.html
41. Improved Dynamics for the Maximum Common Subgraph Problem - arXiv, otwierano: października 28, 2025, https://arxiv.org/html/2403.08703v1
42. [1802.08509] Graph Similarity and Approximate Isomorphism - arXiv, otwierano: października 28, 2025, https://arxiv.org/abs/1802.08509
43. Graph (abstract data type) - Wikipedia, otwierano: października 28, 2025, https://en.wikipedia.org/wiki/Graph_(abstract_data_type)
44. What Are the Different Types of Graph Algorithms & When to Use Them? - Neo4j, otwierano: października 28, 2025, https://neo4j.com/blog/graph-data-science/graph-algorithms/
45. Boost Graph Library: VF2 (Sub)Graph Isomorphism, otwierano: października 28, 2025, https://www.boost.org/doc/libs/1_83_0/libs/graph/doc/vf2_sub_graph_iso.html
46. libs/graph/example/vf2_sub_graph_iso_example.cpp - 1.80.0 - Boost, otwierano: października 28, 2025, https://original.boost.org/doc/libs/1_80_0/libs/graph/example/vf2_sub_graph_iso_example.cpp
47. Chapter 17. Graph isomorphism - igraph Reference Manual, otwierano:

października 28, 2025, https://igraph.org/c/html/0.10.16/igraph-Isomorphism.html
48. Chapter 17. Graph Isomorphism - igraph Reference Manual, otwierano: października 28, 2025, https://igraph.org/c/html/0.9.0/igraph-Isomorphism.html
49. Chapter 21. Graph isomorphism - igraph Reference Manual, otwierano: października 28, 2025, https://igraph.org/c/html/main/igraph-Isomorphism.html
50. Isomorphism — NetworkX 3.5 documentation, otwierano: października 28, 2025, https://networkx.org/documentation/stable/reference/algorithms/isomorphism.html
51. ISMAGS Algorithm — NetworkX 3.5 documentation, otwierano: października 28, 2025, https://networkx.org/documentation/stable/reference/algorithms/isomorphism.ismags.html
52. ISMAGS.largest_common_subgraph — NetworkX 3.5 documentation, otwierano: października 28, 2025, https://networkx.org/documentation/stable/reference/algorithms/generated/generated/networkx.algorithms.isomorphism.ISMAGS.largest_common_subgraph.html
53. networkx.algorithms.isomorphism.ISMAGS.largest_common_subgraph, otwierano: października 28, 2025, https://networkx.org/documentation/networkx-2.4/reference/algorithms/generated/networkx.algorithms.isomorphism.ISMAGS.largest_common_subgraph.html
54. Ullman Algorithm - An Algorithm for Subgraph Isomorphism - GitHub, otwierano: października 28, 2025, https://github.com/betterenvi/Ullman-Isomorphism
55. kpetridis24/vf2-pp: A state of the art algorithm for the Graph Isomorphism problem, adjusted for Sub-Graph and Induced Sub-Graph Isomorphism and extended to directed and multigraph settings. - GitHub, otwierano: października 28, 2025, https://github.com/kpetridis24/vf2-pp
56. OwenTrokeBillard/vf2: VF2 subgraph isomorphism algorithm in Rust. - GitHub, otwierano: października 28, 2025, https://github.com/OwenTrokeBillard/vf2
57. stefanoquer/Maximum-Common-Sugraph: Graph Isomorphism - GitHub, otwierano: października 28, 2025, https://github.com/stefanoquer/Maximum-Common-Sugraph
58. Algorithm Selection for Maximum Common Subgraph - GitHub, otwierano: października 28, 2025, https://github.com/dilkas/maximum-common-subgraph